Experiments on automation of formal verification of devices at the binary level

THOMAS LACROIX

Master in Computer Science
Date: June 5, 2019
Host supervisor: Mads Dam
School supervisor: Pierre-Édouard Portier
Computer Science department - INSA Lyon
Host organization: Division of Theoretical Computer Science - KTH

Abstract

With the advent of virtualization, more and more work is put into the verification of hypervisors. Being low-level softwares, such verification should preferably be performed at binary level. Binary analysis platforms are being developed to help perform these proofs, but a lot of the work has to be carried out manually.

In this thesis, we focus on the formal verification of a Network Interface Controller (NIC), more specifically we look at how to automate and reduce the boilerplate work from an existing verification. We base our work on the HolBA platform, its hardware-independent intermediate representation language BIR and supporting tools, and we experiment on how to perform this proof by leveraging existing tools.

We first replaced the existing NIC model written in HOL4 to an equivalent one written using BIR, enabling the use of HolBA tools. Secondly, we developed some visualization tools to help navigate and gain some insight into the existing proof and its structure. Thirdly, we experimented the use of Hoare Triples in conjunction with an SMT solver to perform contract verification. Finally, we proved a simple contract written in terms of the formal NIC model on the BIR implementation of this model, unlocking the way of performing more complex proofs using the HolBA platform.

Keywords: binary analysis, formal verification, proof producing analysis, theorem proving

Abstract

Avec la démocratisation de la virtualisation, de plus en plus d'efforts sont consacrés à la vérification des hyperviseurs. S'agissant de logiciels de bas niveau, une telle vérification devrait de préférence être effectuée au niveau binaire. Des plates-formes d'analyse binaire sont en cours de développement pour aider à réaliser ces preuves, mais une grande partie du travail doit encore être effectuée manuellement.

Dans cette thèse, nous nous concentrons sur la vérification formelle d'un Contrôleur d'Interface Réseau (NIC), plus spécifiquement sur la manière d'automatiser et de réduire le travail répétitif d'une preuve existante. Nous nous basons sur la plate-forme HolBA, son langage de représentation intermédiaire indépendant du matériel, BIR et ses outils de support, et nous nous intéressons à la manière de réaliser cette preuve en utilisant des outils existants.

Nous avons d'abord remplacé le modèle NIC existant écrit en HOL4 par un modèle équivalent écrit en BIR, permettant ainsi l'utilisation des outils de HolBA. Deuxièmement, nous avons développé des outils de visualisation pour nous aider à naviguer et à mieux comprendre la preuve existante et sa structure. Troisièmement, nous avons expérimenté l'utilisation des triplets de Hoare en conjonction avec un solveur SMT pour effectuer une vérification par contrat. Enfin, nous avons prouvé un contrat simple écrit en termes du modèle formel du NIC sur l'implémentation de ce modèle en BIR, ouvrant la voie à la réalisation de preuves plus complexes avec la plate-forme HolBA.

Mot-clés : binary analysis, formal verification, proof producing analysis, theorem proving

Contents

Chapter 1

Introduction

This chapter serves as an introduction to the degree project and presents the background of the work along with this thesis objective. Delimitations to the project and the choice of the methodology are also discussed.

1.1 Background

Embedded systems are becoming more and more common with the current advent of **IoT** and mobile computing platforms, such as smartphones. Those systems are fully-fledged computers with powerful hardware, complete operating systems, and access to the Internet. Such systems can run security-critical services, such as a building security system or automatic toll gates, or carry valuable information as it is the case for personal smartphones. Therefore, these two characteristics make them targets of choice for attackers.

The **Provably Secure Execution Platforms for Embedded Systems (PROSPER)** project [**noauthor_prosper:_nodate**] aims to develop a secure and formally verified hypervisor for embedded systems. Hypervisors are thin layers running directly on top of hardware providing the ability to run virtualized applications, such that operating systems or real-time control systems. Those virtualized applications then don't have privileged access to the hardware and have to go through the hypervisor. This allows different applications to share the same hardware while providing strong isolation between them, thus ensuring confidentiality and security. Moreover, security not only means protection from external attacks but also resilience to bugs. If multiple critical systems are running on the same hardware, bugs or crashes in some systems shouldn't affect the others from behaving correctly. Figure 1.1 shows a system running two isolated Linux on top of a hypervisor.

Previous work in the **PROSPER** project achieved to formally verify a simple separation kernel [**noauthor_prosper:_nodate-1**, **dam_formal_2013**], which later re-

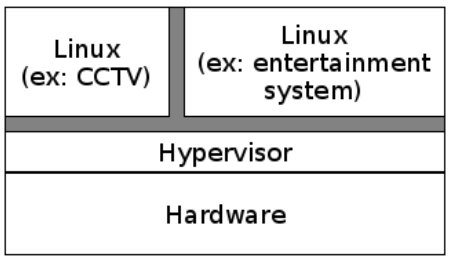

Figure 1.1: Two Linux on top of an hypervisor. They run isolated from each other and from the hypervisor.

sulted into an implementation of a working hypervisor. Then, they achieved to run both Linux and **FreeRTOS** on top of it. Finally, they formally verified memory isolation for virtualized applications [**nemati_trustworthy_2015**]. Nowadays, among other projects, the PROSPER team is working on device virtualization, giving access to hardware devices to virtualized applications. **Network Interface Controller (NIC)** devices are an interesting example, which enable network communication and give the ability to communicate through the Internet.

A formal model of a **NIC** device has already been produced, on which some security theorems have been proved [**haglund_formal_2016**]. These theorems can be seen as high-level proofs relying on a layer of lower-level lemmas. This layer provides an abstraction over the raw formal model. This is illustrated in the left-hand side of Figure 1.2.

Maybe you want to explain at this point what are the security issues related to a NIC

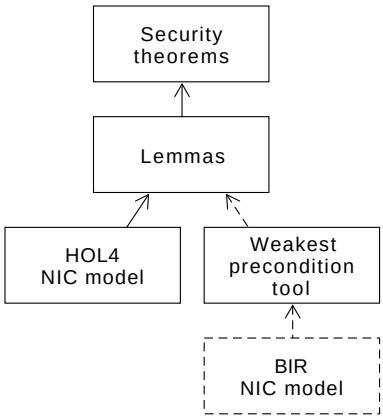

Figure 1.2: HOL4 v. BIR NIC models. The left hand side already exists. This project consists in the dashed elements. The dotted lines represent the work done during this project.

The team is now developing a new framework for performing binary analysis in HOL4, an interactive theorem prover, named **HOL4 Binary Analysis Platform (HolBA)** [**noauthor_holba_2019**]. This framework is based on two papers written in the team. The first one introduces sound **transpilation** from binary to machine-independent code [1][**metere_sound_2017**]. The second paper, "TrABin: Trustworthy Analyses of Binaries" [**lindner_trabin:_2019**], lays the foundations of the **HolBA** platform: it formally models **BIR**, introduces various supporting tools, implements two **proof-producing transpilers** (ARMv8 and Cortex-M0), named lifters, and a proof-producing weakest precondition generator for loop-free programs.

While this kind of **transpilers** and **proof-producing** weakest precondition tools already exist [2], the novelty in this work is that the transpiler is proof-producing, i.e. it produces a formal proof that both binary representations are equivalent, under the simulation theory, with respect to the **Instruction Set Architecture (ISA)** model [3]. With this method, you no longer need to trust the transpiler. Figure 1.3 gives an overview of the HolBA framework.

Is it clear what is a transpiler?

Not proof-producing weakest precondition

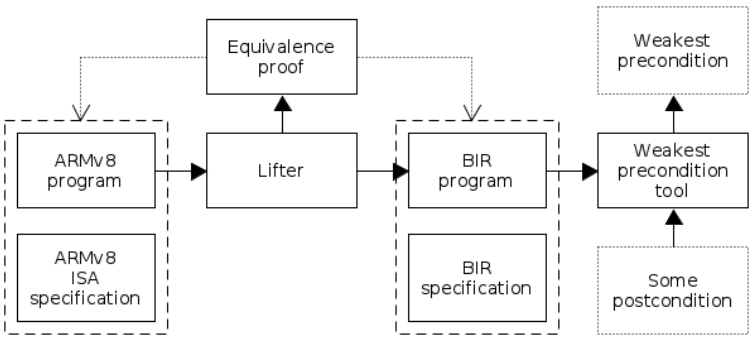

Figure 1.3: The HolBA framework. The lifter generates a BIR program and an equivalence proof from an ARMv8 program. The equivalence proof establishes a simulation property between the ARMv8 binary program and the generated BIR binary program, showing that they have the same behavior with respect to the ARMv8 **ISA** specification and the BIR specification. HolBA also support the Cortex-M0 ISA.

The idea of this work is to translate the formal **NIC** model of [**haglund_formal_2016**] using **BIR**, then use HolBA's proof-producing weakest precondition tool to prove the same lower-level lemmas than the formal model. With all the lemmas proved, the

[1]The machine-independent language used in the work is an implementation of **Carnegie Mellon University's Binary Analysis Platform (CMU BAP)**'s BIL [**noauthor_binary_2019**]. This implementation will evolve later in [**lindner_trabin:_2019**] into BIR that HolBA uses today.

[2]See related discussion in [**lindner_trabin:_2019**].

[3]Several models of different ARM **ISA** have been realized in multiple research institutions, such as ARMv3, ARMv4, ARMv7 [**noauthor_canonical_2019**, **hutchison_trustworthy_2010**] and very recently ARMv8 [**armstrong_isa_2019**].

security properties are implied. Figure 1.2 gives an overview of this idea: using the proof-producing weakest precondition tool to bind together a newly written BIR NIC model and the work done on the formal model.

1.2 Intended readers

In this thesis, formal verification is the central topic. The thesis explores how to model a hardware device using a binary analysis platform and describes some formal verification techniques. A reader interested in this topic may find the results useful for further work. A casual reader will be presented with a light introduction to the concepts that are essential in order to understand the problem that this thesis aims to explore. Those concepts should also be useful in order to understand software verification in general. For a more coherent reading, the concepts are introduced throughout the document. However, the reader is expected to have a background in Computer Science in general, and knowledge in formal verification will make the thesis easier to digest.

1.3 Thesis objective

The primary goal of this thesis project is to explore verification techniques in order to automate parts, if not all, of the verification process of hardware devices using the HolBA platform. The formal **NIC** model of [**haglund_formal_2016**] is used as working example.

The ultimate goal would be to obtain a fully automatic pipeline for performing such verifications. However, it is evident that this goal isn't reachable in such a small amount of time, or even at all. Thus, this thesis focuses instead on exploring what toolkit is needed in order to facilitate this work.

1.4 Delimitations

This work being about exploration of techniques towards automation of verification techniques, instead of being about producing an actual complete proof of a hardware device, some implementations have not been completed in order to save time to explore in more areas. Additionally, this work mainly concerns the HolBA platform that is developed in the team where this thesis took place.

1.5 Choice of methodology

This work has been carried out step-by-step toward an ideal goal, i.e. re-establishing all the security properties. On the road, needs have been identified and tools have been implemented in order to tackle them. This approach made sense in this particular work because the needs were not known in advance, and therefore needed to be identified. This thesis presents the steps taken during this work, the motivations of each tool that have been implemented, and discusses their limitations and future work in the conclusion.

Chapter 2

Related work

*This chapter will present the related work in the domain of secure execution platforms and binary analysis. After briefly introducing memory sharing between **CPU** and devices, it will explain the need for secure execution platforms, present existing binary analysis platforms and explain the novelty of **HolBA**, the platform used in this work.*

2.1 Memory sharing between CPU and devices

Figure 2.1 represents how the **CPU** and devices can share the main memory using **Direct Memory Access (DMA)**. To read from/write to the main memory, the CPU passes through the **Memory Management Unit (MMU)**, which is responsible for virtual/physical address translation. In a nutshell, when an operating system is used, every process uses addresses mapped in a virtual address space. The virtual address spaces are set up by the operating system. When a process wants to access the memory, e.g. for reading its value, the request goes through the MMU that translates the virtual address to a concrete physical address. Then, processes run in apparent isolation and can not access other processes memory without permission from the operating system. This architecture provides strong security guarantees, providing the soudness of the operating system (i.e. absence of bugs).

Similarly, devices can directly access the main memory using DMA. This technique enables to offload the CPU from copying each byte of memory by setting the DMA controller to do so. DMA controllers also give devices direct access to the main memory. While fast and convenient, DMA creates a whole new range of vulnerabilities. Indeed, if misconfigured, the DMA controller can give complete access to the main memory to devices, like kernel private memory, page table or executable memory [**schwarz_formal_2014**].

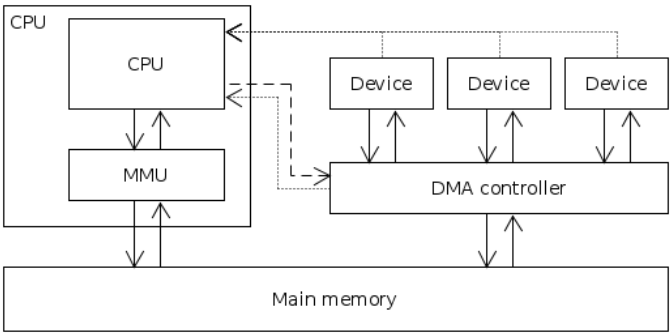

Figure 2.1: CPU schema from a memory point of view. The dashed line between the CPU and the DMA controller represent the capability of the CPU to send commands to the DMA controller through register writes. Dotted lines to the CPU represent the ability to raise interrupts. This schema doesn't represent caches.

2.2 The necessity of secure execution platforms

The **PROSPER** project isn't the only project focused on high-security execution platforms. Platforms such that seL4, Microsoft Hyper-V and INTEGRITY Multivisor are examples of platforms already used in production and providing strong security properties.

seL4 is a recent L4-based microkernel created in 2006 with the goal to produce a completely formally verified implementation of an L4 microkernel. This has been achieved in 2009 [**klein_sel4:_2009**]. At this time, seL4 consisted of 8700 lines of C code and 600 lines of assembler. The implementation of seL4 has been formally proved from its abstract specification down to its C implementation. Later, the validity of the generated assembly code has been proved, removing the need of trusting the compiler [**noauthor_what_nodate**].

Microsoft Hyper-V is Microsoft's hypervisor, widely used today within the Microsoft Azure cloud platform. It has been released in 2008. Hyper-V is a huge codebase, as we can read on VCC's website [1]: "Hyper-V consists of about 60 thousand lines of operating system-level C and x64 assembly code, it is therefore not a trivial target". Microsoft has put a lot of work in formal verification[2]of Hyper-V down to machine code [**leinenbach_verifying_2009**]. However, they don't appear to include

[1] https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/

device drivers in their formal verification.

INTEGRITY Multivisor is a commercial real-time operating system and hypervisor developed by Green Hills Software. Although not much information seems to be publicly available, Green Hills Software has done considerable formal verification work [**richards_modeling_2010**]. Multivisor has several certifications, including, for example, ISO 26262 ASIL D automotive electronics, NSA-certified secure mobile phones or FAA DO-178B Level A-certified avionics controlling life-critical functions on passenger and military aircraft [3].

MINIX 3 is an operating system whose design is focused on high reliability and security. It is based on a tiny microkernel with the least responsibility possible, and the rest of the operating system is running an a number of isolated user processes. MINIX 3 has not been formally verified, but intends to provide strong security guarantees by design.

2.3 Binary analysis platforms

For this project, we will use the **HolBA** framework. However, several other binary analysis platforms have been created for various purposes, such as formal verification or static analysis. A common characteristic of these platforms is to use an **Intermediate Representation (IR)**. IRs are designed to be simpler to use for each platforms' end purpose. As an example, the HolBA platform has BIR as its intermediate representation.

Microsoft Boogie is Microsoft's intermediate verification language. Boogie is the IR for multiple Microsoft tools, including VCC. Boogie as a tool can infer some invariants on the given Boogie program and then generate verification conditions that are passed to an SMT solver [4].

Valgrind is a framework for building program supervision tools, such as memory checkers, cache profilers or data-race detectors [**nethercote_valgrind:_2003**]. As its core, Valgrind is a JIT x86-to-x86 compiler, translating binary programs into its IR

[2] Microsoft has several formal verification projects, many of which are freely available for non-commercial use: https://github.com/Microsoft?q=verifier

[3] https://ghs.com/products/rtos/integrity_virtualization.html

[4] https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/

called UCode. Then, *skins*—tools built using the Valgrind framework—are free to work with the IR in order to perform their analysis.

