Specifying and verifying high-level requirements on large programs : application to security of C programs

Virgile Robles

université
PARIS-SACLAY

Specifying and Verifying High-Level Requirements on Large Programs : Application to Security of C Programs

Spécifier et vérifier des exigences de haut niveau sur des programmes importants : application à la sécurité des programmes C

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 573, Interfaces : approches interdisciplinaires, fondements, applications et innovation
Spécialité de doctorat : Informatique
Graduate School : Sciences de l'Ingénieurie et des Systèmes
Référent : CentraleSupélec

Thèse préparée dans l'unité de recherche du MICS (Université Paris-Saclay, CentraleSupélec), sous la direction de Pascale LE GALL, professeure, le co-encadrement de Virgile Prevosto, ingénieur-chercheur et de Nikolai KOSMATOV, ingénieur-chercheur et la co-supervision de Louis RILLING, ingénieur-chercheur.

Thèse soutenue à Paris-Saclay, le 21 janvier 2022, par

Virgile ROBLES

Composition du jury

Claude Marché Directeur de recherche, INRIA & LMF, Université Paris-Saclay	Président
Alain Giorgetti Maître de conférences HDR, FEMTO-ST, Université de Franche-Comté	Rapporteur & Examinateur
Frédéric Loulergue Professeur, LIFO, Université d'Orléans	Rapporteur & Examinateur
Mathieu Jaume Maître de conférences HDR, LIP6, Sorbonne Université	Examinateur
Pascale Le Gall Professeure, CentraleSupélec, Université Paris-Saclay	Directrice de thèse

THESE DE DOCTORAT
NNT : 2022UPAST001

Remerciements

Je remercie Alain Giorgetti et Frédéric Loulergue d'avoir accepté de rapporter ma thèse, qui plus est pendant leurs vacances de Noël. Leur travail m'aura permis de corriger de nombreuses petites erreurs et imprécisions dans le manuscrit. Merci également à Claude Marché d'avoir accepté de présider le jury de ma soutenance et Mathieu Jaume d'avoir bien voulu en faire partie.

Cela va de soi, je remercie chaleureusement toute mon équipe d'encadrement. Tout d'abord Nikolai Kosmatov, qui malgré sa migration du CEA vers Thalès m'aura accompagné quotidiennement pendant cette thèse de trois ans et le stage qui l'a précédée, aura toujours su m'aider lorsque j'étais bloqué et aura été d'une aide précieuse lors de la rédaction d'articles, contribuant souvent à des heures improbables. Virgile Prevosto, mon ancre au CEA avec qui j'ai eu le plaisir de beaucoup voyager, a toujours su apporter sa bienveillance ainsi que des remarques constructives même (surtout !) concernant des sujets très pointus ou obscurs. Je le remercie également pour avoir trouvé cet acronyme douteux que j'aurai tant écrit dans ce manuscrit. Louis Rilling, mon tuteur à la DGA, a pu donner un point de vue différent sur mes travaux, et a suggéré des cas d'études qui se sont avérés cruciaux. Et enfin Pascale Le Gall, ma directrice de thèse, a pu m'apporter son expérience du monde de la recherche, me faire découvrir celui de l'enseignement, et sans nécessairement suivre le détail de mes travaux m'aura permis de mieux les communiquer.

Je remercie mes professeurs qui à Bordeaux m'ont conduit à faire cette thèse. En particulier je tiens à remercier David Renault pour la passion qu'il m'a partagée pour les types, l'algorithmique, la vérification logicielle et l'informatique obscure, passion qui continue encore aujourd'hui. Bien entendu je remercie également Frédéric Herbreteau, qui m'aura donné

envie de poursuivre mes études, fait découvrir l'université et la recherche, et de manière cruciale m'aura permis de faire mon stage de fin d'études au CEA.

Une thèse serait beaucoup moins agréable sans toutes les personnes qui sont autour et rendent le quotidien plus coloré. À commencer par mon ami et voisin de bureau depuis notre premier jour de stage, Thibault, que je remercie pour son amitié et nos conversations. Je n'oublie pas les autres qui ont égayé notre bureau : Alexandre, pour son rôle de mentor au CEA, Lionel, une grande inspiration pour de nombreuses parties de ma thèse et Christophe, qui m'a toujours impressionné par la quantité de travail qu'il abat.

Je tiens également à remercier les (post-)doctorant·e·s et permanent·e·s du laboratoire, pour leur contribution à la vie sociale à l'intérieur et à l'extérieur du laboratoire, malgré ma présence amoindrie pendant la deuxième moitié de ma thèse. Dans le désordre : David, Quentin, Allan, Maxime, Valentin, Florent, Pauline, Lesly, Yaëlle, André, Olivier, Myriam, Julien & Julien, Dara, Soline, Guillaume et bien d'autres... Merci à tous·tes pour toutes ces pauses café, soirées escalade, Ghibli, randonnées et autres expériences culinaires.

Au sein du laboratoire, au CEA comme à Centrale, je remercie également chaudement les secrétaires et assistant·e·s qui auront rendu facile l'organisation du quotidien et des missions, répondant toujours patiemment à mes incompréhensions administratives. Je pense notamment à Frédérique et Suzanne.

Je remercie tous les amis, dont certain·e·s déjà cité·e·s, qui m'ont entouré avant, pendant et après la thèse. Je pense notamment à Dorian, Maxime, Hugo, Jérémy, Kériann, Cyril, Vianney, Mathieu, Fabien et Loup. Enfin, de tout mon cœur, je remercie Naya d'avoir été à mes côtés pendant tout ce temps.

Pour finir, je remercie ma famille – ma mère, mon père et mes deux frères Félix et Hippolyte – pour m'avoir supporté dans tous les sens du terme et ce malgré mon éloignement. Votre soutien lors de mes nombreux blocages et interrogations a été précieux.

Résumé en français

Bien que les logiciels continuent de se complexifier et que la cyber-sécurité soit en train de devenir un enjeu majeur d'une société où l'automatisation est maintenant la norme, la spécification et la vérification d'exigences de haut niveau (comme des propriétés de sécurité, telles que l'intégrité des données ou la confidentialité) sur des logiciels de taille importante reste un défi pour l'industrie.

Cependant, de telles exigences de haut niveau représentent la majeure partie des cahiers des charges écrits et compris par les ingénieurs, et il est nécessaire de faciliter l'utilisation des méthodes formelles par ces derniers sur ces exigences. Cette thèse présente un cadre formel pour les exigences de haut niveau appelé les meta-propriétés, décrites pour un langage de programmation abstrait, et centrées sur les propriétés liées aux manipulations de la mémoire et les invariants globaux. Une première partie de la thèse décrit ce formalisme et la sémantique des méta-propriétés.

Ce cadre formel est dans un second temps appliqué au langage C avec HILARE, une extension d'ACSL (un langage de spécification formel pour les programmes C), qui permet la spécification d'exigences haut niveau sur des programmes C de grande taille avec facilité. La lien est fait avec les méta-propriétés en exposant une syntaxe concrète pour leur spécification, en plus d'un grand nombre d'extensions pratiques. Des techniques de vérification pour HILARE, basées sur la génération d'assertions locales et la réutilisation des analyseurs de Frama-C existants (Wp, EVA, E-ACSL), sont présentées et implantées dans le greffon MetAcsl pour Frama-C. Le lien direct avec la sémantique des propriétés fait que ces approches sont correctes par constructions. Le processus est appuyé par de nombreux exemples sur un petit cas d'étude faisant intervenir des notions de confidentialité.

Une fois les techniques de base présentées, une méthodologie complète pour l'évaluation des propriétés de grands programmes est détaillée, articulant les méta-propriétés, les HILARES, les techniques de vérification et les particularités du C. Y sont exposés les procédures optimales pour utiliser les moyens présentés et les pièges à éviter. La méthodologie est illustrée par un large ensemble d'exemples et de propriétés de sécurité, et est en particulier appliquée à un cas d'étude imaginaire impliquant des propriétés complexes d'un micronoyau.

Par la suite, nous explorons une autre manière de vérifier qu'une exigence de haut niveau est vérifiée par un programme en la déduisant à partir d'autres exigences déjà prouvées. Un cadre pour la déduction des méta-propriétés est construit en identifiant un certain nombre de situations où une déduction est désirable, en mécanisant une partie de la formalisation des méta-propriétés via Why3/Coq et en prouvant que de telles déductions sont correctes au sein de ce système. Les déductions sont ensuite traduites sous forme de règles Prolog qui sont intégrées à MetAcsl et utilisées pour déduire automatiquement un sous-ensemble de méta-propriétés, sans avoir recours à des assertions locales.

Enfin, nous présentons une application de toutes les techniques présentées (HILAREs, vérification locale, vérification alternative) articulées selon la méthodologie exposée, à un cas d'étude industriel et réaliste: le chargeur d'amorçage de WooKey, un périphérique de stockage USB sécurisé développé par l'ANSSI.

Contents

List of Figures

Chapter 1

Introduction

In the last decades, computer systems have become deeply entrenched in our society. The recent years have seen a dramatic increase of not only user-facing systems (in particular through smartphones and personal computers) but also industrial systems. The latter exist at different scales: from large – operation of large structures such as trains and power plants – to small – embedded devices that are pervasive in our everyday life: household appliances, medical equipment, etc. Furthermore, more and more of these systems are now connected to the Internet, and their behaviour is becoming deeply intertwined with the good operation of the global network.

Naturally, humans designing and programming these systems are bound to make mistakes: users are accustomed to software bugs and the Internet has an abundance of people reporting bugs and offering solutions for circumventing them. While for most bugs the worst result is mild annoyance and perhaps an awful Friday evening, some of them can be truly catastrophic. Indeed, a safety bug in an industrial system can very well result in physical [Gen92], environmental [SJ97] or material harm [Lio96]. Similarly, today flaws in even apparently simple user-facing systems can have dire consequences in terms of privacy and security in general, as the increasing stream of large-scale hacks tends to indicate. This raises the need for efficient and accessible methodologies to detect flaws in computer programs, small and large.

A common but somehow limited strategy is *testing*: trying to run a program multiple times on different inputs that are representative of its normal usage (such as user actions), and checking that the resulting behaviour is as expected (either by comparing the behaviour with a pre-defined one or by ensuring it is compliant with some requirements).

However, in most cases, the input space cannot be exhaustively explored, which means that testing cannot ensure the absence of bugs in a program. This can be alleviated by checking for a set of test coverage criteria: metrics of a test suite determining for example how much of the whole code base is actually executed during the test suite. While this technique can be used to consider a wide variety of situations in the tests, it is usually very hard to achieve complete coverage on complex programs. Partial coverage is certainly better than nothing and may be largely sufficient for non-critical systems, where failure does not entail significant damages of any kind. However, there are other programs where the consequences of bugs are signifiant enough to justify the need for stronger guarantees.

1.1 Formal Methods

Formal methods, and more precisely formal specification and verification, are a set of techniques meant to assess the quality of programs and increase our trust towards them, by leveraging theoretical principles.

The goal of these techniques is to specify a desired property about programs (or models of programs) and deliver a judgement: does a particular program respect that property? This is made possible by describing the precise semantics of each syntactic element of a programming language and their composition, allowing to reason about a whole program as a mathematical object, about which various logical properties can be expressed.

Automation, soundness, completeness. To remain accessible by the industry, these techniques must be as automated as possible, to both avoid human error in the verification process itself and keep the costs at a manageable level. This goal is severely hindered by Rice's theorem [Ric53] which states that non-trivial properties of a program are undecidable (a generalization of Turing's famous halting problem), and more generally by the negative answer given to Hilbert's *Entscheidungsproblem* by Church [Chu36] and Turing [Tur37]. This means that a formal method approach cannot be at once *sound*, *complete* and *automated*: it cannot always deliver a *correct* judgement on *every kind of program* without the need for *human intervention*. To circumvent this impossibility, all techniques must make a compromise by giving up at least one of these three properties.

Note that soundness and completeness are defined with respect to a goal. For example, as a bug-detecting technique, testing is sound (a detected bug really is a bug) but incomplete. Throughout this thesis, the goal

will be to *verify* that a program respects a property without any doubt, in *all of its possible executions*. Any mention of soundness or completeness will be relative to this goal. In that respect, testing is complete (it confirms all properties respected by the program) but unsound (it does not eliminate the possibility of a bug in general).

Some general techniques for formal verification include:

- *Deductive verification* [Hoa69] expresses the correctness as a set of mathematical statements (called verification conditions), the truth of which imply the conformance of the program to its specification. The verification conditions are handled by automatic or interactive theorem provers. It is sound and complete, but needs human intervention for writing suitable proof-guiding annotations for the proof of some verification conditions.

- *Abstract interpretation* [CC77] maps memory locations to a more abstract domain that over-approximates the values they can take during the union of all possible executions. It is sound and automated but not complete, since it generates false alarms (cases where a problem is detected in a correct program) on some programs.

- *Model checking* [CES86] verifies a program (or an abstract model of it) by completely exploring the set of all possible states. It is not complete, since exploring all possible states is generally impossible. On the other hand, *bounded* model checking chooses to explore only a finite subset of these states (states reachable after a fixed number of steps), gaining completeness at the price of soundness.

- *Symbolic execution* [Kin76] executes a program but with variables mapped to symbols that represent arbitrary input, discovering the constraints that relate the variables. It is generally automated, giving up soundness and/or completeness depending on how the constraints are managed.

Remark 1 (Soundness and completeness). When a technique gives up soundness or completeness, in some cases it can be reduced to a sound and complete technique (hence not automated) by introducing appropriate abstractions, constraints, kinds of properties, etc.

Verification, static and dynamic. The above techniques are mostly *static* (analysing the program without concretely executing it). There are also *dynamic* techniques that alleviate the undecidability problem by giving up guaranteed termination (and are not complete for detecting correct properties in general, as they are more oriented towards bug detection). For example, *Runtime Assertion Checking* [CR06] transforms specification into executable checks that allow detecting flaws at runtime. Some other techniques, like *Concolic Testing* [SMA05], combine both static and dynamic verification.

While the contributions of this thesis are not particular to any of these techniques, we will mostly focus on deductive verification and runtime assertion checking when discussing verification.

Specification of properties. The mentioned techniques are meant to *assess* that a program is valid with respect to a property, but this does not say anything about how such a property should be specified. In this thesis, we will focus our specification efforts on two major specification mechanisms:

- *ad-hoc assertions* inserted directly within a program, expressed using a logical language that embeds semantic elements of the programming language. They allow reasoning about the current state of the program at the point where they are inserted;

- *function contracts*, introduced notably by the Eiffel programming language and its "design by contract" paradigm [Mey92]. Each function of the program can be annotated with a contract, stating preconditions, postconditions and possibly other properties of the function, using the same formalism as assertions.

Function contracts. Let us see an example of function contract. Algorithm 1 below is a function computing the greatest common divisor (GCD) of two integers, annotated with a contract: a *precondition*, stating that the two parameters must not be zero; and a *postcondition*, ensuring that the result of the function is indeed a common divisor. Notice that this contract is partial since it does not ensure that the result is the *greatest* common divisor: a function always returning 1 is compliant with the postcondition bit is not a proper GCD function.

Algorithm 1 A GCD function annotated with a contract

Requires: $a \neq 0$ and $b \neq 0$ ▷ Precondition
Ensures: $a \bmod \texttt{result} = 0$ and $b \bmod \texttt{result} = 0$ ▷ Postcondition

```
function GCD(a, b)
    r ← a mod b
    while r ≠ 0 do
        a ← b
        b ← r
        r ← a mod b
    end while
    return b
end function
```

Modular verification. Function contracts are a popular and efficient way of specifying large programs. Indeed, they enable an important factor of efficiency in formal verification: modular verification.

