Formal Verification for Autopilot – Preliminary state of the art

Baptiste Pollien, Xavier Thirioux, Christophe Garion,
Gautier Hattenberger and Pierre Roux

June 10, 2021

Abstract

This document is a preliminary state of the art for the formal verification of the autopilot of an Unmanned Air Vehicle (UAV). We will first present UAV autopilots and more specifically the Paparazzi autopilot developed at ENAC which will be our case study. We then present which properties could be verified and on which representation of the autopilot (source code, model). A more complete state of the art of current formal methods will be then detail and focus on deductive methods, abstract interpretation, model checking and proof assistants. Finally, some immediate perspective for the thesis are proposed.

Contents

1 Introduction

The present document is a preliminary state of the art report written between October, 1st 2020 and November, 30th 2020 for the PhD thesis entitled “Formal verification of an autopilot”. This PhD thesis is done by Baptiste Pollien in the context of the CONCORDE project funded by DGA which aims at proposing methodology and tools for the design and development of Unmanned Air Vehicles (UAV).

The PhD thesis is supervised by Prof. Xavier Thirioux (ISAE-SUPAERO) and co-supervised by Dr. Christophe Garion (ISAE-SUPAERO), Dr. Gautier Hattenberger (ENAC) and Dr. Pierre Roux (ONERA).

Why use formal verification? Program verification is usually done by *testing* programs using predefined inputs and outputs, but citing Edsger Disjktra [19]:

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Formal methods are mathematical theories, techniques and tools to formally prove the correctness of programs or systems. In the context of the thesis, we will tackle the formal verification of the Paparazzi UAV autopilot developed at ENAC. Even if formal methods are powerful tools and are currently used in critical systems industry (aerospace [41], automotive, medical, or cybersecurity [36]), using them on industrial use cases can be tricky and often needs not only to develop tools, but also the theories behind them.

The present report is organized as follows. Part I will briefly present UAVs and particularly the Paparazzi autopilot architecture. It will also present which kind of formal verification can be done on such an architecture. Part II will go deeper into formal methods, namely deductive methods, abstract interpretation, model checking and proof assistants. Finally, Part III will present the immediate perspective of the PhD thesis.

Part I
UAV

An *Unmanned Aerial Vehicle* (*UAV*) or also called drone is an aircraft for which there is no human pilot on board. Currently, all drones are controlled more or less directly by humans. Indeed, UAVs can be controlled directly by a remote controller or it can receive missions from a ground base that it will achieve autonomously thanks to its autopilot software.

UAVs were originally developed by the army to be sent to risky missions instead of humans in order to protect their lives. Now, drones have become democratized and we can find them for law enforcement, agriculture, construction or for recreative purposes like taking photos or videos. There are also future projects of delivery drones for medical supply or public transport. In order to be able to use drones for these large-scale projects, it will be necessary to drones that can be autonomous and *reliable* i.e., drones that have been tested and validated rigorously.

Section 2 will present the UAVs autopilots focusing on Paparazzi (an autopilot developed at ENAC and that will our case of study). Section 3 will introduce the different properties and development methods that can be verified in embedded and critical systems such as UAVs autopilots.

2 Autopilots

The goal of *autopilots* is to allow UAVs to perform entire missions in total autonomy, without human interventions with a controller. The missions are parameterized by the operators using ground control stations. The autopilot then controls the UAV to complete the tasks asked. It exists many open-source autopilots and among the main ones, we can cite: Ardupilot, PX4, Paparazzi or LibrePilot. These autopilots are still under development and have a focus on autonomy. They provide several versions or have a modular architecture in order to support different types of drones: rovers, helicopters, quadcopters, hexacopters, planes, vertical take-off planes…

In the present report, we focus on the Paparazzi autopilot.

2.1 Paparazzi

Paparazzi [33] is an open-source autopilot (under GPL license) whose development started in 2003 at ENAC. It is mainly designed for autonomous flight but supports manual control. It has also been developed to support different types of drones as presented earlier, and it is able to control multiple UAVs simultaneously from a unique system.

Paparazzi have a list of built-in modes, covering most of the usual needs. A dedicated control stack is called for each mode, although it is possible to choose the control loop being used at the moment. However, an experimental mode allows the implementation of a custom autopilot state machine. It is then possible to change or extend the number of modes, or even run several parallel control loops[1].

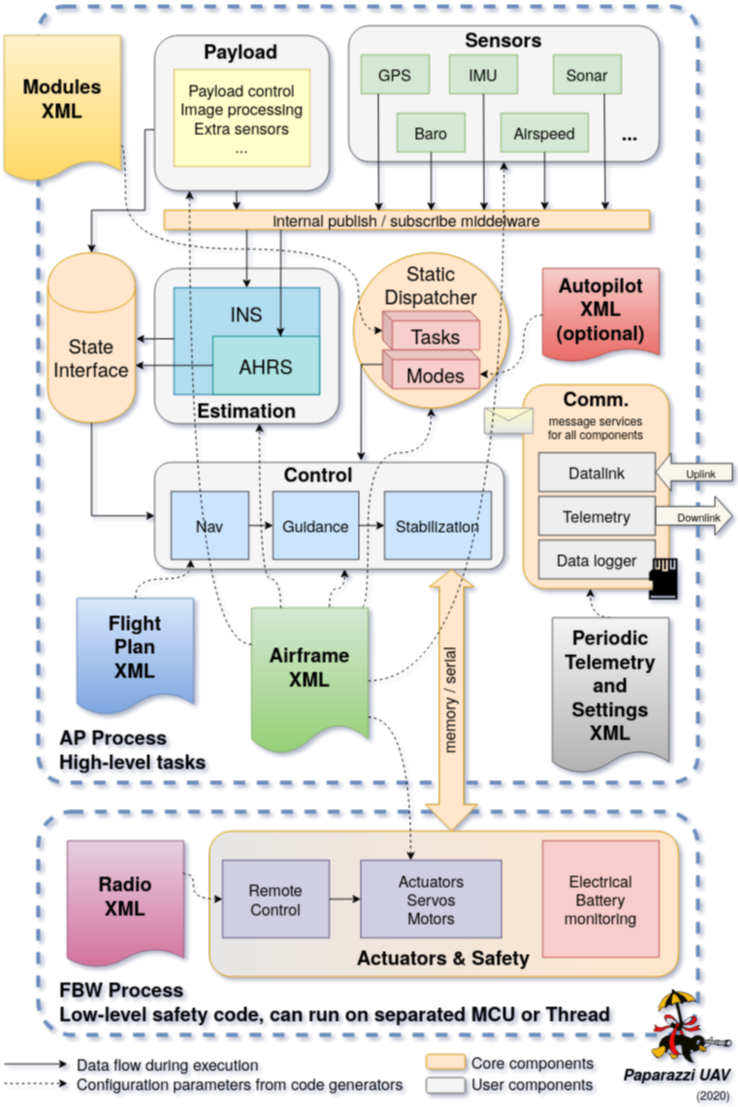


Figure 1: Architecture of the Paparazzi autopilot (embedded code)

As shown on Figure 1, these components are separated between critical (such as AHRS, IMU, INS and GPS) and non-critical components, with statically define periods in a sequential order. Although, efforts are being made for a transition to a real-time operating system (RTOS) based solution. The latest operating system used is based on ChibiOS (an efficient and light RTOS for microcontrollers). The main part of the autopilot is handled in a single thread, but most of the low-level drivers have their own threads, with either high priority (communication with internal sensors) or low priority (Secure Digital (SD) logging) compared to the autopilot task. It is also possible to create threads for certain heavy tasks related to payload management or parameter estimation in dedicated modules. This ensures a proper

[1] See http://wiki.paparazziuav.org/wiki/Autopilot_generation for more details

timing for the core autopilot control stack.

In addition, the data between the sensors and the estimation filters are managed by a publisher/subscriber mechanism, where the estimated state is published in a blackboard type interface for the control and navigation loops. Conversion between the different attitude or position representations (Euler, DCM, quat, lat/lon, ecef...) are automatically handled by this state interface.

The control loop is a core attitude control loop with the possibility of other control loops to execute on top of this one with slower rates. However, the navigation layer is based on a specific flight plan language (based on XML files generating C code) allowing complex mission description[2]. It is also possible to use a dynamic task-based approach[3]. Both approaches are based on built-in flight patterns like "go to point", "fly segment" or "around circle", but can be extended with custom navigation patterns.

Paparazzi provides a simplified way to add modules without having to know the functioning of other modules, by using standardized XML file descriptions. The developer still needs to know how functions are called in other modules so that it can interact with them. It is also possible to create interaction between modules by using the internal publish/subscribe software bus.

3 Verification

Generally, the development of a system or a program can be divided into 3 parts:

1. **Specification**, i.e. definition of functional needs (what are the tasks that the program has to realize) and the guarantees wanted in terms of safety and security. Material constraints can also be specified in particular in the context of the development of embedded systems.

2. **Implementation**, i.e. transformation of the specification into code using a programming language.

3. **Verification**, i.e. use of different testing methods in order to verify that the implementation corresponds to the specification and there are no unexpected behaviors.

Verification is an important step that often takes time. Usually, engineers and developers test different set of inputs and then verify that the system has the behavior defined by the specification without any errors. The inputs used in testing are often a small subset of all the possible inputs and it is not possible to test all the cases, except for very simple systems. This is generally sufficient for the majority of mainstream systems or software to verify the absence of functional bugs.

However, this method does not ensure that there are no bugs nor errors. Certain domains, especially for embedded and critical systems, need the guarantee of total absence of bugs. For example, we do not want an autopilot for drones to crash during a flight due to a division by 0 or a null pointer dereferencing. These errors are called *Runtime Errors* (RTE) as they occur

[2] See `http://wiki.paparazziuav.org/wiki/Flight_Plans` for the documentation of flight plans.

[3] For more details on missions, see `http://wiki.paparazziuav.org/wiki/Mission`.

during the execution of the program. Generally, such errors cause unexpected behaviors of the program or, in the most serious cases, the system might stop.

Another type of errors that embedded and critical systems want to avoid is unexpected behavior compared to the specification due to the implementation. These problems might be related to a developer implementation error or a misunderstanding of the specification.

There are different type of properties to specify absence of RTE and that the specification is respected. There are also different methods to verify these properties on a program by not just analyzing some possible executions, but rather the whole set of the possible executions. The section 3.1 presents the type of properties that can be verified based on code analysis methods.

The misunderstanding of the specification is a common problem on projects due to the use of natural language as a specification language, which introduces ambiguity. To avoid this problem, new development methods appear, in particular with the utilization of *models*. Models represent the system and its architecture and are defined with formal or semi-formal languages that reduce ambiguity. This method, called *Model Based Systems Engineering* (MBSE) is presented in section 3.2.

Finally, when the model is defined and correct, there are tools that generate the code directly from the model to avoid implementation errors from the developer. They are presented in section 3.3.

3.1 Code analysis

Code analysis is a family of methods that can verify properties on a program. These methods can analyze the code *statically* i.e., without executing it, or *dynamically* i.e., during execution of the code. In this report, we will focus on *static analysis* methods and particularly on *Deductive Methods* in section 5) and *Abstract Interpretation* in section 6. These methods are used to verify properties that can be divided into two types:

- **General properties**: this kind of property describes errors or behaviors that any system wants to avoid. These properties generally state the absence of RTE. They can be about errors that can stop the application (division by 0, null pointer dereferencing) or unwanted behavior after an integer overflow where the programs might give wrong results. Another example of unwanted behavior is when the system becomes unresponsive. This can happen when a process is blocked in an infinite loop (*livelock*) or if two threads are interlocked (*deadlock*).

- **Specific properties**: these properties state the expected behavior of the system. They are generally given by the specification. For example, a property for an UAV might be: “The UAV should take off only if the battery level is sufficient to do at least an emergency landing”.

Properties can also be classified into groups. For instance, there are properties about *safety* and properties about *liveness*. Safety properties define error states of the system that should never be reachable. An example might be that the autopilot of a quadcopter drone should not stop the four rotors when the drone is flying. Liveness properties refer to all states of the system that should be reached during the execution. For example, if a drone has been ordered to take off and all the necessary conditions to fly are satisfied, then the drone will eventually fly.

Remark: one type of property that has not been discussed and which will not be detailed in this document are the properties about numerical computation. Indeed, as real numbers are usually represented with floating-point numbers in computers, the algorithms have to be adapted to reduce and control errors generated by finite precision [34]. In addition to the algorithms used, it is important to consider that the compiler may also introduce computation errors due to optimization choices [46]. These computation errors generated by a compiler are also a problem for verification of proofs of mathematical theorems assisted with computers (see section 8 for the presentation of proof assistants). Thus, some rigorous verification methods have been developed to provide strong results about floating-point arithmetic [53].

3.2 MBSE

Model Based Systems Engineering (MBSE) is a development approach that uses models to define the specification of a system. The models can describe different levels of complexity of the system: the different software and hardware components and how they communicate, but also different threads that can evolve in parallel etc. These models can be represented using semi-formal languages (as UML or SysML) or formal languages (for example, finite state machines or Petri nets). These representations offer an accurate overview of the project without any ambiguity.

In MBSE, in addition to the description of the models, different properties that the systems should satisfy can also be defined. These properties are similar to the type of properties defined in the previous section (*general* and *specific* properties, *safety* and *liveness* properties). In order to reduce ambiguity, natural language is avoided and formal languages are rather used. *Temporal logics* are formal languages used to specify models with properties that describe the dynamic behavior of the system (see section 7.1). Another formal language that can be used is *Floyd-Hoare logic*, used mainly to specify imperative programs with contracts. This logic allows to specify the expected output of programs only if the inputs respect some given properties (see section 5.1).

The definition of models is correlated to the definition of properties expected on the system. Indeed, the model should verify the different properties on the system. There are different tools that can formally verify if a model satisfies some given properties. For instance, Model Checking allows to verify temporal properties on systems modeled with automata or Petri Nets (see section 7).

Remark: the term “model” in Model Checking differs from “model” in MBSE i.e., an abstract representation of the systems. Indeed, in Model Checking, models represent generally the set of all possible executions of the system. The verification of properties on a model corresponds to the verification that all possible executions of the systemwill satisfy the property.

Once the model is defined and verified, the next step is to implement it. The implemented program must follow the specification of the model in order to ensure that the properties are always respected. There are different methods to guarantee that the program matches the model. One of them is to use deductive methods when the specification uses contracts and Floyd-Hoare logic to define the properties (see section 5). Another technique, that will be presented in the next section, is to generate a program directly from a model.

3.3 Code generation

Code generation is a process that translates a model into source code. For embedded systems, this process often generates C or Ada code, two programming languages mainly used to implement such systems. When possible, every MBSE tool has generally a code generation plugin. This is possible only if the description language used is sufficiently accurate (formally defined) and relatively closed to the target programming language. Code generation can be used at different levels.

First, code generation tools can be used on models that describe the whole system. However, this technique is rarely used as the models are not always formally defined and they often describe systems that are only simulable and not executable which prevents code generation. One such modelization language is AADL (Architecture Analysis and Design Language), mainly used in the avionics domain. Indeed, the tool Ocarina can generate C or Ada code from a model described with AADL.

The second possibility is to use code generation tools on components (subparts of the model). These components can be represented using different languages, but for embedded systems and especially in avionics domain, synchronous languages [5] are used. These languages allow describing sub-components synchronized with a clock that react simultaneously to all inputs. This corresponds to the behavior of embedded systems, in particular when they execute control-command programs. Lustre [31] is an example of a synchronous language. The Lustre team provides also tools to simulate models in order to verify properties and tools to generate C code.

In avionics and other critical systems, certification is needed in order to commercialize the system. Certification is obtained by providing several documents about the conception and the implementation of the system to the certification authorities. These documents are generally based on the models and implementation must correspond to the specification. This certification process limits therefore the utilization of code generation tools. For example, software must follow the models from which it has been generated (keeps the same classes, procedures...), which reduces the possibility of optimization. However, there are some code generation tools that generate a certified code. This is the case, for example, of SCADE suite, developed by ANSYS, based on Lustre and used by Airbus. SCADE allows to generate C code that respects the aeronautical standard DO-178.

In addition to the functional aspects of the system, models can also describe some properties that the programs must guarantee. There are two types of code generation tools that can help to prove these properties. First, there are some tools that can generate code in a programming language and also in formal languages. The code in formal languages is not executable, but it can be used by formal tools in order to verify the properties defined by the specification. The second solution is that the tool will directly annotate the code generated. The properties will then be verified on the code. For example, some tools that generate C code can add ACSL annotations and assertions in order that Frama-C[4] will be able to verify the properties.

The major problem of code generation tools is that they generally not provide the guarantee that the generated code or the annotations correspond exactly to the specification. Indeed, the process to generate code is complex and it is often difficult to prove it. However, there exist some tools that are verified by construction. This is the case for instance for Vélus, a Lustre compiler that has been proved in Coq (a proof assistant presented in section 8).

[4] Frama-C is a tool that applies formal methods on C code and presented in section 5.3.2

Part II
Formal methods

Formal methods are mathematically-based languages, techniques and tools to verify software systems [10]. They are several “families” of formal methods and the objective of this part of the report is to present the theory behind them, how they work and for which type of proof or property they are adapted. Simple examples will illustrate the presentation.

Section 5 presents *deductive methods*, a first family of formal methods in which program verification is translated into validity checking for a particular mathematical logic formula. Section 6 presents the *abstract interpretation* framework in which programs semantics is abstracted in order to made verification easier. Section 7 presents *model checking*, a technique used for several years to prove properties on models that represent not only code, but more generally systems. Finally, section 8 presents proof assistants, powerful software that help to write and verify formal proofs and can be used to prove complex software like compilers for instance.

4 Simple Imperative Language

Throughout this document, a very simple imperative language will be used to illustrate the different formal methods work.

Programs in this language can defined variables and manipulate different types of numbers (integer, real or float values). The variables are elements of the set $\mathbb{V}$ and the state of the memory is defined by the function ρ that associates variables to their value. If not specified, the variables must be considered to be integer variables and thus the state of the memory is defined by the environment function $\rho : \mathbb{V} \rightarrow \mathbb{Z}$. The 3 basic arithmetic operations are supported : $+, -$ and $\times$. Also, for boolean condition, all arithmetic comparison operators are supported, i.e. $<, \leq, >, \geq, =$, and $\neq$. Note that $=$ is the equality comparison operator and $:=$ is reserved for assignment as presented below.