LLVM is a compiler infrastructure which supports a unique multi-stage optimization system [**lattner_llvm:_2002**]. LLVM is built around its **IR**, LLVM Virtual Instruction Set, which can be described as a strict RISC architecture with high-level type information. This IR made LLVM successful because it is a pragmatic IR suitable for optimizations at multiple stages (link-, post-link, and run-time) and supporting a wide variety of transformations. Leveraging this IR, an ecosystem grew around LLVM, providing tools such as symbolic execution (LLVM KLEE), benchmarking environments or static and dynamic analyzers.

Mayhem is a system for automatically finding exploitable bugs in binary programs and generating working exploits as proof of the discovered vulnerabilities [**cha_unleashing_2012**]. It leverages BAP, the Binary Analysis Platform from Carnegie Mellon University (CMU BAP) [**brumley_bap:_2011**], as its IR. It proceeds by fist JIT-ing each instruction to the BAP intermediate language (IL) and then performing a custom symbolic execution.

There are several other tools, platforms, and frameworks, such as **CMU BAP** [**noauthor_binary_2019**] or Angr. An interesting note though is that BIR's design is based upon CMU BAP's intermediate language.

The novelty introduced in HolBA is that proofs are performed directly on the generated assembly code, not at the source code level. Therefore, proofs can be performed on programs without needing their source code, and regardless on the programming language used as long as it can be compiled in assembly code in an **ISA** supported by the platform.

Chapter 3

Overview of the formal verification of the NIC device

*This chapter will introduce the formal verification of [***haglund_formal_2016***] that will serve as a guideline throughout this work. It will first introduce Interactive Theorem Proving and formal verification in HOL4, then give a brief overview of the formal NIC model and its proof.*

3.1 Interactive Theorem Proving and HOL4

Interactive theorem provers are software producing formal proofs, in an interactive fashion, i.e. a human can step through the proof interactively while the proof assistant provides some automation (like rewriting of terms, arithmetic evaluation, integration with external tools like SMT solvers, ...). Coq, HOL4 or Isabelle are such tools.

HOL4 [**noauthor_hol_nodate**] stands for Higher-Order Logic. It is a programming environment deeply embedded into the **Standard ML (SML)** programming language enabling to prove theorems and write **proof-producing** programs. Since its first version in 1988, HOL has been focused on hardware verification and has been successfully used in this domain, as shows its `examples/` directory [**noauthor_canonical_2019**].

HOL4 is built around a very small kernel: 3 axioms, 10 inference rules, two predefined constants, and two types [**tuerk_interactive_nodate**] [1]. A theorem can only be built using this core, every other higher-level theory must be built from lower-level ones. Furthermore, the kernel consists of very few lines of code, leaving less space for bugs. Hence, for this reason, the trust needed is far inferior to standard systems.

[1] This is true for the Hol Light Kernel, however, the HOL4 kernel has slightly deviated from this simplicity for historical and performance reasons.

A proof is a demonstration that a proposition is true. In HOL4, proofs are held inside *theorems*, stating that its proposition *can be proved*. This theorem prover offers two ways of building proofs: in a forward and a backward style. A forward proof derives new theorems using axioms and inference rules: one starts with the building blocks and describes how to combine them. Backwards proofs start instead from the goal to be proved and progress in the proof towards obtaining axioms or known theorems. HOL4 puts emphasis on backward proofs, using tactics. Since every theorem is inferred from the small kernel, forward proofs must be ultimately used. Therefore, automation and tactics are written as forward proofs, and used in backward proofs.

To illustrate the idea of forward and backward proofs, and tactics, let's prove the same proposition using the two approaches (since forward proofs quickly requires wide knowledge of HOL4, the following theorem is a simple contrived example).
Theorem 3.1.1.

$$\forall p.\ p \implies p \tag{3.1}$$

Forward proof. Let's start with a free variable p of boolean type. Then, assuming p we have that p is true (assumptions are written in square brackets):

$$[p]\ \ p$$

Let's discharge the hypothesis. We obtain:

$$p \implies p$$

Finally, we generalize this theorem over all boolean variable, concluding the proof. □

In this proof, the discharge and generalize operations are inference rules, available as base lemmas already proved from the kernel. In the following backward proof, emphasised words in squared brackets are tactics corresponding to the written explanation of the sentence.

Backward proof. Universal quantifier can be added to free variables, so let's remove them [*GEN_TAC*]. The goal becomes:

$$p \implies p$$

Let's discharge the implication into an assumption [*DISCH_TAC*]. The goal becomes:

$$[p]\ \ p$$

Finally, we can use the assumptions in order to prove the goal [*ASM_REWRITE_TAC*]. □

In this contrived example, the two proofs are exact opposite of each other. In more complex proofs, it is often infeasible to use forward proofs because it requires to many operations and doesn't offer automation.

In HOL4, a theory is a coherent set of theorems and definitions. Theories are written in **SML** script files, then compiled into a theory file. A definition is the equivalent concept than when using the $\stackrel{def}{=}$ symbol: a symbol or function is defined to be a synonym of a more complex formula.

3.2 Overview of the formal NIC model

This part is a bit dense. I do not know how much is it clear for a reader that has no previous knowledge about the NIC

The formal NIC model of [**haglund_formal_2016**] has been designed from reading the hardware specification of the device because no model nor device driver is freely available.

The NIC model is designed as a transition system with four types of transitions: register read, register write, autonomous, and memory read request reply transitions. They are described by four functions which constitute its public interface: `read_nic_register`, `write_nic_register`, `nic_transition_autonomous` and `memory_read_reply`. Each of these functions update the given NIC state, simulating the real behavior of the NIC. To make the model sound, the state is marked *dead* if the model is asked to describe any transition or operation that is not described by the specification. Dead states represent undefined states and cannot be left with further transitions.

The model is composed of five finite state automata describing the inner transitions of the whole system, each automata describing a part of the NIC. The five automata are initialisation, transmission, transmission teardown, reception and reception teardown. The function `nic_execute` performs one autonomous step of one of the five automata. The automaton that takes a step is decided by a scheduler, depending on the NIC inner state. If more than one automaton are in a ready state, their order is not deterministic [2]. Each finite state automata is defined as a set of transitions working on NIC states. Each transition takes a NIC state and returns an update one, possibly dead.

Being designed as a transition system, each of the four functions described earlier is loop-free, aspect that is crucial in order to apply contract-based verification as described in Section 5. Each of the transition is implemented as a HOL4 function using

[2] To reason about non-determinism, the notion of oracles has been introduced in [**haglund_formal_2016**]. To put it simply, an oracle can be seen as a list of steps to perform, but where each element of the list is undefined. As we will not directly reason about the oracle, we will not explain how this works in more details. The reader can read a thorough explanation in [**haglund_formal_2016**].

exclusively **if-then-else** and state modification statements. There is no recursive definition. Hence, every function can be seen as some kind of decision tree in which some of the nodes "mutate" the state.

The five automata are of different complexity, as shown in Table 3.1, with the initialization automaton having only 4 transitions of which only one is autonomous, and the reception automaton being the most complex with 20 transitions. Calling the `nic_transition_autonomous` function makes one of the automata take an autonomous step. Non autonomous transitions are performed by the three other functions.

Automaton	# of (autonomous) transitions	LoC (w/o comments)
Initialization	4 (1)	21
Transmission	7 (5)	182
Transmission teardown	5 (4)	67
Reception	20 (20)	280
Reception teardown	7 (6)	93

Table 3.1: Statistics on each of the automata in the formal NIC model.

The NIC state is defined as a nested *datatype*—a record [3] of records of words, booleans and enumerations. However, there is one exception: the NIC works on a memory data structure, called `CPPI_RAM`, which is represented in the formal model as a function from addresses to values.

The structure of the implementation of `read_nic_register` and `write_nic_register` is similar to the implementation of `nic_transition_autonomous` in terms of the statements they use. `memory_read_reply` is implemented as a non autonomous transition of the transmission automaton.

3.3 Overview of the formal proof of the NIC model

It has been formally proved with HOL4 that with **PROSPER**'s hypervisor, augmented with a monitor and in the presence of a **NIC**, only signed Linux code is executed. This security policy has been defined as an invariant of the NIC state, then the proof consists of verifying that each transition of the NIC transition system preserves the invariant. This invariant states that the NIC isn't in an undefined state (represented by the dead state), that the transmission and reception queue are well-behaving (non-overlapping, finite) and that the NIC is in a valid state (restrict the values of the NIC state variables to valid configurations). This invariant is stated as a conjunction of the invariants of each automaton plus a not-dead conjunct:

[3] Records in HOL4 are analogous to structures in imperative languages.

$$I_{NIC} \stackrel{def}{=} \neg NIC.dead \wedge I_{init} \wedge I_{tx} \wedge I_{td} \wedge I_{rx} \wedge I_{rd} \tag{3.2}$$

Each sub-invariant is proved individually on its respective part of the NIC model, then they are composed together in order to form the final one in Equation 3.2.

The majority of the lemmas are phrased in the same way, that we shall see in Chapter 5 correspond to the equation of Hoare Triples:

$$I\ nic \wedge nic' = transition\ nic \implies I\ nic' \tag{3.3}$$

3.3.1 Visualizing proof dependencies

The model of the NIC consists of 1500 lines of HOL4 code and required around three man-months of work. The NIC invariant consists of 650 lines of HOL4 code and the proof consists of approximately 55 000 lines of HOL4 code including comments. Identifying the invariant and implementing the proof in HOL4 required around one man-year of work [**haglund_trustworthy_nodate**].

? is to . remove ,

The proof being consequent, it is composed of a multitude of lemmas spread out in multiple layers. The ideal objective of this work being to prove again the base lemmas on a different but equivalent BIR program, we therefore need to identify the relevant lemmas to be replaced. The proof is divided into two major parts, reception, and transmission, and lemmas are scattered across a few files, each file being for a different part or abstraction level. Optimally, identifying the fringe of the dependency graph of the lemmas and theorems would give exactly the set of lemmas that all the other lemmas and theorems rely on, and therefore the smallest set of lemmas that are enough to prove in order to imply the security properties already proved in [**haglund_trustworthy_nodate**] by using the existing proofs.

As an attempt to visualize this fringe, we developed a tool, called DepGraph [4], that can extract the dependency structure of HOL4 proofs in the form of a dependency graph. DepGraph can extract dependencies between HOL theories, i.e. compiled **SML** files containing proofs of lemmas and theorems, and between definitions, theorems, and lemmas. Figure 3.2 shows the dependencies between theories about the transmission part of the NIC proof, and Figure 3.3 shows the dependencies between definitions, theorems and lemmas of the same part of the proof. However, as can be seen on Figures 3.2 and 3.3, this tool presents some critical shortcomings:

- The theory dependencies exporter uses files generated by Holmake, the HOL4 compile system, in order to get the dependencies between theories. However,

[4] DepGraph's design is presented in Appendix A.

those files don't really represent dependencies but the files to be loaded before this script can be loaded, in a recursive fashion. Therefore, they represent the transitive reduction of the dependency graph. Figure 3.1 presents transitive reduction. Because of this fact, precious knowledge is lost and cannot be recovered by using this method: edges representing direct dependencies can be removed if the remaining edges still account for this dependency. Therefore, we are still able to tell which nodes depend on some node n, but we cannot identify the aforementioned fringe. In order to solve this problem, different approaches exist, such as implementing a simplified SML parser that looks only at dependencies, or injecting code inside the dependency resolution of an existing SML compiler. However, this would involve too much work that isn't the direct focus of this thesis.

- The definition, theorem and lemma dependencies exporter uses word-based heuristics in order to extract dependencies, and is as such not quite reliable and cannot give any guarantee. As above, there exist similar solutions in order to get multiple levels of guarantees, such that implementing a SML parser or injecting code inside HOL4 theory and definitions handling, but this would also require too much work. Destructuring HOL4 theories does not work because of how HOL4 has been designed, i.e. no trace is kept on disk about how to prove a given saved theorem, except of course with the corresponding script file. Moreover, such dependency graphs become quickly big, making them unusable in practice, and some additional work would be needed in order to represent them in a convenient way. Therefore, as above, no further work has been put into this exporter.

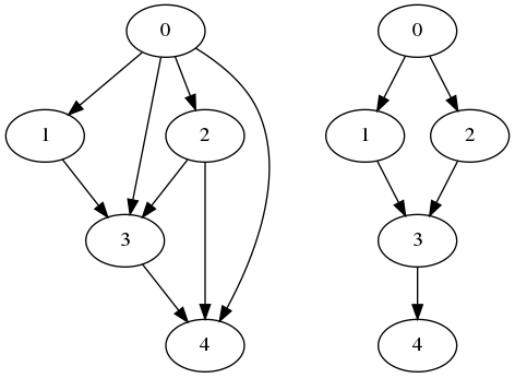

Figure 3.1: The right-hand side graph represent the transitive reduction of the left-hand side graph. Edges $(0 \to 3)$, $(0 \to 4)$ and $(2 \to 4)$ have been deleted because the transitive dependency relationship is preserved in the remaining edges. For example, the dependency $(0 \to 3)$ is represented in paths $(0 \to 1 \to 3)$ and $(0 \to 2 \to 3)$.

Figure 3.2: Dependency graph between theories about the transmission part of the formal NIC proof. An edge from a node A to a node B means that A depends on B. On this figure, edges are directed from left to right. Therefore, on this figure, higher-level files are on the left and lower-level ones on the right. Yellow nodes represent theories that depend on *txTheory*, i.e. the theory describing the transmission automaton presented in Section 3.2. Red nodes represent the leaves of the dependency graph, i.e. files without dependencies. Each node's edges are of different colors.

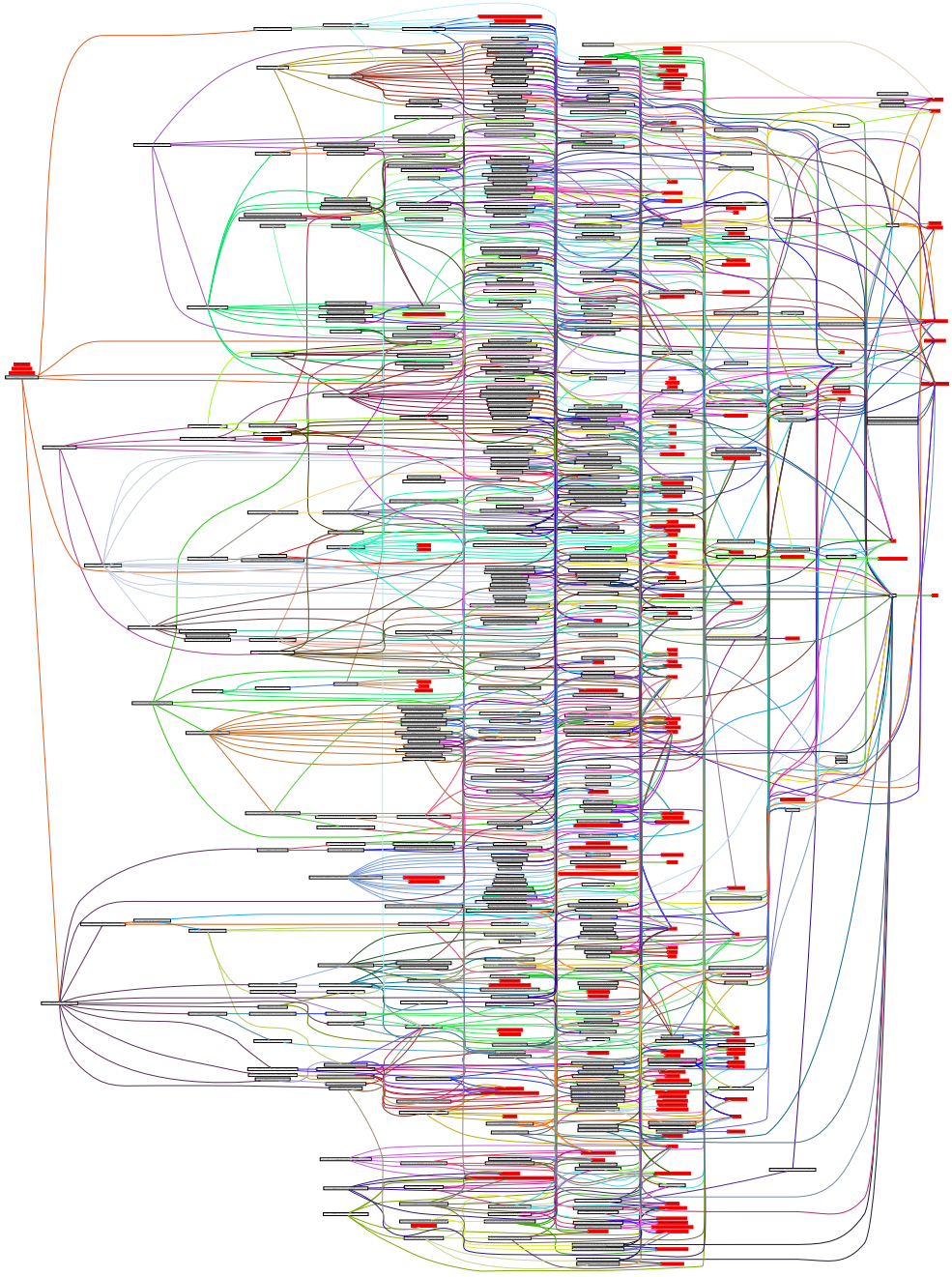

Figure 3.3: Dependency graph between definitions, theorems, and lemmas of the transmission part of the formal NIC proof. An edge from a node A to a node B means that A depends on B. Therefore, on this figure, higher-level files are on the left and lower-level ones on the right. Each node's edges are of different colors. Red nodes represent leaf nodes, i.e. theorems and definitions without dependencies. The red nodes on the left-hand side of the Figure are nodes for which the heuristic failed to capture the dependencies.