In that setting, if we want to prove the contract of a function f calling another function g, we can just read the contract of g and assume its postcondition is correct after the call. There is no need to inspect the body of g, provided we prove that the preconditions of g hold at the call site.

Assuming every function is annotated with a contract and individually proved similarly, we can prove that the whole program is correct in a modular manner.

However, while function contracts are very relevant for specifying properties that are *local* to a function, it is inherently hard to specify *global* requirements with this tool.

1.2 Motivation

In the natural language specification of a system, one will often find high-level requirements: requirements that are not specific to small code units (which are abstracted away) but rather pertain to large components of the system or the system itself. The goal of this thesis is to offer a way to specify such high-level properties of programs, and develop means to assess programs with respect to these properties. We focus on the following points:

High-level requirements for security. While there are many safety requirements that are high-level, this is especially apparent with security

requirements, where large sets of functions in the software are required to uphold constraints. For example, we might want to isolate components of a software from one another by forbidding a component to read or write the resources owned by another, in order to reduce the attack surface. Or we might want to ensure that a software is compliant with stringent confidentiality and privacy requirements, such as the ones mandated by the General Data Protection Regulation (GDPR) [GDPR16]: for example ensuring that user data is only used by components of a software which have a purpose matching the consent given by the user. Hence in this thesis we will want to ensure that a rich variety of high-level security requirements can be expressed. In particular, we will focus on expressing access control and information flow requirements.

Accessibility and automation. As discussed previously, automation and ease of use are important stakes since they greatly influence the option of a technique by industrial users, which is an overall goal of this work. Hence, we will want our means of specification to be completely automated: in particular, the size of the code base under analysis should not change the amount of effort needed to specify high-level requirements. Similarly, we will want the verification of requirements to be as automated as possible, considering the limitations described in the previous Section. Laslty, we would like our techniques to be applicable to programs written in a mainstream programming language: the C programming language.

Scalability. Expressing high-level requirement at scale with contracts has several problems, which our contributions will try to address:

> **Expressiveness.** Some simple global requirements can be expressed as a single contract, distributed over the set of relevant functions. In the general case, however, this is hard or even impossible. Even when possible, it is necessary to find a complex encoding of the requirement fitting the contract paradigm. This makes the specification much harder to read and the subsequent proof effort difficult and mostly manual.
>
> **Maintenance.** If the global requirement can be expressed as a contract, it can then be attached to every function of the relevant component. While this is possible, this is a tedious and error-prone manual effort, especially on large code bases. Furthermore, if the requirement happens to change, every subsequent contract must be changed.

Traceability. If two or more global requirements pertain to overlapping components of a program, then the contracts from both of these requirements are mixed in the overlapping functions. Without a way to explicitly link the requirements to the low-level contract clauses, it is hard to determine what requirements a single function is subject to, to check that no function has been missed or to track what global requirements are proved or not at a given time. In other words, contracts do not allow to take a step back from low-level details.

Previous works have addressed the need for a way to specify and verify specific kinds of requirements. Most notably, the work of Pavlova et al. [Pav+04] proposes a method for specifying high-level *security policies* on Java programs running on smart cards, and verifying them by automatically generating low-level annotation that are woven into the source code and then verified using an external tool. However, while the security policies they consider are indeed high-level in the sense that they have impact on the whole application, the properties that the authors consider are firmly linked with the Java Card architecture: applet life cycle, atomicity of transactions, exceptions, etc. As the authors note, "there exists many relevant security properties such as specifying memory management, information flow and management of sensitive data. Identifying all relevant security properties and expressing them formally is an important ongoing research issue". In that sense, their approach has not the general purpose that we are looking for. And other previous approaches, while efficient for their purpose, present the same problem (see Chapter 8 at the end of the thesis for a review of such works and our position compared to them).

A secure vault. We will now introduce a small use case that will serve as an illustrative example to present the concepts introduced throughout this thesis. Let us consider an application managing a physical vault, illustrated in Algorithm 2, with components managing the locking and unlocking of the vault, handling user interaction, authentication, communication with other systems and alarms, etc. Let us now suppose that to interface with the physical system moving the locks, the application must simply write a boolean into a global variable whose address is mapped to that system. A very simple high-level security requirement that we would like to express, is that *only the component responsible for locking and unlocking should ever write to that variable.*

Algorithm 2 The vault manager, a large multi-component application

```
vaultOpen ← false                          ▷ Global variable controlling the vault

component LOCK MANAGEMENT                  ▷ Functions interacting with the vault
  function LOCK
  function UNLOCK
  ...
component CRYPTOGRAPHY                     ▷ Other unrelated components
  function GCD
  function AUTHENTICATE
  ...
component KEYPAD
component NETWORK
...
```

1.3 Contributions and Document Structure

To address these issues, we outline a general-purpose framework for working with high-level requirements that can be applied to any programming language. We instantiate this formal framework to the C programming language, a language created in the early 1970s which remains nowadays among the most widely used programming languages in the world, especially for embedded, safety- and security-critical programs. We explore a wide variety of high-level safety and security requirements over several use cases, and discuss different means of verifying them on C programs.

Within this framework, the vault requirement described above will simply be expressed by the following statement, and easily provable on large code bases:

```
meta \prop,
    \name(vault_security),
    \targets(\diff(\ALL, LOCK_COMPONENT)),
    \context(\writing),
\separated(&vaultOpen, \written);
```

This statement, that will be called a HILARE later in this thesis, is meant to be inserted at the end of a C program. It uses a special syntax for describing our requirement: a simple name, a set of functions where it should apply (here, all functions except the ones in the locking component), and a description of the requirement: "when these functions write

to memory, the written variable should not overlap with the vault state variable".

While this statement is seemingly simple, making precise sense of it and its underlying specification and verification methodology is the whole purpose of this thesis.

Contributions. The main contributions of this thesis are:

- the description of a new class of high-level properties to express global requirements over large programs. In this document, we define a simple imperative abstract language, upon which we formalize the notion of *meta-properties*. We show that this formalism is useful to express concrete requirements that were hard to specify with previous approaches;

- a concrete syntax for expressing meta-properties on programs written in the C programming language. The formalism is transposed from the abstract language to C, yielding an extension of the ANSI/ISO C Specification Language called the HILARE language. Meta-properties are adapted to fit quirks of the language, and several useful extensions of the basic concept of meta-properties are presented;

- a verification strategy leveraging the existing Frama-C environment for assessing HILARE properties on C programs. This approach is based on a translation from the HILARE language to sets of local assertions, which can then be handled by existing verification tools. It handles simple meta-properties as well as their extensions. The HILARE syntax as well as its verification strategy is implemented within a Frama-C plugin called MetAcsl. This plugin has been used to validate the design of meta-properties on various examples;

- a detailed methodology for tackling specification and verification tasks on large programs involving high-level requirements. This methodology highlights a set of specification patterns that can be used as building blocks for designing complex properties with the HILARE language. It is illustrated as applied to a set of complex requirements in an artificial micro-kernel;

- a study of the application of our methodology to WooKey, a real large-scale codebase. Several security requirements of the bootloader of WooKey, a secure USB key prototype developed from the ground

up by the French National Agency for the Security of Information Systems (ANSSI), are specified and verified on a program spanning hundreds of functions;

- an extensible framework for reasoning about meta-properties. We establish foundations for a sound system that allows deducing meta-properties from others at the global level, without having to resort to local assertions. This framework is based on a soundness proof in Why3 and an efficient solver in Prolog, which is seamlessly integrated in the MetAcsl plugin.

Structure of the document. This thesis is structured as follows:

- Chapter 2 formalizes the notion of meta-property from the ground up on a simplified language and illustrates it on several examples, highlighting its usefulness in the context of our illustrative example;

- Chapter 3 takes that formalization and transposes it to the C programming language and the Frama-C platform, which is briefly presented along with its specification language ACSL. The resulting specification language is called HILARE. Several extensions to the formalization are presented, and a small confidentiality-oriented case study is used to illustrate its usage on concrete programs;

- Chapter 4 describes a verification strategy for HILARE specification, based on translation into local annotations. The case study from Chapter 3 is continued and used to illustrate the soundness of the approach. The MetAcsl plugin for Frama-C is described in this chapter;

- Chapter 5 crystallizes the elements of the previous chapters into a detailed methodology for tackling complex specification and verification tasks on C programs. It discusses common pitfalls, and describes the entire process from approach to complete verification, illustrated by requirements on a micro-kernel;

- Chapter 6 discusses an alternative verification strategy for HILARE properties avoiding local reasoning: deduction of high-level requirements from others. It describes foundations for a complete and extensible framework for reasoning about HILARE automatically, and its seamless integration within MetAcsl;

- Chapter 7 presents a real, large-scale case study of the application of our methodology to the specification and verification of security requirements to the bootloader of WooKey, a secure USB key prototype developed by the ANSSI;
- Chapter 8 presents previous efforts related to our work, and how this thesis is positioned compared to them;
- Chapter 9 concludes this thesis, summarizing the results achieved in the thesis, their limitations and ideas of future works.

Remark 2 (Environments and text emphasis). Throughout the thesis, coloured environment such as this one are used to delimit elements such as remarks, definitions or examples.

In definitions, *italic* is used to signal the new terms introduced by the definition. In the rest of the document, *italic* is meant to draw the attention of the reader to a particular word, signalling its significance to properly understand a concept. To avoid ambiguity, **bold** is used to signal such words in definitions.

Publication of the results. Some contributions of this thesis are already published:

- [Rob+19a] introduces the concept of meta-property (Chapter 2) through its application within the Frama-C framework. It describes en early version of the HILARE language and MetAcsl (Chapters 3 and 4).

  The key contributions of this paper are summarized in French in [Rob+19b];
- [Rob+19c] lays out a partial formalization for meta-properties, upon which the formalization of Chapter 2 is built. It also presents some extensions of the HILARE language mentioned in Section 3.3 as well as the complete benchmark reproduced in Section 4.5;
- [Rob+21a] presents a complete methodology for specification and verification with HILAREs, supported by an artificial case study on micro-kernels as well as the WooKey case study. Most of this work is reproduced in Chapter 5, with the WooKey part developed in its own chapter (Chapter 7).

The key contributions of this paper are summarized in French in [Rob+21b].

Furthermore, the work presented in this thesis has been used by Thales in a large-scale industrial verification project [DHK21], detailed in Section 8.3.

Chapter 2

A Formalization of High-Level Requirements

As the goal of this thesis is to offer a way to specify high-level properties of programs, formally defining the class of these properties is a useful first step. Having a formal, abstract frame for these properties allows us to sketch an approach of high-level specification and verification on real C programs in later chapters.

As a working example during this chapter, we use the simple abstract program (mentioned in Chapter 1) managing an imaginary vault, with some defined functions e.g. for locking or unlocking the vault, as well as a set of other unspecified functions. A basic requirement on such a system is: "the vault can only be locked or unlocked by the appropriate locking/unlocking functions". In particular, if these functions perform some kind of authentication, we do not want another function to spuriously bypass this and unlock the vault anyway. This example also serves as a minimal goal of what should be expressible.

At the end of the chapter, we will have a formal framework for expressing at least such a requirement with precise semantics, and more generally memory-related high-level properties. After the presentation of a set of notations used throughout this chapter in Section 2.1, Section 2.2 formalizes an abstract programming language. Section 2.3 builds upon that, describing a notion of context to select points and local information in such programs. A few of these contexts that will be useful throughout the thesis are listed in Section 2.4. Section 2.5 then gives several examples of concrete high-level properties we want to be able to express. Finally, Sections 2.6 and 2.7 describe the appropriate formal framework for achieving the desired specification goal.

2.1 Notation

Throughout this chapter, we will use the following notations as a support for our formal framework.

2.1.1 Sets

We use the common symbols $\varnothing$, $\cup$, $\cap$, $\setminus$ to designate respectively the empty set, set union, intersection and difference. Furthermore, we use the symbol $\uplus$ to denote disjoint set union, that is union of sets which must be disjoint.

For any set S, we denote by $\mathcal{P}(S)$ the *power set* of S i.e. the set of all subsets of S (including $\varnothing$).

We denote by $\mathbb{N}$ the set of natural integers with zero, $\mathbb{N}^+$ the same set without zero, and $\mathbb{Z}$ the set of all integers:

$$\mathbb{N} = \{0, 1, 2, \ldots\};$$
$$\mathbb{N}^+ = \mathbb{N} \setminus 0;$$
$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

2.1.2 Functions

We denote by

$$f : A \to B$$

a total function f associating to every element of A an element of B.

For a function $f : A \to D$ where $D = \{g \mid g : B \to C\}$ (that is, for a function that returns functions), we also write

$$f : A \to (B \to C)$$

2.1.3 Mappings

We denote by

$$f : A \rightharpoonup B$$

a partial function or *mapping* f, associating to some elements of A an element of B.

We denote by $\mathrm{dom}(f)$ the domain of f i.e. the set on which f is actually defined.

When the domain set A is empty, there is only one such mapping (having no bindings), that we will denote $\varnothing$.

Let $f, g : A \rightharpoonup B$ be partial mappings.

Furthermore, we denote by $f[x \mapsto y]$ the new mapping defined as f plus a new association (or *binding*) from $x \in A$ to $y \in B$. If a mapping for x existed in f, then it is shadowed:

$$\forall v \in \mathrm{dom}(f) \cup \{x\}, f[x \mapsto y](v) = \begin{cases} y \text{ if } v = x; \\ f(v) \text{ otherwise.} \end{cases}$$

Additionally, we introduce the notation $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ to denote the definition of a mapping *in extenso*:

$$[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n] := \varnothing[x_1 \mapsto y_1][...][x_n \mapsto y_n]$$

Given two functions f and g with disjoint domains, we will use the notation $f \uplus g$ to designate a new mapping combining the bindings of both f and g such that $\mathrm{dom}(f \uplus g) = \mathrm{dom}(f) \uplus \mathrm{dom}(g)$ and $\forall x \in \mathrm{dom}(f), (f \uplus g)(x) = f(x)$ (and symmetrically for g).

Lastly, we denote by $f|_S$ the new mapping defined as f whose domain has been restricted to S:

$$\mathrm{dom}(f|_S) = \mathrm{dom}(f) \cap S;$$
$$\forall x \in \mathrm{dom}(f|_S), \quad f|_S(x) = f(x).$$

2.1.4 Inference Rules

We use *derivation rules* to describe what can be deduced in a formal system, typeset as follows:

$$\textsc{Name}\ \frac{\mathrm{Premise}_1 \qquad \mathrm{Premise}_2 \qquad \ldots}{\mathrm{Conclusion}}\ \text{(external conditions)}$$

where Name is the name of the rule, $\mathrm{Premise}_{1,2,...}$ and $\mathrm{Conclusion}$ are formulas of the formal system and the optional external conditions are logical conditions on $\mathrm{Premise}_{1,2,...}$ expressed outside of the system, within the classical logic used throughout this thesis.

Such a rule expresses that the conclusion can be derived from the premises given that the external conditions hold. A rule can have no premises, in which case the conclusion can be derived immediately.

2.1.5 Graphs

A *directed graph* is a pair (V, E) where V is a finite set and $E \subseteq V \times V$. V is called the set of *vertices* (or nodes) of the graph, and E its set of *edges*. We say that there is an edge from v_1 to v_2 if $(v_1, v_2) \in E$.

