Interactive Runtime Verification: Formal Models, Algorithms, and Implementation

Raphaël Jakse, Yliès Falcone, Jean-François Méhaut

Interactive Runtime Verification: Formal Models, Algorithms, and Implementation

RAPHAËL JAKSE, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France
YLIÈS FALCONE, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France
JEAN-FRANÇOIS MÉHAUT, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France

Interactive runtime verification (i-RV) combines runtime verification and interactive debugging. 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 its internal state interactively. 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 approaches using runtime verification.

In this paper, we provide a formal model for interactively runtime verified programs. For this, we model how of a program executes under a debugger composed with a monitor (for verdict emission) and a scenario (for steering the debugging session). We provide guarantees on the soundness of the verdicts issued by the monitor by exhibiting a weak simulation (relation) between the initial program and the interactively runtime verified program. Moreover, we provide an algorithmic view of this model suitable for producing implementations. We present Verde, an implementation based on GDB to interactively runtime verify C programs. We report on a set of experiments using Verde assessing the usefulness of Interactive Runtime Verification and the performance of our implementation. Our results show that, even though debugger-based instrumentation incurs non-trivial performance costs, i-RV is applicable performance-wise in a variety of cases and helps to study bugs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: runtime verification, property, monitoring, debugging

1 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. In this paper, we lay down the foundations for *interactive runtime verification*, an approach addressing the aforementioned challenges by combining two verification approaches, namely interactive debugging and runtime verification. We briefly recall the features of interactive debugging and runtime verification as well as their shortcomings.

Interactive Debugging. Interactive Debugging aims at studying and understanding a bug [34]. Fixing bugs is usually performed by observing a bad behavior and starting a debugging session to find the cause. 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 misbehavior. A debugging session generally consists in repeating the following steps: executing the program in a debugger, setting breakpoints on statements or function calls executed before the suspected cause of the bug. These steps aim to find the point in the execution where it starts being erratic and inspecting the internal state (call stack, values of variables) and to determine the cause of the problem. The execution is seen at a low

Authors' addresses: Raphaël Jakse, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France, Raphael.Jakse@univ-grenoble-alpes.fr; Yliès Falcone, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France, Ylies.Falcone@univ-grenoble-alpes.fr; Jean-François Méhaut, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France, Jean-Francois.Mehaut@univ-grenoble-alpes.fr.

level (assembly code, sometimes mapped to the source code) while the software developer reasons at the source level and thus would ideally want the execution to be abstracted. The debugger links the binary code to the programming language. The program state 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.

Runtime Verification. Runtime verification (aka monitoring) [1, 20, 28, 38] aims at checking the correctness of an execution w.r.t. a specification. The execution is abstracted into a sequence of events of program-state updates. Runtime verification 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 check runtime properties. While runtime verification is a versatile and effective method in providing formal guarantees on the correctness of program executions, it still remains an approach used by experts in this technique. Runtime verification is not yet integrated in developer tools nor used by developers.

Challenges and contributions. We introduce Interactive Runtime Verification (i-RV). i-RV aims at easing bug study by allowing both *bug discovery* and *bug understanding* at the same time. To fulfill this goal, i-RV combines interactive debugging and runtime verification by augmenting debuggers with runtime verification approaches, making i-RV a practical debugging method backed with a strong formal background. Gathering interactive debugging and runtime verification is challenging for several reasons. To the best of our knowledge, interactive debugging has not been formalized in previous work. In this paper, we therefore provide an abstraction of the execution of a program being debugged that is compatible with formalisms used in runtime verification. Moreover, instrumentation techniques traditionally used in runtime verification, like aspects or dynamic binary instrumentation do not allow controlling the execution and the set of monitored events is often static. Therefore, an important aspect of interactive runtime verification is the way the richness and expressiveness of the instrumentation as well as the control provided by the debugger are leveraged. Usage of verdicts issued by the monitor for guiding the interactive debugging session also has to be specified.

The key idea of interactive runtime verification is to use runtime verification as a guide to interactive debugging. The program is run in an interactive debugger, allowing the developer to use traditional approaches to study the internal state of the program. The debugger is also used as a means of instrumentation to produce the execution trace evaluated using runtime verification. We introduce the notion of scenarios in i-RV. Scenarios are a way to guide and automate the interactive debugging session by reacting to verdicts produced by the monitor and modify the execution or the control flow of the program. Scenarios are a means to separate the concerns of evaluating and controlling the execution of the program. Scenarios make use of checkpoints that allow saving and restoring the program state. They are a powerful way to explore the behavior of programs by trying different execution paths.

We aim to provide the foundations of a formal verification approach combining runtime verification and interactive debugging. We describe i-RV at several abstraction levels. First, we formally describe our execution model using natural

operational semantics. This model provides a solid and precise theoretical framework. In defining the model, we assume that programs are sequential, deterministic, and do not communicate with the outside. The model does not account for timing issues. We detail these assumptions in Sec. 5.1.

This framework provides a basis for reasoning and to ensure the correctness of our approach. Second, we give an algorithmic description using pseudo-code based on this operational view. This algorithmic view aims at helping build an actual implementation of i-RV. Developers using the approach are not required to have a full understanding of these descriptions. Finally, we provide and present a *full-featured implementation* 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 asserts the usefulness of i-RV and its applicability in terms of performances. Verde can be retrieved at [24].

A this point, we mention that this paper is an extended version of a paper that appeared in the proceedings of the 28th IEEE International Symposium on Software Reliability Engineering (ISSRE 2017) [23]. This paper significantly extends the original conference version with the following additional contributions:

- a formal operational model of the behavior of interactive runtime verification expressed as a formal composition of the behavior of an interactive debugger, a monitor, and a scenario with the program under scrutiny (Sec. 6);
- a theorem (with its proof) stating that the debugged program is observationally equivalent to the original program and that verdicts from the monitor are guaranteed to correspond to an execution of the initial program (Sec. 7);
- algorithms implementing the operational semantics model of the interactively runtime verified model (Sec. 8);
- we provide additional and updated experiments to evaluate and validate the effectiveness of interactive runtime verification (Sec. 10).

Paper organization. In Sec. 2, we overview interactive runtime verification. In Sec. 3, we present existing approaches for finding and studying bugs and compare them to our work. In Sec. 4, we define some notations used in this paper. In Sec. 5, we define some notions related to interactive runtime verification. In Sec. 6, we give an operational view of our model. In Sec. 7, we give guarantees on our model. Proofs of these guarantees are given in Appendix A In Sec. 8, we present an algorithmic view of our model. In Sec. 9, we present our proof-of-concept implementation of this approach, Verde. In Sec. 10, we evaluate our approach. In Sec. 11, we conclude and outline avenues for future work.

2 OVERVIEW OF INTERACTIVE RUNTIME VERIFICATION

In i-RV (Fig. 1), the program to be interactively verified at runtime is run alongside a debugger, a monitor and a scenario. The developer can drive the debugger as usual, like in an interactive debugging session. The debugger also provides tools to control and instrument the program execution, mainly breakpoints and watchpoints. The monitor evaluates a developer-provided property against the program execution trace and produce a sequence of verdicts. See Fig. 2 for an example of such a property. To evaluate the property, the monitor requests events (function calls and memory accesses) to the debugger. The debugger instruments the program by setting breakpoints and watchpoints at relevant locations and notifies the monitor whenever such an event happens, effectively producing a trace of the execution that is relevant for the property being evaluated. When an event stops influencing the evaluation of the property, the corresponding instrumentation (breakpoints, watchpoints) becomes useless and is therefore removed: the instrumentation is *dynamic*.

The developer-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

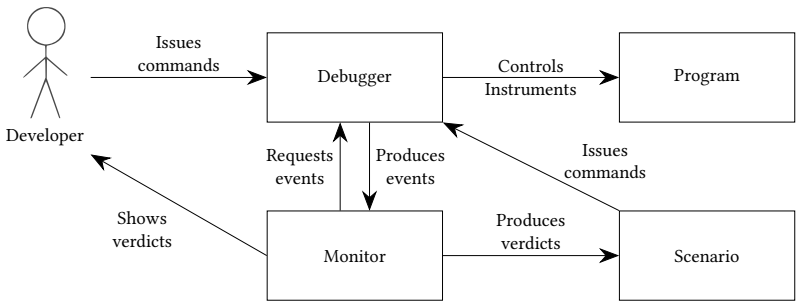

Fig. 1. Overview of Interactive Runtime Verification. The developer issues commands to the debugger, the debugger controls and instruments the program, the monitor requests program events to the debugger and issues verdicts to the scenario and the scenario applies developer-defined reactions.

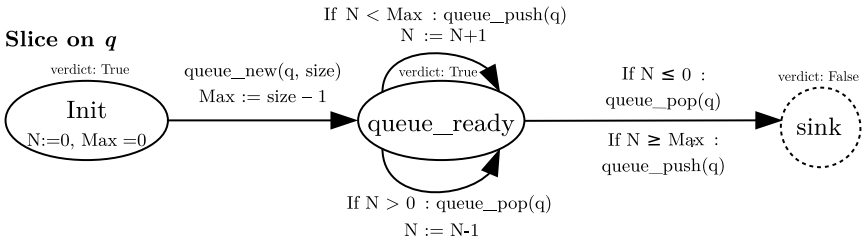

Fig. 2. Property for the absence of overflow for each queue q in a program. When a queue is created, the automaton is in state queue_ready for this queue. Pushes and pops are tracked. When a push (resp. pop) causes an overflow (resp. underflow) of the queue, the property is in state sink.

the queue overflows), the execution is suspended to let the developer inspect and debug the program in the usual way, interactively; save the current program state (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”, “state X has been left”, “an accepting state has been entered”, “a non-accepting state has been left”.

Comparison with interactive debugging. Fig. 3 depicts a comparison between a traditional interactive debugging session and an interactive runtime verification session. In using any of the approaches, we have a program and a specification. The specification and the program are related in that the program is either written from the specification or the specification describe (part of) the expected behavior of the program. During the execution of the program, the developer observes a fault; this can be a crash, an incorrect result, a failing test case, etc. Using this observation, the call stack trace, and the specifications, the developer makes hypotheses on the error that led to the fault. Making hypothesis consists in identifying events (function call, variable accesses, etc.) that are involved in the error. Events can carry data from the program, e.g., the effective parameter of a function call. The developer usually believes that something went wrong with these events during the execution (absence/presence of an (un)expected event, wrong ordering between events, etc.).

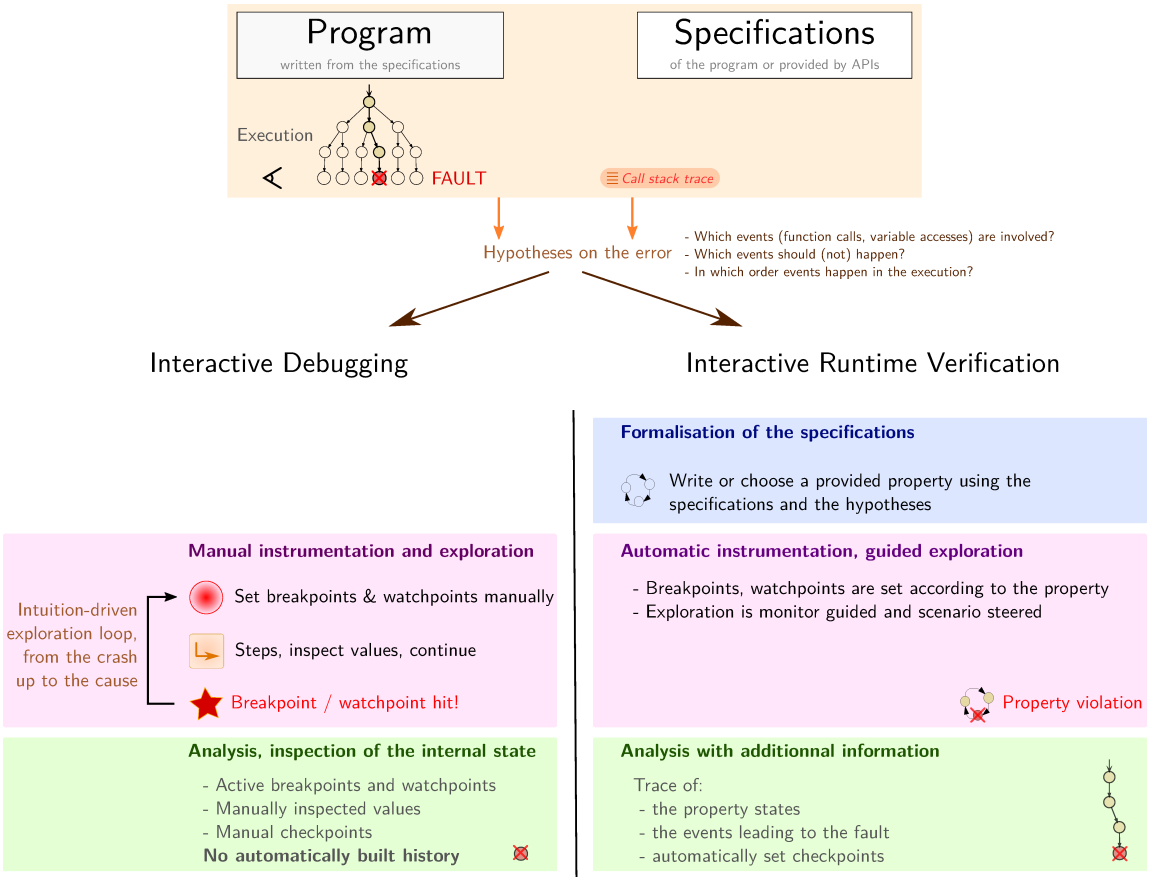

Fig. 3. Interactive Debugging versus Interactive Runtime Verification

During an **interactive debugging** session, the developer explores the execution *manually*, from the observation of the fault up to the cause, by (i) using the call stack trace (ii) manually setting breakpoints and watchpoints (iii) inspecting values and stepping through the program. This process is iterative. While studying the program behavior, the developer has some information: the currently active breakpoints and watchpoints, the values of manually inspected variables, and the set of manually set checkpoints.

During an **interactive runtime verification** session, properties are chosen among those provided with the APIs used by the program or written from the specifications. Each property formally expresses an expected behavior of the program, which is selected from the specification. A property formally defines the events and their expected ordering during an execution and associates each sequence of events with a verdict indicating whether the behavior is desirable. The monitors of the properties are typically finite-state machines that serve as decision procedures (for the properties). The program is *automatically* instrumented by adding the breakpoints and watchpoints so that the monitor can observe the events involved in the properties. Hence, the execution is monitored at runtime, relieving the developer from the corresponding manual exploration in classical interactive debugging. When a property is violated, the developer analyzes the bug and is provided with a trace containing several pieces of information that are automatically produced:

events relevant to the property that happened during the execution, the states through which the monitor went, as well as the taken checkpoints. These checkpoints can be seen as bookmarks in the execution. Contrarily to interactive runtime verification, during a classical interactive debugging session, no information is automatically produced: any piece of information comes from manual inspection by the developer.

```c
#define size 16

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

typedef struct {
	int pos_c;
	int pos_v;
	char consonants[size / 2];
	char vowels[size / 2];
} double_queue_t;

double_queue_t* queue_new() {
	double_queue_t* queue = malloc(sizeof(double_queue_t));
	queue->pos_c = 0;
	queue->pos_v = 0;
	bzero(queue->consonants, size / 2);
	bzero(queue->vowels, size / 2);
	return queue;
}

void queue_destroy(double_queue_t* queue) {
	free(queue);
}

void queue_push(double_queue_t* queue, char c) {
	if (strchr("aeyuio", c))
		queue->vowels[queue->pos_v++] = c;
	else if (strchr("zrtpqsdfghjklmwxcvbn", c))
		queue->consonants[queue->pos_c++] = c;
}

void queue_push_str(double_queue_t* queue, char* s) {
	while (s[0] != '\0')
		queue_push(queue, (s++)[0]);
}

void queue_display_result(double_queue_t* queue) {
	printf("Consonants: ");
	for (int i = 0; i < queue->pos_v; i++)
		putchar(queue->consonants[i]);
	printf("\nVowels: ");
	for (int i = 0; i < queue->pos_v; i++)
		putchar(queue->vowels[i]);
	puts("");
}

int main() {
	double_queue_t* queue = queue_new();
	queue_push_str(queue, "oh, a nasty bug is here!");
	queue_display_result(queue);
	queue_destroy(queue);
	return 0;
}
```

Fig. 4. Faulty C program. Vowels and consonants are supposed to be split in two separate arrays. However, according to the program output (Consonants: hnstbgsh and Vowels: raayuiee), the split is not done correctly.

Illustrative example. We compare a traditional interactive debugging session with an interactive runtime verification session using a sample C program depicted in Fig. 4. This program purposely contains a fault. The program splits the vowels and consonants from a given string in a custom queue structure which consists of two arrays. When run, the program displays a consonant among the vowels.

In an interactive debugging session (reproduced in Fig. 29, p. 60), the developer tracks the bug by running a debugger and may study the bug as follows. First, to check that the problem does not come from the function that displays the queue, setting a breakpoint at line 53 allows the developer to print the value of q->vowels. The debugger displays "raayuiee". A consonant was indeed put in the array of the structure that is dedicated to vowels. To study how the vowels are added to the queue, the developer may restart the program and put a breakpoint on line 30. When the execution is suspended, the developer types command display c to show the character being added to the vowels. The breakpoint is reached 8 times and no consonant is added to the array vowels. The developer may want to know how the first cell of array vowels was filled with a consonant. For this purpose, setting a breakpoint after the declaration of the queue in function main allows the developer to set a watchpoint on q->vowels[0]. The watchpoint is reached a first time. A vowel ("o") is stored in this cell at line 32, which is an expected behavior. After continuing the execution, the watchpoint is reached a second time at line 32 and "o" is replaced by "r". This line is supposed to write in array

consonants. Command print q->pos_c - 1 displays 8. Array consonants is defined on line 11 with size 16/2 = 8. Value 8 is therefore out of bounds. Since array vowels is defined right after array consonants, this overflow leads to a rewrite of the first cell of array vowels.

In an interactive runtime verification session, the developer thinks about which properties should be verified in the code and may use the one depicted in Fig. 2 (the queue should not overflow). When the program is interactively runtime verified with this property[1], a breakpoint is automatically put at the beginning of function queue_new and another at the beginning of function queue_push. The first breakpoint is reached once and the second one 17 times, without interrupting the execution, saving the developer from having to carry out a manual inspection. The non-accepting state of the property is reached and the execution is suspended. In the debugger prompt, the developer can display the stack trace and observe that the fault happened at line 37 in queue_push_str, called in function main at line 52.

In both cases, the developer deduces that checks should be added in the program code to avoid overflows. In the latter case, the typical exploration phase of interactive debugging is avoided: breakpoints are set and handled automatically and the property state gives an higher-level explanation of the bug (too many letters were pushed in the queue, causing the cell to be rewritten later), and the bug is located earlier in the execution, before the cell is rewritten. This example features a typical array overflow, a common fault in C programs. Such overflows often have security implications. Unfortunately, they are not always easily noticed because array accesses are not automatically checked at runtime when using C. Here, interactive runtime verification helps the detection of this fault, caused by the misuse of a data structure, by checking a property over this data structure.

3 RELATED WORK

i-RV is related to several families of approaches for finding and fixing bugs. In this section, we overview some of these approaches, their drawbacks and benefits, and how they are suitable for discovering different sorts of bugs in different situations. Their relevance is also related to a phase of the program life cycle. We note already that none of them combines bug discovery and understanding, as i-RV does.

3.1 Testing

Manual testing. Most obvious bugs can easily be spotted this way during the development of the software. Modifications to the code are manually tested, possibly by a team responsible for testing the software [22]. Bugs are also spotted by final users of the software, which may be undesirable. Manual testing can be tedious. i-RV aims at reducing manual and tedious human intervention by guiding and partly automating the exploration of misbehaviors.

Automatic testing. Among the numerous automated testing approaches, unit testing is one of the most popular and adopted ones. Unit tests aim to limit regressions and to check the correctness of the code for a restricted set of inputs [21]. Many unit testing frameworks exist, including for instance JUnit and CppUnit. Some research efforts have been carried out on the automatic generation of unit tests. For instance, [8] generates test oracles from formal specifications of the expected behavior of a Java method or class. In that, unit testing uses some form of verification at runtime of the program execution. i-RV is complementary to automatic testing as it is more focused on getting insight on the program behavior. Thanks to the debugger, i-RV offers some interactivity and more accurate information in case of a breakage (while automatic testing generally issues verdicts).

[1] With Verde: verde –prop queue.prop ./faulty, where the contents of file queue.prop is given in Fig. 25 (p. 46). A session is reproduced in Fig. 30 (p. 61). The program is compiled with gcc -g3.

3.2 Heavyweight Verification

With heavyweight verification approaches (e.g., static analysis [11]), 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, these approaches can be slow, limited to certain classes of bugs or safety properties and can produce false positives because of the required abstractions on the verified software. Model checking [16] is an automatic verification approach for finite-state reactive systems. Model checking consists in checking that a model of the system verifies temporal properties [9].

Heavyweight verification approaches are usually limited in the expressiveness of the properties they can check. Heavyweight verification approaches are complementary to dynamic verification approaches. Acknowledging the incompleteness of dynamic verification approaches allows checking more complex behavioral properties and thus i-RV can be used to cover the cases and situations left uncovered by heavyweight verification approaches. In particular, any verdict produced by the monitors corresponds to the associated evaluation of the property and thus any bug leading to the violation of a property will be spotted.

3.3 Interactive and Reverse Debugging

Tools used in interactive debugging [34, 35] are mainly debuggers such as GDB, LLDB and the Visual Studio debugger. GDB is a cross-platform free debugger from the Free Software Foundation. LLDB is the cross-platform free debugger of the LLVM project, started by the University of Illinois, now supported by various firms such as Apple and Google. The Visual Studio debugger is Microsoft's debugger. Reverse debugging [17, 19, 37] is a complementary debugging method. 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. Reverse debugging is still akin to a traditional, manually driven interactive debugging session. 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. We provide i-RV with an execution model sufficiently detailed for the purpose of ensuring guarantees about its correctness (see Sec. 7). Work has been done to focus on the modelizsation of program execution, compilers, and debuggers, to ensure correct and intuitive behavior of the debugger [13], more suitable to implement and reason about interactive debuggers.

3.4 Runtime Verification

Runtime verification consists in checking properties at runtime. Checks are performed on the sequence of events produced during the execution. Producing events requires instrumentation. Different instrumentation techniques exist. In this section, we give some of the most important ones. Compared to runtime verification, i-RV adds interactivity and access to the internal state of the program, and provides a way to use verdicts issued by the monitor. Runtime verification is a building block of i-RV.

Compile-Time Instrumentation. RiTHM [32] is an implementation of a time-triggered monitor, i.e., a monitor ensuring predictable and evenly distributed monitoring overhead by handling monitoring at predictable moments. Instrumentation is added to the code of the program to monitor. i-RV does not modify the code of the program nor does it requires recompiling the program.

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 a virtual machine-like environment. Valgrind [33] is a tool that leverages DBI and can interface with GDB. It provides a way to detect memory-related defects. Dr. Memory [3] is another similar tool based on DynamoRIO [4]. DynamoRIO and Intel Pin [29] are both DBI frameworks that allow to write dynamic instrumentation-based analysis tools. 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 a function like free. DBI also does not provide a straightforward way to suspend the execution and add interactivity.

Instrumentation Based on the VM of the Language. For some languages like Java, the Virtual Machine provides introspection and features like aspects [26, 27] used to capture events. The Jassda framework [2], which uses CSP-like specifications, LARVA [10] and JavaMOP [6] are monitoring tools for programming languages based on a virtual machine (mainly Java). This is different from our model which rather depends on the features of the debugger. JavaMOP [25] 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 [7]. In our work, events are dispatched in slices in a similar way. We do not implement all the concepts defined by [7] 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 [18]. This differs 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.

Debugger-Based Instrumentation. Morphine [14], Opium [15] and Coca 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, trace analyzers rely on the debugger to generate events. However, they do not provide interactivity and do not permit affecting the execution.

Frama-C [12]. 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.

With these runtime verification approaches, the conceptual frameworks as well as the implementations do not provide much information to the developer in case of error. In best cases, a runtime verification framework indicates the line in the source code at which a violation occurred, as well as the abstract sequence of events that lead to the error. With i-RV, the provided information is much more detailed: the whole internal state of the program can be inspected thanks to the debugger.

