Vérification interactive de propriétés à l'exécution

Raphaël Jakse

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

**DOCTEUR DE LA
COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES**

Spécialité : informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Raphaël Jakse

Thèse dirigée par **Jean-François Méhaut** et encadrée par **Yliès Falcone**, Université Grenoble Alpes

préparée au sein du **Laboratoire d'Informatique de Grenoble**
à **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Interactive Runtime Verification

Vérification de propriétés interactive à l'exécution

Thèse soutenue publiquement le **18 Décembre 2019**,
devant le jury composé de :

Fabrice Dupros
Ingénieur de recherche, ARM, *examinateur*
Yliès Falcone
Maître de conférence, Université Grenoble Alpes, *encadrant de thèse*
João Lourenço
Professeur, Universidade Nova de Lisboa, *rapporteur*
Jean-François Méhaut
Professeur, Université Grenoble Alpes, *directeur de thèse*
Hervé Marchand
Charché de recherche, Inria Rennes Bretagne-Atlantique, *examinateur*
Gwen Salaün
Professeur, Inria Grenoble, Examinateur, *président du jury*
Alexandre Termier
Professeur, Inria Rennes Bretagne-Atlantique, *rapporteur*

Abstract

Résumé en Français

Les ordinateurs sont partout. Nous leur faisons confiance pour un grand, et grandissant, nombre de tâches, parmi lesquelles certaines sont critiques. Les conséquences des bogues logiciels sont diverses, de l'agacement léger à la mort de plusieurs personnes. Ainsi, il est important de s'assurer que les logiciels sont fiables.

Corriger les bogues est une activité très chronophage du processus de développement logiciel. Dans cette thèse, nous présentons la vérification interactive à l'exécution, qui combine la vérification à l'exécution et le débogage interactif. La vérification à l'exécution est une méthode formelle pour étudier le comportement d'un système à l'exécution. Elle consiste à faire correspondre des traces d'exécutions d'un système avec des propriétés comportementales. Ces propriétés font partie des spécifications du système. Le débogage interactif consiste à étudier un système durant son exécution pour comprendre ses bogues et les corriger en inspectant interactivement son état interne. La vérification interactive à l'exécution a pour objectif de rendre le débogage interactif moins fastidieux et plus systématique en s'appuyant sur les aspects automatiques et rigoureux de la vérification à l'exécution. Nous avons pour but de faciliter la partie débogage du processus de développement logiciel. Nous définissons une manière efficace et pratique de vérifier des propriétés comportementales automatiquement sur un programme en utilisant un débogueur interactif. Nous rassemblons la détection et la compréhension de bogues dans une méthodologie intégrée, en guidant le débogage interactif avec la vérification à l'exécution.

Nous fournissons un modèle formel pour les programmes vérifiés interactivement à l'exécution. Nous modélisons l'exécution d'un programme en train d'être débogué, composé avec un moniteur (pour l'émission de verdicts) et un scénario (pour conduire la session de débogage). Nous fournissons des garanties sur la validité des verdicts produits par le moniteur en nous appuyant sur une relation de simulation faible entre le programme initial et le programme vérifié interactivement. De plus, nous fournissons une vue algorithmique du modèle adaptée à l'écriture d'implémentations. Nous introduisons ensuite un cadre et une architecture distribuée pour la vérification interactive à l'exécution. Cela permet de vérifier plusieurs propriétés simultanément et de déboguer un système distribué, composé de multiples processus communicants. Les moniteurs, le scénario et les programmes débogués eux-mêmes s'exécutent de façon distribuée en utilisant un protocole que nous avons vérifié avec le vérificateur de modèles SPIN. Notre architecture distribuée est conçue pour s'adapter à des composants existants.

Nous présentons Verde, une implémentation de la vérification interactive à l'exécution. Une première version est basée sur le débogueur GNU (GDB) pour vérifier interactivement des programmes C et C++. Une deuxième version, Dist-Verde, est une implémentation de notre architecture distribuée compatible avec les programmes C et C++ à travers GDB et les programmes Java à travers JDB, le débogueur Java.

Nous présentons des expérimentations en utilisant Verde, évaluant l'utilité de l'approche et les performances de notre implémentation. Nos résultats montrent que la vérification interactive à l'exécution est applicable dans une variété de cas et aide à étudier les bogues.

Mots-clés : vérification à l'exécution, débogage interactif, fiabilité, ingénierie logicielle, systèmes distribués, propriétés, bogues

Abstract in English

Computers are ubiquitous. We trust them for a huge and increasing number of tasks, some critical. Consequences of software defects are various, from little annoyances to the loss of multiple lives. Hence, ensuring software reliability is instrumental.

Fixing bugs is a very time-consuming activity of the software development cycle. In this thesis, we present interactive runtime verification (i-RV), which combines runtime verification and interactive debugging. Runtime verification is a formal method to study the behavior of a system at runtime. It consists in matching runtime traces of a system at runtime against behavioral properties. These properties are part of the system specification. Interactive debugging consists in studying a system at runtime in order to understand its bugs and fix them, inspecting its internal state interactively. Interactive runtime verification aims to make interactive debugging less tedious and more systematic by leveraging the rigorous and automated aspects of runtime verification. We aim to ease the debugging part of the software development cycle. We define an efficient and convenient way to check behavioral properties automatically on a program using an interactive debugger. We gather bug detection and bug understanding in an integrated workflow, by guiding interactive debugging using runtime verification.

We provide a formal model for interactively runtime verified programs. We model the execution of a program 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 then introduce a distributed and adaptive framework for interactive runtime verification. It allows checking several requirements simultaneously and debugging a distributed system composed of communicating processes. The monitors, the scenario and the debugged programs themselves run distributed using an orchestrating protocol which we verified using the SPIN model checker. Our distributed framework is designed to adapt to existing components.

We present Verde, an implementation of interactive runtime verification. A first version is based on the GNU Debugger (GDB) to interactively runtime verify C and C++ programs. A second version, Dist-Verde, is an implementation of our distributed framework compatible with C and C++ programs through GDB and Java programs through JDB, the Java Debugger.

We report on experiments using Verde assessing the usefulness of interactive runtime verification and the performance of our implementation. Our results show that interactive runtime verification is applicable in a variety of cases and helps to study bugs.

Keywords: runtime verification, interactive debugging, reliability, software engineering, distributed systems, property, bugs.

Merci !

Je tiens à remercier Chloé, Maman et Papa, pour m'avoir accompagné, guidé et soutenu tout au long de ce doctorat, et bien avant. Merci pour ces weekends tranquilles et rassurants loin de l'agitation et de la polution citadine, ces apéros, ces moments studieux ensemble, ces randonnées agréables, ces petites marches ou ces tours de vélo autour de la maison. On vient quand même malgré la connexion moisie, parce qu'on y est bien. Merci aussi pour ces petits restos sur le campus.

Je voudrais remercier Mamie et Papi, qui ont toujours été confiants sur le bon déroulement de mes études, y compris ce doctorat. Merci aussi à toute la famille : Camille, Donavan, Émilie, Geneviève, Gérald, Laurent, Lisa, Marie-Noelle, Marion, Rémi, Séverine. Vous êtes géniaux. Merci Bruno, Céline et Ludo, plus loins mais là.

Je tiens aussi à remercier Solène, qui a décidé de bien vouloir me supporter au cours de la dernière année de ce doctorat. Une tâche ardue s'il en est. Merci pour toute cette joie que tu m'apportes.

Je voudrais remercier les colocs' que j'ai eu au cours de ce doctorat, ou juste avant : Doriane, Philippe et Cyril, qui ont rendu ma vie plus riche et plus agréable au quotidien, à leur manière. Merci Maxime et Sophia pour cette amitié forte et durable. Merci pour cette complicité.

Merci bien sûr à la chorale, les Rainbow Swingers, aussi bien au CA, à Nila, qu'aux choristes. Je suis rentré dans cette chorale parce que j'ai vu la lumière, et c'était l'une des meilleures décisions de ma vie. Vous êtes ma famille grenobloise ! Merci pour ces sourires, ces répétitions, ces soirées à EVE et ailleurs, et ces weekends et ces concerts magnifiques. Merci pour ces soirées et ces moments de complicité. Merci Anaïs, Andrea, Antoine, Ben, Charlotte, Chloé, Claire, Clémentine, David, Esther, Katline, Kevin, Ludo, Miriam, Marine, Max, Solène, Túng, Véronique. Merci Cécile, Cécile, Joanna, Lila, Lucie, Kevin, Solène. Merci à tous les choristes que je n'ai pas cités ici mais que je suis heureux de connaitre et de voir, « le cœur béat », à chaque répétition. Ce rendez-vous hebdomadaire a eu un rôle clé dans le déroulement de ce doctorat.

Merci aux potes de la fac : Jérémy, Rémy et Timothée, pour des moments de discussions intéressantes et amusantes. Merci à Sébastien pour avoir gardé contact et avec qui j'ai probablement le fil de conversation par courrier électronique le plus long.

Merci à Alexis et Myriam, là depuis mon entrée à la fac. Merci Julien, Marcelin, Typhaine et Alexandre. On s'est peu vus, mais bien. Merci à Thibaut pour cette belle amitié qui dure, et à Lise.

Merci aux Fillon, Hugon, Léry, Asensi, Mounier, à Gérard et Ida pour ces moment très agréables.

Merci aux collègues de l'équipe CORSE pour ces discussions intéressantes, ces repas de midi à la cantine ou à la cafet', ce soutien et aussi ces quelques bières : Antoine, Ali, Auguste, Byron, Diogo, Éric, Fabian, Fabrice, Farah, Francieli, Florent, François, Georgios, Kevin, Luís, Laércio, Manuel, Mathieu, Naweiluo, Nicolas, Nicolas, Patrick, Pedro, Théo, Thomas. Merci Fred et Emmanuelle pour ces courses à pied, c'était chouette ! Merci Imma d'être toujours à l'écoute et bienveillante.

Merci aussi aux personnes avec qui j'ai eu la joie d'interagir plus ou moins régulièrement dans le bâtiment IMAG, notamment en tant que squateur de bureau (merci pour l'accueil !).

Merci Yliès et JF pour votre confiance et votre aide, et de m'avoir poussé à faire mieux. M'encadrer ne devait pas être une mince affaire. Merci également aux membres du jury, pour avoir accepté de bien vouloir relire cette thèse et assister à la soutenance.

Merci à toutes les personnes que j'aurais honteusement oublié de mentionner ici, mais qui ont rendu ma vie meilleure par un sourire ou au détour d'une discussion enrichissante. Vous n'avez pas eu votre place dans cette page, mais vous l'avez dans mon cœur et cela est plus important.

Avant-Propos

L'informatique comme domaine de recherche

Pendant ce doctorat, plusieurs personnes m'ont demandé ce sur quoi je travaillais. Certaines étaient surprises que l'informatique est un domaine de recherche. D'après le Trésor de la Langue Française informatisé [TLF], l'informatique est la "science du traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social", et un ordinateur est une "machine algorithmique composée d'un assemblage de matériels correspondant à des fonctions spécifiques, capable de recevoir de l'information, dotée de mémoires à grande capacité et de moyens de traitement à grande vitesse, pouvant restituer tout ou partie des éléments traités, ayant la possibilité de résoudre des problèmes mathématiques et logiques complexes, et nécessitant pour son fonctionnement la mise en oeuvre et l'exploitation automatique d'un ensemble de programmes enregistrés". Un aspect important de les ordintateurs d'aujourd'hui, outre le traitement des données et les calculs, est la communication. Dans ce paragraphe, je vais donner au lecteur et à la lectrice non convaincu · e mon opinion sur pourquoi la recherche en informatique est importante.

Les enjeux environnementaux. Notre impact en tant qu'espèce sur l'environnement est devenu une préoccupation importante et les chercheurs et chercheuses en informatique ont leur rôle à prendre. Des travaux ont été menés pour étudier et minimiser la consommation énergétique et réduire les émission carbonnes dans les bâtiments [AZ19], processus de fabrication [JZ19], moteurs à induction [SKN19]. Les systèmes informatiques eux-mêmes consomment beaucoup d'énergie. Il est estimé que faire fonctionner internet demande entre 1,1% and 1,9% [RM11] de l'énergie utilisée globalement par l'humanité. Une première solution évidente à ce problème est d'éviter d'utiliser les ordinateurs quand cela n'est pas nécessaire. Cependant, cela ne doit pas nous décourager d'étudier et améliorer la consommation d'énergie des systèmes informatiques. Réduire notre dépendance aux ordinateurs n'est pas toujours la solution évidente pour réduire notre impact écologique. Ils peuvent remplacer des solutions coûteuses énergétiquement, comme voler, avec des solutions comme la visioconférence. Des travaux ont été menés sur la consommation énergétique des systèmes informatiques. Les auteurs de [PNP+19] ont étudiés l'impact des mauvaises pratiques de programmation dans les applications mobiles, et les auteurs de [JRLD19] se sont concentrés sur la mesure et la réduction de la consommation d'énergie dans les réseaux sans-fils publics. Les ordinateurs et les batteries qui font fonctionner un grand nombre de machines sont faits à partir de métaux rares et des matériaux non réutilisables. L'extraction minière, le rafinage et le recyclage de ces matériaux ont des impacts importants sur la végétation, la santé des animaux et des humains et pose des questions de respect des droits humains et des conditions de travail dans les mines. Une situation acceptable dans ce domaine se fait attendre et des travaux dans plusieurs domaines, y compris l'informatique, est nécessaire pour réduire cet impact.

Enjeux sociaux. Un aspect spéficique du 21ème siècle, par rapport aux précédents, est l'omniprésence de plus en plus grande des ordinateurs dans la société humaine. Un jour typique d'un être humain vivant dans un pays de l'ouest peut commencer par la sonnerie d'un téléphone. Avec ce téléphone, il peut lire les nouvelles, vérifier la météo, écrire des messages avec ses pairs et gérer son

calendrier. La plupart des téléviseurs d'aujourd'hui sont des ordinateurs. La plupart des moyens de transports motorisés comprennent un ou plusieurs ordinateurs. Beaucoup d'aspects quotidien, comme les tâches administrative, faire les courses (magasiner, diraient nos amis canadiens francophones), ou préparer un itinéraire impliquent des ordinateurs. Les ordinateurs ont aussi un impact profond sur beaucoup d'aspects de la vie social d'un être humain : ils rendent les communications avec des gens partout dans le monde facile et pratique ; ils changent notre manière d'interagir avec les autres. Ils ont un rôle majeur dans des domaines comme le divertissement et l'art. Cet aspect a été le sujet du journal Computers In Entertainments (CIE), qui a fonctionné pendant 15 ans jusqu'en décembre 2018. Ils posent aussi beaucoup de questions éthiques autour de "la vie privée, l'autonomie, la sécurité, la dignité humaine, la justice et les rapports de pouvoirs"[1] [RTKvE18]. De même, les ordinateurs assistent — ou aliènent – les humains dans leur travail, ce qui a des conséquences substentielles dans le monde du travail et son organisation : en automatisant des tâches (potentiellement périleuses ou fastidieuses) historiquement réalisées par des humains, ils réduisent la quantité de travail que l'humanité doit réaliser[2], et en permettant des résultats autrement inateignables, ils peuvent aussi augmenter la quantité de travail à faire. Des travaux ont été menés sur l'implication des ordinateurs dans le monde du travail [EMS18].

Sûreté et fiabilité. Les ordinateurs sont aussi impliqués dans les systèmes critiques. La sureté de chaque vol, des fusées et des réacteurs nucléaires dépendant du bon fonctionnement de plusieurs ordinateurs. Les services d'urgences et de secours dépendent aussi des ordinateurs pour la communication et la localisation. Nous commençons à dépendre des ordinateurs pour conduire des voitures et nous assister dans un nombre grandissant d'activités. La sûreté et la fiabilité est donc essentielle. S'assurer de la sûreté et la fiabilité des ordinateurs est difficile : les ordinateurs sont des systèmes complexes, composées de multiples parties, elles mêmes complexes. Les réseaux d'ordinateurs le sont encore plus. Cette complexité rend ardu le raisonnement sur ces systèmes, ce qui les rends sujets à des défauts de conceptions qui peuvent avoir des conséquences variées, du simple agacement à la perte de plusieurs vies. De nombreux journax (RESS[3], STVR[4], MR[5]) and conférences (ISSRE[6], SAFECOMP[7]) se concentrent sur les méthodologies et techniques visant à éviter, détecter et étudier de tels problèmes.

[1] texte original : "privacy, autonomy, security, human dignity, justice, and balance of power"

[2] C'est ensuite notre responsabilité en tant que société d'utiliser et profiter de ce temps libre additionel de la meilleure manière possible.

[3] Reliability Engineering & System Safety

[4] Software Testing, Verification & Reliability

[5] Microelectronics Reliability

[6] IEEE International Symposium on Software Reliability Engineering

[7] International Conference on Computer Safety, Reliability, and Security

Foreword

Computer Science as a Research Topic

During this Ph.D., people asked me what I was working on. Some were surprised that computer science is a research field. According to WordNet, "computer science is the branch of engineering science that studies (with the aid of computers) computable processes and structures" [worb] and computers are "machines for performing calculations automatically" [wora]. In addition to computation and data processing, an important feature of today's computers is communication. In this paragraph, I shall provide an opinionated explanation on why I think computer science is important to the unconvinced reader.

Environmental matters. Our impact as a species on the environment has become a central concern and computer scientists have their role to take in this matter. Research in computer science has been done to study and minimize energy consumption and reduce carbon emission in buildings [AZ19], manufacturing processes [JZ19], induction motors [SKN19]. Computer systems themselves consume a lot of energy. It is estimated that powering Internet requires between 1.1% and 1.9% [RM11] of the global energy usage by the humanity. A first obvious solution to this concern is to avoid using computers when not necessary. However, this must not deter us from studying and improving the energy consumption of computer systems. Lowering our reliance on computers is not always the obvious solution to reducing our ecological impact. They can replace power hungry solutions, like flying, with solutions like videoconferencing. Work has been done on power consumption of computer systems. Authors of [PNP+19] studied the impact of bad programming practices in mobile applications, and authors of [JRLD19] focused on measuring and reducing power consumption in public wireless networks. Computers and batteries powering a number of widely used devices are made from rare-earth metals and non-reusable materials. Mining, refining and recycling these materials have a serious impact on vegetation, human and animal health, and raise concerns with respect to human rights and labor conditions in mines. An acceptable picture in this respect is yet to be seen and work in multiple fields including computer science is needed to reduce this impact.

Social matters. A special trait of the 21^{th} century, in contrast to the previous ones, is the ubiquity of computers in the human society, and increasingly so. A typical day of a human living in a western country may start with a phone ringing. With this phone, this human may read the news, check out the weather forecast, message their relatives, manage their calendar. Most today's TVs are computers. Most motorized means of transports embed one or several computers. Many day-to day activities like paperwork, shopping, getting directions involve computers. Computers also have a profound impact on a number of aspects of the social life of a human: they make it easy and convenient to communicate with people all around the world; they change the way they interact with other people. They have a major role in entertainment and art creation. This aspect was the focus of the journal Computers In Entertainments (CIE) that was run for 15 years until December 2018. They also raise many ethical concerns like "privacy, autonomy, security, human dignity, justice, and balance of power" [RTKvE18]. Computers also assist — or alienate — humans in their labor, which has substantial consequences in the world of work and its organization: by automating (potentially perilous or tedious) tasks historically done by human beings, they reduce the amount of work the

humanity has to do[8], and by enabling otherwise unreachable results, they may also increase the amount of work to do. Research has been done on the implications of computing in the world of work [EMS18].

Safety and reliability matters. Computers are also involved in critical systems. The safety of each flight, spacecraft and nuclear power plant rely on the correct operation of several computers. Emergency services and rescue services also rely on computers for communication and geolocation. We are beginning to rely on computers to drive cars and to assist us in a growing number of activities. Safety and reliability of computers is therefore essential. Ensuring safety and reliability of computers is difficult: computers are complex systems, composed of many parts, themselves complex. Networks of computers, even more so. This complexity makes it hard to reason about these systems and make them prone to design flaws that can have various consequences, from a slight annoyance to the loss of multiple human lives. Numerous journals (RESS[9], STVR[10], MR[11]) and conferences (ISSRE[12], SAFECOMP[13]) have focused on methodologies and techniques aiming to avoid, detect and study such flaws.

[8] It is then the responsibility of the society to take advantage of this increased free time in the best way.
[9] Reliability Engineering & System Safety
[10] Software Testing, Verification & Reliability
[11] Microelectronics Reliability
[12] IEEE International Symposium on Software Reliability Engineering
[13] International Conference on Computer Safety, Reliability, and Security

Contents

List of Figures

Chapter 1

Introduction

Contents

In the space of a few decades, computers have, for better or worse, become involved in many aspects of many people's lives and of the society. We require them to handle our wake-up alarms, our communications with our peers, our safe transportation, our entertainment, our medical records, answering our questions, predicting the whether, processing big chunks of data for scientific work. We require them to work flawlessly, instantaneously and at any time. We rely on them for critical matters, and increasingly so. We are trusting them with our lives. Some people rely on a pacemaker to regulate their heart. Many safety-critical devices in places like hospitals and in nuclear plants are computer-based. We are now tasking computers to drive cars and trains autonomously, and do to so far more reliably than human beings.

1.1 Bugs Plague Our Computer Systems

1.1.1 What Are Bugs

A computer is composed of hardware running software. Network of computers are composed of several computers, Internet arguably being the biggest one. Each of these levels are designed by error-prone human beings, and are increasingly complex. The more a system is complex, the more it is likely to contain errors. Errors are due to various factors such as tiredness, distractions, difficult-to-understand parts, bad communications between people, bad team or time management and excessive complexity due to bad design. Errors, as well as wear and dust causing hardware failures, can lead to undesirable behaviors called bugs.

Bugs can cause slight annoyances, but can also lead to the loss of multiple lives and have all sorts of impact in between. Correct and safe operation of planes, spacecrafts, nuclear power plants, emergency services as well as an increasing number of other critical applications requires reliable software and computers. Unfortunately, the complexity of computers and networks makes them prone to design flaws, causing bugs. Hence, the demand for avoiding, understanding, fixing and more generally handling bugs is high. Hardware and software reliability is an active research topic. Numerous journals (RESS[1], STVR[2], MR[3]) and conferences (ISSRE[4], SAFECOMP[5]) have focused on methodologies and techniques aiming to avoid, detect and study bugs. Work has been done to help improve software reliability on the processes involved in software development [DD08] to address issues related to time management and to communication between the different stakeholders in a software project, as well as on how to handle errors in the code (model checking [EC80], static analysis [CC77], software testing [IML09, IML07], runtime verification [BF18a, HG05, LS09, SHL12]).

In this thesis, we focus on software bugs rather than on hardware bugs and do not address the causes of coding errors themselves.

1.1.2 Why Are They Hard to Tackle

Detecting, locating and understanding bugs may be hard for various reasons. Programs are often built from large codebases. In Table 1.1, we give estimations of the size of the code of several well-known projects. An error in the code may cause a bad computation during the execution that has consequences later, making this error difficult to track. The distance in the code, in terms of number of executed lines of code between the manifestation of a bug and its cause may be important. Large codebases are composed of multiple components. As the number of components grows, understanding the interactions between them may be more difficult and testing these interactions exhaustively, with the number of possible states of each component being huge, becomes impractical. According to a survey from Stripe [Str] on 1,000 developers and 1,000 C-level executives, maintenance work

[1]Reliability Engineering & System Safety
[2]Software Testing, Verification & Reliability
[3]Microelectronics Reliability
[4]IEEE International Symposium on Software Reliability Engineering
[5]International Conference on Computer Safety, Reliability, and Security

Project	Size (LoC)
The VLC Media Player	0.7 M
The Gimp	0.7 M
The OsmAnd mapping Application	1 M
Python	2 M
The Blender 3D creation suite	2 M
The GNU Debugger	3 M
PHP	4 M
The GNU Compiler Collection	8 M
The LibreOffice suite	10 M
The Mozilla Firefox browser	20 M
The Chromium browser	27 M
The Linux kernel	37 M

Table 1.1: Size of various projects in terms of lines of code (LoC).

```
1  int r = dice();
2  if (r = 6) {
3      puts("Win!");
4  }
```

Figure 1.1: Wrong C operator

like debugging and fixing bad code makes up for as much as half the development time on average. Work to reduce this time is necessary.

1.2 Existing Approaches to Tackle Bugs

Various tools and approaches exist to handle bugs, from avoiding them to detecting, finding and fixing them.

1.2.1 Linting

Linters check the code for formatting and matters related to coding style like indentation, line lengths, placement of braces and variable names. A coding style issue may be caused by a lack of focus or a mistake that may also be the cause of an actual error. Coding styles and consistency make the code more readable and avoid distractions caused by the lack of uniformity. Enforcing this uniformity also hopefully eases spotting obvious errors. Linters are used to forbid constructions deamed error-prone by a project or the community around the involved programming languages. Figure 1.1 shows a problematic C code that can be spotted by a linter. In the condition, the assignement operator is used instead of the equality operator. The code is still valid C, but does not behave as intented and may lead to the wrong result. A linter may advise either using the equality operator, or surrounding the assignment by parentheses, to clarify that the use of the assignement operator is intentional. Widespread compilers such as GCC or Clang actually emit a warning in such a case when the right compiler option is used, effectively acting as linters. Figure 1.2 shows a piece of code that was responsible of a security vulnerability in Apple's implementation of SSL called `goto fail`. A mistake made a programmer duplicate the line containing `goto fail`. This line, meant to be run conditionally, was running unconditionally because the duplicate is not guarded by the `if` condition, wrongly short-cutting the following security checks. An appropriately configured linter would complain that the duplicate line is wrongly indented, potentially making a programmer wrongly believe that the line belongs to the body of the `if` statement.

```
1  ...
2  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
3      goto fail;
4      goto fail;
5  if (...)
6  ...
```

Figure 1.2: The Apple goto fail vulnerability

1.2.2 Static Analysis

Static analyzers look for potential mistakes and catch problems in the code that linters may not. They parse the code of a project to find *code smells* (pieces of code that look suspicious) like wrongly copy-pasted lines, unreachable code and tautological conditions. In Figure 1.2, a static analyser may warn that the code following the second `goto fail` statement is unreachable, or that the duplicate `goto fail` line is suspicious. Static analysis is an active research field [Pod18] and may involve more advanced issues like array bound checks, detection of loop invariants and infinite loops. Abstract interpretation [CC77] is a form of static analysis in which a code is partially executed to approximate possible value ranges for variables or control flows in the code, without performing all the computations. Static analysis helps finding bugs before actually executing a program, but may produce false positives and false negatives. It is also proved that static analysis is undecidable: it may not find all the bugs due to missing information, only available at runtime [Lan92]. Some bugs that could theoretically be found using static analysis may also require a lot of time and ressources, making this method effectively impractical for these bugs. Therefore, complementary methods are needed for bugs that are not handled by static analysis.

1.2.3 Model Checking

Static analysis is limited by the fact that runtime information is required for spotting some bugs (i.e., for checking some desirable properties). One immediate solution to this limitation is exhaustive check, by running the program to check with all the possible inputs. However, even for small programs, this may be impractical or outright impossible. The set of possible inputs may be infinite, or too big to make the process practical. The program may not stop for all inputs, or effectively take too much time for the process to be practical. The idea of model checking is to check a restricted, finite set of possible inputs on a model of a program, built in a language that guarantees that the model can only have a finite set of possible states for any input. Model checking [CGP01] may be used to check a design before implementing it. The obvious drawback of model checking is that it requires a lot of memory and computing resources and rapidly becomes impractical for even moderately sized models. Furthermore, the implementation is not being checked, only a model is, but there is no guarantee that the implementation correctly implements the model.

1.2.4 Manual and Beta Testing

Since some bugs may only be spotted at runtime, an obvious way to check a program for bugs is to run it. Manual testing consists in manually running the program during the development with manually chosen inputs and check that it behaves as intended. Beta testing [SDB94, JSNQ17] consists in actually using the program. The program is tested by users for usability, to try new features and to check for bugs. Manual testing has one obvious drawbacks: it is a tedious way to find bugs and may only be practical during early development, to quickly check that a code that has just been written does not break in an obvious way. Quality assurance testing is half-way between manual and beta testing. The program is manually tested for bugs, to check that it works as intented in known scenarios. This technique is good to ensure that the program does not break in obvious ways, but may not detect bugs affecting corner cases. All these approaches share another drawback: while bugs can be detected, they may not link it to the code of the program and do not provide insight on the cause of the bugs.

1.2.5 Unit Testing

Unit testing consists in running small parts (functions) of the code with various inputs. A unit test suite contains many such tests. This method helps detect breakages on cases where results are known for the given inputs. Tests may be added as bugs are spotted and fixed to avoid similar regressions in the future. When a test does not pass, the cause of the bug ought to be understood easily, since the part of the code being tested is small. A non-passing test may either be caused by an actual error in the code, or by a test that became erroneous because the assumptions evolved. In the former case, the involved part of the code may be fixed. In the later case, a test may be updated or deleted. Continuous integration runs tests on a regular basis, as the code is written, making unit testing systematic. Policies for writing unit tests may be enforced, ensuring that as much code as possible is tested. Unit testing does not, however, cover all bugs, especially corner cases that have not been anticipated. Moreover, the correctness of tested functions does not guarantee that the interactions between these functions are correct. Many other approaches based on testing exist [HKKT18].

1.2.6 Runtime Verification

Some bugs may be detected by checking that the runtime behavior of the program is correct with respect to some given properties. This process is called runtime verification [JLSU87, LKK+99, Hav00, HR02, BF18a, HG05, LS09, SHL12]. Properties can be derived from the program specification, of the library used in the program, and are formally expressed in a suitable formal logic system. To evaluate the program against a property, the program is instrumented. This instrumentation is specific to this property. As functions of interests are called, variables of interest are accessed or other events of interest happen in the program, the instrumented program produces a trace that is fed to a monitor, a device in charge of evaluating the property against this trace. The moniror produces a verdict, indicating whether the program verifies the given property during this run. Contrary to beta testing, runtime verification may find a bug before it manifests, bringing the developer closer to the error in the code causing the bug, because the property being verified is violated at this point in the execution. However, an usual limitation of runtime verification is that it does not provide an explanation on a bug in relation to the internal state of the program. This limitation makes it difficult to understand the cause of the bug when the error in the code is not obvious to spot.

1.2.7 Interactive Debugging

Contary to the approaches mentioned above, interactive debugging [PSK+16a, Pou14] is a method to study bugs rather than to detect them. When using interactive debugging, developers typically know the existence of a bug, and want to understand its cause. To debug a program interactively, developers use tools called interactive debuggers. These tools allow them to run the program step by step, inspect the memory and values of variables. Using interactive debuggers, developers can set breakpoints, which are like bookmarks set at some locations in the program code. This lets them execute the code normally up to these locations, after which the interactive debugger suspends the execution. Interactive debuggers allow modifying a value in the program during the execution, or executing some code, testing a fix without restarting the program, or, more generally, studying the behavior of the program. Interactive debugging is powerful but often tedious and not well-understood: few research studies have conducted to understand the interactive debugging process. [PSK+16b] introduces a framework to conduct such a study and presents a few observations on this approach. Notably, developers follow different debugging patterns and they complete debugging tasks faster when they toggle breakpoints carefully. When studying a bug, developers often observe the bug in the debugger, and try to get closer and closer to the error causing the bug by setting breakpoints further and further from the manifestation of the bug. This process is manual, iterative and involves a lot of trial and error. It is not backed by any formal method. Interactive debuggers provide features to study a program, but do not provide any guidance as to how to use these features to accomplish this task.

1.3 About This Thesis

1.3.1 Let Us Gather Interactive Debugging and Runtime Verification

We believe that interactive debugging and runtime verification are complementary. Runtime verification helps detect bugs, while interactive debugging helps study them. Interactive debugging is mostly manual, while runtime verification is mostly automatic. Runtime verification provides an interpretation of the program execution while interactive debugging lacks guidance. Interactive debugging provides features to study an execution in depth, while runtime verification provide an high level view of the execution, related to the specification of the program. We think that this complementary may be leveraged to ease the task of studying bugs. In this thesis, we introduce, define and evaluate interactive runtime verification, a combination of interactive debugging and runtime verification. We aim to reduce the time of an interactive debugging session, a fundamentally empirical and manual process, leveraging **detection** and **evaluation** features of runtime verification to **guide** and **automate** an interactive debugging session, by adding **automation** and rigor to it, while keeping the **interactivity** provided by interactive debugging, valuable to freely inspect the program state at any point of the execution. A key idea is to use the verdicts of the monitor to control the execution, using the interactive debugger as an **instrumentation**, **control** and **observation** platform. This combination aims to address the major drawbacks of the two methods by using runtime verification as a guide to interactive debugging, and by using interactive debugging as an instrumentation platform for runtime verification, using the verdicts produced by the monitor to take decisions on the control of the execution, and to start the interactive exploration of a bug closer to its cause, before its manifestation.

1.3.2 Contributions and Results

In this thesis, we provide two frameworks for interactive runtime verification.

Firstly, we define a formal operational model of the behavior of an interactive runtime verification session involving one process and one monitor. We model a program under interactive debugging, capable of receiving requests for instrumentation. We compose this program under interactive debugging with a monitor and a component, the scenario, in charge of reacting to verdicts produced by this monitor. We provide algorithms implementing the operational semantics model of the interactively runtime verified model.

Secondly, we introduce a framework for distributed interactive and adaptive runtime verification, bringing support for distributed systems and the concurrent verification of multiple properties, and allowing existing components such as monitors and interactive debuggers to be adapted for i-RV. We present an architecture and a protocol for this framework, with a description of the behavior of each component of this architecture. We report on our modeling and model-checking of the protocol.

Thirdly, we present our implementation, Verde. A first version implements the formal operational model. A second version implements the distributed framework.

Lastly, using our implementation, we provide experiments to evaluate and validate the effectiveness of interactive runtime verification.

1.3.3 Publications and workshops

A first paper, *Interactive Runtime Verification — When Interactive Debugging Meets Runtime Verification*, introducing interactive runtime verification, has been published in ISSRE'17 [JFMP17]. An extended version of this paper, including the formal model and the proofs presented in Chapter 3, as well as a paper presenting the contributions in Chapter 4 are to be submitted. A part of this work has also been presented at the 13th TAROT Summer School on Software Testing, Verification & Validation and in the workshop AFADL18 (Approches Formelles dans l'Assistance au Développement de Logiciels).

1.3.4 Organisation of this thesis

In Chapter 2, we present interactive debugging and runtime verification, the building blocks of interactive runtime verification, and we give a high-level view of interactive runtime verification, presenting how we gather these two approaches.
In Chapter 3, we present a first approach to interactive runtime verification. We present a formal model described using operational semantics. This model precisely defines the behavior of a program under interactive runtime verification. We state that the model preserves the behavior of the original program (weak simulation). We provide an algorithmic view of the model, suitable for implementation.
In Chapter 4, we introduce a framework providing the ability to handle distributed systems and the possibility to monitor different properties. We propose a protocol to allow reusing existing debuggers and monitors. We present a model for this protocol and check it with the SPIN model against various properties. These properties ensure that the events produced by the debugged programs lead to a production of verdicts to which the scenario react by controlling the execution.
In Chapter 5, we present Verde, our implementation of the frameworks presented in the two previous chapters. This implementation is composed of two versions. The first version is an implementation of the model presented in Chapter 3, based on the GNU Debugger. The second version is an implementation of the framework presented in Chapter 4. With this second version, we provide an adapter for the GNU Debugger, based on the first version, and an adapter for the Java Debugger.
In Chapter 6, we present experiments assessing the usability and usefulness of i-RV using Verde.
In Chapter 7, we present related work in the domain of detecting and handling bugs. We first present testing-based approaches. We then present work based on static analysis and abstract interpretation. We provide a comparison between interactive debugging and interactive runtime verification. We present work based on runtime verification and instrumentation techniques, and work addressing the monitoring and debugging of distributed systems.
In Chapter 8, we conclude the thesis.

Appendices. In Appendix A.1, we provide a proof for the weak simulation theorem given in Chapter 3. In Appendix A.2, we provide the specification of the protocol presented in Chapter 4, as implemented in Verde, presented in Section 5.2. In Appendix A.3, we present a property used in our experiment involving Zsh.

Chapter 2

Background

Contents

Summary. Interactive runtime verification (i-RV) combines interactive debugging and runtime verification. In this chapter, we introduce two approaches: interactive debugging and runtime verification and the aspects of them we consider important. We motivate their combination and introduce our approach: interactive runtime verification.

We first introduce interactive debugging in Section 2.1. We present interactive debuggers and their typical use to debug a program. We then present their main features: breakpoints (Section 2.1.2), step by step debugging (Section 2.1.3), language support (Section 2.1.4), checkpointing (Section 2.1.5) and we discuss the limitations of interactive debugging relevant to our work (Section 2.1.6). We are interested in interactive debuggers as an instrumentation platform, using breakpoints and other features of interactive debuggers to observe the program execution. As such, we take a close look at the different kinds of breakpoints and how they are handled by the debugger.

We then introduce runtime verification in Section 2.2. We discuss the different instrumentation techniques used in runtime verification and introduce the notions of traces and events (Section 2.2.1). We give a short overview on how property and specifications can be described (Section 2.2.2). We present how instrumentation is used to evaluate properties and produce verdicts (Section 2.2.3). We present some limitations of runtime verification (Section 2.2.4).

We list features from the two approaches we want to preserve and other considerations raised when combining the two approaches in Section 2.3. We give an overview of interactive runtime verification in Section 2.4. We conclude the chapter by comparing an interactive debugging session with an i-RV session to illustrate i-RV.

Illustrative example. We use a sample C program depicted in Figure 2.1 as an example throughout this chapter. 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.

2.1 Interactive Debugging

Interactive Debugging [PSK+16a, Pou14] consists in studying the behavior of a program using an interactive debugger. More specifically, interactive debugging is often used to study bugs. 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: the developer can observe its whole internal state. The execution is seen as a sequence of program states that the developer inspects step by step using a debugger in order to understand the cause of misbehavior. Interactive debugging lets a developer observe and fiddle with the internal state of a program at any time during its execution. Interactive debuggers are programs that make use of debugging features provided by operating systems to inspect running processes. Common interactive debuggers include the GNU Debugger (GDB)[1] and the LLVM Debugger (LLDB)[2]. They support debugging programs written in programming languages like C and C++. The reference implementation of an interactive debugger for Java is JDB (The Java Debugger). For Java and most programming languages based on the Java Virtual Machine, interactive debugger can also be found in most IDEs like Eclipse[3] or IntelliJ[4].

2.1.1 A Debugging Session

A debugging session generally consists in repeating the following steps: making hypotheses on the cause of the bug, executing the program in a debugger, setting breakpoints on statements or function calls executed before the suspected cause of the bug (see Figure 2.2). 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) to determine the cause of the problem. This may be done by starting from the

[1] `https://sourceware.org/gdb/`
[2] `https://lldb.llvm.org/`
[3] `https://eclipse.org/`
[4] https://www.jetbrains.com/idea/

```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_c; 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;
}
```

Figure 2.1: Faulty C program

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.

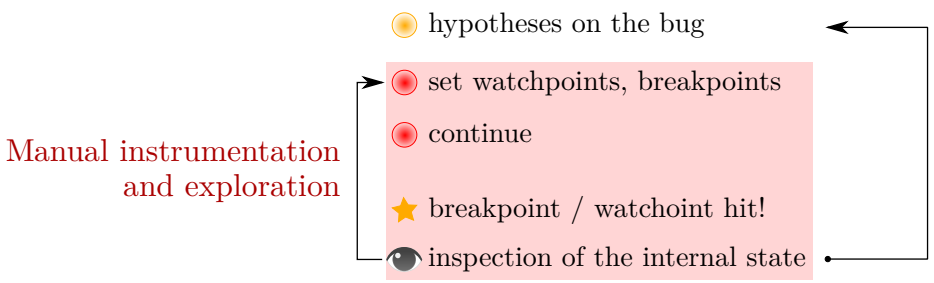


Figure 2.2: A typical interactive debugging session.

end of the erratic execution, to the point where the bug can be observed, and then going back up to the actual error in the code. This process can require testing several hypotheses and re-executing the program several times, each time with more knowledge gained from the observation of the internal state. The execution is seen at a low level (assembly code, often mapped to the source code). 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.