2.2 An Abstract Language

In this chapter, we are working in the context of an abstract language, which we call A-Lang. Very few hypotheses are initially made on that language: it must have a notion of (i) functions (separation of code units), (ii) global locations, (iii) local variables and (iv) instructions (either a local location declaration, a function call or an abstract operation). Note that in this chapter we will use the words *location* and *variable* interchangeably.

This language serves as a support for the definition of an interesting class of properties which we will call *meta-properties*. Keeping an abstract support language allows us to later refine it to a real programming language such as C in the rest of this work, while demonstrating the generality of our approach. We define A-Lang from the bottom up, going from identifiers and instructions to whole programs.

Definition 1 (Identifiers). We define the following, mutually disjoint, finite sets of possible *identifiers*:

- $\mathcal{I}_F$: for function names;
- $\mathcal{I}_M$: for meta-variable names;
- $\mathcal{I}_G$: for global location names;
- $\mathcal{I}_L$: for local location names;
- $\mathcal{I}_V$: for logic variables.

We also define the following sets respectively called the *location identifier set* and the *extended location identifier set*:

$$\mathcal{L} := \mathcal{I}_L \uplus \mathcal{I}_G;$$
$$\mathcal{L}_F := \mathcal{L} \uplus \mathcal{I}_F.$$

The sets being disjoint means that an identifier cannot be used for more than one kind of objects.

We now define a set of A-Lang instruction kinds. As the language itself, these instruction kinds are abstract and do not define a concrete program by themselves, just families of similar instructions. They allow sorting concrete instructions (as we will see in Remark 3) into different categories.

Definition 2 (A-Lang: Instructions). An *instruction* of the A-Lang language is one of the following kinds:

- Begin;
- End;
- Branch;
- Var l where $l \in \mathcal{I}_L$;
- Compute;
- Call f where $f \in \mathcal{I}_F$.

The set of all instructions is denoted by $\mathbb{I}$.

Before talking about execution semantics of these instruction kinds and concrete instructions (which will be defined in Definition 7 for details), we can already give an intuition about their meaning:

- Begin and End are dummy instructions that just delimit a function: its entry and exit points. There is one of each in a given function. We will call every other instruction *meaningful*.
- Branch is a conditional jump to another instruction. The condition is left abstract.
- Var l is a *local* variable declaration. From this point, the variable l is declared until the end of the function, with a default value. If the variable was already declared, it remains declared but is reset to the default value (which will be described in Definition 5).
- Compute is an abstract, deterministic computation. Unlike the previous instruction kinds, it can have side effects on the existing global state.
- Call f is a function call to f. Arguments are assumed to be conveyed through global variables rather than explicitly.

Rather than working directly on A-Lang functions by mean of a grammar, it will be convenient for this work to refer directly to their control-flow graphs [All70] (CFG), which describe the structure of the control-flow of a function.

Definition 3 (A-Lang: Function, Control-flow graph). An A-Lang *function* f is defined by its *control-flow graph* (V_f, E_f), that is, a directed graph with the following properties:

- Each vertex is equipped with two labels accessed through labelling functions id and instr

$$\mathrm{id} : V_f \to \mathbb{N}$$
$$\mathrm{instr} : V_f \to \mathbb{I}$$

  where for any vertex $v \in V_f$, $\mathrm{id}(v)$ is an integer identifier and $\mathrm{instr}(v)$ is an A-Lang instruction kind (see Definition 2).

- There is exactly one vertex of the Begin instruction kind, and one of End:

$$\exists! v_1 \in V_f, \quad \mathrm{instr}(v_1) = \textsc{Begin};$$
$$\exists! v_2 \in V_f, \quad \mathrm{instr}(v_2) = \textsc{End}.$$

  Since these two vertices are unique within a function, we name them $\mathrm{begin}(V_f)$ and $\mathrm{end}(V_f)$.

- $\mathrm{begin}(V_f)$ has no predecessor, while every other node has at least one predecessor.

- $\mathrm{end}(V_f)$ has no successor, while every other node has at least one successor.

- Conditional jumps have two successors in the graph while other nodes have at most one.

- Every node is structurally reachable from $\mathrm{begin}(V_f)$.

  The set of all correct control-flow graphs is denoted by $\mathcal{F}$.

The structural properties of the CFG imply the following facts for a given function:

- There is a single edge from $\mathrm{begin}(V_f)$ to the first meaningful instruction of the function.

- There is no structurally unreachable code.

Intuitively, an edge $(v_1, v_2) \in E_f$ exists if and only if the control flow can structurally go from v_1 to v_2 during the execution of the function (Definition 8 will give more details on execution). In that case, we say that v_1 and v_2 are *successive*.

> **Remark 3 (From instructions to concrete execution).** Instructions as defined in Definition 2 are simply kinds. Two nodes in a control-flow graph can have the same instruction kind. However, this does not mean that the concrete execution of these nodes has to be the same.
>
> As we will see in Definition 7, the execution behaviour of a node is determined not only by its instruction kind but also by its unique (see Definition 4) identifier.

Also notice that edges are only defined *within* the function: function calls are not represented by edges.

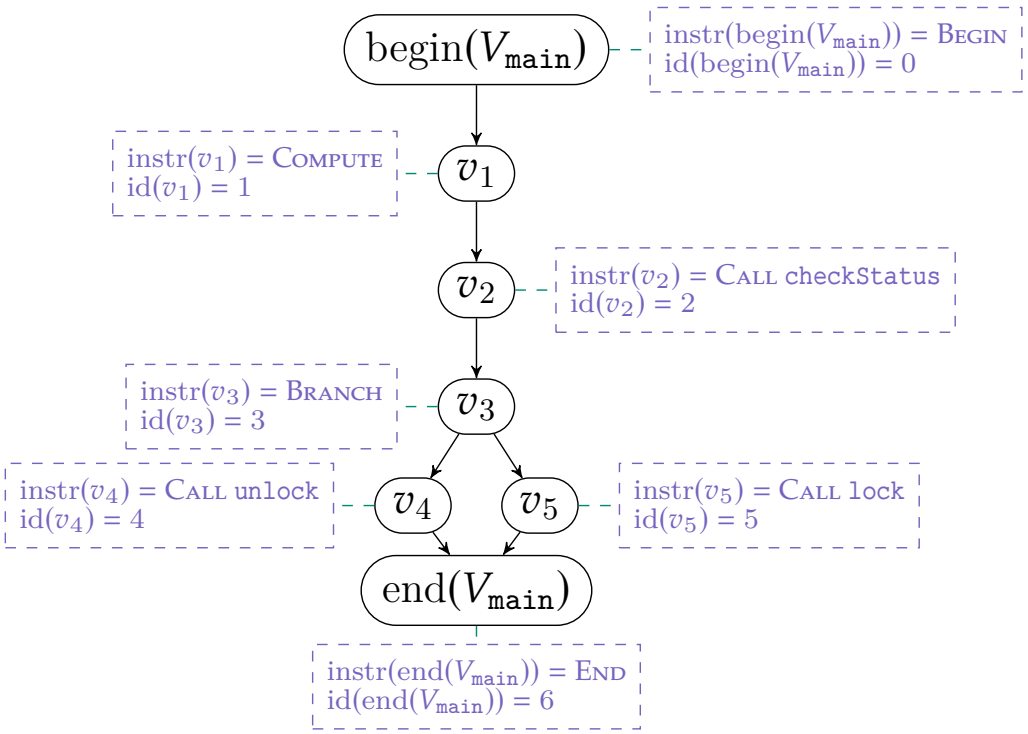

Figure 2.1: The control-flow graph of a `main` function

Example 1. An example of a function defined by its control-flow graph is illustrated in Figure 2.1. Each node is labelled with its unique identifier id and its instruction instr. It represents a simple program performing a computation, calling another function `checkStatus` and then branching to one of two other calls. Again, notice that this program is left very abstract: nothing is said about the performed computations – only the kinds of instructions and the control flow are represented.

In further CFG representations (e.g. Figure 2.2), we will simplify the graphs by omitting the identifiers, the node names, and putting directly the instruction kind inside the node. While the pruned information is still stored in the graph, it will not be useful to represent it.

We can now define an A-Lang program: a set of functions and a set of predefined global variables (which is by Definition 1 distinct from variables that may be declared locally).

Definition 4 (A-Lang: Program). An A-Lang *program* is a triple

$$\Big(F, G, (V_f, E_f)_{f \in F}\Big)$$

where:

- $F \subseteq \mathcal{I}_F$ is a set of A-Lang function names;
- $G \subseteq \mathcal{I}_G$ is a set of global location identifiers;
- $(V_f, E_f)_{f \in F}$ is an F-indexed family of functions (i.e. a set of CFGs, according to Definition 3).

A program $\Big(F, G, (V_f, E_f)_{f \in F}\Big)$ is said to be well-formed if:

- all called functions are defined:

$$\forall f \in F, v \in V_f, f_c \in \mathcal{I}_F, \quad \mathrm{instr}(v) = \textsc{Call}\ f_c \implies f_c \in F;$$

- functions do not share nodes:

$$\forall f_1, f_2 \in F, \quad f_1 \neq f_2 \implies V_{f_1} \cap V_{f_2} = \varnothing;$$

- vertex identifiers are unique across all functions of the program:

$$\forall f_1, f_2 \in F, v_1 \in V_{f_1}, v_2 \in V_{f_2}, \quad v_1 \neq v_2 \implies \mathrm{id}(v_1) \neq \mathrm{id}(v_2).$$

It is always assumed that programs are well-formed. The set of all well-formed programs is denoted by O.

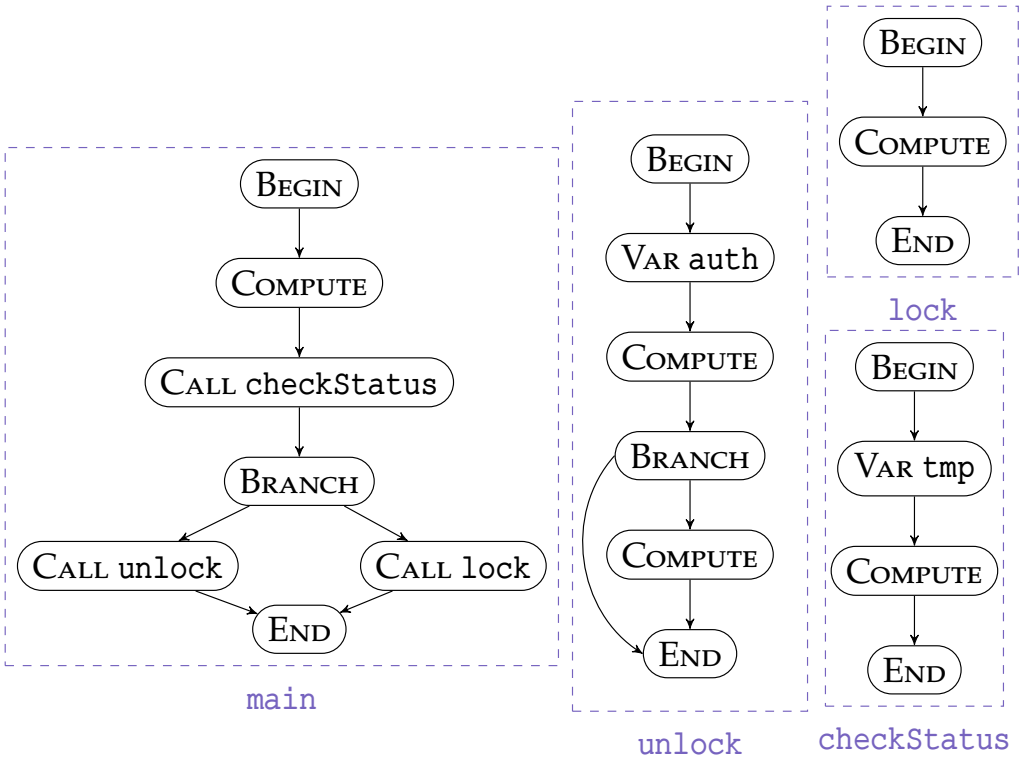

Figure 2.2: A few A-LANG functions

Remark 4 (Notation). To keep the notation light, from now on, we will refer to a fixed well-formed program $O = \left(F, G, (V_f, E_f)_{f \in F}\right)$. Any reference to F, G, V_f or E_f will implicitly refer to the corresponding component of O unless otherwise stated.

To make things more concrete, let us now introduce a small example of an A-LANG program.

Example 2 (A Secure Vault). We can define an A-LANG program

$$(\{\texttt{main}, \texttt{lock}, \texttt{unlock}, \texttt{checkStatus}\}, \{\texttt{vaultOpen}, \texttt{init}\}, \mathcal{E})$$

where $\mathcal{E}$ is the family of graphs represented in Figure 2.2 (with a simplified representation, see Example 1).

This program contains four functions as well as two global locations `vaultOpen` and `init` (which we will assume to contain integer values in the next examples). This program is a well-formed A-LANG program.

Having defined the structure that an A-LANG program must have, we now attach semantics to such a program. In particular, we start with a definition of state for storing values of declared variables.

A-LANG programs can declare both global (via the G set) and local locations (via the VAR instruction). Once declared, each of these locations has a *value* that can change over time.

Definition 5 (A-LANG: Values, State). We denote by $\mathcal{V}$ the set of values that can be stored into locations. Furthermore, we denote by $v_0 \in \mathcal{V}$ the *default value*. We assume no particular property of v_0.

A *state* is a partial function

$$\mathcal{L} \rightharpoonup \mathcal{V}$$

that assigns *values* to location identifiers (either local or global). States will be denoted by $\sigma, \sigma_1, \sigma_2, \ldots$ and the set of all states is denoted by Σ.

A program state always assigns a value to global locations:

$$\forall \sigma \in \Sigma, G \subseteq \mathrm{dom}(\sigma).$$

Notice that for the sake of clarity, all variables have the same type in this formalization. However, it could handle multiple types by adding a parameter to the state function or by having separate stores for each type of location.

Example 3 (Integer locations). In all examples of this section, we always assume that we are working with *integer* locations, with 0

being the default value. Hence, we use the set of values:

$$\mathcal{V} := \mathbb{Z};$$
$$v_0 := 0.$$

A set of operators for these values will be defined in Example 12 (page 42).

We now define a semantics to describe what happens to the state upon execution: the concrete execution is entirely characterized by any relation between *configurations* observing a set of properties, as described below.

Definition 6 (A-Lang: Configuration). Let $f \in F$ be a function, $v \in V_f$ one of its instructions and $\sigma \in \Sigma$ a state of the program.

We call the pair $\langle v, \sigma \rangle$ a *configuration* of f.

Intuitively speaking, a configuration $\langle v, \sigma \rangle$ represents a snapshot of execution in which σ represents the current values of locations and v represents the next instruction to be executed (i.e. σ is the state preceding the execution of v).

Note that the following definitions for instruction and function executions (Definitions 7 and 8) are mutually recursive: the execution of a single instruction ($\Downarrow$) and of a whole function ($\Downarrow^O$) are intertwined in the case of function calls.

Definition 7 (A-Lang: Instruction execution). Let $f \in F$ be a function, $v, v' \in E_f$ be two successive instructions of f and $\sigma, \sigma' \in \Sigma$ two program states.