Chapter 4

The BIR NIC model

This chapter discusses the different ways of implementing an equivalent BIR model from the formal NIC model. After an introduction of BIR, it will present different possible ways of doing this translation and present three approaches. Then, the final BIR model will be introduced along with the new tools that have been implemented in order to build it.

4.1 HolBA's Binary Intermediate Representation (BIR)

HolBA's Binary Intermediate Representation (BIR) [**lindner_trabin:_2019**], introduced in the Introduction, is a machine independent binary representation. It aims to be the simplest possible while still being able to represent all possible binary programs but self-modifying programs. It does so by having a limited syntax—introduced in Table 4.1—and forbidding implicit side-effects. A statement can only have explicit state changes and can only affect one variable.

This representation allows producing proofs more easily that with classical binary representations, whose design are focused on execution speed rather than offline analysis. Moreover, BIR doesn't have unspecified behavior.

BIR is implemented as a set of HOL4 *datatypes*, and possesses a completely defined semantic. Section 5.5 contains a more thorough discussion of the BIR semantic. Section 4.2.3 implements a toy BIR program and presents the concrete BIR syntax.

Among its supporting tools, HolBA features a tool to visualize the **Control Flow Graph (CFG)** of BIR programs.

$$
\begin{aligned}
prog &:= block^* \\
block &:= (string \mid integer,\ stmt^*,\ estmt) \\
stmt &:= \textbf{assign}\ (string, exp) \mid \textbf{assert}\ (exp) \\
estmt &:= \textbf{jmp}\ (exp) \mid \textbf{cjmp}\ (exp,\ exp,\ exp) \mid \textbf{halt}\ (exp) \\
exp &:= integer \mid \textbf{var}\ string \\
&\quad \mid \textbf{if-then-else}\ (exp,\ exp,\ exp) \\
&\quad \mid \diamond_u\ exp \mid exp\ \diamond_b\ exp \\
&\quad \mid \textbf{load}\ (exp,\ exp,\ \tau) \mid \textbf{store}\ (exp,\ exp,\ exp,\ \tau)
\end{aligned}
$$

Table 4.1: BIR's syntax. Valid BIR programs must be well-typed. $integer$s represent bounded N-bit integers. $\diamond_u$ and $\diamond_b$ represent respectively unary and binary operators. BIR blocks are tuples, with the first element being its label, the second a list of statements and the third the end statement. BIR syntax contains some other statements that won't be used in this work and that have been omitted. For more information, see [**lindner_trabin:_2019**].

4.2 Exploring multiple approaches to translating the formal NIC model

Multiple approaches to the translation of the formal NIC model described in Section 3.2 to an equivalent BIR program have been considered:

- **handwritten BIR program**: The most straightforward way would be to directly write the BIR program by hand by looking at the actual formal model and directly converting it to BIR. This is easily feasible because of the limited set of statement used in the implementation of the NIC model which are all representable in the BIR syntax. However, this does not seem optimal if more than one such translation need to be performed. One remedy of this problem would be to implement a set of tools that facilitate the implementation of device models and reduce the boilerplate. This idea is explored in the following Sections 4.2.3 to 4.2.5.
- **lifted C program**: An alternative would be to implement the NIC model in a more convenient higher-level programming language, compile it to assembly code and leverage the existing lifter to generate a BIR program. One advantage of this method is the possibility to use existing frameworks and tools that already exist in other languages, in which the ease of development is higher. However, the gained ease of development is maybe not worth the complexity introduced

by those steps. The following Section 4.2.2 experiments this idea.

- **device model specific IR**: This idea rejoins the previous one in the sense that it uses an additional intermediate representation (C in the previous idea) to implement the model, then uses some compiler to generate the BIR program (standard C compiler and HolBA's lifter). This idea would have the advantage that it introduces a new higher-level IR that would be common to possible later other device modelizations. Tools could also be developed for this IR. However, this approach would require to develop a new IR and, if we later want to develop a **proof-producing** lifter (compiler) the implementation would be costly, especially since this language would be more complex. The following Section 4.2.1 tries to use visual flowchart language as IR.

4.2.1 Using flowcharts as Intermediate Representation

As already discussed in the previous Section, the structure of the formal model looks like a tree. Therefore, using flowcharts can be a convenient way to represent such structures. An attempt has been made to design a flowchart representation. Figures 4.1, 4.2 and 4.3 show respectively a preview of the scheduler, transmission automaton and particular transition of this automaton.

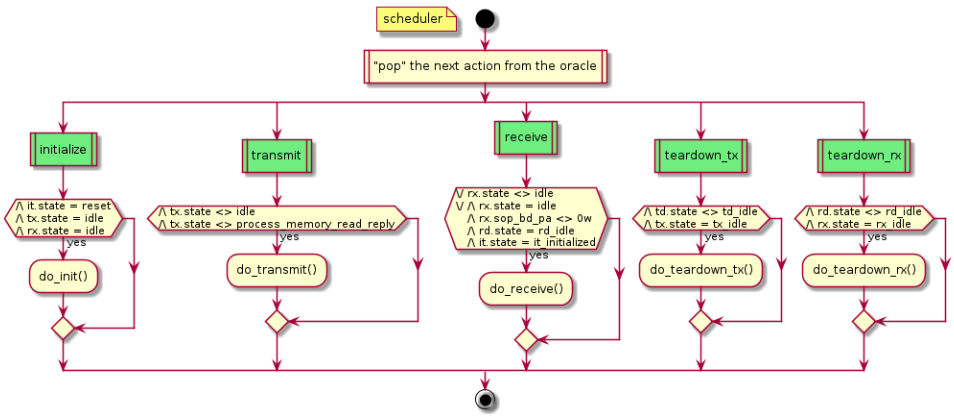

Figure 4.1: Flowchart of the scheduler of the NIC model. Green nodes represent condition statements, here they represent the value of the popped action from the oracle. The full dot represents the entry point, and the other point the exit point.

While this visual representation was useful to get to know the formal NIC model, we encountered several shortcomings:

- Flowcharts of each transition rapidly grew in size with the complexity of its formal counterpart. Possible workarounds include the use of nested diagrams,

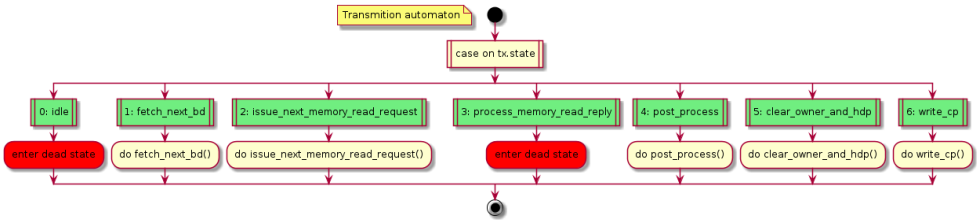

Figure 4.2: Flowchart of the transmission automaton of the NIC model. Green nodes are similar to the ones of Figure 4.1. Red nodes represent non autonomous transitions leading to dead states.

as it is the case of Figure 4.3 representing one node of Figure 4.2, namely `fetch_next_bd`, or usage of shorter ways of representing common patterns, as it is the case for representing dead transitions on Figure 4.3.

- It is hard to define a coherent visual language able to represent the full set of features needed in order to realize device models. Additionally, this language must be compatible or easily translatable to BIR.
- It is hard to design a textual representation of this visual language other than conventional programming languages, so using such representation would require a substantial implementation effort in order to implement all the tools needed to use it. Developing visualization tools for a conventional language appears to be a more reasonable approach than developing a visual language.

For those reasons, it has been decided to not go further with this visual representation, and to focus instead of existing tools of the HolBA platform. However, as we shall see in the rest of this thesis, we will explore other visualization tools, and flowcharts will be used as a visual help while designing the model using other methods.

4.2.2 Writing the model in C

One-to-one translation of the transmission automaton, scheduler, state, and *CPPI_RAM* related definitions of the formal model have been realized in C. This translation has been quite easy to perform and has been completed in roughly one day, and no difficulty arose during this translation. Regarding the NIC state, C has all types needed to represent it. An array has been used to represent the function type of *CPPI_RAM*. Enumerations have been used in order to represent HOL4 enumerations. While not optimal because of their non-restricted usage allowed by the C language—they can be freely modified as integers without checking their validity in C, while HOL4 and functional languages in general are restrictive and force to use correct values—enumerations

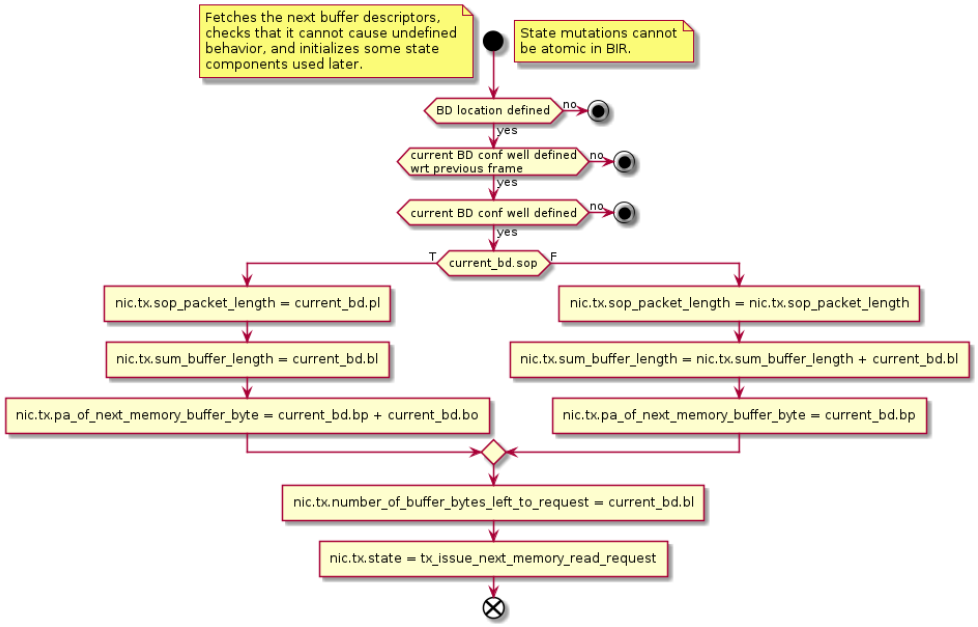

Figure 4.3: Flowchart of the `fetch_next_bd` transition of the transmission automaton of the NIC model. The full dot represent the entry point, ⊗ represents the exit point and the other dots are shorthands to represent dead transitions (the symbols have been changed because of technical limitations of the tool used to draw the diagram).

are enough if used correctly while being well-aware of those shortcomings.

When studying the compiled assembly code and lifted BIR program, we noticed that all the convenient naming that we can use in the formal or the C model is lost and replaced with abundant usage of the stack. While this is completely normal behaviour for a C compiler, this is not convenient when performing later proofs on the model: we would first need to rename them in the proof using some definitions, and this process would be lengthy and cumbersome (if not automated), resulting in more code than if the model had been directly written in BIR in the first place. This experiment made us realize that writing the model is a rapid operation and that we should rather focus on making the verification step as smooth as possible because it is the most difficult one to perform.

For those reasons, it has been decided to try using BIR directly to write the NIC model. The following Section 4.2.3 presents a first prototype that has been realized in order to visualize the shape of a BIR program representing the NIC model, and identify the tools that would be needed in order to reduce the boilerplate of such implementation in BIR.

4.2.3 Implementing a toy BIR model

Before writing the whole NIC model by hand, we shall identify the structure of the model and develop tools that facilitate its implementation. Using well-designed tools can reduce the boilerplate work of the implementation, helping to focus only on the meaningful content of the implementation, and can also reduce the chance of introducing bugs as the code is factored and mechanically shorter.

It has been decided to implement a transition system similar to the one of the NIC model, with two inner independent automata: Alice and Bob. The two automata feature a simple linear transition system, and each of them has a non-autonomous transition, that is performed respectively by two external functions `bus_arrived` and `taxi_arrived` that represent memory accesses from the CPU in the NIC model. The function `autonomous_step` performs one autonomous transition via the scheduler. Figure 4.4 gives an overview of the two automata, Figure 4.6 shows the role of the three functions and Figure 4.7 describes in details Alice's automaton. (Flowcharts from the previous Section 4.2.1 have been used in this section in order to facilitate designing the toy model. However, no effort has been made to come up with a coherent and formalized language.)

From the design presented in Figures 4.4, 4.6 and 4.7, writing the BIR program is a repetitive but straightforward step. The resulting BIR program is 450 lines of code long. The following issues have been identified:

- BIR, as a HOL4 embedded language, is very verbose. Simple operations like additions or assignments require long constructions, as shown in Listing 4.1. Section 4.2.4 presents BSL, a less verbose way of writing BIR programs.

Listing 4.1: BIR

```
(* A1. Eating *)
<|bb_label := BL_Label "alice_automaton.step.1";
  bb_statements := [
    (* state.alice.hunger -= 20 *)
    BStmt_Assign (BVar "state.alice.hunger" (BType_Imm Bit32))
      (BExp_BinExp BIExp_Minus
        (BExp_Den (BVar "state.alice.hunger" (BType_Imm Bit32)))
        (BExp_Const (Imm32 20w)));
    (* state.alice.step := 2 *)
    BStmt_Assign (BVar "state.alice.step" (BType_Imm Bit32))
      (BExp_Const (Imm32 2w))
  ];
  bb_last_statement := BStmt_Jmp (BLE_Label (BL_Label "alice_automaton.end"))
|>;
```

- BIR features only one conditional statement controlling the control flow of a program: conditional jumps. Hence, BIR is not convenient for representing **if-then-else** statements with more than two branches. Section 4.2.5 presents some helper functions that have been implemented in order to be able to abstract the raw BIR code and work at a higher level.

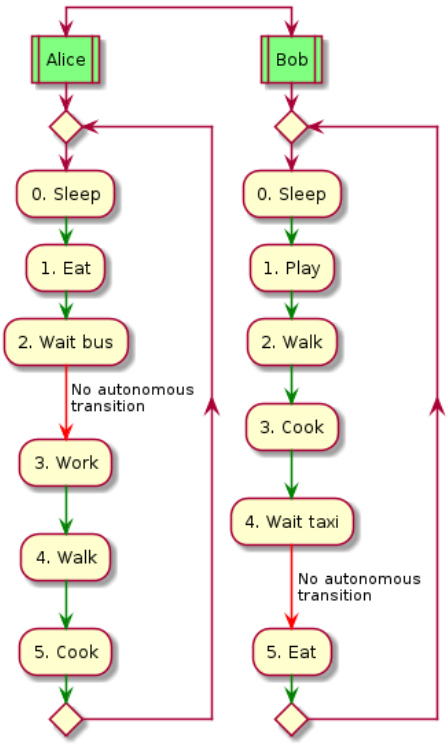

Figure 4.4: Overview of the two independent automata. The green arrow represent the autonomous transitions and the red ones the non-autonomous. As in Section 4.2.1, the green nodes represent conditional statements depending on the scheduler's oracle, as shown in Figure 4.5.

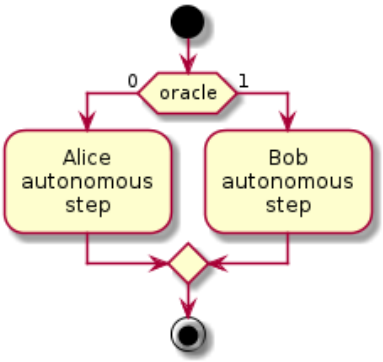

Figure 4.5: Scheduler of the toy BIR model. We can see how the oracle decides which automaton takes a step. If an automaton is in a state whose transition isn't autonomous, the automaton state is returned unchanged.