Illustration. In Figure 2.3, we present how a developer could track the bug of the sample C program given in Figure 2.1 using an interactive debugger. First, to check that the problem is not in 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 the rest of this section, we list common and notable features of interactive debuggers.

2.1.2 Breakpoints

Intuitively, a breakpoint is a bookmark set on a particular instruction in the code of a program. When this bookmark is reached, the execution is suspended, letting the user inspect the internal state of the program at this point of the execution.

Breakpoints were invented for the ENIAC[5] (1945–1956), the first general purpose programmable electronic computer [eni]. To set a breakpoint, the cable leading to the instruction on which the programmers wanted the instruction to stop was unplugged. Though full-featured interactive debuggers did not exist yet, one could argue that this invention marks the beginning of interactive debugging. This invention allowed the programmers to inspect the state of a program during its execution rather than trying to guess what happened after the execution, with the result or a trace. See Section 2.1.2.6 for an explanation on how breakpoints are implemented in modern interactive debuggers.

2.1.2.1 Location of a Breakpoint

The location of a breakpoint can be specified in different ways. One of the most common ways of specifying the location of a breakpoint is by giving a file name and a line number in the source code of the program. Breakpoint specified in such a way are called *line breakpoints*. Another common way of specifying a breakpoint is by giving the name of a function. The breakpoint is set at the first instruction of the specified function. Breakpoint specified in such a way are called *function breakpoints*, or *method breakpoints*. *Exit breakpoints* can be set to trigger when a specified function (method) exits. *Finish breakpoints* can be set to trigger when the current function (method) exits.

Example 2.1.2.1 (Breakpoint)**.** We consider the sample C program given in Figure 2.1. In GDB, `break 32` (or `break main.c:32`) sets a line breakpoint on line 32. `break queue_push` is a function breakpoint on function `queue_push`. During the execution, if the program is suspended by

[5] Electronic Numerical Integrator and Computer

```
$ ./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
```

Figure 2.3: Interactive debugging of the faulty C program.

one of these breakpoints, `finish` sets a finish breakpoint. The execution continues until function `queue_push` returns.

2.1.2.2 Data Breakpoints (Watchpoints)

Intuitively, a watchpoint (also called data breakpoint) is a bookmark set on a particular location in the memory. Watchpoints suspend the execution when the specified location is read, modified or both.

Remark 2.1.2.1. Most interactive debuggers allow defining watchpoints on an expression: the execution is suspended when the value of an expression changes. Internally, the debugger sets watchpoints on every location related to the value of the expression.

Example 2.1.2.2 (Watchpoint)**.** We consider the sample C program given in Figure 2.1. In GDB, `watch queue->pos_v` sets a watchpoint on variable `queue->pos_v`, triggered when this variable is modified.

2.1.2.3 Catchpoints

Catchpoints trigger on system calls (syscalls), interrupts, signals and other events like C++ exceptions and Ada assert failures. Syscalls are done using special instructions (rather than functions). These instructions are requests to the operating system, and depend on the operating system. Requests are granted by the operating system by running privileged code. Regular breakpoints cannot be used to catch: specific support from the operating system and the debugger is required.

Example 2.1.2.3 (Watchpoint)**.** In GDB, `catch signal 11` sets a catchpoint on signal 11 (segmentation fault).

2.1.2.4 Conditional Breakpoints and Watchpoints

Breakpoints and watchpoints may be *conditional*. That is, a boolean expression can be attached to a breakpoint. When the corresponding instruction is reached, the execution is suspended, the expression is evaluated. If the expression evaluates to false, the execution is resumed without the user noticing the suspension. If the expression evaluates to true, the execution is suspended.

2.1.2.5 Multi-Location Breakpoints

Sometimes, several locations in the program relate to a single breakpoint. The breakpoint will be inserted at all these locations and the execution will be suspended when any of these locations is reached.

2.1.2.6 Hardware and Software Support

Breakpoints and watchpoints are implemented using the debugging features provided by the operating system. Debuggers, using a dedicated system API like ptrace on Unix-like systems, ask the system to attach to the process to debug. With such an API, debuggers can access and modify the state of this process.

Hardware breakpoints. Hardware support for breakpoints exists in several architectures and can be used for a limited number of breakpoints[6]. Hardware breakpoints are usually set by storing the address of the breakpoint in a dedicated register. The execution stops and the debugger is called when the specified address is reached during the execution.

Software breakpoints. Breakpoints are usually implemented in software by replacing an instruction in the running program by a special breakpoint instruction[7]. When such an instruction is

[6] Usually up to 4 or 6, depending on the architecture and the specific implementation.

[7] On x86, this instruction is `int 3`, called the *trap (to the debugger) instruction*. On ARM (resp. RISC-V), this instruction, called the breakpoint instruction, is `BKPT` (resp. `EBREAK`).

reached, the execution of the program *traps*, that is, the execution is suspended and an interruption is produced. The interruption handler notifies the debugger of the situation. To resume the execution, the breakpoint instruction is temporarily replaced by the original instruction. The instruction is run, and replaced once again by the special breakpoint instruction.

Hardware watchpoints. Like hardware breakpoints, a limited number of watchpoints can be set by storing their location in a dedicated register. The execution is usually suspended after the location is accessed.

Software watchpoints. Handling a larger number of watchpoints requires heavy software support. They are implemented by suspending the execution after each instruction, and by checking whether the last instruction accessed the location of any watchpoint. It should be noted that just checking whether the values at the watchpoint locations changed is not sufficient: some watchpoints must trigger when the value is read, and it may be desirable to trigger a watchpoint if a value is set, even if it is not modified. Thus, the implementation is specific to each architecture and requires knowing what each instruction does.

2.1.3 Step by Step Debugging

Interactive debuggers provide tools to run an execution step by step, that is, line by line, or instruction per instruction, stopping the execution after completion of each instruction. This feature[8] lets the user of the interactive debugger inspect and modify the state of the program after each step. Most interactive debuggers also allow skipping the execution of some statements.

2.1.4 Language Support

First interactive debuggers [ddt] only supported debugging at assembly level. Source-Level Debuggers, also called Symbolic Debuggers, supporting higher programming languages like C, appeared in the late 1970s[9].

2.1.5 Checkpoints

Some interactive debuggers, like GDB, have limited support for checkpointing. Checkpoints can be seen as a bookmark in the execution. They are a snapshot of the internal state of the program at the moment they are created. They can be used to go back in time without completely restarting the execution, and to explore several execution paths from the same point in the execution.

2.1.6 Strengths and Weaknesses

Interactive debugging is a **widespread** method among the developers. It allows the developers to **consult** and **modify** the internal state of their program during the execution to reason about their behavior and find the cause of a bug. Advanced interactive debuggers provide valuable features such as reverse debugging and checkpointing. Unfortunately, the interactive debugging process can be tedious and prone to a lot of trials and errors. Using interactive debugging is mostly an empirical and a manual process. This process **lacks guidance, rigor and automation** that usually comes with formal methods. Moreover, the low-level abstraction and the number of details can be overwhelming. Interactive debugging **does not target bug detection**: usually, a developer already knows the bug existence and tries to understand it.

[8] Appeared in the late 1960s. See DYDE, the Dynamic Debugger, an online debugger for OS/360 presented in 1969 [Jos69]

[9] `adb` (Advanced Debugger), the standard UNIX debugger, was one of the first source-level debuggers [MB]

2.2 Runtime Verification

Runtime verification [JLSU87, LKK+99, Hav00, HR02, BF18a, HG05, LS09, SHL12] (aka monitoring) is a formal method to check the behavior of a system at runtime, aiming at checking the correctness of an execution with respect to a specification. This verification is done on a sequence of events produced using instrumentation. The program is seen as a black box: its complete state at runtime is not accessible to the monitor. In this section, we give an overview of some important notions of runtime verification.

2.2.1 Instrumentation, Traces and Events

Verifying an execution relies on retrieving information at runtime. This process is called instrumentation. Instrumentation can either be *static* or *dynamic*. Static instrumentation is set before running the program and cannot be modified at runtime. Dynamic instrumentation can be modified as-needed during the execution.

2.2.1.1 Instrumentation Techniques

In this section, we list common instrumentation techniques used in runtime verification.

Compile-Time Instrumentation. Instrumentation is done at the level of the source code or at the binary (or bytecode) level, by adding special instructions. RiTHM [NJW+13] 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) using Compile-Time Instrumentation. Frama-C [CKK+12] is a modular platform aiming at analyzing source code written in C and provides plugins for monitoring programs using Compile-Time Instrumentation.

Instrumentation Based on the VM of the Language. Programs written in some languages like Java, Python and Javascript are run using a Virtual Machine (VM). The VM can provide introspection features that can be used as instrumentation. The Jassda framework [BM02], which uses CSP-like specifications, uses the Java Debugging Interface provided by the Java Virtual Machine (JVM) to build traces of programs to check.

Dynamic Binary Instrumentation. With Dynamic Binary Instrumentation (DBI), instrumentation is done by dynamically adding instructions to its binary code at runtime and run in a virtual machine-like environment. Valgrind [NS07], DynamoRIO [BZA12] and Intel Pin [LCM+05] are three DBI frameworks that allow writing dynamic instrumentation-based analysis tools.

Aspect-Oriented Programming. Aspect-Oriented Programming [KLM+97] is a paradigm in which points of instrumentation (point cuts) are linked to some pieces of code are run (advices). An aspect is the combination of point cuts and advices. In Aspect-Oriented Programming, instrumentation itself can be done using Compile-Time Instrumentation. AspectJ [Kic02, Kat06] is an implementation of aspects for JVM-based programming languages and AspectC++ [MSS02] is an implementation for the C++ programming language. LARVA [CFG11] and JavaMOP [CR07] are monitoring tools for programming languages based on a virtual machine (mainly Java). Most Java-based runtime verification tools rely on AspectJ to instrument programs.

Debugger-Based Instrumentation. Although less usual, it is also possible to instrument programs using features provided by a debugger. Several debuggers can be scripted. They allow setting breakpoints and watchpoints automatically. When reached, functions provided in the script are executed. This instrumentation can be dynamic: instrumentation can be added for events newly needed for the property evalutation, and instrumentation for events that are not needed anymore can be removed.

```
1  main()
2  queue_new() -> 0xe5b270 FIXME arrow
3  queue_push_str(0xe5b270, "oh, a nasty bug is here!")
4  queue_push(0xe5b270, 'o')
5  queue_push(0xe5b270, 'h')
6  queue_push(0xe5b270, ',')
7  ...
8  queue_push(0xe5b270, 'r')
9  queue_push(0xe5b270, 'e')
10 queue_push(0xe5b270, '!')
11 queue_display_result(0xe5b270)
12 queue_destroy(0xe5b270)
```

Figure 2.4: Execution trace of the sample C program given in Figure 2.1.

2.2.1.2 Events and Trace

The instrumentation produces pieces of information called *events*.

Kinds and granularity of events. There are different kinds of events. An event can be a *signal* (e.g., the light was just turned on) [MN04], a *state change* (e.g, the light went from off to on) or a (partial) *description of the current state* (the light is currently on). The kind of event the instrumentation is set to produce depends on how properties are described or checked. Event can be produced by *sampling* (every specified number of executed instruction or time unit) [SBS+11], each time the program state is updated, or anything in between.

Trace. The set of events produced by the instrumented program during the execution is called a *trace*. When instrumenting a sequential program, the trace is a *sequence* of events. When instrumenting a distributed system, the trace may be a graph, in which events are partially ordered [EF18].

A trace constitutes an *abstraction* of the actual execution of the program. This abstraction is specific to the property being checked. It usually does not contain a comprehensive view of the execution. Rather, it (only) contains events useful to check the property. Parts of the execution that are not relevant to the property are discarded. Moreover, there can be many ways to describe what happens in a program. As such, events of interest already constitute an interpretation of what happens in the program. The trace can be analyzed during or after the execution.

Example 2.2.1.1. In Figure 2.4, we give a trace of an execution of the sample C program given in Figure 2.1.

2.2.2 Specification and Property Formalisms

A *specification* is a description of a behavior that must (or must not) be observed during an execution of a program. Properties are formal representations of the specification. A property is a mathematical object that can be used to check whether a trace complies with a specification. It is a predicate that sets apart correct traces from incorrect traces with respect to the specification. There are several ways to describe properties. The simplest properties are *assertions*. Assertions are properties that must hold true at certain points of the execution. Assertions are usually described using formulas. A *precondition* (resp. *postcondition*) is an condition that must hold true at the beginning (resp. end) of a block of code (e.g., a function). An *invariant* is an assertion that must hold true at some point during the execution. Specifically, a *loop invariant* is an assertion that holds true at the beginning (or the end) of a loop body, for each iteration of the loop. A *safety property* [AS87] is an assertion that most hold true at any point of the execution: "something bad should never happen". *Liveness properties* [AS85] are complementary to invariants. They describe something good that should happen during the execution: "something good should eventually happen".

There exist several theoretical frameworks to describe properties. First-order logic can be used to describe simple properties that apply on one, or each element of the trace. Regular expressions and

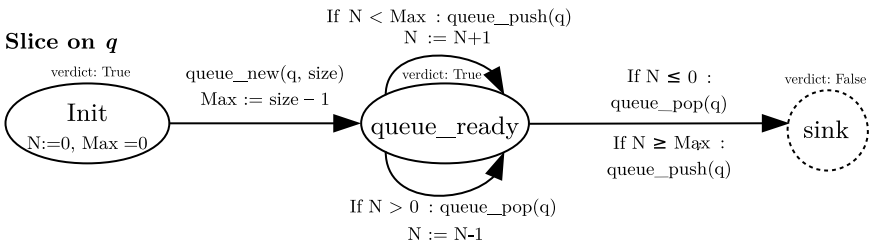


Figure 2.5: Property for the absence of overflow for each queue q in a program
Property for the absence of overflow for each queue q in a program. When a queue is created, the automaton is in state `queue_ready` for this queue. Pushes and pops are tracked. When a push (resp. pop) causes an overflow (resp. underflow) of the queue, the property is in state `sink`.

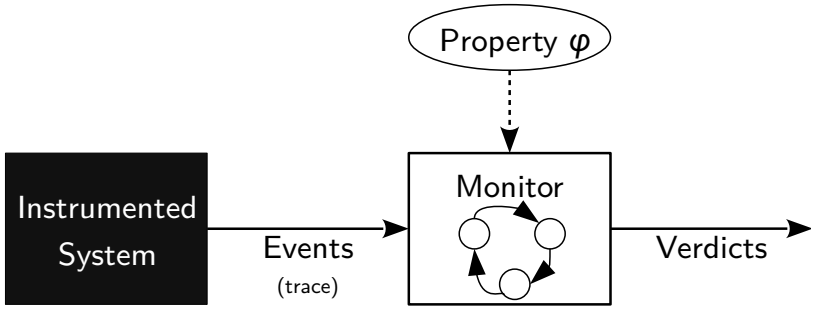


Figure 2.6: Runtime Verification.

variants of finite state machines allow defining patterns matching execution traces. Temporal-logic based formalisms add the notion of time when dealing with properties. One of the most widely used temporal logic formalism is the Linear Temporal Logic [Pnu77] (LTL), that builds upon propositional logic and adds temporal operator. There are many other Temporal-logic based formalisms like Signal Temporal Logic [MN04] suitable for monitoring continuous real-time systems or Interval Temporal Logic [MM83] suitable for monitoring traces composed of state intervals. In this work, we are interested in formalisms that apply to traces that can be produced using a debugger. That is, traces that are sequences of discrete parameterized events for sequential programs.

Example 2.2.2.1. We give two properties for the sample C program:

- the program only uses a queue after creating this queue, and before closing it (as a quantified regular expression).

$$\forall q, (\texttt{queue\_create}() \rightarrow q \ \cdot \ (\texttt{queue\_push}(q) \mid \texttt{queue\_pop}(q))^* \ \cdot \ \texttt{queue\_destroy}(q))^*$$

- the program does not overflow a queue (as a automaton): see Figure 2.5.

2.2.3 Monitoring and Verdicts

Checking an execution against a property has many names, among them *runtime verification* or *monitoring*. See Figure 2.6. The system is instrumented. The instrumentation produces events that are fed to a device called a *monitor*. The monitor can be a dedicated piece of hardware or a regular piece of software. It checks the trace against the property and produces *verdicts*. Verdicts are a reflection of the state of the program with respect to the property under evaluation. Monitor may, or may not produce verdicts for each event of the trace, and may, or may not produce temporary verdicts. That is, some monitors may, or may not, produce only one final verdict that indicates whether the trace fulfills the property. Verdicts may be booleans [PZ06], they possibly can also take values such as *unknown*, *currently true*, *currently false* when no definitive verdict has be found yet [FFM12], or even be values containing more information and context.

2.2.4 Strengths and Weaknesses

Runtime verification is a framework to check programs for correctness. Being a formal method, runtime verification provides a **rigorous and automated** approach to **detect** bugs. However, since monitoring tools work on an abstraction of an execution, they cannot provide a full explanation of a defect. Information that is provided is limited to the events of the trace. The complete **internal state** of the program being checked is **not available**. This hidden state may be essential to understand the cause of a bug. Moreover, despite the domain gaining traction and the method being increasingly adopted by the industry as well as the in the research community, runtime verification tools are still **not present by default in the developer toolbox** and no mainstream Integrated Development Environment features runtime verification for checking correctness of the programs being developed.

2.3 Combining Interactive Debugging and Runtime Verification

Interactive debugging and runtime verification both are interesting and useful approaches to tackle software defects. These approaches both have their strengths and limitations, summarized in the following table.

Feature	**Interactive Debugging**	**Runtime Verification**
Widespread	Yes	No
Access to the internal state	Yes	No
Modify the execution state	Yes	No
Detection	No	Yes
Rigor	No	Yes
Automation	No	Yes

In this work, we present an approach, interactive runtime verification, that aims to combine their strengths and overcome their limitations. This combination requires unifying a number of concepts. In this section, we describe how we adapt the two approaches to each other.

Wish list. While building our approach, our requirement is to keep the good parts of each approach. Interactive debugging provides powerful means to study the program *interactively*, with a *high precision* level. Access to the internal state of the program is *comprehensive*: every value can be observed. The state of the program can be *altered* (values can be changed), saved and restored. Interactive debugging enables program study and exploration by *fiddling*.

Runtime verification provides a *high-level*, global picture, abstracted view of the program execution. It constitutes an *automated* way to *detect faults*. Its *formal* aspect is important as well.

Debugger-Based Instrumentation and Signal-like events. While not designed for this goal, we use the interactive debugger as an instrumentation platform. Using an interactive debugger, a developer usually observes the state of the program, at a point of interest, when the program is suspended. The execution is suspended using features like breakpoints, watchpoints, catchpoints and any feature allowing to suspend the execution of the program on a specific event (the specified function was called, line was reached, variable was accessed, UNIX signal was received, system call was performed). We base our instrumentation on these features. In the remainder of this thesis, we will call these features *points*. They allow producing signal-like events (see Section 2.2.1.2).

Parameterized events. Properties that we want to be able to check also depend on the values of variables in the program. For instance, checking a property on the good usage of a queue (no overflow happens) requires knowing the size of the queue. When the program execution is suspended, debuggers allow observing values of variables present in the current frame (local variables and function arguments), in the call stack, values of global variables and more generally, any value present in the current program state. This allows attaching values of interest to events, which makes them a *partial description of the current state.*

Sequences of parameterized signals. In sequential programs, events are naturally totally ordered. Some interactive debuggers handle parallel programs. By default, they suspend the whole execution when reaching a point (breakpoint, watchpoint, catchpoint), forcing a total order on the events.

Remark 2.3.0.1 (Non-stop mode)**.** Some interactive debugger also provide a mode in which only the thread in which a point was reached is suspended, letting other threads run until a point is reached. This mode is called Non-stop mode in GDB[10]. We did not explore this mode, which is therefore out of the scope of this thesis. However, by their design, current interactive debuggers force a total order on events anyway.

This total order makes the trace of one program execution a sequence. In this thesis, we will also consider systems composed of several programs, each producing its own sequence.

Blocking, Dynamic Instrumentation. Contrary to most instrumentation techniques, debugger-based instrumentation is dynamic: it can be added or removed as-needed. It is also blocking: once reached, the execution of the whole program is suspended until explicitly resumed. Because of this blocking aspect, and because breakpoints and other debugger-based instrumentation features involve traps to the privileged kernel code and system calls, the instrumentation is costly. Not having unnecessary breakpoints is desirable, and removing any unused instrumentation (producing events that do not make the monitor change state) is therefore useful. Setting instrumentation specific to the property being checked requires a mechanism allowing the monitor to advertise what events are relevant to the property, so points can be added to produce these events. Another mechanism should resume the execution when the monitor is ready.

Reaction to Verdicts. Having mechanisms to know what events to instrument and to resume the suspended execution is not enough. There needs to be a way to use the verdicts issued by the monitor. Since the instrumentation is blocking, the program can be altered using features provided by the interactive debugger before resuming the execution. It can also be decided to leave the program suspended and hand over the control of its execution to the developer. We call such decisions *reactions.*

In the next section, we describe our approach, including the definition and handling of these reactions, in more details.

2.4 Overview of Interactive Runtime Verification

In the previous sections, we presented the two approaches, interactive debugging and runtime verification, on which interactive runtime verification is based, and their main features. We then presented important considerations and features to preserve when combining them. In this section, we give an overview of interactive runtime verification, building upon these considerations.

i-RV aims to ease 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

[10] `https://sourceware.org/gdb/onlinedocs/gdb/Non_002dStop-Mode.html`

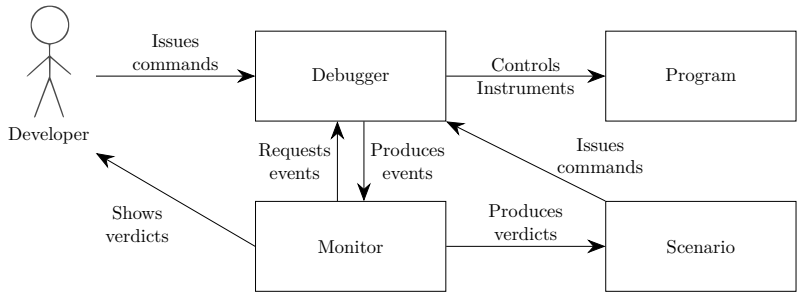


Figure 2.7: Overview of Interactive Runtime Verification. The developer issues commands to the debugger, the debugger controls and instruments the program, the monitor requests program events to the debugger and issues verdicts to the scenario and the scenario applies developer-defined reactions.

augmenting debuggers with runtime verification, combining a practical debugging method with a formal approach.

Gathering interactive debugging and runtime verification is challenging for several reasons. To the best of our knowledge, there are no formalizations for interactive debugging suitable for using interactive debuggers as instrumentation platforms for runtime verification. 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, and interactive debugging as a means to react to verdicts emitted by monitors. 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 an instrumentation platform 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.

In i-RV (Figure 2.7), 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 produces a sequence of verdicts. See Figure 2.5 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 scenario provided by the developer defines what actions should be taken during the execution according to the evaluation of the property, in reaction to the verdicts. Examples of scenarios are:

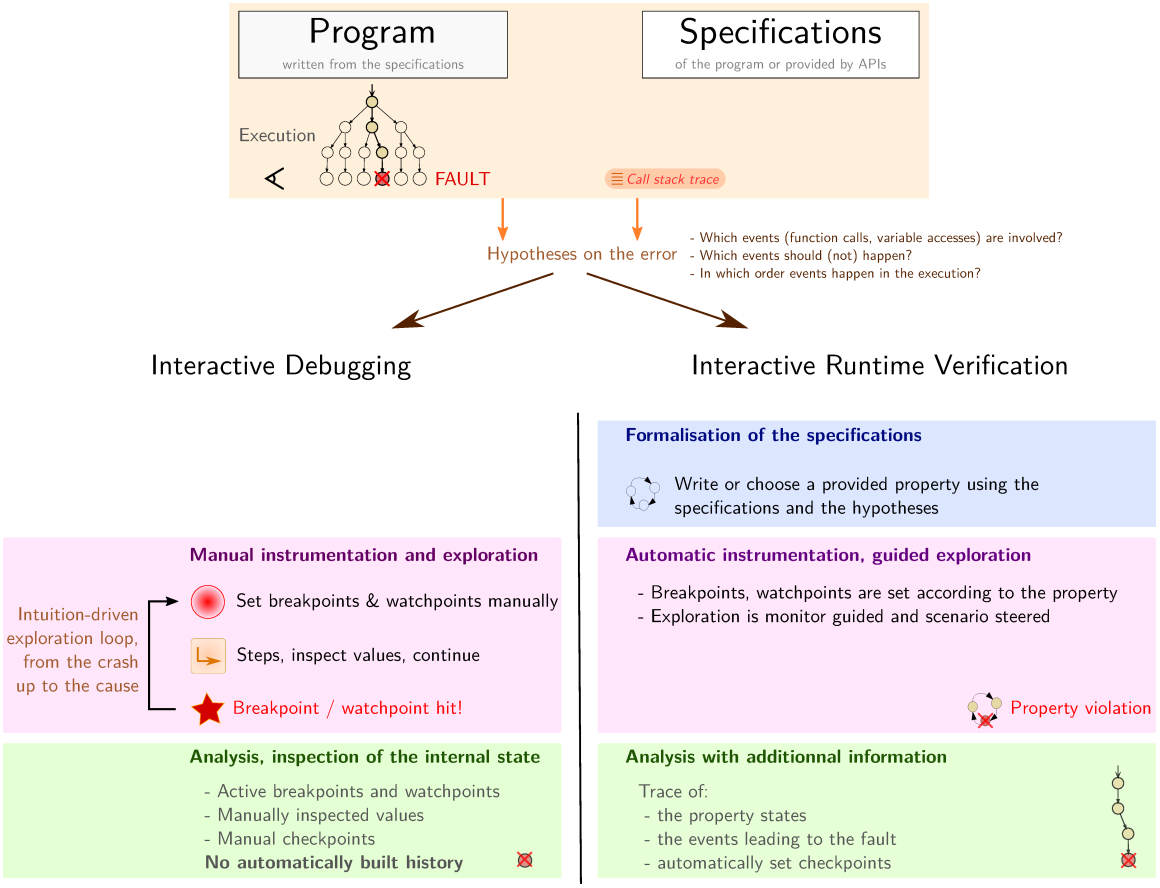


Figure 2.8: Interactive Debugging versus Interactive Runtime Verification.

– when the verdict given by the monitor becomes false (e.g., when the queue overflows), the execution is suspended to let the developer inspect and debug the program in the usual way, interactively;
– save the current 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”.

2.5 Comparing I-RV and Interactive Debugging Sessions

We give an intuition of how interactive runtime verification works by comparing an interactive runtime verification session with a traditional interactive debugging session. Figure 2.8 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 an 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.).

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, part of the program specification, formally expresses an expected behavior of the program. 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.

Illustrative example. We presented a traditional interactive debugging session using the sample C program depicted in Figure 2.1 in Section 2.1.1. We present an interactive runtime verification session with the same program.

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 Figure 2.5 (the queue should not overflow). When the program is interactively runtime verified with this property[11], 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 the interactive debugging session and the interactive runtime verification session, 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 which may have security implication. In C, array accesses are not runtime-checked, and thus not easily detected. 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.

[11] With Verde: `verde -prop queue.prop ./faulty`, where the contents of file `queue.prop` is given in Figure 5.6. A session is reproduced in Figure 2.9. The program is compiled with `gcc -g3`.

```
$ 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
```

Figure 2.9: Interactively runtime verifying the faulty C program. Compared to the interactive debugging session depicted in Figure 2.3, almost all interactions between the debugger and the developer are avoided.

2.6 Conclusion

In this chapter, we presented interactive debugging and runtime verification, the two basic blocks of interactive runtime verification is based. We presented the important features of these basic blocks, and argued that they are complementary. We presented the important considerations to tackle when combining them. We then gave an overview of interactive runtime verification, which is our way of combining them while tackling these considerations. We then illustrated interactive runtime verification by comparing an i-RV session with an interactive debugging session. In Chapter 3, we present a framework for interactive runtime verification suitable for monoprocess programs.

Chapter 3

Interactive Runtime Verification for Monoprocess Programs

Contents

In this chapter, we present a first approach to interactive runtime verification (i-RV). This approach is suitable for handling programs composed of one process. 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 Section 3.2.1. 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 to help building an actual implementation of i-RV. Developers using the approach are not required to have a full understanding of these descriptions.

Chapter organization. In Section 3.1, we define some notations used in this chapter. In Section 3.2, we define some notions related to interactive runtime verification. In Section 3.3, we give an operational view of our model. In Section 3.4, we give guarantees on our model. Proofs of these guarantees are given in Appendix A.1. In Section 3.5, we present an algorithmic view of our model, suitable for building an implementation. We conclude the chapter in Section 3.6.

3.1 Notations and Definitions

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

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

Definition 3.1.1.1: 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 $E_1, \ldots, E_n$) and a bijection $f_{\mathrm{ind}} : \{1, \ldots, n\} \to \mathrm{Name}$ 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 3.1.1.1. In the remainder of the thesis, given a bijection f_{ind}, we use a tuple t and the corresponding named tuple $r = (t, f_{\mathrm{ind}})$ interchangeably.

3.1.2 Notation and Notions Related to Labeled Transition Systems

Labeled transition systems. In Chapter 3, we model the components involved in interactive runtime verification as Labeled Transition Systems (LTSs) and Input-Output Labeled Transition Systems (IOLTS).

Definition 3.1.2.1: 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 3.1.2.2: 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:

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

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 $-$.
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 weak simulation relation [Mil89] to express correctness properties on our models.

Definition 3.1.2.3: Weak simulation

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. LTS S_1 weakly simulates LTS S_2 if there exists a relation $R \subseteq Q_1 \times Q_2$ such that:

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

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

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

(a) 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

(b) 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 $\overline{Obs}$.

3.2 Notions

In this section, we give a formalization of programs, events, and some concepts related to debuggers.

3.2.1 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, programs do not communicate with the outside, do not read user input and are 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 widely used in today's

operating systems, it is usually disabled by interactive debuggers like GDB and LLDB by default to ease debugging. 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[1] 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 3.2.1.1: Value, address, memory

Val 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. We denote the set of addresses in a program by Addr $\subseteq$ Val. A memory maps addresses to values. Mem $\stackrel{\text{def}}{=}$ [Addr $\to$ Val] is the set of memories.

Definition 3.2.1.2: Access

Access $\stackrel{\text{def}}{=}$ Addr $\times \{\mathtt{r}, \mathtt{w}\} \times$ Val $\times$ Val is the set of memory accesses. An element ($addr$, $mode$, old, new) $\in$ Access represents a read access at address $addr$ if $mode = \mathtt{r}$, or a write access to address $addr$ if $mode = \mathtt{w}$. 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).

Definition 3.2.1.3: Symbol

A symbol is the name of a variable or a function. Sym is the set of symbols used in a program. Symbols are linked to addresses in a program using a symbol table.

Definition 3.2.1.4: Parameter

A grammar describing the set of valid parameters Param is given in Figure 3.1. A parameter in Param is: either $v \in$ Sym (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 3.2.1.1 (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.

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

Definition 3.2.1.5: Symbol table

A symbol table symT is used to get the address of an object in the program memory at runtime from its description. symT $\in$ SymbolTable $\stackrel{\text{def}}{=}$ Mem $\times$ pc $\times$ Param $\rightarrow$ 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.

Remark 3.2.1.1. Symbols are resolved using the symbol table, built at compilation time, at runtime. Some symbols, such as global variables, may be resolved statically. However, local variables have a stack-dependant position. To be as general as possible, we chose to represent the symbol table as a function taking the complete information about the execution. The symbol table is meant to represent the behavior of a debugging data format such as DWARF [DWA], a widespread and standardized debugging data format which is Turing complete.

Definition 3.2.1.6: Program

A *program* is a 5-tuple $(\text{symT}, m_p^0, \text{start}, \text{runInstr}, \text{getAccesses})$ where:

- symT $\in$ 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) $\rightarrow$ (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) $\rightarrow$ Accesses* is a function returning the sequence of accesses that will be made when running the next instruction.

Example 3.2.1.2 (Program)**.** In the remainder of this section, we will use program P given by the following source code to illustrate the concepts:

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

Definition 3.2.1.7: Configuration of the program

A configuration of the program is a 2-tuple $(\mathtt{m} \mapsto m, \mathtt{pc} \mapsto pc) \in \text{Conf}_\text{p} \stackrel{\text{def}}{=}$ 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;
- 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 3.2.1.3 (Configuration of the program)**.** For program P given in Example 3.2.1.2, just after the execution of the second instruction, the configuration of the program is (m_p, pc_3) where pc_3 is the address of the code that corresponds to the third instruction of P, $m_p[\text{symT}(m_p, pc, a)] = 0$ and $m_p[\text{symT}(m_p, pc, b)] = 1$.

Definition 3.2.1.8: 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.

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

Figure 3.1: Grammar of valid parameter names.

Example 3.2.1.4 (Program checkpoint)**.** For the program given in Example 3.2.1.2, 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.

3.2.2 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 3.2.2.1: Symbolic event

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

- $t \in \text{EventTypes} = \{\texttt{Reach}, \texttt{ValueRead}, \texttt{ValueWrite}, \texttt{ValueAccess}\}$ is the type of event: an instruction is reached (e.g., a function call), a value read, a value write or a value access (read or write).
- $n \in \text{Sym}$ 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 3.2.2.1 (Symbolic event)**.** $(\texttt{Reach}, \texttt{push}, (q, v))$ is an event triggered when function `push` is called. Parameters q and v are retrieved when producing the event.

Definition 3.2.2.2: Runtime event

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

Example 3.2.2.2 (Runtime event)**.** $(\texttt{Reach}, \texttt{push}, (0x5650653c4260, 42))$ 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 3.2.2.1. In practice, `Reach` events are captured using breakpoints and `ValueWrite`, `ValueRead` and `ValueAccess` are captured using watchpoints.

Remark 3.2.2.2. Current interactive 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, though implementation can use it.

3.2.3 Instrumentation Provided by the Debugger and Event Handling

The interactive 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 $a \in Address$ and a watchpoint when a given address containing data of interest is accessed (read, written, or both).

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 [Ram94].

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. In the following, we define breakpoints and watchpoints as represented in an interactive debugger. In these structures, the origin of the point (the developer, the monitor or the scenario) is stored.

Remark 3.2.3.1. In an actual interactive debugger, not especially designed for i-RV, the information about the origin of the point is limited to whether the point was set by the developer, or programmatically. However, for simplicity, we assume the interactive debugger is able to know whether a point set programmatically was set by the monitor or the scenario. This information can be maintained by an i-RV-compatible implementation (e.g., by a regular interactive debugger that has been extended to support i-RV).

Definition 3.2.3.1: Breakpoint

A breakpoint is a named tuple ($\texttt{addr} \mapsto addr,\ \texttt{for} \mapsto f$), 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: $\text{Bp} \stackrel{\text{def}}{=} \text{Addr} \times \{\texttt{dev}, \texttt{scn}, \texttt{evt}\}$.

Example 3.2.3.1 (Breakpoint)**.** A breakpoint set by the developer on the second instruction of the program given in Example 3.2.1.2 is $(pc_2, \texttt{dev})$ 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.

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.

Definition 3.2.3.2: Watchpoint

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

Example 3.2.3.2 (Watchpoint)**.** A watchpoint set by the developer on variable `b` in the program given in Example 3.2.1.2 is $(\&b, \texttt{w})$, 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).

$$\text{evt2pts}(m, pc, \text{symT}, e) = \begin{cases} \{(\text{symT}(m, pc, e.\texttt{name}), \{\texttt{r}\}, \texttt{evt})\} & \text{if } e.\texttt{type} = \texttt{ValueRead} \\ \{(\text{symT}(m, pc, e.\texttt{name}), \{\texttt{w}\}, \texttt{evt})\} & \text{if } e.\texttt{type} = \texttt{ValueWrite} \\ \{(\text{symT}(m, pc, e.\texttt{name}), \{\texttt{r}, \texttt{w}\}, \texttt{evt})\} & \text{if } e.\texttt{type} = \texttt{ValueAccess} \\ \{(\text{symT}(m, pc, e.\texttt{name}), \texttt{evt})\} & \text{if } e.\texttt{type} = \texttt{Reach} \end{cases}$$

Figure 3.2: Event instrumentation: definition of function evt2pts.

Remark 3.2.3.2 (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.

Event Instrumentation and Instantiation. We denote the set of breakpoints and watchpoints by Point = Bp∪Wp. An element of Point is referred to as a point. We present the relations between the instrumentation provided by the debugger and events.

In Figure 3.2, we define function evt2pts : Mem × Addr × SymbolTable × SymbolicEvent → $\mathcal{P}$(Point) that maps a memory, a program counter, a symbol table and an event to a set of points.

We define instantiate : SymbolTable × SymbolicEvent × Mem × Addr, the function which maps a symbol table, a symbolic event, a memory and a program counter to a runtime event.

$$\text{instantiate}(\text{symT}, e_f, m, pc) = (e_f, l) \text{ with } l_k = m[\text{symT}(m, pc, e_f.\texttt{params}_k)]$$

That is, at runtime, a symbolic event is instantiated by getting values of its parameters from the current memory using the symbol table.

3.3 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 does not depend on 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 does not depend on the other components neither. 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 and the program behavior independently is not meaningful. We describe the behavior of the program under debug (or execution controller).

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

Table 3.1: 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
ckpt	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

3.3.1 Interface of the Program Under Interactive Runtime Verification

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.

3.3.1.1 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 execution controller. The set of debugger commands is:

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

The usage of these commands is detailed in Table 3.1. 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 $I_{\text{i-RV}} \stackrel{\text{def}}{=} \{\texttt{INIT}\} \cup \text{DbgCmd} \cup \{\text{scnCmd(c)} \mid c \in \text{DbgCmd} \cup \{\text{nop}\}\}$.

3.3.1.2 Output Symbols

The i-RV-program outputs values (in Val) 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 execution controller 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 $O_{\text{i-RV}} = \text{Val} \cup (\mathbb{N} \times \mathbb{N})$.

3.3.2 The Program

In this section, we present the behavior of a program.

Definition 3.3.2.1: Operational semantics of a program

Let $Pgrm(\text{symT}, m_p^0, \text{start}, \text{runInstr}, \text{getAccesses})$ be a program following Definition 3.2.1 (see Section 3.2.1, page 30). The operational semantics of $Pgrm$ is the IOLTS $(\text{Conf}_\text{P}, \{\text{EXEC}(m, a) \mid m \in \text{Mem} \wedge a \in \text{Addr}\}, \emptyset, \{\texttt{TRAP}\}, \rightarrow_\text{P})$ where Conf_P is the set of configurations as per Definition 3.2.1 and $(\texttt{m} \mapsto m_p^0, \texttt{pc} \mapsto \text{start})$ is the initial configuration, and $\rightarrow_\text{P}$ is the least set of transitions abiding by the rules in Figure 3.3.

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

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

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

Figure 3.3: Execution of the program.

is output when the execution reaches a breakpoint. Recall that we do not model the standard input and output nor the communications with the outside. The evolution of configuration is given by relation $\rightarrow_{\text{P}}$ which follows the two rules given in Figure 3.3:

- 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 $(\texttt{m} \mapsto m, \texttt{pc} \mapsto pc)$ 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 3.3.2.1. The configuration of the program remains unchanged when encountering a breakpoint instruction. The execution controller temporarily replaces the breakpoint instruction by the original instruction in the program when this instruction is to be executed (see Section 3.3.3). 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.

3.3.3 The Execution Controller

In this section, we present the behavior of the execution controller, which models the execution of a program with a debugger. We consider a program $Pgrm(\text{symT}, m_p^0, \text{start}, \text{runInstr}, \text{getAccesses})$ following Definition 3.2.1 (see Section 3.2.1, page 30) with initial configuration $P_0 \in \text{Conf}_{\text{P}}$.

The execution controller can receive debugger commands in the set DbgCmd defined in Section 3.3.1.