4 NOTATIONS AND DEFINITIONS

In this section, we define some notations and concepts used throughout the paper.

4.1 Sets and Functions

Definition and image domains. For two sets E and F, a function from E to F is denoted by $f : E \to F$. We denote the set of functions from E to F by $[E \to F]$. Let $f : E \to F$ be a function. We denote the domain of function f by

$\mathrm{Dom}(f)$, the subset of E on which the function is defined. We denote the image of function f by $\mathrm{Im}(f)$, defined by $\mathrm{Im}(f) = \{f(x) \mid \exists x \in E\}$. Let X be a set and e an expression. We denote by $\{x \mapsto e \mid x \in X\}$ the function f such that $\mathrm{Dom}(f) = X$ and $f(x) = e$.

Function substitution. Function substitution is used to lighten notation in proofs, especially when proving equalities. Let f and g be two functions. We denote by $f \dagger g$ the function h such that: $\forall x \in \mathrm{Dom}(g),\ h(x) = g(x)$ and $\forall x \notin \mathrm{Dom}(g),\ h(x) = f(x)$. Let $f : E \longrightarrow F$, for $x_1 \in X$ and $v \in F$, function $f[x_1 \mapsto v]$ denotes function f' such that $f'(x) = f(x)$ for any $x \neq x_1$, and $f'(x_1) = v$.

Powerset. The powerset of a set E is denoted by $\mathcal{P}(E)$. Let f be a function. Function $f_{\setminus E}$ is such that $\forall e \in \mathrm{Dom}(f) \setminus E$, $f_{\setminus E}(e) = f(e)$ and $\mathrm{Dom}(f_{\setminus E}) = \mathrm{Dom}(f) \setminus E$ (this function is undefined on elements in E).

Sequences. Moreover, ϵ is the empty sequence. 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 by $s \cdot s'$. We denote the number of elements in a finite sequence or a set C by $|C|$. Sequences are used to represent lists of breakpoints or watchpoints in the debugger.

Named tuples. Throughout the paper, we use named tuples to describe configurations of the components of our architecture, for their readability and explicitness over simple tuples.

Definition 4.1 (Named Tuple). A named n-tuple $r = (t, f_{\mathrm{ind}})$ is pair composed of a n-tuple $t = (t_1, \ldots, t_n) \in E_1 \times \cdots \times E_n$ (for any sets $E_1, \cdots, E_n$) and a bijection $f_{\mathrm{ind}} : \{1, \ldots, n\} \rightarrow \mathrm{Name}$ that maps indexes to names.

The shorthand notation $(f_1 \mapsto v_1, \ldots, f_n \mapsto v_n) \in E_1 \times \cdots \times E_n$ denotes the named tuple $((v_1, \ldots, v_n), [1 \mapsto f_1, \ldots, n \mapsto f_n])$ such that $(v_1, \ldots, v_n) \in E_1 \times \cdots \times E_n$.

Moreover, we shall use the field notation: for $\mathit{field} \in \mathrm{Im}(f_{\mathrm{ind}})$, $r.\mathit{field}$ denotes $t_{f_{\mathrm{ind}}^{-1}(\mathit{field})}$, that is $r.\mathit{field}$ is the value in tuple t at index i such that $f_{\mathrm{ind}}(i) = \mathit{field}$.

Substitution for named tuples. We use substitution to describe the evolution of a configuration. Let $r = (t, f_{\mathrm{ind}})$ be a named n-tuple. We denote by $r' = r[f_1 \mapsto v_1, \ldots, f_k \mapsto v_k]$ the named tuple equal to r, except for fields $f_1, \ldots, f_k$, which are equal to values $v_1, \ldots v_k$, respectively. That is, r' is such that $\forall i \in \{1, \ldots, k\}, r'.f_i = v_i$ and $\forall i \notin \{1, \ldots, k\}, r'.f_i = r.f_i$.

Remark 1. In the remainder of the paper, given a bijection f_{ind}, we use a tuple t and the corresponding named tuple $r = (t, f_{\mathrm{ind}})$ interchangeably.

4.2 Notation and Notions Related to Labeled Transition Systems

Labeled transition systems. Later in the paper, we model the components involved in interactive runtime verification as Labeled Transition Systems (LTSs) and Input-Output Labeled Transition Systems (IOLTS).

Definition 4.2 (LTS). An LTS is a tuple $(Q, A, \rightarrow)$, where Q is the set of configurations, A is a set of actions and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation between configurations.

$(q, a, q') \in \rightarrow$ is denoted by $q \xrightarrow{a} q'$ and $q \rightarrow q'$ is a short for $\exists a \in A : q \xrightarrow{a} q'$. If $q, q' \in Q$ are two configurations such that $q \rightarrow q'$, the LTS is said to evolve from configuration q to configuration q'. Configuration q' (resp. q) is said to be a successor (resp. predecessor) of configuration q (resp. q').

We also use IOLTSs, which are LTSs with inputs and outputs.

Definition 4.3 (IOLTS). An IOLTS is a tuple $(Q, A, I, O, \rightarrow)$. Q is the set of configurations, A is the set of actions, I and O are the sets of input and output symbols respectively, $\rightarrow \subseteq Q \times A \times I \times Q \times O$ is the transition relation between configurations.

Whenever $(q, a, i, q', o) \in \rightarrow$, we note it $q \xrightarrow{a/i/o} q'$ and $q \xrightarrow{i/o} q'$ is a short for $\exists a \in A : q \xrightarrow{a/i/o} q'$.

If $q, q' \in Q$ are two configurations, $i \in I$ is an input symbol and $o \in O$ an output symbol such that $q \xrightarrow{i/o} q'$, the IOLTS is said to output symbol o and evolve from configuration q to configuration q' when the next input symbol is i. Vocabulary and notations on the transitions of LTSs are extended to IOLTSs in the natural way.

Semantic rules. We use semantic rules to define the transition relation of LTSs and IOLTSs. More precisely, the transition relation is the least set of transitions that satisfy semantic rules. Semantic rules are written using the following standard notation:

$$\text{R}\ \frac{\text{conditions on } q \text{ and } i}{q \xrightarrow{i\ /\ o} q'}$$

This notation reads as follows: for all $q, q' \in Q$, $i \in I$, $o \in O$, $q \xrightarrow{a/i/o} q'$ holds if the conditions on q and i hold, where a is r, the name of the rule, unless otherwise explicitly specified.

Transitions without an input (resp. output) symbol are permitted. In this case, i (resp. o) is written $-$.

The set of semantic rules that define a transition relation $\rightarrow$ is denoted by Rules($\rightarrow$).

Given states q and q' and an action a, $q \xrightarrow{a} q'$ means $(q, a, q') \in \rightarrow$ if $\rightarrow$ is the transition relation of an LTS, $\exists\, (i, o) \in I \times O : (q, a, i, q', o) \in \rightarrow$ if $\rightarrow$ is the transition relation of an IOLTS.

Given any two configurations q and q' and a regular expression E over A, $q \xrightarrow{E} q'$ means: there exists a finite sequence $(a_1, \ldots, a_k)$ of actions of A matched by E and intermediate configurations such that $q \xrightarrow{a_1} \ldots \xrightarrow{a_k} q'$.

Selection of configurations by fields. If q is a set of tuples, constraints to the application of a semantic rule can also be given using the selection notation:

$$\frac{\text{conditions on } q}{q\,\langle f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\rangle \xrightarrow{i\ /\ o} q'}$$

The application of the semantic rule is limited to configurations q for which fields $f_1, \ldots, f_n$ in q are equal to values $v_1, \ldots v_n$, respectively.

Similarly, for any named tuples q, q' and any relation $\rightarrow$, $q \rightarrow q'\langle f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\rangle$ is a short for $(q, q') \in \rightarrow$ and fields $f_1, \ldots, f_v$ in q' are equal to values $v_1, \ldots, v_n$, respectively.

Simulation relation. We use the simulation relation to express correctness properties on our models. Let $S_1 = (Q_1, q_0^1, \rightarrow_1, Obs \cup \overline{Obs})$ and $S_2 = (Q_2, q_0^2 \rightarrow_2, Obs \cup \overline{Obs})$ be two LTSs over a common set of actions $Obs \cup \overline{Obs}$ where Obs (resp. $\overline{Obs}$) is the set of observable (resp. unobservable) actions. We recall below the notion of weak simulation.

Definition 4.4 (Weak simulation [31]). LTS S_1 weakly simulates LTS S_2 if there exists a relation $R \subseteq Q_1 \times Q_2$ such that:

(1) $\forall (q_1, q_2) \in R, \forall \theta \in \overline{Obs}, \forall q_1' \in Q_1 : \left(q_1 \xrightarrow{\theta} q_1' \implies \exists q_2' \in Q_2 : (q_1', q_2') \in R \wedge q_2 \xrightarrow{\overline{Obs}^*} q_2' \right)$.

(2) $\forall (q_1, q_2) \in R, \forall \alpha \in Obs, \forall q_1' \in Q_1 : \left(q_1 \xrightarrow{\alpha} q_1' \implies \exists q_2' \in Q_2 : (q_1', q_2') \in R \wedge q_2 \xrightarrow{\overline{\text{Obs}}^* \cdot \alpha \cdot \overline{Obs}^*} q_2' \right)$.

Relation R is a weak simulation if for any related pair of states (q_1, q_2):

(1) for any unobservable action $\theta \in \overline{Obs}$, for any state q_1' that can be reached from q_1 with θ, one can find another state q_2' that is reached by a sequence of unobservable actions in $\overline{Obs}$; and
(2) for any observable action $a \in Obs$, for any state q_1' that can be reached from q_1 with a, one can find another state q_2' that is reached by a sequence of actions in $\overline{\text{Obs}}^* \cdot a \cdot \overline{Obs}^*$, i.e., a sequence composed of unobservable actions in $\overline{Obs}$, action a, and unobservable actions in $\overline{Obs}$.

5 NOTIONS USED IN INTERACTIVE RUNTIME VERIFICATION

In this section, we present fundamental notions of our approach, namely a formalization of programs, events, and some concepts related to debuggers.

5.1 Program

We do not aim to give a realistic model of programs. Rather, our model aims for simplicity and minimalism, suitable for expressing correctness properties on our approach. We first discuss the assumptions applying to our model. We then define the notions of values, addresses, memory, accesses, symbols, names, events and parameters used in our representation of a program. We then define the program itself.

Assumptions. We consider deterministic and sequential programs without side effects. In particular, the program does not communicate with the outside and does not read user input and is not subject to interrupts. We do not account for physical time as we aim to verify properties with logical (discrete) time. Though i-RV applies to programs that do communicate, these assumptions simplify the expression of properties of this model. Moreover, we do not consider mechanisms like Address Space Layout Randomization (ASLR), aiming to mitigate some kinds of attacks relying on knowing or predicting the location of specific data or functions. Though ASLR is widespread, we note that it is disabled by debuggers like GDB and LLDB by default to ease debugging. We do not model ASLR. We also do not consider self-modifying programs and program that execute their data. As a security feature, on many operating systems, programs run with write protection on their code and execution protection on their data by default. Just-in-Time compilers execute runtime generated code. Debugging these programs require specific runtime support[2] that we do not model. We also do not consider programs that read their code to detect that they are executing under a debugger.

Definition 5.1 (Value, address, memory). Value is the set of values that can be stored in variables of a program. Values are machine words, either data (values of variables) or program instructions. An address is an integer indexing a value in a memory. Since an address can be stored in a variable, it is also a value. Addr $\subseteq$ Value denotes the set of addresses in a program. A memory maps addresses to values. Mem $\stackrel{\text{def}}{=}$ [Addr $\rightarrow$ Value] is the set of memories.

Definition 5.2 (Access). Access $\stackrel{\text{def}}{=}$ Addr $\times$ {r, w} $\times$ Value $\times$ Value is the set of memory accesses. An element $(addr, mode, old, new) \in$ Access represents a read access at address $addr$ if $mode$ = r, or a write access to address $addr$ if $mode$ = w. old is the value being accessed, and new is the new value in case of a write access (and is not defined for a read access).

[2] https://www.llvm.org/docs/DebuggingJITedCode.html

Definition 5.3 (Symbol). A symbol is the name of a variable or a function. Symbol is the set of symbols used in a program. Symbols are linked to addresses in a program using a symbol table.

Definition 5.4 (Parameter). A grammar describing the set of valid parameters Param is given in Figure 5. A parameter in Param is: either $v \in$ Symbol (a variable or function name), $*p$ (the value pointed by p, with $p \in$ Param), $\&p$ (the address of variable p), arg i (the current value of parameter of index $i \in \mathbb{N}$), $\hat{}p$ (a parameter p in a deeper frame in the current call stack) or ret (the "return value" of the function call for function call events).

Example 5.5 (Parameter). We present four examples of parameters.

- arg 2 is a parameter referring to the second argument of the current function in the running program.
- $\hat{}$ arg 2 is a parameter referring to the second argument of the caller of the current function in the running program.
- counter is a parameter referring to the current value of variable counter in the running program.
- $*$ptr is a parameter referring to value stored at the address stored in variable ptr.

We model a program that executes instructions and stops when the end of the code is reached. This matches the behavior of programs in common operating systems. For the sake of generality, our abstraction of a program is platform-independent and language-independent. This abstraction assumes a program loaded in memory.

Definition 5.6 (Symbol table). A symbol table Sym is used to get the address of an object in the program memory at runtime from its description. Sym $\in$ SymbolTable $\stackrel{\text{def}}{=}$ Mem $\times$ pc $\times$ Param$\times$ $\longrightarrow$ Addr maps a memory, a program counter and a parameter to an address. This parameter may be the name of a variable or a function, or any other parameter. The symbol table is built at compile time and resolves symbols at runtime.

Definition 5.7 (Program). A *program* is a 5-tuple (Sym, m_p^0, start, runInstr, getAccesses) where:

- Sym $\in$ SymbolTable is the *symbol table*.
- $m_p^0 \in$ Mem is the *initial memory*,
- start $\in$ Addr is an address that points to the first instruction to run in the memory,
- runInstr : (Mem $\times$ Addr) $\longrightarrow$ (Mem $\times$ Addr) is a function representing and abstracting the instruction set of the processor on which the program runs. This function takes a memory and a program counter and returns a new (updated) memory and a new program counter, and
- getAccesses : (Mem $\times$ Addr) $\longrightarrow$ Accesses* is a function returning the sequence of accesses that will be made when running the next instruction.

Example 5.8 (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 5.9 (Configuration of the program). A configuration of the program is a 2-tuple $(\mathrm{m} \mapsto m, \mathrm{pc} \mapsto pc) \in \mathrm{Conf_p} \stackrel{\text{def}}{=}$ Mem $\times$ Addr, where:

- m is the memory of the program, represented as a sequence of memory cells containing either values of variables or executable instructions;

p	::=	v	(*a defined variable*)
	|	$*p$	(*the value pointed by p*)
	|	$\&p$	(*the address of variable p*)
	|	$\hat{}p$	(*p, in a deeper frame in the call stack*)
	|	arg $i, i \in \mathbb{N}$	(*the current value of parameter i*)
	|	ret	(*the return value for call events*)

Fig. 5. Grammar of valid parameter names

- pc (the program counter) is an address, that is, an index of a cell in the memory m that is the next instruction to execute.

Example 5.10 (Configuration of the program). For program P given in Example 5.8, 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, $m_p[\text{Sym}(m_p, pc, a)] = 0$ and $m_p[\text{Sym}(m_p, pc, b)] = 1$.

Definition 5.11 (Program checkpoint). A program checkpoint is a configuration of a program $(m, pc) \in \text{Conf}_\text{P}$ that is used as a snapshot of the program state, that can be restored later.

Example 5.12 (Program checkpoint). For the program given in Example 5.8, a checkpoint taken when the third instruction is about to be executed, is $(([a \mapsto 0, b \mapsto 1], pc_3))$, where pc_3 is the location of the third instruction in the memory.

5.2 Events

I-RV relies on capturing events from the program execution with the debugger. Events are generated during the execution of the program. Upon the reception of events, the monitor updates its state and generates verdicts. The scenario reacts to verdicts by executing actions.

In this section, we define symbolic events, used to describe properties. We then define runtime events, generated during the execution.

Definition 5.13 (Symbolic event). A symbolic event is a named tuple $e = (\text{type} \mapsto t, \text{name} \mapsto n, \text{params} \mapsto p) \in \text{EventTypes} \times \text{Symbol} \times \text{Param}^*$ such that:

- $t \in \text{EventTypes} = \{\text{BeforeCall}, \text{AfterCall}, \text{ValueRead}, \text{ValueWrite}, \text{ValueAccess}\}$ is the type of event: before a function call, after a function call, a value read, a value write or a value access (read or write).
- $n \in \text{Symbol}$ is the name of the event. For a function call (resp. variable access), the event name is the name of the considered function (resp. variable) found in the symbol table of the program,
- $p \in \text{Param}^*$ is a sequence of parameters.

Example 5.14 (Symbolic event). $(\text{BeforeCall}, \text{push}, (q, v))$ is an event triggered when function push is called. Parameters q and v are retrieved when producing the event.

Definition 5.15 (Runtime event). A runtime event is a pair $(e_f, v) \in \text{Event} \stackrel{\text{def}}{=} \text{SymbolicEvent} \times \text{Value}^*$ where e_f is a symbolic event and v a sequence of values.

Example 5.16 (Runtime event). $(\mathrm{BeforeCall}, \mathrm{push}, (0x5650653c4260, 42))$ is an event triggered when function push is called with these runtime parameters. Runtime parameters $0x5650653c4260$, 42 q are instances of symbolic parameters q and v.

Remark 2. In practice, BeforeCall and AfterCall events are captured using breakpoints and ValueWrite, ValueRead and ValueAccess are captured using watchpoints.

Remark 3. Current debuggers allow watching the value of an expression involving several variables by setting several watchpoints automatically. We do not consider this feature, as well as other kinds of events (e.g., system calls, signals) in our model for simplicity.

5.3 Instrumentation Provided by the Debugger and Event Handling

The debugger provides two mechanisms to control and instrument the execution of the program: breakpoints and watchpoints. These primitives can be used by the developer during an interactive debugging session, by the scenario to automate some action and as a means to generate events for the monitor. A breakpoint stops the execution at a given address $a \in \mathit{Address}$ and a watchpoint when a given address containing data of interest is accessed (read, written, or both).

5.3.1 *Breakpoint*

A breakpoint can be intuitively understood as a bookmark on an instruction of the program. When encountered, the program shall suspend its execution and inform the debugger. Software breakpoints are commonly implemented by replacing instructions in the program code on which the execution shall be suspended by a special instruction [36].

We denote such a special instruction by BREAK. When setting a breakpoint, the original instruction must be saved in the configuration of the debugger in order to be able to restore it when this instruction must be run.

A breakpoint may be set either by the developer, or programmatically. When set by the developer, the debugger shall enter the interactive mode and wait for input from the developer. When a non-developer breakpoint is reached, the debugger must notify the entity that created this breakpoint.

Definition 5.17 (Breakpoint). A breakpoint is a named tuple $(\mathrm{addr} \mapsto \mathit{addr}, \mathrm{for} \mapsto f)$, where *addr* is the address of this breakpoint in the program memory, and b a value that indicates whether this breakpoint is a developer, a scenario, or an event breakpoint (that is, a breakpoint set for the monitor). The set of breakpoints is defined as: $\mathrm{Breakpoint} \stackrel{\mathrm{def}}{=} \mathrm{Addr} \times \{\mathrm{dev}, \mathrm{scn}, \mathrm{evt}\}$.

Example 5.18 (Breakpoint). A breakpoint set by the developer on the second instruction of the program given in Example 5.8 is (pc_2, dev) where pc_2 is the memory address at which the second instruction is loaded. The second instruction is stored as the second element of the tuple and the third component indicates that this breakpoint is set by the developer.

5.3.2 *Watchpoint*

A watchpoint can be intuitively understood as a bookmark on a value of the program. When encountered, the program shall suspend its execution and inform the debugger. Like a breakpoint, a watchpoint can be set by the developer or programmatically. A watchpoint can be triggered by different kinds of accesses: read, write or both.

$$\text{evt2pts}(m, pc, \text{Sym}, e) = \begin{cases} \{(\text{Sym}(m, pc, e.\text{name}), \{\text{r}\}, \text{evt})\} & \text{if } e.\text{type} = \text{ValueRead} \\ \{(\text{Sym}(m, pc, e.\text{name}), \{\text{w}\}, \text{evt})\} & \text{if } e.\text{type} = \text{ValueWrite} \\ \{(\text{Sym}(m, pc, e.\text{name}), \{\text{r}, \text{w}\}, \text{evt})\} & \text{if } e.\text{type} = \text{ValueAccess} \\ \{(\text{Sym}(m, pc, e.\text{name}), \text{evt})\} & \text{if } e.\text{type} = \text{BeforeCall} \\ \{(a, \text{evt}) \mid a \in \text{retPts}(\text{Sym}, m, pc, e.\text{name})\} & \text{if } e.\text{type} = \text{AfterCall} \end{cases}$$

Fig. 6. Event instrumentation: definition of function evt2pts

Definition 5.19 (Watchpoint). A *watchpoint* is a named tuple $(\text{addr} \mapsto addr, \text{access} \mapsto a, \text{for} \mapsto f)$ where *addr* is the address of this watchpoint, a is the the kind of access (r for read, w for write, rw for both) and $f \in \{\text{dev}, \text{scn}, \text{evt}\}$ a value that indicates whether this watchpoint is a developer, a scenario or an event watchpoint. The set of watchpoints is defined as $\text{Watchpoint} \stackrel{\text{def}}{=} \text{Addr} \times (\mathcal{P}(\{\text{r}, \text{w}\}) \setminus \{\emptyset\}) \times \{\text{dev}, \text{scn}, \text{evt}\}$.

Example 5.20 (Watchpoint). A watchpoint set by the developer on variable b in the program given in Example 5.8 is $(\&b, \text{w})$, 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).

Remark 4 (About hardware and software watchpoints). On actual systems, the notification of an access happens whenever a watchpoint is reached. No access lists are built before running each instruction. There are hardware and software watchpoints. Hardware watchpoints are handled by the processor. Debuggers ask the processor to watch some memory cells, and a trap happens whenever a watchpoint is reached. For variables that are stored in registers, the processor must support register watchpoints. Hardware watchpoints are efficient but their number is limited. To overcome these limitations, debuggers implement software watchpoints. The program is run step by step by the debugger. Before the execution of each instruction, the debugger emulates this instruction and computes the list of accesses done by this instruction. Software watchpoints slow down the execution dramatically.

Depending on the platform, watchpoints are triggered before or after the corresponding access. In our model, we chose to trigger watchpoints before the access.

5.3.3 *Event Instrumentation and Instantiation*

We denote the set of breakpoints and watchpoints by $\text{Point} = \text{Breakpoint} \cup \text{Watchpoint}$. An element of Point is referred to as a point. We present the relations between the instrumentation provided by the debbuger and events.

In Fig. 6, we define function $\text{evt2pts} : \text{Mem} \times \text{Addr} \times \text{SymbolTable} \times \text{SymbolicEvent} \to \mathcal{P}(\text{Point})$ that maps a memory, a program counter, a symbol table and an event to a set of points.

Function $\text{retPts} : \text{SymbolTable} \times \text{Mem} \times \text{Addr} \times \text{Symbol}$ maps a symbol table, a memory, a program counter and a symbol to the set of address of exist instructions in the function that corresponds to the given symbol. This function depends on the instruction set of the program and is not defined here for simplicity.

We define $\text{instantiate} : \text{SymbolTable} \times \text{SymbolicEvent} \times \text{Mem} \times \text{Addr}$, the function which maps a symbol table, a symbolic event, a memory and a program counter to a runtime event. $\text{instantiate}(\text{Sym}, e_f, m, pc) = (e_f, l)$ with $l_k = m[\text{Sym}(m, pc, e_f.\text{params}_k)]$. That is, at runtime, a symbolic event is instantiated by getting values of its parameters from the current memory using the symbol table.

Table 1. Debugger Commands

Command	Usage
set(s, v) (resp. set(a, v))	set value of symbol s (resp. at addr. a) to v
get(s) (resp. get(a))	get value of symbol s (resp. at addr. a)
getPC	get value of the pc
setPC(a)	set value of the pc to address a
checkpoint	set a checkpoint
restore(n)	restore checkpoint n
continue (resp. INT)	continue (resp. interrupt) execution
step	execute one step of the program
setPoint(p) (resp. rmPoint(p))	set (resp. remove) a point p

6 OPERATIONAL VIEW

In this section, we describe the behavior of a program under i-RV (the i-RV-program) using operational semantics. I-RV relies on the joint execution of different components: the program, the debugger, the monitor and the scenario.