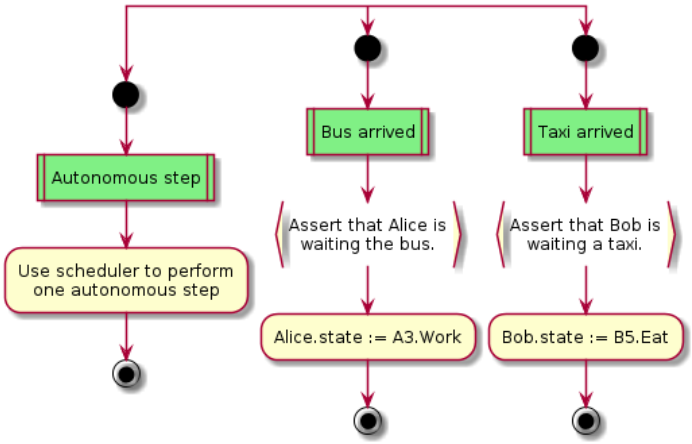

Figure 4.6: The three entrypoints of the toy program: `autonomous_step`, `bus_arrived` and `taxi_arrived`.

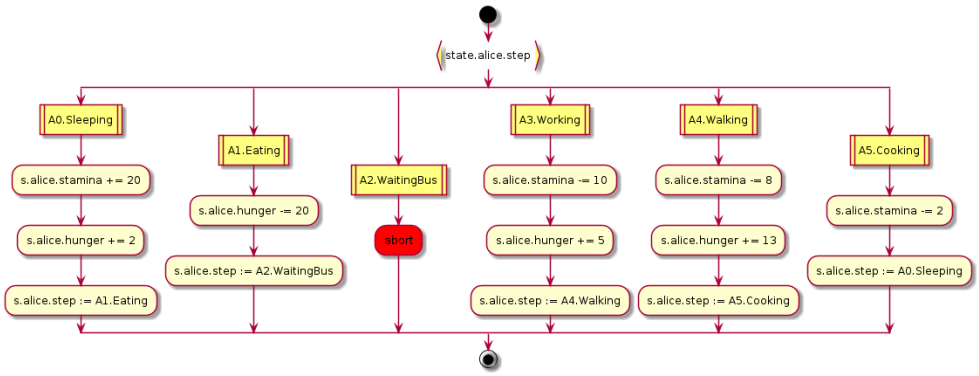

Figure 4.7: The six transitions of Alice's automaton. Each transition only updates the state. Conditional statements hasn't been added in this toy model because it already features conditional statements in the scheduler. The third transition is a non-autonomous transition; `bus_arrived` is in charge of updating Alice's state from *WaitingBus* to *Working*.

Listing 4.2: Example of prelude of a SML script that defines functions tailor-made to its use. The function *o* is used for function combination.

```
val bvarstate = bvarimm32
val bdenstate = (bden o bvarstate)
val bstateval = bconst32
val bjmplabel_str = (bjmp o belabel_str)
```

4.2.4 BSL: BIR Simple Language

Because of its extreme verbosity, BIR is not a language convenient to manually use. Its primary use has been as an output from the **proof-producing** lifter and as a machine-independent intermediate representation with a very limited feature-set convenient to reason about. A less verbose and more convenient method for producing BIR code is needed in order to open this language to other uses. Therefore, a library has been written that offers the same expressiveness than BIR but with shorter constructs. This library has been kept simple and will serve as the base layer of possible later abstractions. As such, it has been decided that no feature other than pure syntactic construct, such that type inference, would be included. This library has been named BIR Simple Language (BSL) and is implemented in the file *bslSyntax.sml* [1].

BSL is composed of a set of functions with short names and a coherent interface. Since this library will be directly included in every **SML** script file in order to avoid usage of the lengthy `bslSyntax.function_name` SML construction, we decided to prefix every function with the letter `b`. As BSL doesn't feature type inference, types must be explicitly given when necessary. For every such function, such as `bconst`, implementations for every possible type have been added, in addition to general functions taking the size as an argument. Thanks to partial function application in SML, composing functions is easy and scripts can create the set of functions that they need. However, in practice, a BIR program uses a very limited set of types and it is often enough in scripts to statically define the needed set of sized functions (as shown in Listing 4.2). Finally, BSL has been designed to be fully interoperable with HOL4 terms; it exists for every BIR operation a related BSL function taking a HOL4 term as a parameter, and every BSL function return a HOL4 term. Scripts can therefore not be limited by the restricted feature-set of BSL.

[1] This file has been named in order to follow the HOL4 conventions: **Syntax.sml* files contain functions that create, destruct and check HOL4 terms. *bslSyntax.sml* is a library for creating only BIR terms, and should, therefore, follow this convention.

Listing 4.3: Call to the `gen_state_helpers` function that generates a record containing the list of the states, the list of the autonomous states, a function from state name to state id, a function from state name to boolean telling if the state is autonomous and a function that generates BIR code of the state id from the state name. `gen_state_helpers` takes as parameters the name of the automaton and a list of its states with an id and a boolean telling if the state is autonomous.

```
val tx_state = gen_state_helpers "tx" [
  ("tx1_idle",                          (1, false)),
  ("tx2_fetch_next_bd",                 (2, true)),
  ("tx3_issue_next_memory_read_request",(3, true)),
  ("tx4_process_memory_read_reply",     (4, false)),
  ("tx5_post_process",                  (5, true)),
  ("tx6_clear_owner_and_hdp",           (6, true)),
  ("tx7_write_cp",                      (7, true))
]
```

4.2.5 Implementing the real model

From the knowledge gained in the previous section 4.2.3, a set of helper functions has been implemented, mainly in order to facilitate reasoning about the state machine. Listings 4.3 and 4.4 show the SML code respectively used to represent the transmission automaton and to generate the BIR blocks that jumps to the right label of the program depending on the current step of the automaton (analogous to a *case* or *switch* construct in higher-level languages).

Because of time constraints, not every transition of the formal NIC model have been implemented in this new model. Instead, we decided to focus on only some transitions in two automata: initialization and transmission. This focus is pragmatic for two reasons: (a) we decided to write the model along with the proof, only needed transitions for the current proof, in order to grow the model at a reasonable pace and to no clutter it with unneeded features or at the contrary missing critical aspects during the first sketch (b) as the verification was performed at the same time, we decided to start with easier, but not obvious, transitions, hence the choice to postpone verification of the more complex reception automaton to later in the work.

When visualizing the **CFG** of the BIR NIC model (Figure 4.8), we notice that it has the shape of a tree, rejoining the idea introduced in Section 3.2: each node of the tree is a BIR block, each edge a jump. Since every jump in the NIC model go to a static target, a jump can only have one or two destinations. Hence, nodes can only have one or two outgoing edges. In the model, each node represents either an empty jump block (possibly conditional) or a mutation of the state.

The static and loop-free control-flow of this BIR program is very suitable for use with

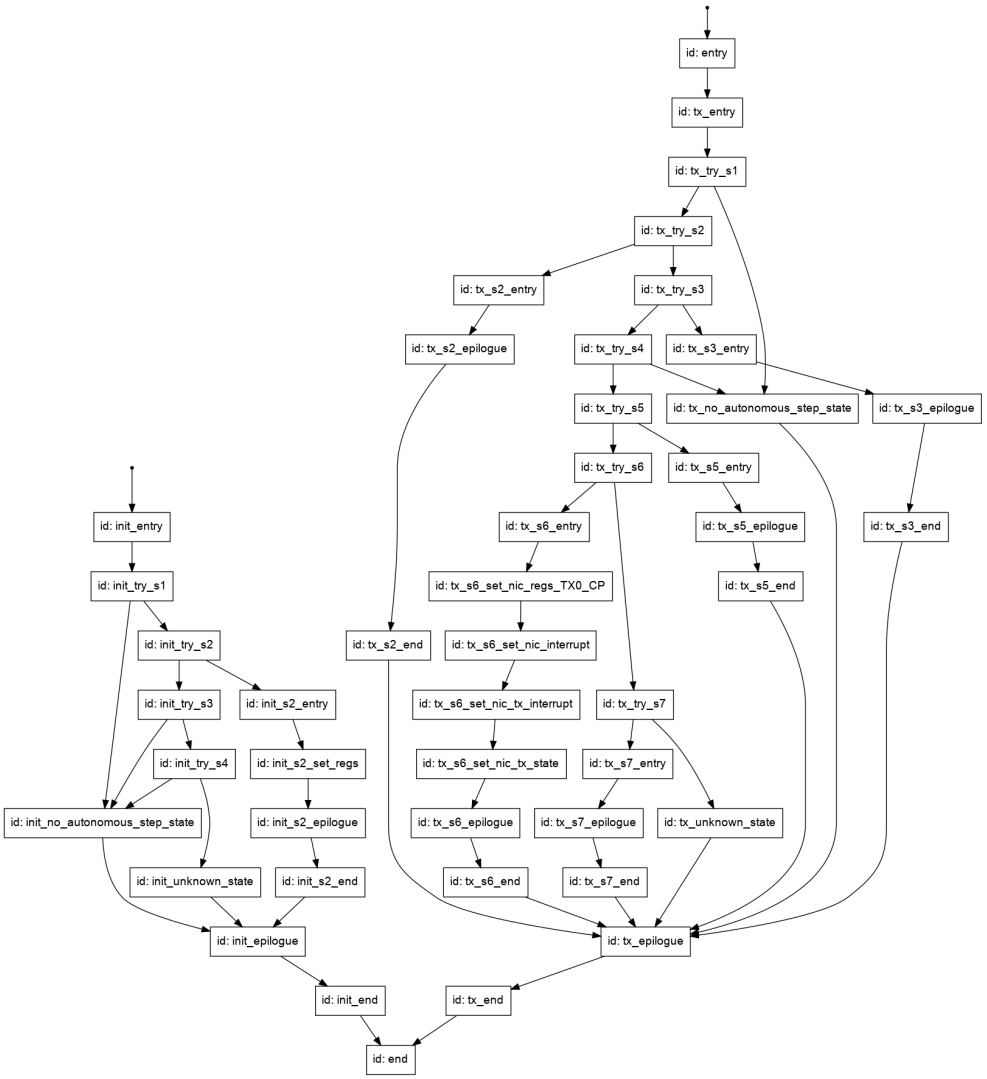

Figure 4.8: **CFG** of the implemented transitions of the NIC BIR model. Each node represents one BIR block and each edge one jump target. Because the scheduler hasn't been implemented in this model, the **entry** block is jumping arbitrarily on *tx_entry*; all contract verification procedures take labels in order to define entry and end points of the section of a program that the tools should generate the weakest precondition from, so this isn't a problem in practice as long as no attempt to perform proofs on more than one automaton are made. The CFG of the full program would resemble a tree with one node at the top, *entry*, and the tree getting larger with the depth because of all the conditional jumps that go to the transition being performed.

Listing 4.4: Autonomous transition jump table used for the transmission automaton. This table uses the names of each state and transition to generate the BIR blocks that will perform the conditional jumps depending on the value of the current transmission state. The parameters of the function are: a tuple of the name of the variable containing the current state, the label of the block to jump to when the current state isn't handled and a function that gives the state number from state name, and a list of jumps, each jump being a tuple of the label of the conditional BIR block, the value of the state that leads to this block and the name of the label to go to.

```
(* Autonomous transition jump *)
@ bstate_cases ("nic_tx_state", "tx_unknown_state", bstateval_tx) [
  ("tx_try_s1", "tx1_idle",                          "tx_no_autonomous_step_state"),
  ("tx_try_s2", "tx2_fetch_next_bd",                 "tx_s2_entry"),
  ("tx_try_s3", "tx3_issue_next_memory_read_request", "tx_s3_entry"),
  ("tx_try_s4", "tx4_process_memory_read_reply",     "tx_no_autonomous_step_state"),
  ("tx_try_s5", "tx5_post_process",                  "tx_s5_entry"),
  ("tx_try_s6", "tx6_clear_owner_and_hdp",           "tx_s6_entry"),
  ("tx_try_s7", "tx7_write_cp",                      "tx_s7_entry")
]
```

HolBA's weakest precondition tool in order to perform contract-based verification, as we shall see in the next chapter.

Chapter 5

Contract-based verification

This chapter will present contract-based verification, explaining the underlying theory: Hoare Triples, weakest precondition derivation and the use of SMT solvers. It will also present the current status of contract-based verification in HolBA, and present some of the work needed in order to reason about BIR memories with SMT solvers.

5.1 Hoare triples

Contract-based verification is a powerful approach for verifying programs. For a given program $prog$ consisting of a list of instructions and two predicates P and Q called respectively pre- and postcondition, a Hoare triple $\{P\}\ prog\ \{Q\}$ states that when executing the program $prog$ from a state S terminates in a state S', if P holds in S then Q will hold in S' (Equation 5.1). Hereafter, we assume programs and states to be well-typed.

$$\{P\}\ prog\ \{Q\} \triangleq S' = exec(S, prog) \implies P(S) \implies Q(S') \tag{5.1}$$

For example, $\{P\} \varnothing \{P\}$ holds because an empty program doesn't change the state of the execution. $\{n = 1\}\ n := n+1\ \{even(n)\}$, with $n \in \mathbb{N}$, holds because $1+1 = 2$, which is even.

In order to perform the verification, the Hoare logic introduces a set of axioms describing the effect of each instruction of a given language over the execution state [**hoare_axiomatic_1969**]. For an assignment $x := f$ where x is a variable identifier and f an expression without side-effects, Equation 5.2 defines the axiom of assignment, where $P[f/x]$ denotes the substitution of all occurrences of x by f in P.

$$\{P[f/x]\}\ x := f\ \{P\} \tag{5.2}$$

5.2 Weakest precondition derivation

While Hoare logic introduces sufficient preconditions, Dijkstra introduced the concept of necessary and sufficient preconditions, called "weakest" preconditions. Such weakest preconditions can be automatically derived from a program $prog$ and a postcondition Q. Let's call $WP(prog, Q)$ such a weakest precondition. Then, from Equation 5.1 follows:

$$\forall(prog, Q), \{\, WP(prog, Q)\} \; prog \; \{Q\} \tag{5.3}$$

For the program $n := n + 1$ mentioned in the previous section, we can generate the weakest precondition for the postcondition $even(n)$. First, we can rewrite $even(n)$ as $n \; MOD \; 2 \; = \; 0$ with MOD denoting the arithmetic modulo. Then, we derive the weakest precondition of the statement $n := n + 1$ by transforming the predicate $n \; MOD \; 2 = 0$ by substituting all occurrences of n by $n + 1$:

$$WP(\text{“}n := n + 1'', n \; MOD \; 2 = 0) = (n + 1 \; MOD \; 2 = 0) \tag{5.4}$$

From the properties of the modulo, we can simplify $n+1 \; MOD \; 2 = 0$ to $n \; MOD \; 2 = 1$ or $odd(n)$. Therefore, $\{odd(n)\} \; n := n+1 \; \{even(n)\}$, i.e. incrementing the value of an odd integer variable by one makes it even.

While the triple $\{n = 1\} \; n := n + 1 \; \{even(n)\}$ uses a sufficient precondition for establishing its postcondition, the triple $\{odd(n)\} \; n := n + 1 \; \{even(n)\}$ uses the weakest precondition. The later being the weakest precondition of the former, the two contracts are in relation:

$$n = 1 \implies odd(n) \tag{5.5}$$

More generally, for a triple $\{P\} \; prog \; \{Q\}$ to hold, P must be stronger than the weakest precondition, i.e. we need to prove that $P \implies WP(prog, Q)$:

$$(P \implies WP(prog, Q)) \implies \{P\} \; prog \; \{Q\} \tag{5.6}$$

5.3 Using SMT solvers to prove contracts

From Equation 5.6 we see that, in order to prove that a triple $\{P\} \; prog \; \{Q\}$, we need to prove $P \implies WP(prog, Q)$. While multiple methods exist to perform such proofs, **SMT** solvers offer a convenient and automatic solution.

Satisfiability Modulo Theories (**SMT**) problem is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions [**nikolaj_bjorner_programming_2019**]. **Satisfiability Modulo Theories (SMT)** problem is a generalization of **Boolean SATisfiability Problem (SAT)** problem supporting more theories. When given a formula, a **SMT** solver decides if the formula is satisfiable, i.e. if there exist a valuation of its variables where the formula evaluates to true. As a **SMT** solver can fail to decide a given instance, there are three possible outputs: "satisfiable", "unsatisfiable" and "unknown". Another useful feature of some **SMT** solvers is the ability to ask for a satisfying model, which represents a counter-example of a false predicate.

A predicate P holds if it evaluates to true for all possible values of its variables. Alternatively, the negation of a predicate $\neg P$ holds if there exist no valuation of its variables where the predicate evaluates to true, i.e. if the instance is unsatisfiable. Therefore, if a **SMT** solver report that $\neg P$ is "unsatisfiable", then P holds.