Definition 3.3.3.1: Operational semantics of an execution controller

The operational semantics of the execution controller *Prgm* is the IOLTS $(\text{Conf}_\text{P} \times \text{Conf}_\text{D}, A_\text{EC}, I_\text{EC}, O_\text{EC}, \rightarrow_\text{EC})$ 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;
  - a configuration of the debugger in $\text{Conf}_\text{D} = \{\textsc{p}, \textsc{i}\} \times \mathcal{P}(\text{Bp}) \times \mathcal{P}(\text{Wp}) \times \text{Cp}^* \times [\text{Point} \rightarrow \text{Event}] \times \mathcal{P}(\text{Point}) \times \text{Mem}$;
- A_EC is the set of actions defined as $\{\textsc{setBreak}(b), \textsc{rmBreak}(b), \textsc{setWatch}(\text{w}), \textsc{rmWatch}(w), \textsc{devBreak}, \textsc{scnBreak}(b), \textsc{evtBreak}(e), \textsc{devWatch}, \textsc{scnWatch}(w), \textsc{evtWatch}(e), \textsc{int}, \textsc{cont}, \textsc{trapNoBreak}, \textsc{clearEvents}, \textsc{setSym}(s, v), \textsc{setAddr}(a), \textsc{setPC}(a), \textsc{checkpoint}(n), \textsc{restore}(n), \textsc{exec}(m, a) \mid w \in \text{Wp} \wedge b \in \text{Bp} \wedge e \in \text{Event} \wedge n \in \mathbb{N} \wedge m \in \text{Mem} \wedge a \in \text{Addr}\}$,
- $I_\text{EC} \stackrel{\text{def}}{=} \text{DbgCmd} \cup \{\text{instr}(evts) \mid evts \in \mathcal{P}(\text{Event})\} \cup \{\text{clearEvts}\}$ 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
- $O_\text{EC} \stackrel{\text{def}}{=} \mathcal{P}(\text{Point}) \cup \mathcal{P}(\text{Event}) \cup \text{Val} \cup \mathbb{N}$ is the set of outputs.
- $\rightarrow_\text{EC}$ is the least set of transitions defined by semantics rules described later in the next paragraph.

The initial configuration of the execution controller is $((\texttt{mode} \mapsto \textsc{i}, \texttt{bpts} \mapsto \epsilon, \texttt{wpts} \mapsto \epsilon, \texttt{cpts} \mapsto \epsilon, \texttt{evts} \mapsto \emptyset, \texttt{hdld} \mapsto \emptyset, \texttt{oi} \mapsto (addr \mapsto \texttt{BREAK})), P_0)$.

The execution controller receives commands from the developer or the scenario. It also receives 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:

- $mode \in \{\textsc{p}, \textsc{i}\}$ indicates the mode of the debugger. In passive mode ($\textsc{p}$), 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 ($\textsc{i}$), 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.
- $\mathscr{C} \in \text{Cp}^*$ is the list of checkpoints saved in the debugger. A checkpoint $c \in \text{Cp}$ 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 execution controller, 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.
- $oi \in \text{Mem}$ (for *o*riginal *i*nstructions) is a mapping that stores instructions in the program memory that have been replaced by breakpoint instructions.

An output of the execution controller can be either: a set of points $pts \in \mathcal{P}(\text{Point})$ for the scenario, a set of events $events \in \mathcal{P}(\text{Event})$ for the monitor, a value $v \in \text{Val}$ 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 execution controller). The behavior of the execution controller 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 execution controller triggers this point. When

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

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

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

Figure 3.4: Handling watchpoints.

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 for 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, producing an event, producing 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 Figure 3.4 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 × Wp → $\mathbb{B}$ tests whether an access matches a watchpoint and is defined as $\text{match}(a, w) = (a.\texttt{addr} = w.\texttt{addr}) \wedge a.\texttt{mode} \in w.\texttt{mode}$, for any $a \in \text{Access}, w \in \text{Wp}$.

- Rule devWatch applies when a developer watchpoint is reached. The execution controller performs action devWatch. The list of accesses 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

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

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

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

$$\textsc{trapNoBreak}\ \frac{P \xrightarrow{-\ /\ \texttt{TRAP}} P \qquad pc' = D.\texttt{oi}(pc)}{\left(P\left\langle\begin{array}{l}\texttt{m} \mapsto m,\\ \texttt{pc} \mapsto pc\end{array}\right\rangle, D\left\langle\begin{array}{l}\texttt{mode} \mapsto \mathbf{P}, \texttt{hdld} \mapsto h,\\ \texttt{bpts} \mapsto \mathscr{B}\end{array}\right\rangle\right) \xrightarrow{-\ /\ -} (P[\texttt{m} \mapsto m[pc \mapsto pc']], D)}$$

Figure 3.5: Handling breakpoints.

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 : $\mathrm{Bp}^* \times \mathrm{Addr} \times \mathrm{Mem} \to \mathrm{Mem} \times \mathrm{Mem}$ defined as

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

- Rule SCNWATCH applies when rule DEVWATCH does not apply, and thus a scenario watchpoint can be triggered. The execution controller performs action SCNWATCH(w), where w is the produced 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 execution controller performs action EVTWATCH(e), where e is the produced event.

We now present rules in Figure 3.5 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 Figure 3.4 do not apply). An execution step of the program outputs symbol `TRAP`.

- Rule DEVBREAK If a developer breakpoint is set on address *pc*, the debugger is set to interactive

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

Figure 3.6: Normal execution and end of execution.

mode (I) and the breakpoint is marked as handled (rule DEVBREAK). The execution controller 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 execution controller 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 Figure 3.8). 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 Figure 3.6 applies. The program memory is uninstrumented using function unInstr : $\text{Mem} \times \text{Bp}^* \times \text{Mem} \to \text{Mem}$ defined as $\text{unInstr}(M, \mathscr{B}, oi) = m \dagger \{b.\texttt{addr} \mapsto oi(b.\texttt{addr}) \mid b \in \mathscr{B}\}$. One step of the program is executed. breakpoint instructions are restored using function restoreBPs : $\text{Bp}^* \times \text{Mem} \to \text{Mem} \times \text{Mem}$ defined as $\text{restoreBPs}(\mathscr{B}, m) = (m \dagger \{b.\texttt{addr} \mapsto \texttt{BREAK} \mid b \in \mathscr{B}\}, [b \mapsto \texttt{BREAK}\ \forall b \in \text{Addr}] \dagger \{b.\texttt{addr} \mapsto m[b.\texttt{addr}] \mid b \in \mathscr{B}\})$. The list of handled points is emptied. When the program ends, event STOP is output. When the rule applies, the execution controller performs action $m[pc]$ such that $(P\langle \texttt{m} \mapsto m, \texttt{pc} \mapsto pc\rangle, D)$ is the configuration of the execution controller to which the rule is applied.

Remark 3.3.3.1. 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 execution controller in Figure 3.7.

- 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).

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

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

Figure 3.7: Instrumenting events.

- Rule INSTRUMENT. The execution controller 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*. Function watchEvents is such that watchEvents(P, *evts*, ϵ) = ($\emptyset$, *evts*) and watchEvents(P, *evts*, $e \cdot l$) = ($pts \cup pts'$, $evts' \dagger \{p \mapsto e \mid p \in pts\}$) where pts = evt2pts(P.m, P.pc, symT, e) and (pts', $evts'$) = watchEvents(P, *evts*, l). Points are added to the execution controller using rule SETBREAK or rule SETWATCH, given in Figure 3.8 and described later in this section.

We now present the rules provided by the execution controller to set and remove points in Figure 3.8.

- Rule SETBREAK is used to set a breakpoint, which consists in:
  - saving the instruction of the program memory at the breakpoint address in the debugger, if no breakpoint has been set at this location;
  - replacing this instruction by `BREAK`;
  - updating the set of breakpoints in the debugger.

  When the rule applies, the execution controller 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 execution controller 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 3.3.3.2. 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 Figure 3.8.

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

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

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

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

Figure 3.8: Setting breakpoints and watchpoints.

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

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

Figure 3.9: Checkpointing.

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

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

$$\textsc{int}\ \frac{}{(P, D) \xrightarrow{\text{INT}\ /\ -} (P, D[\texttt{mode} \mapsto \textbf{I}])}$$

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

Figure 3.10: Stepping and interrupting the execution.

- Rule CKPT 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 3.3.3.3. 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 Figure 3.10, related to stepping and mode switching.

- 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 execution controller 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 Figure 3.11.

- 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 symT
- Rules GETPC and SETPC. Rule GETPC (resp. rule SETPC) outputs (resp. sets) the program counter.

$$\textsc{setAddr}\ \frac{a \in \text{Addr}, v \in \text{Val}}{(P, D) \xrightarrow{\text{set}(a,v)\ /\ -} (P[\mathtt{m} \mapsto m[a \mapsto v]], D)}$$

$$\textsc{getAddr}\ \frac{}{(P\langle \mathtt{m} \mapsto m \rangle, D) \xrightarrow{\text{get}(a)\ /\ m[a]} (P, D)}$$

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

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

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

$$\textsc{getPC}\ \frac{}{(P\langle \mathtt{pc} \mapsto pc \rangle, D) \xrightarrow{\text{getPC}\ /\ pc} (P, D)}$$

Figure 3.11: Setting and getting values in the program.

3.3.4 The Monitor

In this section, we define the i-RV monitor. The monitor evaluates a property against a trace, giving a verdict upon the reception of each event. Our model does not depend on a particular specification formalism. We assume a monitor given in terms of a set of states Q, an initial state $q_{\mathtt{init}} \in Q$ an initial state, a transition function $\rightarrow_{\mathtt{mon}}$: $Q \times \text{Event} \rightarrow Q$ which updates the state of the monitor upon each event, and a function verdict : $Q \rightarrow$ Verdict which maps states to verdicts. Before composing the monitor with the execution controller, we augment its behavior as follows.

Definition 3.3.4.1: Operational semantics of a monitor

Let $Mon(Q, q_{\mathtt{init}}, \rightarrow_{\mathtt{mon}}, \text{verdict})$ be a monitor. The operational semantics of monitor Mon is the IOLTS $(Q \times Q^*, A_\text{M}, I_\text{M}, O_\text{M}, \rightarrow_M)$, where $Q \times Q^*$ is the set of configurations and:

- $A_\text{M} \stackrel{\text{def}}{=} \{\textsc{event}(e) \mid e \in \text{Event}\} \cup \{\textsc{ckpt}(n), \textsc{restore}(n) \mid n \in \mathbb{N}\}$ is the set of actions,
- $I_\text{M} \stackrel{\text{def}}{=} \text{Event} \cup \{\text{ckpt}\} \cup \{\text{restore}(n) \mid n \in \mathbb{N}\}$ is the set of input symbols,
- $O_\text{M} \stackrel{\text{def}}{=} \text{Verdict} \times \mathbb{N}$ is the set of output symbols,
- $\rightarrow_M$ is the transition relation defined as the least set of transitions abiding to the rules given in Figure 3.12.

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

In a configuration $(\mathtt{state} \mapsto q, \mathtt{cpts} \mapsto \mathscr{C})$ 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 ckpt used to set a checkpoint, or the symbol restore(n) is used to restore a checkpoint. An

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

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

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

Figure 3.12: Evolution of the monitor.

output symbol is either: a verdict $v \in \text{Verdict}$ 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 Figure 3.12.

- Upon the reception of an event, rule EVENT applies and the monitor performs action 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 : $Q \rightarrow \mathcal{P}(\text{SymbolicEvent})$ 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 to the new state.
- Rules CKPT and RESTORE describe checkpointing for the monitor. When these rules apply, the monitor performs actions CKPT(n) and RESTORE(n) respectively, where n is the index of the checkpoint being output. Rule CKPT 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.

3.3.5 The Scenario

In this section, we define the i-RV scenario. 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.

Remark 3.3.5.1. In practice, and in our implementation, the scenario may be described using a specific language to define 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 Section 3.3.1.

Definition 3.3.5.1: 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 the reception of a verdict from the monitor or a point from the debugger, the scenario updates its state and either output nothing, or a symbol in $I_{\text{i-RV}}$, which is a command for the i-RV program.
- In reply to a command sent to the i-RV program, the scenario receives a symbol scenarioReply(r) such that $r \in O_{\text{i-RV}}$. The scenario updates 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 every verdict from the monitor. For every configuration S of the scenario, for every verdict $v \in \text{Verdict}$, 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 $(S, v, S', c) \in \rightarrow_S$.
- The scenario handles every answer from the i-RV program. For every configuration S of the scenario, for every reply scnCmdReply(r) such that $r \in O_{\text{i-RV}}$, there should exist a command c for the i-RV program in $I_{\text{i-RV}} \cup \{\text{nop}\}$ and a new configuration S' such that (S, scnCmdReply(r), $S', c) \in \rightarrow_S$.
- The scenario handles every point from the execution controller. 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.

  For every configuration S of the scenario, for every 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 $(S, p, S', c) \in \rightarrow_S$.

3.3.6 The Interactively Verified Program

We consider an execution controller EC, a monitor M and a scenario S.

Definition 3.3.6.1: Operational semantics of the i-RV-program

Let us consider:
- a program $Pgrm(\text{symT}, m_p^0, \text{start}, \text{runInstr}, \text{getAccesses})$,
- a monitor $Mon(Q, q_{\texttt{init}}, \rightarrow_{\texttt{mon}}, \text{verdict})$, and
- a scenario Scn with a set of configurations Conf_S, a set of actions A_S and a transition relation $\rightarrow_S$.

The operational semantics of the program under interactive runtime verification (the i-RV-program) $Irv(Pgrm, Mon, Scn)$ is an IOLTS $(\text{Conf}_{\text{i-RV}}, A_{\text{i-RV}}, I_{\text{i-RV}}, O_{\text{i-RV}}, \rightarrow_{i-RV})$ where:
- $A_{\text{i-RV}} = A_\text{EC} \cup A_\text{M} \cup A_\text{S}$ is the set of actions of the i-RV-program. Actions happening in the i-RV-program are actions happening in the execution controller, the monitor or the scenario.
- $\text{Conf}_{\text{i-RV}} = \text{Conf}_\text{P} \times \text{Conf}_\text{D} \times \text{Conf}_\text{M} \times \text{Conf}_\text{S}$ is the set of configurations of the i-RV-program.

The sets of input symbols $I_{\text{i-RV}}$ and of output symbols $O_{\text{i-RV}}$ are defined in Section 3.3.1. We define the transition relation $\rightarrow_{i-RV}$ in the remaining of this section.

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

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

Figure 3.13: Intialization and execution.

3.3.6.1 Initialisation and Execution

In Figure 3.13, we describe the initialization and an execution step of the interactively verified program.

Rule INIT (Figure 3.13) initializes the i-RV-program. Upon the 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 execution controller for instrumentation and the verdict is transmitted the scenario. The scenario issues a command that is run by the i-RV-program.

Rule NORMALEXEC (Figure 3.13) does one execution step. During this step, no scenario or monitor point is triggered. After application of this rule, the execution controller may, or may not, be suspended by a developer point.

3.3.6.2 Events and Non-Developer Points

In Figure 3.14, we describe rules that trigger events for the monitor and non-developer points for the scenario.

Rules EXECPOINT and EXECEVENT (Figure 3.14) trigger a non-developer point. In rule EXECPOINT, a point is output by the execution controller. 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 execution controller. 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 execution controller for instrumentation. The verdict is forwarded to the scenario. The scenario outputs a command. The command is executed by the i-RV-program.

Rules STEPPOINT and STEPEVENT apply whenever command step is used and the execution controller outputs a point or an event.

3.3.6.3 Command From the Scenario

Rules SCNCMDNOP and SCNCMD (Figure 3.15) 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.

3.3.6.4 Stepping, Checkpointing and Other Debugger Commands

Rules in Figure 3.16 handle the debugger commands. Rule CKPT handles command checkpoint. The execution controller is checkpointed, the monitor is checkpointed, and the i-RV-program outputs the index of the resulting checkpoint by combining indexes generated by the execution controller and the monitor.

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

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

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

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

Figure 3.14: Events and non-developer points.

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

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

Figure 3.15: Command from the scenario.

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

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

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

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

Figure 3.16: Stepping, Checkpointing and Other Debugger Commands.

Rule CKPT handles command restore(n). The checkpoints of the execution controller and the monitor are restored.

Rule STEP handles command step whenever rules STEPPOINT and STEPEVENT do not apply, that is, no points or events will be output by the execution controller.

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 execution controller.

3.4 Correctness of Interactive Runtime Verification

We demonstrate the correctness of our models. Showing correctness is important because it allows ensuring that:

– any verdict found by monitors (and in particular those corresponding to a violation of the monitored property) are actual verdicts applying on the system;
– monitors do not miss verdicts.

3.4.1 Verifying the Behavior of the I-RV-Program

We show that the program and the corresponding i-RV-program are observationally equivalent. More precisely, given the behavior of the program and the i-RV-program defined by their LTS as per Section 3.3.2 and Section 3.3.6 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 program and the i-RV-program.

Definition 3.4.1.1: Relation between the configurations of the program and the i-RV-program

We consider a program *Prgm* and an i-RV-program *Irv*. Conf_P is the set of configurations of *Prgm* and $\text{Conf}_\text{P} \times \text{Conf}_\text{D} \times \text{Conf}_\text{M} \times \text{Conf}_\text{S}$ is the set of configurations of *Irv*.
Relation $\mathcal{R}$ between the configurations of *Prgm* and *Irv* is denoted by $\mathcal{R} \subseteq (\text{Conf}_\text{P} \times \text{Conf}_\text{D} \times \text{Conf}_\text{M} \times \text{Conf}_\text{S}) \times \text{Conf}_\text{P}$ and is defined as follows. For any configuration $(P_{\text{dbg}}, D, M, S)$ of *Irv* and for any configuration P of *Prgm*, $((P_{\text{dbg}}, D, M, S), P)$ is in $\mathcal{R}$ if the following equalities (3.1), (3.2) and (3.3) hold:

$$P.\texttt{m} = \text{unInstr}(P_{\text{dbg}}.\texttt{m}, D.\texttt{bpts}, D.\texttt{oi}) \tag{3.1}$$

$$P.\texttt{pc} = P_{\text{dbg}}.\texttt{pc} \tag{3.2}$$

$$\forall a \in \text{Addr}, P_{\text{dbg}}.\texttt{m}[a] \neq \texttt{BREAK} \implies P_{\text{dbg}}.\texttt{m}[a] = P.\texttt{m}[a] \tag{3.3}$$

The above equations can be understood as follows:

(3.1): removing the instrumentation from the i-RV-program memory ($\text{unInstr}(P_{\text{dbg}}.\texttt{m}, D.\texttt{bpts}, D.\texttt{oi})$) (see Section 3.3.3) results in the memory of the initial program ($P.\texttt{m}$);

(3.2): the program counters of the program ($P.\texttt{pc}$) and the i-RV-program ($P_{\text{dbg}}.\texttt{pc}$) are the same;

(3.3): at any address, the memory content in the i-RV-program ($P_{\text{dbg}}.\texttt{m}[a]$) is either a breakpoint or the memory content at the same address in the initial program ($P.\texttt{m}[a]$).

Intuitively, $\mathcal{R}$ relates configurations of an i-RV-program and its program as follows: if breakpoints are removed from the memory of the i-RV-program, the resulting memory and the program counter of the i-RV-program are equal to the memory and the program counter of the program.

Restrictions. We restrict this proof to cases where:

1. the interactive debugging features that modify the program are not used, by forbidding the use of rules that modify the program behavior, that is, rules RESTORE, SETSYM, SETADDR, SETPC and CKPT[2].
2. the scenario does not define any reaction (i.e., lets the execution continue under any circumstance). This is necessary, because scenario reactions are meant to be able to modify the program execution.

We define an observing scenario. An observing scenario does not modify the program behavior, that is, only uses debugger commands from this restricted set of debugger commands:

$$\begin{aligned}\text{DbgCmd}_{\texttt{obs}} \stackrel{\text{def}}{=} &\{\text{get}(s), \text{get}(a), \text{getPC}, \mid s \in \text{Sym} \wedge v \in \text{Val} \wedge a \in \text{Addr}\} \\ &\cup \{\text{setPoint}(p), \text{rmPoint}(p) \mid p \in \text{Point}\}\end{aligned} \tag{3.4}$$

Definition 3.4.1.2: Observing scenario

A scenario S is observing if, when receiving a verdict, a scenario command reply or a point, S outputs an element $c \in \text{DbgCmd}_{\texttt{obs}}$

Remark 3.4.1.1. $\text{DbgCmd}_{\texttt{obs}} = \text{DbgCmd} \setminus \{\{\text{set}(s, v), \text{set}(a, v), \text{setPC(a)} \mid s \in \text{Sym} \wedge v \in \text{Val} \wedge a \in \text{Addr} \cup \{\text{ckpt}, \text{continue}, \text{INT}, \text{step}\} \cup \{\text{restore}(n) \mid n \in \mathbb{N}\}\}$

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

[2]Using rule CKPT is meaningless without rule RESTORE. We also don't consider rules CKPT and RESTORE of the monitor for the same reasons.

Proposition 3.4.1.1: the program weakly simulates the i-RV-program

Let us consider:
- the transition relation $\rightarrow_P$ of the program;
- the transition relation $\rightarrow_{i-RV}$ of the i-RV-program where rules RESTORE; SETSYM, SETADDR, SETPC and rule CKPT are excluded;
- the set of observable actions $Obs = \{\text{EXEC}(m, a) \mid m \in \text{Mem} \ \wedge \ a \in \text{Addr}\}$;
- a program *Prgm*, a monitor *Mon* and an observing scenario *Scn* and the i-RV-program $Irv(Prgm, Mon, Scn)$.

Relation $\mathcal{R}$ between the configurations of *Irv* and *Prgm* (Definition 3.4.1) is a weak simulation as per Definition 3.1.2. That is, program *Prgm* weakly simulates *Irv*.

Moreover, the i-RV-program simulates the program, as stated by the following proposition.

Proposition 3.4.1.2: the i-RV-program weakly simulates the program

Let us consider:
- the transition relation $\rightarrow_P$ of the program;
- the transition relation $\rightarrow_{i-RV}$ of the i-RV-program where rules SETSYM, SETADDR, SETPC, RESTORE and rule CKPT are excluded;
- the set of observable actions $Obs = \text{Mem} \times \text{Addr}$;
- a program *Prgm*, a monitor *Mon*, an observing scenario *Scn* and the i-RV-program $Irv(Prgm, Mon, Scn)$;
- Relation $\mathcal{R}$ between the configurations of *Irv* and *Prgm*.

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

We prove Proposition 3.4.1 in Appendix A.1.1 and Proposition 3.4.1 in Appendix A.1.2.

3.4.2 Guarantees on Monitor Verdicts

Since the initial program and the i-RV-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 events that could be deduced using this information from the execution of the initial program can be produced from the execution of the i-RV-program assuming correct instrumentation (completeness) and any sequence of events produced by a correct instrumentation by the i-RV-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 i-RV-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 execution of the i-RV-program (defined in Section 3.2.3, page 32), 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.

3.5 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 Section 3.3. 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 (Section 3.5.1), then the execution controller (Section 3.5.2), the scenario (Section 3.5.3) and then the i-RV-program (Section 3.5.4). The i-RV-program drives the execution. It uses the execution controller, the monitor and the scenario as components.

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
```

3.5.1 The Program

As in Section 3.3.2, 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.

3.5.2 The Execution Controller

As in Section 3.3.3, the configuration of the execution controller 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 Section 3.5.4. P (resp. D) is a global variable referencing the configuration of the program (resp. the debugger).

3.5.2.1 Handling the Execution of an Instruction in the Program

In Algorithm 2, we describe an execution step of the execution controller. The program can only evolve in passive mode (D.`fieldMode` = `P`). First, the list of accesses done by the next instruction is computed (Line 10). These accesses are later matched in the algorithm (using function match defined in Section 3.3.3) with the set of watchpoints managed by the debugger and that have not been handled yet (Computed at Line 11).

- If a developer watchpoint matches (Line 12), the debugger is set to interactive mode (Line 15), which corresponds to rule DEVWATCH (Figure 3.4), and returns `OK` (Line 16). The watchpoint is added to the set of handled points in the debugger (Line 14).
- Otherwise, the list of accesses matching a scenario watchpoint is built (Line 2). If this list is not empty (Line 16), the watchpoint matching the first access of this list is returned (Line 19), which corresponds to rule SCNWATCH (Figure 3.4). The watchpoint is added to the set of handled points in the debugger (Line 7).
- Otherwise, the list of accesses matching a monitor watchpoint is built (Line 2). If this list is not empty (Line 21), the event corresponding to the watchpoint matching the first access of this list is returned (Line 22), which corresponds to rule MONWATCH (Figure 3.4). The watchpoint is added to the set of handled points in the debugger (Line 7).

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 6 and 13).

If no watchpoint is triggered, a program execution step is done (Line 23). The execution controller checks whether a breakpoint is triggered.

- If no breakpoint instruction is encountered, `OK` is returned (Line 24), which corresponds to rule NORMALEXEC (Figure 3.6), or `STOP` if the execution reaches the end of the program (Line 26).
- If a breakpoint instruction is encountered (Line 28), breakpoints are evaluated.

Algorithm 2 Handling the execution of an instruction in the program.

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

– If a developer breakpoint matches (Line 30), the debugger is set to interactive mode, which corresponds to rule DEVBREAK (Figure 3.5), and the breakpoint is added to the set of handled points.
– If a scenario breakpoint matches (Line 34), the breakpoint is returned, which corresponds to rule SCNBREAK (Figure 3.5), and the breakpoint is added to the set of handled points.
– If a monitor breakpoint matches (Line 37), the corresponding event is returned, which corresponds to rule EVTBREAK (Figure 3.5), and the breakpoint is added to the set of handled points.
– If no breakpoint matches (Line 40), the original instruction is temporarily restored in the program memory and `TRAP` is returned (which corresponds to rule TRAPNOBREAK (Figure 3.5)). This instruction will be executed or and the breakpoint instruction will be set back again.

Algorithm 3 Setting and removing points.

```
1:  function EC::SETBREAK(b ∈ Bp)
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 EC::RMBREAK(b ∈ Bp)
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 EC::SETWATCH(w ∈ D.wpts)
14:     D.wpts ← D.wpts ∪ {w}
15: function EC::RMWATCH(w ∈ D.wpts)
16:     D.wpts ← D.wpts \ {w}
17: function EC::SETPOINT(p ∈ Point)
18:     if p ∈ Bp then
19:         SETBREAK(p)
20:     else
21:         SETWATCH(p)
22: function EC::RMPOINT(p ∈ Point)
23:     if p ∈ Bp then
24:         RMBREAK(p)
25:     else
26:         RMWATCH(p)
```

3.5.2.2 Setting and Removing Points

In Algorithm 3, we define functions to set and remove points, corresponding to rules in Figure 3.8.
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 are no other breakpoints 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.

3.5.2.3 Instrumentation

In Algorithm 4, we define functions corresponding to rules CLEAREVENTS and INSTRUMENT defined in Figure 3.7. 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

Algorithm 4 Instrumentation.

```
1: function EC::CLEAREVENTS
2:     addrs ← {b.addr | b ∈ Dom(D.evts) ∩ Bp} \ {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 EC::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
```

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 Section 3.3.3. 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: $D.\texttt{mode} = \textsc{i}$

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

Precondition: $D.\texttt{mode} = \textsc{p}$

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

Precondition: $D.\texttt{mode} = \textsc{i}$

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

3.5.2.4 Stepping and Interrupting the Execution

In Algorithm 5, we describe commands STEP, INTERRUPT and CONTINUE, corresponding to rules in Figure 3.10.

Function STEP (Line 1) implements command step of the interactive mode of the debugger ($D.\texttt{mode} = \textsc{i}$). 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 (Figure 3.10, Section 3.3.3) is taken into account in function EXEC (Algorithm 2).

Algorithm 6 Controlling the program memory and counter.

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

3.5.2.5 Controlling the Program Memory and Counter

Functions in Algorithm 6 correspond to rules in Figure 3.11 and are used to set and get values in the program.

Algorithm 7 Checkpointing.

```
1: function EC::CKPT
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 EC::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:    EC.INSTRUMENT(events)
```

3.5.2.6 Checkpointing

Functions in Algorithm 7 are used to checkpoint and restore the execution controller. Function CKPT 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 3.5.2.1. 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 $D.\texttt{cpts}_{cid}$ given by the parameter identifier *cid* from the list of checkpoints $D.\texttt{cpts}$ 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.

3.5.3 The Scenario

We present an implementation of the scenario. In the operational view (Section 3.3.5), for the sake of generality, the scenario is specified generically. 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.

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.

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))
```

- If *cmd* is an action of the form setPoint(p, a') for some point p and some action a' (Line 7), command 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 execution contoller.
- 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.

3.5.4 The Interactively Runtime Verified Program

We consider an interactively verified program *ECMS* as defined in Section 3.3.6, using a scenario as defined in Section 3.3.5. 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 $(v, events) \in$ Verdict $\times \mathcal{P}$(SymbolicEvent) where v is a verdict and *events* is a set of symbolic events to track in the execution controller.

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 execution controller EC, 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 execution controller. We present the general behavior at the end of this section, in Algorithm 11.

3.5.4.1 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 event `INIT` is applied: in function APPLYEVENT (Line 3), the monitor is initialized (Line 4). Then, instrumentation for the initial state of the monitor is requested to the execution controller (Line 5). Finally, scenario for the initial verdict is applied (Line 6) and the command returned by the scenario is passed to function APPLYSCENARIOCMD (Line 7).

Function APPLYSCENARIOCMD (Line 18) executes a command (i.e., calls the function given in the command by its name with given parameters) at Line 21. The result of this command is forwarded to the scenario (Line 22), and the new command returned by the scenario is executed using function

Algorithm 9 Initialisation and execution.

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

applyScenarioCmd (Line 23). This recursive process ends when the scenario returns the special command nop (Line 19).

Function exec (Line 8) performs an execution step of the execution controller (Line 9). Making the program evolve requires the debugger to be in passive mode ($D.\texttt{fieldMode} = \textsc{p}$).

- If the execution controller returns a point (Line 11), this point is forwarded to the scenario. The scenario may return a command to perform. This command is executed using function applyScenarioCmd (Line 13).
- If the execution controller returns an event (Line 14), 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 execution controller (Line 5) and the verdict is forwarded to the scenario (Line 6). The scenario may return a command to perform. This command is executed using function applyScenarioCmd (Line 7).
- Any other value (`OK`, `TRAP`) returned by the execution controller is returned to the caller (Line 13).

Algorithm 10 Stepping, checkpointing and other debugger commands.

```
 1: function IRV::CKPT
 2:     return (EC.CKPT(), M.CKPT())
 3: function IRV::RESTORE((c_EC, c_M))
 4:     EC.RESTORE(c_EC)
 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
```

3.5.4.2 Stepping, Checkpointing and Other Debugger Commands

In Algorithm 10, we define functions ckpt, restore and step.

Function ckpt (Line 1) returns a checkpoint composed of a checkpoint of the execution controller and a checkpoint of the monitor.

Function restore (Line 3) restores the execution controller (Line 4) and the monitor (Line 5).

Function step (Line 6) implements command step of the interactive mode of the debugger ($D.\texttt{mode} = \textsc{i}$). It temporarily sets the execution controller 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).

Algorithm 11 General behavior of the i-RV-program.

```
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:                EC.CONTINUE()
14:            case set watch addr read write
15:                EC.SETWATCH((addr, {r | read} ∪
       {w | write}, dev))
16:            case set break addr
17:                EC.SETBREAK((addr, dev))
18:            case unset watch addr read write
19:                EC.RMWATCH((addr, {r | read} ∪
       {w | write}, dev))
20:            case unset break addr
21:                EC.RMBREAK((addr, dev))
22:            case print value at addr
23:                print(EC.GETADDR(addr))
24:            case set value v at addr
25:                EC.SETADDR(addr, v)
26:            case print value of symbol
27:                print(EC.GETSYM(symbol))
28:            case set value v of symbol
29:                EC.SETSYM(symbol, v)
30:            case print program counter
31:                print(EC.GETPC())
32:            case set program counter addr
33:                EC.SETPC(addr)
34:            case checkpoint
35:                print(CKPT())
36:            case restore cid
37:                RESTORE(cid)
```

3.5.4.3 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.

3.6 Conclusion

In this chapter, we presented a first framework for i-RV. We presented an architecture in which components are tightly integrated. We gave a operational semantics view of this architecture, in order to define precisely and provide a solid basis for reasoning on the expected behavior of the join execution of the different components. We gave an algorithmic view of the architecture, suitable for building implementations. We present such an implementation, Verde, in Section 5.1.

This approach handles programs composed of one process, and the evaluation of one property. In practice, implementations can allow multiple monitors. However, the behavior with respect to the scenario and the reactions to different verdicts is not well defined. Some programs are complex enough that the ability to evaluate several properties at the same time is desirable. We thus need to extend our architecture to allow several monitors. We also want to target entire distributed systems, composed of multiple processes. Debuggers do not allow debugging multiple processes at the same time in the general case. Some system are also composed of processes written in multiple programming langages, each requiring a different debugger, thus, multiple execution controllers.

In the architecture presented in this section, the number of components is fixed: one of each kind. Supporting multiple execution controllers and multiple monitors requires coordinating events coming from different execution controllers, event requests and verdicts from different monitors, and defining the behavior of the scenario in such a setting. Such support requires major adaptations. We need to move from a static architecture, in which the number of components is fixed, to a modular architecture in which any number of monitors and execution controllers can be used. Additionally, distributed systems may run on different computers. This requirement of modularity, as well as this last observation prompted us to design a network-based architecture, and a protocol to coordinate the communications between the monitors, the execution controllers and the scenario, which we present in Chapter 4.

Chapter 4

Distributed and Adaptative Interactive Runtime Verfication

Contents

4.1 Introduction

Distributed systems are notoriously hard to debug. They consists of multiple intercommunicating processes that may run on different computers and can be written in several programming languages. Trace-based approaches and approaches based on the analysis or the visualisation of the exchanged messages allow gathering knowledge to find the cause of a bug. Traces may be huge, making them hard to analyze, and require a lot of bandwidth. As the number of nodes in a distributed system grows, handling traces may even be harder. Work has been done to query and filter traces to manage this complexity. These approaches provide a good step toward understanding the cause of a bug: the developer may notice duplicated, missing or wrong messages, or inconsistant elements in the traces. However, traces also do not always contain sufficient information needed to study a bug. This information may be provided by inspecting the internal state of the nodes involved in the wrong behavior. For programs composed of a single process, interactive debugging is a widespread approach to inspect this internal state. However, interactive debugging as is may be inconvenient when dealing with a system composed of multiple processes. Each node needs to be run with an interactive debugger. The developer has to inspect the traces or the messages using a visualisation tool and decide when to manually suspend the execution of one or more nodes, which may not always be practical. An incorrect action may also happen in a node without immediately appearing in the trace. Such a silent action, as well as an incorrect message or element in a trace may be detected by evaluating the execution of the node or of a distributed system against a runtime property.

In Chapter 3, we presented a first approach to interactive runtime verification handling monoprocess programs. In this chapter, we extend interactive runtime verification to support multiprocess programs and programs that are subject to complex behaviors better described using multiple properties. These aspects raise two main challenges.

Handling concurrent executions. In the approach presented in Chapter 3, the execution is suspended as soon as an event is produced. The monitor consumes the event, produces a verdict and the scenario reacts accordingly while the execution is suspended before resuming the execution. Monitoring multiple processes implies that multiple events can be received concurrently by monitors. The scenario cannot control the whole system anymore: some processes may still be running. Moreover, interactive debuggers handle one process at a time. As such, debugging multiple processes entails managing several concurrent interactive debuggers. For instance, in a client-server application, there is one debugger per server and per client.

Reacting to verdicts from multiple properties. Several independent properties are checked concurrently, with a dedicated monitor per property. Contrary to the approach presented in Chapter 3, debuggers handle instrumentation requests for events from several monitors at once. Requests from different monitors can concern the same location in the code. As such, instrumentation may be shared between several monitors. One point of instrumentation may produce several events at once. Moreover, since monitors produce a verdict for each received event, the scenario can no longer react to verdicts, but should have their reaction defined over sets of verdicts. As a consequence, the scenario should expect all verdicts from the monitors that are concerned with the events produced by the system. When interactively runtime verifying multiple processes, the scenario needs to be aware of the processes involved in the production of a verdict to be able to send them commands to control their executions.

In this chapter, we will use the example of a leader election, using the Raft consensus algorithm, between three nodes: A, B and C. Consensus algorithms are used to set up clusters of replicated servers to handle fault tolerance. Raft is designed around the notion of terms. During each term, a leader election is organized. If the leader stops responding, a new election is organized, in a new term. Terms are associated with increasing numbers. We will consider the 4 following properties:

1. during each term, all processes in a cluster must agree on the same leader (agreement);
2. consensus is reached (i.e., all processes reach the highest known term; termination);

3. a process becomes leader only if a process voted for this process (integrity);
4. only one process becomes a leader (safety).

Summary. We introduce a framework for distributed interactive and adaptive runtime verification, bringing support for distributed systems and the concurrent verification of multiple properties, and allowing existing components such as monitors and debugger to be adapted for i-RV. In Section 4.2, we define the scope of our work and give design considerations for our architecture. In Section 4.3, we present an architecture and a protocol for distributed i-RV. The architecture fulfills the previously presented requirements, namely the support for multiple programming languages or debuggers, checking several properties at once, debugging a distributed system composed of multiple communicating multi-threaded programs spread across multiple computers. The architecture allows suspending the execution at the process level (not at the level of the entire system). The architecture provides means for monitors and properties to target each event request to a different set of programs. We provide a description of the behavior of each component of this architecture. In Section 4.4, we report on our modeling and model-checking of the protocol with SPIN.

4.2 Scope and Design Considerations

With distributed i-RV, we are interested in interactively verifying properties on systems composed of multiple processes. In this section, we detail the scope of our work. We detail our assumptions on the systems we handle. We them present the execution model we consider. We conclude by presenting requirements to consider and some design choices to be made for our framework.

4.2.1 Assumptions

We consider systems composed of multiple processes. Processes may run on different computers and be written using different technologies (libraries, programming languages). We make no assumptions on how processes communicate with each other. In the leader election example, we are not concerned about how the processes communicate and whether they use reliable channels. Messages may be lost as part of the election process and we may still obtain the desired result.

We also make no assumptions on the existence of a global clock or synchronisation point in the execution. We do not assume any observable global state and don't control the network. As a consequence, we may not freeze the whole system to inspect it. The set of processes we study may be part of a larger system that we do not study. In our leader election example, the processes may be part of a larger system, and processes communicate with the outside world, maybe serving requests from users, which we cannot freeze. Therefore, we may only suspend, control, inspect and instrument each process individually, while other processes may still be running.

4.2.2 Execution Model

In a system composed of one process monitored for one property, the trace that is produced by the instrumented process is the same as the trace received by the monitor. This trace is totally ordered. In a system composed of multiple processes, each process produces a trace. Each trace is totally ordered. However, events from different traces are not ordered. For instance, the property that function g is called in a process B before function f is called in a process A cannot be checked in the general case: the monitor checking the property may receive event g before event f. Should a new state be reached during the evaluation of a property if function f in process A is called and function g is called in process B, both orders need to be taken in account (e.g., the regular expression $(f \cdot g|g \cdot f)$ should be used). In the leader election example, an event from process A becoming the leader and then an event from process B setting the leader to A should be evaluated similarly when receiving the events in the reverse order.

However, in i-RV, each event production leads to the corresponding program to be suspended. The execution is resumed by the scenario after the monitor handles the event. In particular, for

function call events, programs are suspended at the beginning of the function. Consequently, if function f in process A causes function g to be called in process B (because A communicates with B when f is called), the monitor will receive event f before event g, and may verify this behavior using regular expression $f \cdot g$. The expected causal order of events in the distributed system needs to be known when writing properties for it.

Liveness properties are evaluated to true after a given (good) event happens. We note a restriction on how such properties may be used: reacting to the absence of this event in a process (resp. system) is not possible before the process (resp. the system) stops. In the leader election example, it may be possible for the developer to see whether property 2 (consensus is reached at some point) holds, but if the consensus is not reached, no event will make the scenario react.

4.2.3 Requirements and Design Choices

An architecture for distributed i-RV shall fulfil these requirements:

- is not specific to any programming language (to write the components of the architecture as well as for the program being debugged);
- handle a dynamic set of processes (processes may appear and disappear at any time);
- allow monitors to receive events from any process;
- allow monitors to target specific processes;
- allow a scenario to receive verdicts from the monitors;
- allow a scenario to control the execution of the processes;
- allow the processes to start only when the instrumentation is fully set up;
- allow a scenario to know the number of verdicts to expect at any time during the execution.