The program is independent from the other components and describing its own behavior without the other components is meaningful. In this model, the program executes instruction by instruction and interrupts its execution when a breakpoint instruction is encountered or when reaching the program end.

The monitor is also independent from the other components. The monitor issues a verdict whenever it receives an event. In this model, the monitor is able to save its state and restore a saved state.

The debugger does not depend on the monitor or the scenario, but it depends on the program. Describing the debugger behavior independently from the program behavior is not meaningful. We describe the behavior of the program under debug (or debugged program).

The interactively runtime verified program (i-RV program) is composed of these the debugged program, the monitor and the scenario. We introduce and describe the behavior of the scenario and the i-RV program.

6.1 Interface of the Program Under i-RV

In this section, we present the interface of the i-RV-program, that is, its input and output symbols. We use this interface when defining the components of the i-RV-program, and its behavior , which implements this interface.

6.1.1 *Input symbols.*

The i-RV-program initializes its components upon reception of symbol INIT. The developer communicates with the i-RV-program by issuing debugger commands that are forwarded to the debugged program. The set of debugger commands is

$$\begin{aligned}\text{DbgCmd} \stackrel{\text{def}}{=}\ & \{\text{set}(s,v), \text{get}(s), \text{set}(a,v), \text{get}(a), \text{getPC}, \text{setPC(a)} \mid s \in \text{Symbol} \wedge v \in \text{Value} \wedge a \in \text{Addr}\} \\ & \cup \{\text{checkpoint}, \text{continue}, \text{INT}, \text{step}\} \cup \{\text{restore}(n) \mid n \in \mathbb{N}\} \cup \{\text{setPoint}(p), \text{rmPoint}(p) \mid p \in \text{Point}\}.\end{aligned}$$

The usage of these commands is detailed in Table 1. Input symbols in $\{\text{scnCmd(c)} \mid c \in \text{DbgCmd} \cup \{\text{nop}\}\}$ are also used internally to handle the scenario. The set of input symbols of the i-RV-program is $I_{\text{i-RV}} \stackrel{\text{def}}{=} \{\text{INIT}\} \cup \text{DbgCmd} \cup \{\text{scnCmd(c)} \mid c \in \text{DbgCmd} \cup \{\text{nop}\}\}$.

$$\textsc{normalExec}\ \frac{m[pc] \notin \{\mathrm{BREAK}, \mathrm{STOP}\} \qquad (m', pc') = \mathrm{runInstr}(m, pc)}{P\,\langle \mathrm{m} \mapsto m, \mathrm{pc} \mapsto pc\rangle \xrightarrow{-\ /\ -} P\,\left[\mathrm{m} \mapsto m', \mathrm{pc} \mapsto pc'\right]}$$

$$\textsc{bpHit}\ \frac{m[pc] = \mathrm{BREAK}}{P\,\langle \mathrm{m} \mapsto m, \mathrm{pc} \mapsto pc\rangle \xrightarrow{-\ /\ \mathrm{TRAP}} P}$$

Fig. 7. Execution of the program

6.1.2 *Output symbols.*

The i-RV-program outputs values (in Value) requested by the developer and identifiers of checkpoints (in $\mathbb{N} \times \mathbb{N}$) set by the developer. Checkpoint identifiers of the i-RV-program are pairs composed of a checkpoint identifier from the debugged program and a checkpoint identifier from the monitor. The i-RV-program outputs values requested by the developer and identifiers of checkpoints set by the developer. The set of output symbols of the i-RV-program is $O_{\text{i-RV}} = \mathrm{Value} \cup (\mathbb{N} \times \mathbb{N})$.

6.2 The Program

In this section, we present the behavior of a program. We consider a program $Pgrm \stackrel{\text{def}}{=} (\mathrm{Sym}, m_p^0, \mathrm{start}, \mathrm{runInstr}, \mathrm{getAccesses})$ following Definition 5.7 (see Sec. 5.1, p. 13).

Definition 6.1 (Operational semantics of a program). The operational semantics of *Pgrm* is the IOLTS $(\mathrm{Conf_P}, A_\mathrm{P}, \emptyset, \{\mathrm{TRAP}\}, \rightarrow_\mathrm{P})$ where $\mathrm{Conf_P}$ is the set of configurations as per Definition 5.9 and $(\mathrm{m} \mapsto m_p^0, \mathrm{pc} \mapsto \mathrm{start})$ is the initial configuration, and $\rightarrow_\mathrm{P}$ is the least set of transitions abiding by the rules in Fig. 7.

The program has no input symbols: in our model, the program behavior only depends on its initial memory and its original program counter. The program has one output symbol TRAP, which is output when the execution reaches a breakpoint. Recall that we do not model the standard input and output nor the communication with the outside. The evolution of configuration is given by relation $\rightarrow_\mathrm{P}$ which follows the two rules given in Fig. 7:

- Rule normalExec. The end of the program code is represented by instruction STOP. When encountering this instruction, no rule applies, which ends the execution. BREAK is a breakpoint instruction. If current instruction $m[pc]$ in the program memory is not STOP nor BREAK, rule normalExec applies. During an execution step, the memory and the program counter of the program are updated by running the current instruction, using function runInstr. rule normalExec is linked to action (m, pc) such that $(\mathrm{m} \mapsto m, \mathrm{pc} \mapsto pc)$ is the configuration of the program to which the rule is applied.
- Rule bpHit. This rules applies when the current instruction is BREAK. When this instruction is reached, symbol TRAP is output.

Remark 5. The configuration of the program remains unchanged when encountering a breakpoint instruction. The debugged program temporarily replaces the breakpoint instruction by the original instruction in the program when this instruction is to be executed (see Sec. 6.3). Without this mechanism, applying rule bpHit would lead to an infinite loop. This is not possible since without the debugger no breakpoint instruction appears in the code of a program.

6.3 The Debugged Program

In this section, we present the behavior of the debugged program, which models the execution of a program with a debugger. We consider a program $Pgrm \stackrel{\text{def}}{=} (\text{Sym}, m_p^0, \text{start}, \text{runInstr}, \text{getAccesses})$ following Definition 5.7 (see Sec. 5.1, p. 13) with initial configuration P_0.

The debugged program can receive debugger commands in the set DbgCmd defined in Sec. 6.1.

Definition 6.2 (Operational semantics of a debugged program). The operational semantics of the debugged program associated with *Prgm* is the IOLTS $(\text{Conf}_\text{P} \times \text{Conf}_\text{D}, A_\text{PD}, I_\text{PD}, O_\text{PD}, \rightarrow_\text{PD})$ where:

- $\text{Conf}_\text{P} \times \text{Conf}_\text{D}$ is the set of configurations composed of a configuration of the program in Conf_P, and a configuration of the debugger in Conf_D defined as

$$\{\text{P}, \text{I}\} \times \mathcal{P}(\text{Breakpoint}) \times \mathcal{P}(\text{Watchpoint}) \times \text{Checkpoint}^* \times [\text{Point} \rightarrow \text{Event}] \times \mathcal{P}(\text{Point}) \times \text{Mem};$$

- A_PD is the set of actions defined as
$\Big\{$setBreak(b), rmBreak(b), setWatch(w), rmWatch(w), devBreak, scnBreak(b), evtBreak(e), devWatch, scnWatch(w), evtWatch(e), int, cont, trapNoBreak, clearEvents, setSym(s, v), setAddr(a), setPC(a), checkpoint(n), restore(n), exec(m, a) $\mid w \in \text{Watchpoint} \ \wedge \ b \in \text{Breakpoint} \ \wedge \ e \in \text{Event} \ \wedge \ n \in \mathbb{N} \ \wedge \ m \in \text{Mem} \ \wedge \ a \in \text{Addr}\}$,
- $I_\text{PD} \stackrel{\text{def}}{=} \text{DbgCmd} \cup \{\text{instr}(evts) \mid evts \in \mathcal{P}(\text{Event})\} \cup \{\text{clearEvts}\}$ is the set of input symbols, where: instr($evts$) is used to add instrumentation for events for the monitor, clearEvts is used to remove all instrumentation set for events for the monitor
- $O_\text{PD} \stackrel{\text{def}}{=} \mathcal{P}(\text{Point}) \cup \mathcal{P}(\text{Event}) \cup \text{Value} \cup \mathbb{N}$ is the set of outputs.
- $\rightarrow_\text{PD}$ is the least set of transitions defined by semantics rules described later in the next paragraph.

The initial configuration of the debugged program is $((\text{mode} \mapsto \text{I},\ \text{bpts} \mapsto \epsilon,\ \text{wpts} \mapsto \epsilon,\ \text{cpts} \mapsto \epsilon,\ \text{evts} \mapsto \emptyset,\ \text{hdld} \mapsto \emptyset, \text{oi} \mapsto (addr \mapsto \text{BREAK})), P_0)$.

The debugged program receives commands from the developer or the scenario. It also receive requests from the monitor and answers them. A configuration of the debugger $(mode, \mathscr{B}, \mathscr{W}, \mathscr{C}, evts, h, oi) \in \text{Conf}_\text{D}$ is such that:

- $mode \in \{\text{P}, \text{I}\}$ indicates the mode of the debugger. In passive mode (P), the program executes normally and the debugger waits for a breakpoint or a watchpoint to be reached, or for the developer to interrupt the execution. In interactive mode (I), the debugger waits for the developer to issue commands.
- $\mathscr{B}$ (resp. $\mathscr{W}$) is the set of breakpoints (resp. watchpoints) handled by the debugger.
- $\mathscr{C} \in \text{Checkpoint}^*$ is the list of checkpoints saved in the debugger. A checkpoint $c \in \text{Checkpoint}$ is a pair $(p, events) \in \text{Conf}_\text{P} \times \mathcal{P}(\text{Event})$ where p is snapshot of the state of the program, containing its entire memory and the program counter and *events* is the list of events tracked for the monitor.
- *evts* is a function that maps points to events. When an event is requested to the debugged program, points that trigger this event are added.
- h is a set of points that have already been triggered during an execution step. This set is used to ensure that points are not triggered more than once per execution step.
- $oi \in \text{Mem}$ is a mapping that stores instructions in the program memory that have been replaced by breakpoint instructions.

$$\textsc{devWatch}\ \frac{a = \text{getAccesses}(P) \qquad \exists w \in \mathscr{W} \setminus h : \exists k \in \{0, \ldots, |a| - 1\} : \text{match}(a_k, w) \wedge w.\text{for} = \text{dev} \qquad (m', oi') = \text{restoreBP}(\mathscr{B}, pc, oi, m)}{\left(P\left\langle \text{m} \mapsto m \right\rangle, D\left\langle \begin{array}{l} \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \text{bpts} \mapsto \mathscr{B}, \\ \text{wpts} \mapsto \mathscr{W}, \text{oi} \mapsto oi \end{array} \right\rangle\right) \xrightarrow{- \,/\, -} \left(P\left[\text{m} \mapsto m'\right], D\left[\begin{array}{l} \text{mode} \mapsto \text{I}, \text{oi} \mapsto oi', \\ \text{hdld} \mapsto h \cup \{w\} \end{array}\right]\right)}$$

$$\textsc{scnWatch}\ \frac{a = \text{getAccesses}(P) \qquad WPs = (\mathscr{W} \setminus h) \setminus \text{Dom}(evts) \qquad f = \min(\{k \in \{0, \ldots, |a| - 1\} \mid \exists w \in WPs : \text{match}(a_k, w)\}) \qquad \exists w \in WPs : \text{match}(a_f, w) \qquad (m', oi') = \text{restoreBP}(\mathscr{B}, pc, oi, m)}{\left(P\left\langle \text{m} \mapsto m \right\rangle, D\left\langle \begin{array}{l} \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \text{bpts} \mapsto \mathscr{B}, \\ \text{wpts} \mapsto \mathscr{W}, \text{oi} \mapsto oi \end{array} \right\rangle\right) \xrightarrow{- \,/\, w} \left(P\left[\text{m} \mapsto m'\right], D\left[\begin{array}{l} \text{oi} \mapsto oi', \\ \text{hdld} \mapsto h \cup \{w\} \end{array}\right]\right)}$$

$$\textsc{evtWatch}\ \frac{a = \text{getAccesses}(P) \qquad WPs = (\text{Dom}(evts) \cap \mathscr{W}) \setminus h \qquad f = \min(\{k \in \{0, \ldots, |a| - 1\} \mid \exists w \in WPs : \text{match}(a_k, w)\}) \qquad \exists w \in WPs : \text{match}(a_f, w) \qquad (m', oi') = \text{restoreBP}(\mathscr{B}, pc, oi, m) \qquad e = \text{instantiate}(\text{Sym}, evts(w), m, pc)}{\left(P\left\langle \begin{array}{l} \text{m} \mapsto m, \\ \text{pc} \mapsto pc \end{array} \right\rangle, D\left\langle \begin{array}{l} \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \text{bpts} \mapsto \mathscr{B}, \\ \text{wpts} \mapsto \mathscr{W}, \text{oi} \mapsto oi, \text{evts} \mapsto evts \end{array} \right\rangle\right) \xrightarrow{- \,/\, e} \left(P\left[\text{m} \mapsto m'\right], D\left[\begin{array}{l} \text{oi} \mapsto oi', \\ \text{hdld} \mapsto h \cup \{w\} \end{array}\right]\right)}$$

Fig. 8. Handling watchpoints

An output of the debugged program can be either: a set of points $pts \in \mathcal{P}$ (Point) for the scenario, a set of events $events \in \mathcal{P}$ (Event) for the monitor, a value $v \in$ Value requested by the developer or the scenario, an integer $n \in \mathbb{N}$ when setting a checkpoint, used as an identifier for the checkpoint.

Transition relation (evolution of the debugged program). The behavior of the debugged program depends on the current debugger mode. In interactive mode, the debugger waits for the developer to issue commands. In passive mode, program instructions are executed. During the execution, breakpoints and watchpoints can be reached. Points can be set by the developer, the monitor or the scenario. When a point is reached, the debugged program triggers this point. When a developer point is triggered, the debugger mode is set to interactive (I). The execution is therefore suspended, since rules that lead to the execution of an instruction require the debugger to be in passive mode. A non-developer point is either set for the monitor or by the scenario. Monitor points are mapped to events. Whenever a monitor point is triggered, the corresponding event is output. Scenario points are output as-is and handled by the scenario. In passive mode, an execution step consists in executing an instruction, outputting an event, outputting a point or switching to interactive mode. An instruction can trigger several points. An instruction will only be executed when every point it triggers has been triggered. Each time a point is triggered, it is saved in field *hdld* of the debugger. When an instruction is executed, *hdld* is emptied.

We first present rules in Fig. 8 related to watchpoints. In these rules, function getAccesses takes the current program state and returns the list of accesses that will be made when executing the next instruction. For instance, an instruction that adds two variables and saves the result in a third variable will make two read accesses, then one write access. Before executing the next instruction, this list of accesses is checked against the set of watchpoints that are registered in the debugger. Function match : Access × Watchpoint → $\mathbb{B}$ tests whether an access matches a watchpoint and is defined as $\text{match}(a, w) = (a.\text{addr} = w.\text{addr}) \wedge a.\text{mode} \in w.\text{mode}$, for any $a \in$ Access, $w \in$ Watchpoint.

$$\textsc{devBreak}\ \frac{P \xrightarrow{-\ /\ \text{TRAP}} P \qquad \exists b \in (\mathscr{B} \setminus h) : b.\text{addr} = pc \wedge b.\text{for} = \text{dev}}{(P\left\langle \begin{matrix} \text{m} \mapsto m, \\ \text{pc} \mapsto pc \end{matrix} \right\rangle, D\,\langle \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \text{bpts} \mapsto \mathscr{B} \rangle) \xrightarrow{-\ /\ -} (P, D\,[\text{mode} \mapsto \text{I}, \text{hdld} \mapsto h \cup \{b\}])}$$

$$\textsc{scnBreak}\ \frac{P \xrightarrow{-\ /\ \text{TRAP}} P \qquad \exists b \in (\mathscr{B} \setminus h) : b.\text{addr} = pc \wedge b.\text{for} = \text{scn}}{(P\left\langle \begin{matrix} \text{m} \mapsto m, \\ \text{pc} \mapsto pc \end{matrix} \right\rangle, D\,\langle \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \text{bpts} \mapsto \mathscr{B}, \text{evts} \mapsto evts \rangle) \xrightarrow{-\ /\ b} (P, D\,[\text{hdld} \mapsto h \cup \{b\}])}$$

$$\textsc{evtBreak}\ \frac{P \xrightarrow{-\ /\ \text{TRAP}} P \qquad \exists b \in (\text{Dom}(evts) \setminus h) \cap \mathscr{B} : b.\text{addr} = pc}{\left(P\left\langle \begin{matrix} \text{m} \mapsto m, \\ \text{pc} \mapsto pc \end{matrix} \right\rangle, D\left\langle \begin{matrix} \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \\ \text{bpts} \mapsto \mathscr{B}, \text{evts} \mapsto evts \end{matrix} \right\rangle\right) \xrightarrow{-\ /\ \text{instantiate}(\text{Sym}, evts(b), m, pc)} (P, D\,[\text{hdld} \mapsto h \cup \{b\}])}$$

$$\textsc{trapNoBreak}\ \frac{P \xrightarrow{-\ /\ \text{TRAP}} P}{(P\left\langle \begin{matrix} \text{m} \mapsto m, \\ \text{pc} \mapsto pc \end{matrix} \right\rangle, D\,\langle \text{mode} \mapsto \text{P}, \text{hdld} \mapsto h, \text{bpts} \mapsto \mathscr{B} \rangle) \xrightarrow{-\ /\ -} (P\,[\text{m} \mapsto m[pc \mapsto D.\text{oi}(pc)]], D)}$$

Fig. 9. Handling breakpoints

- Rule DEVWATCH applies when a developer watchpoint is reached. The debugged program performs action DEVWATCH. The list of access done by the next instruction in the program is computed. A developer watchpoint known to the debugger (in field *wpts*) matches an access of this list. The debugger becomes interactive and the watchpoint is marked as handled. If a breakpoint is set at the current address in the debugger, the breakpoint instruction is restored in the program memory. This ensures that a breakpoint instruction is always set for any known breakpoint. This is done using function restoreBP : Breakpoint* × Addr × Mem → Mem × Mem defined as

$$\text{restoreBP}(\mathscr{B}, pc, oi, m) = \begin{cases} (m[pc \mapsto \text{BREAK}], oi[pc \mapsto m[pc]]) & \text{if } \exists b \in \mathscr{B} : b.\text{addr} = pc, \\ (m, oi) & \text{otherwise.} \end{cases}$$

- Rule SCNWATCH applies when rule DEVWATCH does not apply, and thus a scenario watchpoint can be triggered. The debugged program performs action SCNWATCH(w), where w is the output watchpoint. Scenario watchpoints are watchpoints that are not in the domain of function *evts* and that are not developer watchpoints. When rule SCNWATCH applies, a scenario watchpoint known to the debugger (in field *wpts*) matches an access. This scenario watchpoint is output and marked as handled.
- Rule EVTWATCH applies when neither rule DEVWATCH nor rule SCNWATCH applies. That is, if there is no developer or scenario watchpoints, rule EVTWATCH may apply. Rule EVTWATCH applies when a monitor watchpoint (that is, a watchpoint in the domain of function *evts*) matches an access. This watchpoint is marked as handled and is converted to an instantiated event which is output. The debugged program performs action EVTWATCH(e), where e is the produced event.

We now present rules in Fig. 9 related to breakpoints. If the current instruction in the program is BREAK, function getAccesses returns an empty list. Therefore, no watchpoint is triggered (rules in Fig. 8 do not apply). An execution step of the program outputs symbol TRAP.

$$\textsc{normalExec}\ \frac{\begin{array}{c}\forall a \in \text{getAccesses}(P), \forall w \in \text{Watchpoint} \setminus h : \neg\text{match}(a, w) \\ P\left[\text{m} \mapsto \text{unInstr}(m, bpts, oi)\right] \xrightarrow{- \,/\, -} P' \left\langle \text{m} \mapsto m_t \right\rangle \\ (m', oi') = \text{restoreBPs}(\mathscr{B}, m_t) \\ o = \text{STOP if } m_t[pc] = \text{STOP, nothing (-) otherwise}\end{array}}{\left(P \left\langle \text{pc} \mapsto pc \right\rangle, D \left\langle \text{mode} \mapsto \text{P}, \text{oi} \mapsto oi, \text{bpts} \mapsto \mathscr{B} \right\rangle\right) \xrightarrow{- \,/\, o} \left(P' \left[\text{m} \mapsto m'\right], D \left[\text{hdld} \mapsto \emptyset, \text{oi} \mapsto oi'\right]\right)}$$

Fig. 10. Normal execution and end of execution

- Rule devBreak If a dev breakpoint is set on address *pc*, the debugger is set to interactive mode (I) and the breakpoint is marked as handled (rule devBreak). The debugged program performs action devBreak.
- Rule scnBreak If no developer breakpoints match address *pc* (rule devBreak does not apply), address *pc* is checked against scenario breakpoints (rule scnBreak). If such a breakpoint exists, it is output and marked as handled. The debugged program performs action scnBreak(b), where b is the output breakpoint.
- Rule evtBreak If neither rule devBreak nor rule scnBreak apply, rule evtBreak may apply. Address *pc* is checked against monitor breakpoints (that are in the domain of function *evts*. If such a breakpoint exists, the corresponding event is instantiated, output and the breakpoint is marked as handled. Rule evtBreak performs action evtBreak(e), where e is the produced event.
- Rule trapNoBreak If no breakpoints match address *pc*, all breakpoints at this address have already been handled. rule trapNoBreak applies. The breakpoint instruction is replaced with the original instruction that was stored in the debugger when the breakpoint was set (see rule setBreak in Fig. 12). during the next steps, watchpoints for this instruction may happen, and the instruction will be executed if the scenario does not change the execution of the program. The breakpoint instruction will be restored each time a watchpoint is triggered, or after the execution of the instruction. Rule trapNoBreak performs action trapNoBreak.

When no watchpoints and breakpoints are triggered, either because all points have already been triggered for this instruction, or because the instruction does not trigger any point, rule normalExec in Fig. 10 applies. The program memory is uninstrumented using function $\text{unInstr} : \text{Mem} \times \text{Breakpoint}^* \times \text{Mem} \to \text{Mem}$ defined as $\text{unInstr}(M, \mathscr{B}, oi) = m \dagger \{b.\text{addr} \mapsto oi(b.\text{addr}) \mid b \in \mathscr{B}\}$. One step of the program is executed. breakpoint instructions are restored using function $\text{restoreBPs} : \text{Breakpoint}^* \times \text{Mem} \to \text{Mem} \times \text{Mem}$ defined as $\text{restoreBPs}(\mathscr{B}, m) = (m \dagger \{b.\text{addr} \mapsto \text{BREAK} \mid b \in \mathscr{B}\}, [b \mapsto \text{BREAK}\ \forall b \in \text{Addr}] \dagger \{b.\text{addr} \mapsto m[b.\text{addr}] \mid b \in \mathscr{B}\})$. The list of handled points is emptied. When the program ends, event STOP is output. When the rule applies, the debugged program performs action $m[pc]$ such that $(P \langle \text{m} \mapsto m, \text{pc} \mapsto pc \rangle, D)$ is the configuration of the debugged program to which the rule is applied.

Remark 6. In actual implementations, for performance reasons, the program is not necessarily uninstrumented at each execution step. Debuggers assume that breakpoint instructions do not affect the execution, which is the case if the program does not read its own instructions and no breakpoint is set on a data (which a debugger should forbid).

We now present the rules related to instrumentation features of the debugged program in Fig. 11.