4.1 Syntax

An imperative program is a statement *stm* in the following grammar:

$$
\begin{array}{ll}
\mathit{stm} ::= & \mathit{stm};\, \mathit{stm} \mid v \mathrel{\mathbf{:=}} \mathit{expr} \mid v \mathrel{\mathbf{:=}} \mathbf{?}(X, Y) \\
 & \mid \mathbf{if}\ \mathit{cond}\ \mathbf{then}\ \mathit{stm}\ \mathbf{else}\ \mathit{stm}\ \mathbf{fi} \\
 & \mid \mathbf{while}\ \mathit{cond}\ \mathbf{do}\ \mathit{stm}\ \mathbf{od} \\
\mathit{cond} ::= & \mathit{expr} < \mathit{expr} \mid \mathit{expr} \leq \mathit{expr} \\
 & \mid \mathit{expr} > \mathit{expr} \mid \mathit{expr} \geq \mathit{expr} \\
 & \mid \mathit{expr} \neq \mathit{expr} \mid \mathit{expr} = \mathit{expr} \\
\mathit{expr} ::= & v \mid X \mid \mathit{expr} + \mathit{expr} \mid \mathit{expr} - \mathit{expr} \mid \mathit{expr} \times \mathit{expr}
\end{array}
$$

with $v \in \mathbb{V}$ and if we consider integer variables, $X, Y \in \mathbb{Z}$. Throughout the report, we will have this notation and we will note P and Q two statements *stm*, C a condition *cond* and E an expression *expr*.

Example 1. *Simple program to present the capabilities of the language. This program will produce in the variable M the maximum value read from 10 entries, each one being an integer chosen randomly between* 0 *and* 100*.*

```
M := $-1$;
I := 1;
R := 0;
while (I $\leq$ 10) do
        I := I + 1;
        R := ?(0:100);
        if R > M then
             M := R;
        fi
od
```

To keep things as simple as possible for this introduction on formal methods, the language contains only the above constructs. It could be extended with functions calls, pointers, elaborate data structures,...However, this tiny language is already Turing complete and these additional features would not be useful for this presentation.

4.2 Collecting Semantics

First, we informally define the semantics of our language:

- v := *expr* is the assignment operator. Classically, v:= 2 + 3 means "assign value of variable v to the result of the evaluation of $2 + 3$". Thus, the left operand of := must be a variable, and the right value of := must be a valid arithmetic expression defined in first-order logic. The right value might also be ?(X, Y) that represents a random choice of a number between X and Y. This value can be seen as an input of the program.

- *stm*; *stm* is the sequence operator. P;Q means "execute P then execute Q".

- **if** *cond* **then** *stm* **else** *stm* **fi** is the selection operator. The statement "**if** C **then** P **else** Q **fi** " means "if the boolean condition C is true, execute P else execute Q".

- **while** *cond* **do** *stm* **od** is the iteration operator. **while** C **do** P **od** means "execute repeatedly P until C is false".

- The *cond* and *expr* corresponds respectively to the evaluation of the Boolean and arithmetic expression.

This informal definition can be now translated to the formal collecting semantics. We will note $\rho : \mathbb{V} \rightarrow \mathbb{Z}$ an environment and $R \subseteq (\mathbb{V} \rightarrow \mathbb{Z})$ a set of environments.

Semantics of expressions $[\![E]\!]$

The semantics $[\![E]\!](\rho) \in \mathbb{Z}$ of an expression E computes the corresponding integer value for a given environment ρ and it is defined as:

$$\begin{aligned}
&[\![v]\!](\rho) := \rho(v) \\
&[\![X]\!](\rho) := X \\
&[\![E_1 \diamond E_2]\!](\rho) := [\![E_1]\!](\rho) \diamond [\![E_2]\!](\rho) \qquad \text{for } \diamond \in \{\, +, -, \times \,\}\,.
\end{aligned}$$

Semantics of condition $[\![C]\!]$

The semantics $[\![C]\!](R) \subseteq (\mathbb{V} \to \mathbb{Z})$ defines the set of possible environments that are in the previous set of environments R and which respect the condition C. The semantics is formally defined as:

$$[\![E_1 \bowtie E_2]\!](\rho) := \{\rho \in R \mid [\![E_1]\!](\rho) \bowtie [\![E_2]\!](\rho)\} \text{ for } \bowtie \in \{\, >, \geq, <, \leq, \neq, = \,\}$$

Semantics of statements $[\![P]\!]$

The semantics of $[\![P]\!](R) \subseteq (\mathbb{V} \to \mathbb{Z})$ defines how the new set of environments is computed after the execution of the statement P with the initial set of environments R:

$$\begin{aligned}
&[\![P; Q]\!](R) := [\![Q]\!]\,([\![P]\!](R)) \\
&[\![v := E]\!](R) := \{\rho[v \mapsto [\![E]\!](\rho)] \mid \rho \in R\} \\
&[\![v := ?(\mathrm{X}, \mathrm{Y})]\!](R) := \{\rho[v \mapsto Z] \mid \rho \in R, Z \in \mathbb{Z}, X \leq Z \leq Y\} \\
&[\![\textbf{if } \mathrm{C} \textbf{ then } \mathrm{P} \textbf{ else } \mathrm{Q} \textbf{ fi}]\!](R) := [\![P]\!]\,([\![C]\!](R)) \cup [\![Q]\!]\,([\![\neg C]\!](R)) \\
&[\![\textbf{while } \mathrm{C} \textbf{ do } \mathrm{P} \textbf{ od}]\!](R) := [\![\neg C]\!]\,(\mu\,(\mathcal{X} \mapsto R \cup [\![P]\!]\,([\![C]\!](\mathcal{X}))))\,.
\end{aligned}$$

with $\mu\, f$ the least fixpoint of function f and $\rho[v \mapsto X]$ the same environment as ρ but where the variable v is now associated with the value X.

Proof. For $[\![P]\!]$ to be well defined, the existence of the least fixpoint has to be proven. This can be done by proving $[\![P]\!]$ monotone, i.e., $[\![P]\!]$ is well defined and for all $R, R' \subseteq (\mathbb{V} \to \mathbb{Z})$, if $R \subseteq R'$ then $[\![P]\!](R) \subseteq [\![P]\!](R')$. The proof being carried by structural induction on statements P, the only non trivial case is the last while loop case.

Thus, considering a condition C and a statement P and assuming that $[\![P]\!]$ is monotone, we have to prove that $[\![\textbf{while } \mathrm{C} \textbf{ do } \mathrm{P} \textbf{ od}]\!]$ is well defined on $2^{(\mathbb{V} \to \mathbb{Z})}$ and that for all $R, R' \subseteq (\mathbb{V} \to \mathbb{Z})$ such that $R \subseteq R'$, the following holds:

$$[\![\textbf{while } \mathrm{C} \textbf{ do } \mathrm{P} \textbf{ od}]\!](R) \subseteq [\![\textbf{while } \mathrm{C} \textbf{ do } \mathrm{P} \textbf{ od}]\!](R').$$

Let us first notice that $(2^{(\mathbb{V} \to \mathbb{Z})}, \subseteq)$ is a complete lattice (since for any set S, the set of its subsets 2^S equipped with the order $\subseteq$ is a complete lattice). Moreover, by induction hypotheses, the function $f_R : \mathcal{X} \mapsto R \cup [\![P]\!]\,([\![C]\!](\mathcal{X}))$ is monotone on this lattice. Then, according to the Knaster-Tarski theorem [42, 56], the least fixpoint $\mu\, f_R$ is uniquely defined as

$$\mu\, f_R = \bigcap \left\{ \mathcal{X} \in 2^{(\mathbb{V} \to \mathbb{Z})} \;\middle|\; f_R(\mathcal{X}) \subseteq \mathcal{X} \right\}$$

and the same holds for R'. Since $R \subseteq R'$, by definition of f_R and $f_{R'}$, for all $\mathcal{X}$, $f_R(\mathcal{X}) \subseteq f_{R'}(\mathcal{X})$ which implies that $\{\mathcal{X} \mid f_{R'}(\mathcal{X}) \subseteq \mathcal{X}\} \subseteq \{\mathcal{X} \mid f_R(\mathcal{X}) \subseteq \mathcal{X}\}$. Thus $\mu\, f_R \subseteq \mu\, f_{R'}$, hence finally the result. □

Semantics of Programs $[\![P]\!]$

Finally the semantics of a program P, which is also a statement, is given by the semantics $[\![P]\!](\mathbb{V} \to \mathbb{R})$ of P starting from an unknown state.

This denotational semantics is very classic and can be found in textbooks about program semantics [48].

5 Deductive methods

Deductive methods are processes that use mathematical logics to model and prove properties we want to guarantee about a program. Another definition given by Jean-Christophe Fillâtre in *Deductive Program verification* [23] is: "Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it."

The process to apply deductive methods can be summarized in four points. First, the developer has to define the problem that needs to be proved. That might be a property about overflows or property representing the expected behavior of a function[5]. The second step is to find a language adapted to formalize the problem. There are many specification languages that are often specifics to a programming language. In the following section, we will not present a particular language, but rather the Floyd-Hoare logic [35] that is mainly used by all specification languages (see section 5.1). During the third step, the different properties have to be defined using the specification language chosen. Fourth, the problem of verifying if the program meets the property previously defined must be translated into a logic formula to be proven (for instance by using weakest preconditions, see section 5.2). Finally, the obtained formula need to be proved with the help of automatic solvers like CVC4, Z3 or Alt-Ergo or proof assistants like Coq or Isabelle/HOL. The section 5.3, will present examples of properties adapted for deductive methods and the Frama-c tools to apply these methods on C programs.

5.1 Deductive methods foundations: Floyd-Hoare logic

Let us now present Floyd-Hoare logic [35] using the simple imperative language presented in section 4.

5.1.1 Hoare triple

The Floyd-Hoare logic describes the expected behavior of a program P by considering that the initial memory state of the program respects a predicate φ. The execution of the program must result in a memory state that verifies a predicate ψ. These three elements are called a *Hoare triple* which is noted as follows:

$$\{\varphi\}P\{\psi\}$$

[5] This point is very dependent on the studied program and how it is used, therefore it will not be detailed here. However, the different type of properties that can be proven will be explained in order to give an idea of when deductive methods can be used.

This Hoare triple must be read as: “if the precondition φ is verified and P is executed, then ψ is verified at the end of the execution of P”. In certain contexts, when this triple describes the behavior of a function, the term *contract* might also be used.

The preconditions and postconditions are expressed here in first-order logic with equality, in which classical arithmetic for natural numbers is available. For example, the assertion “if the value of variable X is equal to 3, then the value of variable Y is equal to the value of variable Z plus 1” will be expressed as $(X = 3) \rightarrow (Y = Z + 1)$.

Example 2. *Euclidian division*

The classical program for euclidian division can be expressed in our programming language by a program P defined as follows:

```
$Q := 0$;
$R := X$;
while $(Y \leq R)$ do
        $Q := Q + 1$;
        $R := R - Y$
od
```

The Hoare triple associated with P is thus $\{X \geq 0 \wedge Y > 0\}\ \ P\ \ \{X = Q \times Y + R \wedge 0 \leq Q \wedge 0 \leq R < Y\}$, *specifying P in the following way:*

- *we only consider the cases in which the input variables X and Y verify* $X \geq 0$ *and* $Y > 0$.

- *if the preconditions are true, then P terminates and the output variables Q and R are such that Q is the quotient and R the remainder of the euclidian division of X by Y.*

5.1.2 Inference rules

A formal system is composed of a formal language defining a set of expressions E and a *deductive system* on E. A deductive system (or inference system) on a set E is composed of a set of rules used to derive elements of E from other elements of E. They are called *inference rules.*

If an inference rule allows to derive e_{n+1} (conclusion) from $\mathcal{P} = \{e_1, \ldots, e_n\}$ (premises), it will be noted as follows:

$$\frac{e_1\ e_2\ \ldots\ e_n}{e_{n+1}}$$

When an inference rule is such that $\mathcal{P} = \emptyset$ it is called an *axiom.* If e_1 is an axiom, it is either noted $\frac{}{e_1}$ or simply e_1.

A *proof* or a *deduction* of an element e of E in a formal system $\mathcal{F}$ from a set of hypotheses $\mathcal{H} \subseteq E$ can be represented by a tree whose

- root is e

$$\frac{}{A \to (B \to A)}\,{\scriptstyle (A1)}$$

$$\frac{}{(A \to B) \to ((A \to (B \to C)) \to (A \to C))}\,{\scriptstyle (A2)} \qquad \frac{A \qquad A \to B}{B}\,{\scriptstyle (MP)}$$

$$\frac{}{(A \to B) \to ((A \to \neg B) \to \neg A)}\,{\scriptstyle (A3)}$$

$$\frac{}{\neg\neg A \to A}\,{\scriptstyle (A4)}$$

Formal system 1: The Hilbert-Ackermann formal system for propositional logic

- leaves are axioms or elements of $\mathcal{H}$
- internal nodes are instantiation of inference rules using for premises nodes situated at the next level (starting from the start)

For instance, let us consider the classical Hilbert-Ackermann system for propositional logic with 4 axioms and one inference rule, Modus Ponens, presented on figure 1. In this system, the formula $p \to p$ can be proved using the following proof tree:

$$\dfrac{\dfrac{}{p \to ((p \to p) \to p)}\,{\scriptstyle (A1)} \quad \dfrac{\dfrac{}{p \to (p \to p)}\,{\scriptstyle (A1)} \quad \dfrac{}{(p \to (p \to p)) \to ((p \to ((p \to p) \to p)) \to (p \to p))}\,{\scriptstyle (A2)}}{(p \to ((p \to p) \to p)) \to (p \to p)}\,{\scriptstyle (MP)}}{p \to p}\,{\scriptstyle (MP)}$$

Notice that you can build the tree starting from the root (*analysis*) or from the leaves (*synthesis*). Synthesis is difficult to use as you have to choose axioms to start (look for instance at the proof of $p \to p$ where the axioms are not trivial to choose). The Hilbert-Ackermann system is also not a good system to perform analysis, as the only inference rule provided by the system does not respect the subformula property, i.e. you have to find which "A" to use.

The Floyd-Hoare formal system is presented on figure 2. We will describe the axioms and inference rules in the following.

The assignment rule (:=) is the only axiom of the system. Basically, it says that if postcondition φ holds after v := E, then the precondition must be $\varphi[v/E]$, i.e. φ in which all free occurrences of v have been replaced by E:

$$\frac{}{\{\varphi[v/E]\} \;\; \text{v := E} \;\; \{\varphi\}}\,{\scriptstyle (:=)}$$

It may seem counter-intuitive to have such an axiom in which the precondition is defined from the postcondition and not the contrary, but the reader can easily convince herself that this is a valid formulation. There is a version of this axiom defining the postcondition from the precondition, but it is more complicated:

$$\frac{}{\{\varphi\} \;\; \text{v := E} \;\; \{\exists v' \; (v = E[v/v']) \land \varphi[v/v']\}}\,{\scriptstyle (:=^*)}$$

where v' is a new variable.

There is also a special axiom when the assignment uses random choice:

$$\frac{}{\{\forall v' \ (X \leq v' \leq Y \rightarrow \varphi[v/v'])\} \ \ \text{v := ?(X,Y)} \ \ \{\varphi\}} \ (?)$$

To understand this axiom, it is important to consider that all the predicates in the postcondition about v are deduced from the only information given by the assignment: $X \leq v \leq Y$. Thus, all the variables in the precondition that respect also this condition ($\forall v' \ X \leq v' \leq Y$) then verify the same predicates about v.

The sequence rule (Seq) is rather intuitive: if you can prove the two Hoare triples $\{\varphi\}$ P $\{\gamma\}$ and $\{\gamma\}$ Q $\{\psi\}$ then the precondition of P;Q is φ and its postcondition is ψ. It may be the case that you cannot directly find the common precondition/postcondition γ which allows to apply the rule. To solve the problem, the rule (Cons) and its two variants (Weak) and (Str) allow to strengthen the preconditions or weaken the postconditions of a program.

The conditional rule (Cond) is also intuitive: given a conditional program of the form **if** C **then** P **else** Q **fi**, if you can prove the triple $\{\varphi \wedge C\}$ P $\{\psi\}$ (meaning that you consider executions of P in the cases where φ and the condition C hold) and the triple $\{\varphi \wedge \neg C\}$ Q $\{\psi\}$ (meaning that you consider executions of Q in the cases where φ holds and the condition C does not hold), then the triple $\{\varphi\}$ **if** C **then** P **else** Q **fi** $\{\psi\}$ is proved. Notice again that you may have to use (Cons) and its variants to correctly establish the triples for P and Q.

The iteration rule (It) is the most complicated rule, as it is the only one concerning both partial and total correctness. Let us consider the $\{\varphi\}P\{\psi\}$ Hoare triple. **Partial correctness** for the triple is the following property: "if φ holds when beginning P and P finishes, then ψ holds after the execution of P". **Total correctness** is the following property: "if φ holds when beginning P, then P will finish and ψ will hold after the execution of P". Total correctness proof provides a stronger property, but it is harder to prove, especially when there are complex loops. Now, let us first explain the rule only on the partial correctness problem:

$$\frac{\{\varphi \wedge C\} \ \ \text{P} \ \ \{\varphi\}}{\{\varphi\} \ \ \textbf{while} \text{ C } \textbf{do} \text{ P } \textbf{od} \ \ \{\varphi \wedge \neg C\}} \ \text{(It)}$$

First, the premise of the rule is a Hoare triple $\{\varphi \wedge C\}$ P $\{\varphi\}$ meaning that it should be the case that if the program is in a state verifying φ and C (the loop condition), then executing P brings to a state verifying φ. The formula φ is called a *loop invariant*. It is a formula specifying what is true and what should remain true at each loop iteration. Notice that C does not have to hold at the end of executing P as we have to be able to exit the loop. The conclusion of the rule is the Hoare triple $\{\varphi\}$ **while** C **do** P **od** $\{\varphi \wedge \neg C\}$ meaning that the invariant should be true before entering the loop and is preserved when exiting the loop (in this case, C does not hold anymore). Loop invariant finding is a difficult problem and there is no systematic method to find the formulas expressing a strong invariant of a loop, but some procedures like abstract interpretation may be used to find them. The interested reader can refer to [24] to find loop invariants for classical algorithms.