We define an *execution relation* as any relation $\Downarrow$ between two configurations of a function such that

$$\langle v, \sigma \rangle \Downarrow \langle v', \sigma' \rangle$$

respects the following constraints:

- Begin and Branch do not modify the state. [a] If $\mathrm{instr}(v) =$ Begin or $\mathrm{instr}(v) =$ Branch, we have:

$$\sigma' = \sigma;$$

- VAR l defines a binding from l to the default value $v_0 \in \mathcal{V}$ in the state. If $\mathrm{instr}(v) = \text{VAR } l$ for some location l, we have:

$$\sigma' = \sigma[l \mapsto v_0];$$

- CALL g executes function g, correctly managing the interaction between the state of the caller and the callee (see Definition 8 below). If $\mathrm{instr}(v) = \text{CALL } g$ for some function g and if g terminates when executed on σ, then we have:

$$\exists \sigma_g \in \Sigma, \quad (g, \sigma|_G) \Big\Downarrow^O \sigma_g \quad \wedge \quad \sigma' = (\sigma_g|_G \uplus \sigma|_{I_L});$$

- COMPUTE arbitrarily modifies the state without adding or removing bindings. If $\mathrm{instr}(v) = \text{COMPUTE}$, we have:

$$\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma').$$

Furthermore, for every instruction except CALL and END, the relation must guarantee the **existence** and the **unicity** of a next configuration from a given one:

$$\forall v \in V_f, \sigma \in \Sigma,$$
$$\mathrm{instr}(v) \neq \text{CALL} \implies \exists!(v', \sigma') \in (V_f \times \Sigma), \quad \langle v, \sigma \rangle \Downarrow \langle v', \sigma' \rangle.$$

For CALL, there is **at most** one next configuration [b] from a given one.

[a] END is not allowed to modify state either, since there is no execution step from it because it does not have a successor. Recall that a return from a function call is not modeled by an edge but is part of the execution of the call.

[b] Because a function may not terminate, as we will see in Remark 5.

Each instruction kind is given a semantics that matches the natural intuition associated to its name. In particular, the CALL g instruction calls g on the current state of the caller (that is, restricted to global variables), then removes the callee's local variables from the resulting state after the call, and finally adds back the local variables of the caller.

We can then define the execution of a whole function: simply a chain of executions from a starting state to a *potential* final state (the function may not terminate).

Definition 8 (Chain of executions, Function execution). For any execution relation $\Downarrow$ and two arbitrary configurations $\langle v, \sigma \rangle$ and $\langle v', \sigma' \rangle$, we say there is a *chain of executions* from configuration $\langle v, \sigma \rangle$ to configuration $\langle v', \sigma' \rangle$ and note

$$\langle v, \sigma \rangle \Downarrow^{+} \langle v', \sigma' \rangle$$

if either:

- $\langle v, \sigma \rangle \Downarrow \langle v', \sigma' \rangle$;
- or there is a **finite** number $n \in \mathbb{N}$ of intermediate configurations $\langle v_1, \sigma_1 \rangle, \ldots, \langle v_n, \sigma_n \rangle$ such that

$$\big(\langle v, \sigma \rangle \Downarrow \langle v_1, \sigma_1 \rangle\big) \wedge \big(\langle v_1, \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle\big) \wedge \ldots \\ \ldots \wedge \big(\langle v_n, \sigma_n \rangle \Downarrow \langle v', \sigma' \rangle\big).$$

In other words, we denote by $\Downarrow^{+}$ the transitive closure of $\Downarrow$.

Let $f \in F$ be a function and $\sigma_1, \sigma_2 \in \Sigma$ a pair of program states. We say that f yields σ_2 when executed on σ_1, and write

$$(f, \sigma_1) \Downarrow^{O} \sigma_2$$

if there is a chain of instruction executions from its beginning vertex $\mathrm{begin}(V_f)$ to its end vertex $\mathrm{end}(V_f)$ that yields σ_2 when executed on σ_1:

$$\langle \mathrm{begin}(V_f), \sigma_1 \rangle \Downarrow^{+} \langle \mathrm{end}(V_f), \sigma_2 \rangle.$$

Remark 5 (Non-termination of a function). In the event a function does not terminate on a given state (if for example there is an infinite loop), then the (partial) execution relation does not have an image for that state since in Definition 8 a chain of executions must be finite.

This means that while non-terminating programs are valid in A-Lang, they cannot be executed.

With that, we have completely defined our support language A-Lang. It stays abstract to simplify the following definitions, but the bridge from abstraction to execution is represented by the execution relation, which

can be used to map an abstract A-Lang CFG to a concrete language and program.

2.3 The Concept of Context

Now that the base language is defined, we begin formalizing what a high-level requirement of that program is. To that end, we develop the concept of *context*. This will serve to describe what kind of requirement we are dealing with by characterizing the situations in functions in which it applies.

First, we will need a notation to refer to the set of local variables that have been declared in all paths leading to a given program point so that they can be used at that point.

Definition 9 (Local variables). Let $f \in F$ be a function, $e \in E_f$ one of its edges and $v \in V_f$ one of its nodes.

We denote by $\mathrm{loc}_f(e)$ (resp. $\mathrm{loc}_f(v)$) the set of local variables such that a node declaring them exists in every path leading to e (resp. v) in the control-flow graph of f.

In the extended semantics of A-Lang given in Section 2.4, the CFG node will only be able to use a variable if it was previously declared in all paths leading to that node.

Next, we define the notion of a *selector family*: a family of selected sets of function edges in the control-flow graph.

Definition 10 (Selector family). A *selector family* (or selector for short) is a family of sets $(S_f)_{f \in F}$ such that

$$\forall f \in F, \quad S_f \subseteq E_f.$$

A selector *selects* a set of edges from the functions of the program. As an example, we can take a function of the program previously illustrated in Example 2 and select a subset of its edges.

Example 4. The following family is a selector that selects edges leading to instructions that call other functions:

$$S^{\text{calling}} := \left(S_f^{\text{calling}}\right)_{f \in F}$$

$$\text{where } S_f^{\text{calling}} = \left\{(v_1, v_2) \in E_f \mid \exists g \in \mathcal{I}_F, \text{instr}(v_2) = \textsc{Call}\ g\right\}.$$

For the program presented in Example 2, $S^{\text{calling}}_{\texttt{main}}$ is shown in Figure 2.3 where all selected edges are dashed red (the annotations on the edges can be ignored for now): only edges leading to Call nodes are selected.

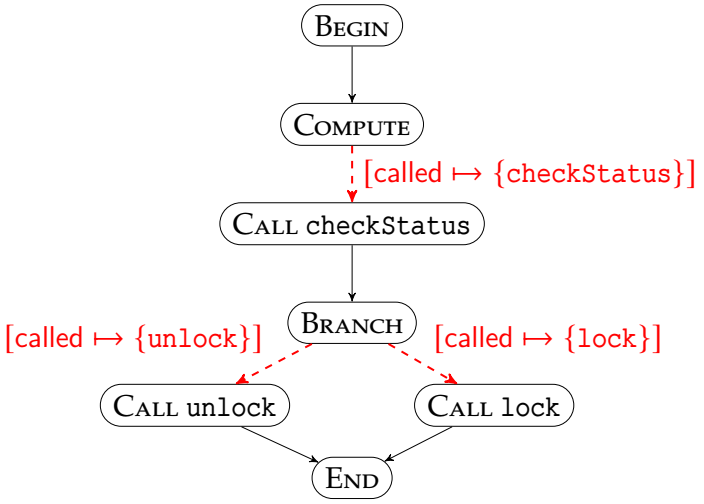

Figure 2.3: Illustration of the calling context on function `main` from Figure 2.1

With selector families being a mean to select edges (i.e. "program points"), we will next define a way to collect local information at this edge: contextualization functions.

Intuitively, given a set of meta-variables (which are arbitrary identifiers, see Definition 1), a contextualization function takes a program function, an edge of that function selected by a selector and aggregates some local information about this edge via a mapping from the meta-variables to that information.

This will be useful for gathering information about operations we are concerned about: if a requirement is about instructions that call functions, what are the called functions? If it is about instructions modifying variables, what are the modified variables? This way, we will be able to write a requirement that says "instructions modifying variable x only do it under

certain conditions“ and give it precise semantics with respect to the CFG of a program.

Definition 11 (Contextualization function, Context). Let $(S_f)_{f \in F}$ be a selector and $\mathcal{M} \subseteq \mathcal{I}_M$ a set of meta-variables (see Definition 1).

A *contextualization function* for (S_f) and $\mathcal{M}$ is a total function θ with the following signature:

$$\theta : \bigcup_{f \in F} S_f \to (\mathcal{M} \to \mathcal{P}(\mathcal{L}_F))$$

with the constraint that the function returns an environment mapping elements of $\mathcal{M}$ to sets of identifiers of global variables, functions and *declared* local variables:

$$\forall f \in F, e \in S_f, M \in \mathcal{M}, \quad \theta(e)(M) \subseteq \mathcal{I}_G \cup \mathcal{I}_F \cup \mathrm{loc}_f(e)\,.$$

Finally, we call *context* any triple $\left((S_f)_{f \in F}, \mathcal{M}, \theta\right)$ such that:

- $(S_f)_{f \in F}$ is a selector (Definition 10);
- $\mathcal{M} \subseteq \mathcal{I}_M$ is a set of meta-variables;
- θ is a contextualization function for $(S_f)_{f \in F}$ and $\mathcal{M}$.

A contextualization function parameterized by (S_f) and $\mathcal{M}$ associates to each edge of f selected by (S_f) an environment (that is, another mapping) that associates sets of identifiers to meta-variables.

These identifiers may be function names (i.e. belonging to $\mathcal{I}_F$), global (belonging to $\mathcal{I}_G$) or local (belonging to $\mathcal{I}_L$) variables. If local, they must have been declared before the edge in question.

Informally, a context is a criterion for selecting a set of program points (edges) and gathers specific information at each of these points, associating it to a set of meta-variables. It is basically a combination of a selector, a set of meta-variables and a contextualization function parameterized by the first two elements.

Let us immediately see an example of context and its application.

Example 5. Let us define a meta-variable with the name called. Recall that in Example 4, we defined a selector family S^{calling} that selects every edge in a control-flow graph leading to a calling instruction.

Let $\mathcal{M}_{\text{calling}}$ be the { called } set where called is simply an identifier.

Let us now define the following contextualization function for S^{calling} and $\mathcal{M}_{\text{calling}}$:

$$\theta_{\text{calling}} : (v_1, v_2) \mapsto [\text{called} \mapsto \{g\}] \quad \text{where } \text{instr}(v_2) = \textsc{Call}\ g.$$

In other words, this contextualization function maps any edge leading to a function call, to an environment associating the called identifier to (the global location of) the called function.

It is illustrated in Figure 2.3, where each selected edge e is decorated with the environment returned by $\theta(e)$. For example, the edge leading to Call `lock` is decorated with a mapping associating called to the `lock` singleton.

This contextualization function can easily be extended to a full context. We can define the *calling* context as the triple

$$C_{\text{calling}} := \left(S^{\text{calling}}, \{\text{called}\}, \theta_{\text{calling}}\right).$$

Concretely, the C_{calling} context selects all edges in a function leading to a function call and at each of these edges, binds the called identifier to the called function. Figure 2.3 illustrates this context as a whole.

The concepts defined in this section can be explained intuitively via a functional programming analogy. Essentially, a selector is a `filter` function on the edges of a function, while a contextualization function is a `map` function that maps edges to some local information about them. Overall a context is simply a `filter_map` function combining the two previous elements: it yields new data for some edges (an environment where meta-variables are bound).

As will be described in more details in Section 8.1.2, another way of understanding the notion of a context is through the Aspect Oriented Programming (AOP) paradigm [Kic+97]: contexts are pointcuts (a criterion identifying a set of control-flow points) at the specification level. Contrary to actual pointcuts in AOP however, contexts are entirely static: they do not depend on runtime parameters and must be statically determined.

2.4 A Few Useful Contexts

As mentioned in Section 2.3, a *context* can essentially be seen as a criterion for selecting edges in a function that may gather additional information (meta-variables) to be later used by the meta-predicate of a meta-property.

With the basic A-Lang, we have already defined one useful context: C_{calling} in Example 5, which designates every edge leading to a function call (and records the name of the called function).

We can come up with two other very simple contexts which do not use meta-variables but are still useful: *strong invariant* and *weak invariant*.

Example 6 (Strong invariant). We first define a selector family S^{all} that selects every edge in a function, then a contextualization function θ_{NOP} that always returns an empty mapping.

$$S^{\text{all}} := (E_f)_{f \in F};$$
$$\theta_{\text{NOP}} : e \mapsto \varnothing.$$

We can then define a context that gathers no meta-variables from the edges and just selects all of them:

$$C_{\text{strong invariant}} := \left(S^{\text{all}}, \varnothing, \theta_{\text{NOP}}\right).$$

This context will be used to stipulate that a predicate must hold at every step of a function: a strong invariant.

We can also define a notion of *weak invariant* based on the same principle, but selecting fewer edges.

Example 7 (Weak invariant). Instead of S^{all}, we define a selector family $S^{\text{frontiers}}$ that only selects edges at the entry and the exit of a function (see Definition 3 for begin and end):

$$S^{\text{frontiers}} := \left(\left\{(v_1, v_2) \in E_f \mid v_1 = \text{begin}(V_f) \vee v_2 = \text{end}(V_f)\right\}\right)_{f \in F}.$$

Then we can define a new context by simply replacing the selector

family:

$$C_{\text{weak invariant}} := \left(S^{\text{frontiers}}, \varnothing, \theta_{\text{NOP}}\right).$$

It will be used to specify that a predicate must hold in every possible initial state of a function as well as when it returns. However, the property can be broken locally. This is the definition of a *weak invariant*.

Both $C_{\text{weak invariant}}$ and $C_{\text{strong invariant}}$ are illustrated in Figure 2.4, as applied on function `checkStatus` of Figure 2.2. The selected edges are dashed and red (and each time, an empty environment is associated to them).

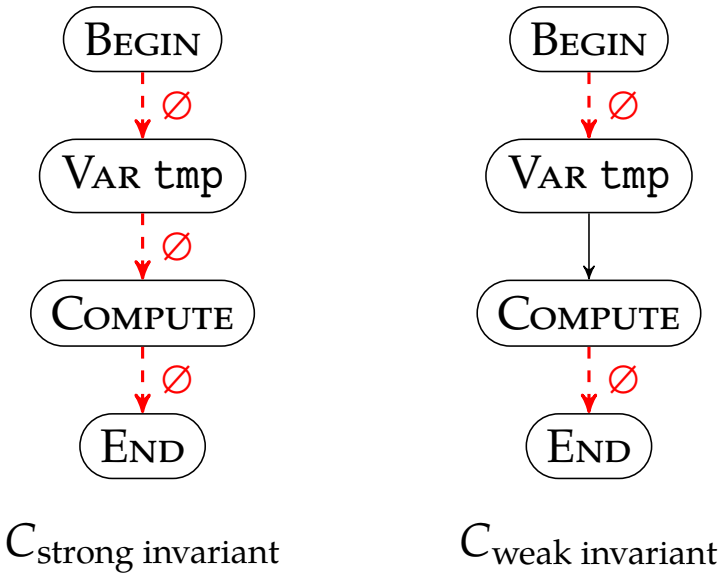

Figure 2.4: Illustration of invariant contexts on function `checkStatus` from Figure 2.2

Let us now refine A-LANG a bit to define what the *memory footprint* of an instruction is, for a given execution relation. It will allow us to characterize which variables an A-LANG instruction reads and writes, according to the execution relation. More specifically, the smallest set of locations such that any location outside of it is guaranteed to be left untouched, no matter the initial state.

The variables of the footprint must have been defined at the corresponding program point: they will belong either to global location identifiers G or to the set of previously defined local identifiers $\text{loc}_f(v)$ of the current node v.

