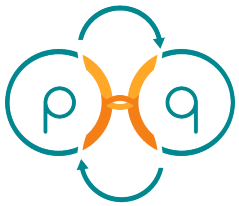
Master Thesis:

Deductive Verification of Reactive Programs



By Térence Clastres

supervised by Frédéric Dabrowski

LMV team, LIFO, Université d'Orléans

January - July 2024

Acknowledgments

I thank F. Dabrowski for his guidance and availability throughout the internship, and F. Loulergue for providing careful and detailed reviews that helped shape this master thesis into its current form.

I am also grateful for the kindness of the two PhD students who shared their office with me: F. Groult and J. Ischard. Their humility, advice and candid sharing of their experiences solidified my determination to pursue a doctorate.

Contents

List of Figures

Acronyms

API Application Programming Interface 18, 46

AST Abstract Syntax Tree 46, 54, 55

CTL Computational Tree Logic 25, 30

DV Deductive Verification 2, 3, 10, 13, 18, 19, 60–62

FOL First Order Logic 11–13, 19, 27, 35, 40, 43, 46, 47, 52, 53, 56, 60

LGBA Labelled Generalized Büchi Automaton 32, 33

LTL Linear Temporal Logic 2, 3, 25–28, 30, 31, 35–37, 40, 43, 46, 60, 62, 63

PID Process Identifier 51–53

PLC Programmable Logic Controller 57, 58

pLTL past Linear Temporal Logic 56, 61

RP Reactive Programming 23

SMC Symbolic Model-Checking 47, 53, 61

SMT Satisfiability Modulo Theories 19, 23, 47, 53, 54, 57, 59, 60

SRP Synchronous Reactive Programming 23, 24

TL Temporal Logic 1, 2, 25, 26, 29

VC Verification Condition 2, 19, 21–23

WP Weakest Precondition 18

Introduction 1

Reactive programs [44] are software in close interaction with their environment. They never stop executing as they must be ready to 'react' and process any change happening. Paired with the hardware required to execute them, their environment and possibly other reactive programs, they constitute a *reactive system*. Historically, reactive programs modelled electronic circuits, where changes are propagated at each tick of a clock. This led to synchronous reactive programming languages like Esterel [8], Lustre [26] and Signal [22]. Nowadays, reactive programming finds use in embedded systems (weather-station, smart-watch, video-game console, microwave oven...), some of which can be critical (plane autopilot, nuclear reactor control, missile defence...)[1]. With critical systems, any anomaly can cause catastrophic human and environmental consequences. Thus, one must first specify as precisely as possible what is the expected behaviour of the system (so for our case, the program(s) involved). Then, it is checked whether the system behaves accordingly.

To specify reactive programs, the formal framework of Temporal Logic (TL) [31] is generally used. Indeed, as reactive programs execute indefinitely, one cannot simply view the program as a 'transformational machine' [27] that, given an input, produces an output in finite time. Instead, one must be able to describe how the program state evolves through time: Is it certain this event triggers that action? Does the output stabilize at some point? Can the program deadlock? Rephrased as properties about the program, these questions (and any others one might have in regard to the program's behaviour) all fall into two categories [32]: *safety* (e.g. "there will never be a deadlock") and *liveness* (e.g. "eventually, the output must stabilize","after this event, this action must eventually happen"). Safety is therefore about invariants that must always hold during execution while liveness is about the certainty of a particular event occurring (albeit at an unknown time). The dif-

[1] While some of these systems are real-time, it is not a hard requirement.

ferent derivatives of TL, such as Linear Temporal Logic (LTL), allow for specifying both types of property concisely [20].

To verify a program respects its specification, multiple approaches exist. Testing might seem like a good solution at first glance: it's a generic method that is relatively easy and cheap to implement. In practice, one takes a program and repeatedly applies it to different inputs, checking if the corresponding outputs are what is expected. More formally, for finite systems i.e. systems that terminate after a finite number of states, one can provide an initial state s_i and an expected final state s_f, make the system run until it stops and check if the last state reached matches s_f. Provided the system is deterministic, it is then guaranteed that any run beginning in s_i ends in s_f. However, nothing can be said for other untested initial states. To totally characterize the system, one needs to repeat the process for all possible initial states. Thus, it is only achievable if the number of states is finite and relatively small.

While not as straightforward, the same can be done for infinite systems by abstracting them as finite ones using automata. Because those systems are infinite, there is no 'final state' (rather, the notion of reaching a set of final states infinitely often), so all possible states reachable during execution must be checked. This is what model-checking proposes to do in a somewhat efficient way, as long as the domain is small, like with electronic circuits. In practice, the user provides a finite state automaton representing the program and a specification written in TL. Then, after converting the specification to an automaton as well, the model-checker will make sure any path taken by the program's automaton corresponds to a path accepted by the one of the specification. Hence, this technique can also be viewed as a type of test over the different configurations the program can take while running.

However, while model-checking is vastly used and numerous optimizations at different stages have been proposed, when it comes to checking programs written in an expressive and high-level language, the state space is simply too important for this technique to produce results in a usable period of time. Moreover, because the program is abstracted into a model, it must be *refined* to a concrete version where it is shown the original model behaviour is preserved. Hence, other techniques could be more suitable.

Deductive Verification (DV) is a formal approach which directly links the program's code to its specification. Indeed, inference rules over the program syntax 'propagate' the specification across the code, producing proof obligations. These proof obligations, also called Verification Condition (VC), must then be proven us-

ing any technique one would use for regular theorems. The proofs can be computer-checked and sometimes even automatically found. When all proofs are discharged, the program *functional correctness* (the respect of its specification), holds. Because we mathematically proved the program exhibits the right properties, there is no need to do any tests (provided the specification is correct of course). Thus, strong guarantees are provided about the program's 'good' behaviour.

We introduce Hardy, a DV tool for reactive (synchronous) programs. The language accepted by Hardy consists of a setup procedure that will execute once at the start and a main procedure looping indefinitely, akin to Arduino [35]. The main procedure interacts with the environment through one or many inputs and outputs and also has an internal state to keep information between each iteration. The specification language uses LTL via one formula characterizing possible inputs and another one for the possible outputs. So far, Hardy is able to prove safety properties. To accomplish this, it begins by building the synchronized product of 2 Büchi automata generated from the aforementioned formulas. Then, it takes the resulting automaton and the setup/loop procedures to make a set of *Hoare triples*. These triples are finally sent to Why3 [19], which can prove them semi-automatically. The principle of this approach and its correctness have been formalized in the Coq proof assistant [7]. As far as we know, transforming a reactive program into subprograms whose correction using DV implies the correctness of the whole reactive program has never been attempted. Hence, it constitutes the original content brought by this research internship.

After the preliminaries about program verification and reactive programming are set (chapter 2), the present document will carefully describe our tool (chapter 3). Then, after discussing other attempted approaches and the current limitations of Hardy (chapter 4), it will go on to talk about related work (chapter 5). Finally, it will conclude by recalling the contributions made and what is planned in the future (chapter 6). The appendix contains more details regarding the Coq formalization (**??**).

Preliminaries 2

Contents

2.1 Programs & Verification

Since the beginning of computer science and programming languages, there has always been a mismatch between what the machine does and what the user (programmer) expects it to do. On the one hand, programming errors (syntax, semantic...) can result in a non-executable program, an undefined behaviour (arithmetic overflow, use of an uninitialized variable...) or a crash. On the other hand, even if the program runs and has a determined behaviour, it can still shift from its specification (expected output from a given input): this is a functional correctness issue. Even though both aspects are fundamentally different, their consequences are in all cases named 'bug'.

While a rounding error, a non-working button or a slow execution speed might not have a huge impact in everyday programs, they are a big concern for critical systems where human life is at risk: avionics, nuclear plant, medical machines...

Unfortunately, at all levels, from what the user wants (specification) to its realization (implementation) and then execution by a machine, bugs can happen. The compiler, the operating system or even processors' silicon executing the instructions are one of many examples of software and hardware entities that can have bugs.

Assuming a correct specification, will we ever be capable of certifying from beginning to end a program? Probably never in a general case. However, one can take care of one link in the chain, assuming correctness before and after this link. This is better than having no guarantee at all.

2.1.1 Formalization

To verify programs rigorously, their programming language associated must first be formalized. In general, we give a definition of its syntax (how to write programs in it) and of its semantics (what is the meaning of these programs).

For illustrative purposes, we will work on a very simple imperative language that only allows successive assignment of variables to values.

Syntax

At the most basic level, a language has an *alphabet* which contains letters (symbols recognized by the language). A finite sequence of letters form a *word* and a finite sequence of words form a *sentence*. The subset of all the sentences that can be formed constitutes a *language*. The *grammar* is what decides if a sentence is part of the language (well-formed).

In general, to give the grammar of a language, the Backus-Naur form [5] (BNF) or one of its variants is used:

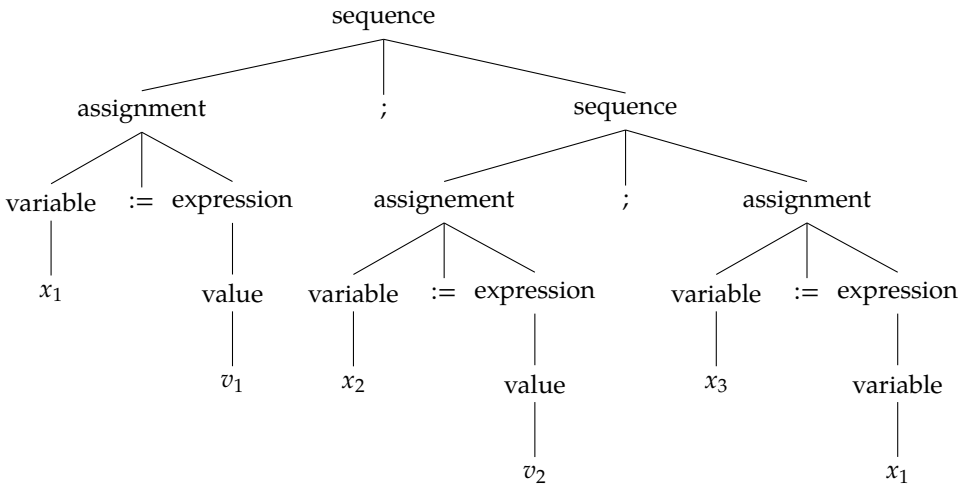
$$
\begin{array}{lcll}
c\text{: Command} & ::= & x := e & \text{assignment} \\
 & | & c \ ; \ c & \text{sequence} \\
e\text{: Expression} & ::= & x & \text{variable} \in \mathsf{Vars} \\
 & | & \mathrm{v} & \text{value} \in \mathsf{Vals}
\end{array}
$$

This (context-free) grammar contains two major syntactic classes: commands and expressions. We also add the singleton class of variables Vars and values Vals for convenience. A command can be an assignment or a sequence of commands (notice the recursion enabling the construction of arbitrarily long words). An expression is either a variable or a value (for our purposes, we do not have to precisely define what a value is). The grammar generates a very simple language, but sufficient for our example. A more involved one will be provided later.

Here is a sentence generated from the grammar:

$$x_1 := v_1 \; ; x_2 := v_2 \; ; x_3 := x_1$$

We can represent one way to construct it with a *derivation tree*:



By reading the leaves from left to right, we get back the original sentence.

Note this grammar is ambiguous because there exists more than one possible derivation tree for the same sentence. An easy way to realize that is by adding parenthesis:

$$(x_1 := v_1 \; ; x_2 := v_2) \; ; x_3 := x_1$$

$$x_1 := v_1 \; ; (x_2 := v_2; x_3 := x_1)$$

The tree represents the second way of parsing the sentence.

This is not ideal because it means there are different valid ways to parse the same sentence, which could produce different semantics. To solve this issue, one can either add *precedence and associativity rules* (explicitly tell what derivation to

choose) or *stratify* the grammar [42]. We can for example make the sequence right-associative so that its derivation tree is always the one presented here.

An alternative way of presenting the grammar is by using sets and inference rules:

$$\frac{x \in \mathsf{Vars} \qquad e \in \mathsf{Expression}}{x := e \in \mathsf{Command}} \qquad \frac{c_1, c_2 \in \mathsf{Command}}{c_1 \ ; \ c_2 \in \mathsf{Command}} \qquad \frac{\mathsf{e} \in \mathsf{Vars} \cup \mathsf{Vals}}{\mathsf{e} \in \mathsf{Expression}}$$

An inference rule $\frac{A \qquad B}{C}$ can be read 'from A and B we can infer C'. A, B and C are *judgements*. Judgements can be *premisses* or *conclusions* based on whether they are below or above the bar, respectively. An inference rule without any premise is an *axiom* and is always applicable as it does not depend on anything. To illustrate, consider the left rule. It states that if we have a variable x and an expression e, then, the construct $x := e$ is a command. We then define the language to be the smallest set closed by the rules, i.e. all objects in the sets (here, the words of the language) must have been constructed by repeatedly applying the rules a finite number of time.

Semantics

To characterize a program's semantics, we generally look at how its used memory evolves after each instruction, which we call its *environment*. The environment can be represented by a set of tuples $\langle id, value \rangle$ named *binders* associating variables manipulated by the program to their respective values. A specification is then a set of predicates over this environment.

There are 3 main ways of formalizing programming language semantics:

- *Operational Semantics* [41] focuses on the different states a program can be in by associating a valid execution to a series of valid inference rule. This will be the approach we take for this example.
- *Axiomatic Semantics* [28] ignores how the states of a valid program look but rather describes what are the observable effects of its computations.
- *Denotational Semantics* [46] expresses the program as an abstract model using mathematical objects, like functions.

Operational semantics is given either by big-step (also called natural) semantics or small-step (also called structural) semantics. Big-step semantics *evaluates* a program term to a value while small-step semantics *reduces* it to another term. Big-step

semantics provides a concise and intuitive way to understand a programming language, but it cannot be used to prove properties of diverging (non-terminating) programs, or to prove a program doesn't raise an exception. Indeed, a successfully applied big-step semantics always reduces to a value. On the other hand, small-step semantics is more precise with a greater number of rules. However, these rules may not represent how a compiler for the target language behaves. It is also harder to have an overview of the language semantics with this level of detail.

If one were to give both the big-step and small-step semantics of a programming language, it would be reasonable to show their equivalence. This would come down to proving that one 'big' step is a non-empty sequence of 'small' steps and, conversely, if a non-empty sequence of 'small' steps produces a value, it must be given by one 'big' step.