The complete iteration rule adds some formula to the precondition and postcondition of its premise:

$$\frac{}{\{\varphi[v/E]\}\ \ \text{v := E}\ \ \{\varphi\}}\ (:=)$$

$$\frac{}{\{\forall v'.(X \leq v' \leq Y \rightarrow \varphi[v/v'])\}\ \ \text{v := ?(X,Y)}\ \ \{\varphi\}}\ (?)$$

$$\frac{\{\varphi\}\ \ \text{P}\ \ \{\gamma\} \qquad \{\gamma\}\ \ \text{Q}\ \ \{\psi\}}{\{\varphi\}\ \ \text{P;Q}\ \ \{\psi\}}\ (\text{Seq})$$

$$\frac{\varphi \rightarrow \varphi' \qquad \{\varphi'\}\ \ \text{P}\ \ \{\psi'\} \qquad \psi' \rightarrow \psi}{\{\varphi\}\ \ \text{P}\ \ \{\psi\}}\ (\text{Cons})$$

$$\frac{\varphi \rightarrow \varphi' \qquad \{\varphi'\}\ \ \text{P}\ \ \{\psi\}}{\{\varphi\}\ \ \text{P}\ \ \{\psi\}}\ (\text{Str})$$

$$\frac{\{\varphi\}\ \ \text{P}\ \ \{\psi'\} \qquad \psi' \rightarrow \psi}{\{\varphi\}\ \ \text{P}\ \ \{\psi\}}\ (\text{Weak})$$

$$\frac{\{\varphi \wedge C\}\ \ \text{P}\ \ \{\psi\} \qquad \{\varphi \wedge \neg C\}\ \ \text{Q}\ \ \{\psi\}}{\{\varphi\}\ \ \textbf{if}\ \text{C}\ \textbf{then}\ \text{P}\ \textbf{else}\ \text{Q}\ \textbf{fi}\ \ \{\psi\}}\ (\text{Cond})$$

$$\frac{\{\varphi \wedge C \wedge v = V \wedge v \in D\}\ \ \text{P}\ \ \{\varphi \wedge v \prec V\} \qquad (\text{D}, \prec)\ \text{is wf}}{\{\varphi \wedge V \in D\}\ \ \textbf{while}\ \text{C}\ \textbf{do}\ \text{P}\ \textbf{od}\ \ \{\varphi \wedge \neg C \wedge V \in D\}}\ (\text{It})$$

Formal system 2: The Floyd-Hoare formal system

$$\frac{\{\varphi \wedge C \wedge V = v \wedge v \in D\}\ \ \text{P}\ \ \{\varphi \wedge v \prec V\} \qquad (\text{D}, \prec)\ \text{is wf}}{\{\varphi\}\ \ \textbf{while}\ \text{C}\ \textbf{do}\ \text{P}\ \textbf{od}\ \ \{\varphi \wedge \neg C\}}\ (\text{It})$$

In the Hoare triple $\{\varphi \wedge C \wedge V = v\}$ P $\{\varphi \wedge v \prec V\}$, the expression v is called *the loop variant* and V is a predefined variable. It is an expression based on the program global and local variables whose value should decrease during the execution of the loop. The other premise of the rule, "$(\text{D}, \prec)$ is wf", means that $\prec$ is a well-founded relation on the set D and thus there is a minimal element in the set of possible values for v (for the rest of this section, we will consider $(\mathbb{N}, <)$). Therefore, as the value of v decreases at each loop iteration, we can guarantee that we will exit the loop and that the program terminates.

Example 3. *Let us prove the Hoare triple* $\{\varphi\}$ P $\{\psi\}$ *where* φ, ψ *and* P *are defined by the previously presented program:*

```
{X ≥ 0 ∧ Y > 0}
Q := 0;
R := X;
```

```
while ($Y \leq R$) do
        $Q := Q + 1$;
        $R := R - Y$;
od
$\{X = Q \times Y + R \wedge 0 \leq Q \wedge 0 \leq R < Y\}$
```

In order to simplify the writing of the proof, P1 will denote the subprogram of P consisting of the two first assignments, P2 will denote the subprogram of P consisting in the ***while*** *loop and P2.1 will denote the subprogram of P consisting of the two assignments in the loop.*

We have first to find an invariant and a variant for the loop to establish the proof of the Hoare triple. Clearly, $X = Q \times Y + R \wedge 0 \leq R \wedge 0 \leq Q$ *is a loop invariant and* R *is a loop variant. The well founded relation chosen here is the classic* $<$ *order on natural numbers. We will denote by* $I(A, B)$ *the formula* $X = A \times Y + B \wedge 0 \leq B \wedge 0 \leq A$ *to simplify the proof writing. For instance,* $I(Q, R)$ *represents* $X = Q \times Y + R \wedge 0 \leq R \wedge 0 \leq Q$*.*

The complete proof of the Hoare triple is presented on proof figure 1. The proof tree has been separated into several small proof trees to improve readability and to fit on the page. Notice that the proof has been in fact established "by hand" using analysis, i.e. starting from $\{\varphi\}$ P $\{\psi\}$ *and by going backwards to build the tree using of course the Hoare triples already established by finding the variant and the invariant of the loop.*

Formulas in red boxes are proof obligations*, i.e. assertions that must be proved to be true to be able to use the (Cons) rule or its variants. These proof obligations are normally discharged using a theorem prover. Notice that the proof obligations of the proof are rather intuitive to prove, e.g.*

$$X \geq 0 \wedge Y > 0 \rightarrow X = X \wedge 0 \leq 0 \wedge 0 \leq X$$

but formally writing their proofs here will take a lot of time and space and is not relevent in the present report.

The Hoare triple in a green box cannot normally be used with the (It) rule of the formal system, as we have to prove for instance that if $R = v + Y$*, then if* $R = v$ *after the execution of P2.1 then* $R < v$ *holdsre. Again, we have omitted the corresponding proof obligations due to lack of space.*

The Floyd-Hoare system is sound and complete with respect to classic program semantics (see [48]). This formal system presented here is the simplest one. In particular, extensions to pointer reasoning via separation logic can be found for instance in [49].

5.2 Weakest preconditions: a calculus for Floyd-Hoare logic

The Floyd-Hoare formal system allows to build proofs on Hoare triple, and thus allows to prove programs. But there is no mechanical method or algorithm to build such a proof. Dijkstra has introduced in [20] the *predicate transformer semantics*, a particular semantics for imperative programs close to denotational semantics. This semantics is in fact a reformulation of Floyd-Hoare logic and can be used as a complete strategy to build deductions in Floyd-Hoare logic. Predicate transformer semantics associates with each programming statement a total

$$\frac{\dfrac{}{\{I(O,X)\}\ \ \mathrm{Q} := 0\ \ \{I(Q,X)\}} \qquad \dfrac{}{\{I(Q,X)\}\ \ \mathrm{R} := \mathrm{X}\ \ \{I(Q,R)\}}}{\{I(0,X)\}\ \ \mathrm{P1}\ \ \{I(Q,R)\}}$$

$$\frac{\dfrac{}{\{I(Q+1,R-Y)\wedge(R-Y)=v\}\ \ \mathrm{Q} := \mathrm{Q} + 1\ \ \{I(Q,R-Y)\wedge(R-Y)=v\}} \qquad \dfrac{}{\{I(Q,R-Y)\wedge(R-Y)=v\}\ \ \mathrm{R} := \mathrm{R} - \mathrm{Y}\ \ \{I(Q,R)\wedge R=v\}}}{\{I(Q+1,R-Y)\}\ \ \mathrm{P2.1}\ \ \{I(Q,R)\wedge R=v\}}$$

$$\frac{\dfrac{I(Q,R)\wedge Y\leq R\rightarrow I(Q,R)\wedge(R-Y)=v \qquad \dfrac{I(Q,R)\wedge(R-Y)=v\rightarrow I(Q+1,R-Y)\wedge(R-Y)=v \qquad \{I(Q+1,R-Y)\wedge(R-Y)=v\}\ \ \mathrm{P2.1}\ \ \{I(Q,R)\wedge R=v\}}{\{I(Q,R)\wedge(R-Y)=v\}\ \ \mathrm{P2.1}\ \ \{I(Q,R)\wedge R=v\}}}{\{I(Q,R)\wedge Y\leq R\wedge(R-Y)=v\}\ \ \mathrm{P2.1}\ \ \{I(Q,R)\wedge R=v\}}}{\{I(Q,R)\}\ \ \mathrm{P2}\ \ \{I(Q,R)\wedge\neg(Y\leq R)\}}$$

$$\frac{\dfrac{\varphi\rightarrow I(0,X) \qquad \dfrac{\{I(0,X)\}\ \ \mathrm{P1}\ \ \{I(Q,R)\} \qquad \{I(Q,R)\}\ \ \mathrm{P2}\ \ \{I(Q,R)\wedge\neg(Y\leq R)\}}{\{I(0,X)\}\ \ \mathrm{P}\ \ \{I(Q,R)\wedge\neg(Y\leq R)\}}}{\{\varphi\}\ \ \mathrm{P}\ \ \{I(Q,R)\wedge\neg(Y\leq R)\}} \qquad I(Q,R)\wedge\neg(Y\leq R)\rightarrow\psi}{\{\varphi\}\ \ \mathrm{P}\ \ \{\psi\}}$$

Proof 1: The proof trees of the Hoare triple corresponding to the euclidian division program function between two predicates representing the preconditions and the postconditions of the statement. Depending on the domain and co-domain of the function, predicate transformers are *weakest preconditions* (from postconditions to preconditions) or *strongest postconditions* (from preconditions to postconditions). We will only present weakest-preconditions.

The principle for weakest precondition calculus is to determine the minimal precondition that implies the postcondition we want to prove. The computation of the weakest-precondition is performed with the function $wp(P, \psi)$. P is the program executed and ψ the postcondition. The wp is defined such that the following Hoare triple is always correct for every programs P and postconditions ψ:

$$\{wp(P, \psi)\}P\{\psi\}$$

The wp function is relatively easy to compute. It only requires to apply simple formulas that will be presented later. When the weakest precondition of a program is computed, the last step is to prove the following relation between the precondition of the program and the weakest precondition:

$$\{\varphi\}P\{\psi\} \iff \varphi \to wp(P, \psi)$$

In less formal words, the triple $\{\varphi\}P\{\psi\}$ is correct if and only if the formula φ is sufficient to logically imply the weakest-precondition needed to have ψ holding after executing the program P (that is $wp(P, \psi)$). The proof of this equivalence is not as immediate as the computation of wp. It can often be achieved by using automatic solvers like Alt-Ergo, Z3 or CVC4, but sometimes the problem is complex and it requires to manually prove it. In this case, proof assistants such as Coq or Isabelle/HOL can be used (see section 8).

Definition of the wp function

All programs are composed of sequences of simple instructions that implement complex algorithms. It is too complicated to compute the weakest-precondition by analysing the program as one block, it has to be divided into smaller problems. Following what has been presented in previous section, we will use the same syntax for imperative language that the one presented in section 4.

The computation of the weakest-precondition of a complex program is reduced to the computation of the weakest-precondition for simple instructions. wp can be defined for the instructions that can be found in the majority of programming languages: assignment, sequence of instructions, condition, loops and are present in our imperative language.

We start with the simplest instruction: **variable assignment**. For the assignment $v := E$ and a postcondition ψ, the weakest-precondition is the formula ψ where all the free occurrences of x are replaced by E (noted $\psi[v/E]$):

$$wp(v := E, \psi) := \psi[v/E]$$

Indeed, if there is a predicate $Pred(v)$ about v in the postcondition and its value is known due to the assignment ($v := E$), the precondition has to be such that the predicate $Pred(E)$ holds. This is also true if the expression E contains the variable v as presented in the following example:

Example 4. *We want to compute the weakest precondition for the instruction $v := v + 3$ and the postcondition $\{v = 5\}$:*

$$\begin{aligned} wp(v := v + 3, \{v = 5\}) &:= \{v = 5\}[v/v + 3] \\ &= \{v + 3 = 5\} \\ &= \{v = 2\} \end{aligned}$$

And we get the Hoare triple:

$$\{v = 2\}v := v + 3\{v = 5\}$$

There is also the computation of an assignment using **random choice**. This assignment gives us the information that, in the postcondition, the variable v is bound by the values X and Y. The predicates in the postcondition about v are only based on this information. Then, the variables that are also bound verify the same predicats as v. Finally, we get the following definition:

$$wp(v :=?(X, Y), \psi) := \forall v'(X \leq v' \leq Y \rightarrow \psi[v/v'])$$

We note P and S two programs and we want to compute the weakest precondition of **the sequence** $P; Q$ considering a postcondition ψ. In order to find the weakest-precondition of the program P using wp function, we need a postcondition ψ_1. We want to guarantee that we will produce the weakest-precondition of the sequence $P; Q$. Thus, ψ_1 needs to be minimal and it needs to imply the weakest-precondition of Q for the postcondition ψ: $\psi_1 \rightarrow wp(Q, \psi)$. These properties are ensured using this definition: $\psi_1 := wp(Q, \psi)$. Therefore, wp can be defined as follows for a sequence of instructions:

$$wp(P; Q, \psi) := wp(P, wp(Q, \psi))$$

For the computation of weakest precondition for the **conditional instruction**, **if** C **then** P **else** Q **fi**, two cases have to be considered. When the condition C is `true`, we only have a hypothesis on C and we have to prove ψ by executing the sequence of instruction P contained in the *then* statement. Thus, C must be sufficient to imply the weakest precondition of P for the postcondition ψ computed with the function wp (we want $C \rightarrow wp(P, \psi)$). When the condition C is `false`, the same principle is applied, but with the sequence of instructions Q in the *else* statement. In the end, we get this definition for the condition instruction:

$$wp(\textbf{if } C \textbf{ then } P \textbf{ else } Q \textbf{ fi}, \psi) := (C \rightarrow wp(P, \psi)) \wedge (\neg C \rightarrow wp(Q, \psi))$$

The computation of the weakest-precondition for **loop** (of the form **while** C **do** P **od**) is different from what has been presented earlier. Indeed, it requires an *invariant* I that is always `true` at the beginning and at the end of each execution of the body of the loop as seen in the previous section. Most of the time, the invariant is given before every loop of a program using the same specification language that the one used to specify the pre- and postconditions.

When the invariant I is known, the weakest-precondition of the loop can be defined as follows:

$$wp(\textbf{while } C \textbf{ do } P \textbf{ od}, \psi) := I \wedge ((C \wedge I) \to wp(P, I)) \wedge ((\neg C \wedge I) \to \psi)$$

To understand this definition, we split the formula in three parts:

- I: the invariant must always be `true` when entering the loop.
- $(C \wedge I) \to wp(P, I)$: if the condition is true, we will execute the loop body P. The execution of P must result in a state that respects the invariant I (it must always be true). Thus, the weakest-condition of the body's loop is obtained with $wp(P, I)$. This formula limits how much information I can carry because it must be verified at each loop iteration.
- $(\neg C \wedge I) \to \psi$: if the condition C is not respected (and the invariant I is correct), then the body of the loop will never be executed again and the loop ends. It this case, the postcondition must be verified with just the hypothesis of the condition C. Thus, if we want to be able to prove this formula, the invariant I must carry extra information.

These different elements show why it is hard to find a correct invariant. Indeed, it must carry enough information to prove that the postcondition will be verified when the loop ends. However, it cannot carry too much information otherwise it will not be possible to prove that it holds at each iteration.

In addition of the invariant, a *variant* must also be defined. The variant is a value in a well-founded set that must decrease strictly at each iteration of the loop. If taking a variant in $(\mathbb{N}, <)$, this value is therefore an upper bound of the number of iterations for a loop. If the correctness of the variant can be proved (i.e. ensure that it is strictly decreasing and that it is a value of a well-founded set), then we can guarantee that the loop terminates. For this reason, the definition of variant and its proof are essential to ensure the total correctness of a program.

5.3 Deductive methods and real-world languages

As presented earlier, the choice of the specification language is an important step to use deductive methods. This choice is based on different criteria. The language must allow to formally describe the properties to be ensured. However, this choice is very dependent on the programming language used. Indeed, there are various tools helping or automating the realisation of the proof, but they are often compatible with a unique specification language. For example, Frama-C[6] for C programs uses ACSL as specification language, whereas GNATProve only work for the SPARK programming language. Also, for the same

[6] Framework for Modular Analysis of C programs (`https://frama-c.com`). C program analyzer for abstract interpretation or to apply the deductive methods. See section 5.3.2.

programming language, there may be several specification languages: for instance, for C programs, you may use ACSL from Frama-C, but also the Verified Software Toolchain which is completely different.

In the previous section, we have seen the theoretical tools to specify a program. The different types of properties that can be specified will be detailed in the following. We will then present the Frama-C tool to apply deductive methods on a real-world programming language, i.e. the C programming language.

5.3.1 Properties specification

To specify properties of a program, we will use the following semi-formal notation. M is a memory state at a certain point of the program execution. *Adresses* is the set of all existing addresses in all memory M. *Values* is the set of all possible values that can be stored. We note $\&M$ the set of all accessible addresses for the user in M ($\&M \subseteq \mathit{Adresses}$), $M(a)$ the stored value at the address a in M ($a \in \&M$ and $\{M(a) | a \in \&M\} \subseteq \mathit{Values}$). For a function `fun` (or a portion of code), $M^{init}_{\texttt{fun}}$ and $M^{exec}_{\texttt{fun}}$ are respectively the state of the memory before and after executing the `fun` function.

When a Hoare triple is defined for a function, the post and pre-conditions can be classified into 3 types of properties: functional, safety and security properties.

- **Functional properties** describe the expected result after executing a function by assuming that the given parameters respect the precondition.
- **Safety properties** describe the absence of runtime errors (RTE). These errors can cause unexpected behaviors of the program or an unresponsive application. They can be overflows, division by 0, infinite loop...
- **Security properties** describe the absence of possible attack from an unauthorized user. Deductive methods are mainly used for the 2 previous types of properties, but there are some research works to apply them to cybersecurity [36]. The main idea when defining security properties is to have empty preconditions and uses guards to ensure the validity of the parameters or the memory.

Example 5. *To illustrate the difference between these three kinds of properties, we define the Hoare triple for the* ***swap*** *function in C as follows:*

$$\begin{aligned} \forall a, b \in \mathit{Addresses}\ A, B \in \mathit{Values}, \quad & \{\varphi^{swap}_{functional} \land \varphi^{swap}_{safety} \land \varphi^{swap}_{security}\} \\ & \boldsymbol{\mathtt{swap}}(a, b) \\ & \{\psi^{swap}_{functional} \land \psi^{swap}_{safety} \land \psi^{swap}_{security}\} \end{aligned}$$

The ***swap*** *function takes two pointers as arguments and exchanges the pointed values. It specification should be: "If two valid pointers are given in parameter, the* ***swap*** *function must exchanges the two pointed values, without modifying the rest of the memory". This definition is complete and can be specified with the 3 types of properties.*

***Functional properties**: the `swap` function exchanges the values contained in two pointers. We can formalize this property with the following predicates:*

$$\varphi^{\mathtt{swap}}_{functional} := (M^{init}_{\mathtt{swap}}(a) = A \land M^{init}_{\mathtt{swap}}(b) = B)$$

$$\psi^{\mathtt{swap}}_{functional} := (M^{exec}_{\mathtt{swap}}(a) = B \land M^{exec}_{\mathtt{swap}}(b) = A)$$

***Note**: a, b, A and B are constants that are defined in the formula of the triple. They will not change during the execution of `swap`.*

***Safety properties**: we want the pointers passed as parameters to be valid (the pointers correspond to addresses accessible by the user). We specify this property by:*

$$\varphi^{\mathtt{swap}}_{safety} := (a \in \&M^{init}_{\mathtt{swap}} \land b \in \&M^{init}_{\mathtt{swap}})$$

$$\psi^{\mathtt{swap}}_{safety} := True$$

The precondition add parameters verification to ensure that the function will not encounter run time error. The postcondition is only $True$ because this type of property does not add any guarantee on the result or the state of the system after execution.