Definition 12 (Writing memory footprint). Let $f \in F$ be a function, $v \in V_f$ one of its nodes, and $\Downarrow$ an execution relation.
We define its *writing footprint*

$$W(v) \subseteq \Big(G \cup \mathrm{loc}_f(v)\Big) \subseteq \mathcal{L}.$$

It is defined by the following rules, depending on the kind of $\mathrm{instr}(v)$:

- all instructions apart from Compute have an empty writing footprint (**including** Call):

$$\forall v \in V_f, \mathrm{instr}(v) \neq \textsc{Compute} \implies W(v) = \varnothing;$$

- if $\mathrm{instr}(v) =$ Compute, $W(v)$ is defined as the following set:

$$\bigcup_{\sigma \in \Sigma} W_\sigma(v)$$

where $W_\sigma(v) = \{l \in \mathrm{dom}(\sigma) \mid$
$\sigma(l) \neq \sigma'(l), \quad \langle v, \sigma\rangle \Downarrow \langle v', \sigma'\rangle, \quad v' \in V_f, \sigma' \in \Sigma\}$

Similarly, we define the reading memory footprin of an instruction over the state.

Definition 13 (Reading memory footprint). Let $f \in F$ be a function, $v \in V_f$ one of its nodes, and $\Downarrow$ an execution relation.
We define its *reading footprint*

$$R(v) \subseteq \Big(G \cup \mathrm{loc}_f(v)\Big) \subseteq \mathcal{L}.$$

It is defined by the following rules, depending on $\mathrm{instr}(v)$:

- If $\mathrm{instr}(v)$ is one of Begin, End, Var or Call:

$$R(v) = \varnothing;$$

- If $\mathrm{instr}(v) =$ Branch, then $R(v)$ is defined as the smallest set of

locations such that:

$$\forall v_e, v'_e \in V_f, \sigma, \sigma', \sigma_e, \sigma'_e \in \Sigma,$$
$$\sigma|_{R(v)} = \sigma'|_{R(v)} \ \wedge \ \langle v, \sigma\rangle \Downarrow \langle v_e, \sigma_e\rangle \ \wedge \ \langle v, \sigma'\rangle \Downarrow \langle v'_e, \sigma'_e\rangle \implies v_e = v'_e.$$

- If $\mathrm{instr}(v) = \textsc{Compute}$, then $R(v)$ is defined as the smallest set of locations such that:

$$\forall v_e, v'_e \in V_f, \sigma, \sigma', \sigma_e, \sigma'_e \in \Sigma,$$
$$\sigma|_{R(v)} = \sigma'|_{R(v)} \ \wedge \ \langle v, \sigma\rangle \Downarrow \langle v_e, \sigma_e\rangle \ \wedge \ \langle v, \sigma'\rangle \Downarrow \langle v'_e, \sigma'_e\rangle \implies v_e = v'_e \ \wedge \ \sigma_e|_{W(v)} = \sigma'_e|_{W(v)}.$$

Intuitively, $W(v)$ is the set of locations (global or local) that may be *modified* locally by v, and $R(v)$ is the set of locations on which v *relies* for its operation (hence, the *accessed* locations). While this definition is retrofitted on top of the previous formalization, in a concrete programming language, the notions of reading and writing would be naturally expressed in terms of the read and written variables inside the instructions.

Remark 6 (Locality). Note that since in our definition the Call instructions have an empty footprint, a location is part of a writing or reading footprint only if it is *locally* modified or accessed. This means that if an instruction v of f calls another function g and an instruction of g modifies the global location l, then l is *not* part of $W(v)$.

With this additional knowledge about our instructions, we can now define new useful contexts. First, a context concerned with write operations.

Example 8 (Writing context). We first define a family that selects all edges leading to an instruction that may modify one or more locations.

$$S^{\text{writing}} := \Big(\{(v_1, v_2) \in E_f \mid W(v_2) \neq \varnothing\}\Big)_{f \in F}$$

We then define a contextualization function mapping such edges to an environment that associates to the written meta-variable the set of

modified locations:

$$\theta_{\text{writing}} : (v_1, v_2) \mapsto [\mathsf{written} \mapsto W(v_2)].$$

Finally, the writing context can be defined:

$$C_{\text{writing}} := \left(S^{\text{writing}}, \{\mathsf{written}\}, \theta_{\text{writing}}\right).$$

Effectively, this context selects all edges in a function leading to an instruction that may write to a location and at each of these edges, binds the written identifier to the modified location.

Later, this will allow a high-level requirement to reason about the memory modifications of the program. As such, it is one of the most useful contexts, as we will see in the following section and chapters. Next, we define a similar context for read operations on memory.

Example 9 (Reading context). Symmetrically, we can define the *reading context* by substituting W by R and written by read in all the above definitions:

$$S^{\text{reading}} := \left(\{(v_1, v_2) \in E_f \mid R(v_2) \neq \varnothing\}\right)_{f \in F};$$
$$\theta_{\text{reading}} : (v_1, v_2) \mapsto [\mathsf{read} \mapsto R(v_2)];$$
$$C_{\text{reading}} := \left(S^{\text{reading}}, \{\mathsf{read}\}, \theta_{\text{reading}}\right).$$

This context selects all edges leading to a memory access and provides the accessed location as the read meta-variable. Together with C_{writing}, it allows us to reason about all memory operations in a meta-predicate.

While the contexts defined thus far were given as examples, they are actually sufficient to express a vast class of memory-oriented program properties which we can use to specify interesting and realistic requirements. Hence, it is important to remember these contexts, which we call the *base contexts*.

Definition 14 (Base contexts). We name five important *base contexts*:

- the strong invariant $C_{\text{strong invariant}}$ (Example 6);

- the weak invariant $C_{\text{weak invariant}}$ (Example 7);
- the calling context C_{calling} with its called meta-variable (Example 5);
- the writing context C_{writing} with its written meta-variable (Example 8);
- the reading context C_{reading} with its read meta-variable (Example 9).

2.5 Expressing Actual Requirements

We now pause the formalization effort to go back to the initial motivating use case we introduced in the beginning of this thesis. This section, intended as a breather, explores how we could specify the desired requirements with the tools formalized thus far and what is missing. The next two sections will properly formalize the missing parts.

Let us take the secure vault program defined in Example 2 and suppose that we want the security requirement that `vaultOpen` can only be modified in the `lock` and `unlock` functions.

Intuitively, this requirement is related to the *writing context* (Definition 14), which selects program edges where memory is modified and lists the modified locations at these points. Essentially, we would like to express that in all functions (except `lock` and `unlock`), the modified locations at these program edges cannot be `vaultOpen` (that is, $\texttt{vaultOpen} \notin \texttt{written}$).

We gather these facts into a concept called *meta-property*, which will be formalized in Section 2.7:

$$\hat{M}_{\text{security}} := \begin{cases} F \setminus \{\texttt{lock}, \texttt{unlock}\} \\ C_{\text{writing}} \\ \neg(\texttt{vaultOpen} \in \texttt{written}) \end{cases}$$

The first element is the *target set*, the set of functions in which we want to forbid modification. The second is the context: which edges we are concerned with in these functions. Lastly, we state a predicate which should be true at each of these program edges.

We state that in every function except `lock` and `unlock`, the location `vaultOpen` is not in the written set of any edge. Since we use the writing context, in which written refers to the set of modified locations, the meta-property can overall only hold if `vaultOpen` is never modified in the program, except in `lock`/`unlock`.

This meta-property is illustrated in Figure 2.5 where the two target functions are represented. Each node is now annotated with its writing memory footprint W (Definition 12). The writing footprint is assumed to be computed with respect to an execution relation derived from a particular implementation. The selected edges are dashed and red, and correspond as expected to edges leading to an instruction modifying memory. Notice that a call to a function does *not* count, even if the callee modifies memory. Lastly, each selected edge is decorated with the meta-predicate where the meta-variable written has been substituted by its local value.

Overall, we see that if we can verify the predicates next to the two dashed and red edges in Figure 2.5 for all possible executions, then we can verify the meta-property itself. And indeed it is easy to see how a violation of the meta-property (for example if `checkStatus` modifies `vaultOpen` instead of `tmp`) will result in a violation of one of the predicates.

Sections 2.6 and 2.7 properly define the underlying framework of high-level predicates and give semantics to meta-properties that map to the intuition we give in this section.

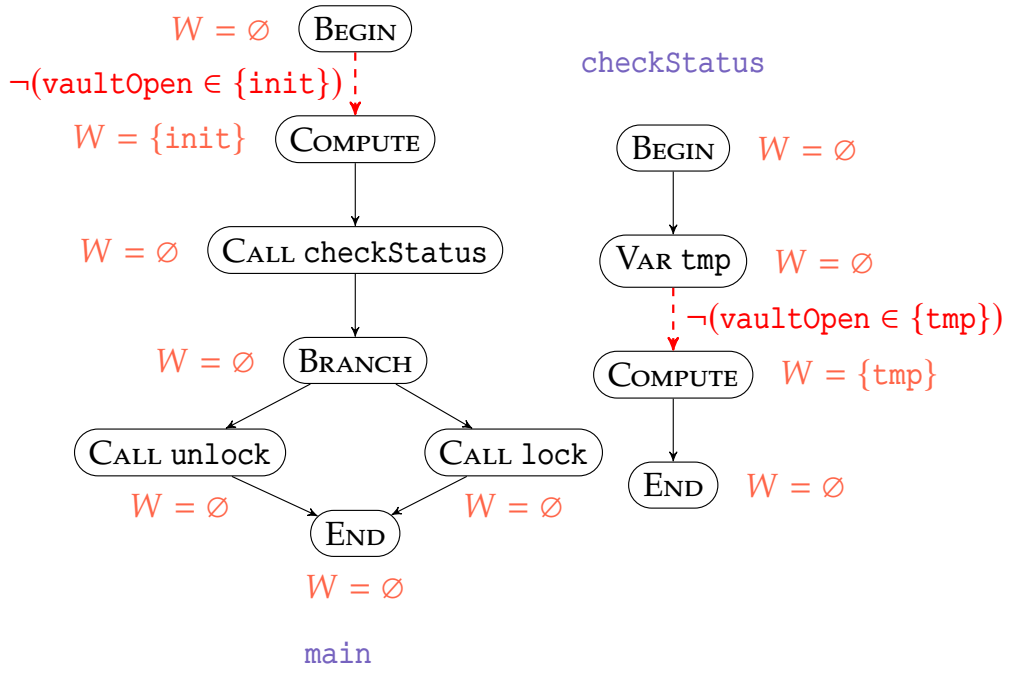

Figure 2.5: Semantics of the requirement on the vault example

We can also build more refined requirements. For example, suppose that our vault can be locked and unlocked in a dozen of different variations

of `lock` and `unlock` functions (and call this set of functions $F' \subseteq F$). Furthermore, a common condition for locking or unlocking the vault should be that a "key" is inserted.

Supposing location `keyInserted` is nonzero when this security feature is validated, this can be easily expressed as a variation of the above meta-property.

$$\hat{M'}_{\text{security}} := \begin{cases} F' \\ C_{\text{writing}} \\ \texttt{vaultOpen} \in \texttt{written} \Rightarrow \text{nonzero}\,\texttt{keyInserted} \end{cases}$$

The meta-predicate states that if `vaultOpen` is in `written` (which means that `vaultOpen` can be modified), then the value of `keyInserted` must not be zero (and we assume that it means the key is indeed inserted). It is complementary to the previous meta-property, which should now hold on $F \setminus F'$.

In general, we will see that meta-properties are a good fit for various security properties such as access control and information-flow control.

2.6 A Logical Framework

While contexts serve to specify precise program edges and gather information about them, we need a logic framework to express high-level properties as presented in the previous section. All further predicates in this chapter are expressed in a simple first-order logic that is partly left abstract, similarly to A-Lang. This logic is based on the standard propositional calculus [Men09], augmented with first-order quantifiers on locations and a membership operation for sets of locations.

> **Definition 15 (Predicate syntax, Expressions).** Let $L \subseteq \mathcal{L}_F$ be a **finite** set of memory locations. A *predicate* P over L is a formula of the language whose grammar is presented in Figure 2.6. It defines a variant of first-order logic relating sets and values. The set of such predicates is denoted by $\mathcal{D}(L)$.
>
> We call an *expression* a formula constructed with the $\langle expr \rangle$ rule of the grammar.

In the grammar of Figure 2.6, $\langle constant \rangle$ are constant values of type $\mathcal{V}$, $\langle log\text{-}unop \rangle$ and $\langle log\text{-}binop \rangle$ are unary and binary logical operators (taking values and yielding predicates), and $\langle unop \rangle$ and $\langle binop \rangle$ are unary and

$$
\begin{aligned}
\langle expr \rangle ::= \ & \langle loc \rangle \\
| \ & \langle constant \rangle \\
| \ & \langle unop \rangle \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \langle binop \rangle \ \langle expr \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle pred \rangle ::= \ & \langle log\text{-}unop \rangle \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \langle log\text{-}binop \rangle \ \langle expr \rangle \\
| \ & \neg \ \langle pred \rangle \\
| \ & \langle pred \rangle \ \vee \ \langle pred \rangle \\
| \ & \langle pred \rangle \ \wedge \ \langle pred \rangle \\
| \ & \langle pred \rangle \ \Rightarrow \ \langle pred \rangle \\
| \ & \forall \ \langle log\text{-}var \rangle, \langle pred \rangle \\
| \ & \exists \ \langle log\text{-}var \rangle, \langle pred \rangle \\
| \ & \langle loc \rangle \in \langle loc\text{-}set \rangle
\end{aligned}
$$

$$\langle loc \rangle ::= \mathsf{l} \mid \langle log\text{-}var \rangle$$

$$\langle loc\text{-}set \rangle ::= \mathsf{S}$$

$$\langle log\text{-}var \rangle ::= \mathsf{v}$$

where l, S and v are quantified, respectively over locations (L), location sets ($\mathcal{P}(L)$) and logic variable identifiers ($\mathcal{I}_V$).

Figure 2.6: The grammar of expressions and predicates

binary expression operators (taking values and yielding values). All of these operators are left abstract, since they depend on the actual set of values $\mathcal{V}$ used.

> **Example 10.** Let $a, b \in G$ be two global locations of a program which have an integer value ($\mathcal{V} := \mathbb{Z}$). We can define logical and arithmetic operators for these values:

$$\langle constant \rangle ::= [\text{'0'} - \text{'9'}]^+$$

$$\langle log\text{-}unop \rangle ::= \text{nonzero}$$

$$\langle log\text{-}binop \rangle ::= \text{'='} \mid \text{'<'}$$

$$\langle unop \rangle ::= \text{'-'}$$

$$\langle binop \rangle ::= \text{'+'} \mid \text{'-'}$$

In this setting, the following formula is a predicate of $\mathcal{D}(\{a, b\})$:

$$P := \forall i, i \in \{a, b\} \land i \leq a \Rightarrow (i = b) \lor i > b + 30.$$

In the next examples, we will describe the semantics of this predicate. For the sake of clarity, we will denote by P' the sub-predicate of P under quantification i.e.

$$P := \forall i, P'$$

We will now define semantic rules for predicates constructed with this grammar. Since predicates are an extension of propositional calculus, let us first define its semantics.

Definition 16 (Propositional calculus). Let $n \in \mathbb{N}$ be an integer, $L \subseteq \mathcal{L}_F$ a set of locations and $P, P_1, \ldots, P_n \in \mathcal{D}(L)$ a set of predicates.

$P_1, \ldots, P_n \vdash P$ denotes that P can be deduced from $P_1, \ldots, P_n$ by the rules of *propositional calculus* [Men09].