Let's give a small-step semantics for this simple language by first introducing some notations:

- $\omega \in \Omega$ is an abstract memory environment. It can be read and written to using $\texttt{set} : \Omega \times \mathsf{Vars} \times \mathsf{Vals} \to \Omega$ and $\texttt{get} : \Omega \times \mathsf{Vars} \to \mathsf{Vals}$ respectively.
- $\Rightarrow \subseteq \Omega \times \mathsf{Expression} \times \mathsf{Val}$ represents a relation between an environment, an expression and a value. It describes how an expression evaluates to a value given an environment.
- $\rightsquigarrow \subseteq \Omega \times \mathsf{Command} \times \Omega$ represents a relation between two environments and a statement. It describes how a statement updates a given environment.

Both arrow notations are infix and a comma separates the first two elements, e.g. $\Omega, v \Rightarrow v$.

We give two different relations because expressions are distinguished from statements, but we could have generalized statement as expressions that always return the same value (its type has a single inhabitant). This value is generally called `unit` and noted `()`. We could then have a unique relation of type $\Omega \times \mathsf{Expression} \times \mathsf{Vals} \times \Omega$.

The memory functions can be defined in terms of equations [4]:

$$\frac{}{\texttt{get}(\texttt{set}(\omega, x, v), x) = v} \qquad \frac{}{\texttt{set}\left(\texttt{set}(\omega, x, v), x, v'\right) = \texttt{set}(\omega, x, v')}$$

$$\frac{x_2 \neq x_1}{\texttt{get}\left(\texttt{set}\left(\omega, x_1, v\right), x_2\right) = \texttt{get}\left(\omega, x_2\right)}$$

Intuitively:

- Getting the value of a variable whose value you just set is the same as directly writing that value.
- Setting twice the same variable is the same as setting once the variable to the last value.
- If the last variable set is different from the variable we want to get, we can ignore it and look at ω before the `set`.

Let's now proceed with the small-step semantics:

Expressions semantics

$$\frac{}{\omega, v \Rightarrow v} \quad (2.1)$$

$$\frac{}{\omega, x \Rightarrow \texttt{get}(\omega, x)} \quad (2.2)$$

Statements semantics

$$\frac{\omega, e \Rightarrow v}{\omega, x := e \rightsquigarrow \texttt{set}(\omega, x, v)} \quad (2.3)$$

$$\frac{\omega, c_1 \rightsquigarrow \omega' \quad \omega', c_2 \rightsquigarrow \omega''}{\omega, c_1 \ ; \ c_2 \rightsquigarrow \omega''} \quad (2.4)$$

Intuitively:

- If an expression is a value, then it directly reduces to this value (2.1).
- If an expression is a variable, we `get` the corresponding value in the memory (2.2).
- If a statement is an assignment of an expression to a variable, and the expression reduces to a value, we `set` the variable to this value (2.3).
- If a statement is a sequence, we can split it into two statements as long as the first statement's last environment is the first environment of the second statement (2.4).

Let's apply these rules to $x_1 := v \ ; \ x_2 := x_1$:

$$\frac{\dfrac{}{\omega, v \Rightarrow v}}{\omega, x_1 := v \rightsquigarrow \texttt{set}(\omega, x_1, v)} \quad \frac{}{set(\omega, x_1, v), x_1 \Rightarrow get(set(\omega, x_1, v), x_1)}}{\texttt{set}(\omega, x_1, v), x_2 := x_1 \rightsquigarrow \texttt{set}\left(\texttt{set}(\omega, x_1, v), x_2; v\right)}$$
$$\omega, x_1 := v; x_2 := x_1 \rightsquigarrow \texttt{set}\left(\texttt{set}(\omega, x_1, v), x_2; v\right)$$

This is a proof tree, a notation created by Gentzen for his natural deduction proof system [23]. A well-formed proof tree has the proposition we want to prove at its

root and the leaves must be axioms. Valid inference rules must separate the root from the leaves.

This proof tree shows that beginning with an environment ω, executing the instructions results in an updated environment where x_1 and x_2 are both bound to v.

2.1.2 Deductive Verification

DV is a formal method used to verify that a program behaves according to its specification. A set of inference rules applied deductively are used to produce mathematical statements from the code. These statements, also called lemmas or theorems must be proven. Proving all the lemmas implies the program is correct. Sometimes, this can be done automatically by a *Theorem Prover* or semi-automatically with a proof assistant where the user 'assists' the computer towards finding a proof.

Specification Language

The specification language is the language used to describe the behaviour of a program. A good starting point is propositional logic.

Propositional Logic

p: Prop	$::=$	v	propositional variables
	$\mid$	$\top$	true
	$\mid$	$p \vee p$	disjunction
	$\mid$	$\neg p$	negation
v: Pvar	$::=$	e	

Propositional logic, whose grammar is given above, is about sentences that can be either true or false and ways to combine them. Here, a sentence is called a *proposition*. The simplest proposition is the one that is always true, noted $\top$. Then, we have *propositional variables*: they can represent anything as long as we have a way to tell if they are either true or false. For our program-checking use case, they are the expressions of our program's grammar. We also have 2 operators called *logical connectives*: the unary operator $\neg$ is the negation. It makes a true proposition false and a false one true. The binary operator $\vee$ is the disjunction. At least one of its two operands must be true. These two operators are sufficient to be *functionally*

complete: any other operator with any number of operands can be described using them. For instance, the false proposition is $\neg\top$, noted $\bot$ and the conjunction, requiring both of its operands to be true, is $\neg(p_1 \vee p_2)$.

Similarly to the programming language semantics, the ones of the specification language must also be given. Ultimately, because this language is about propositions being true or false, it will be easier to give its denotational semantics by a function I called the *interpretation* that maps propositions to $true$ or $false$.

$$I(P) = \begin{cases} \text{true} & \text{if } P = \top \\ V(v) & \text{if } P = v \\ \texttt{not } p & \text{if } P = \neg p \\ I(p_1) \texttt{ or } I(p_2) & \text{if } P = p_1 \vee p_2 \end{cases}$$

$V(v) : \mathsf{Pvar} \to \{\text{true}, \text{false}\}$ is the *valuation* function that associates a propositional variable to its truth value. We could for instance define $V(v) = \text{true} \leftrightarrow$ 'the last value of v in the program is 3'. Sometimes (in most cases), the valuation function is not given or is only partial: certain variables do not have a value. The goal is then to find which truth value to assign to which variable in order to make the proposition *hold* (have its interpretation be true). If such combination exists, the proposition is *satisfiable*, otherwise it is *unsatisfiable*. If any combination make the proposition hold, the proposition is *valid*. The satisfiability problem, abbreviated *SAT*, is a fundamental decision problem. A nice property of propositional logic is its *decidability*: there exists an algorithm that can decide in finite time whether a proposition is satisfiable. While decidability is about being able to give a definitive yes or no answer to a question, it can also be used here to get the variable truth values in case of a positive answer. However, the algorithms found so far are not efficient: they do not scale well as we increase the number of variables. It is believed no efficient algorithm will ever be found (see the $P = NP$ problem)

While propositional logic can be sufficient to specify simple programs, it lacks expressiveness for more complex ones. The next step is to add parameterized propositions we call *predicates*: this is predicate logic, also called First Order Logic (FOL).

First Order Logic

FOL is an extension of propositional logic where we can have 0,1, or more 'holes' in propositions. These holes can be filled with values of a certain non-empty domain

$\mathcal{D}$. We call such propositions *predicates*. The *arity* of a predicate is the number of arguments (holes) it takes. A predicate defines a *relation* between those arguments and is more expressive than a function: while a function can be viewed as a relation between arguments labelled as inputs and those labelled as outputs, a function requires the same inputs to produce the same outputs.

Syntactically, predicates (whose set is noted Π) are of the form $P(a, b, c, \ldots), a\, b\, c\, \ldots \in \mathcal{D}$ and are given an interpretation by the function $I : \mathcal{D}^* \mapsto \{\bot, \top\}$. So, propositions are a special case of predicate that is of 0-arity, so taking no argument. Moreover, those arguments doesn't need to be concrete elements of $\mathcal{D}$, they can be existentially ($\exists$) and universally ($\forall$) quantified upon, i.e., we can now say 'all variables of the program must not be null' or 'there exists $n \geq 2$ such that it is the divisor of v'. However, the increased expressiveness of FOL makes it undecidable in general.

Once a logical framework for specification has been chosen, one must enrich it with theories, that is (carefully chosen) rules and axiom made to capture the different mathematical objects used to talk about programs: numbers, lists, etc.

For example, we can add a[1] notion of equality to FOL, called *FOL with equality*: We give a (infix) binary operator $\equiv$ which represents the equivalence relation $eq \subseteq \mathcal{D} \times \mathcal{D}$. It is described by the following axioms:

$$\frac{}{eq\ x\ x} \qquad \frac{}{eq\ x\ y \leftrightarrow eq\ y\ x} \qquad \frac{eq\ x\ y \wedge eq\ y\ z}{eq\ x\ z}$$

$$\frac{\bigwedge_{i=0}^{n}\ eq\ x_i\ y_i}{eq\ P(x_0, \ldots, x_n)\ P(y_0, \ldots, y_n)} \qquad \frac{\bigwedge_{i=0}^{n}\ eq\ x_i\ y_i}{eq\ f(x_0, \ldots, x_n)\ f(y_0, \ldots, y_n)}$$

The first three axioms (reflexivity, symmetry, transitivity in order) are required for $\equiv$ to be an equivalence relation. The last two gives use a notion of congruence over predicates (functions): the valuations of two predicates (functions) are equal if and only if they have the same symbols and their arguments taken in order are equal.

[1] there exists many ways to define equality

Hoare's Logic

The most famous and widely used DV framework is *Hoare's Logic* [29], which was first applied to imperative programs (instruction order is accounted for) with FOL as the specification language. The program is denoted as a triple $\{P\}\ C\ \{Q\}$ where:

- C is the program's code
- P is a *precondition*, i.e., the specification the program must abide to **before** executing C instructions
- Q is the *postcondition*, i.e., the specification the program must abide to **after** executing C instructions.

Both the precondition and postcondition are parameterized by the program abstract memory environment to be able to mention its variables, the main way to describe its behaviour. For that, we can add to whatever underlying logic we use the `get` memory function we defined earlier (note that this is made implicit in practice as long as program variables can be differentiated from logic ones).

Note that the postcondition represents an over-approximation of the program's final state: every final state produced by the program must make Q hold, but not all states that make Q hold have to be produced by the program.

Let's understand the method by applying it to an example program written in the imperative language defined earlier.

Consider a program that swaps the value of two variables `a` and `b`. Let v_a and v_b be the possible values `a` and `b` can take, respectively. An implementation can be the following sequence of instructions:

```
c := a;
a := b;
b := c
```

Here, `c` is used to remember the value of `a` before it is assigned `b`. Then, `b` is assigned the old value of `a` via `c`.

We want to make sure the program actually does what it is supposed to do: swap the values of `a` and `b`. The first step is to write the specification of our program. Figuring out what language to write the specification with is crucial as it determines

what properties about the program can be expressed and how. We choose to use propositional logic in our example for simplicity.

Before the first instruction, we know `a` contains v_a and `b` contains v_b. So, our precondition P is $a = v_a \wedge b = v_b$ with $a = v_a$ $(b = v_b)$ the proposition that is true if and only if the variable `a` (`b`) is bound to the value v_a (v_b) in the program memory. We can formally express this by defining the interpretation function schema $I(x = v) \leftrightarrow \texttt{get}(\omega, x) = v$.

After the last instruction, the values must be swapped, so our postcondition Q is $a = v_b \wedge b = v_a$.

Consequently, we need to verify the following triple:

$\{a = v_a \wedge b = v_b\}$
```
c := a;
a := b;
b := c
```
$\{a = v_b \wedge b = v_a\}$

Hoare's logic then proposes a deductive system, so composed of axioms and rules to be applied for each syntactic constructs of the language used in C.

Here are 2 that are enough to verify the previous example:

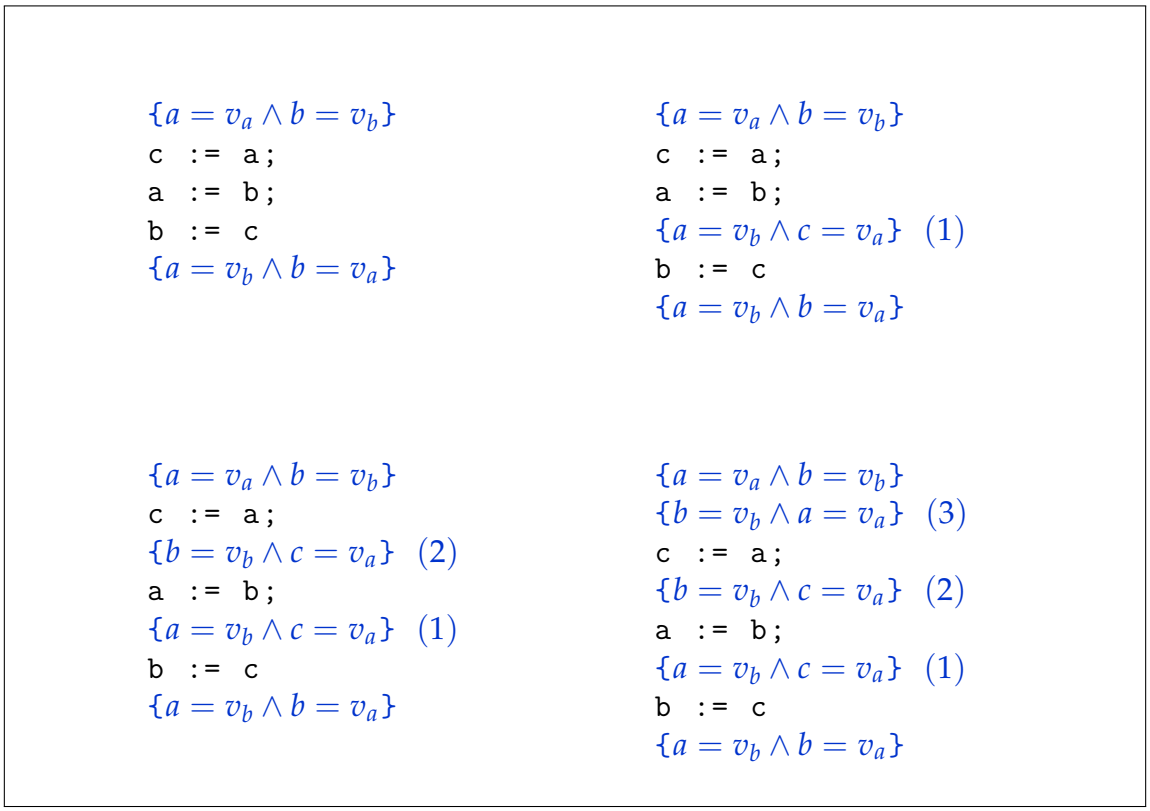
$$\frac{}{\{P[e/x]\}\ x := e\ \{P\}}\ assign \qquad \frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}}\ seq$$

The *assign* rule is a schema [13]: e and x are *metavariables* corresponding to the grammar defined earlier (there are as many *assign* rules as the Cartesian product of the sets containing the inhabitants of x and e).
This rule reflects the fact that if a predicate P holds after x is assigned e, substituting (rewriting) every occurrence of e with x within P must also make it hold before the assignment: $\overline{\{[3+2=5][3/a]\}\ a := 3\ \{a+2=5\}}$.