- Rule clearEvents. When adding instrumentation for a set of events, previous instrumentation (i.e., breakpoints and watchpoints) is removed using rule clearEvents. Rules rmWatch and rmBreak are used to remove points in Dom(*evts*) (i.e., that correspond to events).

$$\textsc{clearEvents}\ \frac{\text{rmPoints}(\{b \in \text{Dom}(evts)\}) \xrightarrow{-\ /\ (P',D')} (P, D)}{(P, D) \xrightarrow{\text{clearEvts}\ /\ -} (P', D')}$$

$$\textsc{instrument}\ \frac{\begin{array}{c}\text{clearEvts} \xrightarrow{-\ /\ (P_t, D_t\ \langle \text{evts} \mapsto evts_t \rangle)} (P, D) \\ (pts, evts') = \text{watchEvents}(P_t, evts_t, events) \\ \text{setPoints}(pts) \xrightarrow{-\ /\ (P',D')} (P_t, D_t\ [\text{evts} \mapsto evts'])\end{array}}{(P, D) \xrightarrow{\text{instr}(events)\ /\ -} (P', D')}$$

Fig. 11. Instrumenting events

$$\textsc{setBreak}\ \frac{\begin{array}{c}b \in \text{Breakpoint} \\ m' = m[b.\text{addr} = \text{BREAK}]\end{array} \qquad oi' = \begin{cases} oi & \text{if } \exists b' \in \mathscr{B} : b'.\text{addr} = b.\text{addr} \\ oi[b.\text{addr} \mapsto m[b.\text{addr}]] & \text{otherwise}\end{cases}}{(P\ \langle \text{m} \mapsto m \rangle, D\ \langle \text{bpts} \mapsto \mathscr{B}, \text{oi} \mapsto oi \rangle) \xrightarrow{\text{setPoint}(b)\ /\ -} (P\ [\text{m} \mapsto m'], D\ [\text{oi} \mapsto oi', \text{bpts} \mapsto \mathscr{B} \cup \{p\}])}$$

$$\textsc{rmBreak}\ \frac{\begin{array}{c}b \in \mathscr{B} \quad a = b.\text{addr} \\ evts' = evts_{\setminus\{b\}}\end{array} \qquad (m', oi') = \begin{cases} (m, oi) & \text{if } \exists b' \in \mathscr{B} \setminus \{b\} : b'.\text{addr} = a \\ (m[a \mapsto oi(addr)], oi[a \mapsto \text{BREAK}]) & \text{otherwise}\end{cases}}{\left(P\ \langle \text{m} \mapsto m \rangle, D \left\langle \begin{array}{l}\text{bpts} \mapsto \mathscr{B}, \text{oi} \mapsto oi, \\ \text{evts} \mapsto evts\end{array} \right\rangle\right) \xrightarrow{\text{rmPoint}(b)\ /\ -} \left(P\ [\text{m} \mapsto m'], D \left[\begin{array}{l}\text{oi} \mapsto oi', \text{bpts} \mapsto \mathscr{B} \setminus \{p\}, \\ \text{evts} \mapsto evts'\end{array}\right]\right)}$$

$$\textsc{setWatch}\ \frac{w \in \text{Watchpoint}}{(P, D\ \langle \text{wpts} \mapsto \mathscr{W} \rangle) \xrightarrow{\text{setPoint}(w)\ /\ -} (P, D\ [\text{wpts} \mapsto \mathscr{W} \cup \{w\}])}$$

$$\textsc{rmWatch}\ \frac{w \in \mathscr{W}}{(P, D\ \langle \text{wpts} \mapsto \mathscr{W} \rangle) \xrightarrow{\text{rmPoint}(w)\ /\ -} (P, D\ [\text{wpts} \mapsto \mathscr{W} \setminus \{w\}])}$$

Fig. 12. Setting breakpoints and watchpoints

- Rule INSTRUMENT. The debugged program can be asked to add instrumentation for a set of events. This happens when the monitor state is updated. Existing instrumentation for events being tracked is cleared using rule CLEAREVENTS. Function watchEvents : $\text{Conf}_\text{P} \times [\text{Point} \to \text{Event}] \times \text{Event}^*$ returns the set of points to add and updates function *evts*. watchEvents is such that $\text{watchEvents}(P,\ evts,\ \epsilon) = (\emptyset,\ evts)$ and $\text{watchEvents}(P,\ evts,\ e \cdot l) = (pts \cup pts',\ evts' \dagger \{p \mapsto e \mid p \in pts\})$ where $pts = \text{evt2pts}(P.\text{m},\ P.\text{pc},\ \text{Sym},\ e)$ and $(pts',\ evts') = \text{watchEvents}(P,\ evts,\ l)$. Points are added to the debugged program using rule SETBREAK or rule SETWATCH, given in Fig. 12 and described later in this section.

We now present the rules provided by the debugged program to set and remove points in Fig. 12.

$$\textsc{checkpoint}\ \frac{m' = m \dagger \{b.addr \mapsto oi(b.addr) \mid b \in \mathscr{B}\} \qquad c = (P\left[\mathrm{m} \mapsto m'\right], \mathrm{Im}(evts))}{(P\langle \mathrm{m} \mapsto m\rangle, D\langle \mathrm{bpts} \mapsto \mathscr{B}, \mathrm{cpts} \mapsto \mathscr{C}, \mathrm{evts} \mapsto evts, \mathrm{oi} \mapsto oi\rangle) \xrightarrow{\text{checkpoint} \,/\, |\mathscr{C}|} (P, D\left[\mathrm{cpts} \mapsto \mathscr{C} \cdot c\right])}$$

$$\textsc{restore}\ \frac{\begin{array}{c}(P_t\langle \mathrm{m} \mapsto m_t\rangle, events) = \mathscr{C}_n \qquad m'_t = m_t \dagger \{b.addr \mapsto \mathrm{BREAK} \mid b \in \mathscr{B}\} \\ \mathscr{B}_{\mathrm{dev}} = \{b \mid b \in \mathscr{B} \ \wedge\ b.\mathrm{for} = \mathrm{dev}\} \qquad oi' = \{b.\mathrm{addr} \mapsto m_t[b.\mathrm{addr}] \mid b \in \mathscr{B}_{\mathrm{dev}}\} \\ (P_t\left[\mathrm{m} \mapsto m'_t\right], D\left[\mathrm{bpts} \mapsto \mathscr{B}_{\mathrm{dev}}, \mathrm{oi} \mapsto oi'\right]) \xrightarrow{\text{instr}(events) \,/\, -} (P', D')\end{array}}{(P, D\langle \mathrm{bpts} \mapsto \mathscr{B}, \mathrm{cpts} \mapsto \mathscr{C}\rangle) \xrightarrow{\text{restore}(n) \,/\, -} (P', D')}$$

Fig. 13. Checkpointing

- Rule setBreak is used to set a breakpoint, which consists in (i) saving the instruction of the program memory at the breakpoint address in the debugger, if no breakpoint has been set at this location (ii) replacing this instruction by BREAK (iii) updating the set of breakpoints in the debugger. When the rule applies, the debugged program performs action setBreak(b), where b is the breakpoint being set.
- Rule rmBreak is used to remove a breakpoint, which consists in:
  - restoring the instruction of the program memory at the breakpoint address from the debugger, if no other breakpoint is set at this location, and, in this case, removing the instruction from the debugger memory,
  - removing this breakpoint in the points to events mapping, if present,
  - updating the set of breakpoints in the debugger.

  When the rule applies, the debugged program performs action rmBreak(b), where b is the breakpoint being removed.
- Rules setWatch and rmWatch are used to set (resp. remove) a watchpoint (rules setWatch and rmWatch) which consists in adding (resp. removing) the watchpoint in the debugger.

Remark 7. Input symbol setPoints(*pts*) (resp. rmPoints(*pts*)) is handled by applying rules setBreak and setWatch (resp. rmBreak and rmWatch) sequentially for each p in *pts*.

We present rules related to checkpointing given in Fig. 12.

- Rule checkpoint is used to set a checkpoint, which consists in saving the program memory without the breakpoint instructions and the set of tracked events in a checkpoint that is added in the debugger. The checkpoint index is output.
- Rule restore is used to restore a checkpointing, which consists in restoring the program memory saved in the checkpoint given by its index, adding the breakpoints instructions for currently set developer breakpoints, and instrumenting the set of events that were being tracked at the moment the checkpoint was set.

Remark 8. The set of breakpoints can be different at the moment of checkpointing and at the moment of restoring the checkpoint. This matches the behavior of GDB when using its built-in checkpointing feature. However, we ensure that breakpoints mapped to event are restored, since the state of the monitor will also be restored at the state corresponding to the moment of checkpointing.

We now present rules in Fig. 14, related to stepping and mode switching.

$$\textsc{stepRedo}\ \frac{\begin{array}{c}\text{Rule } \textsc{devWatch}, \textsc{scnWatch}, \textsc{evtWatch}, \textsc{devBreak}, \textsc{scnBreak},\\ \textsc{evtBreak} \text{ or } \textsc{trapNoBreak} \text{ applies on } (P, D\,[\text{mode} \mapsto \text{P}])\end{array} \quad (P, D\,[\text{mode} \mapsto \text{P}]) \xrightarrow{-\ /\ o} (P_t\,\langle \text{pc} \mapsto pc, \text{m} \mapsto m[pc \mapsto oi(pc)]\rangle, D')}{(P\,\langle \text{pc} \mapsto pc, \text{m} \mapsto m\rangle, D\,\langle \text{mode} \mapsto \text{I}, \text{oi} \mapsto oi\rangle) \xrightarrow{\text{step}\ /\ o} (P', D'\,[\text{mode} \mapsto \text{I}])}$$

$$\textsc{step}\ \frac{(P, D\,[\text{mode} \mapsto \text{P}]) \xrightarrow{-\ /\ o} (P', D')}{(P, D\,\langle \text{mode} \mapsto \text{I}\rangle) \xrightarrow{\text{step}\ /\ o} (P', D'\,[\text{mode} \mapsto \text{I}])}$$

$$\textsc{int}\ \frac{}{(P, D) \xrightarrow{\text{INT}\ /\ -} (P, D\,[\text{mode} \mapsto \text{I}])} \qquad \textsc{cont}\ \frac{}{(P, D\,\langle \text{mode} \mapsto \text{I}\rangle) \xrightarrow{\text{continue}\ /\ -} (P, D\,[\text{mode} \mapsto \text{P}])}$$

Fig. 14. Stepping and interrupting the execution

$$\textsc{setAddr}\ \frac{a \in \text{Addr}, v \in \text{Value}}{(P, D) \xrightarrow{\text{set}(a,v)\ /\ -} (P\,[\text{m} \mapsto m[a \mapsto v]], D)} \qquad \textsc{getAddr}\ \frac{}{(P\,\langle \text{m} \mapsto m\rangle, D) \xrightarrow{\text{get}(a)\ /\ m[a]} (P, D)}$$

$$\textsc{setSym}\ \frac{s \in \text{Symbol}, v \in \text{Value}}{(P\,\langle \text{m} \mapsto m, \text{pc} \mapsto pc\rangle, D) \xrightarrow{\text{set}(s,v)\ /\ -} (P\,[\text{m} \mapsto m[\text{Sym}(m, pc, s) \mapsto v]], D)}$$

$$\textsc{getSym}\ \frac{}{(P\,\langle \text{m} \mapsto m, \text{pc} \mapsto pc\rangle, D) \xrightarrow{\text{get}(a)\ /\ m[\text{Sym}(m,pc,s)]} (P, D)}$$

$$\textsc{setPC}\ \frac{}{(P, D) \xrightarrow{\text{setPC}(a)\ /\ -} (P\,[\text{pc} \mapsto a], D)} \qquad \textsc{getPC}\ \frac{}{(P\,\langle \text{pc} \mapsto pc\rangle, D) \xrightarrow{\text{getPC}\ /\ pc} (P, D)}$$

Fig. 15. Setting and getting values in the program

- Rules INT and CONT. Rule INT changes the debugger mode to interactive, interrupting the execution since normal execution happens only in passive mode. Rule CONT changes the debugger mode to passive, allowing execution to happen normally.
- Rules STEP and STEPREDO. Rule STEP makes the debugged program execute one step. The debugger mode is temporarily set to passive. This allows normal execution. If breakpoints and watchpoints must be handled, or if there is a breakpoint instruction at the current address in the program, rule STEPREDO applies.

We present rules related to commands used to set and get values in the program in Fig. 15.

- Rules GETADDR and SETADDR. Rule GETADDR (resp. rule SETADDR) outputs (resp. sets) a value at the given address in the program memory.
- Rules GETSYM and SETSYM Rule SETSYM (resp. rule SETSYM) outputs (resp. changes) the value at an address of a given symbol using the symbol table Sym
- Rules GETPC and SETPC. Rule GETPC (resp. rule SETPC) outputs (resp. sets) the program counter.

$$\textsc{event}\ \frac{q' = \!\!\rightarrow_M (q, e)}{M\,\langle \text{state} \mapsto q\rangle \xrightarrow{e\ /\ \text{verdict}(q'),\text{eventsNeeded}(q')} M\,[\text{state} \mapsto q']}$$

$$\textsc{checkpoint}\ \frac{}{M\,\langle \text{state} \mapsto q, \text{cpts} \mapsto \mathscr{C}\rangle \xrightarrow{\text{checkpoint}\ /\ |\mathscr{C}|} M\,[\text{cpts} \mapsto \mathscr{C} \cdot q]}$$

$$\textsc{restore}\ \frac{n < |\mathscr{C}|}{M\,\langle \text{cpts} \mapsto \mathscr{C}\rangle \xrightarrow{\text{restore}(n)\ /\ -} M\,[\text{state} \mapsto \mathscr{C}_n]}$$

Fig. 16. Evolution of the monitor

6.4 Monitor

In this section, we integrate the monitor to the debugged program. The monitor evaluates a property against a trace, giving a verdict upon the reception of each event. To be independent from the specification formalism, we assume a monitor given in terms of a set of states Q, a transition function $\rightarrow_M$: $Q \times \text{Event} \rightarrow Q$ which updates the state of the monitor upon each event, and a function verdict : $Q \rightarrow$ Verdict which maps states to verdicts. Before integrating the monitor to the debugged program, we augment its behavior as follows.

Definition 6.3 (Operational semantics of a monitor). The operational semantics of a monitor is the IOLTS $(Q \times Q^*, A_\text{M}, I_\text{M}, O_\text{M}, \rightarrow_{M_\mathscr{C}})$, where $Q \times Q^*$ is the set of configurations and:

- $A_\text{M} \stackrel{\text{def}}{=} \{\textsc{event}(e) \mid e \in \text{Event}\} \cup \{\textsc{checkpoint}(n), \textsc{restore}(n) \mid n \in \mathbb{N}\}$ is the set of actions,
- $I_\text{M} \stackrel{\text{def}}{=} \text{Event} \cup \{\text{checkpoint}\} \cup \{\text{restore}(n) \mid n \in \mathbb{N}\}$ is the set of input symbols,
- $O_\text{M} \stackrel{\text{def}}{=} \text{Verdict} \times \mathbb{N}$ is the set of output symbols,
- $\rightarrow_{M_\mathscr{C}}$ is the transition relation defined as the least set of transitions abiding to the rules given in Fig. 16.

The initial configuration of the monitor is $(q_\text{init}, \epsilon)$.

In a configuration $(\text{state} \mapsto q, \text{cpts} \mapsto \mathscr{C})$ of the monitor, $q \in Q$ is the current state of the monitor, $\mathscr{C} \in Q^*$ is the list of checkpoints of the monitor. Each checkpoint is indexed by its position in this list and is created by copying the current state of the monitor. An input symbol is either an event $e \in \text{Event} \cup \{\text{INIT}\}$ which makes the monitor evolve from one state to the next state, the symbol checkpoint used to set a checkpoint, or the symbol restore(n) is used to restore a checkpoint. An output symbol is either: a verdict $v \in$ Verdict that is output when the state of the monitor changes (after receiving an event), or an integer that is output when setting a checkpoint. This integer identifies the checkpoint.

The transitions between the configurations of the monitor abide by the rules described in Fig. 16.

- Upon the reception of an event, rule EVENT applies and the monitor performs action $\textsc{event}(e)$, where e is the event being received. The new state is computed using function $\rightarrow_M$: $Q \times (\text{Event} \cup \{\text{INIT}\}) \rightarrow Q$, that maps the current state and the received event to the new state (the initial event being INIT). The set of events handled by the transitions of the monitor from this new state is given by function eventsNeeded : $Q \rightarrow \mathcal{P}$ (SymbolicEvent) defined as $\text{eventsNeeded}(q) = \{e_f \in \text{SymbolicEvent} \mid \exists q' \in Q, \exists e \in \text{Event} : \text{symbolic}(e) = e_f \wedge q' = \!\!\rightarrow_M (q, e)\}$, that is, the set of formal events handled by transitions which begin state is the new state. The new verdict is given by function verdict : $Q \rightarrow$ Verdict provided by the monitor. This set of events and the new verdict are output and the configuration of the monitor is updated with the new state.

- Rules CHECKPOINT and RESTORE describe checkpointing for the monitor. When these rules apply, the monitor perform actions CHECKPOINT(n) and RESTORE(n) respectively, where n is the index of the checkpoint being output. Rule CHECKPOINT saves the current state of the monitor in a checkpoint, adds this checkpoint to the list of checkpoints and outputs its index, which is the size of the list before adding the checkpoint. Rule RESTORE takes a checkpoint index and sets the current state of the monitor to the state that was saved.

6.5 The Scenario

In this section, we integrate the scenario to the debugged program. Whenever the monitor issues a verdict, the scenario reacts by executing actions on the i-RV program. In i-RV, the scenario is provided by the developer. For generality, we assume that the behavior of the scenario is similarly described by an IOLTS following the specification described in this section. In practice, and in our implementation, the scenario is described by a small language allowing the developer to specify reactions to monitor verdicts.

The semantics of the scenario depends on the sets $I_{\text{i-RV}}$ and $O_{\text{i-RV}}$ of input and output symbols of the i-RV program defined in Sec. 6.1.

Definition 6.4 (Operational semantics of a scenario). The operational semantics of a scenario is an IOLTS $(\text{Conf}_\text{S}, A_\text{S}, I_\text{S}, O_\text{S}, \rightarrow_S)$ such that:

- $I_\text{S} \stackrel{\text{def}}{=} \text{Point} \times \text{Verdict} \times \{\text{scnCmdReply}(r) \mid r \in O_{\text{i-RV}}\}$ is the set of input symbols,
- $O_\text{S} \stackrel{\text{def}}{=} I_{\text{i-RV}}$ is the set of commands issued by the scenario to the i-RV program

The set of configurations Conf_S, the set of actions A_S and the transition function $\rightarrow_S$ are specific to the definition of a particular scenario and should follow the following rules:

- Upon reception of a verdict from the monitor or a point from the debugger, the scenario shall update its state and either output nothing, or a symbol in $I_{\text{i-RV}}$, which is a command for the i-RV program.
- After sending a command to the i-RV program, the scenario shall receive a symbol scenarioReply(r) such that $r \in O_{\text{i-RV}}$. The scenario shall update its state and either output nothing or a new symbol in $I_{\text{i-RV}}$.
- If the scenario sets a point, it should be able to receive this point as an input symbol.

An input symbol of the scenario can be either a verdict $v \in \text{Verdict}$ generated by the monitor, a point $p \in \text{Point}$ set by the scenario and triggered during the execution, or a symbol scnCmdReply(r), where r is the result of a command issued by the scenario to the i-RV program.

The behavior of the scenario should be such that:

- The scenario handles any verdict from the monitor. For any configuration S of the scenario, for any verdict $v \in \text{Verdict}$, there should exist a command c for the i-RV program in $I_{\text{i-RV}} \cup \{\text{null}\}$ and a new configuration S' such that $(S, v, S', c) \in \rightarrow_S$.
- The scenario handles any answer from the i-RV program. For any configuration S of the scenario, for any reply scnCmdReply(r) such that $r \in O_{\text{i-RV}}$, there should exist a command c for the i-RV program in $I_{\text{i-RV}} \cup \{\text{null}\}$ and a new configuration S' such that $(S, \text{scnCmdReply}(r), S', c) \in \rightarrow_S$.
- The scenario handles any point from the debugged program. This point was requested by the scenario and has been triggered during the execution. The scenario updates its state and outputs $c \in I_{\text{i-RV}} \cup \{\text{nop}\}$ a command for the i-RV program or nop, a command that has no effects.

$$\textsc{init}\ \frac{M \xrightarrow{\text{INIT} \,/\, \mathit{verdict}, \mathit{events}} M' \qquad (P, D) \xrightarrow{\text{instr}(\mathit{events}) \,/\, -} (P', D') \qquad S \xrightarrow{\mathit{verdict} \,/\, c} S_t \qquad (P_{t_2}, D_{t_2}, M_t, S_t) \xrightarrow{\text{scnCmd}(c) \,/\, o} (P', D', M', S')}{(P, D, M, S) \xrightarrow{\text{INIT} \,/\, o} (P', D', M', S)}$$

$$\textsc{normalExec}\ \frac{(P, D) \xrightarrow{-\,/\,-} (P', D')}{(P, D, M, S) \xrightarrow{-\,/\,-} (P', D', M, S)}$$

Fig. 17. Intialization and execution

For any configuration S of the scenario, for any reply $p \in \text{Point}$, there should exist a command c for the i-RV program in $I_{\text{i-RV}} \cup \{\text{nop}\}$ and a new configuration S' such that $(S, p, S', c) \in \rightarrow_S$.

6.6 The Interactively Verified Program

We consider a debugged program PD, a monitor M and a scenario S.

Definition 6.5 (Operational semantics of the i-RV-program). The operational semantics of the program under interactive runtime verification (i-RV-program) associated to debugged program PD, monitor M and scenario S is an IOLTS $(\text{Conf}_{\text{i-RV}}, A_{\text{i-RV}}, I_{\text{i-RV}}, O_{\text{i-RV}}, \rightarrow_{\text{i-RV}})$.

The set of actions of the i-RV-program is $A_{\text{i-RV}} = A_{\text{PD}} \cup A_{\text{M}} \cup A_{\text{S}}$. That is, actions happening in the i-RV-program are actions happening in the debugged program, the monitor or the scenario. The sets of input symbols $I_{\text{i-RV}}$ and of output symbols $O_{\text{i-RV}}$ are defined in Sec. 6.1.

6.6.1 *Initialisation and execution.*

In Fig. 17, we describe the initialization and an execution step of the interactively verified program.

Rule INIT (Fig. 17) initializes the i-RV-program. Upon reception of symbol INIT, the monitor is initialized and outputs an initial verdict and a set of events to track. The set of events is transmitted to the debugged program for instrumentation and the verdict is transmitted the scenario. The scenario issues a command that is run by the i-RV-program.

Rule NORMALEXEC (Fig. 17) does one execution step. During this step, no scenario or monitor point is triggered. After application of this rule, the debugger program may, or may not, be suspended by a developer point.

6.6.2 *Events and non-developer points.*

In Fig. 18, we describe rules that trigger events for the monitor and non-developer points for the scenario.

Rules EXECPOINT and EXECEVENT (Fig. 18) trigger a non-developer point. In rule EXECPOINT, a point is output by the debugged program. The point is forwarded to the scenario. The scenario outputs a command. The command is executed by the i-RV-program.