This definition encompasses all deductions that can be made with propositional calculus i.e. the semantics of operators $\neg$, $\land$, $\lor$ and $\Rightarrow$. Predicates constructed with other operators are considered as atoms for the propositional calculus (they are not interpreted).

Example 11. Let P'' be the following predicate of $\mathcal{D}(\{a, b\})$, which is P' from Example 10 where i has been substituted with b:

$$P'' := b \in \{a, b\} \land b \leq a \Rightarrow (b = b) \lor b > b + 30.$$

This predicate can be rewritten as $A \land B \Rightarrow C \lor D$ where A, B, C and D are predicates that would be considered atoms in propositional calculus.

In propositional calculus, the truth of this formula can be deduced from the truth of A, B and C, by the semantics of implication, conjunction and disjunction:

$$A, B, C \vdash A \wedge B \Rightarrow C \vee D$$

Hence we can write that P'' can be deduced from these three predicates:

$$b \in \{a, b\}, \quad b \leq a, \quad b = b \quad \vdash P''$$

Let us now build on these semantics to give a meaning to expressions and predicates.

Definition 17 (Expression and predicate semantics). Let e be an expression (see Figure 2.6), $v \in \mathcal{V}$ a value and $\sigma \in \Sigma$ a state. We say that e *evaluates* to v in σ and note

$$\sigma(e) = v$$

if this formula can be derived using the following semantic rule along with the abstract operators' semantics.

$$\textsc{Value}\ \frac{}{\sigma(l) = v}\ (l \in L \wedge \sigma(l) = v)\ .$$

Let $\sigma \in \Sigma$ be a state and $P \in \mathcal{D}(L)$ a predicate. We say that P *holds* in σ and note

$$\sigma \vDash P$$

if this formula can be derived using the semantic rules illustrated in Figure 2.7 (see Section 2.1 for notation details) along with the abstract operators' semantics (see Example 12 for an example of such rules).

In Figure 2.7, σ quantifies over states (Σ), $P, P', P_{1,2,\ldots}$ over predicates, v over logic variables ($\mathcal{I}_V$), l over locations ($\mathcal{L}$), S over sets of extended locations ($\mathcal{P}(\mathcal{L}_F)$). Furthermore, for a predicate P, $P[\mathsf{v} := l]$ denotes the *substitution* of every free instance of v by l in P. That means that if P contains a quantifier on the same logic variable, the substitution stops. In other words, logic variables are bound to the nearest quantifier.

The Prop rule lifts the propositional calculus deduction of Definition 16 to allow deducing that a predicate holds on a state if it can be deduced from

$$\textsc{Prop}\ \frac{\sigma \vDash P_1 \qquad \sigma \vDash P_2 \qquad \dots \qquad \sigma \vDash P_n \qquad P_1, \dots, P_n \vdash P}{\sigma \vDash P}\ (n \in \mathbb{N}^+)$$

$$\textsc{Mem}\ \frac{}{\sigma \vDash l \in S}\ (l \in S)$$

$$\textsc{Forall}\ \frac{\sigma \vDash P[v := l_1] \qquad \dots \qquad \sigma \vDash P[v := l_n]}{\sigma \vDash \forall v, P}\ (\{l_1, \dots, l_n\} = \mathrm{dom}(\sigma))$$

$$\textsc{Exists}\ \frac{\sigma \vDash P[v := l]}{\sigma \vDash \exists v, P}\ (l \in \mathrm{dom}(\sigma))$$

Figure 2.7: The semantics of predicates

other predicates with propositional calculus, and if these other predicates hold on the same state.

Furthermore, we introduce quantifiers *over locations* (and not values) with logic variables that are then substituted with every possible location of the state during evaluation.

To sum up, this logic introduces a notion of expression and operators to convert expressions into predicates (see next example for an example of actual evaluation rules on integers) as well as concrete propositional logic operators to compose predicates. Additionally, the logic introduces sets of locations (but not set composition operations) and a set membership operator.

> **Remark 7 (Set membership).** Notice that the Mem rule in Figure 2.7 does not evaluate the terms l and S in $l \in S$ to infer that $\sigma \vDash l \in S$, meaning that this checks that the location itself is part of the set, not its value. In other words, sets are sets of *identifiers*, hence set membership checks their syntactical equality rather than value equality.
>
> This means that the following predicate does not hold on the given state
>
> $$[a \mapsto 42, b \mapsto 42] \nvDash a \in \{b\}$$
>
> even though a and b have the same value.

Expression operations are left completely abstract since they depend on the language. However, we can instantiate them in an example: simple arithmetic over integer values.

Example 12. In Example 10, we syntactically defined new logical and arithmetic operators for integer values. We can now attach inference rules to the logical operators and evaluation rules to the arithmetic ones:

$$\textsc{Zero}\ \frac{\sigma(e) = v}{\sigma \vDash \text{nonzero}\, e}\ (v \neq 0)$$

$$\textsc{Equal}\ \frac{\sigma(e_1) = v_1 \qquad \sigma(e_2) = v_2}{\sigma \vDash e_1 = e_2}\ (v_1 = v_2)$$

$$\textsc{Comp}\ \frac{\sigma(e_1) = v_1 \qquad \sigma(e_2) = v_2}{\sigma \vDash e_1 < e_2}\ (v_1 < v_2)$$

$$\textsc{Minus}\ \frac{\sigma(e) = v'}{\sigma(-e) = v}\ (v = -v')$$

$$\textsc{Add}\ \frac{\sigma(e_1) = v_1 \qquad \sigma(e_2) = v_2}{\sigma(e_1 + e_2) = v}\ (v = v_1 + v_2)$$

$$\textsc{Sub}\ \frac{\sigma(e_1) = v_1 \qquad \sigma(e_2) = v_2}{\sigma(e_1 - e_2) = v}\ (v = v_1 - v_2)$$

where e, e_1, e_2 quantify over expressions and v, v_1, v_2 over values.

Now consider again the predicate presented in Example 10:

$$P := \forall i, i \in \{a, b\} \land i \leq a \Longrightarrow (i = b) \lor i > b + 30.$$

If we consider the program state $\sigma := [a \mapsto 100, b \mapsto -200]$, then $\sigma \vDash P$. A derivation tree to obtain this result is laid out in Figure 2.8. The top tree is a subtree of the bottom one, separated to fit in the page.

The first rule used is FORALL for universal quantification on a predicate: for every location l in $\mathrm{dom}(\sigma)$, the predicate where the logic variable has been substituted by l is a premise. Here, we separate into three cases: for $l = b$, $l = a$ or every other case. The first case is represented in a separate derivation tree for the sake of legibility. The second case is omitted, since it is very similar to the first.

In all cases, the next inference rule used is PROP, which has a set of premises which can be used to deduce the rule only using propositional calculus. The first case (in the top tree) is the deduction that is described in Example 11 where we use propositional calculus to derive $P'[i := b]$ (which is P'') from three sub-predicates. In this derivation tree, we must also prove that these three predicates hold on the state.

In the last case (on the right of the bottom tree) it is enough to prove that $l \notin \{a, b\}$ in order to prove the whole implication. As mentioned in Remark 7, notice that we do not check the values of a, l and b in the state.

$$\textsc{Prop}\ \frac{\textsc{Mem}\ \dfrac{}{\sigma \vDash b \in \{a,b\}} \quad \textsc{Comp}\ \dfrac{\textsc{Value}\ \dfrac{}{\sigma(b) = -200} \quad \textsc{Value}\ \dfrac{}{\sigma(a) = 100}}{\sigma \vDash b \leq a} \quad \textsc{Equal}\ \dfrac{\textsc{Value}\ \dfrac{}{\sigma(b) = -200}}{\sigma \vDash b = b} \quad \ldots \vdash P'[i := b]}{\sigma \vDash P'[i := b] \ \text{(which is } b \in \{a,b\} \land b \leq a \Rightarrow (b = b) \lor b > b + 30)}$$

$$(1)$$

$$\textsc{Forall}\ \frac{(1) \quad \vdots \quad \textsc{Mem}\ \dfrac{\dfrac{}{\sigma \vDash \neg l \in \{a,b\}} \quad \neg l \in \{a,b\} \vdash P'[i := l]}{\sigma \vDash P'[i := l]\ (\text{for } l \neq a, b)}\ \textsc{Prop}}{\sigma \vDash P'[i := b] \qquad \sigma \vDash P'[i := a] \qquad \sigma \vDash \forall i, P'}$$

Figure 2.8: Derivation tree for Example 12. Redundant derivations are omitted.

While we have a definition and semantics for a predicate holding true in a given state, we can also define what it means for a predicate to hold true at a given *program point*: if in every possible execution path to that point, the predicate holds.

Definition 18 (Predicate at program point). Let $f \in F$ be a function, $e = (v_1, v_2) \in E_f$ one of its edges, and $P \in \mathcal{D}(G \cup F \cup \mathrm{loc}_f(e))$ a predicate over the global variables of the program, the function names and the local variables in scope at e.

Given the starting vertex $\mathrm{begin}(V_f)$ of f, we say that *e satisfies P in the context of f* and note

$$e \vDash_f P$$

if $\forall \sigma, \sigma', \sigma'' \in \Sigma$,

$$\langle \mathrm{begin}(V_f), \sigma \rangle \Downarrow^+ \langle v_1, \sigma' \rangle \wedge \langle v_1, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle \implies \sigma'' \vDash_f P$$

i.e. if in every possible execution of f until e, P is satisfied by the state after the execution of e.

Given this basic definition of a predicate which can be evaluated on a state, we can define so-called meta-predicates, which are simply predicates where free variables can either be locations from a given set or identifiers from a meta-variable set (see previous section).

Definition 19 (Meta-predicate). Let $\mathcal{M} \subseteq \mathcal{I}_M$ be a set of meta-variables and $L \subseteq \mathcal{L}$ a set of locations. We define the set of all *meta-predicates* $\hat{\mathcal{D}}(\mathcal{M}, L)$ similarly to $\mathcal{D}(L)$, except that the free identifiers can also be part of the meta-variable set $\mathcal{M}$, in which case they are sets of locations:

$$\begin{aligned}
&\langle \textit{pred} \rangle ::= \langle \textit{log-unop} \rangle \ \langle \textit{expr} \rangle \\
&\quad | \quad \langle \textit{expr} \rangle \ \langle \textit{log-binop} \rangle \ \langle \textit{expr} \rangle \\
&\quad | \quad \ldots \\
&\quad | \quad \langle \textit{loc} \rangle \in \langle \textit{loc-set} \rangle \\
&\langle \textit{loc} \rangle ::= \mathsf{l} \mid \langle \textit{log-var} \rangle \\
&\langle \textit{loc-set} \rangle ::= \mathsf{S} \mid \mathsf{M} \\
&\langle \textit{log-var} \rangle ::= \mathsf{v}
\end{aligned}$$

where S is (again) quantified over location sets ($\mathcal{P}(L)$) and M over $\mathcal{M}$.

A meta-predicate $\hat{P} \in \hat{\mathcal{D}}(\mathcal{M}, L)$ can be transformed back to a normal predicate given an environment $m : \mathcal{M} \to \mathcal{P}(L^+)$ (as returned by a contextualization function, see Definition 11) by substituting every instance of a meta-variable with the set of locations it is associated with in m. This substitution is denoted by $\hat{P}(m)$:

$$\hat{P}(m) \in \mathcal{D}(\mathcal{L}^+).$$

It is called the *instantiation* of a meta-predicate on environment m.

These predicates will allow us to specify properties about high-level concepts such as "all modified locations" or "all called functions" of a program. Let us see a simple example of a dummy meta-predicate and its instantiation.

Example 13. The meta-predicate

$$\hat{P} := \forall i, i \in \mathsf{metav} \land i < a \implies \mathrm{nonzero}\, b$$

is a valid element of $\hat{\mathcal{D}}(\{\mathsf{metav}\}, \{a, b, c\})$, assuming a, b and c are *variable* identifiers (and not function names).

Given the environment $m = [\mathsf{metav} \mapsto \{b, c\}]$, we have

$$\hat{P}(m) = \forall i, i \in \{b, c\} \land i < a \implies \mathrm{nonzero}\, b.$$

Hence, with the program state $\sigma = [a \mapsto 3, b \mapsto 2, c \mapsto 42]$, we have that

$$\sigma \vDash \hat{P}(m).$$

These meta-predicates, along with the notion of context (Section 2.3), are the two essential bricks of our formalization of high-level requirements: meta-properties.

Remark 8 (Locations and extended locations). At the beginning of this chapter (Definition 1), we have defined location identifiers $\mathcal{L}$ and extended location identifiers $\mathcal{L}_F$, the difference between the two being that $\mathcal{L}_F$ includes function identifiers $\mathcal{I}_F$.

A state, as defined in Definition 5, associates values to *location identifiers*, not extended ones. This means that only local and global locations may have an associated value while function names do not.

A contextualization function (see Definition 11) returns an environment mapping meta-variables to *extended* location identifiers, hence a meta-variable can be mapped to a function name.

A predicate (see Definition 15) can contain function names. However, the semantics does not define their evaluation: such names can only be meaningfully used through the set membership operator.

2.7 Definition of Meta-Properties

We are now ready to define the class of high-level properties on A-Lang programs, which we called *meta-properties* in Section 2.5. These properties, with the help of a context and a meta-predicate, define in a single statement a potentially complex property that must hold true in a wide set of edges precisely defined for a given program.

Definition 20 (Meta-property). Let $\left(F, G, (V_f, E_f)_{f \in F}\right) \in \mathcal{O}$ be a well-formed program.

A *meta-property* of that program is a triple $(\hat{F}, C, \hat{P})$ where:

- $\hat{F} \subseteq F$ is a subset of program functions called the *target set*;
- $C := \left((S_f), \mathcal{M}, \theta\right)$ is a context (see Definition 11);
- $\hat{P} \in \hat{\mathcal{D}}\left(\mathcal{M}, G \cup F\right)$ is a meta-predicate (see Definition 19) over the program's global variables, function names and the meta-variables $\mathcal{M}$.

A meta-property $\hat{M} := \left(\hat{F}, ((S_f), \mathcal{M}, \theta), \hat{P}\right)$ is said to *hold true* on a program if

$$\forall f \in \hat{F}, \quad \forall e \in S_f, \quad e \vDash_f \hat{P}\left(\theta(e)\right).$$

In other words, a meta-property holds true if for all functions of $\hat{F}$ and for all edges characterized by the selector (S_f), the meta-predicate $\hat{P}$ instantiated at each of those points with the environment returned by θ holds true when evaluated with every possible execution state at the program point.

This definition and the concept of meta-property is the core of this work, since when instantiated with interesting contexts and languages

more refined than A-Lang, it allows us to define a very wide range of high-level requirements on a program.

To continue the Aspect Oriented Programming [Kic+97] metaphor from the previous section: while contexts can be compared to specification-level pointcuts, a meta-predicate can be seen as an *advice*, that is a piece of specification (instead of code, for AOP) meant to be inserted at various points (join points) quantified by the pointcut. As a whole, a meta-property can then be seen as an *aspect*, defining a cross-cutting concern: a specification that cuts across multiple abstractions (functions, modules, loops, ...) in a program.

Let us see a full example of a simple meta-property.

Example 14. Recall the *strong invariant* context $C_{\text{strong invariant}}$ from Example 6, which selects every edge in a function and gathers no particular information.

