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

Thomas Lacroix — thomas.lacroix@insa-lyon.fr

Département Informatique
INSA de Lyon

2018/2019

Sous la responsabilité de :
Mads Dam : Division of Theoretical Computer Science – KTH
Pierre-Édouard Portier : Département Informatique – INSA Lyon

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 proof. 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 with 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

Résumé

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

1 Introduction

This section 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 PROSPER project [1] 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.

Previous work in the PROSPER project achieved to formally verify a simple separation kernel [2, 3], which later resulted 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 [4]. However, hardware devices were not included in the verification, so devices, like Network Interface Controller (NIC), cannot be used by the virtualized applications reducing the value of the hypervisor. In order to solve this issue, verification of hardware devices is being carried out.

A formal model of a NIC device has already been produced, on which some security theorems have been proved [5]. 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. However, there exist more devices that are of interest to verify, so it is very desirable to automate formal verification of devices and use a standard model for reasoning about them.

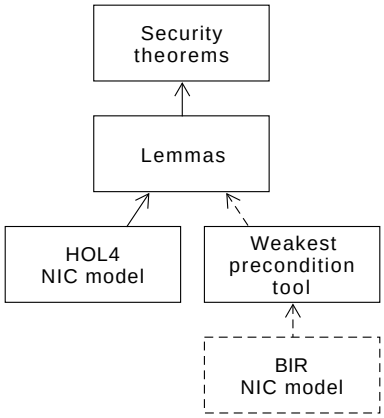

Figure 1: Formal v. BIR NIC models. The left hand side already exists. The dashed elements represent the work done during this project.

The team is now developing a new framework for performing binary analysis in HOL4, an interactive theorem prover, named HolBA [6]. This framework features a machine-independent Binary Intermediate Representation (BIR), a proof-producing transpiler from ARMv8 and Cortex-M0 assembly code to BIR, called the lifter, a proof-producing weakest precondition generator for loop-free programs, and supporting tools [7, 8].

The idea of this work is to translate the formal NIC model of [5] 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 security properties are implied. Figure 1 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 Thesis objective

The 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 [5] is used as working example.

1.3 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.

1.4 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.

1.5 Related work

1.5.1 Secure execution platforms

The PROSPER project isn't the only project focused on high-security execution platforms. Platforms such that seL4, Microsoft Hyper-V, INTEGRITY Multivisor or even MINIX 3 are examples of platforms 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 [9]. 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 [10].

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 [11]. However, they don't appear to include 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 [12]. 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.

1.5.2 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 their use of 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].

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

[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/

Valgrind is a framework for building program supervision tools, such as memory checkers, cache profilers or data-race detectors [13]. As its core, Valgrind is a JIT x86-to-x86 compiler, translating binary programs into its IR 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 [14]. 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 [15]. It leverages BAP, the Binary Analysis Platform from Carnegie Mellon University (CMU BAP) [16], 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 Angr. An interesting note though is that BIR's design is based upon CMU BAP's intermediate language.

The novelty introduced in HolBA, however, 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.

2 Definitions and relevant theories

This section intends to provide brief introductions for concepts and the foundation of theories that are essential in order to understand the problem that this project aims to explore.

2.1 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, or integration with external tools like SMT solvers). Coq, HOL4 or Isabelle are such tools.

HOL4 [17] stands for Higher-Order Logic. It is a programming environment deeply embedded into the SML programming language enabling to prove theorems and write proof-producing programs. HOL4 uses a very small kernel in order to provide very high guarantees of correctness.

2.2 HolBA's BIR

HolBA's Binary Intermediate Representation (BIR) [8], 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 and by forbidding implicit side-effects. A statement can only have explicit state changes and can only affect one variable. Valid BIR programs must be well-typed.

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 data types, and possesses a completely defined semantic. Section 5.2 contains a more thorough discussion of the BIR semantic. Section 4.3 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.

2.3 Hoare Triples

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 1). Hereafter, we assume programs and states to be well-typed. A Hoare Triple is

also called *a contract*.

$$\{P\}\ prog\ \{Q\} \stackrel{def}{=} S' = exec(S, prog) \wedge P(S) \implies Q(S') \quad (1)$$

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.

2.4 Weakest preconditions

While Hoare logic introduces sufficient preconditions, Dijkstra introduced the concept of necessary and sufficient preconditions, called "weakest" preconditions [18]. Such weakest preconditions (WP) can be automatically derived from a program *prog* and a postcondition Q. Let's call $WP(prog, Q)$ such a WP. Then, from Equation 1 follows:

$$\forall (prog, Q), \{\, WP(prog, Q)\}\ prog\ \{Q\} \quad (2)$$

For the program $n := n+1$ mentioned in the previous section, the WP for the postcondition $even(n)$ is $odd(n)$, i.e. incrementing the value of an odd integer variable by one makes it even.