Rule EXECEVT outputs an event by the debugged program. This event is either generated by reaching a point in the program or STOP, when the execution ends. The event is forwarded to the monitor. The monitor outputs a verdict and a new set of events to track. The set of events is forwarded to the debugged program for instrumentation. The verdict is forwared to the scenario. The scenario outputs a command. The command is executed by the i-RV-program.

$$\textsc{execPoint}\ \frac{(P, D) \xrightarrow{-\ /\ p \in \text{Point}} (P_t, D_t) \qquad S \xrightarrow{p\ /\ c} S' \qquad (P_t, D_t, M, S_t) \xrightarrow{\text{scnCmd}(c)\ /\ o} (P', D', M', S')}{(P, D, M, S) \xrightarrow{-\ /\ o} (P', D', M', S')}$$

$$\textsc{execEvent}\ \frac{\begin{array}{c}(P, D) \xrightarrow{-\ /\ e \in \mathcal{P}(\text{Event})} (P_t, D_t) \qquad M \xrightarrow{e\ /\ verdict, events} M_t \\ (P_t, D_t) \xrightarrow{\text{instr}(events)\ /\ -} (P_{t_2}, D_{t_2}) \qquad S \xrightarrow{verdict\ /\ c} S_t \\ (P_{t_2}, D_{t_2}, M_t, S_t) \xrightarrow{\text{scnCmd}(c)\ /\ o} (P', D', M', S')\end{array}}{(P, D, M, S) \xrightarrow{-\ /\ o} (P', D', M', S')}$$

$$\textsc{stepPoint}\ \frac{(P, D) \xrightarrow{\text{step}\ /\ p \in \text{Point}} (P_t, D_t) \qquad S \xrightarrow{p\ /\ c} S' \qquad (P_t, D_t, M, S_t) \xrightarrow{\text{scnCmd}(c)\ /\ o} (P', D', M', S')}{(P, D, M, S) \xrightarrow{-\ /\ o} (P', D', M', S')}$$

$$\textsc{stepEvent}\ \frac{\begin{array}{c}(P, D) \xrightarrow{\text{step}\ /\ e \in \mathcal{P}(\text{Event})} (P_t, D_t) \qquad M \xrightarrow{e\ /\ verdict, events} M_t \\ (P_t, D_t) \xrightarrow{\text{instr}(events)\ /\ -} (P_{t_2}, D_{t_2}) \qquad S \xrightarrow{verdict\ /\ c} S_t \\ (P_{t_2}, D_{t_2}, M_t, S_t) \xrightarrow{\text{scnCmd}(c)\ /\ o} (P', D', M', S')\end{array}}{(P, D, M, S) \xrightarrow{\text{step}\ /\ o} (P', D', M', S')}$$

Fig. 18. Events and non-developer points

$$\textsc{scnCmd}\ \frac{\begin{array}{c}(P, D, M, S) \xrightarrow{c\ /\ r} (P_t, D_t, M_t, S_t) \\ S_t \xrightarrow{\text{scnCmdReply}(r)\ /\ c'} S_t' \\ (P_t, D_t, M_t, S_t') \xrightarrow{\text{scnCmd}(c')\ /\ o} (P', D', M', S')\end{array}}{(P, D, M, S) \xrightarrow{\text{scnCmd}(c)\ /\ o} (P', D', M', S')}$$

$$\textsc{scnCmdNop}\ \frac{}{(P, D, M, S) \xrightarrow{\text{scnCmd(nop)}\ /\ -} (P, D, M, S)}$$

Fig. 19. Command from the scenario

Rules stepPoint and stepEvent apply whenever command step is used and the debugged program outputs a point or an event.

$$\textsc{checkpoint}\ \frac{(P,D) \xrightarrow{\text{checkpoint} \,/\, c_{\text{PD}}} (P',D') \qquad M \xrightarrow{\text{checkpoint} \,/\, c_{\text{m}}} M'}{(P,D,M,S) \xrightarrow{\text{checkpoint} \,/\, (c_{\text{PD}},c_{\text{M}})} (P',D',M',S)}$$

$$\textsc{restore}\ \frac{(P,D) \xrightarrow{\text{restore}(c_{\text{PD}}) \,/\, -} (P',D') \qquad M \xrightarrow{\text{restore}(c_{\text{M}}) \,/\, -} M'}{(P,D,M,S) \xrightarrow{\text{restore}(c_{\text{PD}},c_{\text{M}}) \,/\, -} (P',D',M',S)}$$

$$\textsc{step}\ \frac{(P,D) \xrightarrow{step \,/\, -} (P',D')}{(P,D,M,S) \xrightarrow{\text{step} \,/\, -} (P',D',M,S)} \qquad \textsc{otherCmd}\ \frac{(P,D) \xrightarrow{\text{c} \,/\, o} (P',D')}{(P,D,M,S) \xrightarrow{\text{c} \,/\, o} (P',D',M,S)}$$

Fig. 20. Stepping, checkpointing and other debugger commands

6.6.3 *Command from the scenario.*

Rules scnCmdNop and scnCmd (Fig. 19) execute a command from the scenario. If the command is nop, rule scnCmdNop applies and nothing happens. Otherwise, rule scnCmd applies. The command is run by the i-RV-program. This command may produce an output. This output is forwarded to the scenario. The scenario outputs a new command. This new command is run by the i-RV-program.

6.6.4 *Stepping, checkpointing and other debugger commands.*

Rules in Fig. 20 handle the debugger commands. Rule checkpoint handles command checkpoint. The debugged program is checkpointed, the monitor is checkpointed, and the i-RV-program outputs the index of the resulting checkpoint by combining indexes generated by the debugged program and the monitor.

Rule checkpoint handles command restore(n). The checkpoints of the debugged program and of the monitor are restored.

Rule step handles command step whenever rules stepPoint and stepEvent do not apply, that is, no point or event will be output by the debugged program.

Rule otherCmd handles other debugger commands (setAddr(a, v), setSymb(s, v), getSymb(s), getAddr(a), getPC, setPC(a), continue, int, setPoint(p), rmPoint(p)). These commands are forwarded to the debugged program.

7 CORRECTNESS OF INTERACTIVE RUNTIME VERIFICATION

We demonstrate the correctness of our models. Correctness is important because it allows to demonstrate that 1) any verdict found by monitors (and in particular those corresponding to a violation of the monitored property) are actual verdicts applying on the system, and 2) monitors do not miss verdict.

7.1 Verifying the Behavior of the Debugged Program

We consider the initial program and the debugged program and show that they are observationally equivalent. More precisely, given the behavior of the initial and debugged programs defined by their LTS as per Sec. 6.2 and Sec. 6.3 respectively, we show that these LTSs are observationally equivalent, that is, they *weakly simulates* each other.

For this purpose, we define a relation between the configurations of the initial and the debugged programs.

Definition 7.1 (Relation between the configurations of the initial and the debugged programs). The relation between the configurations of the initial program (Conf_P) and the debugged program ($\text{Conf}_\text{P} \times \text{Conf}_\text{D}$) is denoted by $\mathcal{R} \subseteq (\text{Conf}_\text{P} \times \text{Conf}_\text{D}) \times \text{Conf}_\text{P}$ and is defined as follows. Any two configurations (P_{dbg}, D) and P of the debugged and the initial programs are in $\mathcal{R}$ if (1), (2) and (3) hold, where:

$$P.\text{m} = \text{unInstr}(P_{\text{dbg}}.\text{m}, D.\text{bpts}, D.\text{oi}) \tag{1}$$

$$P.\text{pc} = P_{\text{dbg}}.\text{pc} \tag{2}$$

$$\forall a \in \text{Addr}, P_{\text{dbg}}.\text{m}[a] \neq \text{BREAK} \implies P_{\text{dbg}}.\text{m}[a] = P.\text{m}[a] \tag{3}$$

The above equations can be understood as follows:

- (1) means that removing the instrumentation from the memory of the debugged program (unInstr(P_{dbg}.m, D.bpts, D.oi)) (see Sec. 6.3) results in the memory of the initial program (P.m);
- (2) means that the program counters of the initial (P.pc) and debugged programs (P_{dbg}.pc) are the same;
- (3) means that at any address, the memory content in the debugged program (P_{dbg}.m[a]) is either a breakpoint or the memory content at the same address in the initial program (P.m[a]).

Intuitively, $\mathcal{R}$ relates configurations of the debugged and the initial programs as follows: if breakpoints are removed from the memory of the debugged program, the resulting memory (resp. the program counter) is equal to the memory (resp. the program counter) of the initial program.

The set of actions is Mem × Addr ∪ {setWatch, rmWatch, getPC, getSym, getAddr, userBreak, scnBreak, evtBreakint, cont, setBreak, rmBreak, userWatch, scnWatch, evtWatch}, trapNoBreak, clearEvents, instrument, stepRedo}. The set of observable actions is Mem × Addr and is denoted by Obs and other actions are considered inobservable.

When restricting the behavior of the debugged program by forbidding the use of rules that modify the program behavior, $\mathcal{R}$ is a weak simulation, as stated by the proposition below.

Proposition 7.2. *Let us consider $\rightarrow_P$ (resp. $\rightarrow_{PD}$) the transition relation of the initial program (resp. the debugged program) where rules* restore, setSym, setAddr, setPC *and rule* checkpoint[3] *are excluded. Let us also consider the set of observable actions Obs. Relation $\mathcal{R}$ (Definition 7.1) is a weak simulation as per Definition 4.4. That is, the initial program P weakly simulates the debugged program (P, D).*

Moreover, the debugged program simulates the initial program, as stated by the following proposition.

Proposition 7.3. *Relation $\mathfrak{R} = \{((P_{\text{dbg}}, D), P) \mid (P, (P_{\text{dbg}}, D)) \in \mathcal{R}\}$ is a weak simulation.*

We prove Proposition 7.2 in Appendix A.1 and Proposition 7.3 in Appendix A.2.

7.2 Guarantees on Monitor Verdicts

Since the initial program and the debugged program weakly simulate each other, their execution produce the same sequence of observable actions. This sequence contains the whole information of the program execution, needed to produce any possible sequence of events abstracting the execution of the initial program. Therefore, any sequence of

[3]Using rule checkpoint is meaningless without rule restore.

Algorithm 1 Executing the program

```
1: function PRGM::STEP
2:     if mem[pc] = BREAK then
3:         return TRAP
4:     if mem[pc] = STOP then
5:         return STOP
6:     (mem, pc) = runInstr(mem, pc)
7:     return OK
```

events that could be deduced using this information from the execution of the initial program can be produced from the execution of the debugged program assuming correct instrumentation (completeness) and any sequence of events produced by a correct instrumentation by the debugged program corresponds to a sequence of event that could be deduced from the execution of the initial program (soundness). Therefore, assuming correct instrumentation, verdicts issued by a monitor from a sequence of events produced by the debugged program correspond to verdicts that would be issued for the execution of the initial program. We point out that instantiate, the instrumentation function that generates events during the debugged program execution, only depends on the state of the program (its memory and its program counter), its symbol table (that is immutable) and the breakpoints set for the monitor.

8 ALGORITHMIC VIEW

We present an algorithmic view of the behavior of the i-RV program. This view is complementary to the operational view given in Sec. 6. This algorithmic view provides a lower-level and more practical description of the behavior of the i-RV-program. This formalization is not needed to adopt the approach. However, it offers a programming-language independent basis for implementation. In this view, we use object-oriented programming style pseudo-code. We first present the program (Sec. 8.1), then the debugged program (Sec. 8.2), the scenario (Sec. 8.3) and then the i-RV-program (Sec. 8.4). The i-RV-program drives the execution. It uses the debugged program, the monitor and the scenario as components.

8.1 The Program

As in Sec. 6.2, the configuration of the program is a 2-tuple $(mem, pc) \in \text{Mem} \times \text{Addr}$. Algorithm 1 describes function PRGM::STEP used to perform an execution step of the program. If the current instruction is a breakpoint (BREAK), TRAP is returned. If the current instruction is the end of the program (STOP), STOP is returned. Otherwise, the current instruction is executed using runInstr. The memory and the program counter are updated and OK is returned.

8.2 The Debugged Program

As in Sec. 6.3, the configuration of the debugged program is a pair consisting of a configuration of the initial program and the configuration of the debugger. In the algorithms of this section, we define functions called by the i-RV-program described in Sec. 8.4. P (resp. D) is a global variable referencing the configuration of the program (resp. the debugger).

8.2.1 *Handling the execution of an instruction in the program.*

In Algorithm 2, we describe an execution step of the debugged program. The program can only evolve in passive mode (D.fieldMode = P). First, the list of accesses done by the next instruction is computed (Line 2). These accesses

Algorithm 2 Handling the execution of an instruction in the program

Precondition: D.fieldMode = P

```
1:  function DBGPROG::EXEC
2:      accessList ← getAccesses(P)                          ▷ the list of accesses done by the next instruction in the program.
3:      wps ← D.wpts \ D.hdld
4:      if ∃w ∈ wps : ∃a ∈ accessList : match(a, w) ∧ w.for = dev then          ▷ Dev watchpoint
5:          (P.m, D.oi) ← restoreBP(D.bpts, P.pc, D.oi, P.m)
6:          D.hdld ← D.hdld ∪ {w}
7:          D.mode ← I
8:          return OK
9:      a_scn ← [a ∈ accessList | ∃w ∈ wps : match(a, w) ∧ w.for = scn]      ▷ Accesses matching a scenario watchpoint
10:     if a_scn is not empty then
11:         let w ∈ wps such that match(head(a_scn), w) ∧ w.for = scn
12:         (P.m, D.oi) ← restoreBP(D.bpts, P.pc, D.oi, P.m)
13:         D.hdld ← D.hdld ∪ {w}
14:         return w
15:     a_evt ← [a ∈ accessList | ∃w ∈ wps : match(a, w) ∧ w.for = evt]      ▷ Accesses matching a monitor watchpoint
16:     if a_evt is not empty then
17:         let w ∈ wps such that match(head(a_evt), w) ∧ w.for = evt
18:         (P.m, D.oi) ← restoreBP(D.bpts, P.pc, D.oi, P.m)
19:         D.hdld ← D.hdld ∪ {w}
20:         return instantiate(Sym, D.evts(w), P.m, P.pc)
21:     switch P.EXEC() do
22:         case OK                                          ▷ No watchpoints, no breakpoints
23:             return OK
24:         case STOP                                        ▷ We reached the end of the program
25:             return STOP
26:         case TRAP                                        ▷ A breakpoint instruction was encountered
27:             B ← {b ∈ D.bpts \ D.hdld | b.addr = P.pc}    ▷ Set of breakpoints at this address that have not been handled
28:             if ∃b ∈ B : b.for = dev then                 ▷ Developer breakpoint
29:                 D.hdld ← D.hdld ∪ {b}
30:                 D.mode ← I
31:                 return OK
32:             if ∃b ∈ B : b.for = scn then                 ▷ Scenario breakpoint
33:                 D.hdld ← D.hdld ∪ {b}
34:                 return b
35:             if ∃b ∈ (Dom(D.evts) \ D.hdld) ∩ D.bpts : b.addr = P.pc then   ▷ Monitor breakpoint
36:                 D.hdld ← D.hdld ∪ {b}
37:                 return instantiate(Sym, D.evts(b), P.m, P.pc)
38:             P.m[pc] ← D.oi(P.pc)                         ▷ No breakpoint to handle, we temporarily restore the original instruction
39:             return TRAP
```

are later matched in the algorithm (using function match defined in Sec. 6.3) with the set of watchpoints managed by the debugger and that have not been handled yet (Computed at Line 3).

- If a developer watchpoint matches (Line 4), the debugger is set to interactive mode (Line 7), which corresponds to rule DEVWATCH (Fig. 8), and returns OK (Line 8). The watchpoint is added to the set of handled points in the debugger (Line 6)
- Otherwise, the list of accesses matching a scenario watchpoint is built (Line 9). If this list is not empty (Line 10), the watchpoint matching the first access of this list is returned (Line 14), which corresponds to rule SCNWATCH (Fig. 8). The watchpoint is added to the set of handled points in the debugger (Line 13)

- Otherwise, the list of accesses matching a monitor watchpoint is built (Line 15). If this list is not empty (Line 16), the event corresponding to the watchpoint matching the first access of this list is returned (Line 20), which corresponds to rule monWatch (Fig. 8). The watchpoint is added to the set of handled points in the debugger (Line 19)

Each time a watchpoint is triggered, if a breakpoint exists at the current instruction, the breakpoint instruction is set at the current program counter to ensure breakpoints are never missed even if a watchpoint makes the scenario or the developer changes the execution flow (Lines 5, 12 and 18).

If no watchpoint is triggered, a program execution step is done (Line 21). The debugged program checks whether a breakpoint is triggered.

- If no breakpoint instruction is encountered, OK is returned (Line 22), which corresponds to rule normalExec (Fig. 10), or STOP if the execution reaches the end of the program (Line 24).
- If a breakpoint instruction is encountered (Line 26), breakpoints are evaluated.
  - If a developer breakpoint matches (Line 28), the debugger is set to interactive mode, which corresponds to rule devBreak (Fig. 9), and the breakpoint is added to the set of handled points.
  - If a scenario breakpoint matches (Line 32), the breakpoint is returned, which corresponds to rule scnBreak (Fig. 9), and the breakpoint is added to the set of handled points.
  - If a monitor breakpoint matches (Line 35), the corresponding event is returned, which corresponds to rule evtBreak (Fig. 9), and the breakpoint is added to the set of handled points.
  - If no breakpoint matches (Line 38), the original instruction is temporarily restored in the program memory and TRAP is returned (which corresponds to rule trapNoBreak (Fig. 9)). This instruction will be executed or and the breakpoint instruction will be set back again.

Algorithm 3 Setting and removing points

```
1:  function dbgprog::setBreak(b ∈ Breakpoint)
2:      if ∄b′ ∈ D.bpts : b′.addr = b.addr then
3:          D.oi ← D.oi[b.addr ↦ m[b.addr]]
4:      m′ ← m[b.addr = BREAK]
5:      D.bpts ← D.bpts ∪ {b}
6:  function dbgprog::rmBreak(b ∈ Breakpoint)
7:      a ← b.addr
8:      D.evts ← D.evts_{\{b}}
9:      if ∄b′ ∈ D.bpts \ {b} : b′.addr = a then
10:         P.m ← P.m[a ↦ D.oi(addr)]
11:         D.oi ← D.oi[a ↦ BREAK]
12:     D.bpts ← D.bpts \ {b}
13: function dbgprog::setWatch(w ∈ D.wpts)
14:     D.wpts ← D.wpts ∪ {w}
15: function dbgprog::rmWatch(w ∈ D.wpts)
16:     D.wpts ← D.wpts \ {w}
17: function dbgprog::setPoint(p ∈ Point)
18:     if p ∈ Breakpoint then
19:         setBreak(p)
20:     else
21:         setWatch(p)
22: function dbgprog::rmPoint(p ∈ Point)
23:     if p ∈ Breakpoint then
24:         rmBreak(p)
25:     else
26:         rmWatch(p)
```

8.2.2 *Setting and removing points.*

In Algorithm 3, we define functions to set and remove points, corresponding to rules in Fig. 12.

Function setBreak (Line 1) sets a breakpoint. This consists in adding a breakpoint to the set of breakpoints of the debugger (Line 4). If there is no other breakpoint at the address of the breakpoint (Line 2), the instruction at this address is saved in field oi of the debugger (Line 5). Instruction BREAK is placed at the address of the breakpoint (Line 3).

Likewise, function rmBreak (Line 6) removes a breakpoint. This consists in removing a breakpoint from the set of breakpoints of the debugger (Line 11). If there is no other breakpoint at the address of the breakpoint (Line 9), we restore the original instruction at this address from *oi* (Line 10).

Function setWatch (Line 13) sets a watchpoint. This consists in adding a watchpoint to the set of watchpoints of the debugger.

Likewise, function rmWatch (Line 15) removes a watchpoint. This consists in removing a watchpoint from the set of watchpoints of the debugger.

Function setPoint (Line 17) calls setBreak (Line 1) or setWatch (Line 13) depending on whether the point in parameter is a breakpoint or a watchpoint. Likewise, function rmPoint (Line 22) calls rmBreak (Line 6) or rmWatch (Line 15) depending on whether the point in parameter is a breakpoint or a watchpoint.

Algorithm 4 Instrumentation

```
1:  function DBGPROG::CLEAREVENTS
2:      addrs ← {b.addr | b ∈ Dom(D.evts) ∩ Breakpoint} \ {b.addr ∈ D.bpts \ Dom(D.evts)}
3:      P.m ← P.m † {a ↦ D.oi(a) | a ∈ addrs}
4:      D.wpts ← D.wpts \ Dom(D.evts)
5:      D.bpts ← D.bpts \ Dom(D.evts)
6:      D.evts ← ∅
7:  function DBGPROG::INSTRUMENT(events ∈ 𝒫(Event))
8:      clearEvents()
9:      (pts, evts) ← watchEvents(P, D.evts, events)
10:     for all p ∈ pts do
11:         setPoint(p)
12:     D.evts ← evts
```

8.2.3 *Instrumentation.*

In Algorithm 4, we define functions corresponding to rules clearEvents and instrument defined in Fig. 11. In function clearEvents (Line 1), the list of addresses of event breakpoints is built (Line 2), corresponding instructions in the program memory are restored (Line 3) and breakpoints and watchpoints corresponding to events are removed (Lines 4 and 5). In function instrument (Line 7), current events are cleared (Line 8), the list of points needed for events to instrument and the mapping from these points to these events are built (Line 9) using function watchEvents defined in Sec. 6.3. These points are set (Line 11) and the mapping of events to points in the debugger is updated (Line 12) according to the result computed at (Line 9) by function watchEvents.

Algorithm 5 Stepping and interrupting the execution

Precondition: $D.\text{mode} = \text{I}$

```
1:  function DBGPROG::STEP
2:      D.mode ← P
3:      r ← exec()
4:      D.mode ← I
5:      return r
```

Precondition: $D.\text{mode} = \text{P}$

```
6:  function DBGPROG::INTERRUPT
7:      D.mode ← I
```

Precondition: $D.\text{mode} = \text{I}$

```
8:  function DBGPROG::CONTINUE
9:      D.mode ← P
```

8.2.4 *Stepping and interrupting the execution.*

In Algorithm 5, we describe commands STEP, INTERRUPT and CONTINUE, corresponding to rules in Fig. 14.

Function STEP (Line 1) implements command step of the interactive mode of the debugger ($D.\text{mode} = \mathbf{I}$). It sets the debugger in passive mode (Line 2), does an execution step (Line 3) and then restores interactive mode (Line 4). An execution step can produce an event or a point that is returned at Line 5.

Function INTERRUPT (Line 6) sets the debugger in interactive mode.

Function CONTINUE (Line 8) sets the debugger in passive mode. The behavior of rule STEPREDO (Fig. 14, Sec. 6.3) is taken into account in function EXEC (Algorithm 2).

Algorithm 6 Controlling the program memory and counter

```
1:  function DBGPROG::SETADDR(a ∈ Addr, v ∈ Value)
2:  |  P.m[a] ← v
3:  function DBGPROG::GETADDR(a ∈ Addr)
4:  |  return P.m[a]
5:  function DBGPROG::SETSYM(s ∈ Symbol, v ∈ Value)
6:  |  P.m[Sym(P.m, P.pc, s)] ← v
7:  function DBGPROG::GETSYM(s ∈ Symbol)
8:  |  return P.m[Sym(P.m, P.pc, s)]
9:  function DBGPROG::SETPC(SETPC(a))
10: |  P.pc ← a
11: function DBGPROG::GETPC(GETPC(a))
12: |  return P.pc
```

8.2.5 *Controlling the program memory and counter.*

Functions in Algorithm 6 correspond to rules in Fig. 15 and are used to set and get values in the program.

Algorithm 7 Checkpointing

```
1: function DBGPROG::CHECKPOINT
2: |  D.cpts.push(((P.m † {b.addr ↦ oi(b.addr) | b ∈ D.bpts}, P.pc), Im(D.evts)))
3: |  return D.cpts.length
4: function DBGPROG::RESTORE(cid ∈ ℕ)
5: |  ((m_t, pc), events) ← D.cpts_cid
6: |  P.pc ← pc
7: |  D.bpts ← {b ∈ D.bpts | b.for = dev}
8: |  D.oi ← {b.addr ↦ m_t[b.addr] | b ∈ D.bpts}
9: |  P.m ← m_t † {b.addr ↦ BREAK | b ∈ D.bpts}
10: |  PD.INSTRUMENT(events)
```

8.2.6 *Checkpointing.*

Functions in Algorithm 7 are used to checkpoint and restore the debugged program. Function CHECKPOINT stores the program memory (with breakpoint instructions replaced by the original instructions of the program), the program counter and the set of events being tracked by the debugger (Line 2). The function returns the identifier of the checkpoint, which is the index in the sequence of checkpoints known to the debugger (Line 3).

Remark 9. The set of breakpoints is not saved: the set of active developer breakpoints in the restored program will be the set of active developer breakpoints before restoring.

Function RESTORE restores the checkpoint $D.\text{cpts}_{cid}$ given by the parameter identifier *cid* from the list of checkpoints $D.\text{cpts}$ known to the debugger. The program counter is restored (Line 6), non-developer breakpoints are discarded (Line 7), the map of original instructions in the debugger is built (Line 8), the memory is restored while keeping developer breakpoints instructions (Line 9) and checkpointed events are instrumented (Line 10) by function INSTRUMENT defined in Algorithm 4 at Line 7.

8.3 The Scenario

We present an implementation of the scenario. In the operational view (Sec. 6.5), for the sake of generality, the scenario is specified in a generic way. In this section, we propose a more specific, practical definition of a scenario. The scenario behavior is defined using a map that links verdicts from the monitor to actions. An action a in $\mathcal{A}$ is a function that commands the i-RV program and updates the environment of the scenario referenced by variable *env*. Algorithm 8 defines the behavior of the scenario.