The *seq* rule is also a schema. Intuitively, if c_1 and c_2 are put into a sequence, c_1's postcondition is precisely c_2's precondition.

To make use of them in our example, we will apply the rules in succession starting from the postcondition and 'propagating' it all the way to the precondition using

the *seq* rule and others when applicable. The specification holds if the last predicate we get is a consequence of the precondition.



So, we end up with (3) which, after a simple permutation (allowed because $\wedge$ is a symmetric binary relation) gives us exactly our precondition: the program is *correct*.

Of course, things get more complicated when the language is extended, for example with conditionals (if...else...) and loops (while, for). In the case of loops, one must find an *invariant*, that is, a property that holds at each iteration. Moreover, finding an invariant is not enough, as it only guarantees *partial correctness*. In order to get *total correctness*, the loop must also terminate. Typically, we show there exists a natural number that strictly decreases at each iteration, called a *variant*. As $<$ over $\mathbb{N}$ is a well-founded order (the more general notion behind the variant), i.e., $\exists n \in \mathbb{N},\ \forall n' \in \mathbb{N}, \neg(n' < n)$ holds for $n = 0$, it follows the loop terminates after the last iteration for which $n = 0$. Hence, total correctness is, in the case of imperative programs, about termination. Later, we will see how termination is part of *liveness* properties.

Correctness and Completeness

One can wonder whether Hoare's logic is sound and is indeed able to prove program correctness.

To do so, we must relate the language operational semantics to Hoare's axiomatic semantics:

Definition 2.1.1 *A triple $\{P\}\ C\ \{Q\}$ is valid if applying the language operational semantics to C with an initial state ω satisfying P produces a new state ω' satisfying Q.*

Definition 2.1.2 *There is a proof of $\{P\}\ C\ \{Q\}$ if we can deduce Q from P using Hoare's axiomatic semantics on C.*

Theorem 2.1.3 (Correctness) *If we have a proof of $\{P\}\ C\ \{Q\}$, then $\{P\}\ C\ \{Q\}$ is a valid triple*

Proof. By induction on the proof of {P} C {Q}.

- ◊ *Base Case*: $\{P[e/x]\}\ x := e\ \{P\}$ must be valid.

  According to theorem 2.1.1, we must show

  $$P(\omega)[e/x] \wedge (\omega, x := e \rightsquigarrow \texttt{set}(\omega, x, v)) \rightarrow P(\texttt{set}(\omega, x, v)).$$

  Using the expression semantics, it is easy to see $P(\omega)[e/x]$ and $P(\texttt{set}(\omega, x, v))$ are equivalent.

- ◊ *Induction Step*: $\{P\}\ c_1; c_2\ \{R\}$ must be valid when $\{P\}\ c_1\ \{Q\} \wedge \{Q\}\ c_2\ \{R\}$ are valid triples. That is,

  $$P(\omega) \wedge (\omega, c_1 \rightsquigarrow \omega') \wedge (\omega', c_2 \rightsquigarrow \omega'') \rightarrow R(\omega'')$$

  Applying our first induction hypothesis to $P(\omega)$ and $\omega, c_1 \rightsquigarrow \omega'$, we get $Q(\omega')$. Finally, by applying our second induction hypothesis to $Q(\omega')$ and $\omega', c_2 \rightsquigarrow \omega''$ we get $R(\omega'')$.

□

Theorem 2.1.4 (Completeness) *If $\{P\}\ C\ \{Q\}$ is a valid triple, then we have a proof of $\{P\}$ $C\ \{Q\}$.*

We cannot prove completeness without introducing a new Hoare rule:

$$\frac{\forall \omega . P(\omega) \rightarrow P'(\omega) \qquad \{P'\}\ c\ \{Q'\} \qquad \forall \omega . Q'(\omega) \rightarrow Q(\omega)}{\{P\}\ c\ \{Q\}}\ implication$$

This rule states that a precondition can be strengthened or a postcondition can be weakened. Intuitively, it makes sense that if a statement holds assuming a property, it must also hold if we assume a 'stronger', more general one. Similarly, If we show a certain property holds, then any 'weak', particular derived property must hold too. We of course need to update our correctness proof to take the new rule into account, but we skip it here (the proof is trivial using the induction hypothesis).

This new rule is needed because proving completeness requires showing that a valid triple {P} c {Q} implies the existence of a precondition P' that is just strong enough to be a proof of Q, and that is implied by P. This special precondition is called the Weakest Precondition (WP). In other words, any valid triple {P} c {Q} means P can't be more precise (weaker) than P' if it allows for a proof of Q. Then, by using our new rule, we show that because P is stronger than P' it is also a proof of Q.

This gives a rough idea on how to prove the completeness of the system. However, it must be noted it depends on the completeness of the specification language as well: we assume we have a way to know if an assertion is valid. That is why we in fact showed the *relative* completeness of Hoare's logic.

Mechanization of verification

For more elaborate programs with richer languages, applying DV by hand is as fastidious as it is error-prone. Fortunately, tools have been developed to automate as much as possible this verification: Dafny [33], Viper [40] and Why3 [19] constitute a few examples.

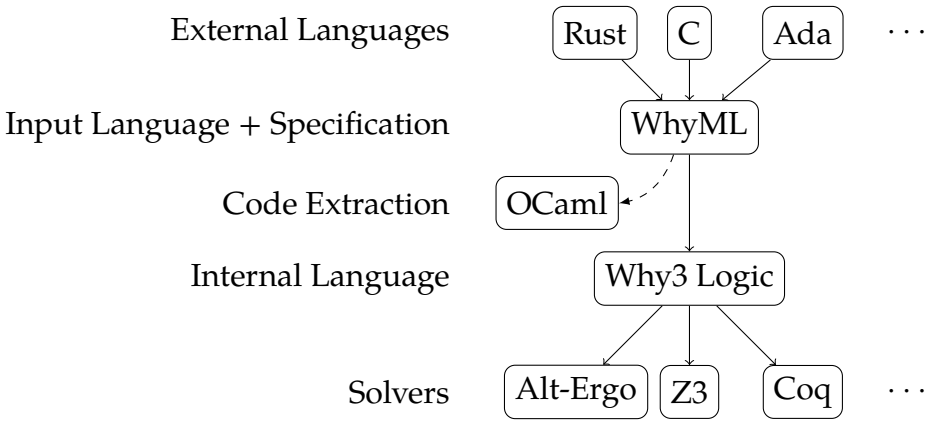
While our approach is general enough that it can make use of any such tool, we chose to use the one we had experience with: Why3. One nice aspect of the tool is that it provides an OCaml Application Programming Interface (API) to build untyped programs, allowing us to delegate the type-checking to the tool directly. This sped-up the development of our prototype.

We now proceed to explain the main aspects of Why3.

2.1.3 Why3 : a tool for deductive verification

Why3 is developed at Inria Saclay-Île-de-France and the Centre National de la Recherche Scientifique. It provides both a command line and graphic interface. Why3 enables DV of programs by providing a programming language extended to support specification called 'WhyML'. From a WhyML program containing some specifications, Verification Conditions (VCs), which are proof obligations, are produced. These VCs are presented to the user as a set of assumptions and unique goal to prove. They are sent to external theorem provers whose role is to come up with a proof for each of them. In general, one first tries to use Satisfiability Modulo Theories (SMT) solvers because they are mostly automatic. A theory here is a set of FOL formulas supposed true, so considered as axioms of the system. Solvers then check if given formula is satisfied under those axioms, so modulo the theory. Why3 contains a base library of FOL theories (algebra, numbers, sets...) that can be extended if needed. It is of course critical that the theory be consistent: it should not be possible to deduce false from the theory. Otherwise, anything could be proven by the famous 'ex falso quodlibet', i.e. *the explosion principle*. If SMT solvers can't immediately solve the VCs, one can guide Why3 for each VCs by applying a set of predefined *transformations* that preserve soundness. If this is still not enough, one can manually write a machine-checked proof with a proof assistant like Coq or Isabelle, both of which Why3 supports. Finally, when all VCs are *discharged*, the program is correct.

Figure 2.1 – *Why3 as a middle-end between languages and solvers*



Why3 can be seen as a middle-end between languages and solvers: indeed, there exists language frontends that make it possible to write programs in a subset of well-known languages like Creusot [16] for Rust, SPARK [9] for Ada or Frama-C's

WP plugin [6] for C (Figure 2.1). These frontends generally translate the original code into WhyML such that the semantics are preserved. Then, when the Why3 program is correct, the program written in the original language will be too. Another approach is to directly write the program in WhyML and then use Why3's extraction mechanism to have a correct-by-construction runnable OCaml code.

Example

Consider the following WhyML declaration:

Listing 2.1 – *swap in WhyML as instructions declaration*

```
use ref.Ref

let swap =
  let v_a = any 't in
  let v_b = any 't in
  let ref a = v_a in
  let ref b = v_b in

  assert { a = v_a && b = v_b };
  let ref c = any 't in
  c := a;
  a := b;
  b := c;
  assert { a = v_b && b = v_a };
```

Listing 2.1 shows what the variable swap example looks like when written in WhyML. To make it the most similar to our pen-and-paper version, we declared `v_a` and `v_b` to be `any` value of a certain type (denoted by `'t`) we know nothing about (it is irrelevant for this program's purpose). Due to the imperative nature of our example, the `ref.Ref` module is used to simulate memory access. `a` and `b` are then declared as references to `v_a` and `v_b`, respectively, with the `ref` keyword. We also have to declare `c` similarly before using it. All the declarations and instructions are contained within the top-level declaration `swap`.

Here, the first `assert` corresponds to the precondition and the last one is the postcondition (a precondition is considered as the first assertion of a series of instructions and a postcondition as the last one).

From this program, the following VC is generated (Listing 2.2):

Listing 2.2 – *VC generated*

```
------------- Goal ----------------
goal swap'vc :
  forall v_a:'t, v_b:'t, v_c:'t.
    c = v_a → (forall a:'t.
      a = v_b → (forall b:'t.
        b = c →
        a = v_b && b = v_a))
```

We can observe how program variables are quantified universally and instruction sequencing becomes imbrication of implications.

This goal is then simplified by instantiating universals and bringing hypothesis from implications from the goal to the context (Listing 2.3):

Listing 2.3 – *Goal simplified*

```
------------- Local Context --------------
type t
constant v_a : t        Ensures : c = v_a
constant v_b : t        Ensures1 : a = v_b
constant a : t          Ensures2 : b = c
constant b : t
constant c : t

----------------- Goal -------------------

goal swap'vc : a = v_b && b = v_a
```

Instead of specifying the instructions for a hardcoded value of `a` and `b`, we can abstract those values by putting the instructions inside a function, which is our `swap` let-declaration enriched with our variables as parameters:

Listing 2.4 – *swap in WhyML as a function declaration*

```
use ref.Ref

let swap (&a:ref 't) (&b:ref 't) =
  ensures { a = old b && b = old a }
  let ref c = any 't in
  c := a;
  a := b;
  b := c
```

With this approach (Listing 2.4), there is now a proper notion of precondition (`requires` keyword), which is what the function assumes to be correct, and a proper notion of postcondition (`ensures` keyword), which is what must happen after its execution. Together, they form the function *contract*.

Additionally, the `old` keyword used in the postcondition enables the retrieval of variable values at the beginning of the function `let` declaration. This way, it is not needed to explicitly mention the variable values and we do not require any particular precondition. Why3 also allows for a more general approach where we can talk about any place in the code by adding a *label* right before it. We can now apply this label to the variables to have their values at this point.

The generated VC is slightly different:

Listing 2.5 – *VC generated*

```
------------- Goal ----------------
goal swap'vc :
  forall a:'t, b:'t, c:'t.
      c = a → (forall a1:'t.
      a1 = b → (forall b1:'t.
        b1 = c →
        a1 = b && b1 = a))
```

Here, when a variable is assigned a value, its logic equivalent creates a new variable, concatenating its previous name with an integer. This allows us to keep the history of values a variable had and so, the initial value is bound to the integer-less name (e.g. `old a` is simply `a` and the last value of `a` is `a1`).

In both cases, it is then attempted to automatically prove these VCs with automatic SMT solvers like Alt-Ergo CVC3, or Z3. Finally, the user is notified whether it was successful or not. Here, we have a single VC associated with our program, but there are generally many. When this is the case, Why3 makes them independent of each other by assuming all prior specifications true and adding these assumptions to the next VC. This provides modularity and facilitates local reasoning. Our simple program is of course proven easily.

For complex programs, some additional information about what the program does needs to be gathered. This is the role of *ghost* code: it is regular program code that can only used by assertions. This code must be *transparent* to the program: it must not change its behaviour. Thus, ghost code can only read program variables and can't change the program's control-flow. Conversely, the program cannot make use of the ghost code in any way. During extraction, the ghost code is removed, having no effect to the program's semantics.

2.2 Reactive programming

At their heart, reactive systems are about constant reaction to their environment. Those systems run permanently, as opposed to transformational systems that take some inputs and stop executing after producing some outputs. Operating systems, graphical user interfaces, embedded systems are all examples of reactive systems. Reactive Programming (RP) tries to facilitate the development of programs for reactive systems via adequate language abstractions. For example, signals are values that can be present (active) or not (inactive). The system can then perform actions conditionned by their presence while also having the ability to create (emit) one. Concepts brought by RP have been implemented in different languages like JavaScript, Scala, or even Java and Haskell.