Centralized vs decentralized. A first design choice to make is whether a central point should be introduced. Introducing a central point introduces a single point of failure, and a potential point of bottleneck. It may also introduce synchronisation points and effect message ordering. It also simplifies the discovery and communication of the different components, making each component simpler. Two aspects may be centralized: the component discovery mechanism, and the communications. For simplicity, in the architecture presented in Section 4.3, we centralize both of these aspects in a component called the bus.

One or several monitors. One monitor component may suffise to check several properties by combining them. However, this requires defining the meaning of the composition at the monitor level. Information about the current state of each property is needed to precisely control the execution from the scenario. As a consequence, the composition of the properties should not abstract away the state of each property. The composition also gathers possibly unrelated properties together. Defining this composition may also be not trivial in the case the properties are expressed using different formalisms. Allowing several monitors is a trade-off between the complexity of a monitor and the complexity of the scenario. With one monitor, the scenario only ever needs to handle one verdict at a time. With several monitors, the scenario needs to handle sets of verdicts. Each monitor receives a specific subset of the traces produced by each execution controller specific to the property it verifies. In the architecture presented in the next section, we allow several monitors, so each monitor is as simple as possible and any decision is made by the scenario.

4.3 Architecture and Protocol for Distributed I-RV

In this section, we present the components of a distributed i-RV-session (see Figure 4.1). We then give an overview of the interactions between them. We then present the behavior and the interface of each component.

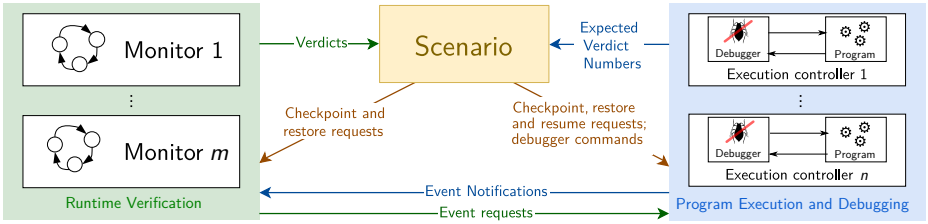


Figure 4.1: Architecture for Distributed I-RV.

4.3.1 Distributed I-RV Components

In this section, we briefly describe the role of each component.

Execution Controllers. Like in the framework presented in Chapter 3, an execution controller is the combination of a program and a debugger. The debugger is used for interactivity, as well as a means to instrument the program, produce events, and provide control over the execution. An execution controller grants event, checkpoint and restore requests as well as debugger commands from other components. It also acts as an interactive debugger that can be driven by the user. Event requests, sent by a component to receive a notification when an event occurs during the program execution, may only be done when the execution is suspended. At the beginning of a session, the execution needs to be suspended to allow initial event requests. Producing an event suspends the execution until a resume request is received. We present the behavior of an execution controller in more details in Section 4.3.3.

Monitors. A monitor checks the execution of the system under i-RV against a user-provided property, requests events to execution controllers, sends verdicts to the scenario, and grants checkpoint and restore requests from the scenario. We present the behavior of a monitor in more details in Section 4.3.4.

The Scenario. The scenario reacts to verdicts from the monitors according to the scenario specification provided by the developer by controlling the execution. It sends checkpoint and restore requests to the monitors and the execution controllers. It sends debugger commands, event and resume requests to the execution controllers. We present the behavior of the scenario in more details in Section 4.3.5.

The Bus. We centralize service discovery and communication in a special component called the bus. A component joins the distributed i-RV session by connecting to the bus. The joining component sends a joining-component message containing its kind (monitor, execution controller, scenario). The bus replies with a unique identifier for the component, forwards the joining-component message and the identifier to the components that already joined the bus, and sends the joining-component messages and identifiers of all these components to the joining component. A component leaves the system by disconnecting from the bus. The bus then sends a leaving-component message with the identifier of the leaving component to the other components. A component sends a message to a target component through the bus by specifying the identifier of this target. The bus then relays this message and ensures message order. Communication channels are assumed to act as FIFOs and not to duplicate nor drop messages. The bus is the conductor and the central point of the protocol presented in Section 4.3.

4.3.2 Overview of the Protocol

Components connect to each other through a bus, providing a centralized discovery mechanism and communication channels.

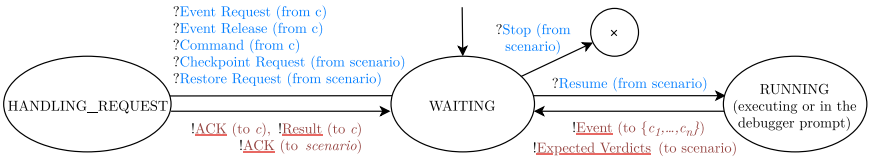


Figure 4.2: Automata-based model of the execution controller.

Contrary to the framework presented in Chapter 3, this architecture allows using several execution controllers. This allows debugging distributed systems composed of several programs, possibly written using different technologies. In the leader election example, this allows interactively verify the three nodes (A, B and C). Execution controllers first start in a suspended state.

The architecture also allows using several monitors. The number of monitors used in the interactive runtime verification session is specified when running the bus (4 in the leader election example). Monitors may only join at the beginning of the session. Each monitor sends event requests to execution controllers for events needed for evaluating their property, and sends an initial verdict to the scenario for each joining execution controller. When the initial verdicts of every monitor has been received, the scenario may send commands and event requests to each execution controller, and eventually resume its execution.

While running, execution controllers produce event notifications for instrumented events. When such an event happen, they suspend their execution and send event notifications to the requesters (monitors and/or the scenario). Monitors then optionally send event requests and event release requests, and send verdicts to the scenario. In the leader election example, for property 4, instrumentation on a function becomeLeader called in a node when it is about to become a leader will be requested by the corresponding monitor. A parameter of the function, currentTerm, an integer representing the current term in the Raft algorithm, will also be requested.

The scenario then reacts to verdicts. It can send event requests and commands to any suspended execution controller and sends resume requests to the execution controllers that produced events. When an event is produced by an execution controller, an event notification is sent to every component that had requested the event and its execution is suspended. Additionally, the execution controller sends the number of monitors that requested the event to the scenario. In the leader election example, any node running function updateLogMetadata will send an event notification to the monitor 3 and 4 (respectively checking properties 3 and 4), suspend their execution and send the number 2 to the scenario. A monitor sends a verdict to the scenario for each event notification it receives, even if the event does not make the monitor change its state. In the leader election example, monitor 3 will send a true verdict (transitioning from state init to state init) after receiving event becomeLeader if it has received an event from a node that voted for the sending node in the current term before. If it has not, monitor 3 will produce a false verdict (transitioning from state init to state bad) , since event becomeLeader should be caused by such a vote. Monitor 4 will produce a true verdict, transitioning from state init to state init, if and only if another event becomeLeader for the same term has not already been received. Before sending the verdict, the monitor may request and release events to the concerned execution controller. When receiving requests from a monitor, an execution controller always sends an acknowledgement in return. The monitor must wait for all acknowledgements before sending a verdict to ensure that the execution controller is not resumed before receiving all the requests from this monitor. The scenario may wait for all the verdicts corresponding to an event before reacting, but does not have to.

Execution controllers may join and quit at any time. The session ends when the scenario quits. Monitors may quit at any time. For instance, if property 3 is not to be checked for each term, the monitor may be stopped as soon as a node becomes a leader. The scenario should not quit while there are execution controllers.

4.3.3 Behavior of an Execution Controller

4.3.3.1 Overview

A model of an execution controller is depicted in Figure 4.2. Its initial state is *WAITING*, in which it waits for event requests or orders from the monitors and the scenario. Upon the reception of an order, the execution controller handles the request and transitions back to state *WAITING*. Upon reception of a resume request, the execution controller transitions to state *RUNNING*. Depending on the resume request, the execution controller expects user input (interactive mode of the debugger) or starts the program execution. When an event is produced, event notifications are sent to the requesters and the execution controller transitions back to state *WAITING*. We describe the inputs, outputs and the internal state of the execution controller in the following paragraphs.

Remark 4.3.3.1 (Number of execution controllers)**.** The number of execution controllers is not specified in the protocol. Since, in the general case, any number of execution controllers can join and leave the session at any time, this number is not known in advance. If the scenario needs to take an action when all execution controllers are joined, it needs to know this number in advance. Therefore, the scenario needs to be specific to this number. In practice, should such requirement happen, the session may be set up such that the scenario or the monitors join the session after every execution controllers. In our implementation, this may be done using bus scripts, presented in Section 5.2.1.

4.3.3.2 Inputs

We describe how an execution controller reacts to incoming messages.

Event requests, event release requests. An event request is sent by a component c to the execution controller to receive notifications when the specified event occurs. This component is said to *listen* for this event. The request contains an identifier i, a location l in the program code and a list of formal parameters p. Location l includes the event type (e.g., function call, variable access). Upon the reception of such a request, the execution controller instruments the code at location l and replies with an acknowledgment with identifier i. An event release request is sent by a component to the execution controller to stop being notified for an event. The request contains an identifier i. When receiving such a request, the execution controller removes instrumentation at location l (if no other component is listening for events associated with this location) and replies with an acknowledgment with identifier i. The pair (c, i) is mapped to location l and parameter p, meaning that different components can use the same request identifier for different events.

Debugger commands. A component requests the execution controller to run a debugger command by sending a debugger command message with command c and identifier i. The execution controller runs this command, and replies with a message with identifier i and containing command results.

Checkpoint and restore requests. A component requests the execution controller to checkpoint the current state of the execution by sending a checkpoint request with identifier i. The execution controller replies with an acknowledgment with identifier i. Similarly, a component requests the execution controller to restore a checkpoint by sending a restore request with the corresponding identifier. The execution controller replies with an acknowledgment with identifier i.

Resume and stop requests. When the execution controller produces an event, its program is suspended. The execution resumes upon the reception of a resume request (from the scenario). The resume request contains a boolean that indicates whether the execution should be resumed or if the debugger prompt should be shown to the user. The execution controller leaves the bus after a stop request.

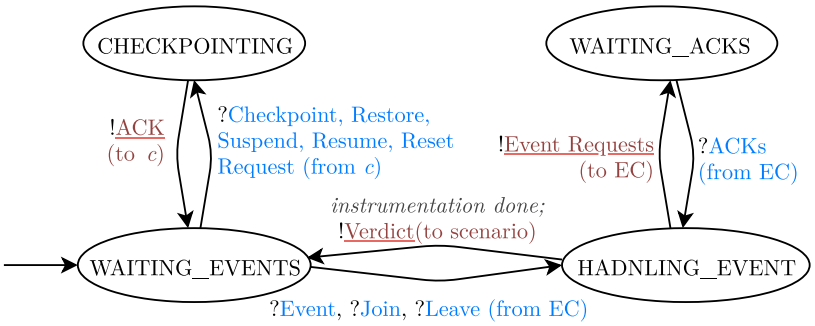


Figure 4.3: Automaton-based model of the monitor. Messages prefixed by "?" (resp. "!") are received (resp. sent) by the monitor.

Leaving component notification. When component c leaves the system, any mapping $(c, i) \mapsto (l, p)$ is forgotten and the corresponding instrumentation (e.g., a breakpoint or a watchpoint) is removed if no other component is listening for an event at location l.

4.3.3.3 Outputs

We describe messages produced by an execution controller.

Event notifications. When the program reaches a location l in the code that is instrumented for an event, the execution of the program is suspended. The execution controller lists tuples (c, i, p) such that (c, i) is mapped to (l, p). The number of tuples (c, i, p) for which c is a monitor is named n. For each (c, i, p), an event notification with identifier i and the list of values values(p) of formal parameters in p is sent to c. The number n of expected verdicts is sent to the scenario. The execution of the program is resumed upon reception of a resume request message.

Acknowledgments. Acknowledgment messages are sent in reply to requests.

4.3.3.4 Internal State

The execution controller keeps track of the program state, the event requests and stores checkpoints requested by the scenario.

4.3.4 Behavior of a Monitor

4.3.4.1 Overview

A model of the monitor is depicted in Figure 4.3. Its initial state is *WAITING_EVENTS*. Whenever an execution controller joins the session, the monitor processes this arrival as an event. It transitions to state *HANDLING_EVENT*. The monitor sends requests for instrumentation to this execution controller and/or other execution controllers (if needed), waits for acknowledgments corresponding to these requests and then sends a verdict to the scenario and transitions back to state *WAITING_EVENTS*, waiting for events. In this state, the monitor can also receive and handle checkpoint and restore requests from the monitor. We describe the inputs, outputs and the internal state of the monitor in the following paragraphs.

4.3.4.2 Inputs

We describe how a monitor reacts to incoming messages.

Event notifications. The monitor receives an event notification for an event it requested to an execution controller c with request identifier i when this event occurs during the corresponding program execution. This event notification contains the request identifier i and the list of values corresponding to the formal parameters given in the event request. The monitor processes the event notification and sends a verdict to the scenario, containing its current state and the event producer c (even in the case the event does not lead to a monitor state update). Exactly one verdict is sent for each event notification. The monitor can request and release events after receiving an event notification and before sending the corresponding verdict. It must wait for all acknowledgments from the execution controller before sending the verdict. Otherwise, the execution controller could receive a resume request from the scenario before receiving every event requests from the monitor, in which case the corresponding events cannot be instrumented.

Joining and leaving component notification. When the monitor receives a joining component notification corresponding to the arrival of an execution controller c, the monitor updates its state, sends event requests to this execution controller (if needed), and then sends a verdict to the scenario containing its new state and the event producer c. Similarly, when the monitor receives a leaving component notification concerning an execution controller, the monitor updates its state and sends a verdict to the scenario.

Checkpoint and restore requests. The monitor replies to a checkpoint request with identifier i by an acknowledgment with identifier i. The monitor saves its current state and maps the sender of the request and this identifier to this saved state. Similarly, it replies to a checkpoint restore request with identifier i by an acknowledgment with identifier i, and restore the previously saved state as its new current state.

Suspend, resume and reset requests. The monitor processes the request with identifier i and replies with an acknowledgment with identifier i. Upon the reception of a suspend request, the monitor sends release requests for all listened events to the corresponding execution controllers. The monitor suspends the evaluation of its property. Upon the reception of a resume request, the monitor sends requests for all events needed to resume the evaluation of its property to the corresponding execution controllers. Upon the reception of a reset request, the monitor resets the evaluation of its property to the initial state and send event requests and release requests the corresponding execution controllers.

Acknowledgments. Various acknowledgment messages are received from the execution controller in reply to requests.

4.3.4.3 Outputs

We describe the messages produced by a monitor.

Event requests, event release requests. A monitor requests events to execution controllers. These requests can be handled by an execution controller as long as it is not running and contain a request identifier i, a location l in the program and a list of formal parameters p. The execution controller sends back an acknowledgment with identifier i. Similarly, a monitor releases an event by sending an event release request containing the corresponding request identifier i and receives back an acknowledgment by the execution controller.

Acknowledgments. Acknowledgments are sent after checkpointing or restoring the monitor state to the requester.

4.3.4.4 Internal State

The monitor keeps track of the current state of the property as well as the events it requested, and stores checkpoints requested by the scenario.

4.3.5 Behavior of the Scenario

The scenario cannot be represented as an automaton since its behavior is user-defined. The scenario drives a distributed i-RV session according to a dynamic set of reactions. These reactions are applied (and the set of reactions can be updated) upon the reception of certain messages. We describe the inputs, outputs and the internal state of the scenario in the following paragraphs.

4.3.5.1 Inputs

We describe how the scenario reacts to incoming messages.

Number of expected monitors. When joining the system, the scenario receives a message containing the number of expected monitors. The scenario waits for every monitor to join the system before sending any resume request, so monitors can request instrumentation to execution controllers before they start.

Verdicts and number of expected verdicts. The scenario receives verdicts with event producer c from monitors, and a number of expected verdicts n from execution controller c for any event generated in this execution controller. The scenario may receive some of these verdicts before or after the number of expected verdicts itself. The scenario reacts after the reception of all these messages (for initial verdicts produced by monitors after an execution controller joins the session, no number is sent. n is the number of monitors known at the time the execution controller joined). The reaction can be a debugger command, an event, checkpoint, restore or resume request sent to c. This resume request contains a boolean indicating whether the execution should be resumed or if the debugger prompt should be displayed to the user. If no resume request is ever sent and the execution is not terminated, execution controller c is stuck forever. When no verdicts from any monitor for this event are expected, n is equal to 0.

Event notifications, command results. The scenario receives notifications requested events and command results. In both cases, the scenario reacts and the execution resumes when the scenario sends a resume request. A corresponding number of expected verdicts message with the same verdict identifier is received before or after the event notification.

Acknowledgments. Various acknowledgment messages are received in reply to requests sent to an execution controller.

4.3.5.2 Outputs

We describe the messages produced by the scenario.

Resume and stop requests. A resume or stop request can be sent to a suspended execution controller (i.e., after reception of an expected verdicts number and before sending a resume or a stop request). The request results in the execution controller resuming its execution, either by continuing the program execution or by showing the debugger prompt to the user, depending on the content of the request, or stopping its execution.

Debugger commands. Debugger commands can be sent to a suspended execution controller. The execution controller replies the result of the command.

Checkpoint requests, restore, suspend, resume and reset requests. The scenario can react to verdict sets, event notifications or command results by sending checkpoint and restore requests to the execution controllers and the monitors. The scenario can also send suspend, resume and reset requests to monitors. Such requests must not be sent to monitors listening to events of running execution controllers.

4.3.5.3 Internal State

The scenario stores the values of variables set and keeps track of events and checkpoint requests done while handling verdicts. It also stores the verdicts that remain to be handled, the number of verdicts to expect as well as the number of monitors in the session.

4.4 Verifying the Distributed I-RV Protocol

We verified the correctness of the distributed i-RV protocol using the SPIN model-checker [Hol97]. We wrote a Promela model of the protocol and specifications. In addition to our verification purposes, the model can also be used as a programming-language independent and high abstraction level description of the protocol, providing a solid basis for implementing distributed i-RV. We wrote specifications that state desirable properties regarding the exchanges of messages in the protocol. These properties ensure that messages are sent and received by the right components and processed in the right order. They also ensure that the protocol does not deadlock, terminates properly and that each key feature of the protocol is well-behaved: monitors correctly react to event notifications by sending verdicts, the scenario correctly react to verdicts by controlling the execution controllers and requesting that they resume their execution, and the execution controllers correctly resume their execution upon request by sending event notifications to the monitors and the scenario.

4.4.1 Architecture and Features of the Model

We focused on the protocol and the interface of the components rather than their internal behavior. Thus, we did not represent the program execution, the property checking, or the scenario execution. We rather modeled the interface of the four components presented in the previous sections: the monitor, the scenario, the execution controller and the bus. The model is composed of a bus, a scenario, a number of monitors and of execution controllers. Components exchange messages that do not contain actual data and are described using Promela processes. These processes are written as infinite loops waiting for and reacting to messages, interrupted when an exit condition is met. The model is a superset of the behaviors that can be observed in a real distributed i-RV session. Thus, a specification applying to every behavior in the model apply every valid behavior of the real protocol. In the next paragraphs, we describe the main simplifications from the protocol presented in Section 4.3.

4.4.1.1 Messages

Instead of using sockets, processes communicate using blocking message queues[1]. Messages are modeled using a structure[2] with a type, a sender, a receiver and an optional custom field used for certain types of messages.

4.4.1.2 Requests, Events and Verdicts

Monitors send a random number[3] of event requests between the reception of an event notification and sending a verdict, and the scenario sends a random number of debugger requests before sending a resume request. An upper bound on these numbers is set when building the model. The execution

[1] Promela channels: `http://spinroot.com/spin/Man/chan.html`
[2] Promela structured data types: `http://spinroot.com/spin/Man/typedef.html`
[3] All the possible numbers will be explored when checking the model. Randomness only exists in simulations.

Atom	Definition
`newMsg`	The bus is handling a new message
`Event`	*msg*.`type` = EventNotification
`Request`	*msg*.`type` = Request
`Resume`	*msg*.`type` = ResumeRequest
`QuitRequest`	*msg*.`type` = QuitRequest
`ExpectedVerdicts`	*msg*.`type` = ExpectedVerdictCount
`Verdict`	*msg*.`type` = Verdict
`rcptScn`	*msg*.`recipient` is a scenario
`rcptMon`	*msg*.`recipient` is a monitor
`rcptEC`	*msg*.`recipient` is an execution controller
`senderEC`	*msg*.`sender` is an execution controller
`senderMon`	*msg*.`sender` is a monitor
`senderScn`	*msg*.`sender` is a scenario
`sender`(*id*)	*msg*.`sender` = *id*
`recipient`(*id*)	*msg*.`recipient` = *id*
`ecmatch`(*ec*)	*msg*.`ec` = *ec*

Table 4.1: LTL atomic propositions
Atomic propositions used in the LTL properties given in Figure 4.4.

controller answers these requests with an acknowledgment. When the execution controller receives a resume request, it sends messages of type EventNotification to a random subset of the processes that are either a monitor or the scenario. Contrary to the real protocol, these messages do not contain any identifier or data except for verdicts, event requests are not differentiated from debugger requests, and an execution controller can send an event notification to a monitor that did not send any request to this execution controller.

4.4.2 Specifications

We checked our model against safety and liveness properties written using Linear Temporal Logic (LTL) [Pnu77] formulas given in Figure 4.4 and described hereinafter. These formulas use atoms and functions defined in Figure 4.1. In these formulas, *msg* refers to the message at the head of the message queue of the bus.

S1 Every event notification is received by a monitor or a scenario.

S2 Requests are sent by monitors and scenarios and received by execution controllers.

S3 Verdicts are sent by monitors to scenarios.

L1 When an event is sent by an execution controller *ec* to a monitor *mon*, a verdict from monitor *mon* corresponding to this event will be sent by *mon*.

L2 Expected Verdict Count messages are followed by a resume request.

L3 Events from any execution controller are followed by a resume or a quit request to this execution controller.

S4 An event notification from any execution controller to any monitor cannot be seen after an event notification from this execution controller to this monitor until a verdict is issued by this monitor caused by this execution controller.

$$\Box\,(\texttt{Event} \implies ((\texttt{rcptScn} \vee \texttt{rcptMon}) \wedge \texttt{senderEC})) \tag{S1}$$

$$\Box\,\big(\texttt{Request} \implies \big((\texttt{senderMon} \vee \texttt{senderScn}) \wedge \texttt{rcptEC}\big)\big) \tag{S2}$$

$$\Box\,\big(\texttt{Verdict} \implies \big(\texttt{senderMon} \wedge \texttt{rcptScn}\big)\big) \tag{S3}$$

$$\begin{aligned}\forall ec \in \text{ECs}, \forall mon \in \text{Monitors}, \Box\Big(&\big[\texttt{newMsg} \wedge \texttt{Event} \wedge \texttt{sender}(ec) \wedge \texttt{rcpt}(mon)\big] \\ \implies &\bigcirc\Big[\big(\texttt{newMsg} \implies \neg\big(\texttt{Event} \wedge \texttt{sender}(ec) \wedge \texttt{rcpt}(mon)\big)\big)\,\mathsf{U}\,\big(\texttt{Verdict} \wedge \texttt{ecIs}(ec)\big)\Big]\Big)\end{aligned} \tag{S4}$$

$$\begin{aligned}&\forall ec \in \text{ECs}, \forall mon \in \text{Monitors}, \\ &\quad\Box\big(\big(\texttt{Event} \wedge \texttt{sender}(ec) \wedge \texttt{rcpt}(mon)\big) \rightarrow \big(\texttt{Verdict} \wedge \texttt{ecIs}(ec) \wedge \texttt{sender}(mon)\big)\big)\end{aligned} \tag{L1}$$

$$\forall ec \in ECs, \Box\big(\big(\texttt{ExpectedVerdicts} \wedge \texttt{rcptScn} \wedge \texttt{sender}(ec)\big) \rightarrow \big(\texttt{Resume} \wedge \texttt{rcpt}(ec)\big)\big) \tag{L2}$$

$$\forall ec \in \text{ECs}, \Box\big(\big(\texttt{Event} \wedge \texttt{sender}(ec)\big) \rightarrow \big(\big(\texttt{Resume} \vee \texttt{QuitRequest}\big) \wedge \texttt{rcpt}(ec)\big)\big) \tag{L3}$$

Figure 4.4: LTL Properties checked against our model of the i-RV protocol. Operator $\Box$ means *always*, $\Diamond$ means *eventually*, $\bigcirc$ means *next*, $\rightarrow$ means *leads to* and U means *strong until*. ECs is the set of execution controllers and Monitors the set of monitors.

4.4.3 Checking the Model in SPIN

Checking the model is done either by exhaustively browsing the state space of the model, or by running a simulation, which is one path in this state space (a trace). A state contains the values for each variable and the current position in each process. From one state, the next reachable states are determined by the executable instructions in each process. The state space includes every possible scheduling and every possible choice[4] (which instruction to execute next) made during the execution of the model. Properties are written using LTL formulas[5] which are translated into never claims checked on every state of the state space or trace.

4.4.4 Results

We exhaustively checked the model against the specifications with different numbers of monitors, execution controllers and limits for the number of requests sent by the execution controllers and the monitors. We present these results in Table 4.2. In these configurations, no errors were found: all the specifications given in Section 4.4.2 are respected, and the model does not deadlock, that is, all the executions end when all processes terminate after a stopping condition is reached. Our results prove that our protocol does not deadlock in the tested configurations, and that events produced by the execution controllers are indeed handled by the monitors (issuing a verdict per event) and the scenario (reacting to the received verdicts). We were limited in the number of execution controllers and monitors we could include in the model because of memory limitations. However, random simulations of the model against the specifications with two monitors and two execution controllers did not raise unexpected conditions.

[4]These choices include picking random numbers.
[5]`http://spinroot.com/spin/Man/ltl.html`

Config.	Time	Mem.	States	Visited	Trans.	Depth
1 ec, 1 mon, 0 req.	0.2 s	137 MiB	12k	28k	61k	482
1 ec, 1 mon, 2 req.	6 s	186 MiB	203k	784k	2M	2k
1 ec, 1 mon, 5 req.	2 min	846 MiB	3M	13M	41M	5k
1 ec, 2 mon, 0 req.	54 s	758 MiB	2M	5M	14M	1k
1 ec, 2 mon, 1 req.	15 min	8 GiB	24M	78M	230M	5k
2 ec, 1 mon, 0 req.	33 min	15.5 GiB	41M	155M	504M	8k

Table 4.2: Model checking results. We checked several configurations of the model. The first column gives the number of execution controllers, monitors and the maximum number of requests. Although configurations with 0 requests do not seem meaningful, they do allow checking event handling, since events are generated regardless of the absence of requests in this model. Column 2 and 3 give the time taken and the amount of memory used to check the model against the properties given in Section 4.4.2. Column 3 give the number of accessible states in the model. Column 4 and 5 give the numbers of visited states and transitions taken when checking conformance against the liveness properties. Column 6 gives the depth of the model: its the longest execution, in terms of state count.

4.5 Conclusion

In this chapter, we presented an architecture for interactive runtime verification suitable for interactively verifying systems composed of several processes, possibly written in different programming languages. This architecture is also suitable for checking several properties in the same interactive runtime verification session, allowing to precisely specify the behavior of the scenario, even in cases different monitors emit conflicting verdict. We also allow existing tools such as monitors and debugger to be adapted to interactive runtime verification. These goals are achieved by defining a proper interface between component, and a carefully crafted protocol based on a bus. We impose some restrictions on what can be checked. Monitor do not have access to the global state of the system being verified, and traces they receive are ony partially ordered: there are no garantees on the order of events from different nodes. To ensure its soundness, we model-checked the protocol we proposed. In Section 5.2, we present Dist-Verde, an implementation of the architecture, and in Section 6.2, we present some experimentations on distributed i-RV.

Chapter 5

Implementation

Contents

In this chapter, we present Verde, an implementation for interactive runtime verification. This implementation allows us to experiment with this approach and assess its usefulness. We first implemented the concepts presented in Chapter 3 as an extension for GDB. We present this extension in Section 5.1. We then implemented distributed i-RV, presented in Chapter 4. We present this implementation in Section 5.2.

Summary. Verde is a tool to evaluate and experiment with interactive runtime verification. We made two versions of Verde. Our first version reflects our first step in interactive runtime verification. In the first part of this chapter, we first present this version, built as an extension for GDB, using its Python interface. We present how the program execution is managed, and then the different modules of the extension. We first present the GDB command module, which is its entry point. We then focus on its central component: the monitor. The monitor module evaluates properties and execute scenarios. The property model is inspired by finite state machines and supports trace slicing, useful to track separately the state of different objets in the program. We then present the animated view of the properties provided by Verde and how we handle checkpointing. We conclude the first part of this chapter by presenting how to use the extension. We present a typical interactive runtime verification session with Verde, and the languages used to write properties and scenarios. In the second part of the chapter, we present the second version of Verde, called Dist-Verde. Dist-Verde has a different architecture from the first version to handle distributed i-RV, but reuses most of its code (both implementations are now maintained in the same codebase). We first present how the different components of distributed i-RV communicate through the bus, using a protocol based on Protobuf [Pro] and sockets, and how components compatible with Dist-Verde can be built. We focus on the implementation of the bus, that can be scripted using bus scripts. This mechanism lets a developer automate the set up of a distributed i-RV session with specific needs, such as using custom components or running components on several computers. We then present Verde-Monitor, a monitor built from the monitor provided in the first version of Verde by implementing an event manager that is compatible with distributed i-RV. This monitor uses the same syntax for properties, with a few additions relevant to distributed i-RV, discussed in Section 5.2.2. After presenting the monitor, we present the two execution controllers included in Dist-Verde: Verde-GDB and Verde-JDB. Verde-GDB is an execution controller based on GDB and a straightforward adaptation of the instrumentation module of the first version of Verde. Verde-JDB is an execution controller based on JDB, the Java debugger provided as a reference implementation of the debugging infrastructure of Java. We extend JDB using AspectJ to make it programmable. Verde-JDB consists of this extended and programmable version of JDB, and an interface to communicate with the bus. We mention how we handle checkpointing in Dist-Verde. We then explain how to use Dist-Verde, either by running each component manually, by using the program we provide to manage a distributed i-RV session more easily, and how to set up a custom distributed i-RV session for a more advanced usage.

5.1 A GDB Extension for Interactive Runtime Verification

We first implemented Verde as an extension for GDB written in Python. In the next section, we give an overview of this implementation, before presenting each components in more details in the following sections.

5.1.1 Overview

We implement the i-RV-program as presented in Algorithm 11 (page 58) (Section 3.5.4). We chose to base our first implementation on top of the GNU Debugger. GDB has several strengths over other debuggers. GDB is a robust, battle-tested free software. Among C and C++ debuggers, it is one of the most widely used and well-known, and among open source debuggers, the most widely used. Most Integrated Development Environments integrate GDB for debugging C and C++ programs.

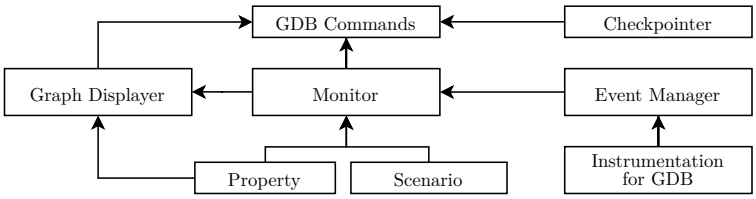


Figure 5.1: Organization of the code of Verde. Arrows show dependency between components ($A \to B$ means B depends on A).

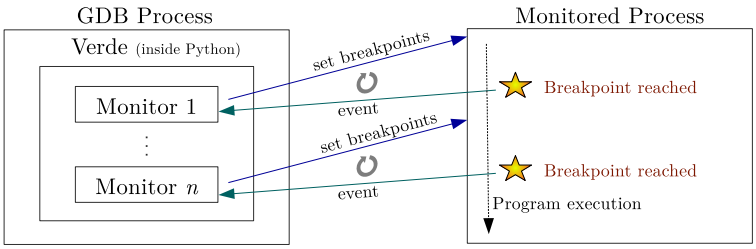


Figure 5.2: Execution flow in Verde.

GDB also supports other programming languages like Fortran, Pascal, Ada, Rust and D[1]. GDB is also a major debugger for these languages. GDB provides a comprehensive Python API to write extensions and scripts. We considered another open source debugger, LLDB, the debugger of the LLVM project. This debugger also provides a Python API. However, because of the respective ages of GDB (released in 1986) and LLDB (released in 2010), choosing GDB over LLDB made it less likely to hit a missing feature. When we started this implementation in 2015, LLDB was less than 5 years old. As of today, LLDB supports fewer languages and does not provide any checkpointing feature and lacks remote debugging on GNU/Linux, which could have been useful in this work. The Python API of GDB is also better documented than the Python API of LLDB. However, LLDB is more and more used. It is the default debugger of XCode. It is also compatible with major IDEs like Eclipse, Visual Studio Code, QtCreator and KDevelop. The GDB-specific code of our extension is isolated to ease a port to a debugger supporting Python scripting such as LLDB.

In Figure 5.1, we show the architecture of the extension. Verde is loaded in GDB using command `source gdbcommands.py`. File `gdbcommands.py` defines GDB commands for the user to control Verde. We present this module in Section 5.1.2. The central component of Verde is the monitor, presented in Section 5.1.3. This component is in charge of evaluating the property and executing the scenario. This evaluation depends on how events are handled in Dist-Verde. Events are presented in Section 5.1.3.1. The property evaluation and scenario execution are presented in Section 5.1.3.2. The monitor relies on instrumentation provided by GDB and an event manager. We present the event manager and how the instrumentation is handled in Section 5.1.5. Verde provides an animated view of properties, presented in Section 5.1.6. We also implemented support for checkpointing, presented in Section 5.1.7. Usage of Verde is presented in Section 5.1.9.

Execution Flow. Figure 5.2 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 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. We describe this execution flow in more details in the following sections.

[1]The full list of programming languages supported by GDB is given at `https://sourceware.org/gdb/onlinedocs/gdb/Supported-Languages.html#Supported-Languages`

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 a 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 5.1: List of Verde commands.

5.1.2 GDB Commands

Verde is loaded in GDB by sourcing the GDB Commands module. This module defines the commands that allow the developer to control Verde. Commands are defined in the namespace `verde` by extending class `gdb.Command` provided by the GDB Python API[2]. A list of commands defined by this module is given in Table 5.1.

5.1.3 The Monitor

The central component of Verde is the monitor. In this component, we implement the monitor defined in Section 3.3.4. In this section, we presented the inference rules in Figure 3.12 (page 44).

The monitor uses the event manager to request and release events from the program execution. We present representation of events in Verde in Section 5.1.3.1. These events are used to evaluate properties as per rule EVENT. We present this evaluation, and the property model used in Section 5.1.3.2. This evaluation produces verdicts used to execute the scenario, presented in Section 5.1.4 and a set of events to instrument. Events to instrument are given to the instrumentation module, presented in Section 5.1.5. Figure 3.12 (page 44) also define rules CHECKPOINT and RESTORE, related to the checkpointing of the monitor. This aspect is presented in Section 5.1.3.3.

5.1.3.1 Events

The representation of events in Verde reflects how we instrument programs with GDB (presented in Section 5.1.5). They are suitable for the model of property used in Verde, presented in Section 5.1.3.2. Events are defined in `src/python/property.py`. By events, we mean either formal events or runtime events. Formal events are event specifications used to make event requests. This specification includes a list of parameters. Runtime events (`RuntimeEvent`) are produced during the execution. A runtime event contains runtime values corresponding to the parameters specified by the corresponding formal event. Instrumentation is done using breakpoints and watchpoints. Breakpoints (resp. watchpoints) are used to produce call events (resp. watch events). Call events (type `CallEvent`, defined in component `property`) are composed of a name (the specification of

[2] `https://sourceware.org/gdb/onlinedocs/gdb/Commands-In-Python.html`

the function or the line on which the event is produced), a list of parameters to get, and a boolean specifying whether the event is produced when the function is entered or exited. Watch events (type `WatchEvent`) are composed of a name (the name of the variable to monitor), a list of parameters to get, and a string specifying whether the event is produced if the variable is read (`"r"`), written (`"w"`) or both (`"rw"`).

Event Parameters. Formal parameters (type `EventParameter`) specify a name used in the property evaluation, as well as which runtime value is to be retrieved. A parameter either refer to:

- a parameter of the function specified by the event (or the current function),
- the name of a variable,
- the address of such a variable,
- the value pointed by such a variable,
- an expression as evaluated in the language of the program, or
- the value given by the evaluation of a Python expression.

The syntax used to describe event parameters in Verde is given in Section 5.1.9.2.

Example 5.1.3.1 (Event)**.** Formal event `before call queue_push(arg 1 as q, arg 2 as v)` is a before call event. This event is triggered when function `queue_push` is called. The event has two formal parameters: the first event parameter is the first parameter of the call, and is named q in the event. The second event parameter is the second parameter of the call, and is named v in the event. Runtime event `before call queue_push(0x1401260, 5)` is associated with this formal event. This event makes the property presented in Figure 2.5 (page 17) transition from state `queue_ready` to state `queue_ready` or state `sink` for queue $0x1401260$ (see Section 5.1.3.2 for an explanation on how properties are evaluated in Verde).

5.1.3.2 Evaluation and Property Model

The property model used in Verde is inspired by finite state machines. See Section 5.1.9.2 for a description of the syntax used to define properties in Verde using this model.

Properties are composed of an initial environment and a list of states, one of which is initial. The initial environment contains variables that can be read and modified during the property evaluation. This environment is written as a block of Python code. A state consists of a name, a flag defining whether it is accepting, and a list of transitions. A transition consists of the specification of a formal event, an optional guard, a success end and an optional failure end. The guard is a boolean expression referring to the values of parameters specified in the formal event of the transition and values in the environment of the property. Success and failure blocks consist of an optional block of Python code that updates the environment, and a destination state. If the guard evaluates to `true` (resp. `false`), the environment is updated using the block of Python code of the success (resp. failure) end, and the new active state is the destination state of this end.

See Figure 2.5 (page 17) for an example of a property that can be evaluated in Verde. In this property, 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`.

A verdict is produced each time an event is processed. A verdict contains the name of the active state, and whether it is accepting. The scenario is executed for each produced verdict.

5.1.3.3 Parametric Trace Slicing

We implemented trace slicing in Verde, inspiring from [CR09]. Trace slicing is a feature that allows evaluating parametric properties. Such properties get instantiated with runtime values and are evaluated over sub-sequences of the whole trace of the program execution instead of the whole trace itself. These sub-sequences of the trace are related to particular instances of parameters (e.g., objects) in the program. For instance, Figure 2.5 (page 17) 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 Figure 2.5 (page 17), the unique parameter in this list is the queue. In the monitor, each active state is mapped to an instance of these parameters (the slice instance). The monitor initially has one active state: the initial state. This state is mapped to the slice instance in which each parameter instance is 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 slice instance:

- If all parameters of the event are equal to the parameters of the slice instance, then the slice is updated according to the property semantics, i.e., by evaluating the guard and the success or the failure block of the active state of the slice.
- Otherwise, a new slice instance is created if:
  - every instantiated parameter in the slice instance 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 slice instance is built by merging the slice instance and the event slice instance. The state of this new slice instance is the active state being considered. The event is evaluated in the context of this new slice instance.

When a new slice instance is created, the environment containing the variables used in the property is also copied, so the new slice instance inherits from the values stored in the slice instance its was copied from. Following the semantics of Python and many other similar programming languages, primitive values are not shared while more complex constructions like dictionaries and lists are. Slice instances are stored in a tree. New slice instance are children of the slice instance from they were created.

Handling Disappearing Instances. Sometimes, an object is destructed (freed), and then a new object is created with the same identifier. However, they 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.

Checkpointing. Our monitor supports checkpointing. A checkpoint is a recursive copy of the tree of slice instances. Restoring this checkpoint consists in restoring this copy.