***Security properties**: we want to ensure that the `swap` function only modifies the elements pointed by the two given parameters. The other elements of the memory should not be modified, and no memory allocation or free should be made. For instance, the following triple defines such a property:*

$$\varphi^{\mathtt{swap}}_{security} := True$$

$$\psi^{\mathtt{swap}}_{security} := \begin{Bmatrix} \forall c \in Adresses : \left(c \neq a \land c \neq b \land c \in \&M^{init}_{\mathtt{swap}}\right) \\ \rightarrow \left(c \in \&M^{exec}_{\mathtt{swap}} \land M^{init}_{\mathtt{swap}}(c) = M^{exec}_{\mathtt{swap}}(c)\right) \end{Bmatrix}$$

The postcondition verifies for all valid addresses that are not a or b (the only values pointed that have been modified) that, after the execution of the function, they are still valid (no memory freed) and the stored values at these memory addresses have not changed.

These 3 types of properties are complementary and very important for the specification of a program. They ensure the correctness of the program and they also guarantee that the program will run without error or unexpected memory modifications.

5.3.2 Frama-C

Frama-C (*Framework for Modular Analysis of C-programs*) is a tool developed by the CEA List and Inria to analyze C programs. Frama-C is able to parse a C program and generate an abstract syntax tree representing the program using CIL (C Intermediate Language). It also supports ACSL (*ANSI C Specification Language*) to add annotations to the abstract syntax tree. The architecture of Frama-C supports several plugins that can analyze the abstract syntax tree. For example, the *EVA* plugin applies abstract interpretation methods

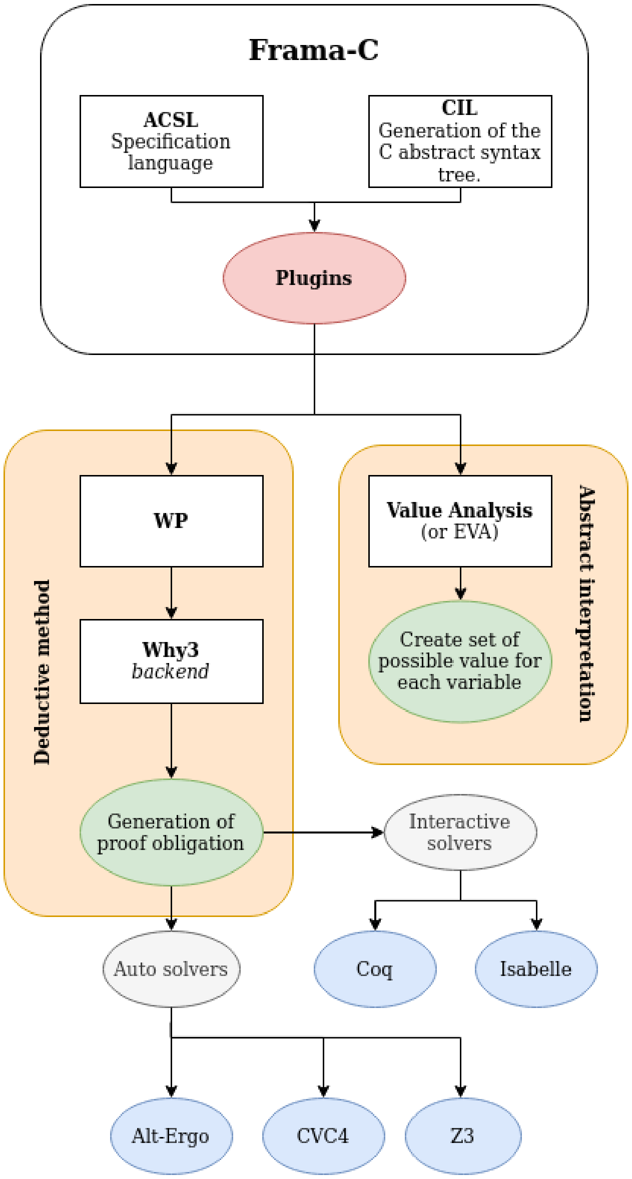


Figure 2: Diagram about Frama-C

(see section 6) and the *WP* plugin applies deductive methods through weakest preconditions calculus. The figure 2 shows how Frama-C and the plugins work.

WP (*Weakest Precondition*) is a Frama-C plugin that applies deductive methods by computing weakest preconditions. It uses ACSL language as a specification language to describe the properties and the contracts to be proved. It also provides several memory models for different levels of abstraction for C programs. In this section, we will not detail how to use Frama-C and WP but the tutorial by A. Blanchard [7] provides all the information needed. When WP analyses the CIL abstract syntax tree, it generates WhyML code which is a language provided by the Why3 platform.

Why3 is deductive verification platform developed by Jean-Christophe Filliâtre and others [23]. For a given WhyML program with its specification, Why3 will compute the weakest preconditions of the program and generate the corresponding proof obligations. It has different interfaces to interact with automatic solvers as well as proof assistants.

The functional specification of a function or a program is not always trivial and most of the time the proof is even more difficult. In massive programs that have not been designed with proof in mind, the specification and the proof are very time-consuming. Unfortunately, most companies cannot afford to spend this time on verification. A partial solution is to use a functionality of WP that creates assertions in the code to prevent RTE. The developer has just to write minimal contracts for each function. The proofs are then easier and can be realised automatically. This technique allows any company to ensure the absence of RTE for a minimal cost.

6 Abstract interpretation

One of the goals of program static analysis is to determine if specific states of the memory exist and are accessible. The basic idea is to compute for each variable and for each point of the program[7] the set of possible values. Unfortunately, this problem is too complex and it is not computable.

Abstract interpretation is a convenient and versatile formal framework to define static analyses of programs [12–14]. The main principle of abstract interpretation is to use *abstract domains* to abstract the set of possible values (see section 6.1). The computation of these sets for each state of the program depends on the instructions of the program. In order to make the calculations easier on these sets, *abstract operators* are defined (see section 6.2). Even with these tools, the computation of the sets when there are loops might not converge. For this reason, some techniques like widening are used as presented in section 6.3. Finally, there are different types of abstract domains and iteration techniques that will be presented in section 6.4.

6.1 Abstract domains

Abstract domains are the basic bricks of abstract interpretation. They are based on a *complete lattice* $\left(\mathcal{D}^\sharp, \sqsubseteq^\sharp_\mathcal{D}\right)$ (the exponent $.^\sharp$ is commonly used to distinguish abstract objects from their concrete counterpart). A set $\mathcal{D}^\sharp$ with a relation order $\sqsubseteq^\sharp_\mathcal{D}$ is a complete lattice if and only if it admits a least upper bound or a greatest lower bound.

Remark: in all the current section about *abstract interpretation* (section 6), the simple imperative language defined in section 4 will be used. The following examples will use the concrete domain $\mathcal{D} = 2^{(\mathbb{V}\to\mathbb{Z})}$ i.e., a set of environments $\rho : \mathbb{V} \to \mathbb{Z}$ associating a value in $\mathbb{Z}$ to each variable in $\mathbb{V}$. Thus, we have the concrete complete lattice $(\mathcal{D}, \sqsubseteq_\mathcal{D}) := \left(2^{(\mathbb{V}\to\mathbb{Z})}, \subseteq\right)$.

The least upper bound (also called join) of the lattice, its greatest lower bound (also called meet), its least and greatest elements are respectively denoted $\sqcup^\sharp_\mathcal{D}$, $\sqcap^\sharp_\mathcal{D}$, $\bot_\mathcal{D}$ and $\top_\mathcal{D}$. Binary versions of the least upper bound and greatest lower bound will commonly be used (i.e., $\sqcup^\sharp_\mathcal{D} \{ x^\sharp, y^\sharp \}$ will be denoted $x^\sharp \sqcup^\sharp_\mathcal{D} y^\sharp$). To lighten notations, the subscript $_\mathcal{D}$ will often

[7] A state of a program can be seen as a state of a finite-state machine. This state contains all possible values of the variables. The transitions correspond to the execution of an instruction of the program

be omitted when it can easily be inferred from the context. Each element in $\mathcal{D}^\sharp$ called an *abstract value* will "abstract" some concrete element in $\mathcal{D}$. What "abstract" means is formally defined by a function from $\mathcal{D}^\sharp$ to $\mathcal{D}$ called a *concretization function*.

A *concretization function* is a function $\gamma_\mathcal{D} : \mathcal{D}^\sharp \to \mathcal{D}$ from the abstract domain $\mathcal{D}^\sharp$ to the concrete $\mathcal{D}$. This function must be monotone:

$$\forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp,\ x^\sharp \sqsubseteq^\sharp_\mathcal{D} y^\sharp \Rightarrow \gamma_\mathcal{D}(x^\sharp) \sqsubseteq_\mathcal{D} \gamma_\mathcal{D}(y^\sharp). \tag{1}$$

The monotonicity constraint allows to give a concrete meaning to the abstract order $\sqsubseteq^\sharp_\mathcal{D}$: when an abstract value $x^\sharp$ is smaller than another $y^\sharp$ in $\mathcal{D}^\sharp$ (i.e., $x^\sharp \sqsubseteq^\sharp_\mathcal{D} y^\sharp$), then $x^\sharp$ represents less environments than $y^\sharp$ (i.e., $\gamma_\mathcal{D}(x^\sharp) \subseteq \gamma_\mathcal{D}(y^\sharp)$). In other words, $x^\sharp \sqsubseteq^\sharp_\mathcal{D} y^\sharp$ means that the abstraction $x^\sharp$ is *more precise* than $y^\sharp$.

Remark: In our example domain $\mathcal{D}$, for any abstract domain $\mathcal{D}^\sharp$ with a concretization function $\gamma_\mathcal{D}$, we will have, $\forall x^\sharp \in \mathcal{D}^\sharp$, $\gamma_\mathcal{D}(x^\sharp) \subseteq 2^{(\mathbb{V}\to\mathbb{Z})}$. We also have $\cup, \cap, \emptyset$ and $2^{(\mathbb{V}\to\mathbb{Z})}$ that are respectively the least upper bound, the greatest lower bound, its least and greatest elements.

Example 6 (Intervals Domain)**.** *A very common abstract domain is the* intervals domain *defined as: $\mathcal{I}^\sharp := \mathbb{V} \to I$ with $I := \bot_I \cup \left\{[a,b] \,\middle|\, a \in \overline{\mathbb{Z}}, b \in \overline{\mathbb{Z}}, a \le b\right\}$ where $\overline{\mathbb{Z}} := \mathbb{Z} \cup \{-\infty, +\infty\}$ and $\le$ is the usual order on $\overline{\mathbb{Z}}$. An order $\sqsubseteq^\sharp_\mathcal{I}$ is defined on $\mathcal{I}^\sharp$ by $x^\sharp \sqsubseteq^\sharp_\mathcal{I} y^\sharp$ when for all $v \in \mathbb{V}$:*

- *either $x^\sharp(v) = \bot_I$;*
- *or $x^\sharp(v) = [a,b]$ and $y^\sharp(v) = [c,d]$ and $c \le a$ and $b \le d$.*

Equipped with this order, $\mathcal{I}^\sharp$ is a complete lattice. Giving a subset S of I, $\sqcup^\sharp_I S$ is defined as:

$$\sqcup^\sharp_I S := \begin{cases} \bot_I & \text{when } S = \emptyset \text{ or } S = \{\bot_I\} \\ \Big[\inf\left\{a \in \overline{\mathbb{Z}} \,\middle|\, [a,b] \in S\right\}, & \text{otherwise} \\ \quad \sup\left\{b \in \overline{\mathbb{Z}} \,\middle|\, [a,b] \in S\right\}\Big] & \end{cases}$$

which allows to define the least upper bound $\sqcup^\sharp_\mathcal{I}$ of $\mathcal{I}^\sharp$:

$$\sqcup^\sharp_\mathcal{I} S := \left(v \mapsto \sqcup^\sharp_I \left\{x^\sharp(v) \,\middle|\, x^\sharp \in S\right\}\right).$$

The infimums $\bot_\mathcal{I}$ and $\top_\mathcal{I}$ are the functions mapping every variable respectively to $\bot_I$ and $[-\infty, +\infty]$.

The concretization function $\gamma_\mathcal{I} : \mathcal{I}^\sharp \to 2^{(\mathbb{V}\to\mathbb{Z})}$ is given by:

$$\gamma_\mathcal{I}(x^\sharp) := \left\{\rho : \mathbb{V} \to \mathbb{Z} \,\middle|\, \forall v \in \mathbb{V}, \exists a, b \in \overline{\mathbb{Z}}, x^\sharp(v) = [a,b] \wedge a \le \rho(v) \le b\right\}.$$

It is worth noting that $\gamma_\mathcal{I}(x^\sharp) = \emptyset$ when $x^\sharp(v) = \bot_I$ for any $v \in \mathbb{V}$.

6.2 Abstract operators

The previous section just defined an abstraction. The goal is to use this abstraction to compute an overapproximation of the concrete states of the program. This requires operations on this abstraction. *Abstract operators* are functions on an abstract domain that mimic *actual operations* such as assignments and guards in order to enable the computation in the abstract domain of an overapproximation of the concrete states of the program.

The *actual operations* are functions defined as $[\![P]\!] : \mathcal{D} \to \mathcal{D}$ where P is a statement (an assignment, a condition, a loop...). For a $x \in \mathcal{D}$ a state of the variables, $[\![P]\!](x)$ is the new state of the variables after the execution of the statement P. These functions are always defined by the semantics of the language and for our simple imperative language, the semantics is defined in section 4.2.

The *Abstract operators* have to be defined formally. Given a variable $v \in \mathbb{V}$ and an expression E, an abstract operator for the assignment $v{:=}E$ is a function $[\![v{:=}E]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$. Similarly, abstract operators for random assignments $[\![v{:=}?(X,Y)]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ and for guards $[\![C]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ are defined[8].

These operators are called *sound* with respect to their concrete counterpart when they fulfill the following condition:

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![v{:=}E]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![v{:=}E]\!]^\sharp(x^\sharp)\right).$$

This condition is called *soundness condition.* Similar conditions are defined for random assignments and guards.

Intuitively, the soundness condition expresses the fact that the abstract operator does not forget any behavior of its concrete counterpart. More precisely, if an environment ρ can be obtained through the assignment $v{:=}E$ starting from an environment represented by $x^\sharp$ (i.e., in $\gamma_{\mathcal{D}}(x^\sharp)$), that is $\rho \in [\![v{:=}E]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right)$, then this environment is required to be represented by the result of the abstract operator $[\![v{:=}E]\!]^\sharp$ on $x^\sharp$, that is $\rho \in \gamma_{\mathcal{D}}\left([\![v{:=}E]\!]^\sharp(x^\sharp)\right)$.

Given these abstract operators, an abstract semantics of statements $[\![P]\!]^\sharp$ can be defined. First, the sequence of statements:

$$[\![P;Q]\!]^\sharp(x^\sharp) := [\![Q]\!]^\sharp\left([\![P]\!]^\sharp(x^\sharp)\right) \tag{2}$$

For $x^\sharp$ a state of the program, we start by computing the abstract domain $x^\sharp_P$ after the execution of the statement P. Then, the final abstract domain can be computed using the abstract operator of Q with the state $x^\sharp_P$.

The abstract operator for **if** C **then** P **else** Q **fi** is relatively simple to understand. The abstract domain generated is the abstract union between the abstract state produced by the execution of P and Q assuming that $x^\sharp$ respects respectively C and $\neg C$. This gives us this definition:

$$[\![\textbf{if}\ \mathrm{C}\ \textbf{then}\ \mathrm{P}\ \textbf{else}\ \mathrm{Q}\ \textbf{fi}]\!]^\sharp(x^\sharp) := [\![P]\!]^\sharp\left([\![C]\!]^\sharp(x^\sharp)\right) \sqcup^\sharp [\![Q]\!]^\sharp\left([\![\neg C]\!]^\sharp(x^\sharp)\right) \tag{3}$$

[8] These definitions depend of the chosen abstract domain. An example for the intervals domain will be given later.

The last abstract operator is the loop **while** C **do** P **od** and its definition is given as follows:

$$[\![\textbf{while } \mathrm{C} \textbf{ do } \mathrm{P} \textbf{ od}]\!]^\sharp(x^\sharp) := [\![\neg C]\!]^\sharp \left(\mu \left(\mathcal{X} \mapsto x^\sharp \sqcup^\sharp [\![P]\!]^\sharp \left([\![C]\!]^\sharp(\mathcal{X})\right)\right)\right). \tag{4}$$

where $\mu(f)$ is the least fixpoint of the function f. In order to understand this definition, let us decompose it.

- Initially, the abstract state is $x^\sharp$, the state before the execution of the loop. Then, we can compute a new abstract state by removing all the states which do not satisfy the condition C and applying the loop body P. This new state is obtained with the operators $[\![P]\!]^\sharp \left([\![C]\!]^\sharp(x^\sharp)\right)$.We can repeat these steps for an arbitrary number of executions and define the sequence $(\mathcal{X}_n)_{n\in\mathbb{N}}$:

  $$\forall n \in \mathbb{N},\ \mathcal{X}_n := \begin{cases} x^\sharp & \text{if } n = 0 \\ x^\sharp \sqcup^\sharp [\![P]\!]^\sharp \left([\![C]\!]^\sharp(\mathcal{X}_{n-1})\right) & \text{otherwise} \end{cases}$$

  $\mathcal{X}_n$ corresponds to the abstract state obtained after n executions of the loop.