2.2.1 Synchronous Reactive Programming

Synchronous Reactive Programming (SRP) is a particular case of RP. Programs written under this paradigm can be seen as a black box M that repeatedly takes an input I and produces an output O. To have a program that is interesting, M also has an internal state X that persists between each new input (see Figure 2.2). A program denotational semantics can be a function $M : I \times X \mapsto O \times X$ that gets repeatedly called, where the state of the last calls feeds the state of the new one

together with the new input. To achieve this, the program takes the inputs from the stream one by one and associate each with an output that is obtained after a certain number of instructions. The time interval that goes from the reception of an input to the production of an output is called an *instant*. An instant cannot last indefinitely, so it must be proven the instructions executed terminate.

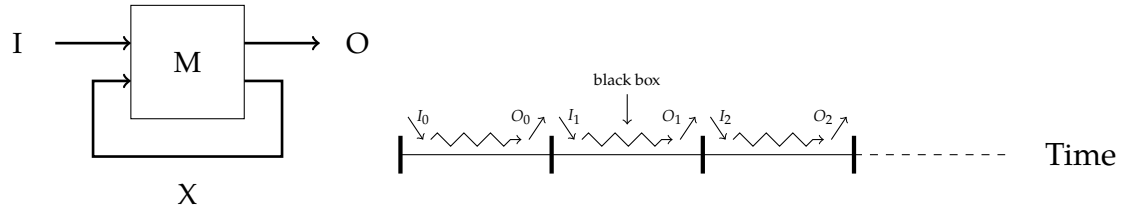


Figure 2.2 – *Schematic representation of a synchronous reactive program*

SRP was initially used to model electronic circuits with languages like Esterel [8], Lustre [26] and Signal [22].

2.2.2 A mathematical representation of infinity

Because of the infinite nature of reactive programs, we must use specialized mathematical tools to describe them. Notably, we saw that they are fed an infinite number of inputs and produce an infinite number of outputs. While a finite number of consecutive elements are called a *list*, an infinite number of consecutive elements are called a *stream*.

Let's begin by formally defining a list. A list over a set A is the *smallest set closed forward* by the following rules:

$$\frac{}{nil \in List} \qquad \frac{l \in List \qquad a \in A}{a \cdot l \in List}$$

The set is *closed forward* because the rules are read from premises to conclusion, which means we can construct the *List* set: we define the function $\mathcal{F} : List \mapsto List$ by $\mathcal{F}(\emptyset) = nil$ and $\mathcal{F}(L) = \bigcup_{l \in L} \{a \cdot l | a \in A\}$ then, *List* is given by the following:

$$List = \mathcal{F}(\emptyset) \cup \mathcal{F}(\mathcal{F}(\emptyset)) \cup \mathcal{F}(\mathcal{F}(\mathcal{F}(\emptyset))) \cup \ldots$$

The fact *List* is the smallest closed set means it is the least fixed point of $\mathcal{F}$, that is, the smallest x for which $\mathcal{F}(x) = x$ Because we can define a partial order relation

$\preccurlyeq$ for *List*, forming a complete lattice, and because $\mathcal{F}$ is monotonic function, the *Knaster–Tarski theorem* tells us this fixed-point indeed exists.

To define a stream, we can take the same rules we used to define a list but define it as the *largest set closed backward*. We say closed backward because we read the rules from conclusion to premises instead of the other way around. Thus, instead of constructing values of that set, we start with a set already made up of potentially infinite objects. We then apply to its elements the given rules backward. Hence, when we take an element (stream) of the set, it must either be nil or it can be decomposed into one element and another stream.

2.2.3 Specification

Because reactive programs have time-varying behaviours, it makes sense to specify them using TL, that is, any logic that allows reasoning about time. It is a refinement of *modal logic*: while in modal logic, there exists only unary time operators which express possibility (noted $\diamond P$, P may or may not happen in the future) and necessity (noted $\Box P$, P must happen in the future), TL is richer. Indeed, one is able to reason more precisely about time as we'll soon see.

There are two ways to view time, which represents two distinct logics:

- Time is **linear**, i.e. there was a past, there is a present and there will be one future. The passing of time is deterministic.
- Time is **branching**, i.e. there was a past, there is a present but there can be multiple possible futures. The passing of time is non-deterministic, unpredictable.

In linear-time logic, there generally is the *next* unary operator, noted $\mathcal{X}\phi$, and the *until* binary operator noted $\phi \; U \; \psi$. $\mathcal{X}\phi$ means the formula must hold at the next instant (time is discretized in finite atomic moments) and $\phi \; U \; \psi$ means every next state must satisfy ϕ until a state satisfying ψ comes up (and it **must** come up at some point).

The branching-time logic re-uses these operators, but they must be preceded by an existential or universal quantification other the possible futures. An example of branching-time logic is the Computational Tree Logic (CTL).

Of course, it is possible to combine the two views in one and have properties use both at the same time. *CTL** is one such system which is a strict superset of both CTL and LTL.

For our purposes, we focus on LTL.

Linear Temporal Logic

LTL is the linear-time TL that is the most commonly used for specification of systems requiring both safety and liveness properties. An LTL formula describes infinite words in a finite manner. Applied to program specification, it describes the trace of execution of a program. Indeed, a reactive program's trace can be seen as an infinite word where each letter represents a particular state and two consecutive letters mark a transition from one state to the other.

Syntax

$$
\begin{array}{lcll}
\phi\text{: LTL} & ::= & P & \text{atomic proposition} \\
 & | & \neg\phi & \text{negation} \\
 & | & \phi \vee \phi & \text{disjunction} \\
 & | & \mathcal{X}\phi & \text{next} \\
 & | & \phi\,\mathcal{U}\phi & \text{until}
\end{array}
$$

LTL is sometimes also called *Propositional Temporal Logic* because when applied to program specification, its base element is a proposition.

For convenience, additional temporal operators are introduced:

$$
\begin{array}{lcll}
\phi\mathcal{R}\psi & = & \neg(\neg\phi\,\mathcal{U}\,\neg\psi) & \text{(release)} \\
\mathcal{G}\,\phi & = & \mathit{false}\,\mathcal{R}\,\phi & \text{(globally)} \\
\mathcal{F}\,\phi & = & \mathit{true}\,\mathcal{U}\,\phi & \text{(eventually)}
\end{array}
$$

Semantics

As time is considered linear and discrete, LTL describes infinite words. More precisely, it is a subset of $\omega - regular$ expressions. They are an extension regular-expressions to infinite words. For example, a^* is a word only made up of a finite number of a while a^ω is a word composed of an infinite number of a. Similarly to their finite counterparts, $\omega - regular$ expressions are decidable, so LTL is also decidable.

LTL is a future-oriented logic: it can talk either about the present or the future, but not the past[2]. Given an alphabet Σ and an infinite word σ over this alphabet (each letter is a subset of Σ, so can be viewed as a unary predicate), an interpretation of LTL using FOL is provided:

$$
\begin{array}{ll}
\sigma \models P & \iff P \in \sigma[0] \\
\sigma \models \neg\phi & \iff \sigma \not\models \phi \\
\sigma \models \phi_1 \vee \phi_2 & \iff \sigma \models \phi_1 \vee \sigma \models \phi_2 \\
\sigma \models \mathcal{X}\phi & \iff \sigma[1..] \models \phi \\
\sigma \models \phi_1 \mathcal{U} \phi_2 & \iff \exists j : j \geq 0.\ \sigma[j..] \models \phi_2 \wedge \forall i : 0 \leq i < j.\ \sigma[..i] \models \phi_1
\end{array}
$$

Intuitively:

- if a formula consists in just a proposition, this proposition must hold for the first letter σ.
- if a formula is the negation of a subformula, it is the negation of its truth value
- if a formula is a disjunction of two formulas, it is the disjunction of their truth value.
- if a formula is the ne$\mathcal{X}$t of a subformula, it must hold starting at the word second letter.
- if a formula is the $\mathcal{U}$ntil of two subformulas, the first subformula is required to hold until the second one holds. Note that the second subformula must be true at some point.

[2] there exists a past-oriented version called PLTL that can be exponentially more succinct to write, but is equivalent to LTL [38]

Verification

Expressivity

There is an equivalence between LTL, star-free omega regular languages and the First-Order Theory of Linear Orders, a result known as *Kamp's theorem* [30]. Hence, LTL does not capture the full expressivity of omega regular languages. For instance, a property such as 'P holds for every even instants' cannot be stated.

Many extensions have been attempted to increase LTL's expressiveness ([2, 15, 34]), however, it remains to be seen how those attempts can be made to work within our framework.

Safety & Liveness

Program Temporal properties are generally classified as either safety or liveness properties. However, [37] proposes to categorize them hierarchically and study them under different views (linguistic, topological, automata theory, logical). For example, if we consider all the possible states a program can be in as an alphabet, a run of a program consists in a succession of states, so a word. When we give a program property, we give an expectation of the program's behaviour. If the program's entire behaviour can be captured using its execution trace, a property distinguishes between valid and invalid traces. So, a property can be seen as a language: it is the language of traces that make the property hold. If the program eventually halts, its execution trace is a finite word. Properties on such traces, so a language over finite words, is called a *finitary* property. If the program never halts, it produces an infinite trace, making up an infinite word. Properties characterizing such words are called *infinitary* properties. By introducing a set of operators, we can create infinitary properties from finitary ones. These operators rely on describing infinitary words in terms of their (finite) prefixes. 4 operators are provided, each corresponding to a class of infinitary properties:

- Safety properties have all their prefixes recognized by a certain finite property
- Guarantees properties have at least one prefix recognized by a certain finite property
- Recurrence properties must have infinitely many prefixes recognized by a certain finite property

- Persistence properties have a fixed number of prefixes not belonging to a finite property

Intuitively, safety properties hold at each step of the execution, guarantees properties hold at least once in the execution, recurrence properties don't hold all the time, but infinitely often and persistence properties hold all the time, except in a finite number of states.

To justify they are indeed classes, they are proved to be closed by union and intersection. They additionally form a hierarchy because Recurrence and Persistence classes properly contain the Safety and Guarantee ones, the formers remaining strictly more expressive.

2.2.4 Verification with Model-Checking

The standard for verification of temporal properties is *model-checking*.

The basic idea is to have a 'model' of the program: a finite transition system describing the ways the system state can evolve, called a *Kripke Structure*. Then, this structure is converted into a finite-state automaton $\mathcal{A}_{program}$. The language $\mathcal{L}_{\mathcal{A}_{pgrm}}$ recognized by this automaton is all the possible execution traces the program can have. Additionally, another automaton $\mathcal{A}_{spec}$ is constructed from the TL specification of the program. Because the formula's atoms describe the program states, the language $\mathcal{L}_{\mathcal{A}_{spec}}$ recognized by this automaton represents all the execution traces of the program that are accepted by its specification. At this point, we want to make sure all possible executions of the program are allowed by its temporal specification, so we want a decision procedure for $\mathcal{L}_{\mathcal{A}_{pgrm}} \subseteq \mathcal{L}_{\mathcal{A}_{spec}}$. This can be converted to a language-emptiness check: $\mathcal{L}_{\mathcal{A}_{pgrm}} \cap \overline{\mathcal{L}_{\mathcal{A}_{spec}}} = \emptyset$, Indeed, taking the complement of the specification represents the language of all forbidden execution traces. If its intersection with $\mathcal{L}_{\mathcal{A}_{pgrm}}$ is non-empty, it means at least one trace of the program violates the specification. From the automata's point of view, one must construct the complement of $\mathcal{L}_{\mathcal{A}_{spec}}$, combine it with $\mathcal{L}_{\mathcal{A}_{pgrm}}$ by doing a synchronized product and finally check if the resulting automaton contains at least one accepting path. However, complementation of such automata is expensive (current algorithms are of exponential complexity). Instead, the temporal formula is negated before constructing its automaton. While it suffices to find only one accepting path to prove the program violates its specification (and this path can be used to show how to reproduce the incorrect behaviour), a well-behaving program requires systematic verification of all possible paths. It has been shown that model-checking with ω-

automata results in a PSPACE upper-bound complexity: the decision procedure can be represented by a deterministic Turing machine using polynomial space. More precisely, model-checking LTL and CTL* formulas is PSPACE-complete and CTL is P-complete.

This verification method as been successfully applied many times, but it still has two main issues:

- The model might not be representative of the actual program
- Complex programs generate very large automata where counterexample search becomes highly impractical due to the massive memory amounts required (it could also happen because of large specification formula but in practice, the model state-space is at least an order of magnitude greater).

Büchi automaton

When LTL is used as a specification language, a Büchi automaton can be computed: it is an ω-automaton (other such automata exists, but only differ by their word-acceptance condition). Since it is a (finite) automaton accepting infinite words, it follows each state as at least one successor.

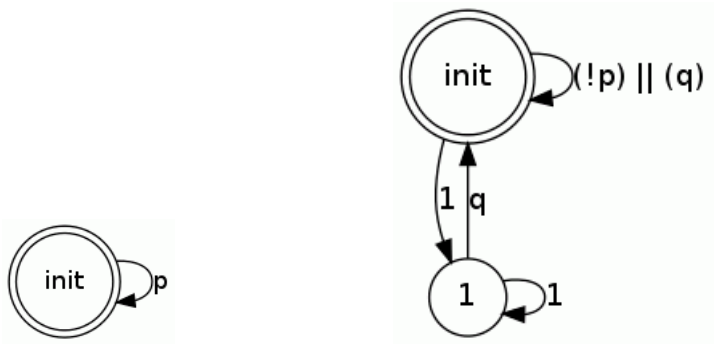
Intuitively, because words are not finite, the notion of final states we would have to be in when a finite word ends does not make sense. Rather, we define a set of acceptant states, and impose that for a path to be valid (a word to be accepted), there must be an infinite amount of states in the path belonging to the set.

More formally, a Büchi automaton $\mathcal{BA}$ is a tuple $\langle Q, \Sigma, q_0, \delta, T \rangle$ where

- Q is a finite set of states
- Σ is a finite alphabet
- $q_0 \subseteq Q$ is the initial state
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation
- $T \subseteq Q$ is the set of acceptant states

We then define $Inf(\sigma)$ as the set of states infinitely present in σ and say a word is recognized by $\mathcal{BA} \iff Inf(\sigma) \subseteq T$. This is called the Büchi automaton acceptance condition.