5.1.4 The Scenario

In our extension, a scenario consists of an initial environment and a list of reactions. The initial environment is specified using a block of Python code. Reactions are composed of a verdict selector and a block of Python code. The verdict selector specifies which verdicts trigger the reaction, that is, trigger the execution of the code block. A reaction can be triggered when entering, or leaving an accepting or non-accepting state, or a particular state specified by its name. The scenario engine,

in charge of executing the scenario, is implemented as per Algorithm 8 (page 56), presented in Section 3.5.3. Information about the state being entered and the state begin left is stored in each verdict produced by the monitor. For each reaction, the verdict selector is used to build the set of actions to run for a given verdict used in line Line 14 in Algorithm 8 (page 56). Functions SCN::EXECACTION, SCN::APPLYCMDREPLY and SCN::APPLYPOINT defined in in Algorithm 8 (page 56) are realized by the Python interpreter and the GDB API.

5.1.5 Managing Events and Instrumentation

The *event manager* is an abstration layer between the monitor and the instrumentation module. The event manager receives requests for instrumenting, or disabling instrumentation for events from the monitor. The event manager controls the instrumentation module to fulfill these requests. In rule EVENT (Figure 3.12 (page 44), Section 3.3.4), the monitor produces the complete set of events for which instrumentation is needed. While this design simplifies the operational semantics of the monitor and the i-RV-program, our implementation slightly differs: in Verde, the monitor requests the event manager for each event to (un)instrument. This design eased the implementation of the monitor. The event manager implements features of EC:INSTRUMENT and EC:CLEAREVENTS defined in Algorithm 4 (page 54), presented in Section 3.5.2. These functions add and remove instrumentation for events provided by the monitor. They take sets of events as parameter, which is consistent with rule EVENT. To reflect the implementation of the monitor, in Verde, we chose to implement functions handling one event instead of a set of events.

In Section 3.5.2, functions defined in Algorithm 4 (page 54) depend on instrumentation functions defined in Algorithm 3 (page 53). These functions, as well as functions defined in Algorithm 2 (page 52) and Algorithm 1 (page 51) are realized by GDB[3] and the operating system. In Verde, the instrumentation module is an interface between GDB and the event manager. The instrumentation module converts events to debugger points and produce runtime events when this points are reached. When the instrumentation module produces an event, the event manager propagates this event to the monitor. The *instrumentation module* is an interface between the GDB Python API and the event manager. It allows the event manager to request instrumentation for events needed by the monitor to evaluate the property. The instrumentation module exposes function `on`, which takes two parameters: the formal event for which instrumentation is needed, and the function to call when the event happens. Function `on` returns an object which can be passed to functions `enable` and `disable`, also defined by the instrumentation module. These functions respectively enable and disable instrumentation for the corresponding event. Watch events and before call events are instrumented by adding breakpoints and watchpoints in the program code. GDB provides an unified API to set breakpoints and watchpoints using class `gdb.Breakpoint`[4]. To set a point, this class must be extended. The specification of the point is given in the parameters of the constructor of the class. When the point is reached, method `stop` of the class is called. This method is to be defined when extending `gdb.Breakpoint`.

Instrumenting after call events requires more work. Instrumentation is set at multiple places. The location at which such events are triggered depend on the call stack. Values of call parameters specified by the event are to be extracted before the code of the function is executed. As such, a regular breakpoint is set as described before. When this breakpoint is reached, parameters are saved. Then, a finish breakpoint is set. Finish breakpoints are breakpoints that are triggered after the end of the current function. Recursive calls are also taken in account.

5.1.6 The Graph Displayer

The graph displayer provides an optional animated view of the active state of the monitor. See Figure 5.3 for a screenshot. The view is built using Graphiz from the list of states and transitions of the property loaded in the monitor. Graphviz produces an image in the Scalable Vector Graphics (SVG) format. This image is shown using a Web engine. When the state of the monitor is updated,

[3] See `https://sourceware.org/gdb/onlinedocs/gdb/Breakpoints-In-Python.html`.
[4] `https://sourceware.org/gdb/onlinedocs/gdb/Breakpoints-In-Python.html`

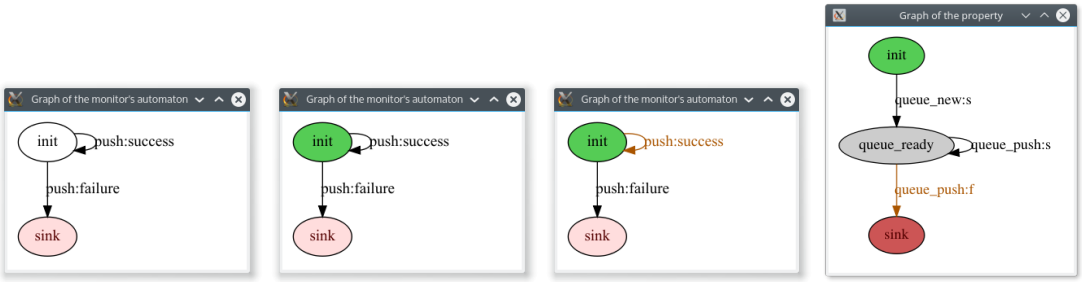


Figure 5.3: The graph displayer
The graph displayer showing a property in its initial state, then in an accepting state, and then in a non-accepting state.

the displayer colors the active states of the property by manipulating the SVG elements of the image. Accepting (resp. non-accepting) states are shown in green (resp. green).

5.1.7 The Checkpointer

In Section 3, checkpointing is described in Algorithm 7 (page 55) and Algorithm 10 (page 57). Checkpointing is done by saving the state of the execution controller and the monitor. We described monitor checkpointing in Section 5.1.3.3. In this section, we describe how we implement checkpointing for the execution controller. As per Algorithm 11 (page 58), checkpointing can be requested by the user (Line 35 and Line 37) and from the scenario, since the scenario can run debugger commands (Algorithm 9 (page 57), Line 21). Creating a checkpoint allows going back to the specific point in the execution at which the checkpoint is created. Verde for GDB 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 multi-threaded programming is not supported since `fork` keeps only one thread in the new process. The second method uses CRIU[5], which supports multi-threaded 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 (See [CRIb] 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 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. GDB itself is not checkpointed: breakpoints set by the developer and the scenario are kept as-is, monitor breakpoints are removed before checkpointing and set againt when restoring a checkpoint. Support for other checkpointing solutions may be added as needed.

[5]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/`.

5.1.8 Implementation of the I-RV-Program

In the previous sections, we presented the different modules of Verde, implementing the different components of the i-RV-program as defined in Chapter 3. In this section, we present how Verde realizes the i-RV-program itself, as defined in Algorithm 11 (page 58, Section 3.5.4). This algorithm assumes the monitor and the scenario are already loaded. The GDB Commands module, presented in Section 5.1.2, defines custom commands to load monitors and scenario. See Section 5.1.9 for an explanation on how to use these commands. The GDB Commands module also provides a command to start the i-RV-program as per Algorithm 11 (page 58): `verde run-with-program`. First, function INIT, defined in Algorithm 9 (page 57), is called. This function initializes the monitor, which produces an initial verdict and provides a set of events to instrument. These events are instrumented in the execution controller, and a first call to the scenario is made. In Verde, the monitor is initialized. Events are instrumented using the event manager (see Section 5.1.5), and the scenario is applied, executing actions linked with the initial state, or (non-)accepting states if the initial state is (non-)accepting (see Section 5.1.4).

Then, the i-RV-program is set in passive mode by setting variable *cont* to `true` in Algorithm 11 (page 58) on Line 2, and by calling `gdb.execute("continue")` in Verde. Then, the i-RV-program enters the read-eval-print loop (REPL). This loop is handled at Line 3 in the model, and by GDB itself with Verde.

In passive mode, the next instruction of the program is executed. This execution can reach a breakpoint or a watchpoint (Line 4). This behavior is described in Algorithm 9 (page 57) at Line 9 and is implemented with Verde by GDB and by the instrumentation module presented in Section 5.1.5.

In interactive mode, the i-RV-program waits for the user to input commands. GDB implements commands `step`, `continue`, `set watchpoint` (using syntaxes `watch`, `awatch` and `rwatch`[6]), `set breakpoint` (using syntax `break wp`), `unset breakpoint`, `set`, `print` and other commands not described in the model. GDB provides commands `checkpoint` and `restart` that do not realize commands `checkpoint restore` of the i-RV-program. These commands only checkpoint the program, without checkpointing the monitor. Instead, we provide commands `verde checkpoint` and `verde restore-checkpoint` to implement the corresponding commands of the i-RV-program. We describe this implementation in Section 5.1.7.

Remark 5.1.8.1 (About support for multithreaded programs)**.** Our model does not support multiple threads. However, we do not prevent GDB from loading multi-threaded programs and interactively verifying multithreaded programs using Verde inherits the same characteristics as debugging multithreaded programs in GDB. More specifically, scheduling may be affected by breakpoints or step by step debugging. GDB provides options to configure the behavior of a breakpoint in a multithreaded environment. A breakpoint can be set to suspend all the running threads, or to suspend only the thread in which it is reached. We do not see any specific issue when using the former option, which is the default behavior. We have not explored the latter option but we expect issues. When a monitor breakpoint is reached, Verde assumes there will be no other breakpoint triggered until the execution is resumed. Triggering more than one event in a row leads to unknown behaviors. Thread creation and deletion can be monitored by monitoring the calls to the functions handling these events (e.g., `pthread_create` with the *pthreads* library).

5.1.9 Usage

In this section, we present how to use Verde. We present a typical usage session in Section 5.1.9.1. We then present the syntax used to write properties and scenarios in Section 5.1.9.2.

5.1.9.1 A Typical Session

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

[6] See `https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html`.

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

Figure 5.4: A typical debugging session with Verde.

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` sub-command (see Figure 5.3). See Figure 5.4 for the reproduction of a typical debugging session with Verde. For convenience, we provide a script, `simple-verde`, that runs GDB on the specified program, loads Verde and the specified property and scenario. This script is run as follows:

```
simple-verde --prop property [-scn scenario] [-show-graph] [-quiet] [--prop ...]
    ↪ prgm
```

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. See Table 5.1 for a comprehensive list of the commands provided by Verde.

5.1.9.2 Syntax of Properties and Scenarios

In this section, we present the syntax used to write properties and scenarios in Verde.

Writing Properties. Verde provides a syntax for writing properties[7]. An informal grammar is provided in Figure 5.5. Figure 5.6 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 Section 5.1.3.3), 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 free Python code block that returns `True` (resp. `False`) if the guard succeeds (resp. fails) and None if the transition should be ignored.

Writing Scenarios. A scenario consists in an optional environment initialization block. This block is written as follows:

```
initialization {
    # Python code
}
```

[7] We did not use pre-existing syntax in order to allow us flexibility as we experiment.

```
1
2  [slice on [param,] +]?
3
4  [initialization {
5      Python code
6  }]?
7
8  [state state_name[[non-]?accepting]? [final]? [action_name()]? {
9    [transition {
10     [before | after]? event event_name([param,] * ) [{
11       Python code returning True False or None
12     }]?
13     [success [{
14         Python code
15     }]? [action_name()]? state_name]?
16     [failure [{
17         Python code
18     }]? [action_name()]? state_name]?
19   }] *
20 }] +
```

Figure 5.5: Informal grammar for the property description language in Verde
Informal grammar for the automaton-based property description language in Verde.

```
1  slice on queue
2
3  initialization {
4      N = 0
5      maxSize = 0
6  }
7
8  state init accepting {
9      transition {
10         event queue_new(queue, size : int) {
11             maxSize = size - 1
12         }
13         success queue_ready
14     }
15 }
16
17 state sink non-accepting sink_reached()
18
19 state queue_ready accepting {
20     transition {
21         event queue_push(queue) {
22             return N < maxSize
23         }
24         success {
25             N = N + 1
26             print("nb elem: " + str(N))
27         } queue_ready
28
29         failure sink
30     }
31
32     transition {
33         event queue_pop(queue) {
34             return N > 0
35         }
36
37         success {
38             N = N - 1
39             print("nb elem: "+str(N))
40         } queue_ready
41
42         failure sink
43     }
44 }
```

Figure 5.6: A verde Property
Verde version of the property in Figure 2.5 (page 17).

```
on entering accepting state {
    print("Scenario: a new state has been reached:" +
        old_state + " -> " + new_state)
    c.append(checkpoint())

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

    after_breakpoint(called_when_checkpoint_ready)
}

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

Figure 5.7: A generic scenario
A generic scenario saving the state of the execution after each verdict.

Reactions are given after the initialization. Each reaction consist of a verdict selector and an action written as a block of Python code as follows:

```
on (leaving|entering) state (name) {
    # Python code
}
```

or:

```
on (leaving|entering) (accepting|non-accepting|initial) state {
    # Python code
}
```

Actions are written using regular Python code. Verde provides an API to set breakpoints and watchpoints, set and restore checkpoints and suspend the execution. If a verdict matches several reactions, these reactions are executed in order of writing. We provide an example of a scenario in Figure 5.7. This scenario sets a checkpoint each time an accepting state is reached. When a non-accepting state is reached, the execution is suspended.
The extension we presented in this section implements the concepts presented in Section 3. In the next section, we adapt and extend this implementation to support distributed runtime implementation as presented in Section 4.

5.2 Dist-Verde: Implementing Distributed I-RV

In this section, we present Dist-Verde, our implementation of the concepts presented in Section 4. We illustrate the architecture of Dist-Verde in Figure 5.8. Dist-Verde consists of a distributed i-RV-compatible monitor, a scenario engine and two debugger extensions (execution controllers) communicating through a bus. We present the bus and the protocol in Section 5.2.1. We present the monitor in Section 5.2.2. We present the scenario in Section 5.2.3. We present the two debugger extensions in Section 5.2.4. We present how checkpointing can be used with Dist-Verde in Section 5.2.5. We present how to use Dist-Verde in Section 5.2.6.

5.2.1 Verde-Bus and the Protocol

We implemented the protocol presented in Section 4.3 using TCP or UNIX sockets (depending on the settings) and Protobuf [Pro]. Protobuf is a widely used tool that produces a parser and a serializer

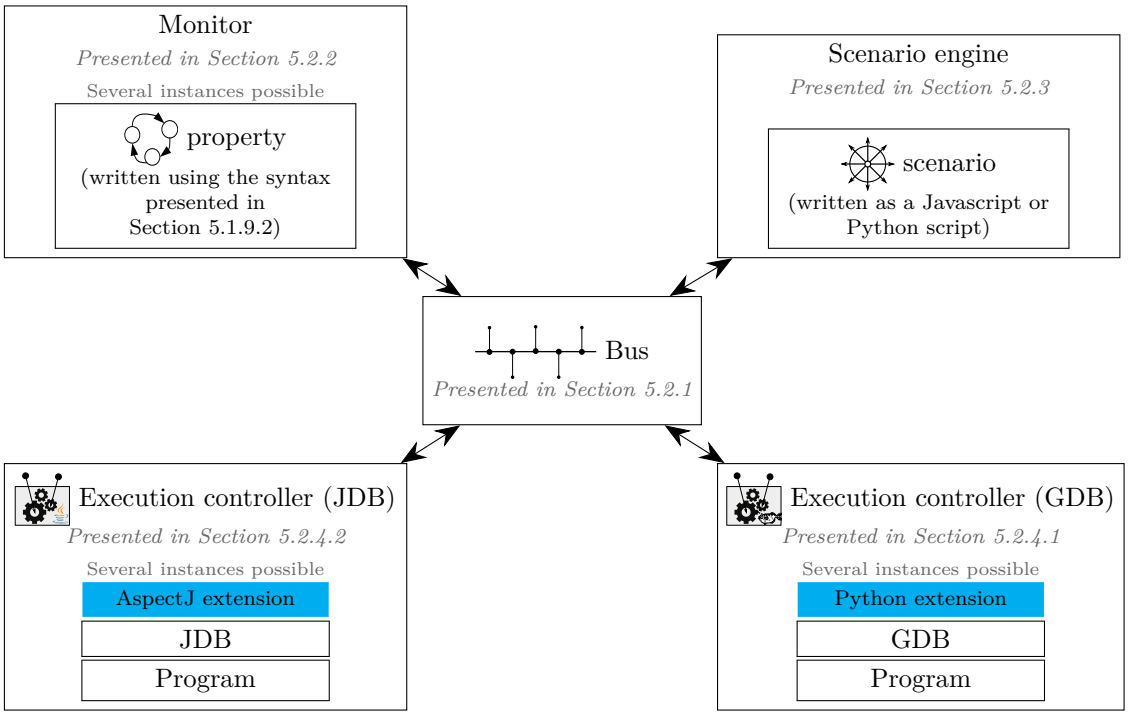


Figure 5.8: Architecture of Dist-Verde. Arrows denote communications between components and the bus. Illustration: gears from vecteezy.com

for structures called messages described in a specification provided by the developer. We chose Protobuf for its ease of use, its network and parsing efficiency, its extensibility, and its compatibility with a variety of programming languages. Our Protobuf specification, provided in Appendix A.2, is around 280 lines of codes (LoC) and defines 29 kinds of message used in our implementation. We provide classes that can be extended to build Dist-Verde-compatible components in Python and Java. Building compatible components in other programming languages is possible by porting these classes (120 LoC in Python, 220 LoC in Java) or by using our Protobuf specification directly.

In Dist-Verde, the bus is a Python program of 300 LoC. As per Section 4.3.1, the bus waits for components to join, forwards messages, advertise joining and leaving components to other components. Components communicate between each other through the bus by sending messages. Communications between the bus and components are assumed reliable: messages arrive in order and are not duplicated. The structure used to communicate between the bus and a component is a message defined in our specification on Line 5. It contains the identifiers of the sender and of the recipient of the message and its content.

Joining the bus. A component joins the bus by sending a message of kind `Hello` (Line 37), containing the version of the protocol, the name (`"monitor"`, `"scenario"` `"executionController"`) and the version of the service, and the name and the version of the implementation. If the component is a monitor, the name of the property can also be sent. If the component is an execution controller, the name of the program and the programming language used to write the program can also be sent. When joining the bus, a component does not have an identifier. The sender and the recipient of the message are set to 0.

When receiving a `Hello` message from a component, the bus assigns a new unique identifier to the component and sends an `Hello` message to the component with this identifier. The bus then stores the `Hello` message from the component with the `sender` field set to the new identifier, and forwards it to each registered component, making them aware of the new component. Then, the bus forwards the `Hello` message of each of these components to the joining component, making it aware of all components currently registered to the bus and their identifier.

Leaving the bus. A component leaves the bus by closing the socket between the bus and itself. The bus then sends a `Bye` message (defined on Line 48), with field `sender` set to the identifier of the leaving component, to each registered component. The bus deletes the `Hello` message of the component.

Bus scripts. To ease automation, the bus provided in Dist-Verde is scriptable using Python scripts. The script provides a function `trigger`, called when certain events happen in the bus. These events include: the bus is initialized, the bus starts waiting for components, a component is joining the bus and a component left the bus. The bus also provides functions that the script can call, such as `setNbMonitors`, used to set the number of monitors that are to be expected by the scenario, and `shutdown`, to stop the bus. Bus scripts can be used to deploy an i-RV session, possibly across different computers, automatically.
In the next sections, we present the different components provided by Dist-Verde: the monitor, scenario and execution controllers as implemented in Dist-Verde.

5.2.2 Verde-Monitor

Verde-Monitor is our implementation of the distributed i-RV monitor presented in Section 4.3.4. To build Verde-Monitor, the monitor of Dist-Verde, we chose to adapt the monitor used in our first implementation. The monitor itself is around 1330 LoC. The event manager that handles communication with the bus is around 690 LoC and is monitor-agnostic. Properties are written using the same syntax as in our first implementation, described in Section 5.1.9.2, with three additions:

- When defining the event of a transition, the event can be targeted at a specific program, or set of programs, by its name.
- Join and leave events can be used. These events are produced when a program joins or leave the distributed i-RV session.
- The identifier of a program in the distributed i-RV-session can be used as an event parameter, using parameter name `#ecid`. This can be used for slicing traces per instances of programs.
- The name of a program in the distributed i-RV-session can be used as a target in properties, to listen to events specific to a program.

The monitor requests events by sending `EventRequest` messages (Line 59), and releases events using `EventRelease` messages (Line 173) to the execution controller. It receives a `RequestOk` (Line 116) message when an event request is acknowledged from the execution controller, and an `Event` message (Line 177) when an event is produced, containing the identifier of the corresponding event requests, and parameters of the event. The monitor receives checkpoint and restore requests from the scenario (`CheckpointRequest`, Line 237 and `CheckpointRestoreRequest`, Line 259), which it grants by responding `Checkpoint` (Line 250) and `CheckpointRestored` (Line 264) messages. The monitor sends verdicts to the scenario using `Verdict` (Line 212) messages. Verdicts contain information about the current state of the property evaluation:
– an optional property name or identifier,
– the identifier of the execution controller that caused the verdict to be produced,
– a flag telling whether the initial state is an active state,
– whether the state is accepting (no, yes, maybe no, maybe yes, unknown),
– a list of current active state names,
– a list of arbitrary values,
– a list of previous state names,
– a list of new active state names.

```
1 scenario.addReaction("(any initial verdict) or (any non-accepting verdict)",
      ↪ lambda: scenario.suspendExecution())
```

Figure 5.9: Scenario written in Python
Scenario written in Python that suspends the execution on non-accepting or initial verdict.

```
1 checkpoints = [];
2 scenario.on("any initial state", () => { checkpoints = []; })
3 scenario.on("all accepting verdict", () => {
4   scenario.createCheckpoint(checkpoint => { checkpoints.push(checkpoint); })})
5 scenario.on("any non-accepting verdict", () => {
6   scenario.debuggerPrompt("Program is wrong. Checkpoints: " +
7   checkpoints + ". Which one should I restore? ",
8   strcpid => {scenario.restoreCheckpoint(parseInt(strcpid),
9   id => print("Checkpoint." + id + " restored"))})})
```

Figure 5.10: Scenario written in JavaScript
This scenario written in JavaScript creates a checkpoint on accepting verdicts and asks the user to choose a checkpoint to restore on non-accepting verdict. Note, if an initial verdict is accepting, both the first and the second reactions apply.

5.2.3 Verde-Scenario

Verde-Scenario is our implementation of the distributed i-RV-scenario presented in Section 4.3.5. The scenario receives verdicts (`Verdict`) messages from the monitor (Line 212). The scenario can send checkpoint (`CheckpointRequest` messages, Line 237) and restore requests (`CheckpointRestoreRequest` messages, Line 259) to the monitors and the execution controllers, debugger commands (`DebuggerCmd` messages, Line 274, containing the command string) as well as event (`EventRequest`, Line 59) and resume requests (`ResumeRequest`, Line 246) to the execution controllers. It also receives events (`Event`, Line 177) and replies to debugger commands (`DebuggerCmdReply`, Line 283) from the execution controllers, and reacts to these events and replies.

In the implementation presented in Section 5.1, the scenario engine is tightly coupled with the monitor and is specific to GDB, making any extraction as costly as rewriting it from scratch. Moreover, a distributed i-RV scenario reacts to sets of verdicts rather than unique verdicts, and must support the presence of several execution controllers. Therefore, we chose to write a scenario engine from scratch. Our scenario engine consists in a Java program composed of two classes: the parser for a language to write formulas describing sets of verdicts (around 600 LoC), and the engine itself (around 900 LoC, plus 130 LoC to communicate with the bus, shared with the execution controller). Examples of formulas are "`any non-accepting verdict`", "`any initial verdict`", "`any verdict with state sink`" and "`(any verdict with state1) or (any verdict with state2)`"[8]; The engine takes a scenario specification as input. This specification is a script written in Python (using Jython [Jyt], a Python interpreter in Java) or JavaScript (using Rhino [Rhi], a Javascript interpreter in Java). This script defines reactions as procedures mapped to formulas describing sets of verdicts. These procedures use an API provided by the engine to query and drive the debuggers. Figure 5.9 presents a scenario that suspends the execution of an execution controller each time it produces an event that leads to a non-accepting verdict. The monitors are checkpointed and restored as well. Figure 5.10 presents a scenario that creates a checkpoint for each set of accepting verdicts and asks the user to choose a checkpoint to restore when a non-accepting verdict is received. An empty file is also a valid scenario: the execution is resumed each time a set of verdict is received.

[8] A monitor can send the name of its active state(s) with the verdict to the scenario, making it possible to define reactions based on the state.

5.2.4 The Execution Controllers

As per Section 4.3.3, an execution controller controls the execution of the program being interactively verified. An execution controller is the combination of the debugger and the program. In Dist-Verde, an execution controller starts the execution of the program, receives event requests (`EventRequest`, Line 59) from other components, sends events (`Event`, Line 177), receives and grants checkpoints and restore requests (`CheckpointRequest`, Line 237 and `CheckpointRestoreRequest`, Line 259) and receives and grants debugger commands (`DebuggerCmd` messages, Line 274, and `DebuggerCmdReply`, Line 283). Event requests and releases may only be done when the execution is suspended. The execution, at the beginning, needs to be suspended to allow initial event requests. Producing an event suspends the execution until a resume request is sent.

We provide two execution controllers: Verde-GDB, based on the GNU Debugger, and Verde-JDB, based on the Java Debugger. Supporting two debuggers let us check that our protocol is indeed usable with different implementations. We chose to reuse our existing work on GDB, to keep supporting C and C++, and to support Java, a widely used programming language. We present Verde-GDB in Section 5.2.4.1 and Verde-JDB in Section 5.2.4.2.

5.2.4.1 Verde-GDB

Our execution controller based on GDB, Verde-GDB, is used to instrument and control programs written in languages supported by GDB like C and C++. Verde-GDB is a straightforward adaptation of our first implementation. We reused the code that drives GDB to instrument and control the program execution (consisting of 600 LoC). Our GDB extension communicates with the bus and use this code to grant event requests, produce events and run GDB commands to control the execution. This extension consists of around 1200 LoC (including the code driving GDB) and allows distributed i-RV for C and C++ programs, and programs written in other languages supported by GDB.

5.2.4.2 Verde-JDB

Our execution controller based on JDB, Verde-JDB, is used to instrument and control programs written in languages running on the Java Virtual Machine, like Java. JDB [JDB] is a command line debugger written in Java that has basic debugging features. JDB is provided in the Java Software Development Kit as a reference implementation for the Java Debugging Interface. Other Java debuggers like those found in common Java Integrated Development Environments (IDEs) provide more features. However, to our knowledge, contrary to GDB, which provides a Python Application Programming Interface, none of these debuggers provide a straightforward way to develop extensions. Thus, adding support for distributed i-RV on these debuggers requires modifying their code. We chose JDB for simplicity: the code of JDB is small and not tied to a large IDE. Unfortunately, JDB does not provide any extension mechanism natively. Instead of modifying the code of JDB, we chose to extend it using [ASP]. AspectJ provides Aspect Oriented Programming (AOP) [KLM+97] to Java. AOP consists in specifying instructions to run at defined points in the code being extended. This allows modifying the behavior of JDB without maintaining our own fork of the tool. In this way, our tool can run on top of JDB unmodified, will work with newer versions of JDB with minimal adaptations (assuming the code of JDB is not significantly re-architectured), and our modifications to JDB are clearly distinct from its original code. Our JDB extension is composed of several parts: a debugger-agnostic interface, an AspectJ file that implements this interface and a controller using this implementation. The *debugger-agnostic interface* provides several features:

- `getValues(parameters) -> values`: returns the values corresponding to a list of variable given in `parameters`.
- `instrument(point, callback) -> requestId`: registers an event request specified by `point`, returns its identifier and calls the given `callback` when the event happens. The callback returns `true` to stop the execution.
- `release(requestId)`: releases event request `requestId`.

- `run(cmd, callback)`: runs command `cmd` in the debugger and calls the given `callback` with the result as parameter.
- `prompt(q, callback)`: asks question `q` to the user and calls the given `callback` with the user input as parameter.
- `print(p)`: prints value p to the user.

The *AspectJ file* (around 700 LoC) is tightly coupled with the code of JDB and implements the debugger-agnostic interface. This extension creates breakpoints and watchpoints by using code available in JDB. Our extension also monitors breakpoint and watchpoints created by JDB itself by monitoring calls to methods `addEagerlyResolve` and `delete` of class `EventRequestSpecList`. When breakpoints or watchpoints are reached in the program, method `EventHandler.handleEvent` in JDB is called. The execution is suspended or continued depending on the return value of this method. Our extension replaces this method using an aspect, finds out which points are triggered. If some points correspond to some requested instrumentation, the relevant callbacks are called. If a point is set by the user, the execution is suspended. Otherwise, the execution is suspended if one of the callbacks returned `true` Our extension uses the Java Debugging Interface to retrieve requested values. To run debugger commands, our extension mimics and calls code from JDB so they are executed as if the user entered them.

The *controller*, using this debugger-agnostic interface to implement an execution controller for distributed i-RV, does not depend on JDB and can be reused with other debuggers. The controller connects to the distributed i-RV bus, grants event and debugger requests, forwards events and more generally implements the specification of an execution controller by using the debugger-agnostic interface.

5.2.5 Checkpointing

Checkpoints are a core feature of i-RV. They allow exploring different paths from the same execution by going back in a previous state and changing values in the program. In i-RV, checkpointing is considered as a feature provided by the monitor and the execution controller. In practice, the monitor provided by Verde supports checkpointing, but many monitors do not support checkpointing. GDB also provides experimental checkpointing support for monothreaded programs using `fork`. Verde provides checkpointing functionality by integrating CRIU [CRIa] with GDB. The JVM and JDB does not provide checkpointing functionality natively. We wrote a Python program (around 300 LoC) that provides limited[9] checkpointing functionality for runtimes (like the Java Virtual Machine) without native support for checkpointing. This program intercepts messages between the bus and the execution controller and uses CRIU to checkpoint and restore the execution controller, maintaining the connection with the bus, and re-establishing the connection between itself and the execution controller. This program may also be used to add this feature to monitors without native support for checkpointing.

5.2.6 Usage

A distributed runtime verification session can be started either by running each component manually, or by using a script doing it automatically. We first present how to run each component manually. We then present `dist-verde`, a script provided for convenience to run an i-RV session on one computer.

Running the bus. To run a distributed i-RV-session, the bus must first be run. This is done by calling the program `verde-bus`. This program runs the Python script `irvbus.py`. The bus will open a UNIX socket at a default location and a local TCP socket on a default port. This location, port and the host of the TCP socket can be set using environment variables `VERDE_BUS_PORT`,

[9] Checkpointing processes writing files or accessing the network is not supported.

`VERDE_BUS_HOST` and `VERDE_BUS_UNIX_SOCKET`. If the environment variable `VERDE_BUS_AUTO_PORT` is set and the specified or default port is unavailable, the bus will choose another, available, port.

Running the monitor and the scenario. The monitor (resp. scenario) is run using program `verde-monitor` (resp. verde-scenario), providing the property file (resp. scenario) as the first parameter. The monitor will connect to the bus using the default port on the local host, unless otherwise specified using variables environment `VERDE_BUS_UNIX_SOCKET` or `VERDE_BUS_PORT` and `VERDE_BUS_HOST`.

Running Verde-GDB and Verde-JDB. Verde-GDB is run either like gdb, just replacing the executable name `gdb` by `verde-gdb` in the terminal, or by providing the program to debug and its parameters. Likewise, Verde-JDB is run like JDB, replacing the executable name `jdb` by `verde-jdb` in the terminal. The execution controller will connect to the bus using the default port on the local host, unless otherwise specified using variables environment `VERDE_BUS_UNIX_SOCKET` or `VERDE_BUS_PORT` and `VERDE_BUS_HOST`.

Running a distributed i-RV session using `dist-verde`. We provide a program, `dist-verde`, to run a distributed i-RV session using an unused port (meaning that several sessions can be run at the same time). The developer provides the scenario, the list of properties to check and the list of programs to run, and will run the necessary components. If several programs and properties are specified, several terminals will be opened so the developer can control each program individually. A property may be made quiet to prevent Dist-Verde from opening a terminal for it. If no scenario is provided, the default scenario is used. This scenario suspends the execution when a non-accepting verdict is produced by a monitor.

Basic usage of this script is as follows. More advanced usage is covered by the help message provided in the program (`dist-verde --help`).