- In general, when abstract interpretation methods are used, the number of iterations is not known. The final abstract states for the execution of the loop can be defined as $\sqcup^\sharp_{n\in\mathbb{N}}(\mathcal{X}_n)$[9]. This abstract state can be computed by finding the least fixpoint of the function:

  $$f_{x^\sharp} : \begin{array}{ccc} \mathcal{D}^\sharp & \rightarrow & \mathcal{D}^\sharp \\ y^\sharp & \mapsto & x^\sharp \sqcup^\sharp [\![P]\!]^\sharp \left([\![C]\!]^\sharp(y^\sharp)\right) \end{array} \tag{5}$$

  Remark: For now, we only defined abstract operators that can be applied independently of the abstract domain chosen. We will see later that for the assignment or for arithmetic operations the abstract operators will depend on the abstract domain chosen. Depending on this domain, it might be necessary to prove the existence of the least fixpoint for the loop.

- The abstract state for the whole loop statement is obtained when the loop ends, i.e., if $\neg C$ is verified. Thus, this state is computed using the least fixpoint defined above and by supposing that the loop condition is not anymore verified. Finally, this formula is obtained by $[\![\neg C]\!]^\sharp \left(\mu f_{x^\sharp}(x^\sharp)\right)$ and it corresponds to the definition given above.

Provided the abstract operators are sound, this semantics is then sound with respect to the concrete one:

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![P]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![P]\!]^\sharp(x^\sharp)\right).$$

The final abstract state of a program P is finally $[\![P]\!]^\sharp(\top)$[10].

When the abstract operators, as well as the least fixpoint μ, can be computed efficiently, this abstract semantics gives a practical algorithm to actually compute an overapproximation

[9] As we are on a complete lattice, $\sqcup^\sharp_{n\in\mathbb{N}}(\mathcal{X}_n)$ is defined.

[10] The initial state is $\top$ as we do not make any hypothesis on the memory state before the execution of P.

of the concrete semantics. If the abstract semantics is precise enough so that it remains included in some property φ, it then gives a method to prove that this property holds on the program. Of course, the converse is not true. If the abstract semantics is not included in φ, this can either means that property φ is false or that the abstract semantics over-approximates the concrete one too much. However, the computation of the least fixpoint μ is far from trivial. Some techniques to facilitate this computation will be presented in the next section.

Example 7 (Abstract Operators for the Intervals Domain)**.** *Abstract operations $+^\sharp$, $-^\sharp$ and $\times^\sharp : I \times I \to I$ can be defined for arithmetic operations:*

$$+^\sharp : (x, y) \mapsto \begin{cases} \bot_I & \textit{when } x = \bot_I \textit{ or } y = \bot_I \\ [a + c, b + d] & \textit{when } x = [a, b] \textit{ and } y = [c, d] \end{cases}$$

$$-^\sharp : (x, y) \mapsto \begin{cases} \bot_I & \textit{when } x = \bot_I \textit{ or } y = \bot_I \\ [a - d, b - c] & \textit{when } x = [a, b] \textit{ and } y = [c, d] \end{cases}$$

$$\times^\sharp : (x, y) \mapsto \begin{cases} \bot_I & \textit{when } x = \bot_I \textit{ or } y = \bot_I \\ \begin{array}{l}[\min(ab, ac, ad, bd), \\ \quad max(ab, ac, ad, bd)]\end{array} & \textit{when } x = [a, b] \textit{ and } y = [c, d]. \end{cases}$$

They allow to give an abstract semantics for expressions:

$$\begin{array}{l} [\![v]\!]^\sharp(x^\sharp) := x^\sharp(v) \\ [\![X]\!]^\sharp(x^\sharp) := [X, X]\,, \textit{with } X \textit{ a number} \\ [\![E_1 \diamond E_2]\!]^\sharp(x^\sharp) := \diamond^\sharp\left([\![E_1]\!]^\sharp(x^\sharp), [\![E_2]\!]^\sharp(x^\sharp)\right) \qquad \textit{for} \diamond \in \{+, -, \times\} \end{array}$$

which eventually enables to define abstract operators for the intervals domain:

$$[\![v{:=}E]\!]^\sharp(x^\sharp) := \begin{cases} \bot_{\mathcal{I}} & \textit{when } x^\sharp = \bot_{\mathcal{I}} \\ x^\sharp\left[v \mapsto [\![E]\!]^\sharp(x^\sharp)\right] & \textit{otherwise} \end{cases}$$

$$[\![v{:=}?(X, Y)]\!]^\sharp(x^\sharp) := \begin{cases} \bot_{\mathcal{I}} & \textit{when } x^\sharp = \bot_{\mathcal{I}} \textit{ or } X > Y \\ x^\sharp\left[v \mapsto [X, Y]\right] & \textit{otherwise} \end{cases}$$

$$[\![C]\!]^\sharp(x^\sharp) := x^\sharp.$$

Where $x^\sharp\left[v \mapsto [X, Y]\right]$ is the same state as $x^\sharp$ but where the variable v is now associated to the interval $[X, Y]$.

To prove these abstract operators sound with respect to their concrete counterparts, arithmetic operators are first proven sound i.e., defining $\gamma_I : I \to 2^{\mathbb{Z}}$ as the function mapping $\bot$ to $\emptyset$ and $[a, b]$ to $\{x \in \mathbb{Z} \mid a \le x \le b\}$, the following property holds:

$$\forall x^\sharp, y^\sharp \in I, \left\{x \diamond y \;\middle|\; x \in \gamma_I\left(x^\sharp\right), y \in \gamma_I\left(y^\sharp\right)\right\} \subseteq \gamma_I\left(\diamond^\sharp\left(x^\sharp, y^\sharp\right)\right)$$

Then soundness of the abstract semantics of expressions can be established by structural induction on expressions (i.e., with the same γ_I: $\forall x^\sharp \in \mathcal{I}^\sharp, [\![e]\!]\left(\gamma_{\mathcal{I}}(x^\sharp)\right) \subseteq \gamma_I\left([\![e]\!]^\sharp(x^\sharp)\right)$) which *eventually allows to prove the soundness of the abstract operator for assignments. Soundness of abstract operators for random assignments and guards can be obtained more directly.*

It is worth noting that, although sound, the abstract operator for guards is not very precise (for instance, $[\![x \leq 0]\!]^\sharp([-\infty, +\infty])$ could be defined as $[-\infty, 0]$ instead of $[-\infty, +\infty]$). A more precise abstract guard is usually defined using a backward semantics of expressions and iterative reductions [12, 29].

6.3 Kleene iterations and widening

Given computable abstract operators, the only missing element to be able to actually compute an abstract semantics of programs is an effective way to compute least fixpoint operators μ appearing in definition of the semantics of while loops. In practice, exactly reaching the least fixpoint being too hard, only an overapproximation thereof is computed. This leads to a further overapproximated abstract semantics but does not break its soundness.

In the following of this section, the function $f_{x^\sharp} : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ defined in (5) will be reused. This function is assumed to give the abstract semantics of a loop guard and body starting from some abstract value $x^\sharp$ (i.e., for a loop "**while** C **do** P **od**", $f_{x^\sharp} := y^\sharp \mapsto x^\sharp \sqcup^\sharp [\![P]\!]^\sharp \left([\![C]\!]^\sharp(y^\sharp)\right)$). The goal is then to compute an overapproximation of $\mu\, f_{x^\sharp}$, i.e., a $y^\sharp \in \mathcal{D}^\sharp$ such that there exists a fixpoint $z^\sharp$ (i.e.,, $f_{x^\sharp}(z^\sharp) = z^\sharp$) smaller than $y^\sharp$ (i.e.,, $z^\sharp \sqsubseteq^\sharp_{\mathcal{D}} y^\sharp$).

When the lattice $\mathcal{D}^\sharp$ satisfies the ascending chain condition (i.e., any sequence $\left(x^\sharp_n\right)_{n\in\mathbb{N}}$ such that for all $i \in \mathbb{N}$, $x^\sharp_i \sqsubseteq^\sharp_{\mathcal{D}} x^\sharp_{i+1}$, is ultimately stationary), the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$, called *ascending Kleene chain*, is ultimately stationary (since $f^0_{x^\sharp}(\bot) = \bot \sqsubseteq^\sharp_{\mathcal{D}} f_{x^\sharp}(\bot)$ and by an immediate induction using $f_{x^\sharp}$ monotonicity[11], the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ is ascending, hence ultimately stationary). This means, there exists a $N \in \mathbb{N}$ such that $f_{x^\sharp}\left(f^N_{x^\sharp}(\bot)\right) = f^N_{x^\sharp}(\bot)$. $f^N_{x^\sharp}(\bot)$ is then a fixpoint of $f_{x^\sharp}$ and by definition an overapproximation of its least fixpoint: $\mu\, f_{x^\sharp} \sqsubseteq^\sharp_{\mathcal{D}} f^N_{x^\sharp}(\bot)$. Moreover, this value can be simply computed by iterating computations of $f_{x^\sharp}$ starting from $\bot$ until a fixpoint is reached.

Unfortunately, lots of interesting abstract domains do not satisfy the ascending chain condition in which case the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ may not converge in finitely many iterations. This is for instance the case of the intervals abstract domain given in Example 6 (the sequence $([0, n])_{n\in\mathbb{N}}$ is not ultimately stationary for instance). To address this issue, the convergence of the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ will be "accelerated" by a *widening operator*.

A function $\nabla_{\mathcal{D}} : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a widening for the abstract domain $\mathcal{D}^\sharp$ when it satisfies the two following conditions:

- $\nabla_{\mathcal{D}}$ is an overapproximation of the least upper bound $\sqcup^\sharp_{\mathcal{D}}$:

$$\forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp, x^\sharp \sqcup^\sharp_{\mathcal{D}} y^\sharp \sqsubseteq^\sharp_{\mathcal{D}} x^\sharp \,\nabla_{\mathcal{D}}\, y^\sharp;$$

[11] This can easily be proved knowing that the equivalent function for the concrete semantics is monotone (see the proof in section 4.2) and the *soundness property of the abstract operators.*

- for any sequence $\left(x^\sharp_n\right)_{n\in\mathbb{N}}$, the sequence $\left(y^\sharp_n\right)_{n\in\mathbb{N}}$ defined as

$$\begin{cases} y^\sharp_0 := x^\sharp_0 \\ y^\sharp_{n+1} := y^\sharp_n \nabla_{\mathcal{D}} x^\sharp_{n+1} \end{cases}$$

is ultimately stationary.

Example 8 (Widening for the Intervals Domain)**.** *Assuming $\nabla_I : I \times I \to I$ defined as:*

$$x \nabla_I y = \begin{cases} y & \textit{when } x = \bot_I \\ x & \textit{when } y \sqsubseteq^\sharp_{\mathcal{I}} x \\ [a, +\infty] & \textit{otherwise, when } x = [a,b],\ y = [c,d] \textit{ and } a \le c \\ [-\infty, b] & \textit{otherwise, when } x = [a,b],\ y = [c,d] \textit{ and } d \le b \\ [-\infty, +\infty] & \textit{otherwise} \end{cases}$$

the following operator is a widening for the intervals domain (the set of variables $\mathbb{V}$ being finite):

$$x^\sharp \nabla_{\mathcal{I}} y^\sharp = \left(v \mapsto x^\sharp(v) \nabla_I y^\sharp(v)\right).$$

Using a widening, the sequence $\left(y^\sharp_n\right)_{n\in\mathbb{N}}$ defined as $y^\sharp_0 := \bot$ and $y^\sharp_{n+1} := y^\sharp_n \nabla_{\mathcal{D}} f_{x^\sharp}(y^\sharp_n)$ is ultimately stationary which enables to compute an overapproximation of the abstract semantics presented in section 6.2.

The abstract semantics of statements is defined by structural induction on statements:

$$\begin{aligned} &[\![P;Q]\!]^\sharp(x^\sharp) := [\![Q]\!]^\sharp\left([\![P]\!]^\sharp(x^\sharp)\right) \\ &[\![\textbf{if } \mathrm{C} \textbf{ then } \mathrm{P} \textbf{ else } \mathrm{Q} \textbf{ fi}]\!]^\sharp(x^\sharp) := \\ &\qquad\qquad [\![P]\!]^\sharp\left([\![C]\!]^\sharp(x^\sharp)\right) \sqcup^\sharp [\![Q]\!]^\sharp\left([\![\neg C]\!]^\sharp(x^\sharp)\right) \\ &[\![\textbf{while } \mathrm{C} \textbf{ do } \mathrm{P} \textbf{ od}]\!]^\sharp(x^\sharp) := [\![\neg C]\!]^\sharp(y^\sharp_\infty(x^\sharp)) \end{aligned} \tag{6}$$

where $y^\sharp_\infty(x^\sharp)$ is the limit of the ultimately stationary sequence defined by $y^\sharp_0(x^\sharp) := \bot$ and $y^\sharp_{n+1}(x^\sharp) := y^\sharp_n(x^\sharp) \nabla \left(x^\sharp \sqcup^\sharp [\![P]\!]^\sharp\left([\![C]\!]^\sharp(y^\sharp_n(x^\sharp)\right)\right)$.

By structural induction on the statement P we can prove the soundness of the abstract semantics. If the abstract operators are sound, this semantics is then sound with respect to the concrete one:

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![P]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![P]\!]^\sharp(x^\sharp)\right).$$

Remark: In the computation of $y^\sharp_\infty$ the expression "$x^\sharp \sqcup^\sharp .$" may seem useless. Although nothing requires it in the definitions, it is very common that $\bot \nabla x^\sharp = x^\sharp$ and the abstract semantics satisfies $[\![.]\!]^\sharp(\bot) = \bot$. In this case, the first terms of the sequence $y^\sharp(x^\sharp)$ in Equation 6 are $y^\sharp_0(x^\sharp) = \bot$ and $y^\sharp_1(x^\sharp) = x^\sharp$.

Moreover, the widening is often such that for all $x^\sharp$, $y^\sharp$ and $z^\sharp \in \mathcal{D}^\sharp$ if $z^\sharp \sqsubseteq^\sharp x^\sharp$ then $x^\sharp \nabla \left(z^\sharp \sqcup^\sharp y^\sharp\right) = x^\sharp \nabla y^\sharp$. In this case, the definition of the following terms of the sequence is equivalent to $y^\sharp_{n+1}(x^\sharp) = y^\sharp_n(x^\sharp) \nabla \left([\![P]\!]^\sharp\left([\![C]\!]^\sharp\left(y^\sharp_n(x^\sharp)\right)\right)\right)$. This can be used to simplify the computation of abstract semantics of loops.

Example 9 (An Analysis with the Intervals Domain)**.** *This example will detail the analysis, i.e., the computation of the abstract semantics, of the following program with the intervals domain introduced in Examples 6, 7 and 8:*

```
x := 0;
y := ?(0, 12);
while (x ≤ 42) do
        x := x+1;
        y := y−x;
od
```

The analysis starts from $(a) := \top_{\mathcal{I}}$ *and first computes*

$$(b) := [\![x := 0]\!]^\sharp(a) = \{\, x \mapsto [0,0], y \mapsto [-\infty,+\infty] \,\}$$

then

$$(c) := [\![y :=?(0,12)]\!]^\sharp(b) = \{\, x \mapsto [0,0], y \mapsto [0,12] \,\} \,.$$

Since the widening for intervals defined in Example 8 fulfills the requirements of the remark above, the analysis now proceeds computing

$$\begin{aligned}
(d) &:= (c) \,\nabla\, \Big([\![x := x+1; y := y-x]\!]^\sharp \left([\![x \le 42]\!]^\sharp(c)\right)\Big)\\
(e) &:= (d) \,\nabla\, \Big([\![x := x+1; y := y-x]\!]^\sharp \left([\![x \le 42]\!]^\sharp(d)\right)\Big)\\
&\vdots
\end{aligned}$$

until a fixpoint is reached. Hence:

$$\begin{aligned}
(d) &= \{\, x \mapsto [0,0], y \mapsto [0,12] \,\} \;\nabla\; \{\, x \mapsto [1,1], y \mapsto [-1,11] \,\}\\
&= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}\\
(e) &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\} \;\nabla\; \{\, x \mapsto [1,+\infty], y \mapsto [-\infty,11] \,\}\\
&= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\} \,.
\end{aligned}$$

A fixpoint is reached and the abstract semantics after the loop is eventually

$$(f) := [\![x > 42]\!]^\sharp(e) = \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\} \,.$$

(e) *proves that* x *always remains non negative at head of the loop and that* y *always remains below* 12*.*

This section gave a brief overview of the abstract interpretation framework. New analyses can be defined in this framework in a nice way, by "just" defining a new abstract domain, that is by giving: (1) a complete lattice defining the domain, along with a concretization function giving a meaning to its abstract values; (2) abstract operators, simulating their concrete counterparts; (3) a widening operator, to enforce the termination of the analyses. Proving the monotonicity of the concretization function, the soundness of the abstract operators and the soundness and termination properties of the widening then guarantees a sound and terminating analysis.

6.4 Some usual abstract domains and iteration techniques

This section intends to give a brief review of some abstract domains and iteration techniques that are used in current implementations of abstract interpreters. Indeed, the choice of the domain and of the iteration method is very dependent on the properties that have to be verified and the size of the system analyzed (number of variables, number of instructions…).

As presented in section 6.1, an abstract domain creates an overapproximate set of possible values for the variables of the system. There are two types of abstract domains:

- *non-relational abstract domains*, presented in section 6.4.1. This type of domains creates independent relations for all variables. For example, in the intervals domain, each variable is associated with one interval.
- *relational abstract domains* that are presented in section 6.4.2. In this type of domains, different variables can be associated in the same relation. For example, the polyhedra domain that creates relation of the form $x + y < 0$ where x and y are integer variables of a program.

Even with an adapted abstract domain and a widening function, the fixed point computation for a loop can be impossible for huge programs. In this case, some iteration techniques might be used to reduce the needed computation time. These different techniques are presented in section 6.4.3.

Finally, some application of abstract interpretation methods on critical systems or numerical problems will be detailed in section 6.4.4.

6.4.1 Non-relational abstract domains

Non-relational abstract domains consider all the variables independently when the abstract set of possible values is defined. Abstract interpretation methods will compute an abstract state where each variable is associated with an abstract set of value.

The first example of this type of abstract domains is the **intervals domain** that has been presented previously. A variable is associated with an interval $[a, b]\,,\ a, b \in \overline{\mathbb{Z}}$. Every possible value that can be assigned to the variable must be contained in the interval $[a, b]$. As explained in the previous section, this abstraction provides useful information for the analysis, but suffers from convergence problems for the computation of the least fixpoint. It is therefore necessary to use widening operators. The operator presented previously allows to solve the problems for the computation of the fixed point, but in return, some informations are lost. Without the widening operator, it would have been possible to compute an interval with finite bounds, which may no longer be the case (only one finite bound or even none could be computed).