Examples



(a) Gp (b) $G(p \rightarrow Fq)$

Figure 2.3 – *2 examples of Büchi automata*

Figure 2.3 shows two Büchi automata representing each an LTL formula.

We can see how the $G\ p$ formula is simply a single accepting state and a single transition to that state labelled p. An accepted word is $\sigma = pppppp\cdots$ Which indeed corresponds to the formula.

The second automaton is more interesting: it has two states and only one is accepting. We get to the non-accepting state when p holds and the only way to get out of it back to the accepting state is to have q. This indeed correctly models the fact that p must imply that q will hold later. An accepted word is $\sigma = pppqpppqpppqq\cdots$.

Generalized Büchi automaton

In general, one does not directly obtain a Büchi automaton from an LTL formula. Instead, it is first translated into a generalized Büchi automaton, where the accepting condition is relaxed: there is now a family of accepting states : $\{T_1, \ldots, T_2\}$ where $T_i \subseteq Q$. A word is accepted if a path passes through infinitely often at least one state of each accepting state family.

2.2.5 Construction

On-The-Fly-construction

The algorithm proposed in [24] provides a set of nodes used to build a Labelled Generalized Büchi Automaton (LGBA). The principle is the same as in [47].

The algorithm takes a node *Node*, a set of nodes *NodeSet* and returns a new set of Nodes.

'Node' is a data structure containing:

- the name of the node
- A set 'Incoming' of all the arcs incoming to the node
- A set 'New' of non-treated formulas that must hold for n
- A set 'Old' of already-treated formulas that must hold for n
- A set 'Next' of formulas to be treated by every direct successor for which the Old formulas hold

Initially, the algorithm is given a node n where *New* only contains Φ and *NodeSet* is empty. Then, it is checked whether its *New* set is empty:

- If *NewSet* is empty, it is checked whether n already exists inside *NodeSet*.
  - If it is the case, its incoming arcs are updated and *NodeSet* is returned.
  - Otherwise, a new node n' whose *New* formulas are the untreated *Next* formulas of n is created. Then, the algorithm is recursively called on $(n', \{n\} \cup \mathit{NodeSet})$
- Else, one formula η is removed from *New* and filtered according to its shape:
  - If it's an **atom**, it is checked whether adding it to the *Old* formulas introduces a contradiction.
    - If there is one (the atom is $\bot$ or its negation is present), *NodeSet* is returned.
    - Else, it is added to *Old* and recursed again over the current node n and *NodeSet*

- If it's a **conjunction** $\phi \wedge \psi$, the current node is updated by splitting the formula in half and adding its two components ϕ and ψ to the *New* formulas. The node is then recursed upon.
- For the other binary cases (**disjunction**, **until** and **release** ...), two new nodes are created and recursed on one after the other. The formula added to the *New* set of each node depends on the actual operator of the formula

The authors then proceed to explain how to take the set of nodes returned and produce a LGBA from it. They finally give the accepting conditions.

Surprisingly, while a proof of correction of the algorithm is given, nothing is said about the termination of the algorithm. It turns out not to be trivial [45].

Hardy

3

Contents

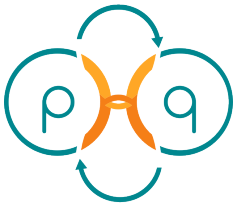


Figure 3.1 – *Hardy's logo: a stylized two-state ω-automaton forming the 'H' of Hardy in the middle*

Hardy is a tool capable of proving functional correctness for synchronous reactive programs. Hardy takes a program written in an imperative language and two formulas providing its specification. The important aspect of this language is how it allows for outside interaction thanks to special input and output variables. Hardy programs are synchronous reactive because they repeat indefinitely the same instructions, and do so in the following manner: Before executing the instructions, input variables are updated with fresh values from the outside. This marks the beginning of the instant. After executing the instructions, the last value of the output variables is forwarded to the outside. This marks the end of the instant. The 'outside' is abstracted to anything that can provide and receive data streams. Additionally, programs can have global variables that persist their values between instants. Thus, an internal state can be maintained, enabling more expressive programs with complex behaviours. Hardy programs are translated into WhyML so that they can be verified deductively in Why3.

A program specification is given by two types of LTL formulas:

- `relies on` formulas are similar to preconditions and constrain the inputs the program receives over time.
- `guarantees` formulas are similar to postconditions over both inputs and outputs, describing how the program outputs evolve over time.

The atoms of the LTL formulas are FOL formulas whose own atoms are program expressions.

The main idea behind Hardy is to reduce the reactive program and its specification to a set of Hoare triples which, when proved correct, guarantee the correctness of the original program. For this purpose, an external tool called LTL2BA [21] translates both specification formulas to Büchi automata labelled by the formulas atoms. These 2 automata are then processed to get the set of Hoare triples. Hardy is not a model checking tool (symbolic or not) simply because it works with actual code extracts of the program and not a model. This eliminates the need to provide a model of the program that might contain too many states, so taking too much time to model check or not enough, and being too imprecise to actually prove anything.

Usage

The following example represents a simple reactive program for a switch controlling a light.

```
input on off : bool;
output light : bool;

relies on { G ( not on | not off ) }
guarantees { on R (not light | on) }
guarantees { G (on -> off R (light | off)) }
guarantees { G (off -> on R (not light | on)) }

setup:
    ensures { light = false }
    light := false;

loop:
    if on then light := true;
    else if off then light := false;
    end
```

It has two boolean inputs on and off that correspond to the presence or absence of an on and off signal the switch can send. Its unique output is the boolean light which commands the light to switch on or off. There is

Initially, the light is off. This is the role of the `setup` routine. Thus, its trivial postcondition is light = false as indicated by `ensures` (note that because the setup instructions are a one-off, there is no point in using an LTL formula here). The instructions to be executed repeatedly are inside the `loop` routine: they state that if we have the on signal present, we turn on the light. Otherwise, if the off signal is present, we turn it off.

With regard to the specification, we obviously don't want any input to be at the same time on and off: `G ( not on | not off )`. Then, there are 3 things we expect our output stream to respect:

- Initially, the light must not be turned on until we get the on signal: on `R (not light|on)`.
- If at any instant we get the on signal, the light will be on from now on until we get the off signal (if we ever get it): `G (on -> off R (light | off))`
- Conversely, if at any instant we get the off signal, the light will be off from now on until we get the on signal (if we ever get it): `G (off -> on R (not light | on))`

After writing the program and its specification, we use Hardy's command line interface, passing it the name of the text file containing the program and a path to LTL2BA: `hardy light -ltl2ba etc/ltl2ba`

A directory is then created, containing:

- the output of LTL2BA for both specification formulas
- a visual representation using the DOT format of the automata
- a *.mlw* file that must be verified using Why3. Its content is shown in Listing 3.1.

We see the generated WhyML program contains our inputs as read only variables (indicated by the val keyword) and our single output as a writable reference. The setup routine is transparently translated, but we find the loop routine instructions have been translated to declarations which only differ by their respective precondition(s) and postcondition(s). Apart from an existential declaration, the specifications directly use the content of the LTL formulas atoms.

We will now proceed to formally define the language and explain in details the translation process.

Listing 3.1 – *mlw file content*

```
module Program
  use ref.Ref
  use int.Int

  let light = ref (any bool)
  val on : bool
  val off : bool

  let setup = begin   ensures { (! light) = false }
    light := false end

  let pre_init_post_S2 = begin
    requires { exists on : bool, off : bool. (¬ off ∧ !light)
                  ∨  (on ∧ (¬ off)) ∧ (! light)
    }
    requires { (¬ on)  ∨  (¬ off) }
    ensures { ((¬ off) ∧ (! light))  ∨  ((¬ on) ∧ off)
                ∧
                (¬ (! light)) }
    if on then (light := true)
    else if off then (light := false) else ()
  end

  let pre_init_post_init = begin
    requires { (!light = false) ∨
               (exists on : bool, off : bool.
                (¬ on ∧ off ∧ ¬ !light)  ∨  ¬ on ∧ ¬ (! light))
    }
    requires { ¬ on  ∨  ¬ off }
    ensures { (on ∧ ¬ off ∧ !light)  ∨  ¬ on ∧ ¬ (! light) }
    if on then (light := true)
    else if off then (light := false) else ()
  end
end
```

3.1 Language

Here is a simplified formal grammar of our language.

δ : Declaration	::=	κ x `:` τ `;`	$\kappa \in \{$`var input output`$\}$

Program	::=	δ^*
		`rely` ϕ
		`guarantee` ϕ
		`setup :`
		  `ensures` π
		  c
		`loop :` c	

π : FOL	::=	`true` | `false`	boolean
	|	e	program predicate
	|	$\neg\ e$	negation
	|	$\pi \circ \pi$	$\circ \in \{\oplus\ \Leftrightarrow \Rightarrow \wedge \vee\}$
	|	`forall` x `:` τ `,` π	universal quantifier
	|	`exists` x `:` τ `,` π	existential quantifier
ϕ : LTL	::=	`true` | `false`	boolean
	|	`[` π `]`	predicate
	|	$\triangleright\ \phi$	unary $\triangleright \in \{F\ G\ X\ \neg\}$
	|	$\phi \square \phi$	binary $\square \in \{U\ R\} \cup \circ$
c : Command	::=	`skip`	
	|	$x := e$	assignment
	|	c `;` c	sequence

	|	`if` e `then`
		  c `else` c
		`end`
			conditional

	|	`while` e `do`
		  `invariant` π
		  `variant` e
		  c
		`done`
			loop

e : Expression	::=	x	variable with $x \in \mathit{ID}$
	|	v	values
	|	$e \diamond e$	binary $\diamond \in \{+\ -\ \times\ \div\ \leq \geq < > =\}$
v : Value	::=	n	machine integer
	|	`tt` | `ff`	boolean
τ : Type	::=	`bool` | `int`	

Hence, a program first consists of some variable declarations that can be of 3 types:

- `var` are global read-write variables representing the program internal state. Their content is preserved between instants.
- `input` are read-only variables that provide outside values to the program. At each instant, the old value is cleared and a new one takes its place.
- `output` are read-write variables that allows the program to provide results of a computation at each instant.

Then, one must declare a `rely` and `guarantee` formula. Specified both using LTL, the former describes how the inputs received will be shaped (e.g. *globally, no negative values*), while the latter is about how the outputs will evolve at each instant. Together, they specify the `main` routine. To talk about the program, FOL formulas' atoms are program expressions while LTL formulas' atoms are FOL formulas.

Next, the `setup` routine is defined. It is the program part that will be executed once, before any input is provided. After its execution, the postcondition denoted `ensures` must hold. Note how the formula is defined using FOL and not LTL, as there is no need to have the notion of time.

Finally, the `loop` routine is the main code of the program and the one that will be executed at each instant.

Accepted instructions inside `setup` and `loop` are very basic: assignment, sequence, conditional, and while loop. Values can be of type `int` or `bool` and can be combined within expressions using the standard unary and binary operations. The `while` instruction requires both `variant` and `invariant` as we've seen in subsection 2.1.2. The variant is a program expression while the invariant is a FOL formula (there is no notion of time inside instructions).

Semantics

Let $X_I, X_O, X \subseteq \mathit{ID}$ be the set of program inputs, output and global variables, respectively.

Let p be a program receiving a stream of inputs of type τ_i and producing a stream of outputs of type τ_o (if inputs/outputs of different types $\tau_a, \tau_b \ldots$ are declared, one can simply take their product $\tau_a \times \tau_b \times \ldots$). The program internal state is maintained through the use of global variables X.

Then, c_{setup} and c_{loop} represent the instructions contained in the `setup` and `loop` routines, respectively. (c with no subscript means statement from any routine).

Finally, Ω is the set of all possible program memory environment, i.e, the set of mappings from $I \cup O \cup X$ to values.

We first assume a deterministic big-step semantics for all program statements, represented by a relation of type $\mathsf{Command} \times \Omega \times \Omega$ noted $\downarrow$. We write $c, \omega \downarrow \omega'$ for a command c coupled with a memory state ω producing a new memory state ω'. The semantics is deterministic: given the same initial environment ω, two (syntactically) equal statements will produce the same updated environment ω'.

Then, we define a relation of type $\mathsf{Command} \times \Omega \times \mathit{stream}\ \tau_1 \times \mathit{stream}\ \tau_2$ noted $\Downarrow$ as the greatest relation closed backward by the rule:

$$\frac{c, \omega[I \mapsto i] \downarrow \omega' \qquad c, \omega', s \Downarrow s'}{c, \omega, i \cdot s \Downarrow \omega'_{|O} \cdot s'}$$

- $e \cdot s'$ decomposes a stream s into its first entry e and the rest of the stream s'.
- $\omega[a \mapsto b]$ is the mappings of ω with an extra mapping of a to b
- $\omega_{|O}$ is ω with its domain restricted to O

So, if an instruction c, a memory state w and an input stream $i \cdot s$ are in relation with (*produce*) an output stream $\omega'_{|O} \cdot s'$, it means ω' was created using c and ω with its input values being the head of the input stream. Then, we re-execute c, but with the new memory state ω', taking the tail of the input and output stream and so on.