```
1  dist-verde [--scn scenario-file<.py|.js>] [--prop property-file.prop [-quiet]]
       ↪ ... [--prog -dbg <gdb|jdb> [-arg ARG1] ... ] ...
```

More advanced usage using bus scripts. A custom distributed i-RV session can be set up using a bus script, as described in Section 5.2.1. Bus scripts can be used to use a scenario or a monitor that is not provided or supported in Dist-Verde and to run a distributed i-RV in a custom setting such as a networked environment, across different computers. A bus script is run by providing its name as the first parameter of (verde-bus) (`verde-bus bus-script.py`). Additional parameters can be given. These parameters will be provided to the bus script. Under the hood, `dist-verde` is implemented by using a bus script to set up the local distributed i-RV session. This bus-script can be used as a reference when building other bus scripts.

5.3 Conclusion

In this chapter, we presented two implementations for interactive runtime verification. These two implementations reflects contributions presented in Sections 3 and 4. They are maintained in the same codebase. The first implementation is an extension for GDB. The second implementation is a set of components that communicate with a bus using a protocol specified using Protobuf through sockets. This protocol allows extending Verde with components that are not (yet) bundled with it, to support existing monitors and debuggers. The two implementations have a lot of code in common, notably the monitor and the instrumentation module for GDB. We presented the property model used in our monitor, and how to use these implementations. In Chapter 6, we present the experimental results we produced using these implementations.

Chapter 6

Experiments

Contents

In this chapter, we report on several experiments carried out with Verde to measure its usefulness in finding and correcting bugs and its efficiency from a performance point of view[1]. We discuss the objective and possible limitations (threat to validity) of each experiment. These experiments also illustrate how a developer uses Verde in practice. We first present experiments related to the framework presented in Chapter 3, conducted using the first version of Verde (see Chapter 5.1). We then present experiments related to the distributed framework presented in Chapter 4 conducted using Dist-Verde (see Chapter 5.2).

6.1 Monolithic I-RV

In this section, we present a set of experiments related to the framework presented in Chapter 3 using the first version of Verde (see Chapter 5.1).

6.1.1 Fixing 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[2]. We did not discover the bug using i-RV[3]. However, it helped us determine its origin in the code of Zsh and fix it. A fix has since been released.

6.1.2 Automatic Checkpointing to Debug a Sudoku Solver

We evaluated i-RV by mutating the code of a backtracking Sudoku solver[4]. 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 9×9 board such that each row, each column and each 3×3 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 (checkpointing only introduces moderate slowdown), and precise instrumentation for a relevant part of it.

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

[2] The code of the property is in Appendix A.3. We worked on commit 85ba685 of Zsh.

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

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

The solver is bundled with several example boards that it solves correctly. We mutated its code using `mutate.py`[5] to artificially introduce a bug without us knowing where the change is. When ran, the mutated program outputs "bad board". We ran it with i-RV. The property enters the state `failure_load`. When restoring a checkpoint and running the code step by step in the function that loads a board, the execution seems correct. The code first runs one loop reading the board using `scanf` by chunks of 9 cells, and then a second loop iterates over the 81 cells to convert them to the representation used by the solver. Setting breakpoints and displaying values during the first loop exhibits a seemingly correct behavior. During the second loop, the last line of the board holds incorrect values. Since we observed correct behavior for the first loop and the 72 first iterations of the second loop, and since both loops do not access the board in the same way, we suspected a problem with the array containing the board. We checked the code and saw that the mutation happened in the type definition of the board, giving it 10 cells by line instead of 9.

A caveat of this experiment is that we had to choose the mutated version of the code such that the code violates the property. We also introduced a bug artificially rather than working on a bug produced by a human and arguably, the program is small enough to be debugged using a traditionnal 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.

6.1.3 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 Figure 2.5) 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 Figure 5.6. We ran the program with Verde. The execution stopped in the state `sink` (defined at line 39 of Figure 5.6). 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 experiment 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. This bug could be detected without Verde using Valgrind (showing erroneous memory accesses) or because the program may eventually segfault. However, Valgrind incurs a big slow down, and segmentation faults are not systematic in such cases.

6.1.4 Micro-Benchmark

In this experiment, we evaluated the overhead of the instrumentation in function of the temporal gap between events. We wrote a C program calling a NOP function in a loop. To measure the minimal gap between two monitored events for which the overhead is acceptable, we simulated this gap by a loop of a configurable duration. The results of this benchmark using a Core i7-3770 @ 3.40 GHz (with a quantum time (process time slice) around 20 ms), under Ubuntu 16.04 and Linux 4.4.0-109 with GDB 7.11, are presented in Figure 6.1.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

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

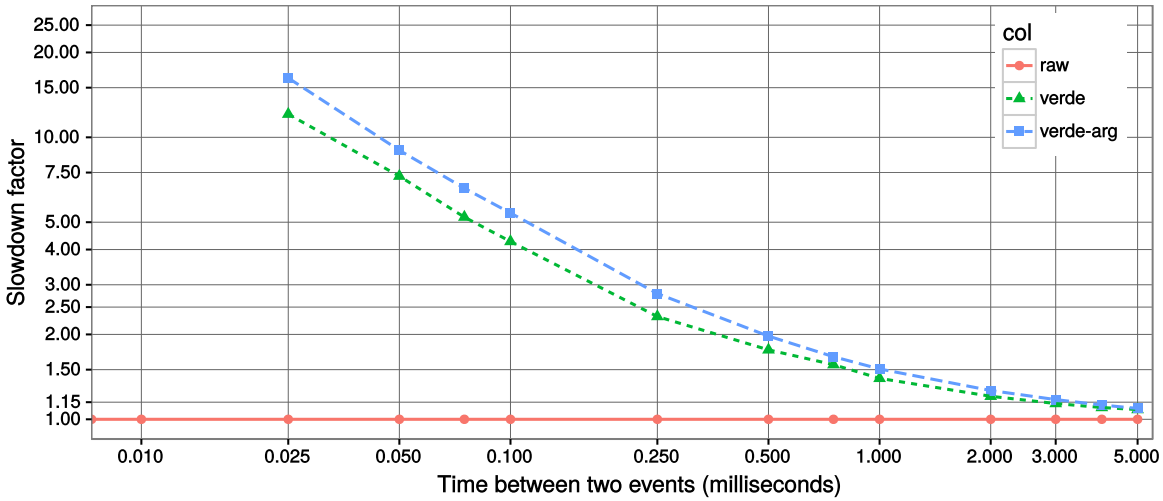


Figure 6.1: Instrumentation overhead with Verde.
`raw` (red) is the execution of the program without verde. `verde` (green) is the execution with verde. The overhead includes the breakpoint-based instrumention. Running the program in GDB per se does not incur any slowdown: programs run in native speed when using interactive debuggers as long as no instrumentation (points) is used. `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.

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[6] 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.

6.1.5 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 Figure 6.2 (axes use a logarithmic scale). The number of queues is represented on 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.

6.1.6 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

[6]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 reveiled in January 2018.

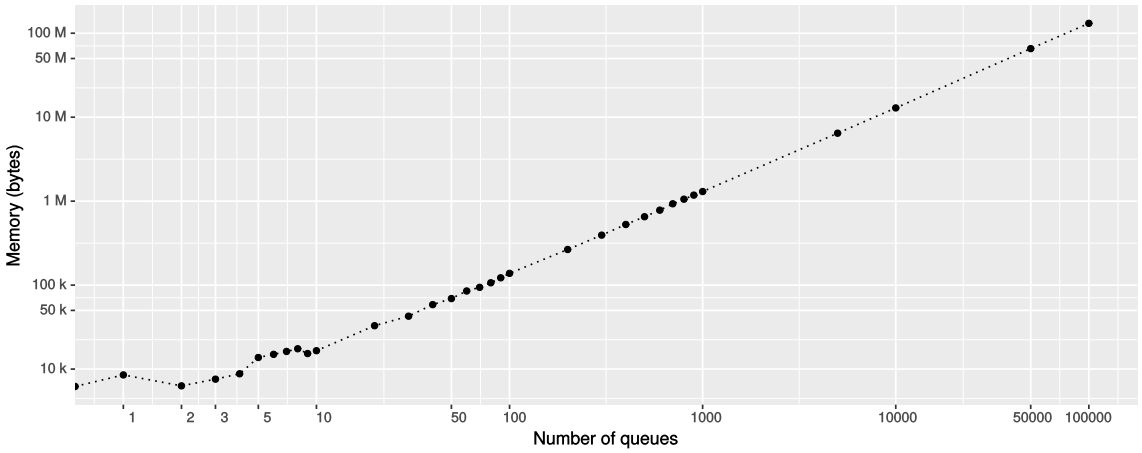


Figure 6.2: Memory consumption per number of tracked queues.

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 Section 6.1.4: 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 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 Chapter 8, we present possible ways to mitigate these limitations.

6.1.7 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.

	RAM				HDD			
	Checkpoint		Restore		Checkpoint		Restore	
Process size (MB)	Avg (s)	±	Avg (s)	±	Avg (s)	±	Avg (s)	±
1	0.0205	0.0026	0.0469	0.0134	0.0215	0.0048	0.0441	0.0129
5	0.0243	0.0148	0.0490	0.0216	0.0238	0.0033	0.0447	0.0220
10	0.0237	0.0052	0.0438	0.0106	0.0250	0.0021	0.0527	0.0260
25	0.0274	0.0008	0.0524	0.0230	0.0337	0.0167	0.0505	0.0141
50	0.0348	0.0046	0.0556	0.0107	0.0435	0.0031	0.0593	0.0151
75	0.0433	0.0057	0.0678	0.0267	0.0545	0.0054	0.0626	0.0117
100	0.0494	0.0067	0.0766	0.0173	0.0674	0.0101	0.0685	0.0123
250	0.0918	0.0063	0.1086	0.0216	0.1370	0.0236	0.1101	0.0229
500	0.1732	0.0688	0.1796	0.0191	0.2508	0.0475	0.1716	0.0107
750	0.2347	0.0197	0.2454	0.0220	0.3645	0.0749	0.2360	0.0206
1000	0.3018	0.0132	0.3098	0.0805	0.4839	0.1323	0.3018	0.0431

Table 6.1: Time to checkpoint and restore
Average time to checkpoint and restore in function of the size of the process, when checkpoints are saved on a Hard Disk Drive or in RAM.

6.1.8 Cost of Checkpointing

In this experiment, we measured the cost of checkpointing a process running in GDB with CRIU. This measure is not directly related to Verde, rather on the tools it uses for checkpointing.

We wrote a C program that allocates a specified amount of memory. We set a checkpoint and restarted it ten times for different sizes, once using the memory as storage for checkpoints, once using a regular hard drive (see Table 6.1). We noticed that checkpointing leads to acceptable costs for debugging purposes, ranging from 0.02 seconds for a 1 MB process to 0.3 seconds for a 1 GB process when storing checkpoints in RAM. We also noticed that the impact of using a Hard Disk Drive for storing saved checkpoints is limited compared to a storage in RAM[7]. The incremental checkpointing feature of CRIU is expected to improve performance when checkpointing a process several times.

6.1.9 Conclusion

In this section, we presented eight experiment.

The first one reports on the usage of interactive runtime verification on a project, Zsh, affected by a bug that was not fixed at the time of the experiment. We were able to study the bug systematically, using the property as a log of the observations done while studying the bug. We submitted a patch to the project. The developers followed-up with an alternative, cleaner fix.

The second experiment explores the usage of checkpointing in interactive runtime verification. The checkpoints allow studying a restricted segment of the faulty execution, easing the track of the cause of the bug.

The third experiment illustrates the usage of Verde on a multi-threaded program. While our theoretical framework does not model multi-threaded programs, Verde has a limited support for it, thanks to GDB's own support for multi-threaded programs. Verde checks for queues overflows and points to the faulty line of code, before the bug appears.

The fourth experiment is a micro-benchmark giving insight on the overhead of Verde. This overhead depends on the performance of breakpoint handling, which depends on the CPU, its architecture, the operating system and the interactive debugger. The overhead on our machines is prohibitive if the gap between the events to capture is under 0.5 ms and not significant over 10 ms. The acceptability of slowdown depends on the case and the time gained by using interactive runtime verification (yet to be evaluated at the time of writing) needs to be taken in account.

[7]This is probably due to caching mechanisms provided by the operating system.

The fifth experiment gives insight on the memory consumption of Verde. This memory consumption should not be an issue in most real-world usage of Verde.

The sixth experiment gives our empiric assessment of the user-perceived slowdowns of Verde in several cases. Properties that require capturing events occuring on each frame of a video (in a video game, or a video player) do not cause any observable slowdown. Monitoring usage of iterator in a graphical application based on a toolkit using iterators intensively may, however, lead to unpleasant slowdowns.

The seventh experiment illustrates a benefit of dynamic instrumentation, where disabling instrumentation for unused events lead to a significantly smaller overhead.

The eighth experiment gives insight on how checkpointing can be realistically used with Verde. While checkpointing every few seconds is conceivable, a smaller gap between checkpoints would incur an important slowdown.

In the next section, we present several experiments related to distributed interactive runtime verification, using Dist-Verde.

6.2 Distributed I-RV

We evaluate distributed i-RV using Dist-Verde in various situations: deadlock detection (Section 6.2.1), checking the respect of requirements (Section 6.2.2) and checking the correctness of a distributed algorithm running on different computers with respect to several properties (Section 6.2.3). Unless stated otherwise, we ran experiments[8] on a laptop with a Core i5-6300U CPU @ 2.40GHz running Debian Buster, Linux 5.0.7.

6.2.1 Detecting Deadlocks in MPI Applications

In this experiment, we use distributed i-RV to study how the execution of a distributed program leads to a deadlock situation because of communications. A deadlock is a situation where there exists a set of tasks waiting for a message from another task in this set. Deadlocks are a widespread class of bugs in distributed applications that are notoriously hard to study. MPI is a standard to write distributed programs composed of several communicating tasks, mainly used in High Performance Computing. We consider an incorrect solution to the dining philosophers problem written using MPI (the Message Passing Interface). The dining philosophers is a well-known problem used to illustrate synchronization issues. A number n of philosophers sit at a round table, each behind a plate. One fork is placed between each pair of adjacent philosophers. Philosophers are either eating or thinking. They can eat only when they picked both of the two fork next to them. They can pick or release adjacent forks while thinking, and cannot pick a fork that a neighbor picked and has not released yet.

In the MPI program, tasks with an odd (resp. even) MPI rank (identifier) represent philosophers (resp. forks). Forks wait for messages from philosophers. Any philosopher p picks fork f at the left ($f = p - 1$) or at the right ($f = p + 1$) by sending message `request` to task f and then waits for a response. When fork f is available, task f sends message `ok` to philosopher p. Philosopher p releases the fork f by sending message `release` to task f. The solution we consider is such that each philosopher p requests fork $p - 1$, then requests fork $p + 1$, then eats for 100ms, then releases fork $p + 1$, then releases fork $p - 1$, then repeats this procedure.

We wrote a property on the absence of cycles in MPI communications. To detect deadlocks, the monitor builds a directed graph in which tasks are connected to the tasks from which they are waiting a message by tracking calls to MPI primitives Recv, IRecv, Waitany and Send. Each time an edge is added to the graph, the monitor looks for a cycle of tasks waiting for messages from each other. When such a cycle is detected, the property enters the deadlock state. The call history of Recv and Irecv is also stored in the monitor state to provide insights on the construction of the deadlock.

[8] See the provided artifact to run these experiments.

P3. No account may end up with a negative balance after being accessed.

P4. An account approved by the administrator may not have the same account number as any other already existing account in the system.

P5. Once a user is disabled by the administrator, he or she may not withdraw from an account until being activated again by the administrator.

P6. Once greylisted, a user must perform at least three external transfers before being whitelisted.

P7. No user may request more than 10 new accounts in a single session.

P9. A user may not have more than 3 active sessions at once.

P10. Transfers may only be made during an active session (i.e., between a login and a logout)

Figure 6.3: Seven properties for the banking transaction system.

Properties	**Events**	**Time**	**Overhead**
5	25	1.22 s	811 % (13 %)
6	31	1.25 s	830 % (16 %)
4, 5, 6	1,788	4.62 s	3,341 % (328 %)
2, 4, 5, 6	3,552	7.13 s	5,217 % (562 %)
2, 4, 5, 6, 7, 9	10,567	14.15 s	10,450 % (1,213 %)
2, 4, 5, 6, 7, 9, 10	18,899	22.90 s	16,966 % (2,024 %)

Table 6.2: Number of events, execution time and overheads for different sets of checked properties. The first overhead is computed compared to the native program execution, and the second, compared to a program execution in JDB.

We ran the simulation with 8 tasks (4 philosophers) using Dist-Verde. When a deadlock occurs, the call history is shown and the task that made the last call involved in the deadlock is suspended, ready for inspection in GDB. The other tasks can be suspended and inspected by typing CTRL+C in GDB. Contrary to timeout-based techniques, this solution to deadlock detection is not subject to false positives.

6.2.2 Banking Transaction System (RV'14)

We assess the applicability of distributed i-RV for checking the conformance with multiple requirements of a program at runtime. We measure the overhead of Dist-Verde in such a setting. We study a benchmark faking a banking transaction system written in Java from the RV'14 competition for monitoring tools [BS14]. Ten properties are provided with the benchmark. We translated seven of them in the formalism used in Verde[9] (see Figure 6.3 for an informal description of these properties). The benchmark contains a test suite, each test being designed to violate or respect one of the provided properties. Without instrumentation, the benchmark takes 135 ms to run outside of JDB, and around 1.1 s in JDB. We run the benchmark with the 7 properties. The program execution completes in 21 seconds. The scenario prints all the violations with the exact location of the bug in the program source code. We found seven bugs, each property being violated once. In Table 6.2, we present the number of produced events as well as the execution times of the program and the overhead with Dist-Verde for different sets of checked properties. The execution times are correlated with the number of events produced during the execution (rather than the number of properties).

[9]The three remaining properties require features that were not yet provided in Dist-Verde at the time of writing: instrumenting Java method exits and calling a program function. Dist-Verde now provides them.

6.2.3 Leader Election with LogCabin

This experiment shows the applicability of our approach for runtime checking the correctness of distributed algorithm implementations. LogCabin[10] is an implementation in C++ of the Raft consensus algorithm [OO14].

We wrote 4 properties relative to the leader election: during each term, all processes in a cluster must agree on the same leader (agreement); consensus is reached at some point (i.e., all processes reach the highest known term; termination), a process becomes leader if a process voted for this process (integrity), and only one process becomes a leader. These four properties, reproduced in Appendix A.4 are checked by monitoring calls to method printElectionState of class RaftConsensus, called when the election evolved. Method updateLogMetadata, monitored for the integrity property, is called in any process sending a proposal (stored in field votedFor). Method becomeLeader is monitored for property 4. First, we ran LogCabin on three different computers and verified that the four properties were verified. Killing one out of the three processes triggers a new leader election, which can be observed by looking at the states of the four monitors. We then set up a cluster of 20 processes on the machine described in Section 4.4.4, which we run 20 times with and without Dist-Verde. We recorded the time taken by each process to reach the end of an election. The average time to reach the end of an election was 1.72 seconds with Dist-Verde, and 1.59 seconds without, corresponding to an overhead of 7.82%.

This experiment exhibits a case in which distributed i-RV has a limited impact on the execution time because few events are needed to check the properties, while providing a powerful way of studying the execution of a distributed system.

6.2.4 Conclusion

We presented three experiments on distributed i-RV.

In the first experiment, we show how distributed i-RV can be used to detect deadlocks in a distributed system and debug them interactively, without the need for adding manual logging (print) statements.

In the second experiment, we use distributed i-RV on a benchmark that was used in a RV competition. This benchmark is provided with ten properties, and a set of tests, each of which breaks one property. We were able to run a distributed i-RV session on these tests, with seven of these properties, detect each bug and study them interactively.

In the third experiment, we run a distributed i-RV session to check a network of LogCabin nodes against 4 common properties in distributed system related to leader election. Dist-Verde incurs a low overhead while still allowing the study of the election process.

As observed in the previous section, debugger-based instrumentation is costly performance-wise; we observe high overheads when checking properties requiring instrumentation for often-called functions. This may prevent using distributed i-RV for certain properties. However, distributed i-RV can realistically be used in various settings. It simplifies the use of debuggers and the runtime verification of several properties on distributed systems as well as on simpler programs, and a high overhead is acceptable as long as it does not slow down a developer too much while debugging.

[10] `https://github.com/logcabin/logcabin` (commit ee6c55a, 27 Jul 2017).

Chapter 7

Related Work

Contents

I-RV is related to several families of approaches for finding and fixing bugs. In this chapter, we give an overview of 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.

7.1 Testing

Manual testing. Most obvious bugs can easily be spotted during the development of the software by testing it manually. Modifications to the code are manually tested, by the developers and possibly by a team responsible for testing the software [IML09]. Manual testing can be tedious. I-RV aims to reduce 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 [IML07]. 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, [CL02] 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).

7.2 Static Analysis and Abstract Interpretation

With static verification approaches (e.g., static analysis [CC77]), 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. Model checking [EC80] is a form of static analysis. It is an automatic verification approach for finite-state reactive systems. Model checking consists in checking that a model of the system verifies temporal properties [CGP01]. Model checking 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.

Static verification approaches are usually limited in the expressiveness of the properties they can check and are complementary to dynamic verification approaches. Complementary to static analysis, dynamic verification approaches allow checking more complex behavioral properties. 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.

7.3 Interactive and Reverse Debugging

Tools used in interactive debugging [Pou14, PSK+16a] 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. We compared an interactive debugging session with an interactive runtime verification session in Section 2.5.

We provide i-RV with an execution model sufficiently detailed for the purpose of ensuring guarantees about its correctness (see Section 3.4). Work has been done to focus on the modeling of program execution, compilers, and debuggers, to ensure correct and intuitive behavior of the debugger [dS92], more suitable to implement and reason about interactive debuggers.

Reverse debugging [Eng12, GM14, RCM93] is a complementary debugging method. A first execution of the program showing the bug is recorded. Then, the execution can be replayed and reversed deterministically, 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 restoring the execution in a previous state using checkpoints, with the help of the monitor and the scenario, adding a level of automation. I-RV can be adapted to use reverse debugging instead of checkpointing, but the ability to modify the execution to try a different path would be lost.

7.4 Instrumentation Techniques and 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. I-RV leverages verdicts issued by the monitor. Runtime verification is a building block of i-RV.

Compile-Time Instrumentation. RiTHM [NJW+13] 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 [NS07] is a tool that leverages DBI and can interface with GDB. It features detection of memory-related defects. Dr. Memory [BZ11] is another similar tool based on DynamoRIO [BZA12]. DynamoRIO and Intel Pin [LCM+05] are both DBI frameworks that allow writing 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 [Kic02, Kat06] used to capture events. The Jassda framework [BM02], which uses CSP-like specifications, LARVA [CFG11] and JavaMOP [CR07] 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 [JMLR12] 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 [CR09]. In our work, events are dispatched in slices in a similar way. We do not implement all the concepts defined by [CR09] 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 [Fal10]. 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 [DJ01], Opium [Duc99] 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 modifying the execution.

Frama-C [CKK+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.

7.5 Distributed Runtime Verification

Distributed runtime verification [FHR13, BF18b] relates to runtime verification of distributed systems, as well as to runtime verification using distributed monitors. Work has been done to split the process of property evaluation into multiple monitors [MB15, EF18], Some work also consider crashing monitors [BFR+16]. While our distributed framework for i-RV allows checking one or several programs against multiple properties using independant monitors, these approaches allow checking complex properties using multiple communicating monitors, to distribute the cost of evaluation and to combine simpler properties to build more complex ones. Our work does not specifically consider distributed monitoring an crashing monitors. In Distributed i-RV, distributed monitoring, possibly with crashing monitor may be abstracted away by using a component that provides an interace of a regular monitor, i.e., the distributed monitors provides event requests done by the component, event notifications are dispatched to the distributed monitors and produced verdicts are transmitted to the scenario through the component. However, while the property evaluation itself would still be distributed, this component would centralize event requests and notifications, and dispatch them to the relevant execution controllers and "partial" monitors. Decentralizing the instrumentation would require major changes to our architecture to allow peer to peer communication between the execution controllers and the partial monitors.

7.6 Trace-Based Debugging of Distributed Systems

The challenges of trace-based debugging of distributed system are numerous. The size of traces and their number (in case of distributed system composed of many nodes) can be overwhelming and, therefore, difficult to interpret and understand. Instrumentation may also affect the behavior of a distributed system by introducing delays.

Pivot Tracing [CD08] instruments distributed system using dynamic instrumentation and allows efficiently querying the generated traces by implementing an operators suitable for distributed systems, the happened-before join operator, based on the happened-before relation [Lam78]. This happened-before join helps analyze the state of a distributed system from lengthy traces. A monitor implementing using the happened-before join operator may be implemented in a monitor as part of the property evaluation, and in distributed i-RV using debugger-based instrumentation instead of dynamic instrumentation. Dynamic intrumentation is less invasive than debugger-based instrumentation. This limitation may be overcome by being more specific when specifying what event to instrument, and adpating this instrumentation depending on the current evaluation of the property, which is not usually done with dynamic instrumentation, where the same events are traced during the whole execution for a given tracing session. [LSZE11] presents an approach to debug distributed systems by recording the execution and building relations from the records that can be queried in an SQL-like language.

D^{3}s [LGW+08] is another approach to debugging distributed system. A compiler rewrites binaries to instrument function calls for some given predicates. This instrumentation (the state exposer) sends data to a central verifier.

Work has been done on integrating testing with debugging of parallel and distributed system [LCK+97]. This approach relies on tracing a distributed system, and allows replaying and inspecting the trace with a symbolic debugger.

In [MMM06], a distributed system composed of similar nodes is debugged by comparing traces. Failures are detected by two means. A first approach consists in detecting fail-stop failures by detecting traces that stopped earlier than the rest of the traces, assuming nodes with a well-synchronised time. A second approach consists in detecting outliers: traces that don't look similar to the other traces. The comparison may include known good traces from previous executions to avoid false positives concerning a master node, that may, by design, behave differently than the other nodes. Distributed i-RV handles distributed systems composed of nodes that can be different, but requires to select or design properties specific to the system under debug, or to the libraries it uses.

7.7 Debugging of Distributed Systems From Message Traces

Debugging distributed system is often done by analyzing traces of exchanged messages. Work has been done to check properties over these messages [TJFY10]. Meld [TJFY10] can be combined with Distributed i-RV by writing a Meld-based component for Distributed i-RV. This component will use debugger-based instrumentation instead of binary instrumentation, which may allow deciding what message are to be monitored dynamically, instead of monitoring all the messages inconditionally, depending on the property to check.

Work has been done to vizualize messages in a distributed system [ROM+18]. Distributed i-RV does not allow vizualizing messages as-is. Distributed i-RV is focused on interactive debugging distributed system, with the assistance of runtime verification. However, extending distributed i-RV with a message vizualization component should be straightforward, as long as messages can be intercepted using debugger-based instrumentation.

[KCD+97] propose an intergrated approach, GRADE, in which the program is written in a custom, graphical language, using an integrated environment providing a program editor, a debugger and tools to monitor and visualize what is happening in the parallel program, leveraging the high level of abstraction used to design the program. Distributed i-RV does not provide such an integrated experience while developping and debugging programs. However, it allows debugging programs written in mainstream languages and allows developers to chose their development environment.

7.8 Fault Detection Based on Statistical Models

Some approaches are based on learning the expected behavior of a distributed system in a first execution phase, and detecting faults in another execution phase by noticing this execution does not behave in the same way.

With AutomaDeD [BLB+10, LGdS+11], a Semi-Markov Models (SMMs) of the distributed system's nodes is built during a first "learning", known-good execution phase. Then, in the testing phase, nodes that do not match any model built in the first phase are considered buggy, and faults can be localized by finding the point at which the model of the execution of the buggy nodes starts being different compared to the closest model.

Vrisha [ZKB11] is concerned about bugs related to scaling. A first phase, using a limited number of node, is used to observe a bug-free execution. Instrumentation is used to build a statistical model of the execution. Bugs are then studied in the testing phase, with a greater number of nodes, in which scaling bugs appear. Bugs are detected when the execution of a node does not match the statistical model built in the first phase.

Rather than statistical models, distributed i-RV uses runtime property checking to detect bugs, making it complementary to these approaches. It also allows inspecting a faulty node using interactive debugging to get insight of the cause of the bug.

7.9 Interactive Debugging Distributed Systems

ARM DDT (formerly Allinea DDT) is a graphical parallel debugger [LW12] based on GDB to debug parallel and distributed applications in the High Performance Computing field. ARM DDT provides a wide range of features to visualize and agregate data from multiple processes, as well as study memory related issues in distributed programs. It also provides the ability to attach an interactive debugger to a particular node if needed, e.g., to check whether a seamingly stuck process is deadlocked, and to study the cause of this deadlock. Interactive runtime verification does not provide such features. However, the distributed architecture presented in Chapter 4 may be extended with a component providing agregate visualization of runtime data with no major changes, using instrumentation provided by interactive debuggers. Contrary to i-RV, to our knowledge, ARM DDT does not offer an automatic fault detection mechanism. The two approaches are complementary.

7.10 Rolling Back Distributed Systems

Work has been done on the field of checkpointing and rolling back distributed systems. For instance, [FMT18] defines a technique to reverse computation for Erlang programs. Distributed i-RV allows checkpointing and restoring checkpoints at the process (a node) level in a distributed system. Our approach does not specifically define a way to coordinate a consistant roll back of the whole (or a part of the) distributed system. It is up to the scenario, considering a specific kind of applications, to define such a roll back and apply techniques to roll back a set of processes consistantly in each specific case.

Chapter 8

Conclusion and Future Work

Contents

8.1 Conclusion

In this thesis, we introduced interactive runtime verification, an approach to ease a part of the software development process. Interactive runtime verification combines runtime verification and interactive debugging to combine the rigor and systematic aspects of runtime verification to the interactiveness of interactive debugging. Interactive runtime verfication provides a mechanism, the scenario, to react to verdicts produced by monitors. These reactions may used to control the execution. Tools to control and instrument the execution are provided by interactive debuggers. Interactive runtime verification takes 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. Interactive runtime verification allows starting the interactive exploration of a bug closer to its cause, before its manifestation by replacing a part of the tedious manual interactive debugging process. Breakpoint are automatically requested by the monitor to produce a trace to check properties describing the expected behavior for the program. The study of bugs starts at the point in time during the execution where these properties are violated, before a crash occurs or a bad result is shown, rather than at the moment of the crash.

In this thesis, we presented two frameworks for interactive runtime verification. The first framework 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. These models are backed up with algorithmic descriptions aimed at easing implementations of i-RV. 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. Its strength are this formal aspect, as well as the simplicity in how the components are bound toghether. This simplicity also has a cost: the number of monitors is fixed and multiprocess programs are not handled. The distributed framework addresses both issues, bringing interactive runtime verification to systems composed of multiple processes and complex programs more easily verified against multiple properties. It provides an interface and a communication protocol permitting the use of existing monitors and debuggers for interactive runtime verification.

We provide an implementation, Verde, for both approaches and a set of experiments to evaluate and experiment with interactive runtime verification. Our experiments show that even though performance issues may prevent the use of interactive runtime verification in certain cases, it remains relevant in cases where properties do not require the production of a high number of events per time unit. We showed that interactive runtime verification may be applicable with software such as video games and video players running at a rate of 60 frames per seconds, and distributed systems. In the next section, we present ideas to further improve performance, as well as features we want to explore and improvements we want to carry out.

8.2 Future Work

We present some perspectives opened by this work.

Studying and modelling usage of an interactive debugger. With interactive runtime verification, we provide the basic ingredients to automate and ease interactive debugging using runtime verification. Further work on this field requires deeper knowledge on how developers use interactive debuggers. Usage patterns should be studied in order to build a really convincing approach to further automate interactive debugging. Knowing and understanding these patterns will allow the design of scenarios that will take the right decisions on the control of the program execution.

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 into account yet for technical reasons. They might be of interest to check properties on drivers or programs dealing with hardware.

Instrumentation. Handling breakpoints is costly [CMP15] and handling watchpoints even more so. Code injection could provide better efficiency [NJW+13, MVT+16] by limiting round trips between the debugger and the program while keeping the current flexibility of the approach.

Stronger checkpointing. We provide basic checkpointing functionality. This feature merits further exploitation. Checkpointing could be used to explore different execution paths and automatic failure detection and repair without restarting the whole system. We aim to allow checkpointing applications that access the network and write files on disk. 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 Btrfs and ZFS. Moreover, we would like to explore the challenges raised by checkpointing distributed systems.

Record and replay and reverse debugging. Record and Replay 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. Reverse Debugging brings reproducibility to interactive debugging, at the cost of the ability to modify the execution. We aim to explore the combination of interactive runtime verification with reverse debugging, taking into account both this strength and this weakness.

Watchpoints. We implemented basic support for watchpoints in Verde. Their slowness [Hen08] may not be acceptable for certain purposes like monitoring read and write accesses to a C array in a loop. We need to find a way to monitor numerous addresses efficiently.

Usability and scalability. Our experiments involve small to medium-sized applications 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 proper programming practice and correct 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. The architecture of our distributed framework may be extended to allow loading and unloading properties dynamically. Combined with heuristics to choose relevant properties to load depending on what is observed during the execution, this mechanism may allow scaling i-RV to a vast library of properties related to correct API usage and programming pratices.

We also plan to improve usability. The specification language we used to write properties over multiple processes has been designed for monolithic applications. Further work shall be done to provide a suitable specification language to interactively verify distributed systems. We want to study the combination of decentralized monitoring [EF17] and stream runtime verification [GS18] with our approach.

Relaxing assumptions on the program. Our current assumptions on the program in our first framework are strong. We plan to improve the models by allowing multi-threaded programs with side effects, possibilities of communicating with the outside and by accounting for the physical time and communication with the outside world.

Integrating existing components. With our second approach, we designed an architecture that allows adopting existing monitors and debuggers. This allows supporting more programming languages and runtime. This also allows using proven and well-known monitors from the runtime verification community in an interactive runtime verification session, using various property models. We plan to adapt and experiment with such tools.

Decentralizing the distributed architecture. Our architecture depends on a central component: the bus. This centralization poses scaling issues. Further work is needed to address this issue and allow components to communicate without using a bus (peer-to-peer communication).

Evaluation. We provided some examples that illustrate distributed i-RV. More thorough evaluation is needed on usability of distributed i-RV as well as on its performance. A case study shall be conducted on real-life bugs with developers trying our tool. We consider using the DbgBench dataset [BSC+17] for this study.

Appendix A

Appendix

Contents

A.1 Proof: a Program and a Corresponding I-RV-Program Weakly Simulate Each Other

Let us consider:
- a program $Pgrm(\text{symT}, m_p^0, \text{start}, \text{runInstr}, \text{getAccesses})$;
- a monitor $Mon(Q, q_{\texttt{init}}, \rightarrow_{\texttt{mon}}, \text{verdict})$;
- an observing scenario Scn (Definition 3.4.1) with a set of configurations Conf_S, a set of actions A_S and a transition relation $\rightarrow_S$;
- the i-RV-program $Irv(Pgrm, Mon, Scn)$.
- the set of observable actions $Obs = \{\text{EXEC}(m, a) \mid m \in \text{Mem} \ \wedge \ a \in \text{Addr}\}$;
- the relation $\mathcal{R}$ between the configurations of $Prgm$ and Irv as defined in Definition 3.4.1.

The set of actions of Irv, as per Definition 3.3.6, when removing rules SETSYM, SETADDR, SETPC, CKPT and RESTORE is:

$$\begin{aligned} A_{\text{i}-\text{RV}} &= (A_\text{EC} \setminus \{\text{SETSYM}, \text{SETADDR}, \text{SETPC}, \text{CKPT}, \text{RESTORE}\}) \cup (A_\text{M} \setminus \{\text{CKPT}, \text{RESTORE}\}) \cup A_\text{S} \\ &= \{\text{SETBREAK}(b), \text{RMBREAK}(b), \text{SETWATCH}(\text{w}), \text{RMWATCH}(w), \\ &\quad \text{DEVBREAK}, \text{SCNBREAK}(b), \text{EVTBREAK}(e), \text{DEVWATCH}, \text{SCNWATCH}(w), \\ &\quad \text{EVTWATCH}(e), \text{INT}, \text{CONT}, \text{TRAPNOBREAK}, \text{CLEAREVENTS}, \\ &\quad \text{EXEC}(m, a) \mid w \in \text{Wp} \ \wedge \ b \in \text{Bp} \ \wedge \ e \in \text{Event} \ \wedge \ m \in \text{Mem} \ \wedge \ a \in \text{Addr}\} \\ &\quad \cup \{\text{EVENT}(e) \mid e \in \text{Event}\} \\ &\quad \cup A_\text{S} \end{aligned}$$

A.1.1 The Program Weakly Simulates the I-RV-Program

We prove Proposition 3.4.1 (Section 3.4, page 48). That is, $Pgrm$ weakly simulates Irv.

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

$$\begin{aligned} &\forall(\text{IRV}, P) \in \mathcal{R}, \forall \text{IRV}' \in \text{Conf}_{\text{i}-\text{RV}}, \forall \theta \in \overline{Obs} : \\ &\qquad \text{IRV} \xrightarrow{\theta} \text{IRV}' \implies (\text{IRV}', P) \in \mathcal{R} \end{aligned} \tag{A.1}$$

$$\begin{aligned} &\forall(\text{IRV}, P) \in \mathcal{R}, \forall \text{IRV}' \in \text{Conf}_{\text{i}-\text{RV}}, \forall \alpha \in Obs : \\ &\qquad \text{IRV} \xrightarrow{\alpha} \text{IRV}' \implies \exists P' \in \text{Conf}_\text{P} : (\text{IRV}', P') \in \mathcal{R} \wedge P \xrightarrow{\alpha} P' \end{aligned} \tag{A.2}$$

Let us consider a configuration $\text{IRV} = (P_\text{dbg}, D, M, S) \in \text{Conf}_\text{P} \times \text{Conf}_\text{D} \times \text{Conf}_\text{M} \times \text{Conf}_\text{S}$ of Irv and a configuration P of the $Prgm$ such that $(\text{IRV}, P) \in \mathcal{R}$. Let us prove these two points in turn.

A.1.1.1 Proof of (A.1)

Let us consider:
- an unobservable action $\theta \in \overline{\text{Obs}}$;
- a configuration of Irv $\text{IRV}' = (P'_\text{dbg}, D', M', S') \in \text{Conf}_\text{P} \times \text{Conf}_\text{D} \times \text{Conf}_\text{M} \times \text{Conf}_\text{S}$.

Let us assume that $(P_\text{dbg}, D, M, S) \xrightarrow{\theta} (P'_\text{dbg}, D', M', S')$.
Let us prove that $((P'_\text{dbg}, D', M', S'), P) \in \mathcal{R}$.

Action θ is either an action of A_S, or triggered by applying rule EVENT in the monitor, or rule SETWATCH, RMWATCH, GETPC, GETSYM, GETADDR, DEVBREAK, SCNBREAK, EVTBREAK, INT, CONT, SETBREAK, RMBREAK, DEVWATCH, SCNWATCH, EVTWATCH, TRAPNOBREAK, CLEAREVENTS, INSTRUMENT, STEPREDO of the execution controller.

Case: $\theta \in A_{\mathrm{S}}$.

Actions of the scenario are performed by rules of the scenario. These rules modify the state of the scenario, but not the state of the program or the debugger. As a consequence, $P = P'$ and $D = D'$. Since $((P_{\mathrm{dbg}}, D, M, S), P) \in \mathcal{R}$, we deduce $((P'_{\mathrm{dbg}}, D', M', S'), P) \in \mathcal{R}$.

Case: rule EVENT (see Figure 3.12), SETWATCH, RMWATCH, GETPC, GETSYM, GETADDR, DEVBREAK, SCNBREAK, EVTBREAK, INT, or CONT applies (see Figure 3.8, Figure 3.10, Figure 3.11).

None of these rules modify the configuration of the program nor fields *bpts* and *oi* of the debugger.

Other fields are not involved in the definition of $\mathcal{R}$.

Since $((P_{\mathrm{dbg}}, D, M, S), P) \in \mathcal{R}$, we deduce immediately $((P'_{\mathrm{dbg}}, D', M', S'), P) \in \mathcal{R}$.

Case: rule SETBREAK (Figure 3.8) applies for some $b \in D.\texttt{bpts}$.

Let us prove $((P'_{\text{dbg}}, D', M', S'), P) \in \mathcal{R}$:

Let us prove (3.1):

Let us define $m_{\text{unInstr}} = P'_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D'.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D'.\texttt{bpts}\}$, the debugger memory $P'_{\text{dbg}}.\texttt{m}$ 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 $\exists b' \in D.\texttt{bpts} : b'.\texttt{addr} = b.\texttt{addr}$, or not.

Case: $\exists b' \in D.\texttt{bpts} : b'.\texttt{addr} = b.\texttt{addr}$.

According to rule setBreak: $oi' = oi$.
Moreover, we have:

$$\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D'.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D'.\texttt{bpts}\} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D'.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D'.\texttt{bpts}\} \\
&\qquad \text{(because } P_{\text{dbg}}.\texttt{m}[b.\texttt{addr}] = \texttt{BREAK} \text{ and thus } P'_{\text{dbg}}.\texttt{m} = P_{\text{dbg}}.\texttt{m}) \\
&\qquad\quad \text{since there is already a breakpoint at } b.\texttt{addr} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts} \cup \{b\}\} \\
&\qquad \text{(because } D'.\texttt{oi} = oi' = oi = D.\texttt{oi}) \\
&= P_{\text{dbg}}.\texttt{m} \dagger \{a \mapsto D.\texttt{oi}(a) \mid a \in \{b'.\texttt{addr} \mid b' \in D.\texttt{bpts}\} \cup \{b.\texttt{addr}\}\} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts}\} \\
&\qquad \text{(because } \exists b' \in D.\texttt{bpts} : b'.\texttt{addr} = b.\texttt{addr}) \\
&= P.\texttt{m} \quad \text{(because of (3.1))}
\end{aligned}$$

Case: $\nexists b' \in D.\texttt{bpts} : b'.\texttt{addr} = b.\texttt{addr}$.

According to rule setBreak, we have:

- $oi' = oi\,[b.\texttt{addr} \mapsto P_{\text{dbg}}.\texttt{m}[b.\texttt{addr}]]$
- $P'_{\text{dbg}}.\texttt{m} = m' = m\,[b.\texttt{addr} \mapsto \texttt{BREAK}]$

We have:

$$\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D'.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D'.\texttt{bpts}\} \\
m_{\text{unInstr}} &= P_{\text{dbg}}.\texttt{m}\,[b.\texttt{addr} \mapsto \texttt{BREAK}] \\
&\qquad \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}\left[b.\texttt{addr} \mapsto P_{\text{dbg}}.\texttt{m}[b.\texttt{addr}]\right](b'.\texttt{addr}) \mid b' \in D.\texttt{bpts} \cup \{b\}\right\} \\
&= P_{\text{dbg}}.\texttt{m}\,[b.\texttt{addr} \mapsto \texttt{BREAK}]\left[b.\texttt{addr} \mapsto P_{\text{dbg}}.\texttt{m}[b.\texttt{addr}]\right] \\
&\qquad \dagger \underbrace{\left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts}\right\}}_{\left(b.\texttt{addr} \text{ is not in the domain of this function because } \nexists b' \in D.\texttt{bpts} : b'.\texttt{addr} = b.\texttt{addr}\right)} \\
&= P_{\text{dbg}}.\texttt{m}\left[b.\texttt{addr} \mapsto P_{\text{dbg}}.\texttt{m}[b.\texttt{addr}]\right] \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts}\right\} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts}\right\} \\
&= P.\texttt{m} \quad \text{(because of (3.1))}
\end{aligned}$$

In both cases, (3.1) holds. □

Since the program counter has not been modified, (3.2) holds.
Finally, $P'_{\text{dbg}}.\texttt{m} = P_{\text{dbg}}.\texttt{m}[b.\texttt{addr} \mapsto \texttt{BREAK}]$.
Therefore, (3.3) holds. □

Case: rmBreak applies (see Figure 3.8) for some input symbol rmPoint(b).

Let us suppose that $b \in D.\texttt{bpts}$.

Let us prove (3.1):

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

Case: $\exists b' \in D.\texttt{bpts} \setminus \{b\} : b'.\texttt{addr} = b.\texttt{addr}$.

We have:

$$\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D'.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D'.\texttt{bpts}\} \\
m_{\text{unInstr}} &= P_{\text{dbg}}.\texttt{m} \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts} \setminus \{b\}\right\} \\
&\qquad \text{(because } D'.\texttt{oi} = oi' = oi = D.\texttt{oi},\ P'_{\text{dbg}}.\texttt{m} = P_{\text{dbg}}.\texttt{m} \text{ and } D'.\texttt{bpts} = D.\texttt{bpts} \setminus \{b\}) \\
&= P_{\text{dbg}}.\texttt{m} \dagger \underbrace{\left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts}\right\}}_{b.\texttt{addr} \text{ is still in the domain because } \exists b' \in D.\texttt{bpts} \setminus \{b\} : b'.\texttt{addr} = b.\texttt{addr}} \\
&= P.\texttt{m} \quad \text{(because of (3.1))}
\end{aligned}$$

Case: $\nexists b' \in D.\texttt{bpts} \setminus \{b\} : b'.\texttt{addr} = b.\texttt{addr}$.

We have:

$$\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\texttt{m} \dagger \{b'.\texttt{addr} \mapsto D'.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D'.\texttt{bpts}\} \\
m_{\text{unInstr}} &= P_{\text{dbg}}.\texttt{m}\,[b.\texttt{addr} \mapsto D.\texttt{oi}(b.\texttt{addr})] \\
&\qquad \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}\,[b.\texttt{addr} \mapsto \texttt{BREAK}]\,(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts} \setminus \{b\}\right\} \\
&\qquad \text{(because in the rule, } oi' = oi\,[b.\texttt{addr} \mapsto \texttt{BREAK}] \text{ and } m' = m\,[b.\texttt{addr} \mapsto oi(b.\texttt{addr})]) \\
&= P_{\text{dbg}}.\texttt{m}\,[b.\texttt{addr} \mapsto D.\texttt{oi}(b.\texttt{addr})] \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts} \setminus \{b\}\right\} \\
&\qquad \text{(because there is no } b' \text{ in } D.\texttt{bpts} \setminus \{b\} \text{ such that } b'.\texttt{addr} = b.\texttt{addr}) \\
&= P_{\text{dbg}}.\texttt{m} \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in (D.\texttt{bpts} \setminus \{b\}) \cup \{b\}\right\} \qquad \text{(simplification)} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \left\{b'.\texttt{addr} \mapsto D.\texttt{oi}(b'.\texttt{addr}) \mid b' \in D.\texttt{bpts}\right\} \qquad \text{(because } b \in D.\texttt{bpts}) \\
&= P.\texttt{m} \quad \text{(because of (3.1))}
\end{aligned}$$

In both cases, (3.1) holds. □

Let us prove (3.2):

Direct from the definition of rmBreak (see Figure 3.8). □

Let us prove (3.3):

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

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

Therefore, (3.3) holds. □

Case: devWatch, scnWatch, or evtWatch applies (see Figure 3.4).

In any of the rules (see function restoreBP):

- $oi' = oi\,[P_{\text{dbg}}.\texttt{pc} \mapsto P_{\text{dbg}}.\texttt{m}[\texttt{P}_{\text{dbg}}.\texttt{pc}]]$
- $m' = m[P_{\text{dbg}}.\texttt{pc} \mapsto \texttt{BREAK}]$

Let us prove (3.1):

We distinguish two cases according to whether $\exists b \in D.\texttt{bpts} : b.\texttt{addr} = pc$, or not.

Case: $\exists b \in D.\texttt{bpts} : b.\texttt{addr} = pc$.

Let us define $a = \text{getAccesses}(P_{\text{dbg}})$ and $w \in D.\texttt{wpts} \setminus D.\texttt{hdld}$ such that $\exists k \in \{0, \ldots, |a|-1\} : \text{match}(a_k, w) \wedge w.\texttt{for} = \texttt{dev}$, This is possible according to the conditions of the rule. This implies $P_{\text{dbg}}.\texttt{m}[\texttt{P}_{\text{dbg}}.\texttt{pc}] \neq \texttt{BREAK}$.

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

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\texttt{m} \dagger \{b.\texttt{addr} \mapsto D'.oi(b.\texttt{addr}) \mid b \in D.\texttt{bpts}\} \\
&= P_{\text{dbg}}.\texttt{m}[P_{\text{dbg}}.\texttt{pc} \mapsto \texttt{BREAK}] \\
&\quad \dagger \underbrace{\left\{b.\texttt{addr} \mapsto D.\texttt{oi}\left[P_{\text{dbg}}.\texttt{pc} \mapsto P_{\text{dbg}}.\texttt{m}[\texttt{P}_{\text{dbg}}.\texttt{pc}]\right](b.\texttt{addr}) \mid b \in D.\texttt{bpts}\right\}}_{P_{\text{dbg}}.\texttt{pc} \text{ is in the domain of this function}} \\
&\quad \left(\text{because } oi' = oi\left[P_{\text{dbg}}.\texttt{pc} \mapsto P_{\text{dbg}}.\texttt{m}[\texttt{P}_{\text{dbg}}.\texttt{pc}]\right] \text{ and } m' = m[P_{\text{dbg}}.\texttt{pc} \mapsto \texttt{BREAK}]\right) \\
&= P_{\text{dbg}}.\texttt{m} \dagger \left\{b.\texttt{addr} \mapsto D.\texttt{oi}\left[P_{\text{dbg}}.\texttt{pc} \mapsto P_{\text{dbg}}.\texttt{m}[\texttt{P}_{\text{dbg}}.\texttt{pc}]\right](b.\texttt{addr}) \mid b \in D.\texttt{bpts}\right\} \\
&\quad \text{(simplification)} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \{b.\texttt{addr} \mapsto D.\texttt{oi}(b.\texttt{addr}) \mid b \in D.\texttt{bpts}\}\left[P_{\text{dbg}}.\texttt{pc} \mapsto P_{\text{dbg}}.\texttt{m}[P_{\text{dbg}}.\texttt{pc}]\right] \\
&\quad \text{(because } b \in D.\texttt{bpts}\text{, so the substitution can be done outside)} \\
&= P.\texttt{m}\left[P_{\text{dbg}}.\texttt{pc} \mapsto P_{\text{dbg}}.\texttt{m}[P_{\text{dbg}}.\texttt{pc}]\right] \quad \text{(because of (3.1))} \\
&= P.\texttt{m}[P.\texttt{pc} \mapsto P.\texttt{m}[P.\texttt{pc}]] \quad \text{(because } P_{\text{dbg}}.\texttt{pc} = P.\texttt{pc} \text{ and } P_{\text{dbg}}.\texttt{m}[\texttt{P}_{\text{dbg}}.\texttt{pc}] \neq \texttt{BREAK}\text{)} \\
&= P.\texttt{m}
\end{aligned}
$$

Case: $\nexists b \in D.\texttt{bpts} : b.\texttt{addr} = pc$.

In the rule, $oi' = oi$ and $m' = m$. In P'_{dbg}, $D = (P', D', M, S)$, $D'.\texttt{oi} = D.\texttt{oi}$, $D'.\texttt{bpts} = D'.\texttt{bpts}$, $P'.\texttt{m} = P.\texttt{m}$ and $P'.\texttt{pc} = P.\texttt{pc}$. Since $(P, P_{\text{dbg},D}) \in \mathcal{R}$, $(P, P'_{\text{dbg}}, D) \in \mathcal{R}$.

In both cases, (3.1) holds. □

(3.2) and (3.3) hold because the program counter and the memory are not modified by these rules.

Case: trapNoBreak apply (see Figure 3.5).

This rule applying implies that rule bpHit of the program applies. Therefore, $P_{\text{dbg}}.\texttt{m}\,[P_{\text{dbg}}.\texttt{pc}] = \texttt{BREAK}$.

Let us prove (3.1):

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

$$
\begin{aligned}
m_{\text{unInstr}} &= P'_{\text{dbg}}.\texttt{m} \dagger \{b.\texttt{addr} \mapsto D'.oi(b.\texttt{addr}) \mid b \in D.\texttt{bpts}\} \\
&= P_{\text{dbg}}.\texttt{m}\left[P_{\text{dbg}}.\texttt{pc} \mapsto D.\texttt{oi}\left(P_{\text{dbg}}.\texttt{pc}\right)\right] \dagger \underbrace{\{b.\texttt{addr} \mapsto D.\texttt{oi}(b.\texttt{addr}) \mid b \in D.\texttt{bpts}\}}_{P_{\text{dbg}}.\texttt{pc} \text{ is in the domain of this function}} \\
&\quad \text{(because in the rule, } P'_{\text{dbg}}.\texttt{m} = P_{\text{dbg}}.\texttt{m}[pc \mapsto D.\texttt{oi}(pc)]\text{)} \\
&= P_{\text{dbg}}.\texttt{m} \dagger \{b.\texttt{addr} \mapsto D.\texttt{oi}(b.\texttt{addr}) \mid b \in D.\texttt{bpts}\} \\
&\quad \text{(simplification: this substitution has no effect)} \\
&= P.\texttt{m} \quad \text{(because of (3.1))}
\end{aligned}
$$

Therefore, (3.1) holds. □

Case: clearEvents applies (see Figure 3.7).

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 (3.1), (3.2) and (3.3) is therefore straightforward by induction.

Case: instrument applies (see Figure 3.7).

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 (3.1), (3.2) and (3.3) is therefore straightforward.

Case: stepRedo applies (see Figure 3.10).

This rule temporarily sets the debugger in passive mode and uses rule evtBreak, devBreak, scnBreak, evtWatch, devWatch, scnWatch or trapNoBreak. Proving (3.1), (3.2) and (3.3) is therefore straightforward.

We proved that $((P'_{\text{dbg}}, D', M', S'), P) \in \mathcal{R}$. □

A.1.1.2 Proof of (A.2)

Let us consider:
- an observable action $\alpha \in \text{Obs}$;
- a configuration $(P'_{\text{dbg}}, D', M', S')$ of *Irv*.

Let us assume that $(P_{\text{dbg}}, D, M, S) \xrightarrow{\alpha} (P'_{\text{dbg}}, D', M', S')$.

Let us prove that $\exists P' \in \text{Conf}_{\text{P}}, ((P'_{\text{dbg}}, D', M', S'), P') \in \mathcal{R} \wedge P \xrightarrow{\alpha} P'$.

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 (3.1):

We have $(P'.\texttt{m}, P'.\texttt{pc}) = \text{runInstr}(P.\texttt{m}, P.\texttt{pc})$ (see rule normalExec).
Since $((P_{\text{dbg}}, D, M, S), P) \in \mathcal{R}$, we have $P.\texttt{m} = \text{unInstr}(P_{\text{dbg}}.\texttt{m},\ D.\texttt{bpts},\ D.\texttt{oi})$ (because of (3.1)) and $P.pc = P_{\text{dbg}}.pc$ (because of (3.2)).
We have:

$$
\begin{aligned}
(D'.\texttt{oi}, P'_{\text{dbg}}.\texttt{m}) &= \text{restoreBP}(D'.\texttt{bpts}, m_t) \\
&\qquad \text{such that } \exists pc' : (m_t, pc') = \text{runInstr}(\text{unInstr}(P_{\text{dbg}}.\texttt{m}, D.\texttt{bpts}, D.\texttt{oi}), P_{\text{dbg}}.pc) \\
&\qquad \text{(according to the rule)} \\
&= \text{restoreBP}(D'.\texttt{bpts}, m_t) \text{ such that } \exists pc' : (m_t, pc') = \text{runInstr}(P.\texttt{m}, P_{\text{dbg}}.pc) \\
&\qquad \text{(because of (3.1))} \\
&= \text{restoreBP}(D'.\texttt{bpts}, m_t) \text{ such that } \exists pc' : (m_t, pc') = \text{runInstr}(P.\texttt{m}, P.pc) \\
&\qquad \text{(because of (3.2))} \\
&= \text{restoreBP}(D'.\texttt{bpts}, m_t) \text{ such that } \exists pc' : (m_t, pc') = (P'.\texttt{m}, P'.pc) \\
&\qquad \text{(because of the program's rule normalExec)} \\
&= \text{restoreBP}(D'.\texttt{bpts}, P'.\texttt{m})
\end{aligned}
$$

Therefore:

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

The same action α is performed in the program and the i-RV-program. This proves (3.1). □

Let us prove (3.2):

$P'_{\text{dbg}}.pc$ is such that there exists m_t such that:

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

Therefore, $P'_{\text{dbg}}.pc = P'.pc$. (3.2) is proved. □

Proving (3.3) is straightforward. □

□

A.1.2 The I-RV-Program Weakly Simulates the Program

We now prove Proposition 3.4.1 (Section 3.4). That is, the i-RV-program *Irv* weakly simulates the program *Prgm*.

Proof. We prove that relation satisfies the two points of the definition of weak simulation given the set of observable actions *Obs*. Since the program does not have any unobservable action, proving that $\mathfrak{R}$ is a weak simulation amounts to proving (A.3).

$$\begin{aligned}&\forall (P, (P_{\text{dbg}}, D, M, S)) \in \mathfrak{R}, \forall \alpha \in Obs, \forall P' \in \text{Conf}_{\text{P}} :\\ &\quad P \xrightarrow{\alpha} P' \implies \exists (P'_{\text{dbg}}, D', M', S') \in \text{Conf}_{\text{P}} \times \text{Conf}_{\text{D}} \times \text{Conf}_{\text{M}} \times \text{Conf}_{\text{S}} :\\ &\qquad (P_{\text{dbg}}, D) \xrightarrow{\overline{Obs}^* \cdot \alpha \cdot \overline{Obs}^*} (P'_{\text{dbg}}, D', M', S') \wedge (P', (P'_{\text{dbg}}, D', M', S')) \in \mathfrak{R}\end{aligned} \tag{A.3}$$

Let us consider:
- two configurations P and P' of program *Prgm* in Conf_{P};
- a configuration $(P_{\text{dbg}}, D, M, S)$ of *Irv* such that $(P, (P_{\text{dbg}}, D, M, S)) \in \mathfrak{R}$;
- an action $\alpha \in Obs$.

By definition of $\mathfrak{R}$, $((P_{\text{dbg}}, D, M, S), P) \in \mathcal{R}$.
Let us suppose that $P \xrightarrow{\alpha} P'$. Since action α is observable, it can be only triggered by applying rule normalExec in program *Prgm* (see Figure 3.3). In configuration $(P_{\text{dbg}}, D, M, S)$, the debugger can be in two modes (either passive or active).
We show the existence of $(P'_{\text{dbg}}, D', M', S') \in \text{Conf}_{\text{i-RV}}$ such that $(P_{\text{dbg}}, D, M, S') \xrightarrow{\overline{Obs}^* \cdot \alpha \cdot \overline{Obs}^*} (P'_{\text{dbg}}, D', M', S')$ and $((P'_{\text{dbg}}, D', M', S'), P') \in \mathcal{R}$ in both cases.

Case: the debugger is in passive mode.

We 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.

Case: Rule normalExec applies.

In this case, the proof is similar to the one for the simulation of the i-RV-program by the program (as the same equalities apply). Resulting configuration $(P'_{\text{dbg}}, D', M', S')$ is such that $((P'_{\text{dbg}}, D', M', S'), P') \in \mathcal{R}$ and therefore $(P', (P'_{\text{dbg}}, D', M', S')) \in \mathcal{R}$.

Case: 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 $(P_{\text{dbg},i}, D_i, M_i, S_i)$ such that $(P_{\text{dbg}}, D, M, S) \xrightarrow{\theta} (P_i, D_i, M_i, S_i)$. Since θ is unobservable and $((P_{\text{dbg}}, D, M, S), P) \in \mathcal{R}$, $((P_i, D_i, M_i, S_i), P) \in \mathcal{R}$ (see proof for the simulation of the i-RV-program by the program).

Showing that successive applications of these rules starting from configuration $(P_{\text{dbg},i}, D_i, M_i, S_i)$ results in a configuration (P_j, D_j, M_j, S_j) such that $((P_j, D_j, M_j, S_j), P) \in \mathcal{R}$ is straightforward by induction on the sequence of unobservable 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 successively triggered by applying these rules.

Let us consider (P_j, D_j, M_j, S_j) the configuration such that $(P_{\text{dbg}}, D, M, S) \xrightarrow{u} (P_j, D_j, M_j, S_j)$.

Since $(P, (P_j, D_j, M_j, S_j)) \in \mathcal{R}$, configuration $(P'_{\text{dbg}}, D', M', S')$ of the i-RV-program such that $(P_j, D_j, M_j, S_j) \xrightarrow{\alpha} (P'_{\text{dbg}}, D', M', S')$ exists and rule normalExec applies (see proof for the simulation of the i-RV-program by the program).

Therefore, $((P_j, D_j, M_j, S_j), P) \in \mathfrak{R}$. 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 i-RV-program in passive mode and call rules that apply in passive mode. Therefore, proving this case is similar to the previous case.

We proved Proposition 3.4.1. □

A.2 Protocol Buffer Specification for Dist-Verde