Signs Domain

Similarly to the interval domain with the corresponding widening operators, the *signs domain*, noted S, is also a non-relational abstract domain. This domain will abstract the values taken by a variable by only keeping the information about its sign. We note γ_S the concretization function for this domain. γ can be defined as follows:

$$\begin{aligned}\gamma_S(\top_S) &= \mathbb{Z}\\ \gamma_S(\leq 0) &= [\![-\infty;0]\!]\\ \gamma_S(\geq 0) &= [\![0;+\infty]\!]\\ \gamma_S(0) &= \{0\}\\ \gamma_S(\bot_S) &= \emptyset\end{aligned}$$

There are only 5 different states for this domain. The computation of fixed points for the loop operator can be done easily, therefore this domain does not require a widening operator. The semantics of the addition abstract operator is given below. We will not specify the semantics of all operators for this abstract domain because it is trivial to define.

Abstract operation $+^\sharp_S$: $S \times S \to S$ for the signs domain, can be defined using the following double entry table describing the result of the function $+^\sharp_S(x,y)$ for the different possible values of $(x,y) \in S^2$:

y x	$\top_S$	≤ 0	≥ 0	0	$\bot_S$
$\top_S$	$\top_S$	$\top_S$	$\top_S$	$\top_S$	$\bot_S$
≤ 0	$\top_S$	≤ 0	$\top_S$	≤ 0	$\bot_S$
≥ 0	$\top_S$	$\top_S$	≥ 0	≥ 0	$\bot_S$
0	$\top_S$	≤ 0	≥ 0	0	$\bot_S$
$\bot_S$	$\bot_S$	$\bot_S$	$\bot_S$	$\bot_S$	$\bot_S$

Constants domain

Another example of a non-relational abstract domain is the *constants domain* noted C. In this domain, each variable is associated with a constant. The concretization function γ_C can be easily defined:

$$\begin{aligned}\gamma_C(\top_C) &= \mathbb{Z}\\ \gamma_C(n) &= \{n\}, \text{for all} n \in \mathbb{Z}\\ \gamma_C(\bot_C) &= \emptyset\end{aligned}$$

Unlike the intervals domain, the computation of a fixed-point for the constants domain converges. Indeed, the constant associated with a variable might change at each turn of a loop and the union of the two values will directly give $\top_C$ and then the iteration will eventually terminate.

This abstract domain has the following advantage: it is possible to compute the exact value of a variable when the program deals only with constants. Indeed, when an abstract operation occurs on two known constants the resulting constant is known. For example, the abstract operation $+^\sharp_C$: $C \to C$ corresponding to addition is such that if we know that $x = n$ and $y = m$, $x, y \in C$ and $n, m \in \mathbb{Z}$ then we have $+^\sharp_C(x,y) := n + m$.

The table describing the function $+^\sharp_C(x,y)$ for the different possible values of $x, y \in C$ is the following:

y x	$\top_C$	m	$\bot_C$
$\top_C$	$\top_C$	$\top_C$	$\bot_C$
n	$\top_C$	$m+n$	$\bot_S$
$\bot_S$	$\bot_C$	$\bot_S$	$\bot_S$

This abstract domain therefore allows to compute precisely the value of certain variables. This type of abstraction is useful for optimization consideration. Indeed, if we can compute the abstract state of the result of a given function and we find that it is a constant, then the function call can be replaced directly by the constant. This technique is already used in the `GCC` compiler, for example with the option `-fipa-pure-const` [12].

6.4.2 Relational abstract domains

Using a non-relational abstract domain, the strategy is to find an abstraction for the set of the possible values for the variables. For example, for a variable that takes its values in $\mathbb{Z}$, a possible abstraction is the intervals domain where the set of possible values is overapproximated by an interval. For relational abstract domains, the approach is different. Indeed, the abstraction is not about a subset of $\mathbb{Z}$ for each variable, but about $2^{\mathbb{V}\to\mathbb{Z}}$ i.e. the set that associates the variables to their possible values. The variables will thus not be considered independently. This approach provides more accurate results but the computation of these abstract domains is complex and costly.

Example 10 (Relational vs non-rational abstract domains)**.** *Consider the following condition in a program:*

```
...                                  (a)
if $X + Y \leq 100$ then
...                                  (b)
fi
```

where X and Y are two positive integer variables (i.e., $X, Y \in \mathbb{V}$) and there is no other information known about their content. We note $\rho : \mathbb{V} \to \mathbb{Z}$ the state function that gives the value corresponding to a variable. We also note $x^{\sharp_a}_{\mathcal{I}}$ the abstract state in the intervals domain $\mathcal{I}$ at the line (a) in the code, defined as:

$$x^{\sharp_a}_{\mathcal{I}} := \{X \mapsto [0; +\infty[, Y \mapsto [0; +\infty[\}$$

When executing the guards of the condition, we get some information and we can deduce the new abstract state using backward analysis:

$$x^{\sharp_b}_{\mathcal{I}} := \{X \mapsto [0; 100], Y \mapsto [0; 100]\}$$

The possible value of the two variables are between 0 and 100 (for example $(\rho(X) = 0 \land \rho(Y) = 0) \to (\rho(X) + \rho(Y) \leq 100)$). This abstract domain is therefore an overapproximation because it represents the case where $\rho(X) = 100$ and $\rho(X) = 100$ while it is not possible.

[12] The GCC documentation for this option says: "*Discover which functions are pure or constant. Enabled by default at -O and higher*".

Thus, due to the fact that intervals domain is non-relational, we cannot keep the information that $\rho(X) + \rho(Y) \leq 100$.

Let us now define a relational abstract domain $\mathcal{P}$ as an example. This domain represents all the possible values (i.e., all the ρ functions) that respect some inequalities. At line (a) we then get:

$$x^{\sharp_a}_{\mathcal{P}} := \{\rho | \, (\rho(X) \geq 0) \wedge (\rho(Y) \geq 0)\}$$

At line (b) if we add the information brought by the condition, we get the following state:

$$x^{\sharp_a}_{\mathcal{P}} := \{\rho | \, (\rho(X) \geq 0) \wedge (\rho(Y) \geq 0) \wedge (\rho(X) + \rho(Y) \leq 100)\}$$

With this abstraction, no information was lost in this example. This type of relational abstract domain is called polyhedra domain.

Polyhedra domain

The polyhedra domain $\mathcal{P}$ has been originally introduced by Patrick Cousot and Nicolas Halbwachs [17]. As presented in the previous example, the main principle is to define linear constraints on the variables.

For a program with n variables, a state of the abstract domain $\mathcal{P}$ with m linear constraints is of the form:

$$x^{\sharp}_{\mathcal{P}} := \left\{ \rho \middle| \bigwedge_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \rho(v_j) \leq b_i \right) \right\}$$

where $(a_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}$ is a matrix of $\mathcal{D}^{m \times n}$ (noted A) with $\mathcal{D}$ which can be $\mathbb{N}, \mathbb{Z}$ or $\mathbb{R}$. $(b_i)_{1 \leq i \leq m}$ is a vector from $\mathcal{D}^m$ (noted b), $(v_j)_{1 \leq j \leq n}$ a vector of variables (noted v) i.e., an element of $\mathbb{V}^n$ and $\rho : \mathbb{V} \to \mathcal{D}$ are state functions that return the value that corresponds to a variable.

Remark: The intervals domain is a particular case of the polyhedra domain where for all the rows of the matrix A, there is a unique element which is not zero:

$$\forall i \in [\![1, m]\!], \exists! j \in [\![1, n]\!], (a_{ij} \neq 0) \wedge (\forall j' \in [\![0, n]\!] \backslash \{j\}, a_{ij'} = 0)$$

The concretization function for $\mathcal{P}$ is similar to the definition:

$$\gamma_{\mathcal{P}}(x^{\sharp}_{\mathcal{P}}) := \{\rho : \mathbb{V} \to \mathcal{D} \mid A\rho^n(v) \leq b\}$$

where ρ^n is the same state function as ρ but for variables vectors i.e., $\rho^n(v) = \left(\rho(v_j)\right)_{1 \leq j \leq n}$.

Considering example 10, the advantage of the polyhedra domain is that the abstract domain preserves the information produced by the linear guards. Indeed, when computing the new state, only a new condition needs to be added in the system $Av \leq b$. This results in the following semantics:

$$\left[\!\!\left[\sum_i c_i v_i \leq d \right]\!\!\right]^{\sharp}_{C} (x^{\sharp}_{\mathcal{P}}) := \left\{ \rho^n : \mathbb{V}^n \to \mathcal{D} \middle| \begin{pmatrix} A \\ c_1 \dots c_n \end{pmatrix} v \leq \begin{pmatrix} b \\ d \end{pmatrix} \right\}$$

Where C is the guard and it is defined as $C := \sum_i c_i v_i \leq d$ with $d \in \mathcal{D}$ and $\forall i [\![1, n]\!], c_j \in \mathcal{D}$.

This domain therefore allows an accurate representation of the concrete state for the programs which have linear constraints on the variables. The constraints of this type are often found in the control loops present in embedded and critical systems, areas where formal verification is heavily used to certify the correctness of the system. Unfortunately, the polyhedra domain suffers from complexity problems that limit its use in abstract interpretation tools. Indeed, all the operations over this abstract domain (addition, mutliplication…) have worst-case exponential time and space complexity over the number of variables n (complexity of $O(2^n)$). This domain can thus produce very accurate results but is often not computable on real programs with a non-trivial number of variables. This domain is therefore not used in critical systems, but rather in smaller programs and in some modern compilers.

Octagon

In order to face the complexity problem of the polyhedra domain, the octagon domain $\mathcal{O}$ has been defined [45]. This domain is a subset of the polyhedra domain restricted to the linear constraint of the form $\pm v_i \pm v_j \leq b_{ij}$.

To simplify the representation, the vector v is encoded in v' where $v'_{2i} := -v_i$ and $v'_{2i-1} := v_i, i \in [\![1, n]\!]$. Thus, all the constraints can now be represented by only using the form $v_j - v_{j'} \leq b_i$. For example, $v_i + v_j \leq b_{ij}$ is then represented by $(v'_{2i-1} - v'_{2j} \leq b_{ij}) \wedge (v'_{2j-1} - v'_{2i} \leq b_{ij})$. Using this representation, the octagon domain can be defined:

$$x^{\sharp}_{\mathcal{O}} := \left\{ \rho \middle| \forall i, j \in [\![1, 2n]\!],\ v'_i - v'_j \leq b_{ij} \right\}$$

This domain provides abstract states that are an overapproximation of the abstract states generated by the polyhedra domains. The states are then less accurate. Nevertheless, this domain has a better complexity for every operation (complexity of $O(n^3)$). The abstract states are therefore computable for the more of programs.

State-of-the-art for relational abstract domains

Polyhedra and the particular case of the octagon are the perfect examples to introduce the relational abstract domain. There are other relational abstract domains and it is not the goal of this section to list them exhaustively. Instead, some interesting domains with their strength will be presented.

The *template* abstract domain is another subclass of the polyhedra domain. An abstraction in this domain is an arbitrary polyhedron of a predefined fixed shape. The constraint is then of the form:

$$a_1 v_1 + ... + a_n v_n \leq b$$

where the coefficients $a_1, ..., a_n$ need to be fixed before the analysis. The different operations as abstraction, intersection, join can be solved efficiently using existing solvers. Thus, this domain offers polynomial worst-case time for the different operations.

Another subclass of the polyhedra domains is the *zonotopes domain* [27]. This abstract domain is suitable for the study of numerical invariants and more particularly for real variables. The properties about real numbers are essentials, in particular for embedded system control, because they can bound the error generated during the computation[13].

The *support function* [55] can also be used for numerical abstract domains. A support function noted δ_S where $S \subseteq \mathbb{R}^n$ is defined as:

$$\delta_S(d) = sup\{\langle x, d\rangle : x \in S\}, \forall d \in \mathbb{R}^n$$

With $\langle . \, , . \rangle$ the scalar product operator. This abstract domain is simpler than polyhedra and allows efficient computation for linear operations because it benefits from algorithms on support functions. As for the template domain, this method requires a fixed set of values but without the need of an external solvers.

The *max-plus polyhedra* [2] is used to abstract value and infer relations of the form :

$$\begin{array}{rcl} \texttt{max}(\lambda_0, v_1 + \lambda_1, ..., v_n + \lambda_n) & \leq & \texttt{max}(\mu_0, v_1 + \mu_1, ..., v_n + \mu_n) \\ \text{or } \texttt{min}(\lambda'_0, v_1 + \lambda'_1, ..., v_n + \lambda_n) & \leq & \texttt{min}(\mu'_0, v_1 + \mu_1, ..., v_n + \mu'_n) \end{array}$$

This approach provides more accurate abstraction than octagons on numerical properties. It also offers the possibility to express non-convex properties without any disjunctive representations. This abstraction also allows the computation of memory manipulation properties, for example for programs working on arrays.

6.4.3 Iteration techniques

Classic abstract interpretation based static analysis [13] relies on the *widening* operator. This operator discards some information in order to enforce termination of the analysis. The *narrowing* operator can be defined to partly recover this lost of information. This method often offers a good trade-off between cost and precision of analyses. However, even if impressive improvements were made to widening [21, and references therein] and narrowing [32], they do not always guarantee precise results.

Another approach, that appeared in the last decade in the software verification community, is the use of dedicated mathematical solvers like linear or semi-definite programming as a way to solve some kind of problems in a verification setting. This led to the definition of *policy iterations* [1, 11, 25, 26], as another way to perform overapproximation but trying to achieve better precision than widening-based analyses.

The basic idea of policy iteration is to use numerical optimization tools to compute those bounds that are hard to guess using widening or to retrieve via narrowing. Another advantage of the method is to abstract sequences of program instructions like loop bodies “en bloc”, avoiding intermediate abstractions which can cause irreversible losses of precision.

The goal of this method is to determine the invariants of loops on numerical programs by iterating. Such techniques have been recently developed for the computation of quadratic invariants for linear systems [1, 26].

[13] See also [28] for the definition of an abstract semantics specific for finite precision computations.

Quadratic invariants are invariants of the form $q(v) \leq b, b \in \mathbb{R}, v := (v_1, ..., v_n) \in \mathbb{R}^n$ and:

$$q(v) := \sum_{i<j}^{n} q_{ij} v_i v_j,\ q_{ij} \in \mathbb{R},\ 1 \leq i \leq j \leq n.$$

Even if they are also known for long as quadratic *Lyapunov functions*, from control theory [30, 44], their use in static analysis dates back from less than two decade. Indeed, a majority of the equations representing physical models have a quadratic form. The first famous use of quadratic invariants for static analysis was two dimensional ellipsoids [52] to bound second order filters [22].

Now, let us consider the template domain presented in the previous section. The shape of the templates to be considered for policy iteration depends on the optimization tools used. For instance, linear programming [25] allows any linear templates whereas quadratic templates can be handled thanks to semi-definite programming and an appropriate relaxation [1, 26].

Although very appealing, usage of these techniques in abstract interpretation based static analyzers is limited by two main issues:

Need for Templates Suitable templates have to be provided by the user.

Lack of Integration in the Abstract Interpretation Framework The approach looks rather orthogonal to Kleene iterations with widening classically used in abstract interpretation tools [37]. This prevents cooperation with other abstract domains through reduced products whereas this was demonstrated to be a key feature of abstract interpretation in the Astrée [15] static analyzer presented in section 6.4.4.

There are therefore different methods depending on the abstract domains used but also on the properties and type of invariants that we want to verify.

6.4.4 Application

The majority of abstract interpretation tools can be divided into three parts as presented in an article by Bertrand Jeannet [37].

First, there are front-end applications that will parse programs and generates the abstract syntax tree. For example, Frama-C, that have been presented in section 5.3.2, parses C program and generates an abstract syntax tree using the CIL language.

Secondly, to realize an abstract interpretation analysis, the abstract domains have to be defined and implemented. The sections 6.4.1 and 6.4.2 have presented different abstract domains. Depending of the properties to be verified, the chosen domain will be different. In order to have generic tools, the different abstract domains have to be defined. Generally, they are implemented in libraries that provide them. For example, the Apron [38] library is specialized for the analysis of numerical variables and using mainly different types of polyhedra (intervals, octagons. . .).

Finally, fixpoint solvers have to be defined to compute the invariants of the loop. These solvers will use the different iteration techniques presented in section 6.4.3.

Remark: as presented in figure 2, Frama-C is a tool that generates an abstract syntax tree for the program to be analyzed, and then different plugins are used for the analysis itself, with different static analysis methods, like WP presented earlier. It exists also a plugin called *EVA (Evolved Value Analysis)* that uses abstract interpretation to compute the set of possible values for each variable of the analysed program.

Different tools using abstract interpretation are commercialized and used in the industrial world.

One of them is the Astrée static analyser commercialized by AbsInt [15]. This tool is used by industrials, especially Airbus, for the verification of critical systems. It analyses C programs and verify the absence of runtime errors (RTE) as division by zero or buffer overflows. This tool uses the combinations of several abstract domains: relational domains (such as the octagons) are applied locally on a few variables of the analyzed program, communicating their results back and forth to less expensive non-relational domains (such as the intervals domain) or other instances of relational domains [16].

Another example of industrial tool is Polyspace from Mathworks. This tool can verify C/C++ and Ada programs using static analysis methods. Like Astrée, Polyspace can prove the absence of runtime errors. It also offers the possibility to verify that a program respects some norms of safety and security.

Finally, there is also the compiler which is used for the development of most programs in all domains and not just for embedded and critical systems. As presented above, the abstract interpretation is used to determine if it is possible to perform optimizations.

7 Model checking

Unlike static analysis methods such as deductive method or abstract interpretation, *model checking* is generally not directly used on code. Indeed, static analysis is considered as a syntactic method (on the code) while model checking is a semantics method i.e., it will analyze the meaning of the code and work on a representation of the programs. Model checking analyzes a representation of a real system to verify properties, which generally implies to compute all the possible executions of the system. This technique is used in different fields and at different stages of development to verify liveness and safety properties. For example, it can be used on the architecture of distributed systems to verify that there are no livelocks or on the specification of a hardware design to ensure that error states are not accessible.

Embedded programs are usually represented using Petri Nets or automata. In this section, the process converting such systems to these representations will not be presented and we will suppose that the representation is given. Also, to simplify explanations, we will only consider automata[14].

As for deductive methods, model checking needs a property specification language. This language must allow to describe properties about a state of the system, but also to describe the dynamic behavior of the system. The majority of specification languages used for model checking are based on *temporal logic*, which will be presented in section 7.1. This familiy of logic allows to simply define, in a single property, states that can be true currently but also

[14] Petri nets are more expressive than automata as they can have an infinite state space. However, the verification of the properties generally requires to use only finite nets which often have an equivalent automaton.

in the future.

In section 7.2, different algorithms and techniques for model checking will then be presented. In particular, we will present algorithms that enumerate explicitly all states, but also symbolic model checking that faces the state explosion problem for non-finite state machines.