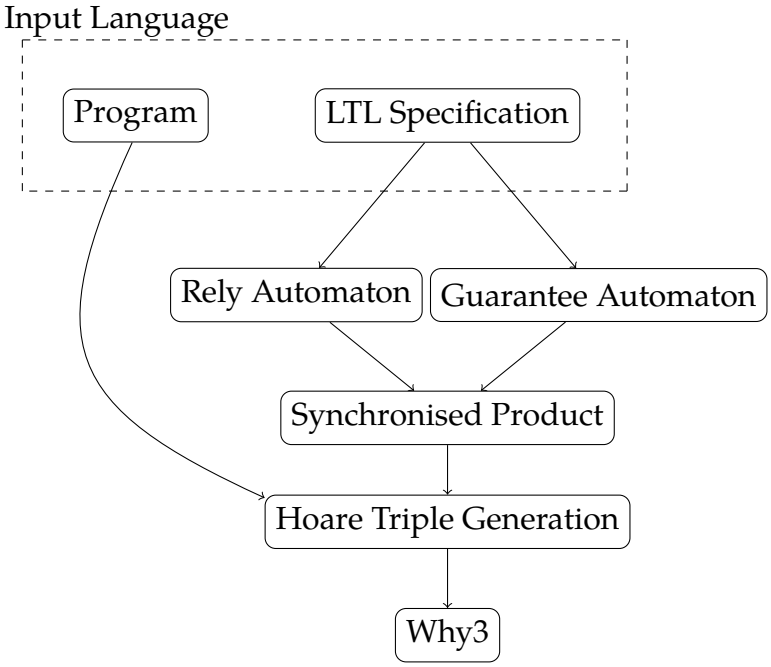
The semantics of p noted $[\![p]\!]$ is then:

$$[\![p]\!] = \{(s, s') \mid (\ c_{setup}, \omega_0 \downarrow \omega) \wedge (c_{loop}, \omega, s \Downarrow s')\}$$

That is, $[\![p]\!]$ can be defined as a set containing all possible infinite traces of its execution, where a trace is a pair of input stream s and output stream s'. To get the input and output streams, we start by executing the setup instructions c_{setup} on an initial state ω_0, producing a new state ω. Then, ω together with the instructions of the loop c_{loop} and an input stream s are used to feed our previously defined relation $\Downarrow$, allowing us to get s'.

3.2 Translation

Figure 3.2 – *Overview of Hardy's translation mechanism*



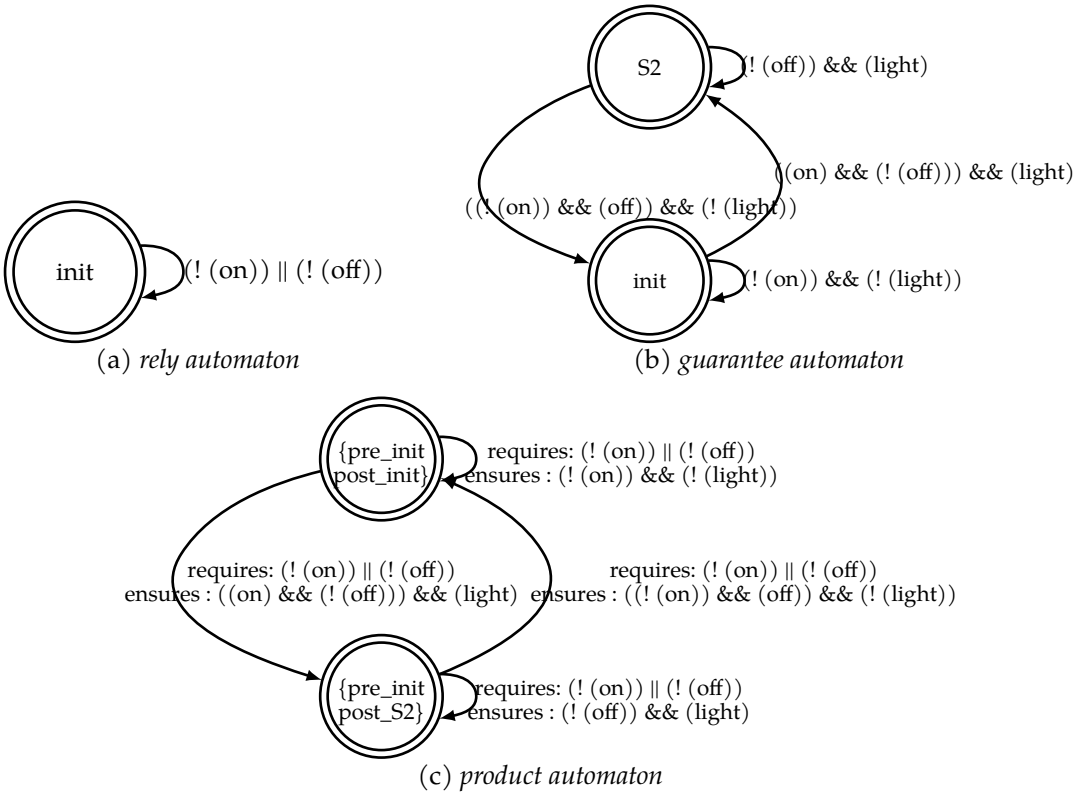
Hardy's translation mechanism has 4 stages (Figure 3.2):

1. Take the `relies on` and `guarantees` specifications and turn them into two Büchi automata
2. Compute the synchronized product of the automata
3. Use the newly created automaton together with the program's code to generate Hoare triples
4. Send them to Why3 for verification

Synchronized product

The rely automaton recognizes every valid stream of inputs the program assumes to receive and the product automaton recognizes every output the program is expected to produce. By taking the synchronized product of both automata, we get (unsurprisingly) an intersection of their respective languages, which means we

Figure 3.3 – *Construction of the synchronized product from the rely and guarantee automata*



(a) *rely automaton* (b) *guarantee automaton*

(c) *product automaton*

have an automaton that models all the possible valid outputs produced over time from the valid inputs received.

Looking back at the above example, we get Figure 3.3a from the `relies on` formula, and, from the conjunction of all `guarantees` formulas, we get Figure 3.3b. From these automata, the temporal operators disappear, and only atoms remains as the transition labels. In our case, the atoms of an LTL formula are FOL formulas over program expressions.

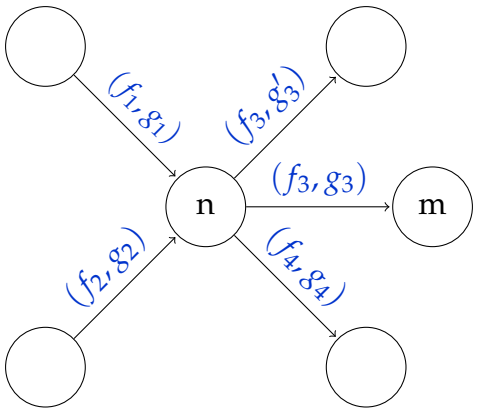
Then, we build the product automaton: for $\mathcal{A}_{rely} = \langle Q, \Sigma, \delta, q_0 \rangle$ and $\mathcal{A}_{guarantee} = \langle Q', \Sigma', \delta', q'_0 \rangle$, $\mathcal{A}_{rely} \times \mathcal{A}_{guarantee} = \langle Q \times Q, \Sigma \times \Sigma', \delta \times \delta', (q_0, q'_0) \rangle$ (Figure 3.3c).

However, contrary to what is generally done for model-checking, we do not join the formulas of each automaton. This is because the triple generation requires us to distinguish between the `relies on` and `guarantees` formulas, as we will now see.

Product Automaton

Consider a node n somewhere in the product automaton. This node has at least one successor m. The transition $n \xrightarrow{(f_3,g_3)} m$ means that the program received inputs satisfying $f3$ and must produce outputs satisfying g_3.

Additionally, n must also have a predecessor written $\cdot \xrightarrow{(f^{-1},g^{-1})} n$. In that case, g^{-1} represents the output produced and input received at the previous instant.



Generation of triples

For every node n within the automaton A and for each f where a transition $n \xrightarrow{(f,\cdot)} \cdot$ exists, we construct the triple

$$\{\lightning pre \wedge f\} \;\; c_{loop} \; \{post\}$$

where

$$pre = \bigvee g \mid \cdot \xrightarrow{(\cdot,g)} n \qquad post = \bigvee g \mid n \xrightarrow{(f,g)} \cdot$$

Here, $\lightning g$ represents the formula g where input variables are existentially quantified. This enables the preservation of information that relies on preceding input.

A special treatment is reserved for the initial node: $pre = \bigvee g \mid \cdot \xrightarrow{(\cdot,g)} n \vee g_{setup}$ where g_{setup} is the postcondition for the setup code.

To take into account the *setup* routine, we also build a triple $\{true\}\; c_{setup} \; \{g_{setup}\}$.

3.3 Correctness

The following theorems were proven using the Coq proof assistant [7]. We additionally showed that mentioning previous states of the program inside an atomic formula is sound (see section 6.2).

Theorem 3.3.1 *If all the triples generated for a program p with specification (P, ϕ, ψ) are valid, then*

$$s \in \mathcal{L}(A_\phi) \rightarrow \exists s'.\ [\![p]\!](s) = s' \wedge s' \in \mathcal{L}(A_\psi)$$

Corollary 3.3.1.1 *If all the triples generated for a program p with specification (P, ϕ, ψ) are valid, then*

$$\phi(s) \rightarrow \exists s'.\ [\![p]\!](s) = s' \wedge \psi(s')$$

3.4 Implementation

Hardy is implemented in OCaml, a multi-paradigm language, but functional at its core. Hardy's language parser is generated using Menhir [43]. The ocamlgraph library [12] is used to construct the product automaton.

As previously mentioned, Hardy relies on the automata generated by LTL2BA. LTL2BA outputs a description of the generated automaton to a text-file using *never claims*, a subset of the Promela specification language leveraged by Spin, a model-checking tool.

An example output is shown below (Listing 3.2).

Listing 3.2 – *Example LTL2BA output*

```
never { /* ([] ((! (f_711727821)) || (! (f_575586166)))) &&  [...] */
accept_init:
        if
        :: (!f_711727821 && !f_862012867) -> goto accept_init
        :: (f_711727821 && !f_575586166 && f_862012867) -> goto accept_S2
        fi;
accept_S2:
        if
        :: (!f_711727821 && f_575586166 && !f_862012867) -> goto accept_init
        :: (!f_575586166 && f_862012867) -> goto accept_S2
        fi;
}
```

To possibly simplify the guarantee automaton, we pass to LTL2BA a conjunction of the `relies on` and `guarantees` formulas. This is correct because our inputs are read-only. Thus, at the end of an instant, the rely formula still holds as inputs are unchanged.

While FOL formulas make up the atoms of our LTL formulas, LTL2BA only allows atoms to be strings. So, before sending the formulas to LTL2BA, we use an integer hashing of our FOL formulas based on their type (this is what the *hash* function of the *Hashtbl* module proposes). This way, two syntactically equal formulas will have the same atomic variable name. After getting back the automata, we substitute the hashes with their associated formula.

To build the synchronized product automaton, we start by merging the initial node of each automaton into a pair and use it to initialize a queue. Then, we repeatedly pop a pair out of the queue until it is empty, and we get the successors of each of the node making the pair. We iterate on the product of the successors and add new pairs to the queue, as long as we did not come across them already.

Then, to build the triples, we iterate over each node of the product automaton. The precondition is a disjunction of in-transitions's *guarantee* with an out-transition *relies* and the postcondition is the *guarantee* of the same out-transition. A triple is generated for each different out-transition *guarantee* because we must ensure every transition is possible (we must never get stuck, otherwise we breach the safety condition).

Once the triples are constructed, we use the Why3 API to translate our simple imperative code to WhyML, where each triple gets its own copy of the loop code (similarly for the setup code, but there is only one instant of the code). Only preconditions and postconditions generated changes.

At the moment, no type verification is made within Hardy because it is enough to use Why3's untyped API, as explained briefly in Theorem 2.1.2. The program terms constructed can then be checked and typed with Why3's own type checker.

Finally, we use the pretty printing facilities of Why3's Abstract Syntax Tree (AST) to output a `.mlw` file for easy debugging. Note that verification could occur directly with the Why3 API, but it is easier to work on the generated file for now.

Discussion 4

Contents

4.1 Current shortcomings

4.1.1 Counter-examples

One appeal of model-checking tools is their capacity at providing a counter-example as a trace of states leading to an incorrect behaviour. This feedback is important as it helps developers understand where the error comes from. While Symbolic Model-Checking (SMC) tools are also able to produce counter examples (as seen in section 5.4), they use a weak form of FOL which is decidable. For our case, failure to prove a triple doesn't necessarily mean a counter-example exists but simply that the SMT provers were not able to automatically find a proof. Thus, while the generation of a counter-example is not incompatible with our work, it remains to be seen whether it can be made systematic.

4.1.2 Trusted code base

Although we gave a proof our approach is correct, we haven't verified Hardy itself. In addition to trusting the automata generated by LTL2BA indeed represent the specification, we depend on the correctness of the translation and triple generation. Finally, we assume the external tool used to verify triples is also correct, so Why3 and its solvers in our case. Later on, Hardy will be formally verified in Coq.

4.1.3 Verifying Liveness

As previously mentioned, we restrict ourselves to only verify safety properties. To ensure it in practice, only *release*, *globally* and *next* operators are allowed. With this restriction, every path of the automaton must go through at least one accepting state. It then suffices to show there exists a finite run for every prefix of a word. However, if we want to support liveness, we must show a path eventually goes through an accepting state and does not indefinitely loop over non-accepting states. This will be explored in future work.

4.1.4 Referring to past values

Consider the following 2 examples:

Listing 4.1 – *delay with booleans*

```
input i : bool ;
output o : bool ;
variable x : bool ;
setup :
    ensures { x = false }
    x := false ;
loop :
    o := x ;
    x := i ;
```

Listing 4.2 – *delay with integers*

```
input i : int ;
output o : int ;
variable x : int ;
setup :
    ensures { x = 0 }
    x := 0
loop :
    o := x ;
    x := i
```

Both programs output the previous input received, but one uses booleans while the other uses integers.

Only the boolean version can be specified with our current tool. Its specification is given below:

```
(* the first output is always false *)
guarantees { not o }
(* the new state is always the last input *)
guarantees { G ( x = i ) }
(* if the current state is true,
    the output will be true at the next instant *)
guarantees { G ( x -> X o ) }
(* if the current state is false,
    the output will be false at the next instant *)
guarantees { G ( not x -> X (not o) ) }
```

Hence, for the boolean version, we enumerate the 2 possible values the input can take. Then, for each case, we tell what the output at the next instant will be. This obviously cannot be done for integers. What we would like to do is to universally quantify over the input: `guarantees{ G (forall a. i = a -> X(o = a))}`. However, we would need to find how to modify the automaton generation to account for it.

4.1.5 Completeness

We proved the (relative) correctness of our approach in section 3.3, that is, if all triples generated by Hardy from a program and its safety specification can be proven, the program is indeed correct. However, we haven't proven its completeness, that is, given any program and its safety specification, can we always prove its correctness this way?