```
syntax = "proto3";

option java_package = "verde";

message Root {
  repeated uint32 recipients           = 1;
  uint32 sender                        = 2;
  string customMessage                 = 3;
  bytes customBytes                    = 4;
  oneof content {
    Hello   hello                      = 5;
    Bye     bye                        = 6;
    EventRequest eventRequest          = 7;
    EventRelease eventRelease          = 8;
    Event event                        = 9;
    JdwpPacket jdwpPacket              = 10;
    Error error                        = 11;
    ResumeExec resumeExec              = 12;
    Verdict verdict                    = 13;
    ExpectedVerdicts expectedVerdicts  = 14;
    CheckpointRequest checkpointRequest= 15;
    Checkpoint checkpoint              = 16;
    CkptRestoreRequest
        ↪ checkpointRestoreRequest = 17;
    CkptRestored checkpointRestored    = 18;
    CkptDiscard checkpointDiscard      = 19;
    DebuggerCmd debuggerCmd            = 20;
    DebuggerCmdReply debuggerCmdReply  = 21;
    GetValues getValues                = 22;
    ValueList valueList                = 23;
    RequestOk requestOk                = 24;
    ResetRequest resetRequest          = 25;
    SuspendRequest suspendRequest      = 26;
    ResumeRequest resumeRequest        = 27;
  }
}

message Hello {
  uint32 protocolVersion     = 1;
  uint32 serviceVersion      = 2;
  string serviceName         = 3;
  string appName             = 5;
  string appVersion          = 6;
  string programName         = 7;
  string programmingLanguage = 8;
  int32  nbMonitors          = 9;
}

message Bye {}

message MonitorStateUpdate {
  uint64 sliceId = 1;
  uint64 stateId = 2;
}

message MonitorStateDeclare {
  string stateName = 1;
}

message EventRequest {
  enum EventKind {
    option allow_alias = true;
    VOID                          = 0;
    SINGLE_STEP                   = 1;
    BREAKPOINT                    = 2;
    FRAME_POP                     = 3;
    EXCEPTION                     = 4;
    USER_DEFINED                  = 5;
    THREAD_START                  = 6;
    THREAD_DEATH                  = 7;
    CLASS_PREPARE                 = 8;
    CLASS_UNLOAD                  = 9;
    CLASS_LOAD                    = 10;
    FIELD_ACCESS                  = 20;
    FIELD_MODIFICATION            = 21;
    EXCEPTION_CATCH               = 30;
    METHOD_ENTRY                  = 40;
    METHOD_EXIT                   = 41;
    METHOD_EXIT_WITH_RETURN_VALUE = 42;
    MONITOR_CONTENDED_ENTER       = 43;
    MONITOR_CONTENDED_ENTERED     = 44;
    MONITOR_WAIT                  = 45;
    MONITOR_WAITED                = 46;
    EXECUTION_START               = 90;
    EXECUTION_INIT                = 90;
    EXECUTION_DEATH               = 99;
    EXECUTION_DISCONNECTED        = 100;
  }

  enum SuspendPolicy { // Suspend:
    NONE         = 0;  //  - no threads
    EVENT_THREAD = 1;  //  - the event
        ↪ thread
    ALL          = 2;  //  - all threads
  }

  uint64 requestId                 = 1;
  EventKind kind                   = 2;

  repeated ValueRequest parameters = 3;

  string sourceName                = 4;
  string className                 = 5;
  string methodName                = 6;
  string fieldName                 = 7;
  int32  lineNumber                = 8;
  uint32 count                     = 9;
  bool   includeCaughtExceptions   = 10;
  bool   includeUncaughtExceptions = 11;
  MethodArgs methodArgs            = 12;
}

message MethodArgs {
   bool specified      = 1;
   repeated string arg = 2;
}

message RequestOk {
  uint64 requestId = 1;
}

message GetValues {
  uint64 requestId                 = 1;
  repeated ValueRequest parameters = 4;
}

message ResumeExec {
  bool suspend = 1;
}

message ValueRequest {
  enum Scope {
    AUTO           = 0;
    FRAME          = 1;
    FUNCTION       = 2;
    CLASS_INSTANCE = 3;
    CLASS_STATIC   = 4;
    FILE           = 5;
    GLOBAL         = 6;
    MACRO          = 7;
  }

  enum Type {
    NONE           = 0;
    BOOL           = 1;
    SIGNED_INT     = 2;
    UNSIGNED_INT   = 3;
    FLOAT          = 4;
    DOUBLE         = 5;
    STRING         = 6;
    BYTES          = 7;
  }

  uint32 requestId = 1;

  Type type                    = 2;
  string specificLanguageType  = 3;
  Scope scope                  = 4;

  int32 dereference              = 5;
    // -1 means "get the address of"
    //  0  means get the value
    //  1  means get the value pointed by
    //  2  means get the value at the
        ↪ address pointed by
    //  ...

  oneof spec {
    string name       = 6;
    uint32 argNumber  = 7; // 0 means :
        ↪ return value
    bool   thisObject = 8;
    string expression = 9;
  }
```

```
}

message EventRelease {
  uint64 requestId          = 1; // 0 means: all events
}

message Event {
  uint64 requestId = 1;
  repeated Value parameters = 2;
}

message ValueList {
  repeated Value value = 1;
  uint64 requestId     = 2;
}

message Value {
  oneof v {
    double     vDouble       = 2;
    float      vFloat        = 3;
    sint64     vSignedInt8   = 4;
    uint64     vUnsignedInt8 = 5;
    sint32     vSignedInt4   = 6;
    uint32     vUnsignedInt4 = 7;
    string     vString       = 8;
    bytes      vBytes        = 9;
    bool       vBool         = 10;
    Error      vError        = 11;
    ValueList  values        = 12;
  }
}

message JdwpPacket {
  bytes data = 1;
}

message Error {
  string info  = 1;
  uint32 errId = 2;
}

message Verdict {
  enum Accepting {
    NO          = 0;
    YES         = 1;
    MAYBE_NO    = 2;
    MAYBE_YES   = 3;
    UNKNOWN     = 5;
    VOID        = 6; // If this verdict should be ignored
  }

  bool      initial           = 1;
  uint32    propertyID        = 2;
  string    propertyName      = 3;
  uint32    fromEC            = 4;
  Accepting accepting         = 5;
  repeated  string state      = 6;
  repeated  Value value       = 7;
  repeated  string old_state  = 8;
  repeated  string new_state  = 9;
}

message ExpectedVerdicts {
    uint32 number = 1;
}

message CkptRequest {
  uint64 requestId = 1;
  uint64 cid       = 2;
}

message SuspendRequest {
  uint64 requestId = 1;
}

message ResumeRequest {
  uint64 requestId = 1;
}

message Ckpt {
  uint64 requestId = 1;
  uint64 cid       = 2;
}

message ResetRequest {
  uint64 requestId = 1;
}

message CkptRestoreRequest {
  uint64 requestId = 1;
  uint64 cid       = 2;
}

message CheckpointRestored {
  uint64 requestId = 1;
  uint64 cid       = 2;
}

message CheckpointDiscard {
  uint64 requestId = 1; // 0 means: all checkpoints.
  uint64 cid       = 2;
}

message DebuggerCmd {
    uint64 requestId  = 1;
    oneof c {
        string cmd    = 2;
        string print  = 3;
        string prompt = 4;
    }
}

message DebuggerCmdReply {
    uint64 requestId = 1;
    string reply     = 2;
}
```

A.3 Property for the Experiment on Zsh

In this appendix, we present a property written in Verde property format. This property is used in the experiment on Zsh in Section 6.1.1. We use the property to find the cause of a segfault in Zsh. 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 to exactly understand the meaning of these different possibilities. Rather, we aimed to find where the pointer was nulled in the code.

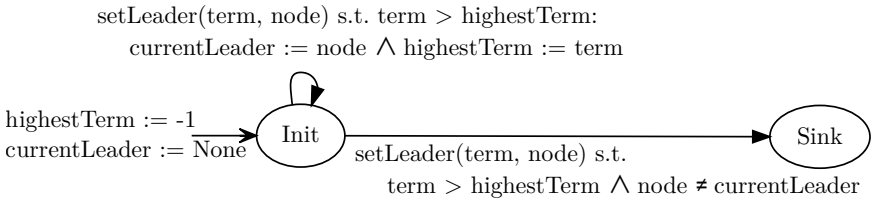
```
initialization {
  import gdb
}

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
  }

  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

    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
  }

  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 non−accepting
```

A.4 Properties for the Experiment on Raft (LogCabin)

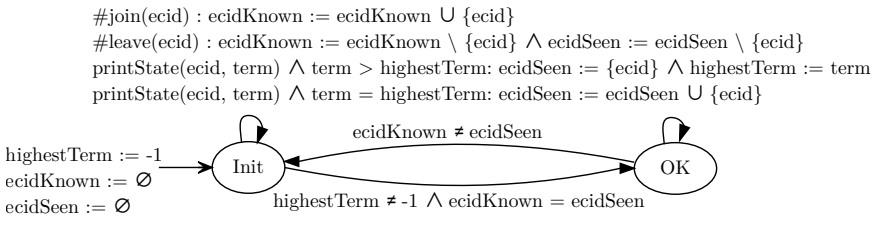
In this appendix, we present 4 properties for the LogCabin software. See the corresponding experiment in Section 6.2.3. All these properties are checked on the highest Raft term. For each property, we give a simplified graphical view, and then its code in the format of Verde.

A.4.1 Property 1: All The Nodes Agreed To Elect The Same Leader




```
1  initialization {
2      highestTerm = −1
3      currentleaderId = None
4  }
5
6  state init {
7      transition {
8          event RaftConsensus.printElectionState(#ecid as ecid, currentTerm, state, leaderId, serverId) {
9              return currentTerm >= highestTerm and leaderId != 0 and ((state == 2 and (leaderId ==
                   ↪ serverId)) or (state == 0))
10         }
11
12         success {
13             highestTerm = currentTerm
14
15             if currentleaderId is None:
16                 currentleaderId = leaderId
17
18             if currentleaderId != leaderId:
19                 return "bad"
20         } init
21     }
22 }
23
24 state bad non−accepting
```

A.4.2 Property 2: A Consensus Has Been Reached

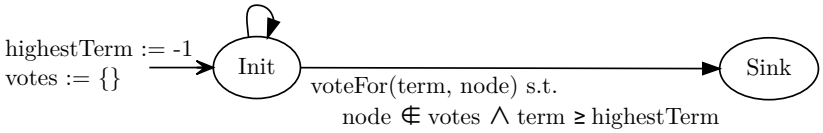



```
1  initialization {
2      ecid_known = set()
3      ecid_seen = set()
```

```
    highestTerm = −1
}

state init {
    transition {
        event #join (#ecid as ecid)
        success {
            ecid_known.add(ecid)
        } init
    }

    transition {
        event #leave (#ecid as ecid)
        success {
            ecid_known.remove(ecid)
        } init
    }

    transition {
        event RaftConsensus.printElectionState(#ecid as ecid, currentTerm, state, leaderId, serverId) {
            return currentTerm >= highestTerm and leaderId != 0
        }

        success {
            if currentTerm > highestTerm:
                highestTerm = currentTerm
                ecid_seen = set()

            ecid_seen.add(ecid)

            if ecid_seen == ecid_known:
                return "ok"
        } init
    }
}

state ok accepting import transitions from state init
```

A.4.3 Property 3: The Elected Leader Has Received a Vote

vote(term, node) s.t. term > highestTerm: votes := {node} ∧ highestTerm := term
vote(term, node) s.t. term = highestTerm: votes := votes ∪ {node}

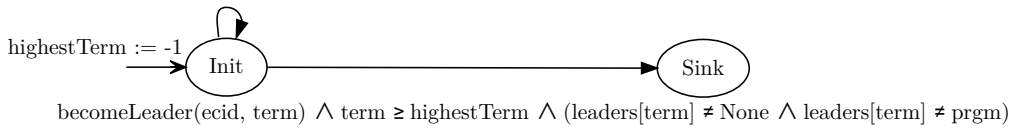



```
initialization {
    highestTerm = −1
    votes = set()
}

state init {
    transition {
        event RaftConsensus.updateLogMetadata(currentTerm, votedFor) {
            return currentTerm >= highestTerm and votedFor != 0
        }

        success {
            if currentTerm > highestTerm:
                highestTerm = currentTerm
                votes = set()

```

```
            votes.add(votedFor)
        } init
    }

    transition {
        event RaftConsensus.printElectionState(#ecid as ecid, currentTerm, state, leaderId, serverId) {
            return currentTerm >= highestTerm and leaderId != 0 and ((state == 2 and (leaderId ==
                ↪ serverId)) or (state == 0))
        }

        success {
            if currentTerm > highestTerm:
                highestTerm = currentTerm
                votes = set()

            if leaderId not in votes:
                return "bad"
        } init
    }
}

state bad non-accepting
```

A.4.4 Property 4: There is Only One Leader

startNewElection(ecid, term) ∧ term ≥ highestTerm: highestTerm := term ∧ leader[term] := None
becomeLeader(ecid, term) ∧ term ≥ highestTerm ∧ (leader[term] = None ∨ leader[term] = prgm): leader[term] := ecid
stepDown(ecid, term) ∧ term ≥ highestTerm ∧ leader[term] = ecid: leader[term] = None




```
initialization {
    knownPrgms = 0
    highestTerm = 0
    leader = {}
}

state init {
    transition {
        event #join(#ecid as prgm)
        success {
            knownPrgms += 1
        }
    }

    transition {
        event #leave(#ecid as prgm)
        success {
            knownPrgms -= 1
        }
    }

    transition {
        event RaftConsensus.startNewElection(#ecid as prgm, currentTerm : int) {
            return currentTerm >= highestTerm
        }

        success {
            highestTerm = currentTerm
```

```
        } waiting_for_leader
    }
}


# A new election started
state waiting_for_leader import transitions from state init {
    transition {
        event RaftConsensus.becomeLeader(#ecid as prgm, currentTerm : int)
        success {
            l = leader.get(currentTerm, None)
            if l and l != prgm:
                return "several_leaders"

            leader[currentTerm] = prgm
        } leader_elected
    }

    transition {
        event RaftConsensus.stepDown(#ecid as prgm, newTerm : int) {
            return newTerm >= highestTerm
        }

        success {
            highestTerm = newTerm

            try:
                if leader[newTerm] == prgm:
                    leader[newTerm] = None
        } waiting_for_leader
    }

    transition {
        event RaftConsensus.updateLogMetadata(#ecid as prgm, currentTerm : int, state : int) {
            return currentTerm >= highestTerm
        }

        success {
            highestTerm = currentTerm

            if state == 0: # FOLLOWER
                try:
                    if leader[currentTerm] == prgm:
                        leader[currentTerm] = None
                except KeyError:
                    pass

            elif state == 2: # LEADER
                l = leader.get(currentTerm, None)
                if l and l != prgm:
                    return "several_leaders"

                leader[currentTerm] = prgm

                return "leader_elected"
        }

    }

    transition {
        event RaftConsensus.startNewElection(#ecid as prgm)
        success waiting_for_leader
    }
}

state leader_elected import transitions from state waiting_for_leader
state several_leaders non-accepting
```

A.5 Promela Model for Distributed I-RV

In this appendix, we provide our model of the architecture for distributed i-RV presented in Section 4.4, written in Promela for the SPIN model checker. We use the M4 preprocessor to emulate support for some quantifiers and operators that are not supported in the LTL formulas in SPIN. We provide the file where the constants for our model are defined. We then provide the model itself. We then provide `betterltl`, our M4 library to use quantifiers and operators not natively supported by SPIN in the LTL formulas. We then provide the Makefile used to build our model. It should be noted that a recent version of Spin, that includes our fixes[1] to support large LTL formulas, should be used to check this model.

The model can be compiled using `make`. A simulation can be run using command `spin irv.pml`, and the model can be checked using command `./pan -a -N prop_all` (`all` may be replaced by the name of one of these properties: eventRecipientsAreMonitorsAndTheScenario, requestsFromEventReceiversAndToEC, verdictsFromMonitorsAndToScenarios, invariants, eventLeadsToVerdicts, expectedVerdictLeadsToResume, eventLeadsToResume, helloForEveryone, notTwoEvents or liveness).

A.5.1 `constants.m4`

```
divert(1)

// Set this to false to disable checking liveness properties
define(HANDLE_VERIF_MESSAGE, true)

// The number of monitors in the system
define(NMONITORS, 1)

// The number of execution controllers in the system
define(NECs, 1)

// The maximum number of monitor requests upon the reception of an event
define(NB_MONITOR_REQUESTS, 0)

// The maximum number of scenario actions when reacting
define(NB_SCENARIO_ACTIONS, 0)

// The size of the message queues
define(MSG_QUEUE_SIZE, 8)

// Should the number of generated events before quitting be strictly limited?
define(LIMIT_EVENTS, false)

// If so, the maximum number of generated events:
define(EVENT_MAX, 2)

// Identifiers of the different components
define(BUS, 0)
define(SCENARIO, 1)
define(FIRST_MONITOR, 2)
define(LAST_MONITOR, eval(FIRST_MONITOR + NMONITORS - 1))
define(FIRST_EC, eval(LAST_MONITOR + 1))
define(LAST_EC, eval(FIRST_EC + NECs - 1))

// Number of components
define(NB_COMPONENTS, eval(1 + NMONITORS + NECs))

define(MIN_COMPONENT_ID, SCENARIO)
define(MAX_COMPONENT_ID, LAST_EC)

divert(0)dnl
```

[1] https://github.com/nimble-code/Spin/commit/79b53c493763dcbb594e5fe59421e87655c008e6

A.5.2 irv.pml.m4

```
// We model the i-RV protocol.
//
// In this protocol, a scenario, a set of monitors and a set of execution
// controllers communicate through a bus.
// In this model, each component is a process.

// The model can be configured in file constants.m4, included here.
include(constants.m4)

// We don't use the regular C preprocessor that SPIN uses by default.
// We use m4, which lets us define recursive macros.
// M4's default starting comment string is "#". We change that to //.
changecom(//)

// Component exchange messages through the bus. These messages have the
// following types.
mtype:type = {

    // A component advertises itself on the bus. This message is broadcast to
    // any component on / joining the bus.
    Hello,

    // A component leaves the system.
    Bye,

    // This message is sent by the bus to the scenario so the scenario knows the
    // number of monitors in the system.
    NbMonitors,

    // This message is sent by the execution controller to the monitors and the
    // scenario when an event in the program happens.
    EventNotification,

    // This message is sent by the execution controller to the scenario.
    // It carries the number of monitors receiving an EventNotification for
    // the last event.
    ExpectedVerdictNumber,

    // This message is sent instead of ExpectedVerdictNumber for the final event
    // generated in the program (which is the program finishing).
    FinalVerdictNumber,

    // This message is sent by the monitor to the scenario when receiving an
    // EventNotification from the execution controller.
    // This is the number of verdicts that the scenario should receive for an
    // event.
    Verdict,

    // This message is sent by the monitors and the scenario to an execution
    // controller to receive an event.
    Request,

    // This message is a request from the scenario to an execution controller to
    // leave.
    QuitRequest,

    // This message is a reply to a request.
    Ack,

    // This message is a request from the scenario to an execution controller to
    // resume its execution (suspended by the generation of an event).
    ResumeRequest
}

// The type of identifiers of components.
define(ProcIdentifier, byte)
```

```
67
68  // This is an enumeration. Though it is not written as one because it did not
69  // work at the time of writing. See https://github.com/nimble-code/Spin/issues/6
70  define(ComponentKind, byte)
71  define(CNone, 0)
72  define(CMonitor, 1)
73  define(CScenario, 2)
74  define(CExecutionController, 3)
75
76  // Let's have proper keyword for booleans
77  define(true, 1)
78  define(false, 0)
79
80  // In the model, we will need a "random generator".
81  // When model checking, each possibility is actually tested.
82  // The macro takes the variable to assign $1, and the max $2, and
83  // sets the given variable $1 to a value in the range [0, N].
84  // The way to generate a random number is documented at
85  // http://spinroot.com/spin/Man/rand.html
86  define(random,
87      $1 = 0;
88      do
89      :: $1 < $2 -> $1++
90      :: break
91      od
92  )
93
94  // This macro checks whether the identifier $1 given in argument is the
95  // identifier of an execution controller.
96  define(isExecutionController,
97      ifelse(NECs, 1, ($1 == FIRST_EC), ($1 >= FIRST_EC && $1 <= LAST_EC))
98  )
99
100 // This macro checks whether the identifier $1 given in argument is the
101 // identifier of a monitor.
102 define(isMonitor,
103     ifelse(NMONITORS, 1, ($1 == FIRST_MONITOR), ($1 >= FIRST_MONITOR && $1 <= LAST_MONITOR))
104 )
105
106 // This macro checks whether the identifier $1 given in argument is the
107 // identifier of the scenario.
108 define(isScenario, ($1 == SCENARIO))
109
110 // This macro checks whether the identifier $1 given in argument is the
111 // identifier of the scenario or one of the monitors.
112 define(isEventReceiver, (isScenario($1) || isMonitor($1)))
113
114
115 // In an array containing a value for each execution controller, this macro
116 // gives the index of the given execution controller's identifier in this array
117 define(indexOfEC, ($1 - FIRST_EC))
118
119 // Structure of messages
120 typedef Message {
121     ProcIdentifier sender
122     ProcIdentifier recipient
123     ComponentKind type
124     byte custom // for Hello, Verdict and NbMonitors messages
125 }
126
127 // When sending an Hello message, a component sets the custom field to the kind of
128 // component (see ComponentKind)
129 define(senderKind, $1.custom)
130
131 // When sending a Verdict message, a monitor sets the custom field to identifier
132 // of the execution controller that sent the corresponding EventNotification
133 define(msgEC, $1.custom)
134
```

```
135 // When sending the NbMonitors message to the scenario,
136 // the bus sets the custom field to the number of expected monitors.
137 define(msgNumber, $1.custom)
138
139
140 // Messages are sent to components using queues, called "channels" in Promela
141 // We use an array of channels where the receiving queue of a component is
142 // indexed by its identifier.
143 chan message_queues[NB_COMPONENTS + 1] = [MSG_QUEUE_SIZE] of { Message }
144
145 // curMsg[id] is the last message retreived by component id.
146 // This is a gloal variable so it can be referred to by properties when checking
147 // the model.
148 Message curMsg[MAX_COMPONENT_ID+1];
149
150 // Getting a message is setting curMsg[id] to the head of the relevel queue.
151 inline GetMessage() {
152     message_queues[self] ? curMsg[self]
153 }
154
155 // Copies a message from b to a.
156 inline setMsg(a, b) {
157     a.sender = b.sender
158     a.recipient = b.recipient
159     a.type = b.type
160     a.custom = b.custom
161 }
162
163 // Sends a message to the given component.
164 // Only used directly by the bus.
165 inline sendMessage(to, msgtype, msgsender, msgrcpt, msgcustom) {
166     atomic {
167         msgToSend.type = msgtype
168         msgToSend.sender = msgsender
169         msgToSend.recipient = msgrcpt
170         msgToSend.custom = msgcustom
171         message_queues[to] ! msgToSend
172     }
173 }
174
175 // Sends a message of the given type to the given recipient with the given
176 // custom field (0 if irrelevant).
177 // In Promela, inline function act like macro; self is the identifier of the
178 // calling component.
179 inline send(msgtype, msgrcpt, msgcustom) {
180     sendMessage(BUS, msgtype, self, msgrcpt, msgcustom)
181 }
182
183 // Send an Hello message of the given kind.
184 inline SayHello(kind) {
185     sendMessage(BUS, Hello, self, BUS, kind)
186 }
187
188 // Send a Bye message
189 inline SayBye() {
190     sendMessage(BUS, Bye, self, BUS, 0)
191 }
192
193 // If verification of properties using temporal logic on message
194 // is enabled, we will need a boolean saying whether a new message
195 // has just been received.
196 ifelse(HANDLE_VERIF_MESSAGE, true, bool newmsg = false)
197
198
199 ////////////////////////////////////////////////////////////////////////////
200 // The bus process //
201 ////////////////////////////////////////////////////////////////////////////
202
```

```
// This function retrieves the new message in the bus.
// If verification of temporal properties is enabled, this
// macro handles the boolean newmsg.
define(awaitNewBusMessage,
    ifelse(HANDLE_VERIF_MESSAGE, true,
        newmsg = true;
        GetMessage();
        newmsg = false;,
        GetMessage()
    )
)

// Bus process: main procedure
active proctype Bus() {
    ProcIdentifier self = BUS

    // This variable stores messages to send
    Message msgToSend

    // A counter for loops
    byte i

    // This array is used for storing hello messages from every components.
    // These stored hello messages are broadcast to any joining component.
    Message knownComponents[NB_COMPONENTS]
    byte nKnownComponents = 0

    // Main loop
    do
    :: awaitNewBusMessage()
        if

        // Case: the current message is an Hello message
        :: curMsg[BUS].type == Hello ->

            if
            // If the joining component is the scenario, we send
            // the number of expected monitors
            :: senderKind(curMsg[BUS]) == CScenario ->
                sendMessage(curMsg[BUS].sender, NbMonitors, BUS, curMsg[BUS].sender, NMONITORS)

            // Otherwise, do nothing.
            // In Promela, any if statement without an "executable" case will
            // block the execution, hence this case that does nothing.
            :: senderKind(curMsg[BUS]) != CScenario ->
                skip
            fi

            // Let's forward this hello message to every known components
            // and send Hello messages from these component to the joining
            // component
            for (i : 0 .. nKnownComponents - 1) {
                message_queues[knownComponents[i].sender] ! curMsg[BUS]
                message_queues[curMsg[BUS].sender] ! (knownComponents[i])
            }

            // Save the Hello message
            setMsg(knownComponents[nKnownComponents], curMsg[BUS])
            nKnownComponents++

        // If this is a Bye message
        :: curMsg[BUS].type == Bye ->

            // Forward the bye message and remove the component
            for (i : 0 .. nKnownComponents - 1) {
                if
                :: i != curMsg[BUS].sender ->
                    message_queues[knownComponents[i].sender] ! curMsg[BUS]
```

```
271
272                 :: i == curMsg[BUS].sender ->
273                     setMsg(knownComponents[i], knownComponents[nKnownComponents - 1])
274                 fi
275             }
276
277             nKnownComponents = nKnownComponents - 1
278
279             // If all components left, stop the process.
280             if
281             :: nKnownComponents == 0 -> break
282             :: nKnownComponents != 0 -> skip
283             fi
284
285         // If the message is not Hello and not Bye, forward it to the recipient.
286         :: curMsg[BUS].type != Hello && curMsg[BUS].type != Bye ->
287             message_queues[curMsg[BUS].recipient] ! curMsg[BUS]
288         fi
289     od
290 }
291
292 ////////////////////////////////////////////////////////////////////////////
293 // The monitor process //
294 ////////////////////////////////////////////////////////////////////////////
295
296 // This function checks whether a monitor is waiting for acknowledgments
297 // for previously sent requests, and, if it does not, sends verdicts
298 inline MaybeSendVerdicts() {
299     if
300     :: pendingRequests != 0 -> skip
301     :: pendingRequests == 0 ->
302         for (i : 0 .. nPendingVerdicts - 1) {
303             if
304             :: scenarioKnown ->
305                 send(Verdict, SCENARIO, pendingVerdicts[i])
306
307             :: !scenarioKnown ->
308                 pendingScenarioMessages[nPendingScenarioMessages].type = Verdict
309                 pendingScenarioMessages[nPendingScenarioMessages].sender = self
310                 pendingScenarioMessages[nPendingScenarioMessages].recipient = SCENARIO
311                 pendingScenarioMessages[nPendingScenarioMessages].custom = pendingVerdicts[i]
312             fi
313         }
314
315         nPendingVerdicts = 0
316     fi
317
318 }
319
320
321 byte curMonId = FIRST_MONITOR
322
323 // Monitor process: main procedure
324 active [NMONITORS] proctype Monitor() {
325
326     // When a monitor is spawned, it computes its identifier.
327     // This is different from the real protocol, in which the bus assign
328     // identifiers to the components, and identifiers for all components of a
329     // kind are not contiguous.
330     ProcIdentifier self;
331     atomic {
332         self = curMonId
333         curMonId++
334     }
335
336     // This variable stores messages to send
337     Message msgToSend
338
```