For a triple $\{P\}\ prog\ \{Q\}$ to hold, P must be stronger than the WP, i.e. we need to prove that $P \implies WP(prog, Q)$. While multiple methods exist to perform such proofs, Satisfiability Modulo Theory (SMT) solvers offer a convenient and automatic solution. With an SMT solver, proving R consist in checking that $\neg R$, its negation, is *unsatisfiable*. SMT solvers can also give counter-example R is false.

HolBA provides a proof-producing tool for automatically deriving WP on loop-free BIR programs whose control flow can be statically identified [8]. However, SMT solvers had never been used before this work.

3 Overview of the formal proof of the NIC model

This section will present the formal verification of [5] and present an attempt that have been made about visualization of the proofs.

The formal verification of a NIC in [5] represents the NIC as a transition system. composed of five finite state automata, each responsible for a different task: initialization, transmission, transmission teardown, reception and reception teardown. These automata have autonomous transitions that represent the standalone operation of the device. Communication with the CPU is represented with non-autonomous transitions. The model also contains a scheduler. Since this model has been realized using the public specification of the device, which is underspecified, the simulated state of the device is marked *dead* if the model is asked to describe any transition or operation that is not described by the specification. Being designed as a transition system, the whole model is loop free, which is convenient for contract-based verification, since that hard composition theorems would otherwise be needed.

The state of the NIC is defined as a nested data type containing registers and a memory called *CPPI_RAM*.

The low-level lemmas of the verification (cf. Figure 1) are stated as Hoare Triples using invariants (Equation 3) that must hold for every possible transition:

$$I_{NIC} \stackrel{def}{=} \neg dead \wedge I_{init} \wedge I_{tx} \wedge I_{rx} \wedge \cdots \quad (3)$$

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 [19].

The proof being consequent and divided into several script files, it is difficult to identify what are the low-level lemmas to be reproved in this work. Therefore, a tool, called DepGraph, has been implemented in order to extract the dependency structure from proof files in the form of a graph. Then, the fringe of the graph represents the smallest set of lemmas that is enough to prove in order to imply the security properties by using the rest of the proofs unchanged.

DepGraph features two frontends that can extract dependencies between HOL4 theories (i.e. compiled SML files containing proofs of lemmas and theorems) and between definitions, theorems, and lemmas. However, 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,

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 2 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. 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.

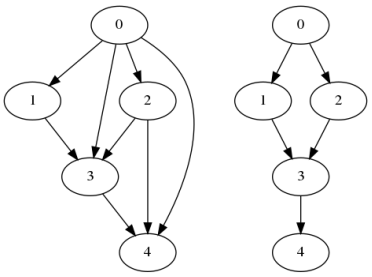

Figure 2: The right-hand side graph represent the transitive reduction of the left-hand side graph. Edges $(0 \to 3)$, $(0 \to 4)$ and $(2 \to 4)$ have been deleted because the transitive dependency relationship is preserved in the remaining edges. For example, the dependency $(0 \to 3)$ is represented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

DepGraph's frontend for theory dependencies is still useful for documentation purpose, and has been used to visualize HolBA's architecture.

4 The BIR NIC model

This section presents the three different approaches made in order to implement an equivalent BIR model from the formal NIC model. Tools that have been implemented in order to build it will also be introduced.

Multiple approaches to the translation of the formal NIC model to an equivalent BIR program have been considered, from handwritting a BIR program (Sections 4.3 to 4.5), lifting a C program (Section 4.2) to developing a new device model specific IR (Section 4.1).

4.1 Using flowcharts as Intermediate Representation

The CFG of the transition system of the formal model looks like a tree. Therefore, using flowcharts could be a convenient way to represent such structures. An attempt has been made to design a flowchart representation. Figures A.1, A.2 and A.3 show respectively a preview of the scheduler, transmission automaton and a particular transition of this automaton.

However, while this visual representation is useful in order to get to know the formal NIC model, we encountered several shortcomings:

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

it is the case of Figure A.3 representing one node of Figure A.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 A.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.

4.2 Writing the model in C

One-to-one translation of the definitions related to the transmission automaton, scheduler, state, and *CPPI_RAM* of the formal model have been realized in C. However, 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. Reasoning about the stack would require more code than a complete rewrite of the model in BIR. 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.

4.3 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.

A transition system similar to the one of the NIC model has been implemented, containing one scheduler and two automata. The two automata feature a simple linear transition system, and each of them has one non-autonomous transition, that is performed respectively by two external functions that represent memory accesses from the CPU in the NIC model.