Algorithm 8 Handling the scenario

```
1:  function SCN::EXECACTION(a ∈ 𝒜)
2:      if a = null then
3:          return nop
4:      (cmd, ar, env′) ← a(env)
5:      env ← env′
6:      actionReply ← ar
7:      if ∃ (p, a′) ∈ Point × 𝒜 : c = setPoint(p, a′) then
8:          points ← points[p ↦ a′]
9:          return setPoint(p)
10:     if ∃p ∈ Point : c = rmPoint(p) then
11:         delete points[p]
12:     return c
13: function SCN::APPLYVERDICT(v ∈ Verdict)
14:     return EXECACTION(actions(verdict))
15: function SCN::APPLYCMDREPLY(r ∈ O_i-RV)
16:     if actionReply = null then
17:         return null
18:     r ← EXECACTION(actionReply(r))
19:     actionReply ← null
20:     return r
21: function SCN::APPLYPOINT(p ∈ Point)
22:     return EXECACTION(points(p))
```

Function APPLYVERDICT is called when the monitor issues a verdict. This function retrieves the action associated with this verdict using map *actions*, given in the definition of the scenario. This action is then executed by function EXECACTION.

Function EXECACTION executes action a with the environment of the scenario in parameter (Line 4). The action returns a triple (cmd, ar, env') where:

- *cmd* is a command to be run by, and returned to, the i-RV program;
- *ar* is a callback function, or null. This function takes the reply of the i-RV program to the command, and returns an action to run.
- env' is the new environment of the scenario.

The scenario environment is updated (Line 5) and the callback function is saved in variable *actionReply* (Line 6) to handle the result of the command that will be run by the i-RV-program.

- If *cmd* is an action of the form $\text{setPoint}(p, a')$ for some point p and some action a' (Line 7), command $\text{setPoint}(p)$ is returned instead. Point p is set and mapped to action a' in *points*. Action a' is run when p is triggered in the debugged program.

- If *cmd* is an action of the form rmPoint(p) for some point p (Line 10), point p is removed from map *points* and *cmd* is returned as is (Line 12).
- Any other command is returned as is (Line 12).

Function applyPoint is called when a point set by the scenario is triggered. The corresponding action in *points* is executed.

When a command is run by the i-RV program for the scenario, the output of this command is forwarded to the scenario by calling function applyCmdReply. If a callback function is present (i.e., *actionReply* is not null), this function is run with the reply as parameter. This function returns an action that is executed.

8.4 The Interactively Runtime Verified Program

We consider an interactively verified program *PDMS* as defined in Sec. 6.6, using a scenario as defined in Sec. 6.5. We suppose a monitor M with initial configuration M_0 that provides a method applyEvent. This method takes an event as parameter and returns a pair $(v, events) \in \text{Verdict} \times \mathcal{P}$ (SymbolicEvent) where v is a verdict and *events* is a set of symbolic events to track in the debugged program.

The initial configuration of the i-RV-program is (P_0, D_0, M_0, S_0). In the algorithms of this section, the global state is represented by the states of the debugged program *PD*, the monitor M and the scenario S. We also allow the i-RV-program to access the states of the program P and the debugger D included in the state of the debugged program. We present the general behavior at the end of this section, in Algorithm 11.

Algorithm 9 Initialisation and execution

```
1:  function IRV::INIT
2:      (verdict, events) ← M.INIT()
3:      PD.INSTRUMENT(events)
4:      cmd ← S.APPLYVERDICT(verdict)
5:      return APPLYSCENARIOCMD(cmd)
Precondition: D.fieldMode = P
6:  function IRV::EXEC
7:      r ← PD.EXEC()
8:      switch r do
9:          case r ∈ Point
10:             cmd ← S.APPLYPOINT(r)
11:             return APPLYSCENARIOCMD(cmd)
12:         case r ∈ Event
13:             (verdict, events) ← M.APPLYEVENT(r)
14:             PD.INSTRUMENT(events)
15:             cmd ← S.APPLYVERDICT(verdict)
16:             return APPLYSCENARIOCMD(cmd)
17:         case other
18:             return r
19: function IRV::APPLYSCENARIOCMD(cmd)
20:     if cmd = nop then
21:         return null
22:     r ← Execute cmd.name(...cmd.params)
23:     cmd ← S.APPLYCMDREPLY(r)
24:     return APPLYSCENARIOCMD(cmd)
```

8.4.1 *Initialisation and execution.*

In Algorithm 9, we describe the initialization and an execution step of the interactively verified program.

In function init (Line 1), the monitor is initialized (Line 2). Then, instrumentation for the initial state of the monitor is requested to the debugged program (Line 3). Finally, scenario for the initial verdict is applied (Line 4) and the command returned by the scenario is passed to function applyScenarioCmd (Line 5).

Function applyScenarioCmd (Line 19) executes a command (i.e., calls the function given in the command by its name with given parameters) at Line 22. The result of this command is forwarded to the scenario (Line 23, and the new

Algorithm 10 Stepping, checkpointing and other debugger commands

```
1: function IRV::CHECKPOINT
2:     return (PD.CHECKPOINT(), M.CHECKPOINT())
3: function IRV::RESTORE((c_PD, c_M))
4:     PD.RESTORE(c_PD)
5:     M.RESTORE(c_M)

Precondition: D.mode = I
6: function STEP
7:     D.mode ← P
8:     r ← EXEC()
9:     D.mode ← I
10:    return r
```

command returned by the scenario is executed using function APPLYSCENARIOCMD (Line 24). This recursive process ends when the scenario returns the special command nop (Line 20).

Function EXEC (Line 6) performs an execution step of the debugged program (Line 7). Making the program evolve requires the debugger to be in passive mode. (D.fieldMode = P).

- If the debugged program returns a point (Line 9), this point is forwarded to the scenario. The scenario may return a command to perform. This command is executed using function APPLYSCENARIOCMD (Line 11).
- If the debugged program returns an event (Line 12), this event is forwarded to the monitor. The monitor returns a verdict and a new set of events to instrument. Instrumentation is requested to the debugged program (Line 14) and the verdict is forwarded to the scenario (Line 15). The scenario may return a command to perform. This command is executed using function APPLYSCENARIOCMD (Line 16).
- Any other value (OK, TRAP) returned by the debugged program is returned to the caller (Line 11).

8.4.2 *Stepping, checkpointing and other debugger commands.*

In Algorithm 10, we define functions CHECKPOINT, RESTORE and STEP.

Function CHECKPOINT (Line 1) returns a checkpoint composed of a checkpoint of the debugged program and a checkpoint of the monitor.

Function RESTORE (Line 3) restores the debugged program (Line 4) and the monitor (Line 5).

Function STEP (Line 6) implements command step of the interactive mode of the debugger (D.mode = I). It temporarily sets the debugged program in passive mode (Line 7), performs an execution step (Line 8), then restore interactive mode (Line 9). The function returns the value produced by the execution step to the caller (Line 10).

8.4.3 *General behavior.*

In Algorithm 11, we describe the general behavior of the interactively verified program. First, the i-RV-program is initialized using function INIT (Line 1). Then, while nothing interrupts the execution, if the debugger is in passive mode (Line 4), an execution step is performed (Line 4). Otherwise, in interactive mode (Line 8), the developer inputs a command (Line 9) which is executed.

9 IMPLEMENTATION: VERDE

To evaluate our approach in terms of usefulness and performance, we implemented it in a tool called Verde. We give an overview of Verde in Sec. 9.1. We then present how Verde handles the program execution, the property evaluation, and the instrumentation in Sec. 9.2. We then present how checkpointing is done in Verde Sec. 9.2.4. We finally explain how to use Verde in Sec. 9.3.

Algorithm 11 General behavior

```
1:  INIT()
2:  cont ← true
3:  while cont do
4:      if D.mode = P then
5:          r ← EXEC()
6:          if r = STOP then
7:              cont = false
8:      else
9:          switch get developer command do
10:             case step
11:                 STEP()
12:             case continue
13:                 PD.CONTINUE()
14:             case set watchpoint addr read write
15:                 PD.SETWATCH((addr, {r | read} ∪
    {w | write}, dev))
16:             case set breakpoint addr
17:                 PD.SETBREAK((addr, dev))
18:             case unset watchpoint addr read write
19:                 PD.RMWATCH((addr, {r | read} ∪
    {w | write}, dev))
20:             case unset breakpoint addr
21:                 PD.RMBREAK((addr, dev))
22:             case print value at addr
23:                 print(PD.GETADDR(addr))
24:             case set value v at addr
25:                 PD.SETADDR(addr, v)
26:             case print value of symbol
27:                 print(PD.GETSYM(symbol))
28:             case set value v of symbol
29:                 PD.SETSYM(symbol, v)
30:             case print program counter
31:                 print(PD.GETPC())
32:             case set program counter addr
33:                 PD.SETPC(addr)
34:             case checkpoint
35:                 print(CHECKPOINT())
36:             case restore cid
37:                 RESTORE(cid)
```

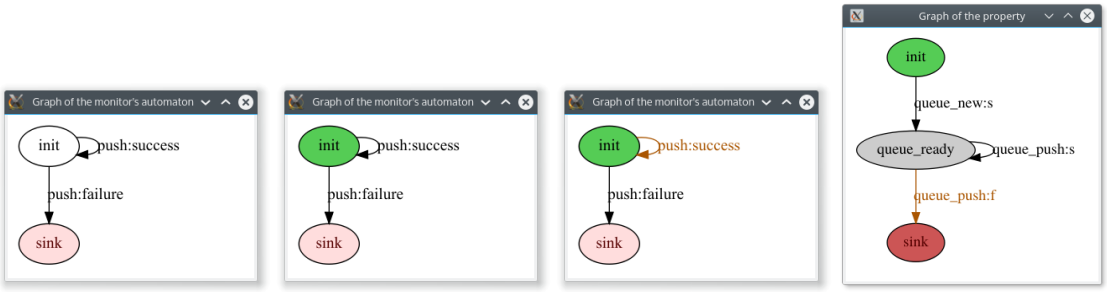

Fig. 21. During the execution of the property given in Fig. 25, the following graphs can be seen respectively before initialization of the property initialization, on initialization, while the property is verified and when the property becomes falsified. Light red, red, brown and gray respectively correspond to non accepting state, a current non accepting state, a transition taken during the last state change and a state which was current before the last state change. Graphs are automatically drawn using Graphviz and colors animated during the execution.

9.1 Overview

Verde[4] is a GDB-based implementation of interactive runtime verification. Verde can be used with programs written in any programming language supported by GDB. Verde supports the verification of several properties by means of monitors executing independently. Each monitor automatically sets and deletes breakpoints according to the events that are relevant to the current states of the monitored properties. Verde provides a graphical and animated view of the

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

properties being checked at runtime (see Fig. 21). The view eases understanding the current evaluation of the property (and, as a consequence, the program) since it offers to the developer a visualization of an abstracted history of the program execution. Verde also lets the developer control the monitors and access their internal state (property instances, current states, environments).

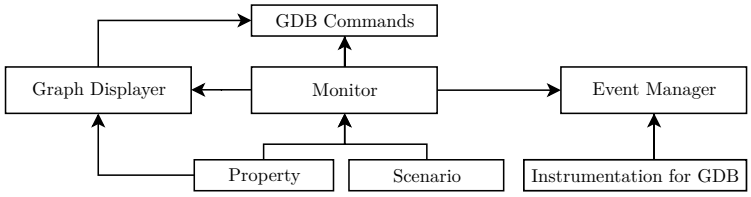

Fig. 22. Organization of the code of Verde

Architecture. Verde is written in Python and works seamlessly as a GDB plugin by using the Python API provided by GDB[5]. The organization of the code is depicted in Fig. 22. The developer controls Verde by using commands defined in module GDB Commands. These commands are available from the GDB command-line interface and allow creating monitors (class Monitor) by loading properties (class Property) and scenarios (class Scenario). Module Graph Displayer provides a graphical view of running monitors. If the view is enabled, Verde shows graphs depicting the properties. Properties are drawn using Graphviz[6] in SVG[7]. As the current state of the monitor changes, the graphical view is updated: the current state is shown in green if it is accepting, in red if it is not accepting. Taken transitions are represented in brown. Module Event Manager defines the interface between monitors and the instrumentation module, which defines methods to handle breakpoints and watchpoints in GDB.

9.2 Program Execution, Property Evaluation, Instrumentation and Checkpointing

In this section, we describe how instrumentation is done, how events are produced, how properties are evaluated and how checkpointing is done during the program execution.

9.2.1 *Program Execution with Verde*

Fig. 23 depicts the execution of a program with Verde. When a breakpoint or a watchpoint in the monitored program is reached, the execution of the program is suspended. If the breakpoint (or the watchpoint) was set by Verde, an event

[5] https://sourceware.org/gdb/onlinedocs/gdb/Python.html
[6] Graphviz is a set of graph drawing tools - https://www.graphviz.org/
[7] the Scalable Vector Graphics format - http://www.w3.org/2000/svg

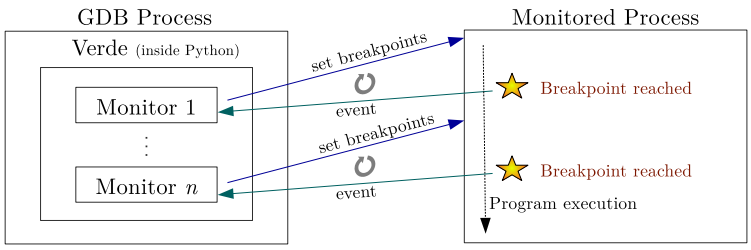

Fig. 23. Instrumentation in Verde

```
on entering non-accepting state {
    print("Scenario: a non-accepting state has been reached! (" +
        old_state + " -> " + new_state + ")")

    c = checkpoint()

    def called_when_checkpoint_ready():
        cid = checkpoint_get_id(c)
        print(
            " *** Checkpoint on this breakage:", cid,
            "\n *** To restore, type verde checkpoint-restart", cid
        )

    after_breakpoint(called_when_checkpoint_ready)
}
```

Fig. 24. This scenario sets a checkpoint each time a non-accepting state is entered. The execution of the program is not suspended.

is produced, the state of the property and the instrumentation are updated. Scenarios react to monitor updates by executing actions that modify the program state, the debugger and the scenario itself. In Verde, each monitor can be associated with a scenario. A scenario is a list of reactions to monitor state changes. When the monitor updates its state, the scenario runs the corresponding action. These actions, written in Python, also have a dedicated environment that can be initialized in the definition of the scenario if needed. An example of a scenario in Verde is given in Fig. 24.

9.2.2 *Handling Instrumentation and Event Production*

Setting and removing breakpoints and watchpoints. When the monitor reaches a new state (including the initial state), the list of transitions of the new state is browsed. Each transition includes a formal event. The event can be a function call or a variable access. Instrumentation is removed for events that are not involved in the evaluation of the property anymore, and instrumentation (breakpoints and watchpoints) is added for events that are involved. Events are requested and released through an event manager. The event manager is an interface between the monitor and the instrumentation module. It keeps track of the active events. It does not depend on the monitor and abstracts away how instrumentation is performed. The event manager sends requests to the instrumentation module, which sets breakpoints and watchpoints by using the API provided by the debugger. Several breakpoints may correspond to one event, and several events can use the same breakpoints. When a breakpoint or a watchpoint is hit, the instrumentation module instantiates the event by retrieving parameters values from the program and forwards this event to the event manager, which in turn forwards the event to the monitor, which updates its state.

Handling after events. After events are triggered when a function ends. GDB does not provide straightforward means to instrument after events out of the box. However, GDB provides finish breakpoints. A finish breakpoints triggers when the frame of the call stack on which it is set returns. To instrument for an after event, a regular breakpoint is first added to the beginning of the function. When the breakpoint is reached, a finish breakpoint is set at the current frame. This finish breakpoint triggers when the function returns. Finish breakpoints are also set on each frame in the call stack that corresponds to the instrumented function, accounting for recursive calls and after events set during the execution of the instrumented function.

9.2.3 *Parametric Trace Slicing*

We implemented trace slicing in Verde, inspiring from [7]. Trace slicing is a feature that allows expressing parametric properties. Such properties get instantiated with runtime values and are evaluated over subsequences of the whole trace of the program execution instead of the whole trace itself. These subsequences of the trace are related to particular instances of parameters (e.g. objects) in the program. For instance, Fig. 2 (p. 13) depicts a property specifying a behavior on each queue created during the program execution. When evaluating this property, each queue is associated with a specific state that does not depend on the states associated with the other queues. Thus, the monitor keeps track of several states and must, for each event, determine which state must be updated. For this property, the queue is a parameter of every event (init, push, pop). This parameter is used to determine the state to be updated. In general, an instance can be defined by multiple parameters. For example, a web page containing some element E can be opened in several tabs in a browser, which itself consists of several windows. An instance of element E is identified by its identifier (unique within a tab) as well as the tab identifier (unique within a window) and the window identifier.

We describe how trace slicing is implemented in Verde. Properties define the list of formal parameters on which the trace must be sliced. In the property depicted in Fig. 2, the unique parameter in this list is the queue. In the monitor, each current state is mapped to an instance of these parameters (the slice instance). The monitor initially has one current state: the initial state. This state is mapped to the slice instance in which each parameter to the special undetermined value $\perp$ (the parameter is not instantiated). When the monitor receives a new event, the slice instance is built from its parameter, and then, for each current state:

- If the state is associated with a slice instance such that all parameters of the event are equal to the parameters of the slice instance, then the state is updated according to the property semantics, i.e., by evaluating the guard and the success or the failure block.
- Otherwise, the state is copied into a new current state if:
  - every instantiated parameter in the slice instance of the state is equal to the corresponding event parameter value, and
  - there is no slice instance in the monitor for which the values of all parameters in the event are equal to the corresponding values in the slice instance.

  The new state is associated to the slice instance built by merging the state slice instance and the event slice instance. The event is directed to this new state.

When a state is copied to form a new current state, the environment containing variables used in the property is also copied, so the new state inherits from the values stored in the state its was copied from. Primitive values are not shared. More complex constructions like dictionaries and lists are, however, shared.

Handling Disappearing Instances. Sometimes, an instance of an object is destructed (freed), and then a new instance is created and gets the same identifier. This can be observed by running the following C program, in which a manually allocated area of 4 bytes is requested using malloc, then freed, than another area of 4 bytes is created:

```
#include <stdlib.h>
#include <stdio.h>
int main() {
    void* p = malloc(4);
    printf("%p ", p); free(p);
```

```
    printf("%p\n", malloc(4)); return 0;
}
```

On a x86-64 machine running Debian Buster, compiling this program with gcc and running it outputs "0x564e53699260 0x564e53699260". However, the two objects that have the same identifier should be handled as distinct objects.

To handle this in Verde, a state can be marked as final, meaning that the state must not be updated anymore. When such a state is reached, the corresponding slice instance is forgotten. As such, when a new instance with the same identifier is seen through an event, a new state will be created in the monitor. This feature also helps keeping the output of the monitor clean and allows optimizing resource usage.

9.2.4 *Checkpointing*

Checkpointing can be done from the scenario to allow going back at a specific point of the execution. Verde features two methods for checkpointing processes 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 is that multithreaded programming is not supported since fork() keeps only one thread in the new process. The second method uses CRIU[8], which supports multithreaded processes and trees of processes. CRIU uses the ptrace API to attach the 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.

Beside CRIU, other checkpointing solutions exist on Linux (cf. the CRIU website[9] for a comparison). We chose CRIU because it does not require preloading any library nor special kernel module. Instead, features required to checkpoint processes with CRIU have been integrated in the mainline Linux kernel. This allows for an easier setup as well as fewer differences between an uninstrumented process and a process under debug. CRIU also seems to be the currently most active and supported solution. CRIU authors state that it is impossible to checkpoint processes debugged under GDB because both tools use the ptrace API provided by the kernel. However, we were able to work around this limitation by suspending and detaching the process from GDB before checkpointing with CRIU, and then reattaching the process to GDB and restoring the breakpoints and watchpoints that were present before checkpointing, and then resuming the execution. Support for other checkpointing solutions may be added as needed.

9.3 Using Verde

In this section, we present how to use Verde. A typical usage session begins by launching GDB and Verde (which can be automatically loaded by configuring GDB appropriately). Then, the developer 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 developer starts the execution. It is also possible to display the graph of the property with the show-graph subcommand (see Fig. 21, p. 40).

```
$ 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
```

[8] Checkpoint/Restore In Userspace is a community-driven project started by Virtuozzo kernel engineers in 2011 to allow checkpointing and restoring on Linux. See https://criu.org/.
[9] https://criu.org/Comparison_to_other_CR_projects

Command	Effect
verde activate	Activates all the commands monitor related commands
verde checkpoint	Sets a checkpoint for the program and each managed monitor
verde checkpoint-restart	Restores a checkpoint
verde cmd-group-begin	Begins a group of commands
verde cmd-group-end	Ends a group of command
verde delete	Deletes a monitor
verde exec	Executes an method in the current monitor.
verde get-current	Prints the name of the current monitor
verde load-functions	Loads a developer defined functions file
verde load-property	Loads a property file and possibly a function file in the given monitor
verde load-scenario	Loads a scenario
verde new	Creates a monitor that will also become the current monitor
verde run	Runs the monitor
verde run-with-program	Running the monitor and the program at the same time
verde set-current	Sets the current monitor
verde show-graph	Shows the graph of the monitor in a window and animates it at runtime

Table 2. List of Verde commands.

```
...
[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)
```

Verde provides more fine-tuned commands to handle cases when properties and functions need to be loaded separately, or when properties and the program need to be run at different times. A list of commands is given in Table 2

Writing Properties. Verde provides a Domain Specific Language for writing properties[10]. An informal grammar is given in Fig. 26. Fig. 25 depicts 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, whether it is final (useful for trace slicing, see Sec. 9.2.3), 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 None. 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

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

```
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 sink non-accepting sink_reached()

state queue_ready accepting {
    transition {
        event queue_push(queue) {
            return N < maxSize
        }
        success {
            N = N + 1
            print("nb elem: " + str(N))
        } queue_ready

        failure sink
    }

    transition {
        event queue_pop(queue) {
            return N > 0
        }

        success {
            N = N - 1
            print("nb elem: "+str(N))
        } queue_ready

        failure sink
    }
}
```

Fig. 25. Verde version of the property in Figure 2

```
[slice on [param,] +]?

[initialization {
    Python code
}]?

[state state_name[[non-]?accepting]? [final]? [action_name()]? {
  [transition {
    [before | after]? event event_name([param,] * ) [{
      Python code returning True False or None
    }]?
    [success [{
        Python code
    }]? [action_name()]? state_name]?
    [failure [{
        Python code
    }]? [action_name()]? state_name]?
  }] *
}] +
```

Fig. 26. Informal grammar for the automaton-based property description language in Verde

free Python code block that returns True (resp. False) if the guard succeeds (resp. fails) and None if the transition should be ignored.

10 EXPERIMENTS

We report on seven experiments carried out with Verde to measure its usefulness in finding and correcting bugs and its efficiency from a performance point of view[11]. We discuss the objective and possible limitations (threat to validity) of each experiment. These experiments also illustrate how a developer uses Verde in practice.

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

10.1 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, 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[12]. We did not discover the bug using i-RV[13]. However, it helped us determine its origin in the code of zsh and fix it. A fix has since been released.

10.2 Automatic Checkpointing to Debug a Sudoku Solver

We evaluated i-RV by mutating the code of a backtracking Sudoku solver[14]. 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[15] 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.

[12] The code of the property is in Appendix C. We worked on commit 85ba685 of zsh.
[13] The bug was reported at https://sourceforge.net/p/zsh/bugs/87/
[14] https://github.com/jakub-m/sudoku-solver
[15] https://github.com/arun-babu/mutate.py

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 and arguably, the program is small enough to be debugged using a traditional interactive debugger. However, the experiment 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.

10.3 Multi-Threaded Producer-Consumers

The purpose of this experiment is 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. 2) 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. 25. We ran the program with Verde. The execution stopped in the state sink (defined at line 39 of Fig. 25). In the debugger, we had access to the precise line in the source code from which the function was 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 shows how Verde helps 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.