4.2 Alternative ideas

4.2.1 Instrumentation

At first, the program was written directly within a loop. Ghost code was included around and inside the loop to gather a trace of the program that could be used to refer to previous values:

A program was composed of global variables contained within an instance of the `var_t` type and i/o vars with the `env_t` type. Then, at the end of the loop, a snapshot of the current i/o and global variables was created. This snapshot was then appended to a list representing the history. Next, we defined predicates to be able to mention the history:

- `at_time i x h`: the element at index `i` of history `h` is `x`
- `forall_i prop h`: `prop` must hold for every element of the history `h`
- `exists_i prop h`: there must exist an element of `h` for which `prop` holds
- `imm prop h i`: `prop` must hold at the `i`-th entry of the history `h`

To 'help' Why3 prove the different properties, we had some helping lemmas, for example:

- `at_time_hd`: if we have `at_time i x h` and `i = length(h)`, then `x = hd h`
- `at_time_tl`: if we have `at_time i x (x::h)` then we must also have `at_time i x h`

We eventually gave up on this solution because it was hard figuring out what helping lemmas Why3 needed. Also, we could only talk about the past.

4.2.2 LTL extended with binders

After experimenting with the automaton approach, there was still expressivity issues due to the lack of quantification within formula atoms. We tried to decorate some operators with binders, which can then be used across operators' imbrication: $G_a(a = \cdots)$ would universally quantify a and $G(X_a(a = \cdots))$ would make getting the previous value at each instant possible. Hence, when we wanted to write $G(\forall a.(i = a \rightarrow X(o = a)))$, we would now write $G_a(i = a \rightarrow X(o = a))$.

The idea was abandoned as it was not obvious how to derive a Büchi automaton from this syntax.

Related Work

5

Contents

The following sections summarize different reactive system verification tools that relates to Hardy. Each tool is described in terms of input language, specification language, verification procedure and what needs to be trusted.

5.1 Verification of Parameterized Reactive Systems

Cubicle [18] is a model-checking tool for proving safety of reactive systems which also generates a 'certificate of correctness' in Why3. Here, a reactive system is a single array-based transition system parameterized by the number processes that will execute it.

Language

Cubicle has its own specific input language: it describes an array-based system composed of multiple processes executing the same piece of code, but with no more than 1 process active at a time. This code is parameterized by a Process Identifier (PID) which uniquely identifies the current process running the code. This enables each process to have disjoint memory storage. As illustrated in Listing 5.1, the system first consists of a declaration of ground types (int, bool, reals),

user types (abstract and sum types) and arrays. Arrays are used to store process-dependent values. That is why they must always be indexed by variables of type *proc*, representing a PID. Instructions are restricted to sequences of global variable assignments. A group of such instructions is called an *action*. The state of the system evolves through transitions that describe how the process-dependent values of some PID change. A transition is composed of a *guard* and an action. A guard is a restricted form of FOL formula over the global variables that gives the conditions for the action to happen. Both the guard and the action are parameterized by a certain number of distinct PIDs. This language describes a non-deterministic system that chooses a transition instance whose guard holds true for the current state, update the state according the transition's action and repeat. Hence, it is more of a modelling language that cannot be executed as-is.

Listing 5.1 – *Example of a Cubicle program*

```
type state = Idle | Want | Crit
type data

var Timer : real
array State[proc] : state
array Chan[proc] : data
array Flag[proc] : bool


init(z) { Flag[z] = False && State[z] = Idle && Timer = 0.0 }

unsafe(x y) { State[x] = Crit && State[y] = Crit }

transition t (i j)
requires { i < j && State[i] = Idle && Flag[i] = False &&
           forall_other k. (Flag[k] = Flag[j] || State[k] <> Want) }

{
     Timer := Timer + 1.0;
     Flag[i] := True;
     State[k] := case
                 | k = i : Want
                 | State[k] = Crit && k < i : Idle
                 | _  : State[k];
}
```

Specification

Cubicle focuses only on safety properties with no notion of time. A global invariant named *unsafe* is represented by a standalone guard. This invariant differentiates between good and bad states: any good state must not make the invariant hold

and all bad states must do. As for the transitions, the invariant is parameterized by a certain number of PIDs to be able to talk about PID-dependent values.

Verification

The system is deemed safe if no bad state can be reached starting from an initial state. A backward-reachability algorithm is used for this purpose: it assumes such bad state exists and see if it can be reached by successive transitions from the initial state. This is expressed in FOL and checked with a specialized version of an SMT solver. Thus, Cubicle does SMC. If a bad state is indeed accessible from an initial state, the system is unsafe and a form of counter-example is represented as a series of transition that led to the bad state. When the system is safe, a Why3 certificate can be generated. It consists of a state-transition representation of the system using FOL, a synthesized global invariant and a set of goals showing its preservation.

Trust

As far as we know, Cubicle itself has not been certified. However, one can verify its output: when the system is unsafe, the counter-example trace can be used to reconstitute an erroneous run. When it is safe, the certificate can be used, assuming that it indeed represents the system and that Why3 and the solvers it calls can be trusted.

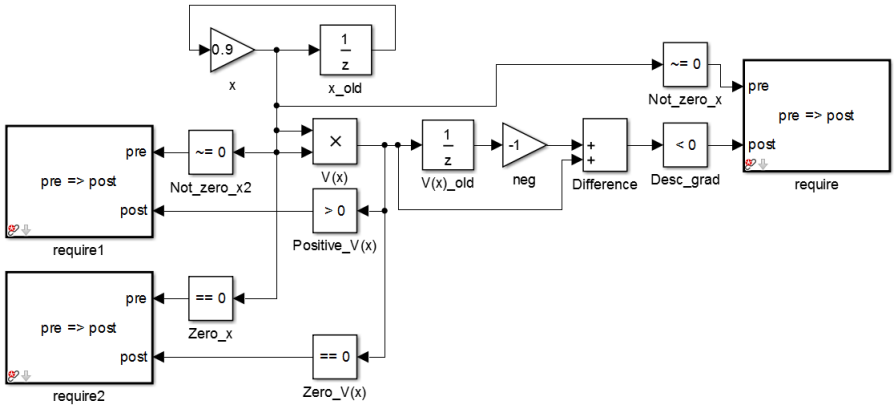
5.2 Verification of Simulink models

Language

Simulink is a MATLAB-based graphical programming software that does modelling, simulation and analysis of dynamic systems. Systems studied here are control systems, which are a type of reactive systems. Such system consists of *blocks* taking as input signals of other blocks and producing signals as output (Figure 5.1). Blocks are of different types: they can do calculations, signal routing, display values, input delay etc.

A Simulink model can be simulated, but it does not constitute an executable program. It can however be translated into a Lustre program, that can then be independently verified using another tool (see section 5.3).

Figure 5.1 – *A simulink model with 'require' blocks*



Specification

To specify a Simulink model, [3] introduces a new type of block called a *Require* block. They permit a form of local specification through precondition and postcondition. So, there is again no notion of time here. The block is an 'enabled subsystem', which is a system only active when a positive signal is present. Here, the positive signal is generated when the precondition provided holds. When it is the case, another custom block called an 'assert' block and contained within the *Require* block checks if the postcondition holds as well.

Verification

The verification process is based on Why3. The main idea is to have a library of Why3 theory for each type of block. A block theory consists of functions parameterized by an integer and providing numerical values (integers, booleans, or reals). They represent the value an input or output signal takes at a certain instant. An axiomatic description of these functions is provided, corresponding to what the actual blocks do. Then, a Why3 represention of the model is generated based on the AST obtained from the block connexions and making use of the associated block theories. However, the 'Require' block is not translated this way, but rather as a Why3 goal with an implication between the precondition and the postcondition. Finally, Why3 attempts solving all the goals using its external SMT solvers.

Trust

The translation of the Simulink model's AST into Why3 has most likely not been verified. In addition, it must be assumed the blocks' semantics is faithfully axiomatized. As usual, Why3 and its solvers must also be trusted.

5.3 Verification of Lustre programs

Language

Lustre is a declarative synchronous dataflow language generally used for reactive embedded systems. A Lustre program is made up of function declarations called *nodes* that transform streams of data. A node is a set of equations over the streams, where outgoing streams are expressed in terms of ingoing streams, possibly using intermediate streams for calculations. A node memory is the history of previous values of the streams up to a finite defined depth.

Listing 5.2 – *Example of a Lustre program*

```
-- true after x has been true once
node after(x: bool) returns (after: bool);
let
   after = x or (false fby after);
tel

node use_fby(x: bool) returns (res: bool);
let
  res = after(x);
tel
```

We now review two verifiers for Lustre program: Kind [25] and one of its recent successor Kind 2 [11].

Specification

Both tools only focus on safety properties as it is argued liveness isn't needed. This is because most Lustre programs are time-critical, so the number of instants before the appearance of an event is known. Thereby, this is a safety property that can be expressed as 'it must not take more than n instants for this event to appear'.

With Kind, properties are not expressed directly in Lustre. Rather, the program is expressed in FOL which is only then specified using quantifier-free formulas parameterized by instants. While such specification can be considered low-level, it at least enables a temporal description of the system.

Contrary to Kind, Kind 2 extends Lustre to be able to annotate nodes with local invariants or assume-guarantee contracts. Both are written using Lustre boolean-expression and can describe time-varying behaviour through a shallow embedding of past Linear Temporal Logic (pLTL) as Lustre nodes. An assumption in a Lustre contract is a predicate over the node's inputs and a guarantee is a predicate over both the node's inputs and outputs. Contracts are written in CoCoSpec [10], a mode-aware specification language, whose principles aren't just restricted to Kind 2. A *mode* influences the treatment of inputs according to past events and is largely used in specification documents. In general, they are encoded as implications between situations and expected behaviours, resulting in a contract that can potentially lose mode-specific information. A CocoSpec contract, however, contains a set of *mode declaration* in addition to the *assume* and *guarantee* predicate of standard contracts. A mode declaration is a set of *require* and *ensures* statements that are well-typed boolean Lustre expressions. Intuitively, the *require* statements define the conditions for the mode to be active and the *ensures* statements describe what the node does when in that mode. When all mode's *require* expressions hold, it must follow that all mode's *ensures* expressions hold as well. Additionally, if the contract holds, at least one of the modes must hold. Otherwise, it means there is an underspecification, as the system can be in an unspecified mode.

Verification

Both Kind and Kind 2 make use of an idealized version of Lustre's denotational semantics as quantifier-free FOL formulas forming a state-transition system. It is called an idealized version because data-types like integer and reels are mapped to their mathematical, infinite precision counterpart. While this mean underflow, overflow or rounding errors can't be taken into account, it makes the system decidable, and efficiently so. For this purpose, k-induction, a generalization of induction; is utilized. It consists of proving the property holds up to the first k consecutive instants (base case) and that if it holds for any k consecutive instants, it must also hold for the k+1 one (inductive step). Then, k is increased until a counterexample is found, or an upper limit deduced from the property is reached. The technique is sound but incomplete as some properties may require an infinitely large k. To increase the number of properties provable with this system, 3 other techniques

are used: path-compression, structural abstraction and refinement. The reasoning over the semantics of Lustre is delegated to external SMT provers such as CVC3 and Yices. Without the different optimizations proposed over k-induction, both tools behaves like a bounded symbolic model-checker: it checks only traces up to a certain length. In case a property does not hold, Kind and Kind 2 produce a counter-example as a trace of execution leading to an incorrect state.

Although its principle is the same, Kind 2 brings a number of improvements, in addition to better specification support: because it is an improved and rewritten-from-scratch version of PKind, which already improved upon Kind, it first benefits from the performance increase of PKind's parallel k-induction. Additionally, Kind2 enables *compositional reasoning* by proving the properties of each node separately, assuming every not-yet-proved contract holds for called nodes and reusing results otherwise.

Trust

Kind is not formally verified, so its translation mechanism must be trusted as well as how it uses the external SMT solvers to verify the system. Additionally, because it uses approximative semantics to model a Lustre program, a program can be considered safe even though it exhibits an unsafe behaviour due to an overflow, an underflow or a rounding error.

While Kind 2 has the same issues, it can generate a proof certificate [39]: it is a k-inductive invariant formula and a fixed k-value that implies all proven properties of the system. Then, an external SMT solver can be used to verify it is indeed k-inductive and does imply the system's properties. This shifts the trust from Kind 2 to the potentially less-complex external solver. However, this still relies on the semantic preservation of the translation. A partial formal solution is proposed by showing Kind 2 representation is *observationally equivalent* to the one of an independent tool used on the same input.

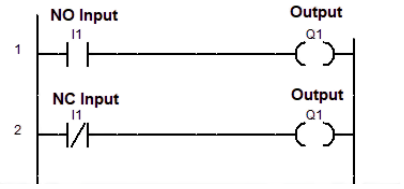
5.4 Verification of Ladder programs

Language

Ladder is the first programming language tailored towards Programmable Logic Controllers (PLCs): industrial computers used to automate the control of manufacturing processes. A Ladder program computes outputs from inputs synchronously

and has an internal memory, similarly to Hardy. Because its initial purpose is to replace the use of hardwired relay circuits, but to keep the same fundamentals, as not to require technician training, a Ladder program is a circuit diagram. Multiple circuits are represented in parallel each by a line. In its most basic form, a line connects a contact (input) to an actuator (output). The contact acts as a push buttons that opens or closes the circuit, firing the actuator. In place of the actuator, an instruction can be called, modifying the internal state of another connected PLC.

Figure 5.2 – *A Ladder circuit diagram*



Specification

In [14], no specification is provided by the user. Indeed, contrary to Hardy, the focus is not on the functional correctness of the program but rather the absence of certain programming errors. Hence, the user doesn't provide any kind of program specification. Moreover, the cyclic aspect of the program is not accounted for, as only a single instant is modelled, potentially producing false-positives. These shortcomings are addressed by [36], where a form of temporal specification, called a *timing chart*, is used to verify the program's behaviour over multiple instants.

Timing charts

Timing charts as used in [36] consist of a sequence of consecutive instants. An instant is represented by inputs and outputs values, capturing the expected instantaneous state of the program for this particular instant. When an instant has at least one value that differs from the previous one, it is called an *event*. Consecutive instants for which no value changes are called a *stable state*. While an event happens at a certain instant, a stable state is maintained over an arbitrary number of instants. Hence, a timing chart is a succession of events separated by stable states. It is important to note that while the timing chart approach enables a form of temporal specification, only safety properties can be expressed. Indeed, they enforce knowl-

edge of events' duration and, even though stable states duration is unknown, they are not required to end at some point.