```
339     // This array stores the verdicts to send to the verdicts when the monitor
340     // receives its hello message
341     Message pendingScenarioMessages[NECs]
342     byte nPendingScenarioMessages = 0
343
344     // This array stores the identifiers of the execution controllers for which
345     // verdicts are to be sent.
346     ProcIdentifier pendingVerdicts[NECs]
347     byte nPendingVerdicts = 0
348
349     // This boolean becomes true when the monitor receives the Hello message
350     // of the scenario
351     bool scenarioKnown = false
352
353     // Number of expected requests from the execution controllers
354     byte pendingRequests
355
356     // Counter for loops.
357     byte i
358
359     // Send an Hello message to the bus, with kind Monitor
360     SayHello(CMonitor)
361
362     do
363     // Get the next message
364     :: GetMessage()
365         if
366
367         // If the new message is Hello from a monitor, let's ignore it.
368         :: curMsg[self].type == Hello && senderKind(curMsg[self]) != CScenario && senderKind(
                ↪ curMsg[self]) != CExecutionController ->
369             skip
370
371
372         // If the new message is Hello from a scenario, we can send pending verdicts.
373         :: curMsg[self].type == Hello && senderKind(curMsg[self]) == CScenario ->
374
375             // In the real protocol, the identifier of the scenario is not
376             // supposed to be known in advance (no SCENARIO constant is available).
377             // We check that the protocol allows the monitor to know the
378             // identifier of the scenario before sending it any message
379             // With the following boolean and the following assert.
380             scenarioKnown = true
381             assert(curMsg[self].sender == SCENARIO)
382
383             // Let's send pending verdicts.
384             for (i : 1 .. nPendingScenarioMessages) {
385                 message_queues[BUS] ! pendingScenarioMessages[nPendingScenarioMessages - i]
386             }
387
388             i = 0
389
390         // If the message is an event notification, or an hello from an execution controller.
391         :: curMsg[self].type == EventNotification || (curMsg[self].type == Hello && senderKind(
                ↪ curMsg[self]) == CExecutionController) ->
392
393             // First, we remember that a verdict for this event must be generated.
394             pendingVerdicts[nPendingVerdicts] = curMsg[self].sender
395             nPendingVerdicts++
396
397             // Then, the monitor is supposed to update its state and request or
398             // release events to the execution controller.
399             // Let's pick a random number of event requests to send.
400             byte pendingRequestsForThisEC
401             random(pendingRequestsForThisEC, NB_MONITOR_REQUESTS)
402
403             if
404                 // If no requests are to be sent to this execution controller,
```

```
405                 // try to send verdicts to the scenario.
406                 :: pendingRequestsForThisEC == 0 ->
407
408                     MaybeSendVerdicts()
409
410                 // Otherwise, send send all the requests to the execution controller.
411                 :: pendingRequestsForThisEC != 0 ->
412
413                     for (i : 0 .. pendingRequestsForThisEC - 1) {
414                         send(Request, curMsg[self].sender, 0)
415                     }
416
417                     atomic {
418                         i = 0
419                         pendingRequests = pendingRequests + pendingRequestsForThisEC
420                         pendingRequestsForThisEC = 0
421                     }
422             fi
423
424
425         // If the message is the acknowledgment of a request
426         :: curMsg[self].type == Ack ->
427             // There is one less acknowledgment to wait.
428             pendingRequests--
429
430             // If there is no acknowledgment to wait anymore, try to send verdicts
431             MaybeSendVerdicts()
432
433
434         // If the message is the notification for a leaving component
435         :: curMsg[self].type == Bye ->
436
437             // If the leaving component is a scenario, let's leave too.
438             if
439             :: scenarioKnown && curMsg[self].sender == SCENARIO ->
440                 SayBye();
441                 break
442
443             :: !(scenarioKnown && curMsg[self].sender == SCENARIO) -> skip
444             fi
445         fi
446     od
447 }
448
449 ////////////////////////////////////////////////////////////////////////////
450 // The scenario process //
451 ////////////////////////////////////////////////////////////////////////////
452
453
454 // This function sends an action (request) to the given execution controller, if
455 // there are still actions to send.
456 // Otherwise, it sends a resume request.
457 inline sendActionOrResume(ec) {
458     if
459     :: pendingActionsByEC[indexOfEC(ec)] == 0 ->
460         send(ResumeRequest, ec, 0)
461
462     :: pendingActionsByEC[indexOfEC(ec)] != 0 ->
463         pendingActionsByEC[indexOfEC(ec)] = pendingActionsByEC[indexOfEC(ec)] - 1
464         send(Request, ec, 0)
465     fi
466 }
467
468 // This function picks a random number of actions to send, and then sends one
469 // action, or a resume request if no actions are to be sent (using
470 // sendActionOrResume)
471 inline sendActionsAndResume(ec) {
472     receivedVerdictsByEC[indexOfEC(ec)] = 0
```

```
473     expectedVerdictsByEC[indexOfEC(ec)] = -1
474
475     // nbActions is defined in the main procedure bellow.
476     random(nbActions, NB_SCENARIO_ACTIONS)
477
478     pendingActionsByEC[indexOfEC(ec)] = pendingActionsByEC[indexOfEC(ec)] + nbActions
479     sendActionOrResume(ec)
480 }
481
482 // Upon reception of a ExpectedVerdictNumber message from an execution
483 // controller, the scenario checks whether it received all the corresponding
484 // verdicts. If so, it sends a action / resume using sendActionsAndResume,
485 // or a quit request in case the ExpectedVerdictNumber actually was a
486 // FinalVerdictNumber.
487 // Otherwise, the number of expected verdicts for this execution controller is
488 // stored and will be used as verdicts are received.
489 inline handleExpectedVerdictNumber(msg) {
490     if
491
492     :: msgNumber(msg) == receivedVerdictsByEC[indexOfEC(msg.sender)] ->
493         if
494         :: finalNumberReceived[indexOfEC(msg.sender)] -> send(QuitRequest, msg.sender, 0)
495         :: !finalNumberReceived[indexOfEC(msg.sender)] -> sendActionsAndResume(msg.sender)
496         fi
497
498     :: msgNumber(msg) != receivedVerdictsByEC[indexOfEC(msg.sender)] ->
499         expectedVerdictsByEC[indexOfEC(msg.sender)] = msgNumber(msg)
500     fi
501 }
502
503 // Scenario process: main procedure
504 active proctype Scenario() {
505     ProcIdentifier self = SCENARIO
506
507     // The number of actions to be sent by execution controller.
508     // Initially, no actions are to be sent.
509     // Action will be sent to an execution controller as reactions to a verdict
510     // steming from this execution controller
511     byte pendingActionsByEC[NECs] = {0}
512
513     // For each execution controller, this array stores the number of expected
514     // verdicts steming this execution controller.
515     // This number is given by the execution controller itself in
516     // ExpectedVerdictNumber messages.
517     int expectedVerdictsByEC[NECs] = {0}
518
519     // For each execution controller, this array stores the number of received
520     // verdicts steming this execution controller.
521     // When this number is different from zero and equal to the number expected
522     // verdicts, reactions should be triggered (actions, and then a resume or a
523     // quit request are sent)
524     int receivedVerdictsByEC[NECs] = {0}
525
526     // For each execution controller, a boolean that is true if we received the
527     // final verdict number, sent by the execution controller when no more
528     // events are expected (because the program has terminated)
529     bool finalNumberReceived[NECs] = {false}
530
531
532     // Kinds for known component, by identifier
533     ComponentKind knownComponents[NB_COMPONENTS] = {CNone}
534
535     // This variable stores messages to send
536     Message msgToSend
537
538
539     // The number of known monitors, used to determine the number of verdicts
540     // that the scenario should receive when a new execution controller is seen
```

```
541     byte nbKnownMonitors = 0
542
543     // The number of known execution controller.
544     // The scenario leaves when every execution controller have left ( = when
545     // this variable is reset to zero).
546     byte nbKnownECs = 0
547
548     // Counter for loops
549     byte i
550
551     // Variable used in sendActionsAndResume to pick a random number of actions
552     // to send
553     byte nbActions
554
555     // Send an hello message to the bus
556     SayHello(CScenario)
557
558     do
559     // Get the next message
560     :: GetMessage()
561         if
562
563         // If this is an Hello message
564         :: curMsg[self].type == Hello ->
565
566             // The scenario stores the kind of the joining component.
567             // This is used to determine the kind of component that is leaving
568             // when receiving a Bye message.
569             knownComponents[curMsg[self].sender - 1] = senderKind(curMsg[self])
570
571             if
572                 // Execution controllers is counted.
573                 :: senderKind(curMsg[self]) == CExecutionController ->
574                     nbKnownECs++
575
576                 // Monitors are not: the number of monitors is given by the
577                 // special NbMonitors message sent by the bus to the scenario.
578                 :: senderKind(curMsg[self]) != CExecutionController ->
579                     skip
580             fi
581
582         // If this is a Bye message
583         :: curMsg[self].type == Bye ->
584             if
585
586             // If the component is an execution controller, keep track of the
587             // number of execution controllers. If all of them left, leave too.
588             :: knownComponents[curMsg[self].sender - 1] = CExecutionController ->
589                 nbKnownECs--
590                 if
591                 :: nbKnownECs == 0 -> SayBye() ; break
592                 :: nbKnownECs != 0 -> skip
593                 fi
594
595             // Keep track of the number of monitors too. This is only correct
596             // if no monitors leave before the NbMonitors message, which is the
597             // case because NbMonitors is the first message sent by the bus
598             // to the scenario.
599             :: knownComponents[curMsg[self].sender - 1] = CMonitor ->
600                 nbKnownMonitors--
601             fi
602
603         // If this is a verdict
604         :: curMsg[self].type == Verdict ->
605             if
606             // If the number of received verdicts (including this one) for this
607             // execution controller matches the number of expected verdicts,
608             // react
```

```
609             :: expectedVerdictsByEC[indexOfEC(msgEC(curMsg[self]))] == receivedVerdictsByEC[
                    ↪ indexOfEC(msgEC(curMsg[self]))] + 1 ->
610                 if
611
612                 // send a quit request if the program stopped
613                 :: finalNumberReceived[indexOfEC(msgEC(curMsg[self]))] ->
614                     send(QuitRequest, msgEC(curMsg[self]), 0)
615
616                 // otherwise, send a number of actions, and then a resume
617                 :: !finalNumberReceived[indexOfEC(msgEC(curMsg[self]))] ->
618                     sendActionsAndResume(msgEC(curMsg[self]))
619                 fi
620
621             // Otherwise, update the number of received verdicts
622             :: expectedVerdictsByEC[indexOfEC(msgEC(curMsg[self]))] != receivedVerdictsByEC[
                    ↪ indexOfEC(msgEC(curMsg[self]))] + 1 ->
623                 receivedVerdictsByEC[indexOfEC(msgEC(curMsg[self]))] = receivedVerdictsByEC[
                        ↪ indexOfEC(msgEC(curMsg[self]))] + 1
624             fi
625
626         // If this is an ExpectedVerdictNumber message, react to this message.
627         // See handleExpectedVerdictNumber.
628         :: curMsg[self].type == ExpectedVerdictNumber ->
629             handleExpectedVerdictNumber(curMsg[self])
630
631         // If this is an FinalVerdictNumber message, same thing but the scenario
632         // remembers that the execution controller should be sent a quit request.
633         :: curMsg[self].type == FinalVerdictNumber ->
634             finalNumberReceived[indexOfEC(curMsg[self].sender)] = true
635             handleExpectedVerdictNumber(curMsg[self])
636
637         // If this is an event notification (presumably because of a prior
638         // request):
639         :: curMsg[self].type == EventNotification ->
640             if
641                 // If no verdicts are expected for this event, let's send
642                 // actions and resume the execution.
643                 :: expectedVerdictsByEC[indexOfEC(curMsg[self].sender)] == 0 ->
644                     sendActionsAndResume(curMsg[self].sender)
645
646                 // Otherwise, actions and resume will be / have been handled
647                 // when verdicts are / have been received
648                 :: expectedVerdictsByEC[indexOfEC(curMsg[self].sender)] != 0 ->
649                     skip
650             fi
651
652         // If this is an NbMonitors messages, let's store the given number and
653         // set the number of expected verdicts for every execution controller
654         // to this number.
655         // This is cheating, since execution controllers are not known yet at
656         // this point in the real protocol. In the real protocol, this is done
657         // upon the reception of hello messages of the execution controllers.
658         // Verdicts for an execution controller can be received before their
659         // hello messages.
660         // This makes the model simpler.
661         :: curMsg[self].type == NbMonitors ->
662             nbKnownMonitors = msgNumber(curMsg[self])
663             for (i : 0 .. NECs - 1) {
664                 expectedVerdictsByEC[i] = nbKnownMonitors
665             }
666
667         // If this is an acknowledgment for a request, let's continue handling
668         // actions and resumes.
669         :: curMsg[self].type == Ack ->
670             sendActionOrResume(curMsg[self].sender)
671         fi
672     od
673 }
```

```
674
675 ////////////////////////////////////////////////////////////////////////////
676 // The execution controller process //
677 ////////////////////////////////////////////////////////////////////////////
678
679 byte curECId = FIRST_EC
680
681 // Bus process: main procedure
682 active [NECs] proctype ExecutionController() {
683     ProcIdentifier self
684
685     atomic {
686         self = curECId
687         curECId++
688     }
689
690     // This variable stores messages to send
691     Message msgToSend
692
693     // The event receivers are the monitors and the scenario.
694     // These are components that can request and receive events.
695     // This array is a list of these component, by order of arrival (reception
696     // of Hello messages by this execution controller)
697     byte eventReceivers[NMONITORS + 1]
698     byte nEventReceivers = 0
699
700     // The number of events that have been generated until now
701     byte eventsSent = 0
702
703     // A counter for loops
704     byte i = 0
705
706     // Send an Hello message to the bus
707     SayHello(CExecutionController)
708
709     do
710     // Get the next message
711     :: GetMessage()
712         if
713
714         // If this is an Hello message
715         :: curMsg[self].type == Hello ->
716             if
717
718             // If the joining component is a monitor or a scenario, its
719             // identifier is stored in the list of event receivers
720             :: senderKind(curMsg[self]) == CMonitor || senderKind(curMsg[self]) == CScenario ->
721                 eventReceivers[nEventReceivers] = curMsg[self].sender
722                 nEventReceivers++
723
724             // Otherwise, the message is ignored
725             :: !(senderKind(curMsg[self]) == CMonitor || senderKind(curMsg[self]) == CScenario) ->
726                 skip
727             fi
728
729         // If this is a leaving component
730         :: curMsg[self].type == Bye ->
731             if
732
733             // If this is an event receiver, it is removed from the list of
734             // known event receivers
735             :: senderKind(curMsg[self]) == CMonitor || senderKind(curMsg[self]) == CScenario ->
736                 do
737                 :: eventReceivers[i] == curMsg[self].sender ->
738                     eventReceivers[i] = eventReceivers[nEventReceivers - 1]
739                     nEventReceivers--
740                     break
741                 :: eventReceivers[i] != curMsg[self].sender -> i++
```

```
742                 od
743                 i = 0
744
745             // Otherwise, the message is ignored
746             :: !(senderKind(curMsg[self]) == CMonitor || senderKind(curMsg[self]) == CScenario) ->
747                 skip
748             fi
749
750         // If this is a resume request
751         :: curMsg[self].type == ResumeRequest ->
752             mtype:type expectedVerdictNumberKind
753
754             // an event is generated, and either the message is final, or not.
755             // if so, final verdict number will be sent. Otherwise, an expected
756             // verdict number will be sent.
757             // The number of generated events can be
758             // - limited (if LIMIT_EVENTS is true), in which case eventsSent
759             // counts the number of sent events (using the fact that
760             // LIMIT_EVENTS = true = 1), and EVENT_MAX is the maximum
761             // number of non-final events to generate,
762             // - or not (if LIMIT_EVENTS is false), in which case eventsSent
763             // remains equal to zero (using the fact that
764             // LIMIT_EVENTS = false = 0), to avoid changing the state of the
765             // system needlessly.
766             //
767             // I'm not sure limiting the number of events is a great idea in this
768             // model anyway, so we are probably better off keeping LIMIT_EVENTS
769             // to false.
770             if
771                 :: ((!LIMIT_EVENTS) || (eventsSent < EVENT_MAX)) -> eventsSent = eventsSent +
                        ↪ LIMIT_EVENTS; expectedVerdictNumberKind = ExpectedVerdictNumber
772                 :: expectedVerdictNumberKind = FinalVerdictNumber
773             fi
774
775             // For each event receiver, either we sent an event notification
776             // or not. The number of monitors receiving the event notification
777             // is computed and will be sent to the scenario.
778             i = 0
779             byte expectedVerdicts = 0
780             for (i : 0 .. nEventReceivers - 1) {
781                 if
782                     :: send(EventNotification, eventReceivers[i], 0)
783                        expectedVerdicts = expectedVerdicts + (eventReceivers[i] != SCENARIO)
784                     :: skip
785                 fi
786             }
787
788             // The number of expected verdicts is sent.
789             send(expectedVerdictNumberKind, SCENARIO, expectedVerdicts)
790
791         // If this is a quit request, the execution controller leaves
792         :: curMsg[self].type == QuitRequest ->
793             SayBye()
794             break
795
796         // If this is a regular request, the execution controller sends an
797         // acknowledgment
798         :: curMsg[self].type == Request ->
799             send(Ack, curMsg[self].sender, 0)
800         fi
801     od
802 }
803
804 ////////////////////////////////////////////////////////////////////////////
805 // Properties //
806 ////////////////////////////////////////////////////////////////////////////
807
808 // We include a library that provides the following macros:
```

```
809 // - forall and exists, quantifiers that are not provided by spin.
810 // - leadsTo, which is simply defined as leadsTo(A, B) = A -> <> B.
811 // - implies, different from operator implies "->" provided by spin
812 // because the implication is computed by the preprocessor and not
813 // at runtime. This allows smaller formulas.
814 // - defineLTL to define an LTL property with a name that can be reused in
815 // another property
816 // - defineAlwaysLTL, same as defineLTL, but the property is prepended with an
817 // always "[]" operator.
818 include(betterltl.m4)
819
820 /*************************
821  * INVARIANTS *
822  *************************/
823
824 // Components receiving event notifications are monitors and scenarios.
825 defineAlwaysLTL(eventRecipientsAreMonitorsAndTheScenario, (
826     (curMsg[BUS].type == EventNotification) -> isEventReceiver(curMsg[BUS].recipient)
827 ))
828
829 // Requests are sents by monitors and scenarios and received by execution controllers.
830 defineAlwaysLTL(requestsFromEventReceiversAndToEC, (
831     (curMsg[BUS].type == Request) -> (
832             isEventReceiver(curMsg[BUS].sender)
833         && isExecutionController(curMsg[BUS].recipient)
834     )
835 ))
836
837 // Verdicts are sent by monitors to scenarios
838 defineAlwaysLTL(verdictsFromMonitorsAndToScenarios, (
839     (curMsg[BUS].type == Verdict) -> (
840             isMonitor(curMsg[BUS].sender)
841         && isScenario(curMsg[BUS].recipient)
842     )
843 ))
844
845 defineLTL(conj_invariants, (
846         eventRecipientsAreMonitorsAndTheScenario
847     && verdictsFromMonitorsAndToScenarios
848     && requestsFromEventReceiversAndToEC
849 ))
850
851 defineLTL(invariants, ([] (conj_invariants)))
852
853 /*************************
854  * Liveness *
855  *************************/
856
857 ifelse(HANDLE_VERIF_MESSAGE, true,
858
859     // These properties are only defined if HANDLE_VERIF_MESSAGE is true.
860     //
861     //
862     // These properties rely on checking values of the current messages at
863     // any time during the execution.
864     //
865     // Only curMsg[BUS] should be used, except for messages that are
866     // broadcast / generated by the bus (Hello messages, NbMonitors).
867     //
868     // It is hard to think about messages from other queues correctly without
869     // doing mistakes. For instance, one could be tempted to write: "Event
870     // messages from any execution controller leads to a resume or quit
871     // request":
872     //
873     // forall mon, forall ec,
874     // curMsg[mon].type == EventNotification and
875     // curMsg[mon].sender == ec
876     // leads to
```

```
877     // curMsg[ec].type == ResumeRequest
878     //
879     // But this properties does not check what we want: the last message
880     // received by the execution controller (curMsg[ec]) could already be a
881     // resume request, making the "lead to" clause pass immediately.
882
883     // A new message happens when newmessage is true, and in the next state,
884     // newmessage becomes false
885     define(NewMessage, (newmsg && (X !newmsg)))
886
887     // When an event is sent by execution controller ec to monitor mon, a
888     // verdict from monitor mon steming from ec will be seen after.
889     defineLTL(eventLeadsToVerdicts, ([] (
890         forall(ec, FIRST_EC, LAST_EC,
891             forall(mon, FIRST_MONITOR, LAST_MONITOR,
892                 leadsTo((
893                         curMsg[BUS].type == EventNotification
894                     && curMsg[BUS].sender == ec
895                     && curMsg[BUS].recipient == mon
896                 ), (
897                         curMsg[BUS].type == Verdict
898                     && msgEC(curMsg[BUS]) == ec
899                     && curMsg[BUS].sender == mon
900                 ))
901             )
902         )
903     )))
904
905     // Expected verdict numbers are followed by a resume request
906     defineLTL(expectedVerdictLeadsToResume, ([] (
907         forall(ec, FIRST_EC, LAST_EC,
908             leadsTo((
909                     curMsg[BUS].type == ExpectedVerdictNumber
910                 && curMsg[BUS].sender == ec
911                 && curMsg[BUS].recipient == SCENARIO
912             ), (
913                     curMsg[BUS].type == ResumeRequest
914                 && curMsg[BUS].recipient == ec
915             ))
916         )
917     )))
918
919     // Events from ec are followed by a resume or a quit request to ec.
920     defineLTL(eventLeadsToResume, ([] (
921         forall(ec, FIRST_EC, LAST_EC,
922             leadsTo((
923                     curMsg[BUS].type == EventNotification
924                 && curMsg[BUS].sender == ec
925             ), (
926                 ((curMsg[BUS].type == ResumeRequest) || (curMsg[BUS].type == QuitRequest))
927                 && curMsg[BUS].recipient == ec
928             ))
929         )
930     )))
931
932     // Hello from any component is seen by any component
933     defineLTL(helloForEveryone,
934       forall(rcpt, MIN_COMPONENT_ID, MAX_COMPONENT_ID,
935         forall(sdr, MIN_COMPONENT_ID, MAX_COMPONENT_ID,
936           implies(sdr != rcpt, (<> (curMsg[rcpt].type == Hello && curMsg[rcpt].sender == sdr)))
937         )
938       )
939     )
940
941     // An event notification from ec to mon cannot be seen after an event
942     // notification from ec to mon until a verdict has been issued by mon
943     // steming from ec.
944     defineLTL(notTwoEvents, (
```

```
        forall(ec, FIRST_EC, LAST_EC,
            forall(mon, FIRST_MONITOR, LAST_MONITOR,
                [] (
                    (
                        NewMessage && (
                                curMsg[BUS].type == EventNotification
                            && curMsg[BUS].sender == ec
                            && curMsg[BUS].recipient == mon
                        )
                    ) -> X ((
                        NewMessage -> !(
                                curMsg[BUS].type == EventNotification
                            && curMsg[BUS].sender == ec
                            && curMsg[BUS].recipient == mon
                        )
                    ) U (curMsg[BUS].type == Verdict && msgEC(curMsg[BUS]) == ec))
                )
            )
        )
    ))

    defineLTL(liveness, (
            eventLeadsToVerdicts
        && expectedVerdictLeadsToResume
        && notTwoEvents
        && eventLeadsToResume
        && helloForEveryone
    ))


    defineLTL(all, (
            invariants
        && liveness
    ))
)
```

A.5.3 betterltl.m4

```
divert(‘1’)

ifelse(‘
 This m4 library provides the following macros:
 - forall and exists, quantifiers that are not provided by spin.
 - leadsTo, which is simply defined as leadsTo(A, B) = A -> <> B.
 - implies, different from operator implies "->" provided by spin
   because the implication is computed by the preprocessor and not
   at runtime. This allows smaller formulas.
 - defineLTL to define an LTL property with a name that can be reused in
   another property
 - defineAlwaysLTL, same as defineLTL, but the property is prepended with an
   always "[]" operator.

Thanks https://www.gnu.org/software/m4/manual/m4.html#Forloop for the for loop’)

define(‘forall’, ‘pushdef(‘$1’, ‘$2’)_forall($@)popdef(‘$1’)’)
define(‘_forall’, ($4)‘ifelse($1, ‘$3’, ‘’, ‘define(‘$1’, incr($1)) && $0($@)’)’)

define(‘exists’, ‘pushdef(‘$1’, ‘$2’)_exists($@)popdef(‘$1’)’)
define(‘_exists’, ($4)‘ifelse($1, ‘$3’, ‘’, ‘define(‘$1’, incr($1)) || $0($@)’)’)

define(‘leadsTo’, (($1) -> <> ($2)))

define(‘apply_noteq_implies’, ‘define(‘implies’, ‘ifelse’(‘eval’(‘$’1), 1, ‘$’2,‘’))dnl
patsubst(patsubst(patsubst(patsubst($1, ‘(([
 ]+)) &&’), ‘&& (([
 ]+))’), ‘([
```

```
29  ]+) &&'), '&& ([
30  ]+)')dnl
31  undefine('implies')')
32
33  define('defineLTL', 'define('$1', 'apply_noteq_implies($2)') ltl prop_$1 {
34      $1
35  }')
36
37  define('defineAlwaysLTL',
38  'define('$1', '$2')
39  ltl prop_$1 { [] ($1) }')
40
41  divert('0')dnl
```

A.5.4 Makefile

```
1  M4?=$(shell which m4)
2  SPIN?=$(shell which spin)
3
4  pan: pan.c
5          cc $< -o $@
6
7  pan.c: irv.pml
8          $(SPIN) -a $<
9
10 irv.pml: irv.m4.pml constants.m4 betterltl.m4
11         $(M4) < $< > $@
12
13 .PHONY: clean
14
15 clean:
16         rm -rf $$(cat .gitignore)
```

Bibliography

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[ASP] AspectJ website. https://www.eclipse.org/aspectj/.

[AZ19] Abdulaziz Almalaq and Jun Jason Zhang. Evolutionary deep learning-based energy consumption prediction for buildings. *IEEE Access*, 7:1520–1531, 2019.

[BF18a] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[BF18b] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[BFR+16] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A. Rosenblueth, and Corentin Travers. Decentralized asynchronous crash-resilient runtime verification. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 16:1–16:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[BLB+10] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, and Martin Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 231–240. IEEE Computer Society, 2010.

[BM02] Mark Brörkens and Michael Möller. Dynamic event generation for runtime checking using the JDI. *Electr. Notes Theor. Comput. Sci.*, 70(4):21–35, 2002.

[BS14] Borzoo Bonakdarpour and Scott A. Smolka, editors. *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*. Springer, 2014.

[BSC+17] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. How developers debug software the dbgbench dataset: poster. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 244–246. IEEE Computer Society, 2017.

[BZ11] Derek Bruening and Qin Zhao. 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*, pages 213–223. IEEE Computer Society, 2011.

[BZA12] Derek Bruening, Qin Zhao, and Saman P. Amarasinghe. Transparent dynamic instrumentation. In Steven Hand and Dilma Da Silva, editors, *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)*, pages 133–144. ACM, 2012.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[CD08] Jon Crowcroft and Michael Dahlin, editors. *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 2008.

[CFG11] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A monitoring tool for erlang. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer, 2011.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[CL02] Yoonsik Cheon and Gary T. Leavens. 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*, pages 231–255, 2002.

[CL18] Christian Colombo and Martin Leucker, editors. *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*. Springer, 2018.

[CMP15] Martial Chabot, Kévin Mazet, and Laurence Pierre. Automatic and configurable instrumentation of C programs with temporal assertion checkers. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pages 208–217. IEEE, 2015.

[CR07] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *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*, pages 569–588. ACM, 2007.

[CR09] Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice*

of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, pages 246–261, 2009.

[CRIa] CRIU website. `https://criu.org/`.

[CRIb] Comparison of checkpoint and restore solutions. `https://criu.org/Comparison_to_other_CR_projects`.

[DD08] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information & Software Technology*, 50(9-10):833–859, 2008.

[ddt] Pdp-1 program library - DEC Debugging Tape. `https://www.computerhistory.org/pdp-1/_media/pdf/DEC.pdp_1.1964.102650078.pdf`. [accessed 21-May-2019].

[DJ01] Mireille Ducassé and Erwan Jahier. Efficient automated trace analysis: Examples with morphine. *Electr. Notes Theor. Comput. Sci.*, 55(2):118–133, 2001.

[dS92] Fabio Q. B. da Silva. *Correctness proofs of compilers and debuggers : an approach based on structural operational semantics*. PhD thesis, University of Edinburgh, UK, 1992.

[Duc99] Mireille Ducassé. Opium: an extendable trace analyzer for prolog. *The Journal of Logic Programming*, 39(1):177 – 223, 1999.

[DWA] The DWARF debugging standard. `http://dwarfstd.org/`.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[EF17] Antoine El-Hokayem and Yliès Falcone. Monitoring decentralized specifications. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 125–135. ACM, 2017.

[EF18] Antoine El-Hokayem and Yliès Falcone. Can we monitor all multithreaded programs? In Colombo and Leucker [CL18], pages 64–89.

[EMS18] Ekkehard Ernst, Rossana Merola, and Daniel Samaan. The economics of artificial intelligence: Implications for the future of work, 10 2018.

[Eng12] Jakob Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–6. IEEE, 2012.

[eni] First generation electronic computers. `https://www.phy.ornl.gov/csep/ov/node10.html`. [accessed 21-May-2019].

[Fal10] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[FMT18] Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto. Reversible choreographies via monitoring in erlang. In Silvia Bonomi and Etienne Rivière, editors, *Distributed Applications and Interoperable Systems - 18th IFIP WG 6.1 International Conference, DAIS 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, volume 10853 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2018.

[GM14] Kiril Georgiev and Vania Marangozova-Martin. Mpsoc zoom debugging: A deterministic record-partial replay approach. In *12th IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2014, Milano, Italy, August 26-28, 2014*, pages 73–80, 2014.

[GS18] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In Colombo and Leucker [CL18], pages 282–298.

[Hav00] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2000.

[Hen08] Laurie J. Hendren, editor. *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4959 of *Lecture Notes in Computer Science*. Springer, 2008.

[HG05] Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *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*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.

[HKKT18] Timo Hynninen, Jussi Kasurinen, Antti Knutas, and Ossi Taipale. Software testing: Survey of the industry practices. In *MIPRO*, pages 1449–1454. IEEE, 2018.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[HR02] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[IML07] Juha Itkonen, Mika Mäntylä, and Casper Lassenius. Defect detection efficiency: Test case based vs. exploratory testing. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, September 20-21, 2007, Madrid, Spain*, pages 61–70. ACM / IEEE Computer Society, 2007.

[IML09] Juha Itkonen, Mika Mäntylä, and Casper Lassenius. How do testers do it? an exploratory study on manual testing practices. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA*, pages 494–497. IEEE Computer Society, 2009.

[JDB] JDB website. `https://docs.oracle.com/en/java/javase/11/tools/jdb.html`.

[JFMP17] Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, and Kevin Pouget. 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*, pages 182–193. IEEE Computer Society, 2017.

[JLSU87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian W. Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987.

[JMLR12] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. Javamop: Efficient parametric runtime monitoring framework. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1427–1430. IEEE Computer Society, 2012.

[Jos69] William H. Josephs. An on-line machine language debugger for OS/360. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '69 Fall Joint Computer Conference, November 18-20, 1969, Las Vegas, Navada, USA*, volume 35 of *AFIPS Conference Proceedings*, pages 179–186. AFIPS / ACM, 1969.

[JRLD19] José Miguel Jiménez, Oscar Romero, Jaime Lloret, and Juan R. Diaz. Energy savings consumption on public wireless networks by SDN management. *MONET*, 24(2):667–677, 2019.

[JSNQ17] Zhengrui Jiang, Kevin P. Scheibe, Sree Nilakanta, and Xinxue (Shawn) Qu. The economics of public beta testing. *Decision Sciences*, 48(1):150–175, 2017.

[Jyt] Jython website. `https://jython.org/`.

[JZ19] Liangliang Jin and Chaoyong Zhang. Process planning optimization with energy consumption reduction from a novel perspective: Mathematical modeling and a dynamic programming-like heuristic algorithm. *IEEE Access*, 7:7381–7396, 2019.

[Kat06] Shmuel Katz. Transactions on aspect-oriented software development I. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, chapter Aspect Categories and Classes of Temporal Properties, pages 106–134. Springer-Verlag, Berlin, Heidelberg, 2006.

[KCD+97] Péter Kacsuk, José C. Cunha, Gábor Dózsa, João Lourenço, Tibor Fadgyas, and Tiago R. Antão. A graphical development and debugging environment for parallel programs. *Parallel Computing*, 22(13):1747–1770, 1997.

[Kic02] Gregor Kiczales. AspectJ(tm): Aspect-oriented programming in Java. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lan92] William Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[LCK+97] João Lourenço, José C. Cunha, Henryk Krawczyk, Piotr Kuzora, Marcin Neyman, and Bogdan Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In *23rd EUROMICRO Conference '97, New Frontiers of Information Technology, 1-4 September 1997, Budapest, Hungary*, page 291. IEEE Computer Society, 1997.

[LCM+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.

[LGdS+11] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Ahn, Martin Schulz, and Barry Rountree. Large scale debugging of parallel tasks with automaded. In Scott Lathrop, Jim Costa, and William Kramer, editors, *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 50:1–50:10. ACM, 2011.

[LGW+08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: debugging deployed distributed systems. In Crowcroft and Dahlin [CD08], pages 423–437.

[LKK+99] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA*, pages 279–287. CSREA Press, 1999.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[LSZE11] Kyu Hyung Lee, Nick Sumner, Xiangyu Zhang, and Patrick Eugster. Unified debugging of distributed systems with recon. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 85–96. IEEE Compute Society, 2011.

[LW12] David Lecomber and Patrick Wohlschlegel. Debugging at scale with allinea DDT. In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2012 - Contributed Papers Presented at the 6th International Parallel Tools Workshop, Stuttgart, Germany, September 25-26, 2012*, pages 3–12. Springer, 2012.

[MB] J. F. Maranzano and S. R. Bourne. A tutorial introduction to ADB. `http://wolfram.schneider.org/bsd/7thEdManVol2/adb/adb.pdf`. [accessed 21-May-2019].

[MB15] Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 494–503. IEEE Computer Society, 2015.

[Mil89] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[MM83] Ben C. Moszkowski and Zohar Manna. Reasoning in interval temporal logic. In Edmund M. Clarke and Dexter Kozen, editors, *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings*, volume 164 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 1983.

[MMM06] Alexander V. Mirgorodskiy, Naoya Maruyama, and Barton P. Miller. Scalable systems software - problem diagnosis in large-scale computing environments. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, page 88. ACM Press, 2006.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.

[MSS02] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program instrumentation for debugging and monitoring with aspectc++. In *5th International Symposiun on Object Oriented Real-Time Distributed Computing, ISORC 2002, Washington, DC, USA, April 29 - May 1, 2002*, pages 249–256. IEEE Computer Society, 2002.

[MVT+16] Reed Milewicz, Rajeshwar Vanka, James Tuck, Daniel Quinlan, and Peter Pirkelbauer. Lightweight runtime checking of C programs with RTC. *Computer Languages, Systems & Structures*, 45:191–203, 2016.

[NJW+13] Samaneh Navabpour, Yogi Joshi, Chun Wah Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for C programs. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *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*, pages 603–606. ACM, 2013.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[OO14] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319. USENIX Association, 2014.

[PNP+19] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information & Software Technology*, 105:43–55, 2019.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[Pod18] Andreas Podelski, editor. *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*. Springer, 2018.

[Pou14] Kevin Pouget. *Programming-Model Centric Debugging for multicore embedded systems / Debogage Interactif des systemes embarques multicoeur base sur le model de programmation.* PhD thesis, University of Grenoble, France, 2014.

[Pro] Protocul Buffers website. `https://developers.google.com/protocol-buffers/`.

[PSK+16a] Fábio Petrillo, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla M. D. S. Freitas, and Yann-Gaël Guéhéneuc. Towards understanding interactive debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*, pages 152–163. IEEE, 2016.

[PSK+16b] Fábio Petrillo, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla M. D. S. Freitas, and Yann-Gaël Guéhéneuc. Towards understanding interactive debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1-3, 2016*, pages 152–163. IEEE, 2016.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[Ram94] Norman Ramsey. Correctness of trap-based breakpoint implementations. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 15–24. ACM Press, 1994.

[RCM93] Jean-François Roos, Luc Courtrai, and Jean-François Méhaut. Execution replay of parallel programs. In *1993 Euromicro Workshop on Parallel and Distributed Processing, PDP 1993, Gran Canaria, Spain, 27-29 January 1993*, pages 429–434, 1993.

[Rhi] Rhino website. `https://developer.mozilla.org/docs/Mozilla/Projects/Rhino`.

[RM11] Barath Raghavan and Justin Ma. The energy and emergy of the internet. In Hari Balakrishnan, Dina Katabi, Aditya Akella, and Ion Stoica, editors, *Tenth ACM Workshop on Hot Topics in Networks (HotNets-X), HOTNETS '11, Cambridge, MA, USA - November 14 - 15, 2011*, page 9. ACM, 2011.

[ROM+18] Patrick Reipschläger, Burcu Kulahcioglu Ozkan, Aman Shankar Mathur, Stefan Gumhold, Rupak Majumdar, and Raimund Dachselt. Debugar: Mixed dimensional displays for immersive debugging of distributed systems. In Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox, editors, *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*. ACM, 2018.

[RTKvE18] Lambèr M. M. Royakkers, Jelte Timmer, Linda Kool, and Rinie van Est. Societal and ethical issues of digitization. *Ethics Inf. Technol.*, 20(2):127–142, 2018.

[SBS+11] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2011.

[SDB94] Elissa D. Smilowitz, Michael J. Darnell, and Alan E. Benson. Are we overlooking some usability testing methods? A comparison of lab, beta, and forum tests. *Behaviour & IT*, 13(1-2):183–190, 1994.

[SHL12] Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *International Journal on Software Tools for Technology Transfer*, 14(3):243–247, 2012.

[SKN19] Gurmeet Singh, T. Ch. Anil Kumar, and V. N. A. Naikan. Efficiency monitoring as a strategy for cost effective maintenance of induction motors for minimizing carbon emission and energy consumption. *Rel. Eng. & Sys. Safety*, 184:193–201, 2019.

[Str] Stripe, Inc. The developer coefficient. `https://stripe.com/files/reports/the-developer-coefficient.pdf`.

[TJFY10] Xuping Tu, Hai Jin, Xuepeng Fan, and Jiang Ye. Meld: A real-time message logic debugging system for distributed systems. In *5th IEEE Asia-Pacific Services Computing Conference, APSCC 2010, 6-10 December 2010, Hangzhou, China, Proceedings*, pages 59–66. IEEE Computer Society, 2010.

[TLF] Le trésor de la langue française informatisé. `http://atilf.atilf.fr/tlf.htm`.

[wora] Wordnet definition of computer. `http://wordnetweb.princeton.edu/perl/webwn?s=computer`. [accessed 20-May-2019].

[worb] Wordnet definition of computer science. `http://wordnetweb.princeton.edu/perl/webwn?s=computer+science`. [accessed 20-May-2019].

[ZKB11] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Vrisha: using scaling properties of parallel programs for bug detection and localization. In Arthur B. Maccabe and Douglas Thain, editors, *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pages 85–96. ACM, 2011.