10.4 Micro-benchmark

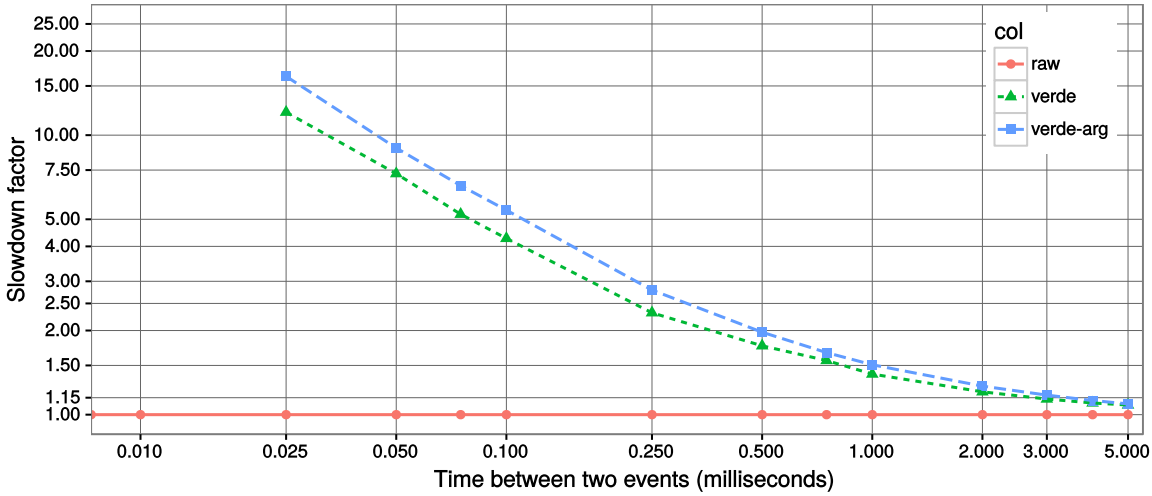

Fig. 27. Instrumentation overhead with Verde. raw (red) is the execution of the program without Verde. verde (green) is the execution with Verde. The overhead includes the breakpoint-based instrumentation. verde-arg (blue) is the execution with Verde, and one parameter is used in the produced events. The overhead includes the instrumentation and the time used to retrieve one value per event.

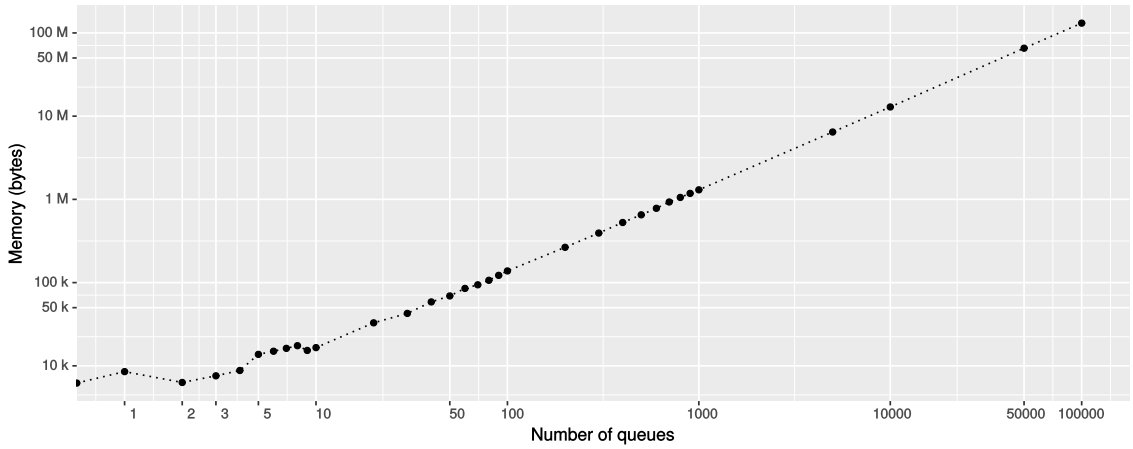

Fig. 28. Memory consumption per number of tracked queues

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 16.04 and Linux 4.4.0-109 with GDB 7.11, are presented in Fig. 27. 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, with a slowdown factor greater than two, and rapidly growing as the delay between event shrinks. From 3 ms, the slowdown is under 20 % and from 10 ms, the slowdown is under 5 %. We noticed 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 162 µs[16] on the same machine and around 300 µs on a slower machine (i3-4030U CPU @ 1.90 GHz). This time constitutes the main cause of the slowdown observed when evaluating properties. 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.

10.5 Memory Consumption

We assess the memory consumption of Verde according to the number of objects being tracked by the monitor in the interactively debugged program. We monitor a program that creates a number of queues given in parameter. We measure the peak memory usage in terms of approximate number of bytes taken by a global object containing the monitor using method asizeof of library Pympler, a first time before creating the queues in the program and a second time after their creation. We deduce the memory used by Verde to keep track of these queues by computing the difference between these two measures. Results are shown in Fig. 28 (axes use a logarithmic scale). The number of queues is represented on

[16] Previously, we measured 95 µs on the same machine running Ubuntu 14.04 and GDB 7.7. Running Ubuntu 16.04 and GDB 7.11 with disabled support for Kernel Page-Table Isolation (KPTI), we measure 152 µs. Running Ubuntu 16.04 and GDB 7.7 with KPTI support, we measure 125 µs. Same versions without KPTI support, we measure 109 µs. We suspect a regression in both newer versions of GDB and in the Linux kernel with support for KPTI, a mechanism to defend against the Meltdown vulnerability reveled in January 2018.

the x-axis and the memory consumption in bytes is represented on the y-axis using log scale. The memory consumption is linear, with nearly 1.3 KB used per tracked queue, making it realistic to track several thousands of instances of an object.

10.6 User-Perceived Performance Impact

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 measured an overhead of less than 10 % for the GDB process. These results correspond to our measurements in Sec. 10.4: 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 an observable overhead for multimedia applications when the events occur at such a limited frequency.

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, the execution of these programs was still fast enough to be debugged in realistic conditions, with lags under the second. 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, as methods of this library are called multiple times during when handling these events. Perceived lags ranged from unnoticeable to several seconds in the worst cases. In Sec. 11, we present possible ways to mitigate these limitations.

10.7 Dynamic Instrumentation on a Stack

We measured the effects of the dynamic instrumentation on the performance. A program adds and removes, alternatively, the first 100 natural integers in a stack. We checked that the integer 42 is taken out of the stack after being added. A first version of this property leverage the dynamic instrumentation. With this version, the call to the remove function was watched only when the monitor knew that 42 is in the stack. A second version of the property made the monitor watch every event unconditionally. The execution was 2.2 times faster with the first version. While this experiment used artificial properties, it shows that dynamic instrumentation has a positive impact on the overhead in that it improves performance.

11 CONCLUSION AND FUTURE WORK

11.1 Conclusion

Interactive runtime verification combines runtime verification and interactive debugging to facilitate and combine bug discovery and bug understanding. Interactive runtime verification aims at taking the best of both approaches by seeing the program as a system that can be monitored to find bugs and, at the same time, as a system that can be debugged interactively to understand the bugs that were found. i-RV replaces a part of the tedious manual process of

setting breakpoints in traditional interactive debugging by automatically set breakpoints by the monitor from properties describing the (expected) behavior for the program.

The monitor and scenario guide debugging. Our approach to i-RV relies on formal models of the behavior of the program, the debugger, and the scenario.The models we introduce allow us to formally describe their composition, thus providing guarantees on the verdicts reported by monitors at runtime. Since interactively runtime verifying the behavior of a program does not modify the behavior of the initial program, any bug found as a violation of a property at runtime is an actual bug and no bug causing the violation of a property can be missed. The formal models are backed up with algorithmic descriptions aimed at easing implementations of i-RV. We also presented Verde, an implementation of this approach, and experiments to evaluate our approach. Our experiments showed that even though the property checker can slow down the execution of the program considerably when events are temporally close to each other, our approach remain relevant in numerous cases showed in our experiments, when properties do not require the production of a high number of events per time unit (See Sec. 10.4). We demonstrated that interactive runtime verification is applicable in realistic use-cases with software such as video games and video players (Sec. 10.6). Our current implementation shows limitations in terms of performance under other use-cases. In the next section, we present ideas to mitigate this issue.

11.2 Future Work

We present some perspectives opened by this work.

Event Types. Our main event types are the function call and variable accesses. A way to make our approach more powerful is to find and include other kinds of events in our model. System calls are an example of event type we have not taken in account yet for technical reasons. They might be of interest for checking properties on drivers or programs dealing with hardware.

Instrumentation. Handling breakpoints is costly [5] and handling watchpoints even more so. Code injection could provide better efficiency [30, 32] by limiting round trips between the debugger and the program would to the bare minimum (for example, when the scenario requires the execution to be suspended to let the developer interact with the debugger) while keeping the current flexibility of the approach.

Checkpointing the File System. We plan to capture 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.

Record and Replay and Reverse Execution. RR is a powerful method 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.

Usability and Scalability. Our largest experiment involves a medium-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.

Another idea to be explored is verifying good programming practice and good API usage at runtime. We think that API designers and library writers could leverage our approach by providing properties with their APIs and their libraries. This would provide a means to check that their APIs are used correctly and make their usage safer. This would also be a means to document these APIs and these libraries.

Relaxing Assumptions on the Program. Our current assumptions on the program in our theoretical model are strong. We plan to improve the model by allowing multithreaded programs with side effects, possibilities of communicating with the outside and by accounting for the physical time.

Modular and Programming Language Independent Interface and Implementation for i-RV.. Our current implementation, Verde, is limited to the languages supported by the GNU Debugger and to our implementation of the monitor. We plan to design a standard interface for interactive runtime verification that is not tied to a particular programming language, runtime, debugger or monitor. This will let us leverage proven implementation of monitors supported by the runtime verification community. We plan to produce an implementation of this interface.

REFERENCES

[1] Ezio Bartocci and Yliès Falcone (Eds.). 2018. *Lectures on Runtime Verification - Introductory and Advanced Topics*. Lecture Notes in Computer Science, Vol. 10457. Springer. https://doi.org/10.1007/978-3-319-75632-5

[2] Mark Brörkens and Michael Möller. 2002. Dynamic Event Generation for Runtime Checking using the JDI. *Electr. Notes Theor. Comput. Sci.* 70, 4 (2002), 21–35. https://doi.org/10.1016/S1571-0661(04)80575-9

[3] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 213–223. https://doi.org/10.1109/CGO.2011.5764689

[4] Derek Bruening, Qin Zhao, and Saman P. Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)*, Steven Hand and Dilma Da Silva (Eds.). ACM, 133–144. https://doi.org/10.1145/2151024.2151043

[5] Martial Chabot, Kévin Mazet, and Laurence Pierre. 2015. 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*. 208–217. https://doi.org/10.1109/MEMCOD.2015.7340488

[6] Feng Chen and Grigore Rosu. 2007. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 569–588. https://doi.org/10.1145/1297027.1297069

[7] Feng Chen and Grigore Rosu. 2009. 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*. 246–261. https://doi.org/10.1007/978-3-642-00768-2_23

[8] Yoonsik Cheon and Gary T. Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*. 231–255. http://link.springer.de/link/service/series/0558/bibs/2374/23740231.htm

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model checking*. MIT Press. http://books.google.de/books?id=Nmc4wEaLXFEC

[10] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. 2011. Elarva: A Monitoring Tool for Erlang. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, Sarfraz Khurshid and Koushik Sen (Eds.), Vol. 7186. Springer, 370–374. https://doi.org/10.1007/978-3-642-29860-8_29

[11] Patrick Cousot and Radhia Cousot. 1977. 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*. 238–252. https://doi.org/10.1145/512950.512973

[12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings (Lecture Notes in Computer Science)*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.), Vol. 7504. Springer, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[13] Fabio Q. B. da Silva. 1992. *Correctness proofs of compilers and debuggers : an approach based on structural operational semantics*. Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/13542

[14] Mireille Ducassé and Erwan Jahier. 2001. Efficient Automated Trace Analysis: Examples with Morphine. *Electr. Notes Theor. Comput. Sci.* 55, 2 (2001), 118–133. https://doi.org/10.1016/S1571-0661(04)00248-8

[15] Mireille Ducassé. 1999. Opium: an extendable trace analyzer for Prolog. *The Journal of Logic Programming* 39, 1 (1999), 177 – 223. https://doi.org/10.1016/S0743-1066(98)10036-5

[16] E. Allen Emerson and Edmund M. Clarke. 1980. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings (Lecture Notes in Computer Science)*, J. W. de Bakker and Jan van Leeuwen (Eds.), Vol. 85. Springer, 169–181. https://doi.org/10.1007/3-540-10003-2_69

[17] Jakob Engblom. 2012. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*. IEEE, 1–6.

[18] Yliès Falcone. 2010. You Should Better Enforce Than Verify. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Lecture Notes in Computer Science)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.), Vol. 6418. Springer, 89–105. https://doi.org/10.1007/978-3-642-16612-9_9

[19] Kiril Georgiev and Vania Marangozova-Martin. 2014. 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*. 73–80. https://doi.org/10.1109/EUC.2014.20

[20] Klaus Havelund and Allen Goldberg. 2005. 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 (Lecture Notes in Computer Science)*, Bertrand Meyer and Jim Woodcock (Eds.), Vol. 4171. Springer, 374–383. https://doi.org/10.1007/978-3-540-69149-5_40

[21] Juha Itkonen, Mika V Mäntylä, and Casper Lassenius. 2007. Defect detection efficiency: Test case based vs. exploratory testing. In *First International Symposium on Empirical Software Engineering and Measurement*. IEEE, 61–70.

[22] Juha Itkonen, Mika V Mantyla, and Casper Lassenius. 2009. How do testers do it? An exploratory study on manual testing practices. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 494–497.

[23] Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, and Kevin Pouget. 2017. Interactive Runtime Verification - When Interactive Debugging Meets Runtime Verification. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*. IEEE Computer Society, 182–193. https://doi.org/10.1109/ISSRE.2017.19

[24] Raphaël Jakse, Yliès Falcone, and Jean-François Méhaut. 2017. Verde Repository. https://gitlab.inria.fr/monitoring/verde.

[25] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 1427–1430. https://doi.org/10.1109/ICSE.2012.6227231

[26] Shmuel Katz. 2006. Transactions on Aspect-Oriented Software Development I. In *Transactions on Aspect-Oriented Software Development I*, Awais Rashid and Mehmet Aksit (Eds.). Springer-Verlag, Berlin, Heidelberg, Chapter Aspect Categories and Classes of Temporal Properties, 106–134. http://dl.acm.org/citation.cfm?id=2168342.2168346

[27] Gregor Kiczales. 2002. 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 (Lecture Notes in Computer Science)*, Mehmet Aksit, Mira Mezini, and Rainer Unland (Eds.), Vol. 2591. Springer. https://doi.org/10.1007/3-540-36557-5_1

[28] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[29] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 190–200. https://doi.org/10.1145/1065010.1065034

[30] Reed Milewicz, Rajeshwar Vanka, James Tuck, Daniel Quinlan, and Peter Pirkelbauer. 2016. Lightweight runtime checking of C programs with RTC. *Computer Languages, Systems & Structures* 45 (2016), 191–203. https://doi.org/10.1016/j.cl.2016.01.001

[31] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

[32] Samaneh Navabpour, Yogi Joshi, Chun Wah Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2013. 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*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 603–606. https://doi.org/10.1145/2491411.2494596

[33] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. 89–100. https://doi.org/10.1145/1250734.1250746

[34] Fábio Petrillo, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla M. D. S. Freitas, and Yann-Gaël Guéhéneuc. 2016. Towards Understanding Interactive Debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*. IEEE, 152–163. https://doi.org/10.1109/QRS.2016.27

[35] Kevin Pouget. 2014. *Programming-Model Centric Debugging for multicore embedded systems / Debogage Interactif des systemes embarques multicoeur base sur le model de programmation*. Ph.D. Dissertation. University of Grenoble, France. https://tel.archives-ouvertes.fr/tel-01010061

[36] Norman Ramsey. 1994. Correctness of Trap-Based Breakpoint Implementations. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 15–24. https://doi.org/10.1145/174675.175188

[37] Jean-François Roos, Luc Courtrai, and Jean-François Méhaut. 1993. Execution replay of parallel programs. In *1993 Euromicro Workshop on Parallel and Distributed Processing, PDP 1993, Gran Canaria, Spain, 27-29 January 1993*. 429–434. https://doi.org/10.1109/EMPDP.1993.336375

[38] Oleg Sokolsky, Klaus Havelund, and Insup Lee. 2012. Introduction to the special section on runtime verification. *International Journal on Software Tools for Technology Transfer* 14, 3 (2012), 243–247. https://doi.org/10.1007/s10009-011-0218-6

A PROVING THAT THE DEBUGGED PROGRAM WEAKLY SIMULATES THE INITIAL PROGRAM, AND VICE VERSA

A.1 The Initial Program Weakly Simulates the Debugged Program

We prove Proposition 7.2 (Sec. 7). That is, the initial program P weakly simulates the debugged program (P, D).

Proof. Since the initial program cannot perform any unobservable action, proving that $\mathcal{R}$ is a simulation relation amounts to proving the two following points:

(1) $\forall((P_{\text{dbg}}, D), P) \in \mathcal{R}, \forall\theta \in \overline{Obs}, \forall(P'_{\text{dbg}}, D') \in \text{Conf}_\text{P} \times \text{Conf}_\text{D} : (P_{\text{dbg}}, D) \xrightarrow{\theta} (P'_{\text{dbg}}, D') \implies ((P'_{\text{dbg}}, D'), P) \in \mathcal{R}$.

(2) $\forall((P_{\text{dbg}}, D), P) \in \mathcal{R}, \forall\alpha \in Obs, \forall(P'_{\text{dbg}}, D') \in \text{Conf}_\text{P} \times \text{Conf}_\text{D} : (P_{\text{dbg}}, D) \xrightarrow{\alpha} (P'_{\text{dbg}}, D') \implies \exists P' \in \text{Conf}_\text{P} : ((P'_{\text{dbg}}, D'), P') \in \mathcal{R} \wedge P \xrightarrow{\alpha} P'$.

Let us consider $((P_{\text{dbg}}, D), P) \in \mathcal{R}$. Let us prove these two points in turn.

Proof of 1. Let us consider $\theta \in \overline{\text{Obs}}$ and $(P'_{\text{dbg}}, D') \in \text{Conf}_\text{P} \times \text{Conf}_\text{D}$ such that $(P_{\text{dbg}}, D) \xrightarrow{\theta} (P'_{\text{dbg}}, D')$. Action θ is triggered by applying either rule setWatch, rmWatch, getPC, getSym, getAddr, devBreak, scnBreak, evtBreak, int, cont, setBreak, rmBreak, devWatch, scnWatch, evtWatch, trapNoBreak, clearEvents, instrument, or stepRedo. We shall prove that $((P'_{\text{dbg}}, D'), P) \in \mathcal{R}$.

- Case: rule setWatch, rmWatch, getPC, getSym, getAddr, devBreak, scnBreak, evtBreak, int, or cont applies. None of these rules modifies neither the configuration of the program nor fields *bpts* and *oi* of the debugger (see Fig. 12, Fig. 15, Fig. 14) for the definitions of these rules). Since $((P_{\text{dbg}}, D), P) \in \mathcal{R}$, relation $\mathcal{R}$ is preserved, and we deduce immediately $((P'_{\text{dbg}}, D'), P) \in \mathcal{R}$.
- Case: rule setBreak (Fig. 12) applies for some $b \in D.\text{bpts}$. Let us prove that $((P'_{\text{dbg}}, D'), P) \in \mathcal{R}$.
  Let us prove (1). Let us define $m_{\text{unInstr}} = P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\}$, the debugger memory $P'_{\text{dbg}}.\text{m}$ where breakpoints have been removed. We distinguish two cases according to whether there is another breakpoint recorded at the same address as the address of breakpoint b in the debugger, that is whether $\exists b' \in D.\text{bpts} : b'.\text{addr} = b.\text{addr}$, or not.
  - Case $\exists b' \in D.\text{bpts} : b'.\text{addr} = b.\text{addr}$.
    According to rule setBreak, $oi' = oi$. Moreover, we have:

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\} \\
&= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\} \\
&\quad \text{(because } P_{\text{dbg}}.\text{m}[b.\text{addr}] = \text{BREAK and thus } P'_{\text{dbg}}.\text{m} = P_{\text{dbg}}.\text{m} \\
&\qquad \text{since there is already a breakpoint at } b.\text{addr)} \\
&= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts} \cup \{b\}\} \\
&\quad \text{(because } D'.\text{oi} = oi' = oi = D.\text{oi)} \\
&= P_{\text{dbg}}.\text{m} \dagger \{a \mapsto D.\text{oi}(a) \mid a \in \{b'.\text{addr} \mid b' \in D.\text{bpts}\} \cup \{b.\text{addr}\}\} \\
&= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts}\} \\
&\quad \text{(because } \exists b' \in D.\text{bpts} : b'.\text{addr} = b.\text{addr)} \\
&= P.\text{m} \quad \text{(because of (1))}
\end{aligned}
$$

  - Second case: $\nexists b' \in D.\text{bpts} : b'.\text{addr} = b.\text{addr}$.

According to rule setBreak, we have: $oi' = oi\left[b.\text{addr} \mapsto P_{\text{dbg}}.\text{m}[b.\text{addr}]\right]$ and $P'_{\text{dbg}}.\text{m} = m' = m\,[b.\text{addr} \mapsto \text{BREAK}]$. We have:

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\}\\
m_{\text{unInstr}} &= P_{\text{dbg}}.\text{m}\,[b.\text{addr} \mapsto \text{BREAK}]\\
&\qquad \dagger \left\{b'.\text{addr} \mapsto D.\text{oi}\left[b.\text{addr} \mapsto P_{\text{dbg}}.\text{m}[b.\text{addr}]\right](b'.\text{addr}) \mid b' \in D.\text{bpts} \cup \{b\}\right\}\\
&= P_{\text{dbg}}.\text{m}\,[b.\text{addr} \mapsto \text{BREAK}]\left[b.\text{addr} \mapsto P_{\text{dbg}}.\text{m}[b.\text{addr}]\right] \dagger \underbrace{\{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts}\}}\\
&\qquad (b.\text{addr is not in the domain of this function because } \nexists b' \in D.\text{bpts} : b'.\text{addr} = b.\text{addr})\\
&= P_{\text{dbg}}.\text{m}\left[b.\text{addr} \mapsto P_{\text{dbg}}.\text{m}[b.\text{addr}]\right] \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts}\}\\
&= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts}\}\\
&= P.\text{m} \quad \text{(because of (1))}
\end{aligned}
$$

In both cases, (1) holds.

Since the program counter has not been modified, (2) holds. Finally, $P'_{\text{dbg}}.\text{m} = P_{\text{dbg}}.\text{m}[b.\text{addr} \mapsto \text{BREAK}]$. Therefore, (3) holds.

- Case: rmBreak applies (see Fig. 12) for some input symbol rmPoint(b). Let us suppose that $b \in D.\text{bpts}$.

  Let us prove (1). Let us define $m_{\text{unInstr}} = P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\}$.

  We distinguish two cases according to whether $\exists b' \in D.\text{bpts} \setminus \{b\} : b'.\text{addr} = b.\text{addr}$.

  – First case: $\exists b' \in D.\text{bpts} \setminus \{b\} : b'.\text{addr} = b.\text{addr}$. We have:

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\}\\
m_{\text{unInstr}} &= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts} \setminus \{b\}\}\\
&\qquad (\text{because } D'.\text{oi} = oi' = oi = D.\text{oi},\ P'_{\text{dbg}}.\text{m} = P_{\text{dbg}}.\text{m} \text{ and } D'.\text{bpts} = D.\text{bpts} \setminus \{b\})\\
&= P_{\text{dbg}}.\text{m} \dagger \underbrace{\{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts}\}}_{b.\text{addr is still in the domain because } \exists b' \in D.\text{bpts} \setminus \{b\} : b'.\text{addr} = b.\text{addr}}\\
&= P.\text{m} \quad \text{(because of (1))}
\end{aligned}
$$

  – Second case: $\nexists b' \in D.\text{bpts} \setminus \{b\} : b'.\text{addr} = b.\text{addr}$. We have:

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.\text{oi}(b'.\text{addr}) \mid b' \in D'.\text{bpts}\}\\
m_{\text{unInstr}} &= P_{\text{dbg}}.\text{m}\,[b.\text{addr} \mapsto D.\text{oi}(b.\text{addr})] \dagger \{b'.\text{addr} \mapsto D.\text{oi}\,[b.\text{addr} \mapsto \text{BREAK}](b'.\text{addr}) \mid b' \in D.\text{bpts} \setminus \{b\}\}\\
&\qquad (\text{because in the rule, } oi' = oi\,[b.\text{addr} \mapsto \text{BREAK}] \text{ and } m' = m\,[b.\text{addr} \mapsto oi(b.\text{addr})])\\
&= P_{\text{dbg}}.\text{m}\,[b.\text{addr} \mapsto D.\text{oi}(b.\text{addr})] \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts} \setminus \{b\}\}\\
&\qquad (\text{because there is no } b' \text{ in } D.\text{bpts} \setminus \{b\} \text{ such that } b'.\text{addr} = b.\text{addr})\\
&= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in (D.\text{bpts} \setminus \{b\}) \cup \{b\}\} \qquad \text{(simplification)}\\
&= P_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D.\text{oi}(b'.\text{addr}) \mid b' \in D.\text{bpts}\} \qquad (\text{because } b \in D.\text{bpts})\\
&= P.\text{m} \quad \text{(because of (1))}
\end{aligned}
$$

  In both cases, (1) holds.