For any meta-predicate $\hat{P} \in \hat{\mathcal{D}}(\varnothing, G) = \mathcal{D}(G)$ (i.e. a simple predicate without meta-variables) and target set $\hat{F} \subseteq F$, we can define a meta-property

$$\hat{M}_{\text{strong invariant}} := \left(\hat{F}, C_{\text{strong invariant}}, \hat{P}\right).$$

Since the context simply selects every edge, $\hat{M}_{\text{strong invariant}}$ holds true if and only if

$$\forall f \in \hat{F}, \forall e \in E_f, \quad e \vDash_f \hat{P}$$

i.e. if $\hat{P}$ holds for every state at every point. Hence, this meta-property really states that $\hat{P}$ is a global strong invariant.

Let us now apply this concept to a security property previously mentioned in Section 2.5.

Example 15. Recall the $\hat{M}_{\text{security}}$ meta-property stated in the beginning of Section 2.5, meant to express the requirement that `vaultOpen` can only be modified in the `lock` and `unlock` functions:

$$\hat{M}_{\text{security}} := \begin{cases} F \setminus \{\texttt{lock}, \texttt{unlock}\} \\ C_{\text{writing}} \\ \neg(\texttt{vaultOpen} \in \texttt{written}) \end{cases}$$

Now that we have a clear semantics for each element of this property, we know that it holds if and only if

$$\forall f \in F \setminus \{\texttt{lock}, \texttt{unlock}\}, \quad \forall e \in S_f^{\text{writing}}, \\ e \vDash_f (\neg(\texttt{vaultOpen} \in \theta_{\text{writing}}(e)(\mathsf{written}))).$$

That is, if for every function except `lock` and `unlock`, at every edge preceding a memory modification, $\texttt{vaultOpen} \notin \mathsf{written}$ holds when written has been replaced by the set of locations modified after the edge.

In other words, outside the authorized functions, `vaultOpen` is never part of the writing footprint of instructions, which is what we wanted.

Through this chapter, we have defined a formal framework for high-level properties called *meta-properties*, defined on an abstract language A-Lang. A meta-property is essentially a predicate that is dispatched across the code of target functions according to a criterion called context. Furthermore, some contexts define *meta-variables*, which aggregate local information where the predicate is dispatched and can be used by it.

There are a few useful *base contexts* which will serve as a basis for the expression of real, complex requirements in all the following chapters. These contexts allow us to easily express memory-related requirements in particular.

While this framework allows us to specify high-level requirements, it also paves the way for techniques enabling their verification. Intuitively, the previous semantics states that a meta-property holds true if its predicate holds at each program point characterized by its context, after substitution of meta-variables. It is easy to imagine how in real code these substituted predicates could simply be assertions inserted at the same code point.

In order to refine this formal framework into a concrete solution to express and validate high-level requirements on a real programming language, the next chapter will review the features and specification language of the Frama-C platform, and apply the formalization of this chapter to obtain actionable meta-properties on C programs.

Chapter 3

High-Level Properties in C: HILARE

Adopt HILARE today, and laugh your bugs away!

While Chapter 2 provided a formal framework of our approach to describe high-level requirements, this chapter presents a concrete instantiation of this approach. The objective is to specify high-level requirements over C programs, through the introduction of a C-specific syntax for meta-properties called the HILARE (**H**igh-**L**evel **A**CSL **Re**quirement) language.

We do not start from scratch: our approach is based on Frama-C and its specification language ACSL, presented in Section 3.1. Section 3.2 then transposes all the concepts linked to meta-properties presented in Chapter 2 into the realm of C and ACSL, exposing the concrete syntax for C-specific meta-properties and their semantics with respect to the original meta-properties.

Section 3.3 describes extensions that venture outside this formalization (and were not formalized in the last chapter to keep it simple) but that we deemed useful to have in order to specify concrete requirements over C programs. Section 3.4 briefly mentions how these concepts are integrated within Frama-C (more on this in Chapter 4). Finally, Section 3.5 presents a longer, realistic example of HILARE specification on a system with confidentiality requirements.

From now on we will assume that the reader has a basic knowledge of the concepts of the C programming language [IsoC], and will recall some important concepts as we go along.

3.1 The Frama-C Framework

This section presents the Frama-C [Kir+15][Bau+21] platform, which provides an analysis framework for programs written in the C language. In particular, it allows users to semi-automatically machine-check formalized properties of programs.

3.1.1 C Program Analysis Framework

The C programming language is still widely used in the industry due to its extreme availability and low abstraction level, which make it particularly suitable as a system programming language. However, it is notoriously easy to write faulty C programs due to an absence of memory safeguards and an abundance of unsafe constructs. Hence, there are a variety of tools dedicated to the analysis of critical C code to alleviate the risk of serious bugs. One of them is Frama-C.

The Frama-C framework is an open-source analysis platform for industrial-scale C programs. Its architecture allows it to provide a wide range of analysis techniques through a variety of *plugins* (as we will see in Section 3.1.3). In particular, Frama-C can perform deductive verification of program properties via the Wp plugin (Section 3.1.3) and runtime assertion checking via E-ACSL (Section 3.1.3).

Frama-C consists of a kernel which is able to parse the C code and to store it in an internal representation. This representation can then be inspected and manipulated by plugins for program analysis and verification purposes. Frama-C also provides a graphical user interface for easier interaction.

The Frama-C kernel as well as its plugins are programmed in OCaml [Cuo+09]. The platform and the main plugins distributed with it are open-source (under the LGPL 2.1 licence).

3.1.2 ANSI/ISO C Specification Language (ACSL)

The ANSI/ISO C Specification Language (ACSL) [Bau+20a] is a formal behavioural specification language for the C programming language. It allows the specification of behavioural properties of a program within its source code.

It builds on the design-by-contract approach of the Eiffel programming language [Mey97] and its syntax is inspired from the Java Modeling Language (JML) [LBR99], which is a similar behavioural language for Java programs.

The rest of this section presents all the concepts of ACSL needed in the thesis. For an exhaustive presentation of ACSL, the reader can refer to its specification document [Bau+20a]. We will use the C function illustrated in Figure 3.1 as a working example.

```
int mem(int* T, unsigned size, int x) {
    int res = 0;
    for(unsigned i = 0 ; i < size ; ++i)
        if(T[i] == x)
            res = 1;
    return res;
}
```

Figure 3.1: The running example, a function `mem` testing the presence of a value in an array

This `mem` function takes an array `T` of integers along with its `size`, and an integer value `x`. It iterates over the array until it finds an element equal to `x`, in which case it sets a local variable `res` to 1 (which can be interpreted as the boolean true in C). It then returns this variable. We will want to specify that this function only returns 1 when the array contains `x`, and 0 otherwise.

Logic expressions. ACSL is based on a core of *logic expressions*, that closely map to C expressions (albeit they must remain pure), with additional constructs that allow the expression of useful logic concepts about C expressions and values.

Syntax	Semantics
`&&  ||  ==>  !`	Logical operators
`\forall type var; P`	Universal quantification
`\exists type var; P`	Existential quantification
`\union(S1,S2)`	Set union
`\at(expr,lab)  \old(loc)`	Value of location at label
`\valid(p)  \valid_read(p)`	Pointer validity
`\separated(loc1, loc2)`	Memory separation
`\overlaps(loc1, loc2)`	Memory overlapping

Figure 3.2: Important ACSL terms and predicates

Figure 3.2 describes most common operators that will be useful during this thesis. Similarly to the predicate language of the previous chapter,

ACSL is based on classical first-order logic: there is a distinction between predicates (which evaluate to true or false) and terms (which evaluate to values). Furthermore, while in the last chapter A-LANG and the predicate language had only one type (the type $\mathcal{V}$ of values), here we deal with all the existing C types.

While the first few operators should be self-descriptive, the last three lines contain constructs that are specific to C and worthy of attention:

- The `\at(expression, label)` construct allows one to state a property about the value of an expression *at a specific point of the program* identified by a *label*. The default label for any expression is `Here`, meaning that the annotation states a property about the value of some expression at the point where the property is defined (see next paragraph). There exist several automatically defined labels depending on the placement of the annotation, such as `Pre` and `Post` which refer to the state before and after the current function. Furthermore, any previously defined C label can be used as a label in `\at`. The `\at` operator can be used as an atom of a larger formula. Lastly, the construct `\old(e)` is a shortcut for `\at(e, Pre)`.

- `\valid(p)` where `p` is a pointer (or set of pointers) specifies that `p` points to a safely allocated memory location, i.e. dereferencing `p` is guaranteed to produce a definite value according to the C standard [IsoC], and writing to the pointed memory block is safe as well. Note that the type of `p` is taken into account since it changes the size of the pointed region: `\valid((int*)p)` and `\valid((char*)p)` [1] are in general not equivalent.

  The `\valid_read(p)` variant is similar but only specifies that the pointer is safe for dereferencing, without any warranty about writing to the pointed memory region.

- `\separated(loc1, loc2)` (where `loc1` and `loc2` are pointers[2]) holds if the blocks pointed by `loc1` and `loc2` do not overlap in memory i.e. if they are disjoint. `\overlaps` is simply the negation of `\separated`, introduced to make specification more legible[3].

[1] In C, `(t*)e` is a *cast* forcing expression `e` to be interpreted as a pointer on values of type `t`.

[2] Or sets of pointers, in which case this is checked for each pair.

[3] `\overlaps` is introduced for the needs of the thesis and does not exist in pure ACSL. However, it does exist in the context of HILARE (see next section) and the associated FRAMA-C plugin.

ACSL annotations. The C source code can be *annotated* with ACSL specification. This specification is written inside comments beginning with the special symbol `@`, either in the global scope or within functions.

Figures 3.3 and 3.4 illustrate the complete specification of the `mem` function presented in Figure 3.1. We will first explain the concepts of the first figure, which illustrates the *contract of the function*.

```
1 /*@
2     requires \valid(T + (0 .. (size - 1)));
3     ensures \result == 0 <==> \forall unsigned j;
4         0 <= j < size ==> T[j] != x;
5     assigns \result \from *(T + (0 .. size - 1)), size, x;
6 */
7 int mem(int* T, unsigned size, int x);
```

Figure 3.3: A contract for `mem`

Function contract. A function contract is an ACSL specification that immediately precedes the definition or declaration of a function in the source code (lines 1-6 in Figure 3.3). It generally contains three parts:

- a list of preconditions: properties that must hold in the state of the program whenever that function is called. Each precondition is specified using the **`requires`** keyword.

  Here on line 2 we assert that the `size` first cells of the array `T` must be valid memory addresses (if `size` is 0 then there is no constraint, which reflects that no cell is visited).

- a list of postconditions: properties that must hold in the state of the program when that function returns. Each postcondition is specified using the **`ensures`** keyword.

  Here, on lines 3-4, we have a postcondition stating the correctness of the function: the result is equal to zero if and only if no cell of the array contains the value `x`.

- a frame clause: an exhaustive list of non-local memory locations that can be modified by that function. It is specified by the **`assigns`** keyword.

  The **`assigns`** statement on line 5 indicates that each assignment inside `mem` may only affect its local variables or the value it returns

(via the `return` statement), symbolized by the term `\result`. As is the case here, the frame clause is optionally followed by a `\from` clause, which is described below.

Memory footprint. The frame clause can contain an optional description of the data-flow of the function with the `\from` keyword. It specifies the provenance of data for each modification of the state. Here, on line 5, we state that the function may only access `x`, `size` or the `size` first addresses of `T` to compute the result.

```
1  int mem(int* T, unsigned size, int x) {
2      int res = 0;
3      /*@ ghost unsigned cells_left = size; */
4      /*@
5          loop invariant 0 <= i <= size;
6          loop invariant res == 0 <==> (\forall unsigned j;
7          0 <= j < i ==> T[j] != x);
8          loop invariant cell_left == size - i;
9      */
10 L:   for(unsigned i = 0 ; i < size ; ++i) {
11         /*@ ghost --cells_left; */
12         /*@ assert rte: mem_access: \valid_read(T + i); */
13         if(T[i] == x)
14             res = 1;
15     }
16     /*@ assert cells_left = 0; */
17     /*@ assert \at(size, Pre) == \at(size, L); */
18     return res;
19 }
```

Figure 3.4: Inline annotations for `mem`

Let us now describe the various concepts illustrated in the *inline annotations* of Figure 3.4.

Assertions. Statement annotations allow writing annotations directly above a statement. The `assert P` clause is a statement annotation that ensures that a predicate `P` holds at a given program point. Hence, this statement may be used as a hypothesis for subsequent proofs in the code that follows it.

There is a variant using the `check P` clause which behaves similarly, except the assertion cannot be used as a hypothesis for further proofs (it must only check that the predicate holds at the given program point).

Loop invariant. The loop invariant that goes from lines 5 to 7 states that the loop counter `i` stays in the range $[0, size]$ and that at each iteration, we can assert that if `res` is still null then `x` was not found in the `i` first cells of the array. Loop invariants must hold at the beginning of each iteration (and after the last one, at the moment the loop condition is evaluated). Notice that when `i` takes its last value `size`, this is equivalent to the postcondition on line 3.

Ghost code. Although it does not serve any purpose in this example, the specification also contains *ghost code*. Ghost variables are variables declared for specification purposes only and cannot be used by the original code, and ghost instructions are statements that may only modify ghost variables (but can read the content of ghost as well as non-ghost variables). Furthermore, ghost code cannot modify the control flow of the original program. Thus, ghost code altogether cannot modify the original semantics of the code.

In our example, we declare the ghost variable `cells_left` on line 3 which corresponds to the number of cells left to explore in the array. This property is verified by the corresponding loop invariant on line 8 and ensured by the ghost instruction on line 11. Finally, we use the `assert` construct on line 16 to check that after the loop, there are no cells left to explore.

Automation. This annotated code can be fed to Frama-C with for example the ***R**un**t**ime **E**rrors* (RTE) plugin enabled. RTE emits annotations ensuring the absence of runtime errors, such as divisions by zero, invalid memory accesses or signed [4] overflows. In our case, RTE adds the assertion at line 12 of Figure 3.3. This checks that the program can always safely read `T[i]`.

3.1.3 The Plugin Ecosystem

The Frama-C kernel provides a framework for parsing C programs annotated with ACSL, and does not perform any analysis. Instead, it is easily

[4] Unsigned overflows have well-defined semantics in C (wrapping around). However, checking for their absence can also be enabled in RTE.

extensible through *plugins*, which can be combined to perform a wide variety of analyses and transformations. Figure 3.5 presents most of the existing plugins of the Frama-C ecosystem. They are grouped in several categories:

- *understanding*: plugins allowing a human to inspect a program and understand its structure and meaning. For example, Callgraph builds a call graph of the program (as can be expected).
- *support*: plugins that help automate the specification and verification of a program. For example, Pilat [OBP16] automatically generates polynomial numerical invariants over the loops of the program.
- *simplification*: plugins that transform the program to simplify it. For example, Constfold replaces constant expressions by their values in the code.
- *expressiveness*: plugins that *extend* ACSL to allow the specification of more properties. For example, Aoraï [SG11] allows the specification of temporal properties. This chapter introduces such a plugin to specify meta-properties: MetAcsl.
- *verification*: plugins allowing to validate specification, either statically or dynamically. Wp [Bau+20b] and E-ACSL [SKV17] are described in the following paragraphs.

ACSL is a specification language common to all plugins and can thus be used as a communication means between them.[5]

Each ACSL annotation represents a property, to which a *validity status* is associated (typically *Unknown*[6] before analyses are performed). Any plugin can decide to emit a validity status for an annotation at its own discretion.

In addition, a plugin can advise that a validity status depends on some hypotheses (other annotations or specific plugin parameters), possibly emitted by the plugin itself. Indeed, a plugin can choose to emit new annotations, in hope that another plugin (or a human) will be able to prove or use them. The Frama-C kernel maintains a global consolidated validity status for each annotation, taking into account the validity of its dependencies and ensuring the absence of dependency cycles.