Another way of thinking about how to prove logical formulas with **SMT** solvers is by using De Morgan's Laws: we know that $\neg(P \implies WP) \equiv (P \wedge \neg WP)$. Therefore, proving that $\neg(P \implies WP)$ is "unsatisfiable" using an **SMT** solver can be seen as proving that there exist no model where P holds and WP doesn't.

5.3.1 The BitVector theory

To reason about fixed-size integers, **SMT** solvers often implement a "BitVector", or "FixedSizeBitVectors", theory. In order to understand its particularities, we can try to prove Equation 5.7. Hereafter, we will use Z3, a popular and efficient **SMT** solver implemented by Microsoft Research[1], and SMT-LIB 2.0, which is a standard format for **SMT** solvers [**barrett_satisfiability_2016**]. Listing 5.1 shows the SMT-LIB 2.0 representation of this proof attempt.

$$\forall x.\ x + 1 > x, \text{with } x \text{ an unsigned 32-bit integer} \tag{5.7}$$

Listing 5.1: SMT-LIB 2.0 representation of Equation 5.7.

```
(declare-const x (_ BitVec 32))
(assert (not
  (bvugt (bvadd x (_ bv1 32)) x)))
(check-sat)
(get-model)
```

When given Listing 5.7 as input, Z3 gives the following output:

[1] Z3 is available on GitHub at https://github.com/Z3Prover/z3/.

Listing 5.2: Z3 output for Listing 5.1.

```
sat
(model (define-fun x () (_ BitVec 32) #xffffffff))
```

Z3 is telling us that Equation 5.7 is false, and gives a counterexample: $x = 2^{32} - 1$. Indeed, with this value of x, $x + 1$ wraps around and result in 0 which is smaller than $2^{32} - 1$. This behavior is due to the bounded nature of fixed-size integers. The correct equation here would be:

$$\forall x.\ x \neq 2^{32} - 1 \implies x + 1 > x, \text{with } x \text{ an unsigned 32-bit integer} \tag{5.8}$$

Listing 5.3 and 5.4 show the input and output of Z3 used to successfully prove Equation 5.8.

Listing 5.3: SMT-LIB 2.0 representation of Equation 5.8.

```
(declare-const x (_ BitVec 32))
(assert (not
  (bvugt (bvadd x (_ bv1 32)) x)))
(assert (not (= x #xffffffff)))
(check-sat)
```

Listing 5.4: Z3 output for Listing 5.3.

```
unsat
```

5.4 Contract-based verification in HolBA

HolBA provides a **proof-producing** tool for automatically deriving weakest preconditions on loop-free **BIR** programs whose control flow can be statically identified [**lindner_trabin:_2019**]. This tool is proof-producing in that it proves Theorem 5.9 which is the instantiation of Definition 5.10, with $(p,\ entry\_l,\ end\_ls)$ defining the program, wp the derived weakest precondition, $post$ the given postcondition.

$$bir\_exec\_to\_labels\_triple\ prog\ entry\_l\ end\_ls\ \mathbf{wp}\ post \tag{5.9}$$

$$
\begin{aligned}
\vdash \forall &(prog : \alpha\ bir\_program\_t)\ (entry\_l : bir\_label\_t)\ (end\_ls : bir\_label\_t \to bool)\\
&(pre : bir\_exp\_t)\ (post : bir\_exp\_t).\\
&bir\_exec\_to\_labels\_triple\ prog\ entry\_l\ end\_ls\ pre\ post \Leftrightarrow\\
&\forall\ (s\ : bir\_state\_t)\ (r\ : \alpha\ bir\_execution\_result\_t).\\
&\qquad bir\_env\_vars\_are\_initialised\ s.bst\_environ\ (bir\_vars\_of\_program\ prog)\\
&\Longrightarrow\ s.bst\_pc.bpc\_index\ =\ 0\ \wedge\ s.bst\_pc.bpc\_label\ =\ entry\_l\\
&\Longrightarrow\ s.bst\_status\ =\ BST\_Running\\
&\Longrightarrow\ bir\_is\_bool\_exp\_env\ s.bst\_environ\ pre\\
&\Longrightarrow\ bir\_eval\_exp\ pre\ s.bst\_environ\ =\ bir\_val\_true\\
&\Longrightarrow\ bir\_exec\_to\_labels\ end\_ls\ prog\ s\ =\ r\\
&\Longrightarrow\ \exists\ (obs\ : \alpha\ list)\ (step\_count\ : num)\ (pc\_count\ : num)\ (s'\ : bir\_state\_t).\\
&\qquad r\ =\ BER\_Ended\ obs\ step\_count\ pc\_count\ s'\\
&\quad \wedge\ s'.bst\_status\ =\ BST\_Running\\
&\quad \wedge\ bir\_is\_bool\_exp\_env\ s'.bst\_environ\ post\\
&\quad \wedge\ bir\_eval\_exp\ post\ s'.bst\_environ\ =\ bir\_val\_true\\
&\quad \wedge\ s'.bst\_pc.bpc\_index\ =\ 0\ \wedge\ s'.bst\_pc.bpc\_label\ \in\ end\_ls
\end{aligned}
\tag{5.10}
$$

It is to be noted that this tool doesn't produce a theorem stating that the generated expression is actually *the* weakest precondition. However, this theorem isn't needed to perform contract-based verification if the generated "weakest" precondition is weak enough so that our precondition can imply it. However, without this theorem, it is impossible to prove that a given precondition P isn't strong enough to establish the postcondition. We will still use the term "weakest precondition" as it is in practice how we are using this tool.

Definitions 5.10 introduces additional conditions about well-typedness and initialization that are needed in BIR today[2], as well as the notion of "Block Program Counter" for multi-statement blocks.

This tool doesn't provide a simple interface to compute weakest preconditions for a given program and postcondition, nor does it provide and support for proving the relation between the precondition and the weakest precondition. Then, in order to prove that the Hoare triple holds from this generated Theorem 5.9, we need to prove:

$$bir\_exec\_to\_labels\_triple\ p\ entry\_l\ end\_ls\ \mathbf{pre}\ post \tag{5.11}$$

[2]Removal of the need of initialization is being discussed at the time of the writing, because actual hardware registers and memories are in facts always initialized: https://github.com/kth-step/HolBA/issues/63

Assuming well-typedness and initialization, after rewriting the definition of $bir\_exec\_to\_labels\_triple$, we have to show bir_eval_exp **wp** $s.bst\_environ = bir\_val\_true$ in order to prove our goal using the *modus ponens* with Theorem 5.9. This correspond to proving the following implication:

$$\begin{aligned} &bir\_eval\_exp \textbf{ pre } s.bst\_environ = bir\_val\_true \\ \implies &bir\_eval\_exp \textbf{ wp } s.bst\_environ = bir\_val\_true \end{aligned} \tag{5.12}$$

In Equation 5.12 we can recognize Equation 5.6 that we discussed how to prove using **SMT** solvers in Section 5.3. However, the expressions are expressed as BIR expressions. We then have to find a way to use an SMT solver [3]. This is the focus of the following of this thesis. Section 6 will use a non **proof-producing** method for translating those BIR expressions into an equivalent formula that SMT solvers can work on, then focus on automating the whole verification process. Chapter 7 will complete this proof and use it to lift properties that have been proved on the BIR implementation to the **NIC** model. The following Section 5.5 will discuss how to make proofs about BIR memories using **SMT** solvers.

5.5 BIR memories and SMT solvers

HOL4 features a library for interfacing **SMT** solvers and HOL4, called *HolSmtLib*. This library supports Yices 1 and Z3 as external provers. Yices 1 being an abandoned project that doesn't support SMT-LIB 2.0, we will focus on Z3 and the standard format SMT-LIB 2.0 [**barrett_satisfiability_2016**]. While *HolSmtLib* supports export for some SMT-LIB 2.0 theories, it doesn't support the *ArraysEx* theory and doesn't know about BIR. In Section 5.4, we discussed the translation from BIR expressions to *wordsTheory*. However, this theory doesn't contain anything about memories or arrays in general. Therefore, some modifications are needed.

BIR memories are semantically defined as functions from addresses to values.

There exist five types of BIR expressions operating directly on memories (cf. Listing 6.2 for the list of BIR expressions, and Section 6.1 for a more precise discussion of the BIR semantic):

- `BExp_Den`: this operation enables reading values from the environment. It is analogous to reading registers or the memory in assembly programs. This

[3] HOL4 *computeLib* provides a convenient way for turning expressions of this form into a non-BIR expressions using BIR's semantic, via its `EVAL` function. However, this approach stops working as soon as the BIR expression contains BIR memories, that are crucial for contract verification with HolBA. As such, this method will not be further mentioned in this document.

operation is semantically equivalent to free variables that *HolSmtLib* already support.

- `BExp_MemEq`: this operation is the equality binary operation on BIR memories. This operation is semantically equivalent to equality between its operands. *HolSmtLib* already supports this operation.
- `BExp_Store`: this operation is used to represent memory writes. It is semantically defined as successive function updates of consecutive segments of the word being stored because the length of the memory value-type can be less or equal than the length of values stored in the memory. A function update in *combinTheory* is defined with Definition 5.13. *HolSmtLib* cannot currently export function update operations.
- `BExp_Load`: this operation is used to read from memories. BIR memories are semantically defined as functions from addresses to values. A function application in *combinTheory* is defined with Definition 5.14. Then, a memory load operation is the concatenation of multiple function application of consecutive addresses. *HolSmtLib* supports function application of uninterpreted functions only.

$$\vdash \forall a\ b.\ a =+ b = (\lambda f\ c.\ \text{if } a = c \text{ then } b \text{ else } f\ c) \tag{5.13}$$

$$\vdash \forall x\ f.\ x :> f = f\ x \tag{5.14}$$

We saw in the previous list that we need to implement the support for *combinTheory* function update and application in the *HolSmtLib* SMT-LIB 2.0 exporter. Since we only need two operations on the memory, load and store, the *ArraysEx* theory is a good fit.

SMT-LIB 2.0 [**barrett_satisfiability_2016**] defines the *ArraysEx* theory using the three following axioms:

Listing 5.5: SMT-LIB 2.0 axioms of the *ArrayEx* theory.

```
(forall ((a (Array s1 s2)) (i s1) (e s2))
  (= (select (store a i e) i) e))

(forall ((a (Array s1 s2)) (i s1) (j s1) (e s2))
  (=> (distinct i j)
      (= (select (store a i e) j) (select a j))))

(forall ((a (Array s1 s2)) (b (Array s1 s2)))
  (=> (forall ((i s1)) (= (select a i) (select b i)))
      (= a b)))
```

If those axioms hold in *combinTheory* then the translation is sound. *combinTheory*'s *UPDATE_APPLY* theorem in Equation 5.15 is equivalent to the first two theorems, the first and the second conjunct corresponding respectively to the first and the second axiom. The third axiom can be proved using both *combinTheory*'s *APP_def* theorem (Theorem 5.14) and *boolTheory*'s *EQ_EXT* theorem (Theorem 5.16)

$$\begin{aligned} \vdash \quad & \forall a\ x\ f.\ (a =\!+\ x)\ f\ a = x \\ \wedge\ & \forall a\ b\ x\ f.\ a \neq b \implies ((a =\!+\ x)\ f\ b = f\ b) \end{aligned} \tag{5.15}$$

$$\vdash \forall f\ g.\ (\forall x.\ f\ x = g\ x) \implies (f = g) \tag{5.16}$$

Since this translation is sound, it has been added in *HolSmtLib*. The translation is direct: from $:>$ and $=\!+$ to respectively `select` and `store`. Section 6.4.2 presents a test using BIR memories.

Chapter 6

Non-proof-producing automatic contract verification

*In the previous section, we learned about contract verification and the current status of **HolBA**'s implementation. To perform verification on the **NIC** model, we would like to automate the process as much as possible. **HolBA** currently offers tools for automatic weakest precondition generation, therefore we need to close the gap between BIR expression and SMT solvers, as well as to implement a convenient interface on top.*

6.1 Exporting BIR expressions to SMT solvers

As an intermediate language for formal verification, **BIR** possesses a precise semantic. The semantic of BIR expressions is expressed as a set of definitions describing what are the equivalent operations using *wordsTheory* and *combinTheory*. These theories contain definitions and theorems about "words", i.e. bounded N-bit integers that are used to reason about integer types in programming languages and hardware memory in general, and function application and update used for BIR memories as already discussed in Section 5.5. For example, the semantic of binary operators in BIR is defined with the following theorems[1]:

$$\begin{aligned} &\vdash bir\_bin\_exp\_GET\_OPER\ BIExp\_And = words\_and \\ &\quad \wedge\ bir\_bin\_exp\_GET\_OPER\ BIExp\_Or = words\_or \end{aligned} \tag{6.1}$$

$$\begin{aligned} &\vdash \forall (bin\_op : bir\_bin\_exp\_t)\ (w1 : word64)\ (w2 : word64). \\ &\quad bir\_bin\_exp\ bin\_op\ (Imm64\ w1)\ (Imm64\ w2) \\ &\quad = Imm64\ (bir\_bin\_exp\_GET\_OPER\ bin\_op\ w1\ w2) \end{aligned} \tag{6.2}$$

[1] Theorems 6.1 and 6.2 have been reduced to only two operators and well-typed 64-bit expressions.

Similarly, a set of definitions and theorems describe the semantic of operations on BIR memories. However, correct handling of endianness, alignment, and genericity over the size of memories cells and addresses, these definitions and theorems are pretty complicated to work with. The same is true for the semantic of operations on BIR variables, because of well-typedness and initialization.

For this reason—i.e. writing **proof-producing** code is costly—, we decided to write a non-proof producing function `bir_exp_to_words` that translates BIR expressions to the equivalent words expression. The obvious downside of such a function is that we now have to trust the translation to be sound because we no longer get any guarantee from the theorem prover. However, the development time is dramatically decreased and offers more time for experimenting. Moreover, this function can later be implemented in a proof-producing way for more trustful verification. Then, in order to have high confidence of correctness, software engineering practices apply:

- write small and understandable pieces of code and compose them, and
- write a comprehensive suite of tests.

BIR expressions are defined as a HOL4 algebraic data type in Listing 6.1. Hence, in order to translate BIR expressions to words expressions, we need to handle every variant. This has been done [2] using an exhaustive `if-then-else` statement[3]. The code is mostly destructuring HOL4 terms and creating new *wordsTheory* and *combinTheory* terms. In order to obtain an easily reviewable code, a balance between expressivity and conciseness has to be carefully decided. Table 6.1 shows that the length of each variant is relatively small in terms of lines of codes, from 1 line for constants to 51 for memory store expressions. This achieves the first point of the previous list.

Testing of `bir_exp_to_words` has been done using a set of $(bir\_exp, expected)$ couples with increasing complexity, where `bir_exp_to_words` is used to translate each bir_exp and the result is compared to $expected$. Then, BIR expression being defined as an algebraic data type, nesting of BIR expressions follow naturally. This achieves the second point of the previous list.

Listing 6.1: bir_exp_t definition.

```
Datatype `bir_exp_t =
    BExp_Const      bir_imm_t
  | BExp_Den        bir_var_t
```

[2] Code available here: https://github.com/kth-step/HolBA/commit/2fcca54dcb04a20716e7697f64b5a4578f8a8af9

[3] Pattern matching would have been optimal but isn't possible because of how HOL4 is embedded in SML.

```
  | BExp_Cast
        bir_cast_t bir_exp_t bir_immtype_t

  | BExp_UnaryExp    bir_unary_exp_t bir_exp_t
  | BExp_BinExp      bir_bin_exp_t bir_exp_t bir_exp_t
  | BExp_BinPred     bir_bin_pred_t bir_exp_t bir_exp_t
  | BExp_MemEq       bir_exp_t bir_exp_t

  | BExp_IfThenElse bir_exp_t bir_exp_t bir_exp_t

  | BExp_Load        bir_exp_t bir_exp_t bir_endian_t
    ↪ bir_immtype_t
  | BExp_Store       bir_exp_t bir_exp_t bir_endian_t
    ↪ bir_exp_t `
```

`bir_exp_t` variant	Lines of code
BExp_Const	1
BExp_Den	21
BExp_Cast	not implemented
BExp_UnaryExp	8
BExp_BinExp	9
BExp_BinPred	10
BExp_MemEq	10
BExp_IfThenElse	9
BExp_Load	48
BExp_Store	51

Table 6.1: Length of each `bir_exp_t` variant in the implementation of `bir_exp_to_words`.