7.1 Temporal logic

Properties that need to be verified using model checking methods describe the dynamic behavior of the systems. These properties may be expressed into first-order logic as presented as follows in an example.

Example 11. *Suppose that the two logical properties φ and ψ about a system have been defined previously. For this example, we consider that the execution time is discrete. Thus, $\varphi(t)$ means that property φ is true at the time t.*

We want to define the following property: "At any time, if the property φ is true, then the property ψ will necessarily be true in the future".

This property can be expressed as follows in first-order logic:

$$\forall t,\ \varphi(t) \rightarrow (\exists t' > t,\ \psi(t'))$$

As seen in this example, we need quantification on t to express the expected property of the system. Unfortunately, even for this basic property, the notation starts to be cumbersome. It gets even worse for complex temporal properties where the reading of the formal property becomes non-trivial. To deal with this problem and thus lighten the notation, temporal logic has been defined. This logic is similar to natural language, so it is easy to write and understand temporal properties.

There are different logic languages for temporal logic, but we will only present here CTL* (CTL stands for Computation Tree Logic) as presented in [54].

CTL* language

CTL* is a language to describe properties about the dynamic behavior of systems. The properties use *atomic propositions* that are the most basic statements that are defined by the specification of the system. For example, for a system describing the control of a door, the atomic proposition `door_open` means that the door is open and ¬`door_open` that it is closed.

To compose properties, the classical boolean combinators can be used as the two constants `true` and `false`: conjunction ∧, disjunction ∨, negation ¬, implication → and equivalence ↔.

Modal operators are introduced to describe temporal properties. For the following, we consider that φ and ψ are two properties of a state that can be either `true` or `false`. All the properties we will define are associated with a state of the automata and this state will be considered as the current state for the property. There are two types of temporal operators. The first type is operators that define *state properties* i.e., properties on the states that are reachable from the current execution. The second type of operators is those which describe *path properties* i.e., properties about the states from all the executions starting from the current states. We will start by defining the operators that describe state formulae.

The first temporal operator we introduce is $\mathtt{X}$ for neXt. $\mathtt{X}\varphi$ means that the next step of the current execution will satisfy the property φ.

The operators $\mathtt{F}$ (for Finally) and $\mathtt{G}$ for (Globally) are then defined. The property $\mathtt{F}\varphi$ states that it exists in the current execution a future state where φ will be verified. The operator $\mathtt{G}$ is more general because $\mathtt{G}\ \varphi$ means that all the future states of the current execution will satisfy φ. The property defined in example 11 can now be easily expressed by $\mathtt{G}(\varphi \rightarrow \mathtt{F}\psi)$. This temporal property can be read as: “in all the future states ($\mathtt{G}$), if φ is satisfied, then ψ will eventually be satisfied ($\mathtt{F}\psi$)”.

$\mathtt{F}$ and $\mathtt{G}$ are often used together and some specific notations have been defined. $\overset{\infty}{\mathtt{F}}\ \varphi$ is equivalent to $\mathtt{GF}\varphi$ and means that there is always a state where φ is verified. Similarly, $\mathtt{FG}\varphi$ can be noted $\overset{\infty}{\mathtt{G}}\ \varphi$ meaning that the property φ will always be satisfied from a certain point.

We will now introduce operators that deal with two properties. First, $\varphi\ \mathtt{U}\ \psi$ ($\mathtt{U}$ for Until) means that φ will hold until ψ is verified. $\mathtt{W}$ for “Weak until” is defined using $\mathtt{U}$: $\varphi\ \mathtt{W}\ \psi$ is similar to $\varphi\ \mathtt{U}\ \psi$ but ψ may never occur and, in this case, φ will always be verified.

Finally, two operators about path formulae or path quantifiers are defined. $\mathtt{A}\varphi$ ($\mathtt{A}$ stands for All) that can be read as all executions from the actual state will satisfy φ. On the other hand, $\mathtt{E}\varphi$ ($\mathtt{E}$ stands for Exist) means that there is an execution that will satisfy φ. $\mathtt{A}$ and $\mathtt{E}$ are very different from $\mathtt{F}$ and $\mathtt{G}$ because they state about *execution paths* instead of position from a given path. Espacially, $\mathtt{A}$ and $\mathtt{E}$ express path properties while $\mathtt{F}$ and $\mathtt{G}$ express properties on a given path.

With the different operators presented above, almost all temporal properties about a system that might want to be proved using model checking can be defined.

Remark: for complexity reasons, CTL* is never directly used in model checking algorithms. Generally there are two temporal logics that are used and are based on a subset of CTL*:

- *CTL*: This logic is based on CTL* but requires that all the operators X, F, G, U and W are used under the immediate scope of A or E quantifiers. This temporal logic thus describes properties of possible executions and allows to express state formulae. In general, model checking algorithms for CTL are efficient.

- *LTL* (*Linear Temporal Logic*): This logic uses the same operators as CTL*, but without A and E operators. LTL could only examine some given executions and not explore the tree of all the possible executions. There is also a variant for LTL called *PLTL* (P stands for *Propositional*) that limits atomic proposition to boolean values.

Example 12. *We suppose that we have ψ a Boolean property. The temporal propertie φ_1, “at any time, ψ is reachable”, can be expressed in CTL by:*

$$\varphi_1 := AG\ EF\psi$$

But is cannot not be expressed in PLTL. However the property φ_2, “ψ is infinitely often verified”, can be expressed in PLTL as follow:

$$\varphi_2 := \overset{\infty}{\boldsymbol{F}}\, \psi$$

It cannot be expressed in CTL with $\boldsymbol{A}\ \overset{\infty}{\boldsymbol{F}}\, \psi$ *as* $\overset{\infty}{\boldsymbol{F}}\, \psi$ *is defined by* $\boldsymbol{GF}\psi$ *which is not a CTL formulae.*

7.2 Model checking algorithms

Model checking exists for more than 30 years and during this period model checking algorithms have evolved. Indeed, with the development of complex systems, the original algorithms faced limitations due to the explosion of the number of states in the automata representing such sytems. Thus, new techniques have been developed using for instance *symbolic model checking*, in which the states are grouped to reduce their total number and simplify the verification of the properties. For example, Binary Decision Diagrams and logical formulas using SAT/SMT solvers are techniques that use symbolic model checking.

Note: In all this section, we will consider that we want to prove a temporal property φ on the automaton $\mathcal{A}$ i.e., $\mathcal{A} \models \varphi$. φ can be specified using different temporal logics that will be specified. Also, the automaton is defined by $\mathcal{A} := \langle Q, E, T, q_0, l\rangle$, where:

- Q is a finite set of states;
- E is a finite set of transition labels;
- $T \sqsubseteq Q \times E \times Q$ is the set of transitions between states;
- q_0 is the initial state of the automaton;
- l is a function which associates each state of Q to a set of elementary properties which hold in that state.

We note $|Q|$ and $|T|$ respectively the number of states and transitions in the automaton.

7.2.1 States explosion problem

Clarke, Emerson and Sistla [9] were the firsts to introduce the idea of the model checking in 1986. They propose an algorithm for model checking properties expressed in CTL.

CTL

The properties, defined using CTL, describe possible executions of the automata by using the operators A and E from a given state. Instead of considering all the possible executions, the model checking algorithm will base its analysis on the fact that CTL expresses state formulae.

This algorithm is based on the principle of marking each state of the automaton $\mathcal{A}$. Every state will be marked by all subformulae ψ of φ that are satisfied in the considered state. The marking step is equivalent to the memorisation of the different subformulae ψ that are satisfied for every state. We note $m(q) \models \psi, q \in Q$ when ψ is marked (and it is verified) in the state q.

The verification of φ is then done recursively by marking each state where every subformulae ψ are satisfied in the current state or in all the reachable states. Finally the verification of the property $\mathcal{A} \models \varphi$ is equivalent to the verification that the initial state verifies φ i.e., $m(q_0) \models \varphi$.

Example 13. *Let us consider the automaton $\mathcal{A}$ presented in figure 3 and the two atomic properties P_1 and P_2. We suppose that these properties are verified in the state q_2 i.e., $P_1, P_2 \in l(q_2)$.*

First, we want to verify the property $\varphi := EX(P_1 \wedge P_2)$. We start the `marking` algorithm recursively on each state (`marking` on q_0 will call `marking` on q_1 that will call `marking` on q_2 and q_3):

- *`marking` on q_2 will not evaluate directly φ because the state does not have successors. Instead, it will verify the sub-formula $P_1 \wedge P_2$. As we have $P_1, P_2 \in l(q_2)$, the states can be marked with $m(q_2) \models P_1 \wedge P_2$.*
- *consider the state q_1. The `marking` function will evaluate $EX\Phi$, with $\Phi := P_1 \wedge P_2$, by verifying that at least one (for the E operator) of its successors (for the X operators) is marked with Φ which is the case of q_2. The state q_1 is then marked i.e., $m(q_1) \models \varphi$.*
- *Finally, the state q_0 can be marked with $m(q_0) \models \varphi$ as we have $m(q_1) \models \varphi$ and q_1 is the direct and only successor of q_0.*

The property φ is then verified for the automata ($\mathcal{A} \models \varphi$).

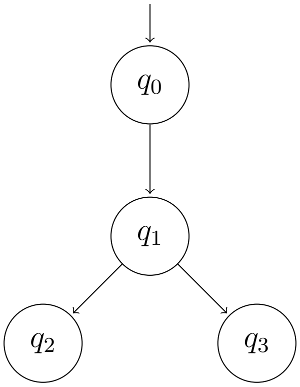


Figure 3: Automaton used for the example

We now want to prove the property $\psi := AX\Phi$ where $\Phi := P_1 \wedge P_2$. With the same reasoning as previously, $m(q_2) \models \Phi$ holds. However, if we consider the state q_1, the marking algorithm should now verify that all the successor states verify Φ as the A temporal operator means that all the executions must satisfy the property and q_2 and q_3 are two possible executions from q_1. As the state q_3 does not satisfy the property, then q_1 does not satisfy ψ.

Recursively, we can deduce that q_0 does not satisfy ψ and thus $\mathcal{A} \not\models \psi$

These two examples present the idea behind the `marking` function. For more details and an example of the `marking` algorithms, see the explanation given in [54].

Remark: in the example, the real algorithm has not been exactly followed. Indeed, there is a function call for each subformula of the property we want to prove, but also for each state or each transition. Verification of CTL properties, using a `marking` algorithm has then a complexity in time of $O(|\mathcal{A}| \times |\varphi|)$ where $|\varphi|$ is the number of subformulae of φ and $|\mathcal{A}|$ is the size of the automaton i.e., $O(|Q| + |T|)$.

PLTL

Unlike CTL which describes state formulas, PLTL defines formulas about paths. Even on finite automaton, there may be an infinity of possible executions and they can be of infinite length. With these differences, the `marking` algorithm cannot be used for PLTL. Instead, the approach adopted is one coming from language theory, which is presented in the book [57].

For each PLTL formula, there is an equivalent ω-regular expression[15]. For example the formula $G(\neg\psi \to X\psi)$, with ψ atomic property, can be described with the ω-regular expression $((\psi)^*(\neg\psi).\psi)^\omega$. Thus, for every PLTL formula φ, there exists an automaton that recognizes every execution which satisfy φ. We note this automaton $\mathcal{B}_\varphi$. Such recognition automaton is defined with the quintuplet $\mathcal{B}_\varphi := \langle Q, E, T, q_0, F\rangle$ similarly as the automata defined previously. The only difference is that l is replace with F which is a set of accepting states ($F \subseteq Q$) and E is the set of all Boolean formulae built from the atomic propositions of φ. The accepting states are shown in the diagrams with a double circle (see 4b). An execution is accepted if and only if it passes an infinite number of times on an accepting state. This definition therefore allows to recognize infinite words.

We defined the operation of synchronization of automata $\mathcal{A}\otimes\mathcal{B}$ with $\mathcal{A} := \langle Q_\mathcal{A}, E_\mathcal{A}, T_\mathcal{A}, q_0^\mathcal{A}, l\rangle$ and $\mathcal{B}_\varphi := \langle Q_\mathcal{B}, E_\mathcal{B}, T_\mathcal{B}, q_0^\mathcal{B}, F_\mathcal{B}\rangle$. This operation produces an automaton that recognizes the execution of a systes represented by $\mathcal{A}$ that verify that satisfies φ. The resulting automaton is defined by $\mathcal{A} \otimes \mathcal{B} := \langle Q, E, T, q_0, F\rangle$ where:

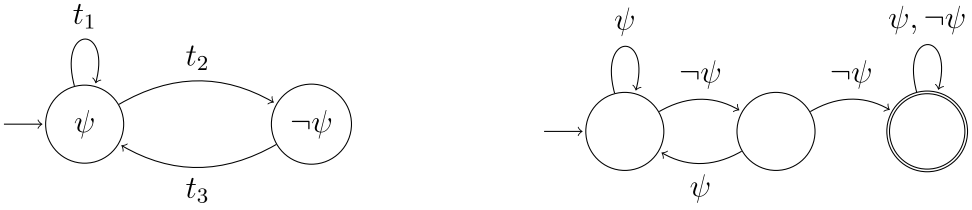
- $Q := Q_\mathcal{A} \otimes Q_\mathcal{B}$ i.e., $Q := \{(q_\mathcal{A}, q_\mathcal{B}) | q_\mathcal{A} \in Q_\mathcal{A} \wedge q \in Q_\mathcal{B}\}$
- $E := E_\mathcal{A} \otimes E_\mathcal{B}$ i.e., $E := \{(e_\mathcal{A}, e_\mathcal{B}) | e_\mathcal{A} \in E_\mathcal{A} \wedge e \in E_\mathcal{B}\}$
- $T := \Big\{(q_\mathcal{A}, q_\mathcal{B})\ (e_\mathcal{A}, e_\mathcal{B})\ (q'_\mathcal{A}, q'_\mathcal{B}) \Big| (q_\mathcal{A}\ e_\mathcal{A}\ q'_\mathcal{A}) \in T_\mathcal{A} \wedge (q'_\mathcal{B}\ e_\mathcal{B}\ q'_\mathcal{B}) \in T_\mathcal{B} \wedge l(q_\mathcal{A}) \models e_\mathcal{B}\Big\}$
- $q_0 := (q_0^\mathcal{A}, q_0^\mathcal{B})$
- $F := Q_\mathcal{A} \otimes F_\mathcal{B}$

These automata for the recognition of temporal formulae are used to verify if the automaton $\mathcal{A}$ satisfies the PLTL property φ. Indeed, we first defined the automaton $\mathcal{B}_{\neg\varphi}$ and then, the automaton $\mathcal{A}$ and $\mathcal{B}_{\neg\varphi}$ are synchronized (the transitions evolve simultaneously) . The synchronization generates an automaton noted $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$. This automaton recognizes the executions that $\mathcal{A}$ could generate and that $\mathcal{B}_{\neg\varphi}$ recognize i.e., $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$ recognizes all the possible execution of $\mathcal{A}$ where the property φ is not satisfied.

[15] ω-regular expressions are expressions that can describe infinite words. We note a^* and a^ω words of such language that contain respectively a finite or infinite number of repetitions of a.

The verification that $\mathcal{A}$ satisfies φ (formally $\mathcal{A} \models \varphi$) is now reduced to the analysis of the language recognized by $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$. If it can be proved that $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$ recognize an empty language then we have $\mathcal{A} \models \varphi$.

Example 14. *We consider the automaton presented in figure 4a that represents a system, with an atomic property ψ to be verified. This property is only verified in the initial state of $\mathcal{A}$. The labels on the transition (t_1, t_2 and t_3) are present only to facilitate the understanding of the construction of the synchronized automaton, but they do not correspond to conditions on the transitions.*



(a) Automaton $\mathcal{A}$ representing the system

(b) Automata $\mathcal{B}_{\neg\varphi}$

Figure 4: Automata considered

We want to prove that the automaton $\mathcal{A}$ satisfies the property $\varphi := G(\neg\psi \rightarrow X\psi)$ (i.e., every time where ψ is false, it will be true in the next state). The automaton $\mathcal{B}_{\neg\varphi}$, defined in the figure 4b, recognizes executions where φ is not satisfied i.e., when ψ is false two consecutive steps.

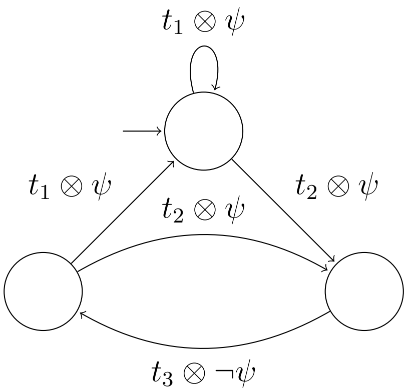


Figure 5: The synchonized automaton $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$

The synchronized automaton $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$ can be easily computed by executing simultaneously executing the two automata. The figure 5 presents the result without the impossible transitions (for example, there cannot be any transition of the form "_ $\otimes \neg\psi$" from a state where ψ holds in the automaton $\mathcal{A}$).

The synchronized automaton is complete (from every state, there is a possible transition for all the possible inputs of t) and there are no accepting states. Thus, $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$ cannot recognize any executions. The automata $\mathcal{A}$ therefore satisfies the PLTL property φ.

Remark: in the worst case, the automaton $\mathcal{B}_{\neg\varphi})$ has a size of $O(2^{|\varphi|}$. The construction of the synchronized automaton $\mathcal{A} \otimes \mathcal{B}_{\neg\varphi}$ is equivalent to a product of the two automata, the size is therefore of $O(|\mathcal{A}| \times |\mathcal{B}_{\neg\varphi}|)$. The verification of a PLTL property φ on an automaton $\mathcal{A}$ has thus a time complexity of $O(|\mathcal{A}| \times 2^{|\varphi|})$ in the worst case.

The two previous algorithms use enumerative methods on all the states of an automaton in order to verify CTL or PLTL properties. This result of a time complexity in the worst case of $O(\mathcal{A} \times |\varphi|)$ for CTL and $O(|\mathcal{A}| \times 2^{|\varphi|})$ for PLTL. Thus, the verification of properties for automata can be done efficiently. Unfortunately, model checking is often applied on automata that are the result of the products (or synchronization) of smaller automata. The product of automata is an exponential operation for the number of states of the generated automaton. The size of the automata is a known problem, called *state explosion problem*, that limits the utilisation of model checking.

7.2.2 Symbolic model checking

The previous techniques rely on the enumeration of the whole state space that causes the *state explosion problem*. In order to overcome this problem, there exists another category of model checking techniques called *symbolic model checking*. The main idea is to represent groups of states and transitions symbolically in order to simplify the verification of temporal properties and therefore reduce the computation time. There are different symbolic representations: binary decision diagrams or logical formulas tha can be verified using SAT/SMT solvers.