This toy model has first been designed using the flowchart representation. Then, writing the BIR program has been 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. Section 4.4 presents BSL, a less verbose way of writing BIR programs.
- 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 4.5 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.

4.4 BSL: BIR Simple Language

A library, named BSL, offering the same expressiveness than BIR but with shorter constructs, has been implemented. 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.

BSL is composed of a set of functions with short names (prefixed with the letter b in order to not clash with names in the global namespace) and a coherent interface offering interoperability with HOL4 quotation system and smart use of partial function application (e.g. for BIR types).

4.5 Implementing the real model

From the knowledge gained in 4.3, a set of helper functions has been implemented, mainly in order

to facilitate reasoning about the state machine.

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.

5 Non-proof-producing automatic contract verification

This section presents the library that has been implemented in this work for automatic contract verification, after discussing about the problems that have been solved in order to build it.

5.1 Exporting BIR expressions to SMT solvers

In order to prove the implication needed to prove Hoare Triples, we must be able to export HOL4 goals to external SMT solvers. HOL4 features a library for interfacing SMT solvers and HOL4, called *HolSmtLib*, using the standard format SMT-LIB 2.0 [20]. However, for obvious reasons, *HolSmtLib* cannot export BIR expressions out of the box. Therefore, the BIR expression must be first translated.

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 HOL4's *wordsTheory* and *combinTheory*.

Because of the complexity of the BIR semantic, we deciced to implement a non proof-producing translation, in order to get more time for other experiments. 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, but a proof-producing version can still be implemented later.

bir_exp_to_words has been implemented as an exhaustive *match* statements on the expression type, each type being then translated to the corresponding non-BIR expression. Reccursive call is used for nested expressions.

5.2 BIR memories and SMT

In order to prove Hoare Triples containing BIR memories, we need to use *combinTheory*. However, it is not supported by *HolSmtLib*. In order to solve this issue, we proceeded in two steps:

1. we looked at how to translate BIR memory operations to SMT-LIB 2.0 and found *ArraysEx*. Then, we verified that this translation is sound by checking that *ArraysEx*'s axioms are correct in HOL4 (using *combinTheory*'s *UPDATE_APPLY* and *APP_def*, and *boolTheory*'s *EQ_EXT*).

2. we extended *HolSmtLib* in order to support this theory, and wrote tests in order to ensure that the implementation is correct, since this library is not proof-producing [5].

5.3 Pretty-printing to visualize BIR expressions

When working with complex constructs, the need for visualization techniques often arise. Generated WP grow quickly with the number of statements in a program, linearly or exponentially depending on the type of statements [6].

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

This expression is relatively small and yet the printed term is 17 lines long. Compared to the BSL expression that is 8 lines long[7], 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.

[5] *HolSmtLib* does have proof-reconstruction capabilities, but they are using an outdated version of Z3, the SMT solver that we used.

[6] Control flow statements produce exponential growth. While clever techniques can be implemented to keep their size reasonable [8], we often need to read and analyze them.

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 **??** shows the 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 **??**, 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 can see this feature in Figure **??** 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.

Appendix. *Figures*

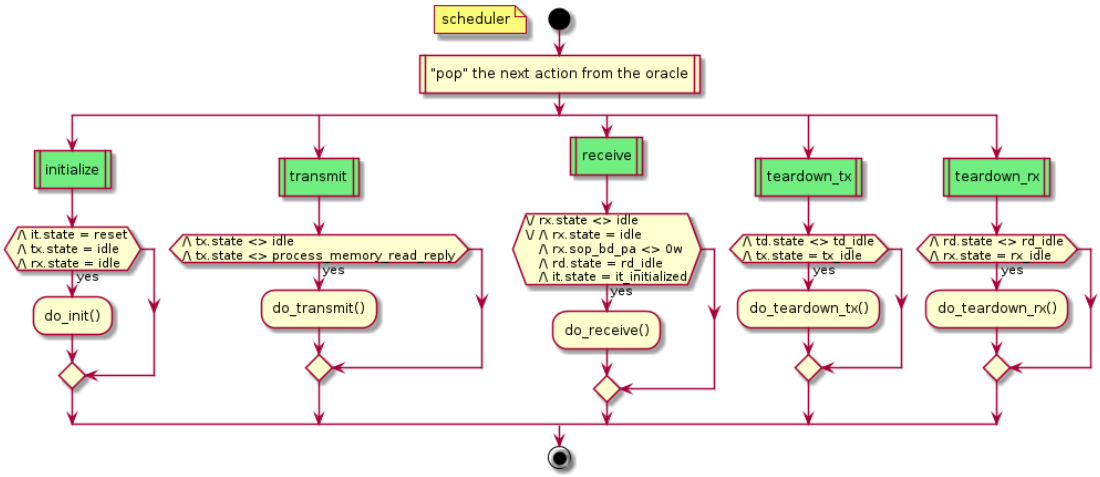

Figure A.1: Flowchart of the scheduler of the NIC model. Green nodes represent condition statements, here they represent the value of the popped action from the oracle. The full dot represents the entry point, and the other point the exit point.