6.2 Pretty-printing to visualize BIR expressions

When working with complex constructs, the need for visualization techniques often arise. Generated weakest preconditions grow quickly with the number of statements in a program, linearly or exponentially depending on the type of statements—control flow statements produce exponential growth. While clever techniques can be implemented to keep their size reasonable [**lindner_trabin:_2019**], we often need to read and analyze them.

Printing of BIR terms in general is very verbose. For example, the expression 6.3 with a 64-bit x integer defined using the BSL code in Listing 6.2 yields the printing in Figure 6.1 using HOL4's default printing capabilities.

Listing 6.2: BSL code

```
bite (
  borl [
    ble ((bden o bvarimm64) "x", bconst64 100),
    bnot (ble (bplus ((bden o bvarimm64) "y",
                      bconst64 1),
                bconst64 10)),
    ble (bplus ((bden o bvarimm64) "x",
                (bden o bvarimm64) "y"),
         bconst64 20)
  ],
  bmult ((bden o bvarimm64) "x", bconst64 2),
  bplus (bmult ((bden o bvarimm64) "x", bconst64 3),
         bconst64 1))
```

$$\text{if } (x \leq 100) \vee (y + 1 > 10) \vee (x + y \leq 20) \text{ then } 2 \times x \text{ else } 3 \times y + 1 \quad (6.3)$$

```
BExp_IfThenElse
  (BExp_BinExp BIExp_Or
     (BExp_BinExp BIExp_Or
        (BExp_BinPred BIExp_LessOrEqual
           (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 100w)))
        (BExp_UnaryExp BIExp_Not
           (BExp_BinPred BIExp_LessOrEqual
              (BExp_BinExp BIExp_Plus (BExp_Den (BVar "y" (BType_Imm Bit64)))
                 (BExp_Const (Imm64 1w))) (BExp_Const (Imm64 10w)))))
     (BExp_BinPred BIExp_LessOrEqual
        (BExp_BinExp BIExp_Plus (BExp_Den (BVar "x" (BType_Imm Bit64)))
           (BExp_Den (BVar "y" (BType_Imm Bit64)))) (BExp_Const (Imm64 20w))))
  (BExp_BinExp BIExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64)))
     (BExp_Const (Imm64 2w)))
  (BExp_BinExp BIExp_Plus
     (BExp_BinExp BIExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64)))
        (BExp_Const (Imm64 3w))) (BExp_Const (Imm64 1w)))
```

Figure 6.1: Default HOL4 printing

This expression is relatively small and yet the printed term is 17 lines long. Compared to the BSL expression that is 8 lines long[4], that is a two-time increase in size. Moreover, lines are long and verbose: for example, a "less-than" binary expression is written as "`BExp_BinPred BIExp_LessOrEqual e1 e2`". Comparatively, the math expression "$e1 \leq e2$" and BSL expression "`ble e1 e2`" are shorter and arguably more readable.

[4] 8 lines correspond to the length in documents where line length is limited to 100 characters, instead

To answer to these kinds of issues, HOL4 provides the ability to implement "pretty-printers", which are custom printing functions for a given type. Four pretty-printers have been implemented to shorten the verbosity of the printed representation and to add colors to the output. Figure 6.2 shows the same expression printed with the pretty-printers enabled.

```
BExp_If
  (BExp_Or
     (BExp_LessOrEqual
        (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 100w)))
     (BExp_Not
        (BExp_LessOrEqual
           (BExp_Plus
              (BExp_Den (BVar "y" (BType_Imm Bit64))) (BExp_Const (Imm64 1w)))
           (BExp_Const (Imm64 10w))))
     (BExp_LessOrEqual
        (BExp_Plus
           (BExp_Den (BVar "x" (BType_Imm Bit64)))
           (BExp_Den (BVar "y" (BType_Imm Bit64))))
        (BExp_Const (Imm64 20w))))
BExp_Then
  (BExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 2w)))
BExp_Else
  (BExp_Plus
     (BExp_Mult
        (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 3w)))
     (BExp_Const (Imm64 1w)))
```

Figure 6.2: Same expression printed with the pretty-printers enabled

The pretty-printers introduce a set of features:

- Simplification of verbose constructs as discussed before (e.g. `BExp_BinExp BIExp_Or` is written as `BExp_Or`).
- Different representation of **if-then-else** statements, simplifying reading the expression when either the condition or the **then** expression are very long.
- Consistent breaking—new lines—of long expressions, because the default printer isn't aware of the structure of printed expressions. In Figure 6.1, we can see inconsistent breaking in addition and multiplication binary operations.
- Highlighting of types, facilitating debugging when the expression isn't well-typed.
- Highlighting of all strings, facilitating reading labels and variable names.
- Gathering of nested binary expressions of the same type on the same level. We

of the 60 in the report. 100 characters are the usual setting for maximum line lengths and correspond more closely to reality.

Listing 6.3: Ideal interface for "prove_contract"

```
fun prove_contract contract_name prog_def
    (precond_lbl, precond_bir_exp)
    postcond_lbl_and_bir_exp_list
```

Listing 6.4: Actual interface for "prove_contract"

```
fun prove_contract contract_name prog_def
    (precond_lbl, precond_bir_exp)
    (postcond_lbl_list, postcond_bir_exp)
```

can see this feature in Figure 6.2 with the two nested **or** binary operators, where the three operands are printed on the same level.

- Rainbow parenthesis, i.e. matching pairs of parenthesis are printed in the same color. This feature is really useful when reading long expressions in order to quickly identify where a sub-expression ends.

6.3 Implementing a convenient interface

In order to perform a high number of proofs on the **NIC** model, we want to hide as much as possible the implementation details of the contract verification procedure. Ideally, we want a function "prove_contract" taking a program fragment, a pre- and a post-condition as parameters, and producing a proof about the Hoare triple if the contract holds or a comprehensive and useful error message if it doesn't. Listing 6.3 shows the ideal interface that we would want, and Listing 6.4 shows the actual interface that have been implemented.

Interface in Listing 6.3 leverages the general idea of how the weakest precondition generation procedure works: it starts from end labels, setting the weakest precondition there to the postcondition, then propagate the weakest precondition of each node of the **CFG** to the previous nodes, and stops when it meets the entry label. Then, it is in theory possible to provide different postconditions to each end label, hence the last parameter being a list of $(end\_label, postcond\_exp)$ pairs. However, the current tool only supports using the same postcondition for the multiple end labels, therefore the interface has been constrained[5].

When implementing this function, high attention has been paid to provide useful and comprehensive feedback in the case of failure. To that end, extensive use of exception

[5] A proposal is being discussed at the time of writing this report about making the weakest precondition generation and the Hoare triple definition more general, and possibly allowing this feature.

Listing 6.5: Equivalent pseudocode of the *cjmp* test.

```
entry:
  x = 1;
  goto (if x=1 then assign_y_100 else assign_y_200)
assign_y_100:
  y = 100;
  goto end
assign_y_200:
  y = 200;
  goto end
end:
```

wrapping has been made in order to give precise context to exceptions, and a logging library has been implemented (cf. Chapter 8).

When using **BSL** to express pre- and post-conditions, this function provides an automatic solution to prove contracts. In the following sections, we will then see usage of this function, first to test it and then to perform proofs on the NIC model.

6.4 Testing the automatic proof procedure

Performing simple proofs is needed in order to test that the proof procedure works. The following examples introduce two of the tests that have been implemented, focusing on the critical parts of each of them. To this end, some liberties have been taken in order to reduce the complexity for the reader. Moreover, even if the test on conditional jumps has been the last one introduced in chronological order [6], it will be presented first because of its relative simplicity.

6.4.1 Conditional jump test

Here we are interested in testing the `prove_contract` function in the presence of a conditional jump with its condition being just an equality test. Feature-wise, this test contains only jump, conditional jump and assignment statements. Listing 6.5 gives a pseudocode representation of this test program, the conditional jump being represented with the "**goto-if-then-else**" construct.

In this test, we want to check that the triple $\{\top\}\ prog\ \{y = 100\}$ holds. Intuitively, this contract means "for every possible initial state S, executing the program will result in a state S' with $y = 100$". It is interesting to note that the precondition $\top$ means

[6] The test on conditional jumps has been introduced in order to fix a bug in the weakest precondition simplification library.

Listing 6.6: Invocation of `prove_contract` for the *cjmp* test.

```
val thm = prove_contract "cjmp"
  cjmp_prog_def
  (* Precondition *) (blabel_str "entry", btrue)
  (* Postcondition *) (
    [blabel_str "end"],
    beq ((bden o bvarimm32) "y", bconst32 100)
  )
```

"for every initial state", analogous to the universal quantifier $\forall$ in logic. This comes from the fact that $\top$ is the weakest precondition possible: $\forall x.\ x \implies \top$. Thus, for this Hoare Triple to hold, the generated weakest precondition must be $\top$. Listing 6.6 shows the invocation of `prove_contract`.

Figure 6.3 shows the auto-generated BIR $P \implies WP$ expression, and Equation 6.4 shows the same expression after translation to a *wordsTheory* expression. This expression can be trivially simplified to $\top$, which SMT solvers can very efficiently do. Hence, this invocation to `prove_contract` succeeds.

```
(BExp_Or
   (BExp_Not (BExp_True))
   (BExp_Not
      (BExp_Equal
         (BExp_Den (BVar "x_wp_0" (BType_Imm Bit32))) (BExp_Const (Imm32 1w))))
   (BExp_And
      (BExp_Or
         (BExp_Not
            (BExp_Equal
               (BExp_Den (BVar "x_wp_0" (BType_Imm Bit32)))
               (BExp_Const (Imm32 1w))))
         (BExp_Equal (BExp_Const (Imm32 100w)) (BExp_Const (Imm32 100w))))
      (BExp_Or
         (BExp_Equal
            (BExp_Den (BVar "x_wp_0" (BType_Imm Bit32)))
            (BExp_Const (Imm32 1w)))
         (BExp_Equal (BExp_Const (Imm32 200w)) (BExp_Const (Imm32 100w))))))
```

Figure 6.3: Auto-generated $P \implies WP$ BIR expression for the *cjmp* test.

$$\top \vee (\neg(x = 1w) \vee ((\neg(x = 1w) \vee 100w = 100w) \wedge (x = 1w \vee 200w = 100w))) \tag{6.4}$$

6.4.2 Load and store test

The **NIC** manipulating a buffer descriptor queue, represented in BIR using memories, we need to ensure that `prove_contract` works with programs containing memories. In this test, we will store a number N in memory at address A, then load a number into x from address **B**. We want to check the following Hoare Triple: $\{A = B\}\ prog\ \{x = N\}$. Listing 6.7 shows the equivalent pseudocode of the test program, Figure 6.4 the auto-generated $P \implies WP$ expression, Figure 6.5 the same expression translated in *wordsTheory* and Listing 6.8 the auto-generated SMT-LIB 2.0 instance featuring `select` and `store` operations.

Listing 6.7: Equivalent pseudocode of the *load and store* test

```
MEM = store(MEM, ADDR1, 42)
x = load(MEM, ADDR2)
```

```
(BExp_Or
   (BExp_Not
      (BExp_Equal
         (BExp_Den (BVar "ADDR1" (BType_Imm Bit32)))
         (BExp_Den (BVar "ADDR2" (BType_Imm Bit32))))))
   (BExp_Not
      (BExp_MemEq (BExp_Den (BVar "MEM_wp_0" (BType_Mem Bit32 Bit8)))
         (BExp_Store (BExp_Den (BVar "MEM" (BType_Mem Bit32 Bit8)))
            (BExp_Den (BVar "ADDR1" (BType_Imm Bit32))) BEnd_BigEndian
            (BExp_Const (Imm16 42w)))))
   (BExp_Equal
      (BExp_Load (BExp_Den (BVar "MEM_wp_0" (BType_Mem Bit32 Bit8)))
         (BExp_Den (BVar "ADDR2" (BType_Imm Bit32))) BEnd_BigEndian Bit16)
      (BExp_Const (Imm16 42w))))
```

Figure 6.4: Auto-generated $P \implies WP$ BIR expression for the *load and store* test.

This expression is harder to prove manually. However, SMT solvers can report very efficiently that the negated expression is unsatisfiable, proving the contract. Therefore, we see that contracts involving BIR memories can be proved, thanks to the work of Section 5.5.

Other preconditions have been tested to verify that `prove_contract` doesn't prove false contracts and succeeds to prove true ones. With the current program, the precondition $load(MEM, B = N) \wedge B = A+2$ can establish the postcondition. The second conjunct is important because N is stored in two consecutive 8-bit memory locations. Interestingly, the precondition $\bot$ works for all contracts, because $\forall x.\ \bot \implies x$, and it works in this particular case.

Listing 6.8: Autogenerated SMT instance for the *load and store* test

```
(set-info :source |Automatically generated from HOL4 by
   ↪ SmtLib.goal_to_SmtLib.
Copyright (c) 2011 Tjark Weber. All rights reserved.|)
(set-info :smt-lib-version 2.0)
(declare-fun v0_ADDR1 () (_ BitVec 32))
(declare-fun v1_ADDR2 () (_ BitVec 32))
(declare-fun v2_MEM_wp_0 ()
  (Array (_ BitVec 32) (_ BitVec 8)))
(declare-fun v3_MEM ()
  (Array (_ BitVec 32) (_ BitVec 8)))
(assert
 (not
  (=
   (bvor
    (bvnot (ite (= v0_ADDR1 v1_ADDR2)
                (_ bv1 1) (_ bv0 1)))
    (bvor
     (bvnot
      (ite
       (= v2_MEM_wp_0
        (store
         (store
          v3_MEM
          (bvadd v0_ADDR1 (_ bv1 32))
          ((_ zero_extend 0) ((_ extract 15 8) (_ bv42
             ↪ 16))))
         (bvadd v0_ADDR1 (_ bv0 32))
         ((_ zero_extend 0) ((_ extract 7 0) (_ bv42 16))
            ↪ )))
       (_ bv1 1)
       (_ bv0 1)))
     (ite
      (=
       (concat
        (select v2_MEM_wp_0 (bvadd v1_ADDR2 (_ bv1 32)))
        (select v2_MEM_wp_0 (bvadd v1_ADDR2 (_ bv0 32))))
       (_ bv42 16))
      (_ bv1 1)
      (_ bv0 1))))
   (_ bv1 1))))
(check-sat)
(exit)
```

```
⊢ ¬(if ADDR1 = ADDR2 then 1w else 0w) ‖
  ¬(if
      MEM_wp_0 =
      MEM |+ (ADDR1 + 1w,(15 >< 8) 42w) |+ (ADDR1 + 0w,(7 >< 0) 42w)
    then
      1w
    else 0w) ‖
  (if MEM_wp_0 ' (ADDR2 + 1w) @@ MEM_wp_0 ' (ADDR2 + 0w) = 42w then 1w
    else 0w) =
  1w
```

Figure 6.5: $P \implies WP$ expression translated in *wordsTheory* for the *load and store* test. `><` is the bitwise extraction operation, `@@` the word concatenation operation, `|+` the memory update operation and `'` the memory load operation. Bit extraction and concatenation is needed because we are working with 16-bit words in a 8-bit memory.

6.5 Simple automatized proofs on the NIC model

Chapter 4 presented the implementation of parts of the NIC model using BIR. The previous sections of this chapter presented the non **proof-producing** contract verification library that has been implemented. This section will present some of the properties that have been proved on the NIC BIR model. However, as we shall see, `prove_contract` has some intrinsic limitations that prevent verifying some contracts.

Section 3.3 presents an overview of the formal HOL4 proof made in [**haglund_formal_2016**] on the formal NIC model. We saw that the properties are phrased in terms of invariants holding in each transition of the model. Such properties can be represented as Hoare Triple and therefore can be proved using this chapter's non-proof-producing contract verification library as long as the pre- and post-conditions can be represented using BIR.

In this section, we will see that the non-proof-producing function `prove_contract` is able to verify some invariants on the NIC BIR model. We will perform the verification of the Hoare Triples shown in Equations 6.5 and 6.6. Those two Hoare Triple respectively represent that, stating from a non-dead initial NIC state, performing one autonomous transition of the transmission automaton doesn't end in an undefined state (i.e. a dead state) and performing one non-autonomous transition of the initialization automaton does end in an undefined state.

$$\{\neg NIC.dead \wedge NIC.tx.state \in \{tx2,\ tx3,\ tx5,\ tx6,\ tx7\}\}$$
$$tx\_automaton\{\neg NIC.dead\} \quad (6.5)$$

$$\{\neg NIC.dead \wedge NIC.init.state \in \{it1,\ it3,\ it4\}\}$$
$$init\_automaton\{NIC.dead\} \quad (6.6)$$

Listing 6.9 shows the SML code used to prove the first contract. Code for the other contract is similar. This example shows that the library implemented in the chapter fulfills its goal of being simple to use and fully automatic when given a Hoare Triple and a program. Combined with BSL and custom helper functions, it becomes easy to express the pre- and post-conditions, and it is still possible to use arbitrary SML functions or HOL4 terms.