  Seeing that (2) holds follows directly from the definition of rmBreak (see Fig. 12).

  Lets us prove that (3) holds. Let us consider $a \in \text{Addr}$ such that $a \neq b.\text{addr}$. We have: $P'_{\text{dbg}}.\text{m}(a) = P_{\text{dbg}}.\text{m}\,[b.\text{addr} \mapsto oi(b.\text{addr})]\,(a) = P_{\text{dbg}}.\text{m}(a)$, and:

$$
\begin{aligned}
P.\text{m}(b.\text{addr}) &= \left(P'_{\text{dbg}}.\text{m} \dagger \{b'.\text{addr} \mapsto D'.oi(b'.\text{addr}) \mid b' \in D'.\text{bpts}\}\right)(b.\text{addr})\\
&= (P'_{\text{dbg}}.\text{m} \dagger \{a \mapsto D'.oi(a) \mid \underbrace{\exists b' \in D'.\text{bpts} : a = b'.\text{addr}}_{\nexists b' \in D'.\text{bpts}: a = b.\text{addr}}\})(b.\text{addr})\\
&= P'_{\text{dbg}}.\text{m}(b.\text{addr})
\end{aligned}
$$

Therefore, (3) holds.

- Case: devWatch, scnWatch, or evtWatch applies (see Fig. 8).
  In any of the rules, $oi' = oi\left[P_{\text{dbg}}.\text{pc} \mapsto P_{\text{dbg}}.\text{m}[\text{P}_{\text{dbg}}.\text{pc}]\right]$ and $m' = m[P_{\text{dbg}}.\text{pc} \mapsto \text{BREAK}]$ (see function restoreBP).
  Let us prove (1). We distinguish two cases according to whether $\exists b \in D.\text{bpts} : b.\text{addr} = pc$, or not.
  - First case: $\exists b \in D.\text{bpts} : b.\text{addr} = pc$.
    Let us define $a = \text{getAccesses}(P_{\text{dbg}})$ and $w \in D.\text{wpts} \backslash D.\text{hdld}$ such that $\exists k \in \{0, \ldots, |a|-1\} : \text{match}(a_k, w) \wedge w.\text{for} = \text{dev}$, This is possible according to the conditions of the rule. This implies $P_{\text{dbg}}.\text{m}[\text{P}_{\text{dbg}}.\text{pc}] \neq \text{BREAK}$.
    Let us define $m_{\text{unInstr}} = P'_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D'.oi(b.\text{addr}) \mid b \in D.\text{bpts}\}$.

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D'.oi(b.\text{addr}) \mid b \in D.\text{bpts}\} \\
&= P_{\text{dbg}}.\text{m}[P_{\text{dbg}}.\text{pc} \mapsto \text{BREAK}] \dagger \underbrace{\{b.\text{addr} \mapsto D.\text{oi}\left[P_{\text{dbg}}.\text{pc} \mapsto P_{\text{dbg}}.\text{m}[\text{P}_{\text{dbg}}.\text{pc}]\right](b.\text{addr}) \mid b \in D.\text{bpts}\}}_{P_{\text{dbg}}.\text{pc} \text{ is in the domain of this function}} \\
&\quad \text{(because } oi' = oi\left[P_{\text{dbg}}.\text{pc} \mapsto P_{\text{dbg}}.\text{m}[\text{P}_{\text{dbg}}.\text{pc}]\right] \text{ and } m' = m[P_{\text{dbg}}.\text{pc} \mapsto \text{BREAK}]) \\
&= P_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D.\text{oi}\left[P_{\text{dbg}}.\text{pc} \mapsto P_{\text{dbg}}.\text{m}[\text{P}_{\text{dbg}}.\text{pc}]\right](b.\text{addr}) \mid b \in D.\text{bpts}\} \\
&\quad \text{(simplification)} \\
&= P_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D.\text{oi}(b.\text{addr}) \mid b \in D.\text{bpts}\}\left[P_{\text{dbg}}.\text{pc} \mapsto P_{\text{dbg}}.\text{m}[P_{\text{dbg}}.\text{pc}]\right] \\
&\quad \text{(because } b \in D.\text{bpts, so the substitution can be done outside)} \\
&= P.\text{m}\left[P_{\text{dbg}}.\text{pc} \mapsto P_{\text{dbg}}.\text{m}[P_{\text{dbg}}.\text{pc}]\right] \quad \text{(because of (1))} \\
&= P.\text{m}\left[P.\text{pc} \mapsto P.\text{m}[P.\text{pc}]\right] \quad \text{(because } P_{\text{dbg}}.\text{pc} = P.\text{pc} \text{ and } P_{\text{dbg}}.\text{m}[\text{P}_{\text{dbg}}.\text{pc}] \neq \text{BREAK}) \\
&= P.\text{m}
\end{aligned}
$$

  - Second case: $\nexists b \in D.\text{bpts} : b.\text{addr} = pc$.
    In the rule, $oi' = oi$ and $m' = m$. In P'_{dbg}, $D = (P', D', M, S)$, $D'.\text{oi} = D.\text{oi}$, $D'.\text{bpts} = D'.\text{bpts}$, $P'.\text{m} = P.\text{m}$ and $P'.\text{pc} = P.\text{pc}$. Since $(P, P_{\text{dbg}, D}) \in \mathcal{R}$, $(P, P'_{\text{dbg}}, D) \in \mathcal{R}$.
  (2) and (3) hold because the program counter and the memory are not modified by these rules.
- Case: trapNoBreak apply (see Fig. 9).
  This rule applying means that rule bpHit of the program applies and so $P_{\text{dbg}}.\text{m}\left[P_{\text{dbg}}.\text{pc}\right] = \text{BREAK}$.
  Let us prove (1). Let us define $m_{\text{unInstr}} = P'_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D'.oi(b.\text{addr}) \mid b \in D.\text{bpts}\}$.

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D'.oi(b.\text{addr}) \mid b \in D.\text{bpts}\} \\
&= P_{\text{dbg}}.\text{m}\left[P_{\text{dbg}}.\text{pc} \mapsto D.\text{oi}\left(P_{\text{dbg}}.\text{pc}\right)\right] \dagger \underbrace{\{b.\text{addr} \mapsto D.\text{oi}(b.\text{addr}) \mid b \in D.\text{bpts}\}}_{P_{\text{dbg}}.\text{pc} \text{ is in the domain of this function}} \\
&\quad \text{(because in the rule, } P'_{\text{dbg}}.\text{m} = P_{\text{dbg}}.\text{m}[pc \mapsto D.\text{oi}(pc)]) \\
&= P_{\text{dbg}}.\text{m} \dagger \{b.\text{addr} \mapsto D.\text{oi}(b.\text{addr}) \mid b \in D.\text{bpts}\} \\
&\quad \text{(simplification: this substitution has no effect)} \\
&= P.\text{m} \quad \text{(because of (1))}
\end{aligned}
$$

- Case: clearEvents applies (see Fig. 11).
  This rule uses rule rmBreak and rule rmWatch sequentially for each point given by the input symbol. These rules are proven to respect the conditions of the weak simulation applying to unobservable actions. Proving (1), (2) and (3) is therefore straightforward by induction.

- Case: INSTRUMENT applies (see Fig. 11).
  This rule uses rule CLEAREVENTS and then rule SETBREAK and rule SETWATCH. These rules are proven to respect the conditions of the weak simulation applying to unobservable actions. Proving (1), (2) and (3) is therefore straightforward.
- Case: STEPREDO applies (see Fig. 14).
  This rule temporarily sets the debugger in passive mode and uses rule EVTBREAK, DEVBREAK, SCNBREAK, EVTWATCH, DEVWATCH, SCNWATCH or TRAPNOBREAK. Proving (1), (2) and (3) is therefore straightforward.

Proof of 2. Let us consider $\alpha \in \text{Obs}$. Let us consider (P'_{dbg}, D') such that $(P_{\text{dbg}}, D) \xrightarrow{\alpha} (P'_{\text{dbg}}, D')$. Let us consider P' such that $P \xrightarrow{\alpha} P'$. Let us prove that $((P'_{\text{dbg}}, D'), P') \in \mathcal{R}$. To produce action a, either rule STEP applies, or rule NORMALEXEC applies. rule STEP uses rule NORMALEXEC, and does nothing else than temporarily switching the debugger mode to passive so rule NORMALEXEC applies. Therefore, we prove the case where rule NORMALEXEC applies. Proof for the case where rule STEP applies is then straightforward.

Let us prove (1). We have $(P'.\text{m}, P'.\text{pc}) = \text{runInstr}(P.\text{m}, P.\text{pc})$ (see rule NORMALEXEC).

Since $((P_{\text{dbg}}, D), P) \in \mathcal{R}$, we have $P.\text{m} = \text{unInstr}(P_{\text{dbg}}.\text{m},\ D.\text{bpts},\ D.\text{oi})$ (because of (1)) and $P.pc = P_{\text{dbg}}.pc$ (because of (2)).

We have:

$$
\begin{aligned}
(D'.\text{oi}, P'_{\text{dbg}}.\text{m}) &= \text{restoreBP}(D'.\text{bpts}, m_t) \\
&\qquad \text{such that } \exists pc' : (m_t, pc') = \text{runInstr}(\text{unInstr}(P_{\text{dbg}}.\text{m}, D.\text{bpts}, D.\text{oi}), P_{\text{dbg}}.pc) \\
&\quad \text{(according to the rule)} \\
&= \text{restoreBP}(D'.\text{bpts}, m_t) \text{ such that } \exists pc' : (m_t, pc') = \text{runInstr}(P.\text{m}, P_{\text{dbg}}.pc) \\
&\quad \text{(because of (1))} \\
&= \text{restoreBP}(D'.\text{bpts}, m_t) \text{ such that } \exists pc' : (m_t, pc') = \text{runInstr}(P.\text{m}, P.pc) \\
&\quad \text{(because of (2))} \\
&= \text{restoreBP}(D'.\text{bpts}, m_t) \text{ such that } \exists pc' : (m_t, pc') = (P'.\text{m}, P'.pc) \\
&\quad \text{(because of the program's rule NORMALEXEC)} \\
&= \text{restoreBP}(D'.\text{bpts}, P'.\text{m})
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
\text{unInstr}(P'_{\text{dbg}}.\text{m}, D'.\text{bpts}, D'.\text{oi}) &= \text{unInstr}(m', D'.\text{bpts}, D'.\text{oi}) \\
&\qquad \text{such that } (D'.\text{oi}, m') = \text{restoreBP}(D'.\text{bpts}, P'.\text{m}) \\
&= P'.\text{m} \qquad \text{(by definition of unInstr and restoreBP)}
\end{aligned}
$$

The same action α is performed in the program and the debugged program. This proves (1). Let us prove (2). $P'_{\text{dbg}}.pc$ is such that there exists m_t such that:

$$
\begin{aligned}
(m_t, P'_{\text{dbg}}.pc) &= \text{runInstr}(\text{unInstr}(P_{\text{dbg}}.\text{m}, D.\text{bpts}, D.\text{oi}), P_{\text{dbg}}.pc) \\
&= (m_t, P'.pc) \quad \text{(as proved previously)}
\end{aligned}
$$

Therefore, $P'_{\text{dbg}}.pc = P'.pc$. (2) is proved. Proving (3) is straightforward.

□

A.2 The Debugged Program Weakly Simulates the Initial Program

We now prove Proposition 7.2 (Sec. 7). That is, the debugged program (P, D) weakly simulates the initial program P.

Proof. We prove that relation satisfies the two points of the definition of weak simulation given the set of observable actions *Obs*. Since the initial program does not have any unobservable action, proving that $\mathfrak{R}$ is a weak simulation amounts to proving:

$$\forall (P, (P_{\text{dbg}}, D)) \in \mathfrak{R}, \forall \alpha \in Obs, \forall P' \in \text{Conf}_\text{P} :$$
$$P \xrightarrow{\alpha} P' \implies \exists (P'_{\text{dbg}}, D') \in \text{Conf}_\text{P} \times \text{Conf}_\text{D} : (P_{\text{dbg}}, D) \xrightarrow{\overline{Obs}^* \cdot \alpha \cdot \overline{Obs}^*} (P'_{\text{dbg}}, D') \wedge (P', (P'_{\text{dbg}}, D')) \in \mathfrak{R}.$$

Let us consider P a configuration of a program and (P_{dbg}, D) a configuration of the debugged program such that $(P, (P_{\text{dbg}}, D)) \in \mathfrak{R}$. By definition of $\mathfrak{R}$, $((P_{\text{dbg}}, D), P) \in \mathcal{R}$. Let us consider $\alpha \in Obs$. Let us consider $P' \in \text{Conf}_\text{P}$ such that $P \xrightarrow{\alpha} P'$. Let us show the existence of $(P'_{\text{dbg}}, D') \in \text{Conf}_\text{P} \times \text{Conf}_\text{D}$ such that $(P_{\text{dbg}}, D) \xrightarrow{\overline{Obs}^* \cdot \alpha \cdot \overline{Obs}^*} (P'_{\text{dbg}}, D')$ and $((P'_{\text{dbg}}, D'), P') \in \mathcal{R}$.

Since action α is observable, it can be only triggered by applying rule normalExec in the program P (see Fig. 7). We consider configuration of the debugged program (P_{dbg}, D) wherein the debugger can be in two modes (either passive or active). We handle each two cases in turn.

- Case: the debugger is in passive mode. We shall prove that rule normalExec applies, triggering action α, possibly after several applications of rules associated to unobservable actions.
  Either rule normalExec applies immediately or not. Let us consider both cases.
  - Rule normalExec applies. In this case, the proof is similar to the one for the simulation of the debugged program by the initial program (as the same equalities apply). Resulting configuration (P'_{dbg}, D') is such that $((P'_{\text{dbg}}, D'), P') \in \mathcal{R}$ and therefore $(P', (P'_{\text{dbg}}, D')) \in \mathcal{R}$.
  - Rule normalExec does not apply. Then, the debugger has at least one point to handle, and necessarily, either rule devWatch, scnWatch, evtWatch, devBreak, scnBreak, evtBreak or trapNoBreak applies. Let us consider θ the action associated to this rule. Let us consider $(P_{\text{dbg},i}, D_i)$ such that $(P_{\text{dbg}}, D) \xrightarrow{\theta} (P_i, D_i)$. Since θ is unobservable and $((P_{\text{dbg}}, D), P) \in \mathcal{R}$, $((P_i, D_i), P) \in \mathcal{R}$ (see proof for the simulation of the debugged program by the initial program). Showing that successive applications of these rules starting from configuration $(P_{\text{dbg},i}, D_i)$ results in a configuration (P_j, D_j) such that $((P_j, D_j), P) \in \mathcal{R}$ is straightforward by induction on the sequence of actions associated to these rules.
    While there exists a watchpoint in $\mathscr{W}$ matching an access done by the current instruction, or a breakpoint in $\mathscr{B}$ matching the current address that is not in *hdld*, one of these rules except trapNoBreak applies and adds this breakpoint or watchpoint in *hdld*. Since there is a finite number of points, these rules stop applying and rule trapNoBreak applies at most once (if a breakpoint is present at the current address in the program).
    Let us call u the finite sequence of unobservable actions triggered by applying these rules sequentially. Let us consider (P_j, D_j) the configuration such $(P_{\text{dbg}}, D) \xrightarrow{u} (P_j, D_j)$. Configuration (P'_{dbg}, D') of the debugged program such that $(P_j, D_j) \xrightarrow{\alpha} (P'_{\text{dbg}}, D')$ exists and rule normalExec applies since $(P, (P_j, D_j)) \in \mathcal{R}$ (see proof for the simulation of the debugged program by the initial program). Therefore, $((P_j, D_j), P) \in \mathfrak{R}$. See previous case.
- Case: the debugger is in interactive mode. rule step either applies, or do not apply. In case rule step does not apply, rule stepRedo applies a certain number of times. In any case, both rules temporarily set the debugged

program in passive mode and call rules that apply in passive mode. Therefore, proving this case is similar to the previous case.

□

B COMPARING A TRADITIONNAL INTERACTIVE DEBUGGING AND AN INTERACTIVE RUNTIME VERIFICATION SESSION

In Fig. 29 and Fig. 30, we reproduce an interactive debugging session and an interactive runtime verification session for the illustrative example presented in Sec. 2 (p. 6).

C PROPERTY ON VALUE CHANGES OF STRING S IN FUNCTION GET_COMP_STRING IN ZSH

In this appendix, we present a property written in Verde property format. This property is used in the experiment on zsh in Sec. 10.1. We use the property to find the cause of a segfault in zsh; see Fig. 31. In this property, we are in an accepting state while the state of zsh seems consistent, that is, no null pointer is used. In state init, we track a call to function get_comp_string. When the call happens, the state becomes in_get_cmp_str_init. In this state, several things can happen. Destination states from state in_get_cmp_str_init correspond to the different continuations we imagined possible after this state by quickly looking at the code. We did not aim at exactly understanding the meaning of these different possibilities. Rather, we aimed at seeking where the pointer was nulled in the code.

```
$ ./faulty
Consonants: hnstbgsh
Vowels: raayuiee
$ gdb ./faulty
[...] Reading symbols from ./faulty...done.
(gdb) start
Temporary breakpoint 1 at 0x1372: file faulty.c, line 51.
Starting program: /tmp/a

Temporary breakpoint 1, main () at faulty.c:51
51 double_queue_t* q = queue_init();
(gdb) break 53
Breakpoint 2 at 0x5...55393: file faulty.c, line 53.
(gdb) cont
Continuing.

Breakpoint 2, main () at faulty.c:53
53 queue_display_result(q);
(gdb) print q->vowels
$1 = "raayuiee"
(gdb) quit
[...]
$ gdb ./faulty
Reading symbols from ./faulty...done.
(gdb) start
Temporary breakpoint 1 at 0x1372: file faulty.c, line 51.
Starting program: /tmp/a

Temporary breakpoint 1, main () at faulty.c:51
51 double_queue_t* q = queue_init();
(gdb) break 30
Breakpoint 2 at 0x5...55233: file faulty.c, line 30.
(gdb) cont
Continuing.

Breakpoint 2, queue_push (q=0x5...59260, c=111 'o') at faulty.c:30
30 q->vowels[q->pos_v++] = c;
(gdb) display c
1: c = 111 'o'
(gdb) c
Continuing.
[...]
```

```
Breakpoint 2, queue_push (q=0x5...59260, c=101 'e') at faulty.c:30
30 q->vowels[q->pos_v++] = c;
1: c = 101 'e'
(gdb) c
Continuing.
Consonants: hnstbgsh
Vowels: raayuiee
[Inferior 1 (process 6646) exited normally]
(gdb) info break 2
Num Type Disp Enb Address What
2 breakpoint keep y 0x00005...55233 in queue_push at faulty.c:30
        breakpoint already hit 8 times
(gdb) quit
$ gdb ./faulty
[...] Reading symbols from ./faulty...done.
(gdb) break 52
Breakpoint 1 at 0x1380: file faulty.c, line 52.
(gdb) start
Temporary breakpoint 2 at 0x1372: file faulty.c, line 51.
Starting program: /tmp/a

Temporary breakpoint 2, main () at faulty.c:51
51 double_queue_t* q = queue_init();
(gdb) cont
Continuing.

Breakpoint 1, main () at faulty.c:52
52 queue_push_str(q, "oh, a nasty bug is here!");
(gdb) watch q->vowels[0]
Hardware watchpoint 3: q->vowels[0]
(gdb) cont
Continuing.

Hardware watchpoint 3: q->vowels[0]

Old value = 0 '\000'
New value = 111 'o'
queue_push (q=0x5...59260, c=111 'o') at faulty.c:33
33 }
(gdb) next
queue_push_str ... at faulty.c:36
36 while (s[0] != '\0')
(gdb) cont
Continuing.

Hardware watchpoint 3: q->vowels[0]

Old value = 111 'o'
New value = 114 'r'
queue_push (q=0x5...59260, c=114 'r') at faulty.c:33
33 }
(gdb) print q->pos_c - 1
$1 = 8
(gdb) quit
```

Fig. 29. Interactive debugging of the faulty C program.

```
$ verde --prop queue.prop --show-graph ./faulty
[...]
[23:09:36] Initialization:
               N: 0
               max: 0

Event: queue_new{'size': 16, 'queue': 93824992252512}
16
[23:09:36] Current state (monitor #1):
   Slice 1 <None>: init (from init)
      N: 0
      max: 16
   Slice 2 <93824992252512>: queue_ready
      N: 0
      max: 16

Event: queue_push{'queue': 93824992252512}
GUARD: nb push: 0 (max: 16)
[23:09:36] Current state (monitor #1):
   Slice 1 <None>: init
      N: 0
      max: 16
   Slice 2 <93824992252512>: queue_ready (from queue_ready)
      N: 1
      max: 16

Event: queue_push{'queue': 93824992252512}
GUARD: nb push: 1 (max: 16)
[23:09:36] Current state (monitor #1):
   Slice 1 <None>: init
      N: 0
      max: 16
   Slice 2 <93824992252512>: queue_ready (from queue_ready)
      N: 2
      max: 16

Event: queue_push{'queue': 93824992252512}
GUARD: nb push: 15 (max: 16)
[23:09:36] Current state (monitor #1):
   Slice 1 <None>: init
      N: 0
      max: 16
   Slice 2 <93824992252512>: queue_ready (from queue_ready)
      N: 16
      max: 16

Event: queue_push{'queue': 93824992252512}
GUARD: nb push: 16 (max: 16)
[23:09:36] Current state (monitor #1):
   Slice 1 <None>: init
      N: 0
      max: 16
   Slice 2 <93824992252512>: sink (from queue_ready) non-accepting
      N: 16
      max: 16
(gdb) where
#0 queue_push (queue=0x5...59260, c=105 'i') at faulty.c:29
#1 0x00005...552a1 in queue_push_str [...] at faulty.c:37
#2 0x00005...55377 in main () at faulty.c:52
(gdb) quit
```

Fig. 30. Interactively runtime verifying the faulty C program. Compared to the interactive debugging session depicted in Fig. 29, almost all interactions between the debugger and the developer are avoided.

```
initialization {
    import gdb
}

state init accepting {
    transition {
        before event get_comp_string()
        success in_get_cmp_str_init
    }
}

state in_get_cmp_str_init accepting {
    transition {
        after event write s(s) {
            return not s
        }

        success {
            print("s = " + str(s))
        } in_get_cmp_str_init_s_not_null

        failure {
            gdb.execute("backtrace")
        } in_get_cmp_str_init_s_null
    }

    transition {
        before event itype_end(ptr) { return not ptr }
        success calling_itype_end_with_null_ptr
    }

    transition {
        before event get_comp_string()
        success in_get_cmp_str_init
    }
}

state in_get_cmp_str_init_s_null accepting {
    transition {
        after event write s(s) {
            return not s
        }

        success {
            print("s = " + str(s))
        } in_get_cmp_str_init_s_not_null

        failure {
            gdb.execute("backtrace")
        } in_get_cmp_str_init_s_null
    }

    transition {
        before event itype_end(ptr) {
            return not ptr
        }

        success calling_itype_end_with_null_ptr

        failure {
            print("called itype_end with non null");
            gdb.execute("backtrace")
        } in_get_cmp_str_init_s_null
    }

    transition {
        before event get_comp_string()
        success in_get_cmp_str_init
    }
}

state in_get_cmp_str_init_s_not_null accepting {
    transition {
        after event write s(s) {
            return not s
        }

        success {
            print("s = " + str(s))
        } in_get_cmp_str_init_s_not_null

        failure {
            gdb.execute("backtrace")
        } in_get_cmp_str_init_s_null
    }

    transition {
        before event itype_end(ptr) { return not ptr }
        success calling_itype_end_with_null_ptr
    }

    transition {
        before event get_comp_string()
        success in_get_cmp_str_init
    }
}

state in_get_cmp_str_init_s_not_null accepting {
    transition {
        before event get_comp_string()
        success in_get_cmp_str_init
    }

    transition {
        after event write s(s) {
            return not s
        }

        success {
            print("s = " + str(s))
        } in_get_cmp_str_init_s_not_null

        failure {
            gdb.execute("backtrace")
        } in_get_cmp_str_init_s_null
    }
}

state calling_itype_end_with_null_ptr non-accepting
```

Fig. 31. Verde property to find the cause of the segfault in zsh