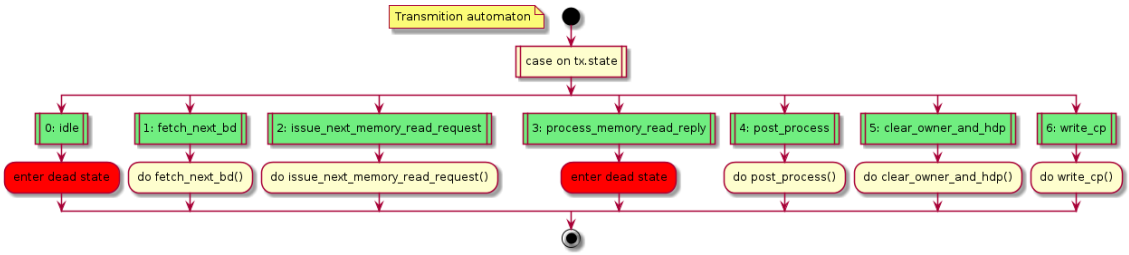

Figure A.2: Flowchart of the transmission automaton of the NIC model. Green nodes are similar to the ones of Figure A.1. Red nodes represent non autonomous transitions leading to dead states.

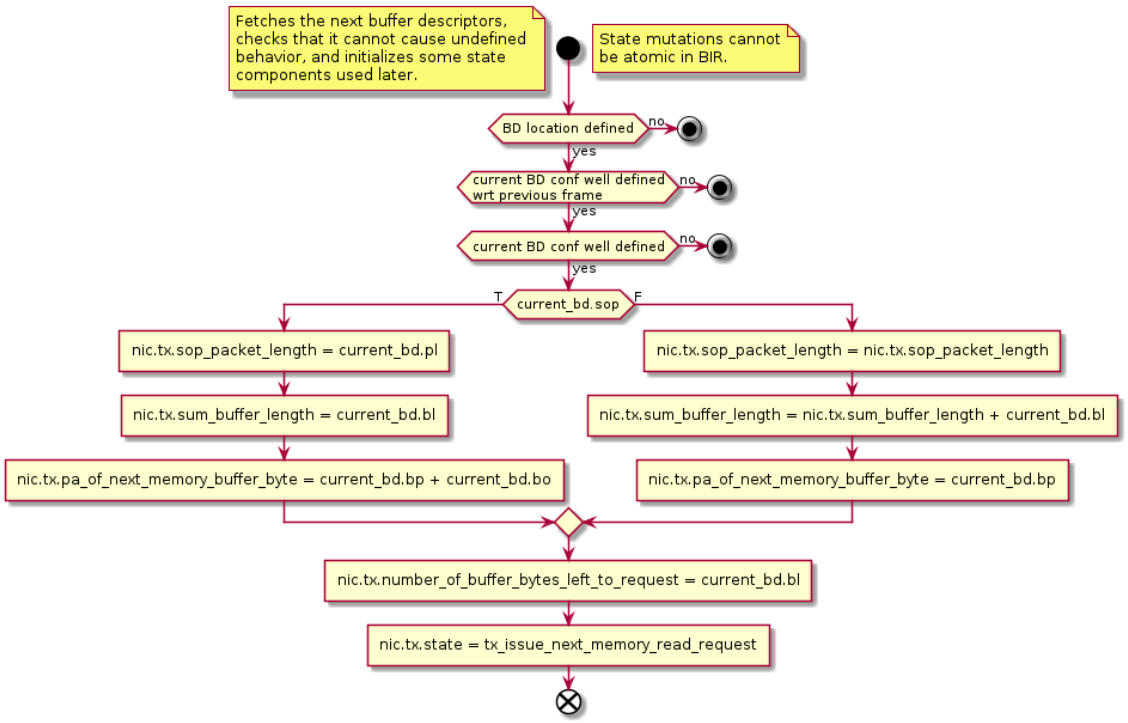

Figure A.3: Flowchart of the `fetch_next_bd` transition of the transmission automaton of the NIC model. The full dot represent the entry point, ⊗ represents the exit point and the other dots are shorthands to represent dead transitions (the symbols have been changed because of technical limitations of the tool used to draw the diagram).