Listing 6.9: SML code used to prove contract 6.5.

```
(* TX automaton: NIC doesn't die in autonomous transition *)
val (_, _, p_imp_wp_word_thm) = prove_contract
  "tx_automaton_doesnt_die"
  nic_program_def
  (* Precondition *) (
    blabel_str "tx_entry",
    bandl [
      invariant_nic_not_dead,
      borl (List.map
        (fn s => beq (bdenstate "nic_tx_state",
                      bstateval_tx s))
        (#autonomous_step_list tx_state))
    ])
  (* Postcondition *) (
    [blabel_str "tx_end"],
    invariant_nic_not_dead
  )
val _ = info "Successfully␣proved:␣tx␣automaton␣doesn't␣die"
val _ = if !level_log >= logLib.INFO
  then (pprint_thm p_imp_wp_word_thm)
  else ();
```

6.5.1 Limitations of this approach

While being convenient and working well for simple contracts, the contract verification library implemented in this chapter suffers from two limitations:

- The first one is straightforward: this library isn't proof-producing. Using HOL4 requires a significant learning effort, and easier non-proof producing tools exist. Hence, when using HOL4, one usually wants to reason about proofs, and this library is not proof-producing and could, therefore, introduce subtle bugs or inconsistencies. Moreover, this library doesn't offer the possibility to compose

Hoare Triples because it doesn't output any related theorem. If one wanted to do so, an oracle would have to be used in order to generate such theorems and enabling theorem composition and further reasoning. Nevertheless, as already explained, proof-producing code is more costly. The desired level of trustworthiness should be carefully decided, and this library offers a pragmatic solution.

- Secondly, this library and its approach of contract-based verification is limited by the expressiveness of BIR, since the pre- and post-conditions are BIR terms. BIR doesn't have quantifiers, and it is, therefore, impossible to prove contracts containing existential quantifiers. The existing formal proof on the NIC of [**haglund_formal_2016**] requires existential quantifiers in order to reason about the buffer descriptor queue in the *CPPI_RAM* memory of the NIC, and this part of the proof can therefore not be replaced by using `prove_contract`. Moreover, as explained in the first point of this list, composing verification produced with this library with raw HOL4 proofs is not straightforward. The following Chapter 7 explores a new proof-producing approach for performing contract-based verification on the BIR model that answer to this issue.

Chapter 7

Trustful analysis on the NIC model

*In the previous chapter, we implemented an automated non-**proof-producing** contract verification library, the non-proof-producing part being the translation from BIR expressions to the equivalent wordsTheory and combinTheory expression. Moreover, this verification library can only produce contracts on BIR programs. In order to perform trustworthy verification on the **NIC** model, we need to make proofs directly on the **NIC** state. In this chapter, we will perform a proof on a BIR program and then lift it to the **NIC** model.*

In order to prove the feasibility of this approach, we will prove a simple property. Listing 7.1 contains the NIC state on which we want to prove a property, Equations 7.1, 7.2 and 7.3 present the property that we want to prove, and Listing 7.2 contains a pseudocode representation of the BIR program on which we will perform the verification. Figure 7.1 represents visually the structure of the verification and the steps that we will take during the proof.

Listing 7.1: NIC state used in this proof

```
Datatype `nic_state = <|
  dead : bool;
  x : word32
|>`
```

$$\vdash \forall nic.\ P_{NIC}\ nic \stackrel{def}{=} \neg nic.dead \wedge nic.x = 0w \tag{7.1}$$

$$\vdash \forall nic\ nic'.\ Q_{NIC}\ nic\ nic' \stackrel{def}{=} \neg nic'.dead \wedge nic'.x = nic.x + 1w \tag{7.2}$$

$$\vdash \forall nic\ nic'.\ P_{NIC}\ nic\ \wedge\ exec\_prog\ nic\ bir\_prog\ nic' \implies Q_{NIC}\ nic\ nic' \tag{7.3}$$

Listing 7.2: Pseudocode of the program used in this proof

```
nic.x := nic.x + 1
if nic.x > 10:
```

```
    nic.dead := true
```

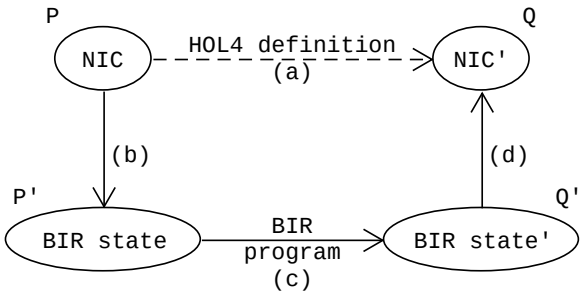

Figure 7.1: Visual structure of the proof. References like (a) to the arrows of this Figure are used throughout the proof to refer to a particular step.

Remark 1. *Q_{NIC} nic nic' is defined on both initial and final states, in order to be able to reason about the initial state in the postcondition. This allows us to write $nic'.x = nic.x + 1w$ instead of $nic'.x = 1w$ for example.*

Equation 7.3 uses a relation $exec\_prog$ that we shall define now. As we want to make a proof on an undefined HOL4 definition (a), we must establish an equivalence between the HOL4 definition (a) and the BIR program (c). In real proofs, this can either be produced by a lifter which generates the BIR program from a given input program and gives a "certificate", i.e. a theorem stating the equivalence, or be a definition which would then mean that we trust that the BIR program is equivalent to the HOL4 definition. In this proof, we will use a definition. This definition shall state that $exec\_prog$ nic bir_prog nic' (a) is equivalent to executing the BIR program from a state bir_state to a state bir_state' (c), where nic is somehow equivalent to bir_state (b) and nic' somehow equivalent to bir_state' (d). To express an equivalence between HOL4 states ("NIC") and BIR states, we introduce a relation R. The relation R nic bir_state is defined as a simple mapping between the BIR state and the NIC state, as shown in Listing 7.3. Then, we define the relation $exec\_prog$ as shown in Equation 7.4[1].

Listing 7.3: Definition of the relation R

```
val R_def = Define `
  R (nic: nic_state) (bir_state: bir_state_t) <=>
      (bir_env_lookup "nic_dead" bir_state.bst_environ
         = SOME (BType_Bool,
                 SOME (BVal_Imm (Imm1 nic.dead))))
```

[1]Definition 7.4 has been annotated to visualize how it is connected to the structure of the proof on Figure 7.1. In addition, the BIR_exec relation is used as a shorthand for $bir\_exec\_to\_labels$ in order to simplify the proof.

```
    /\ (bir_env_lookup "nic_x" bir_state.bst_environ
          = SOME (BType_Imm Bit32,
                  SOME (BVal_Imm (Imm32 nic.x))))`
```

$$
\begin{aligned}
&\vdash \forall nic\ nic'.\ exec\_prog\ nic\ bir\_prog\ nic' && \text{(a)}\\
&\quad \stackrel{def}{=} \forall bir\_state\ bir\_state'.\\
&\qquad (R\ nic\ bir\_state && \text{(b)}\\
&\qquad \wedge\ bir\_state' = BIR\_exec\ prog\ bir\_state) && \text{(c)}\\
&\qquad \Longrightarrow\ R\ nic'\ bir\_state' && \text{(d)}
\end{aligned}
\tag{7.4}
$$

Proof of Equation 7.3. In order to begin the proof, as the goal 7.3 is defined over nic states, we need a theorem about the injectivity of the relation R, stating that for all nic exists a bir_state such that $R\ nic\ bir\_state$ (b). Additionally, the BIR_exec relation will also need some properties on bir_state, that we shall add in this injectivity theorem now.

Theorem 7.0.1. *Injectivity theorem of R*

$$
\begin{aligned}
\vdash\ & \forall nic.\ \exists bir\_state.\\
& \quad R\ nic\ bir\_state\\
& \wedge\ bir\_state.bst\_pc.bpc\_index = 0\\
& \wedge\ bir\_state.bst\_pc.bpc\_label = entry\_label\\
& \wedge\ bir\_state.bst\_status = BST\_Running
\end{aligned}
$$

Proof. After rewriting the relation R, we prove theorem 7.0.1 by exhibiting a satisfying bir_state. □

In possession of a bir_state in relation with a nic, we now need to lift the precondition $P_{NIC}nic$ on this bir_state. First, we need to introduce equivalent pre- —and post-—conditions on the BIR states, then we shall prove that the precondition lifts.

Listing 7.4: Equivalent pre- and postconditions on BIR states

```
val BIR_P_exp_def = Define `BIR_P_exp = ^(bandl [
  beq ((bden o bvarimm1) "nic_dead", bfalse),
  beq ((bden o bvarimm32) "nic_x", bconst32 0)
])`
val BIR_Q_exp_def = Define `BIR_Q_exp = ^(bandl [
  beq ((bden o bvarimm1) "nic_dead", bfalse),
  beq ((bden o bvarimm32) "nic_x", bconst32 1)
])`
val BIR_P_def = Define `BIR_P bstate =
  bir_eval_bool_exp BIR_P_exp bstate.bst_environ`
```

```
val BIR_Q_def = Define `BIR_Q bstate =
  bir_eval_bool_exp BIR_Q_exp bstate.bst_environ`
```

Limitation Q_{BIR} is a function of the end state only. Hence, in order to reason about the initial state, we need in general to introduce ghost variables postcondition. In this proof, since the contract that we are proving is simple, using the actual value of $nic.x$ is enough. However, this may pose a problem if we want to generalize the proof.

Notation $P_{BIR}\, bir\_state$ and $Q_{BIR}\, bir\_state$ are defined using $bir\_eval\_bool\_exp$, which evaluates respectively the expressions P_{BIR}^{exp} and Q_{BIR}^{exp} in a given BIR state. In order to simplify the proof, let's define a new operator $\overset{eval}{=}$ that is used to evaluate given variables, e.g. $bir\_state.x \overset{eval}{=} 0w$.

Theorem 7.0.2. *Lifting of $P_{NIC}\ nic$ to bir_state*

$$\vdash\ \forall bir\_state\ (\exists nic.\ R\ nic\ bir\_state\ \wedge\ P_{NIC}\ nic)\ \Longrightarrow\ P_{BIR}\ bir\_state$$

Proof. Let's do this proof in a backward way. By discharging the antecedent of the implication and using the existential elimination inference rule, we get as assumptions $P_{NIC}\ nic$ and $R\ nic\ bir\_state$. From this, we can deduce that $bir\_state.x \overset{eval}{=} 1w$ and $bir\_state.dead \overset{eval}{=} \bot$. Then, we can substitute those values in the goal, which proves it. □

Assuming that we have a Hoare Triple theorem between initial and final BIR states, we can use Definition 7.4 in order to establish that $R\ nic'\ bir\_state'$. Then, in order to prove $Q_{NIC}\ nic$ (d), we have to transfer the postcondition back from bir_state to nic.

Theorem 7.0.3. *Lowering $Q_{BIR}\ bir\_state$ to nic.*

$$\begin{aligned}\vdash\ \forall\, bir\_state'.\ Q_{BIR}\ bir\_state'\ \Longrightarrow\\ (\forall nic\ nic'\ bir\_state.\ P_{BIR}\ bir\_state\\ \wedge\ R\ nic\ bir\_state\ \wedge\ R\ nic'\ bir\_state'\\ \Longrightarrow\ Q_{NIC}\ nic\ nic')\end{aligned}$$

We introduce bir_state and P_{BIR} in this theorem for the reason explained in Remark 1, i.e. reason about both the initial and final state in the postcondition.

Proof. This proof has been done in HOL4. The reasoning is quite similar to the proof of Theorem 7.0.2, as the backward proof mainly involves rewriting and simplification. We will omit this proof here and redirect the reader to the HOL4 proof available in our source repository [**lacroix_trustful_2019**]. □

We will now prove that the Hoare Triple holds on the BIR program.

Theorem 7.0.4. $\{P_{BIR}^{exp}\}\ bir\_prog\ \{Q_{BIR}^{exp}\}$

Proof. To prove this Hoare Triple, we used the proof-producing procedure implemented in **HolBA** in order to generate the weakest precondition. The automatically derived weakest precondition is shown in Figure 7.2. Section 5.4 already discussed how to perform this proof: we have to prove Equation 5.12 with $\mathbf{wp}$ being the expression in Figure 7.2 and $\mathbf{pre}$ being P_{BIR}^{exp}. Because we want to use a **SMT** solver, we need to turn the goal of the backward proof into a *wordsTheory* expression. *combinTheory* isn't needed in this case since BIR memories are not used. Equation 5.12 uses bir_eval_exp which evaluates an expression in the given BIR state. Therefore, to translate the goal into a *wordsTheory* expression, we need to use BIR's semantic. The semantic needs well-typedness and initialization of the variables. At the time of writing, HolBA offers no support for automatic rewriting with the semantic definitions, so multiple lemmas about initialization, well-typedness, and type equality must be manually proved for every variable. Those are not shown here because they consist of simple rewriting and simplification.

Then, the proof consist of consecutively rewriting following the definition of bir_eval_exp and the definition it uses until the goal only contains

$$bir\_env\_read\ (BVar\ \text{“}nic\_x\text{”}\ (BType\_Imm\ Bit32))\ bir\_state.bst\_environ$$

and similarly for $nic.dead$. Let's call those expressions x_val and $dead\_val$ respectively. For expressions, BIR semantic is defined over immutable and constant values. Therefore, we need to establish an equivalence between the values that we currently have.

Lemma 7.0.5.

$$\exists x\_imm.\ x\_val = BVal\_Imm\ x\_imm$$

Proof. Assuming well-typedness and initialization, this theorem immediately results from the BIR semantic. □

Lemma 7.0.6.

$$\exists x\_word.\ x\_imm = Imm32\ x\_word$$

Proof. This lemma is part of the BIR semantic, as one of the six conjuncts of the `bir_imm_t_nchotomy` theorem, which establishes this existence theorem for every BIR immutable types. □

```
(BExp_And
   (BExp_Or
      (BExp_Not
         (BExp_LessThan
            (BExp_Const (Imm32 10w))
            (BExp_Plus
               (BExp_Den (BVar "nic_x" (BType_Imm Bit32)))
               (BExp_Const (Imm32 1w)))))
      (BExp_And
         (BExp_Equal (BExp_True) (BExp_False))
         (BExp_Equal
            (BExp_Plus
               (BExp_Den (BVar "nic_x" (BType_Imm Bit32)))
               (BExp_Const (Imm32 1w)))
            (BExp_Const (Imm32 1w)))))
   (BExp_Or
      (BExp_LessThan
         (BExp_Const (Imm32 10w))
         (BExp_Plus
            (BExp_Den (BVar "nic_x" (BType_Imm Bit32)))
            (BExp_Const (Imm32 1w))))
      (BExp_And
         (BExp_Equal
            (BExp_Den (BVar "nic_dead" (BType_Imm Bit1)))
            (BExp_False))
         (BExp_Equal
            (BExp_Plus
               (BExp_Den (BVar "nic_x" (BType_Imm Bit32)))
               (BExp_Const (Imm32 1w)))
            (BExp_Const (Imm32 1w)))))))
```

Figure 7.2: Autogenerated weakest precondition for proof of Theorem 7.0.4.

Now, using Lemma 7.0.5, we are able to substitute all occurrences of x_val into $BVal\_Imm\ (Imm32\ x\_word)$, and similarly for $dead\_val$. Finally, rewriting the goal using the full set of BIR semantic theorems and some rewriting rules, the goal reduces to an expression free of BIR terms:

$$\begin{aligned}&(dead\_w = 0w) \wedge (x\_w = 0w) \implies \\ &\Big(\neg(10w <_+ x\_w + 1w) \vee \big((1w = 0w) \wedge (x\_w + 1w = 1w)\big)\Big) \wedge \\ &\Big((10w <_+ x\_w + 1w) \vee \big((dead\_w = 0w) \wedge (x\_w + 1w = 1w)\big)\Big)\end{aligned} \tag{7.0.6.1}$$

An **SMT** solver is able to prove this goal. Interestingly, HOL4 simplification procedures are also able to prove it. However, they won't be able to prove it for more complicated ones or will be less effective than SMT solvers. □

Finally, using the deduction rule with Theorems 7.0.1, 7.0.2, 7.3, 7.0.4, 7.4 and 7.0.3, in that order, concludes this proof. ■

Chapter 8

User experience and good practices

This chapter will present the care that has been put into user experience and good practices throughout this work.