Verification

Runtime errors

To verify ladder programs, [14] leverages Why3. For this purpose, a library of formalized ladder instructions was developed: each instruction is implemented as a verified WhyML function with preconditions and postconditions. The program internal state is modelled using references that are passed as additional arguments to the functions. The program is then translated into a Why3 module made up of boolean variables declarations. These variables are defined in terms of boolean formulas combining other variables and calls to the instruction library. It is then checked whether any instruction's precondition is violated using a SMT solver capable of generating counter-examples. If a violation occurs, the provided counter-example is used to feed the model's initial values, enabling the simulation of an erroneous execution. The collected execution information is used to produce a user-friendly error report, presented graphically over the circuit-diagram.

Timing charts

Due to the cyclic nature of the program, [36] reuses multiple times the single-instant declaration generation done in [14], called the *body* of the program. Instead of a single body, the model is now a sequence alternating between the body and a while-loop containing the body. The while-loop models a stable state whose exit condition is a variable which is assigned a random boolean at each iteration. This models the fact the end of a stable state is unknown. The precondition of a body is the input values that triggers a new event. As a stable state follows the event, which is modelled as the while loop, the postcondition takes the form of a loop invariant. This loop invariant must show the body always produces the correct outputs from the inputs that triggered the new event, as indicated by the chart.

One difficulty brought by this updated verification is that the loop invariants gathered from the timing chart are generally not sufficient. Indeed, additional internal memory devices are used in a ladder implementation to respect the chart, but the behaviour of those devices are of course not present in the chart. While the invariants can be manually strengthened, it is a difficult and here repetitive task that would hinder the adoption of the tool for Ladder developers. Instead, additional

loop-invariants are inferred from the body itself, using a modified version of a Why3 plugin that uses abstract interpretation over boolean and integers.

Trust

Both tools provide a user-friendly counter-example to be checked upon verification failure. However, a program considered safe requires trusting the translation to a WhyML program, ensuring the formalized instructions are correctly modelled and have confidence in Why3 and its external solvers.

5.5 Summary of the approaches

As summarized below, the reviewed approaches attempt to verify either non-executable models ([18] [3]) or actual code ([25] [11] [14] [36]). Only safety properties are considered. While some of them ([36] [11]) have a notion of time, they do not directly make use of a time-friendly and program-independent logic language like LTL. Most of the tools express the system in FOL formulas that are sent to SMT solvers. Only [14] and [14] employ DV. Finally, none of the tools presented here have been formally verified. However, they can generally provide counter-examples on failure or proof certificates on success.

Article	Input	Specification				Verification
		S [a]	L [b]	T [c]	Description	
[18]	M [d]	yes	no	no	function contract + invariants	SMC
[3]	M	yes	no	no	local assertions	SMC
[25]	P [e]	yes	no	yes	parameterized quantifier-free formulas	SMC
[11]	P	yes	no	yes	shallow embedding of pLTL + mode-aware function contracts	SMC
[14]	P	no	no	no	_	DV
[36]	P	yes	no	yes	timing chart	DV

[a] Safety
[b] Liveness
[c] Temporal Specification
[d] Model
[e] Program

Conclusion and future work

6

Contents

6.1 Conclusion

We presented Hardy, a tool that leverages DV to ensure the safety of synchronous reactive programs. Programs are specified using LTL and verified by an approach that draws inspiration from model checking by manipulating the automaton representation of the specification. Yet, it differs from regular model checking by producing Hoare-style proof obligations that are then checked using an external DV tool. We showed the correctness of our approach in the Coq proof assistant and have a working proof-of-concept that has successfully been applied to a few examples.

6.2 Future Work

Past predicates

So far, we can see our predicates as parameterized by the program variables for a certain instant. However, as demonstrated in 4.1.4, this is limiting. Lifting this restriction involves generalizing predicates: their parameters are not just variables denoting their last value, but history of values associated to those variables. Each parameter can be seen as a vector of values indexed from the first instant of the program up to the currently reached state. We extended our proof of correctness to show this new form of predicate is compatible with our approach. However,

we must be careful with this new level of expressivity: if used as-is, LTL could be encoded inside the predicates, which would defeat the purpose of using it in the first place. By restricting each variable to at most one instant quantification, we ensure LTL cannot be encoded as it requires quantification of at least 2 instants. This way, one can talk about all the previous values of a variable, a certain previous value or the existence of a previous value.

Public and private specification

In our current work, temporal formulas are about the program's inputs, outputs and state. However, even though talking about the state is a necessity to correctly specify the program, we want to see the program as a black-box: we shouldn't have the specification tell us information about its internal variables. We want a specification describing the program in terms of its inputs and outputs.

For that purpose, what we plan to do is to have two kinds of specification: a private one ϕ_{priv} that includes the state and a public one ϕ_{pub} that does not mention it. We first prove the correctness of a program using ϕ_{priv}. Then, we check if $\phi_{priv} \Rightarrow \phi_{pub}$. From a language perspective, we must check if $\mathcal{L}_{\phi_{pub}} \subseteq \mathcal{L}_{\phi_{priv}}$. As for regular languages, ω-regular languages are closed under union, intersection and complementation. So, $\mathcal{L}_{\phi_{pub}} \subseteq \mathcal{L}_{\phi_{priv}} \Leftrightarrow \mathcal{L}_{\phi_{pub}} \cap \overline{\mathcal{L}_{\phi_{priv}}} = \emptyset$. Thus, we build the synchronized product of $\mathcal{A}_{\phi_{pub}}$ and $\overline{\mathcal{A}_{\phi_{priv}}}$ and check for emptiness.

Validator for Büchi Automata

As mentioned in subsection 4.1.2, Hardy depends on the validity of the provided automata. That is, a word is accepted by an automaton if and only if it is accepted by its correspond LTL formula. There are two approaches to this dependency:

- Verify the tool that generates the automaton
- Create a validator that checks if a formula's language is the same as the automaton's

The first approach is very time-consuming and proofs must be adapted each time a new algorithm is developed. It has nevertheless been achieved (e.g. [17]) for the basic algorithm.

With the second approach only the validator must be verified, which is assumed to be a much simpler task. Then, any tool can be used as long as its output is checked with the validator.

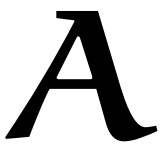
The latter approach is hence what we intend to tackle.

Other ideas

We want to explore the possibility of directly using WhyML as Hardy's input language and how to have multiple processes that can be composed. Ultimately, we want to support richer constructs that will allow us to verify programs written in SaIL, our rust-inspired reactive language [1].

Appendices

Additional example



Listing A.1 – *a metronome that ticks every 3 beats*

```
output
  beat : int
  accent : bool
;

guarantees { G ( [beat = 1] <-> [accent]) }

guarantees { G (
                [beat = 1] -> X [beat = 2]
                              & X X [beat = 3]
                              & X X X [beat = 1] ) }

setup :
    ensures { [beat = 1 & accent] }

    beat := 1;
    accent := true;

loop:
    if beat = 3 then
        beat := 1;
        accent := true;
    else
        beat := beat + 1;
        accent := false;
    end
```

This Hardy program (Listing A.1) represents a metronome. It counts 3 beats per

measure, which it outputs using the beat integer. At the start of a new measure, the number of beats is reset, and a signal is emitted with the accent output.

What we want to ensure is that the accent is signalled if and only if we are at the measure's first beat. Furthermore, we want to show that it starts at beat 1, counts up to 3 included and then starts back at 1 and so on. To translate this last requirement, we say that each time we are on the first beat at an instant n, then it must follow we are on the second beat at $n + 1$, third beat at $n + 2$, and finally back to the first beat at $n + 3$.

The product automaton generated by Hardy contains 8 states and 12 transitions, producing 9 Hoare triples that Why3 manages to solve automatically.

Bibliography

[1] The sail programming language. `https://github.com/sail-pl/SAIL`. (page 64)

[2] Rajeev Alur, Kousha Etessami, Salvatore La Torre, and Doron Peled. Parametric temporal logic for "model measuring". *ACM Transactions on Computational Logic*, 2(3):388–407, 2001. doi: 10.1145/377978.377990. (page 28)

[3] Dejanira Araiza-Illan, Kerstin Eder, and Arthur Richards. Formal verification of control systems' properties with theorem proving. In *2014 UKACC International Conference on Control (CONTROL)*, pages 244–249. IEEE, 2014. doi: 10.48550/arXiv.1405.7615. (page 54, 60, 61)

[4] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1998. doi: 10.1017/CBO9781139172752. (page 8)

[5] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, 1959. (page 5)

[6] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *WP Plug-in Manual*, 2010. URL `http://frama-c.com/download/frama-c-wp-manual.pdf`. (page 20)

[7] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development*. Springer Berlin, Heidelberg, 2004. doi: 10.1093/comjnl/bxh141. (page 3, 45)

[8] F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991. doi: 10.1109/5.97299. (page 1, 24)

[9] Bernard Carré and Jonathan Garnsworthy. Spark—an annotated ada subset for safety-critical programming. In *Proceedings of the Conference on TRI-ADA'90*, pages 392–402, 1990. doi: 10.1145/255471.255563. (page 19)

[10] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. Cocospec: A mode-aware contract language for reactive systems. In *International Conference on Software Engineering and Formal Methods*, pages 347–366. Springer, 2016. doi: 10.1007/978-3-319-41591-8_24. (page 56)

[11] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer, 2016. doi: 10.1007/978-3-319-41540-6_29. (page 55, 60, 61)

[12] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ml functors. *Trends in functional programming*, 8: 124–140, 2007. (page 45)

[13] John Corcoran. Schemata: The concept of schema in the history of logic. *Bulletin of Symbolic Logic*, 12:219–240, 2006. doi: 10.2178/bsl/1146620060. (page 14)

[14] Denis Cousineau, David Mentré, and Hiroaki Inoue. Automated deductive verification for ladder programming. *arXiv preprint arXiv:1912.10629*, 2019. doi: 10.48550/arXiv.1912.10629. (page 58, 59, 60, 61)

[15] S. Demri and R. Lazic. Ltl with the freeze quantifier and register automata. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 17–26, 2006. doi: 10.1109/LICS.2006.31. (page 28)

[16] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verification of Rust programs. In *International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022. doi: 10.1007/978-3-031-17244-1_6. (page 19)

[17] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 463–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-39799-8_31. (page 63)

[18] Nicolas Féral and Alain Giorgetti. A gentle introduction to verification of parameterized reactive systems. In *Formal Methods Teaching Workshop*, pages 34–50. Springer, 2023. doi: 10.1007/978-3-031-27534-0_3. (page 51, 60, 61)

[19] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-37036-6_8. (page 3, 18)

[20] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1980. doi: 10.1145/567446.567462. (page 2)

[21] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi: 10.1007/3-540-44585-4_6. (page 35)

[22] Thierry Gautier, Paul Le Guernic, and Loic Besnard. *Signal: A declarative language for synchronous programming of real-time systems*. Springer, 1987. doi: 10.1007/3-540-18317-5_15. (page 1, 24)

[23] Gerhard Gentzen. Untersuchungen über das logische schliessen in math. *Zeitschrift39*, pages 176–210, 1935. doi: 10.1007/BF01201353. (page 9)

[24] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV*, pages 3–18. Springer US, 1996. doi: 10.1007/978-0-387-34892-6_1. Series Title: IFIP Advances in Information and Communication Technology. (page 32)

[25] George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9. IEEE, 2008. doi: 10.1109/FMCAD.2008.ECP.19. (page 55, 60, 61)

[26] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. doi: 10.1109/5.97300. (page 1, 24)

[27] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, 1989. doi: 10.1007/978-3-642-82453-1_17. (page 1)

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. doi: 10.1145/363235.363259. (page 7)

[29] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. (page 13)

[30] Johan Anthony Wilem Kamp. *Tense logic and the theory of linear order*. University of California, Los Angeles, 1968. (page 28)

[31] F. Kröger and S. Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-68635-4. (page 1)

[32] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, pages 125–143, 1977. doi: 10.1109/TSE.1977.229904. (page 1)

[33] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. (page 18)

[34] Martin Leucker and César Sánchez. Regular linear temporal logic. In *Theoretical Aspects of Computing – ICTAC 2007*, pages 291–305, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-75292-9_20. (page 28)

[35] Leo Louis. Working principle of arduino and using it as a tool for study and research. *International Journal of Control, Automation, Communication and Systems (IJCACS)*, 1(2):21–29, 2016. doi: 10.5121/ijcacs.2016.1203. (page 3)

[36] Cláudio Belo Lourenço, Denis Cousineau, Florian Faissole, Claude Marché, David Mentré, and Hiroaki Inoue. Automated verification of temporal properties of ladder programs. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 21–38. Springer, 2021. doi: 10.1007/978-3-030-85248-1_2. (page 58, 59, 60, 61)

[37] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing - PODC '87*, pages 205–205. ACM Press, 1987. doi: 10.1145/41840.41857. (page 28)

[38] Nicolas Markey. Temporal logic with past is exponentially more succinct. *Bulletin-European Association for Theoretical Computer Science*, 79:122–128, 2003. (page 27)

[39] Alain Mebsout and Cesare Tinelli. Proof certificates for smt-based model checkers for infinite-state systems. In *2016 Formal Methods in Computer-Aided*

Design (*FMCAD*), pages 117–124. IEEE, 2016. doi: 10.1109/FMCAD.2016.7886669. (page 57)

[40] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016. doi: 10.1007/978-3-662-49122-5_2. (page 18)

[41] Gordon D Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 1981. doi: 10.1016/j.jlap.2004.05.001. (page 7)

[42] François Pottier. Correct, fast LR(1) unparsing. In *35es Journées Francophones des Langages Applicatifs* (*JFLA 2024*), 2024. (page 7)

[43] François Pottier Yann Régis-Gianas. Menhir reference manual, 2016. `https://gallium.inria.fr/~fpottier/menhir/manual.pdf`. (page 45)

[44] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. Reactive programming: A walkthrough. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 953–954, 2015. doi: 10.1109/ICSE.2015.303. (page 1)

[45] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi automata for LTL model-checking verified in Isabelle/HOL. In *22nd Intl. Conf. Theorem Proving in Higher-Order Logics* (*TPHOLs 2009*), LNCS, pages 424–439, Munich, Germany, 2009. Springer. doi: 10.1007/978-3-642-03359-9_29. (page 33)

[46] Robert D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, 1976. doi: 10.1145/360303.360308. (page 7)

[47] Pierre Wolper. The tableau method for temporal logic: an overview. *Logique et Analyse*, 28(110):119–136, 1985. (page 32)