Binary Decision Diagrams (BDD)

A Binary Decision Diagram or BDD is a compact data structure to represent a set of boolean vectors, but it can also be seen as a boolean function. Indeed, for a given boolean formula ψ using n variables $(b_1, ..., b_n)$, a BDD represents all the vectors of $\mathbb{B}^n$ that satisfy the formula ψ. The BDD is also equivalent to a function $f_\psi : \mathbb{B}^n \to \mathbb{B}$ such that $f_\psi(v)$ is true if and only if $v \models \psi$, with $v \in \mathbb{B}^n$. The set of vectors represented by the BDD can then be defined as:

$$\left\{ v \in \mathbb{B}^n \middle| f_\psi(v) \right\}$$

BDDs are represented using a binary decision tree, as presented in the figure 6. For a given order $(b_1 < ... < b_n)$, the first node will correspond to the evaluation of the variable b_1. Its left son (resp. right son) corresponds to a sub-binary decision tree about ψ, knowing that b_1 is false (`F`) (resp. true (`T`)). The whole tree is built recursively over the variables. Finally, ψ can be evaluated and the leaves `F` or `T` corresponding to the result are referred by the nodes b_n. Thus, each boolean vectors of $\mathbb{B}^n$ corresponds to a path in the tree and the vector satisfy ψ if and only if the final leaf of all the paths is `T`.

An efficient way to store the BDD is to use a tuple $\langle l_1, ..., l_{2^n} \rangle$ where l_i are the ordered values of the leaves. For example, l_1 (resp. l_{2^n}) is the value of the leaf when all the variables are false (resp. true). Unfortunately, for bigger systems with a large number of variables, this solution is unfeasible. Indeed, if we code the leaves on one bit, a BDD with 33 variables would need 1GB of memory.

A currently used solution is to *reduced* the BDD. The idea is the following: if the two sub-binary decision trees of a node are the same, the node is removed and replaced with the sub-trees. Depending on the boolean formula and the order chosen for the variables, this method allows to reduce the number of nodes in the BDD. Also, to reduce the numbers of leaves, there is only two leaves that are kept, one for `F` and one for `T`, instead of 2^n. The reduction of BDD therefore allows, under certain conditions, to represent the BDD efficiently in terms of memory used, but also in terms of computation time required to verify a Boolean vector as the binary decision tree is smaller.

Example 15. *We want to define the BDD for the Boolean formula $\psi := \neg a \wedge (b \vee \neg c)$ where a, b and c are Boolean variables, with the order $a < b < c$. We can define the Boolean function as follows:*

$$f_\psi(v) := \neg a \wedge (b \vee \neg c), \text{ with } v := \begin{pmatrix} a \\ b \\ c \end{pmatrix} \in \mathbb{B}^3$$

The BDD corresponding to this formula is given in figure 6 and can be encoded with the tuple $\langle$`T, F, T, T, F, F, F, F`$\rangle$.

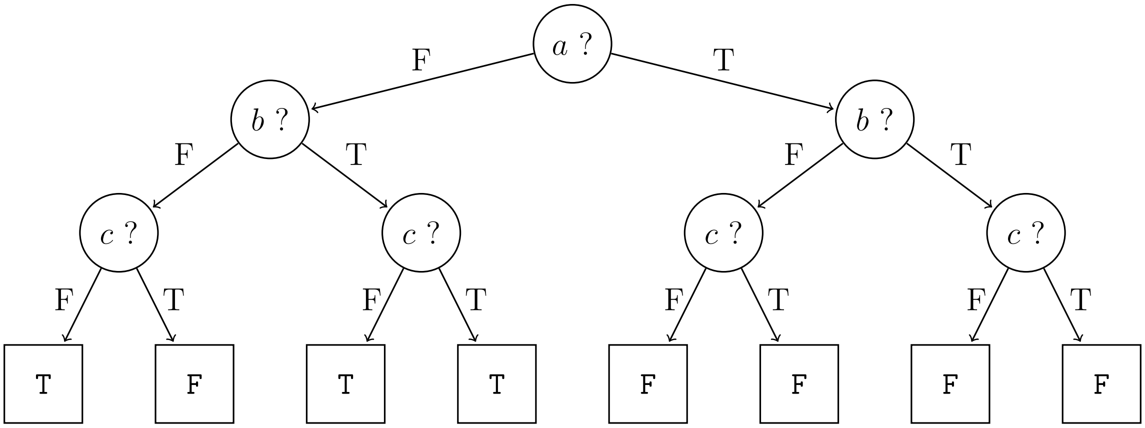


Figure 6: A binary decision tree for $\neg a \wedge (b \vee \neg c)$

The BDD reduced is presented in figure 7.

In this simple example, using a reduced BDD has divided by two the number of states. The representation is therefore more compact and the mean time needed to verify a vector is now about 2.25t[16] *against 3t for the unreduced BDD.*

The compact representation of reduced BDDs is preferred for the verification of properties using model checking in order to decrease the computation time required. Model checking algorithm to verify the property φ about a system can be summarized in 3 steps:

1. **Initialisation**: a BDD B_0 is created to represent the set of the initial states i.e., q_0.

[16] t is the time needed to complete a transition. Also, we consider that for each node, the two branches have the same probability

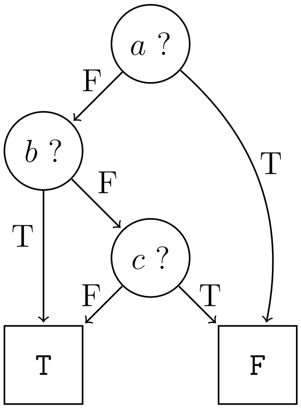


Figure 7: Reduced BDD for $\neg a \wedge (b \vee \neg c)$

2. **Iteration**: for each iteration i, we compute the set of accessible states from the initial states B_i with the function $Acc(B_i)$. These new states are added to the BDD ($B_{i+1} := B_i \bigcup Acc(B_i)$). Then, the intersection I_{i+1} between B_{i+1} and the set of states that satisfy $\neg\varphi$ is computed (I_{i+1} might be represented as a BDD also).

3. **Termination**: the algorithms stops in two cases. First the set I_{i+1} is not empty. This means that there exists an accessible state which do not satisfy the property φ and thus the system. The second case is when a fixed point is reached ($B_{i+1} = B_i$). The system satisfies the property φ as all reachable states satisfy the property.

BDD allow therefore to represent sets of states that respect certain conditions in an efficient way. Unfortunately, binary decision trees cannot always be greatly reduced. When the verified system begins to be large, this method shows its limits in term of memory needed to store the BDD.

SAT/SMT Solving

Another representation for symbolic model checking is to use *logic formulae*. The formulae encode the set of states, the transitions between states and properties to be verified. The verification of the system is then computed with SAT solvers. This approach also allows to define more complex properties using for instance linear arithmetic by using SMT (*Satisfiability Modulo Theory*) solvers [4], like CVC4, Z3 or Alt-Ergo.

Bounded Model Checking or BMC [6] is a method that uses SAT or SMT solvers. This method only allows finding counter-examples of execution for liveness and safety properties. A liveness property is not satisfied if it exists a path to a loop without the specified state i.e., there is a path that will never reach the desired state. To verify that a safety property is not held, an execution that reaches an unwanted state must be found.

The idea of this technique is to verify that there are no counter-examples of the property for the executions whose length is bounded by a fixed k. If there no bug has been found, the maximal execution length k is increased. The problem of finding counter-examples can be encoded with propositional formulas and then verified with SAT solvers. The verification with solvers is done efficiently as SAT will not encounter the problems of explosion of the memory used that appears using BDDs.

Unfortunately, this method will iterate indefinitely if there is no counterexample. The user has thus to provide an upper bound for k in order to end the verification. BMC allows therefore to prove that certain systems do not satisfy a property by finding a counter-example but it cannot prove that the property is satisfied. Indeed, if the method has been used with a fixed k and no bug is found, it might exist an execution with a length of $k+1$ that is a counter-example for the property.

Thus, BMC allows to verify efficiently some properties that do not hold but they cannot be verified with BDDs. However, there are also properties that BMC methods cannot verify but BDDs can. These two methods are therefore not used independently but are rather complementary.

An extension of BMC is k-induction which generalized the induction to k transitions. For example, we want to prove that a property $\varphi(n), n \in \mathbb{N}$ is always verified. A k-induction proof consists to verify the statement I_k defined as follow:

$$k \geq 1,\ I_k := \left(\wedge_{i=0}^{k-1} \varphi(i)\right) \wedge \left(\forall n \geq 1 \left(\wedge_{i=0}^{k-1} \varphi(n+i)\right) \rightarrow \varphi(n+k)\right)$$

And then we have the properties $I_k \rightarrow \forall n,\ \varphi(n)$. This method allows to find counter-examples like BMC, but also to prove invariants on a system.

An implementation of k-induction can be found in Pkind [40], a model checker using k-induction. Pkind allows to verify invariants on Lustre programs [17].

8 Proof assistants

A proof assistant is a software used to write and then verify formal proofs. Mainly used in Computer Science, it can also be used in mathematics. It allows in particular to prove properties related to the execution of a program or mathematical theorems.

Before proof assistants, writing formal proofs in mathematics can be long and tedious, but it has the advantage of making the theorem irrefutable once it has been proven and verified. Unfortunately, the proof of certain trivial properties are often skipped, making the proof questionable. Indeed, there exist some cases where the proof has turned out to be incorrect years after its publication. For these reasons, many projects have been launched to facilitate the writing of complete formal proofs and their verification.

The Automath project [18] was first created by Nicolaas de Bruijn, in 1967. The goal of this project was to create a language which can express all mathematical formulae, and then develop a system for the verification of theorems. In 1976, Appel Kenneth and Haken Wolfgang were the firsts to prove a theorem with the help of a computer (the Four Color Theorem [3]).

These projects were the precursors of current proof assistants. However, it was not until the years 1990-2000 that the proof assistants were developed further. Currently, the most developed are HOL/Isabelle (project of the Cambridge university), and Coq by Inria.

[17] Lustre is a synchronous language [5] defined by N. Halbwachs et al. [31]

Over time, proof assistants have been enriched: they have become more efficient, and allow to prove more complex problems. These softwares are called *proof assistants* because they only allow to check if a proof is correct and, usually, they cannot find it automatically[18].

Unfortunately, the use of a proof assistant can be disputed. Indeed, they are complex softwares and we cannot be sure that they do not have any bugs. A buggy proof assistant is equivalent to something false that can demonstrate the absurd. In order to guarantee a certain level of trust in the tools, some proof assistants, such as Coq, have an architecture that can be divided into two parts:

- First, the front-end that provide an interface or a language for the user to write this proof. The proof is then converted into a list of inference rules to apply and axioms. The front-end is a complicated piece of software trying to make the life of the user as easy as possible while still producing the formal proofs in the minimal language used by the next part.
- The kernel of the proof assistant. Its role is to verify that the inference rules are correctly applied and that the axioms are sufficient to guarantee the proof. Thus, this is the core element to ensure that proofs validated by the proof assistant are actually correct. This element is in general a simple software that can be easily verified and give trust in the tools.

There exist currently different concrete projects that use proof assistants. For instance, CompCert is a C compiler proven with Coq used at Airbus. Sel4 is a operating system microkernel that have been proven with Isabelle. Finally, KeYmaera X is a proof assistant, using a specific logic for Cyber-Physical systems [50], that has been used to model and prove some properties of the Crazyflie quadcopter drone [39].

[18] There are tools that allow to automatically prove certain properties only if they are simple.

Part III
Perspectives of the thesis

The objective of this PhD thesis is to apply formal verification methods on a UAV autopilot. The Paparazzi autopilot developed at ENAC has been chosen as a case study. In this context, the immediate perspectives of the thesis are the following:

1. *proof of Paparazzi mathematical library*: Paparazzi embeds a library for mathematical computation, mainly for UAV state conversion between different representations purposes (see `https://github.com/paparazzi/paparazzi/tree/master/sw/airborne/math` for more detail). This library is rather rich and provides basic algebra functions (operations on matrices for instance), different representation for real (floating point values, fixed point values) and several representation for the UAV state (geodesic etc).

   Our first objective will be to perform verification of the corresponding C code using the Frama-C platform and its plugins on two points:

   - *AoRTE verification*, i.e. Absence of RunTime Errors like integer overflow for instance (for fixed point representation of real) or null pointer dereferencing (pointers are heavily used for formal parameters of the algebra functions).
   - *functional verification*, i.e. verification of the correctness of the library code. Notice that in order to achieve such verification, we should first be able to *specify* all the library functions using ACSL. We will first consider for this proof that floating point values are real values to ease the proofs. We will then consider that floating point values are not real values, which will complicate a lot the proof.

2. *proof of Paparazzi flight plan generator.* Paparazzi users may define flight plans in an XML file, describing waypoints and using a small programming language and standard navigation patterns. Such plans are then translated into "real code" and embedded in the autopilot before compilation. There is no guarantee actually that the flight plan compilation process is correct, i.e. that the embedded code has the expected behavior. It is therefore interesting to prove this compilation process.

   Such verification implies several points:

   - first, the syntax and semantics of the flight plan description language must be clearly defined.
   - the compiler, i.e. the software that translates the flight plans described in XML files into C code, must be verified according to the previously defined semantics. Current approaches for such a verification [8, 43] use a proof assistant like Coq to describe both source and target languages semantics, implement the compiler and prove its correctness.

     Notice that Lelio Brun, one of the author of the Velus compiler [8], will join ISAE-SUPAERO as a postdoc in January 2021. The current thesis could benefit his expertise on this point.

3. *proof of Paparazzi autopilot generator*. Paparazzi proposes a C code generator for autopilot (only available for rovers at the moment). Users can define their autopilot behavior using state machines described in an XML file. Similarly to the previous point, we could also prove that the generation of the C code is correct, i.e. that the C code has the same semantics than the state machine used to generate it.

Part IV
Appendix

List of Figures

List of Tables

References

[1] Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *ESOP*, pages 23–42, 2010.

[2] Xavier Allamigeon, Stephane Gaubert, and Eric Goubault. Inferring min and max invariants using max-plus polyhedra. In *SAS*, pages 189–204, 2008.

[3] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. *Bulletin of the American mathematical Society*, 82(5):711–712, 1976.

[4] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. In *Proceedings of The IEEE*, pages 64–83, 2003.

[6] Armin Biere. Bounded model checking. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 14, pages 457–481. IOS Press, February 2009.

[7] Allan Blanchard. Introduction à la preuve de programmes c avec frama-c et son greffon wp. 2020.

[8] Timothy Bourke, Lélio Brun, and Marc Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, January 2020.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[10] Edmund M. Clarke and Jeannette. M. Wing. Formal methods: State of the art and future directions, 1996.

[11] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, pages 462–475, 2005.

[12] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam, 1999.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[15] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[16] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *ASIAN*, pages 272–300. Springer, 2006.

[17] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[18] Nicolaas Govert De Bruijn. A survey of the project automath. In *Studies in Logic and the Foundations of Mathematics*, volume 133, pages 141–161. Elsevier, 1994.

[19] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[20] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[21] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for c programs with aspic and c2fsm. *Electr. Notes Theor. Comput. Sci.*, 267(2):3–13, 2010.

[22] Jérôme Feret. Static analysis of digital filters. In *ESOP*, number 2986 in LNCS. Springer, 2004.

[23] Jean-Christophe Filliâtre. *Deductive Program Verification.* Thèse d'habilitation, Université Paris-Sud, December 2011.

[24] Carlo Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification and examples. *ACM Computing Surveys*, 46(3), 2014.

[25] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *ESOP*, pages 237–252, 2007.

[26] Thomas Martin Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *SAS*, pages 271–286, 2010.

[27] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *CAV*, pages 627–633, 2009.

[28] Éric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *VMCAI*, pages 232–247, 2011.

[29] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin Heidelberg, 1992.

[30] Wassim M. Haddad and Vijay S. Chellaboina. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach.* Princeton University Press, 2008. Lyapunov, non linéaire.

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[32] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In *SAS*, pages 198–213, 2012.

[33] Gautier Hattenberger, Murat Bronz, and Michel Gorraz. Using the Paparazzi UAV System for Scientific Research. In *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, pages pp 247–252, Delft, Netherlands, August 2014.

[34] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[36] Éric Jaeger. *Study of the Benefits of Using Deductive Formal Methods for Secure Developments. (Etude de l'apport des méthodes formelles déductives pour les développements de sécurité).* PhD thesis, Pierre and Marie Curie University, Paris, France, 2010.

[37] Bertrand Jeannet. Some experience on the software engineering of abstract interpretation tools. *Electr. Notes Theor. Comput. Sci.*, 267(2):29–42, 2010.

[38] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[39] Alexis LE PENVEN Julien BORTOLUSSI and Christophe GARION. Proving crazyflie properties with keymaera x.

[40] Temesghen Kahsai and Cesare Tinelli. Pkind: A parallel k-induction based model checker. In *PDMC*, pages 55–62, 2011.

[41] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Astree: Proving the Absence of Runtime Errors. In J.C. Laprie, editor, *Embedded real time software and systems - ERTS2 2010*, Toulouse, France, May 2010. AAAF, SEE, SIA.

[42] Bronislaw Knaster. Un thórèm sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6, 1928. with Alfred Tarski.

[43] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE.

[44] Aleksandr Mikhailovich Lyapunov. Problème général de la stabilité du mouvement. *Annals of Mathematics Studies*, 17, 1947.

[45] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[46] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[47] Amat N. A new approach for the symbolic model checking of petri nets. Master's thesis, University of Grenoble, 2020.

[48] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer.* Undergraduate Topics in Computer Science. Springer, 2007.

[49] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2001.

[50] André Platzer. *Logical Foundations of Cyber-Physical Systems.* Springer, 2018.

[51] Pierre Roux. *Analyse statique de systèmes de contrôle commande : synthèse d'invariants non linéaires.* PhD thesis, 2013. Thèse de doctorat dirigée par Wiels, Virginie et Garoche, Pierre-Loïc Sureté de logiciel et calcul de haute performance Toulouse, ISAE 2013.

[52] Pierre Roux, Romain Jobredeaux, Pierre-Loïc Garoche, and Éric Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *HSCC*, pages 105–114, 2012.

[53] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, May 2010.

[54] Ph. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[55] Yassamine Seladji and Olivier Bouissou. Numerical abstract domain using support functions. In *NFM*, 2013.

[56] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2), 1955.

[57] Moshe Y. Vardi. *An automata-theoretic approach to linear temporal logic*, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.