In this work, we took the role of a user of the HolBA platform, and of HOL4 by necessity. HOL4 learning curve is quite steep, and very few efforts have been invested towards beginner friendliness because this topic can become really time-consuming and its perceived importance greatly depends on one's experience. Throughout our design and implementation process, great care has been put into having a good user experience while keeping time spent on this issue reasonably low. Section 8.1 will briefly present what has been done in order to increase user experience.

Additionally, binary analysis platforms are complex software systems, and should as such follow well-known best practices in order to make it last and develop serenely. Therefore, Section 8.2 presents some best practices that have been adopted in HolBA.

8.1 Towards better user experience

8.1.1 BSL and the pretty-printer

Already presented in details in Sections 4.2.4 and 6.2 respectively, BSL and BIR pretty-printers are two successful attempts to simplify HolBA users manipulation of BIR, respectively writing and reading BIR code.

BSL is, as the name suggests, a really simple library that enables less verbose writing of BIR terms. There was no strict need for this library, since HOL4 features a powerful quoting system that can already parse raw BIR code. However, the raw BIR syntax is (a) cumbersome to use and arguably becomes hard to maintain as the code grows

in size and (b) isn't easy to mix with SML code. BSL offers a convenient solution to both these problems, and the time spent in order to implement it has been very limited, roughly about two days.

Similarly, BIR pretty-printers were not a hard need, but rather a welcomed feature. When debugging large BIR expressions, we spent a non-negligible amount of time counting parenthesis, grouping operators and searching arms of binary operators. The pretty-printers give a convenient solution to these problems, and required a limited implementation effort, roughly about two days spent in iterative improvements.

8.1.2 Error handling in SML and HOL4

Errors happen. This is a truth in engineering and software engineering is no exception to the rule. However, when handled successfully, errors can increase the reliability and usefulness of a system, because they can force invariants and provide insightful information.

SML offers error handling via exceptions that can be raised and handled, raised on errors and handled at places where recovery is possible and desirable. Being deeply embedded in SML, HOL4 inherit this behavior. However, being a complex system on its own, HOL4 can raise exceptions at a lot of places. When used without attention, HOL4 can raise exceptions at unexpected and often distant (i.e. in dependencies of the functions that are directly used) locations. Hence, when some exceptions are raised, meaningless error messages are raised and dumped at the user. For example, a user incorrectly using some function could receive an exception saying "*Not a conjunction*", whereas the function interface doesn't directly take logical expressions as arguments. Here, the implementation is leaking to the user that should not have to know anything about it.

To answer to this issue, there are two immediate possibilities (in addition to fixing the actual error): (a) code could handle possible exceptions of code that it uses and that could produce some, and either handle the issue (if possible) or raising another more meaningful error, and (b) HOL4 provides an exception wrapping mechanism in its *Feedback* library that offers the possibility of adding information to an exception on the flight, producing a result analogous to backtrace in other programming languages.

8.1.3 LogLib, a logging library

A logging library, called *logLib*, has been implemented. HOL4 provides a tracing system which allows registering *traces*, i.e. references to an integer representing the verbosity level of some system that can be changed at runtime in order to adjust the

```
[TRACE @ nic_helpersLib::prove_p_imp_wp] smt_ready_tm:
¬(if (nic_dead = 0w) ∧ (nic_init_state = 2w) then 1w else 0w) ‖
(¬(if nic_init_state = 1w then 1w else 0w) ‖ if 1w = 0w then 1w else 0w) &&
((if nic_init_state = 1w then 1w else 0w) ‖
 (¬(if nic_init_state = 2w then 1w else 0w) ‖ if nic_dead = 0w then 1w else 0w) &&
 ((if nic_init_state = 2w then 1w else 0w) ‖
  (¬(if nic_init_state = 3w then 1w else 0w) ‖ if 1w = 0w then 1w else 0w) &&
  ((if nic_init_state = 3w then 1w else 0w) ‖
   (¬(if nic_init_state = 4w then 1w else 0w) ‖ if 1w = 0w then 1w else 0w) &&
   if (nic_init_state = 4w) ∨ (1w = 0w) then 1w else 0w))) =
1w
Handled exception: [init_automaton_doesnt_die] Z3_ORACLE_PROVE failed
HOL_ERR:
 - Structure: nic_helpersLib
 - Function: prove_p_imp_wp
 - Message: at Z3.Z3_SMT_Oracle:
Z3 not configured: set the HOL4_Z3_EXECUTABLE environment variable to point to the Z3 executable file.

[DEBUG @ nic_helpersLib::prove_p_imp_wp] Asking Z3 for a SAT model...
[DEBUG @ nic_helpersLib::prove_p_imp_wp] Failed to compute a SAT model. Ignoring.
error in quse /NOBACKUP/tholac/holba-reborn/examples/nic/test-early-wp.sml : HOL_ERR {message = "at Z3.Z3_SMT
_Oracle:\nZ3 not configured: set the HOL4_Z3_EXECUTABLE environment variable to point to the Z3 executable fi
le.", origin_function = "prove_p_imp_wp", origin_structure = "nic_helpersLib"}
error in load test-early-wp : HOL_ERR {message = "at Z3.Z3_SMT_Oracle:\nZ3 not configured: set the HOL4_Z3_EX
ECUTABLE environment variable to point to the Z3 executable file.", origin_function = "prove_p_imp_wp", origi
n_structure = "nic_helpersLib"}
Uncaught exception: HOL_ERR {message = "at Z3.Z3_SMT_Oracle:\nZ3 not configured: set the HOL4_Z3_EXECUTABLE e
nvironment variable to point to the Z3 executable file.", origin_function = "prove_p_imp_wp", origin_structur
e = "nic_helpersLib"}
```

Figure 8.1: Extract of a script output, featuring *trace* and *debug* information from *logLib* (Section 8.1.3), and an exception printed using the exception pretty-printer (Section 8.1.4). The original exception can be seen in white at the bottom of the image.

desired level of debug information. However, HOL4 only provides the system for registering traces and some logging function, but it does not provide conditional logging functions on the current trace level.

logLib offers a convenient way to tracing: when given a trace variable, it returns four functions conditional on the value of the trace variable. The functions are named, in increasing verbosity level, *trace*, *debug*, *info*, *warn* and *error*. Additionally, each function prints the given message with a specific header. This header is coloured depending on the log level, making reading long logs easier. Furthermore, each log line contains location information, facilitating the process of identifying where some error or warning happened. Figure 8.1 shows an execution output with message from the *trace* and *debug* functions.

8.1.4 SML exception pretty-printer

As can be seen in Figure 8.1, default printing of HOL4 exceptions is not really easy to decipher. The is due to the fact that HOL4 has been designed as an **interactive** theorem prover to be used with *Emacs*, which provides some layout and pretty-printing features. Therefore, a small pretty-printing library for SML exceptions has been implemented, called *pretty_exnLib*. It provides only two analogous functions that, given

Listing 8.1: Signature of the *pretty_exnLib* library.

```
signature pretty_exnLib =
sig
  val pp_exn: exn -> exn
  val pp_exn_s: string -> exn -> exn
end
```

Listing 8.2: Typical usage of the *pretty_exnLib* library.

```
(* some code  that can raise ... *)
handle e => raise pp_exn e;

(* or *)
handle e => raise pp_exn "Failed␣to␣perform␣A,␣check␣that␣B." e;
```

an exception as parameter, pretty-prints the exception and returns it unchanged. Listing 8.1 presents the signature of the library, and Listing 8.2 shows how it is used.

8.2 Best practices for a complex platform

When building complex systems, special care should be taken in order to ensure that it grows in good conditions. Despite providing the best capabilities, a system with poor usability will encounter a slow adoption, as shown in the previous section. But when a system gets used by an increasing number of users, a different working organization is needed. This section will first briefly a critical concept that should be mastered for the successful growth of a software platform. Then, we will study the **Continuous Integration (CI)** system that has been set up for HolBA.

8.2.1 Developing simple interfaces

In 1995, the Gang of Four published a book about design patterns for software design [**gamma_design_1995**]. “Program to an interface, not an implementation.” is a key concept presented among the patterns. Since then, this sentence is considered as one of the most important design principles in software development.

This section doesn’t intend to give a thorough explanation of the concept. However, some key insights shall be provided:

- An interface is a contract between a system (such as a function, a library, a platform) and its user. This contract states the behavior and guarantees that this system offers via its public components. It defines the interaction between

the component, the system, and the user. But most importantly, the contract leaves out the implementation. An interface builds an abstraction over the implementation. Using explicit contracts is particularly useful when performing contract-based verification, as it can simplify reasoning about components and allow building a more resilient system.

- Libraries provide features, which sometimes have a very complex implementation. This is particularly true for trustworthy binary code analysis. Building interfaces helps to abstract away complex implementations and helps to compose systems together.
- Programming to an interface instead of an implementation makes the code more resilient to changes. Indeed, if an implementation must be changed in order to provide more features, fix critical bugs or deliver more performance, codes that are not aware of the implementation will not need to be modified.

8.2.2 Continuous Integration: tests and static analysis

Continuous Integration (**CI**) is a development practice where developers of a project *integrate* frequently their code and changes into a single shared source repository, hence the name of this practice. Each integration of any change into the shared repository should be automatically verified by automated builds and a full set of automated tests. This allows to immediately see if changes to the source repository break a feature or introduce a regression, thus allowing to fix it immediately. Since this practice imposes to run a whole battery of tests for each and every change, even the smallest one, automation is key. [**martin_fowler_continuous_nodate**] gives a detailed introduction to this practice.

A CI system has been added to the HolBA repository, and its practices are being adopted by the research team. For each change, the CI system recompiles the whole platform from scratch and runs all the tests and executes a set of example scripts. Additionally, the CI system also performs some static analysis of the source code in order to give developers some key insight of the state of the code. Two static analyzers have been implemented, showing respectively the *cheats* used in the formal proofs and the "*TODO*" comments indicating left work to be done [1].

[1] At the time of writing, this feature has not been added yet to the CI system because of time constraints. However, it is ready to be integrated.

Chapter 9

Conclusions

9.1 Discussion

The goal of this paper was to perform experiments about the automation of formal verification of devices at the binary level. The NIC model of [**haglund_formal_2016**] has served as a central theme throughout the work. Work in this thesis has been divided in practice into three parts, in addition to the learning process due to the very steep learning curve of HOL4:

1. **Implementation of the NIC BIR model**: the formal model of the NIC of [**haglund_formal_2016**] has been partially implemented as a BIR program, and BSL has been added to the HolBA platform in order to make this task more convenient.

2. **Non-proof-producing automatic contract verification library**: a user-friendly (adjective rarely used to describe HOL4), convenient and automatized library for contract verification have been implemented. This library brings HolBA one step closer to "one-button solutions" for performing software verification. It features a non-proof-producing translation of BIR expressions to expressions that can be exported to SMT solvers, and an extension of HOL4's SMT library has been realized in order to add reasoning about BIR memories.

3. **Formal proof of a BIR program on a HOL4 state**: a novel usage of BIR and method for transferring properties from BIR program to formal HOL4 specification has been carried out. Additionally, while not automated, the work needed in order to implement automatic HOL4 tactics has been clearly identified. Every automation work should be preceded by a manual implementation of the task, that this proof intends to be.

Throughout this work, a strong focus has been placed on ease of use and user-friendliness

with BSL, the pretty-printer, the implementation of new logging utilities, guidelines about error handling in SML and with HOL4, and a friendly exception printer. I strongly believe that user experience of such complex platforms are of primary importance because it can dramatically reduce time spent in debugging and the need of exhaustive documentation, and give to users to desire to keep using the platform. If it is decided to put more work in HolBA in order to make it a powerful binary analysis platform, the research team is advised to consider this aspect while expanding HolBA capabilities.

Some failed experiments have also been realized, including the DepGraph tool that revealed to not be adapted to its planned usage or which would have needed to much work for too few benefits (though DepGraph's design is future-proof and the tool can easily be extended later if new needs or novel ideas appear), and the former two implementation attempts of the NIC model using flowcharts or C which have been found to not be suitable for device models.

9.2 Future work

The topic of software verification, while being as old as Computer Science, still have a lot of work to be done and paths to be explored. This thesis opens some paths that should be considered by someone interested in the topic.

The pretty-printer implemented in this work is only a small experiment only scratching the surface of what can be done in HOL4. Further work in this domain should follow the concrete needs that arise with extended usage of BIR. Current limitations and possible further work of the pretty-printer presented in this document include:

- the generated output isn't parsable, which is against HOL4 conventions where every printed term should be parsable by HOL4 quoting system. A solution to this problem would be to implement a theory introducing definitions for the new abbreviations. However, some liberties taken in the pretty-printer cannot be parsed: (a) the flat printing of nested binary expressions of the same state, because they would need multiple arity functions which are not available in HOL4, and (b) **if-then-else** statements in the current representation.
- the pretty-printer doesn't use infix operators, feature widely used in custom syntax used in some theories, like *wordsTheory*. Infix syntax for binary operators can dramatically reduce the verbosity of expressions, and give a more familiar representation that *could* be simpler to work with.
- the choice of colors could be reviewed in order to more closely follow HOL4 convention and usage. For example, BIR types are highlighted in the same color

than free variables. While creating a very limited chance of ambiguities, users familiar with HOL4 conventions can be annoyed with this inconsistency.

The non-proof-producing contract verification library implemented in this work can be really useful for prototyping, but it cannot be used for trustful verification. The limitations it suffers could be reasonably fast to overcome, and they should be fixed especially if HolBA were to become a strong binary analysis platform. Limitations include (a) the impossibility to compose contract in order to reason about longer functions, to compose functions or to prove properties on not loop-free programs, and (b) the obvious limitation of not being proof-producing. Work is ongoing at the time of writing about weak-correctness and contract composition, answering to (a). The main blocker of (b) is the absence of a proof-producing ways of using SMT solvers for proving contracts (Equation 5.12). The proof of Chapter 7 and its SML implementation identify what automatic procedures should be added to HolBA in order to perform step (b) and progress towards proof-producing automatic verification. Additionally, the signature of `prove_contract` should be reworked in order to enable mixing it with proof-producing code, after both (a) and (b) are solved. For example, it should work with HOL4 definitions instead of raw terms.

Appendix A

DepGraph's design

*This appendix will briefly present the design of DepGraph, the tool introduced in Chapter 3. While this experiment didn't reveal to be successful in order to get insights of the formal NIC model of [***haglund_formal_2016***], the tool still presents a powerful architecture and can be useful in order to generate documentation of* ***SML*** *modules.*

DepGraph is a Python software that has been designed with modularity and extensibility in mind. The architecture is centered around a common data structure, the dependency graph, that is created by *frontends*, mutated by *middlewares* and exported by *backends*. Figure shows DepGraph's class diagram. Adding new frontends, middlewares or backends is as simple as adding one independent file and declare it the corresponding *__init__.py*, and doesn't require to change any other code. Therefore, DepGraph allows easy creation of pipelines, that are a succession of one frontend, multiple middlewares and ended with one backend, and it can easily by extended to any need.

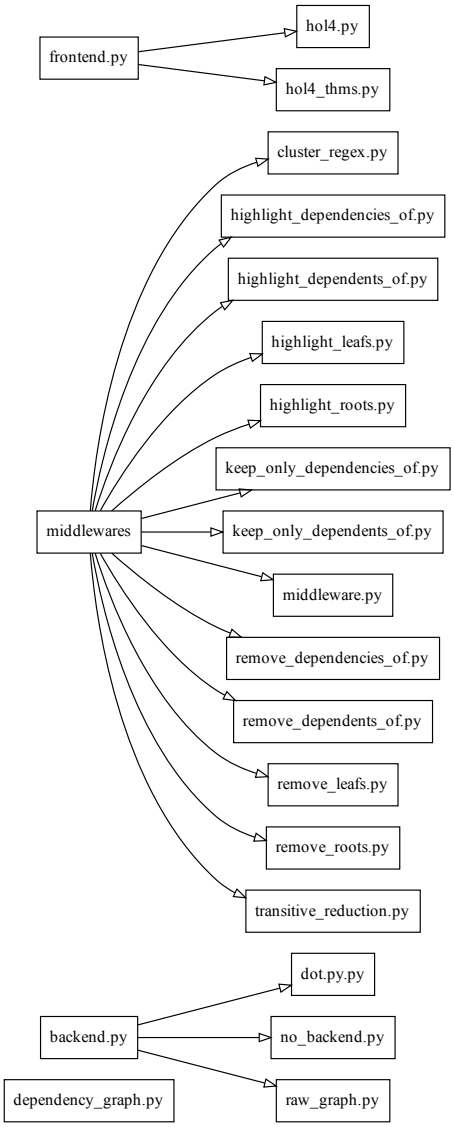

Figure A.1: DepGraph's class diagram, showing frontends, middlewares, and backends. The common intermediate representation, the dependency diagram, is implemented in *dependency_graph.py*.