We now briefly present two plugins that will be used in this thesis: Wp and E-ACSL.

[5] Plugins can also communicate via their respective API when available.

[6] Technically *Never tried*, or *Considered valid* for axioms.

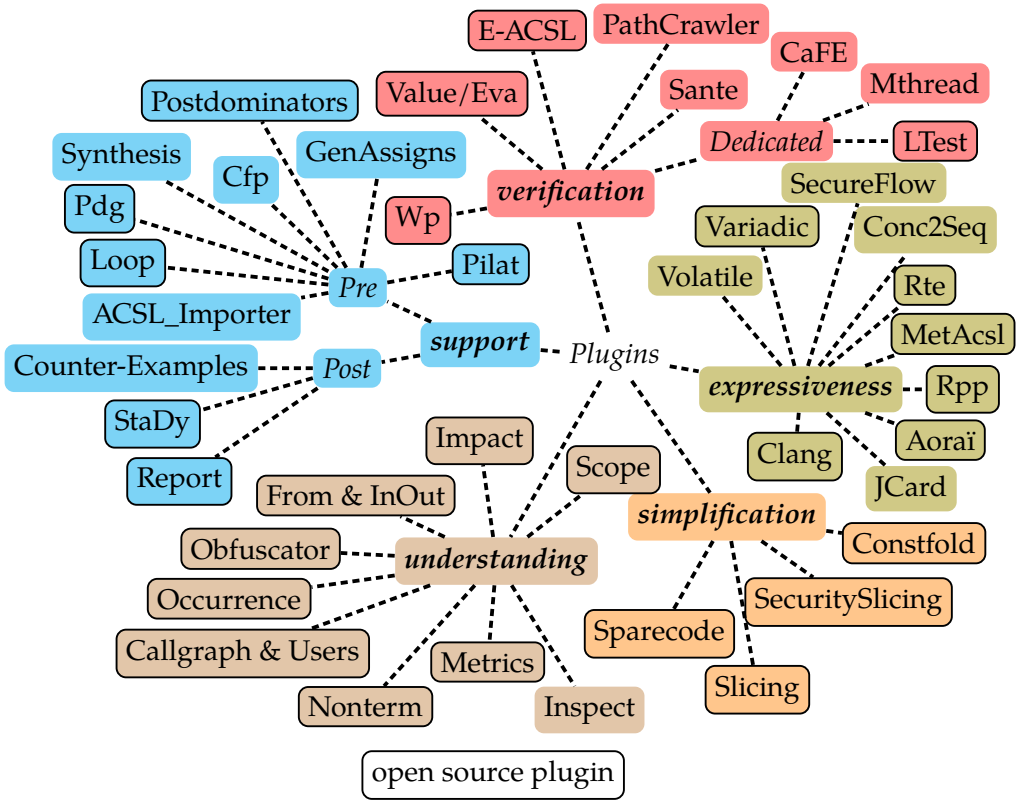

Figure 3.5: The Frama-C plugins, categorized

The Wp plugin. Deductive verification allows one to formally prove that the implementation of a function is correct with respect to its specification (see Chapter 1). The ***W**eakest **P**recondition* plugin (Wp) [Bau+20b] generates logical formulas encoding the semantics of ACSL annotations, known as *proof obligations* or *verification conditions*. It feeds them to the verification platform Why3 [FP13], which dispatches them to automated theorem provers such as Alt-Ergo [AltErgo] or Z3 [DB08] or to proof assistants such as Coq [Coq21], where the user helps unroll the proof.

If all proof obligations are validated, then the body of the function fulfils its specification.

As is always the case with deductive verification, it is sometimes necessary to add auxiliary annotations to help the tool complete the proof. Refer to Chapter 5 near the end of this thesis for more details about the methodology of verification in the context of high-level requirements.

The E-ACSL plugin. The E-ACSL [SKV17] plugin allows one to verify requirements at runtime without additional annotations by performing runtime assertion checking. Given a program annotated with ACSL, it transforms it into another program that can be compiled and fails at runtime if an annotation is violated.

3.2 HILARE Syntax and First Examples

To specify meta-properties (introduced in Chapter 2) in Frama-C, we propose an extension of ACSL syntax. It is meant to explicitly encode each element of the triple $(\hat{F}, C, \hat{P})$ [7] that constitutes a meta-property: a *target set*, a *context* and a *meta-predicate*. The HILARE language is the name of that ACSL syntax extension. We will refer to the C-specific meta-properties written with that syntax as HILARE properties or just HILAREs.

This chapter is essentially a transposition of the concepts presented in Chapter 2 into concrete syntactic and semantic elements for C and ACSL.

Definition 21 (HILARE structure). A *HILARE* is an ACSL annotation in the global scope with the following structure (white space does not matter):

```
/*@ meta \prop,
        \name(...),
        \targets(...),
        \context(...),
    P;
*/
```

where:

- `\name` contains an alphanumeric identifier;
- `\targets` contains an ACSL expression defining a set of function names;
- `\context` contains one of { **`\strong_invariant`**, **`\weak_invariant`**, **`\writing`**, **`\reading`**, **`\calling`**, **`\precond`**, **`\postcond`** };

[7] See Definition 20 in Chapter 2.

- `P` is an ACSL predicate over global variables of the program and meta-variables of the context.

An alternate form allows additional configuration flags (described in the next chapter):

```
/*@ meta \prop, \name(...),
       \targets(...), \context(...), \flags(...), P; */
```

Similarly to the predicates in A-LANG, we talk about the *instantiation* of a HILARE to designate the substitution of meta-variables with actual values within the predicate of that HILARE.

Remark 9 (Locations and addresses). In the previous chapter, meta-variables such as written were associated to A-LANG locations. Locations themselves were simply identifiers used as keys in a state σ.

In C, locations are allocated memory blocks, containing a value of a given type (unlike A-LANG where all locations had the same type, as we will see in Section 3.3.2). A location does not always have an identifier in C (i.e. a variable), instead it can universally be referred to using its *address*. In other words, in C, addresses play the role of locations.

Hence, in a HILARE, meta-variables are associated to *sets of addresses* by contexts. These addresses can refer to memory blocks of different types.

3.2.1 Target Set Specification

The target set is provided using the usual set syntax of ACSL. It can be explicit ($\{\texttt{f1}, \ldots, \texttt{fn}\}$), or use set operators such as `\union` or `\inter`. We also added the `\diff` operator for set difference, which does not exist in ACSL.

Since the goal for meta-properties is to be able to easily specify properties on large code bases, giving the explicit set of targets is rarely a practical solution. Instead, we provide a special variable `\ALL` which refers to the set of all functions in the program and is very convenient, along with the `\diff` operation, to specify target sets of the form "all functions except...".

As an additional way to ease the delimitation of the targets, we provide two constructs `\callees` and `\callers`. `\callees(f)` is the set containing `f` and all functions (transitively) called by `f`. `\callers(f)` is the dual

set containing `f` and all functions that (transitively) call `f`. It is especially useful when dealing with programs with clearly defined entry points.[8]

Since the C programming language lacks a module or name space system, it is common to simply group related functions in files. Consequently, it might be useful to refer to the group of functions defined in a file. This is the purpose of the construct `\in_file(filename)`.

The combination of these simple constructs allows for a convenient way to specify the scope of a meta-property without having to rewrite the target set when new functions are added to the implementation.

> **Example 16 (A complex target set).** The following set can be used as a target in a HILARE to describe "every function of the program except `foo`, `bar` or any function defined in `main.c`"
>
> ```
> \diff(\ALL,
>      \union({foo, bar},
>             \in_file("main.c")))
> ```

3.2.2 Available Contexts

As can be seen in the definition, one cannot use an arbitrary context as described in the previous chapter, but can choose within the *base contexts* previously defined (plus `\precond` and `\postcond`, as defined below). It turns out that these few simple contexts, combined with the expressiveness of ACSL itself, are enough to write quite interesting properties.

Recalling their definition from Chapter 2 (Section 2.4), we refine their semantics in the context of the C programming language instead of A-Lang. In particular, notions of locations, identifiers, etc. are mapped to concepts specific to C.

Weak Invariant, Pre/Postcondition. The `\precond` context returns only the starting edge of a function's CFG with no meta-variable, while `\postcond` does the same with the ending edges, and `\weak_invariant` combines both. Hence, it allows the specification of a predicate which must hold at the beginning of functions, at their end, or both.

Strong Invariant. The `\strong_invariant` context simply provides a contextualization mapping returning every edge of the CFG of a given

[8] This feature relies on the Frama-C plugin Callgraph, which makes gross but sound over-approximations of these sets in the presence of indirect calls (i.e. function pointers).

```
1  int A, B, C;
2  // level is the current confidentiality level
3  unsigned level, secret_size;
4  int* secret; // a secret array with secret_size elements
5
6  void main(); // main entry point
7  void def_level(int val); // set confidentiality level
8  void backdoor_root(); // backdoor that can always access secret
9  // return secret only if level is sufficient
10 int read_secret(unsigned n);
11 /*@
12   //A always remains equal to B in function main
13   meta \prop, \name(AB_same),
14       \targets({main}), \context(\strong_invariant),
15     A == B;
16   //The level can only be modified in def_level or backdoor_root
17   meta \prop, \name(modif_level),
18       \targets(\diff(\ALL, {def_level, backdoor_root})),
19     \context(\writing), \separated(\written, &level);
20   //The secret can only be read if level is at least ROOT_LEVEL
21   meta \prop, \name(can_read_secret),
22       \targets(\ALL), \context(\reading),
23     \separated(\read, &secret[0 .. secret_size - 1])
24     || level >= ROOT_LEVEL;
25   //Function backdoor_root is never called
26   meta \prop, \name(no_backdoor),
27     \targets(\ALL), \context(\calling),
28     \tguard(\called != backdoor_door);
29 */
```

Figure 3.6: Examples of HILARE properties and contexts

function without defining any meta-variables. It allows to specify a predicate that must hold *at every point* of functions. It is illustrated in the `AB_same` property in Figure 3.6.

Upon Writing. As in the previous chapter, the `\writing` context selects program points preceding instructions modifying the state and maps the `\written` meta-variable to the set of modified locations in the state i.e. to a set of *C addresses*.

In the context of C, this means that we can specify that a predicate must hold before every local modification of the state (e.g. through the assignment of a variable), and use the modified locations in the predicate. As before, local modifications do not include state changes made by other called functions (except in some circumstances, see next section).

The action of this context and the mapping to `\written` is illustrated in Figure 3.7.

Since `\written` is a meta-variable of this context, it can then be used by a predicate to form a useful HILARE. A simple example would be to forbid any local modification of some global variable, as shown by meta-property `modif_level` on Line 14 of Figure 3.6. It states that for any function that is not `def_level` or `backdoor_root`, whenever some memory location is modified locally, it must be unrelated to the global variable `level`.

Since we consider only local modifications, a call to `def_level` inside another function not allowed to modify `level` does not violate `modif_level`, even if `def_level` itself modifies `level`. Thus, `modif_level` can be seen as enforcing the proper encapsulation of `level`.

Upon Reading. The `\reading` context is identical to the `\writing` context except that it selects all edges leading to an instruction that *reads from* the memory and associates a meta-variable `\read` with the addresses being read by the instruction. It is illustrated by property `can_read_secret` in Figure 3.6.

> **Remark 10 (Behaviour on undefined functions).** Both the `\writing` and `\reading` contexts consider only local modifications: reads and writes within callees are ignored. This mirrors the behaviour of the memory footprints of A-Lang in the previous chapter.
>
> However, we make an exception for functions that are *declared but not defined within the program*. In C, this is mostly the case for functions

```
char* G;

/*@ assigns
    T[0 .. 40];
*/
void bar(int* T);
//bar is declared
//but not defined

void foo(int* p) {
    int i = *p;
    while(--i) {
        *p = i;
        bar(G);
    }
}
```

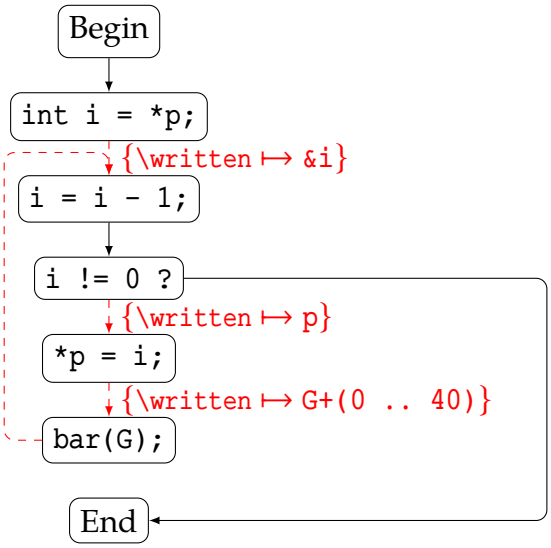

Figure 3.7: Illustration of the writing context on a small C program

that are part of an external library (e.g. the *libc*).

Section 3.1.2 introduces ACSL function contracts and their *frame clause*, which specifies which global locations a function may modify, and the *memory footprint* which specifies which locations are read throughout the function. These two elements are optional in a contract.

For functions that are declared but undefined, we look at their contract and use it to infer writing and reading memory footprints. We then consider these potential reads and writes as actual, local actions (i.e. within the scope of the caller function).

This is why in Figure 3.7, a call to the `bar` function is considered as a writing operation to the location range described in its **`assigns`** clause.

Upon Calling. Similarly, the **`\calling`** context selects all edges preceding a function call and maps a meta-variable `\called` with the function (or function pointer) that is being called.

Since the type signature of the function pointer mapped to `\called` is unknown at specification time, this meta-variable should be handled with care (as we will see in **Typing Troubles** in Section 3.3.2).

Example 17 (Back to the vault). Recall the meta-property stated in Chapter 2 to ensure that the variable `vaultOpen`, controlling a vault, is only modified within two functions `lock` and `unlock`.

$$\hat{M}_{\text{security}} := \begin{cases} F \setminus \{\texttt{lock}, \texttt{unlock}\} \\ C_{\text{writing}} \\ \neg(\texttt{vaultOpen} \in \texttt{written}) \end{cases}$$

This is how we would write it in the HILARE language:

```
meta \prop,
    \name(vault_security),
    \targets(\diff(\ALL, {lock, unlock})),
    \context(\writing),

\separated(&vaultOpen, \written);
```

Here we can easily see the parallel between the abstract and concrete properties.

3.3 Extensions of Meta-Properties in HILARE

The basic definition of the HILARE language presented in the previous section mirrors that of meta-properties in the previous chapter and enables the specification of many useful properties, as seen in the examples. However, the case studies (which will be described in Section 3.5) showed that it had several limitations, in both expressiveness and adaptability to the structure of programs. To address these limitations, we introduce some extensions to meta-properties.

3.3.1 Strong Invariant Relaxing

It is sometimes necessary even for a strong invariant to be temporarily broken. Equality between two variables (e.g. `AB_same` in Figure 3.6) is an example of that, as there is no way to change the value of the two variables in a single instruction. To overcome this issue, we add a `lenient` modifier that can be applied on a block of code to exclude the edges inside it from the scope of strong invariants.

Example 18. Say we have a function assigning a new, identical value to global variables `A` and `B`. In the absence of an atomic operation performing the two assignments, it has to do two successive assignments, therefore temporarily violating the `AB_same` invariant specified in Figure 3.6.

We can locally relax that invariant in the bloc surrounding the two instructions with the `lenient` modifier, preceded by the **`imeta`** clause[a].

```
void setAB(int v) {
    //@ imeta lenient;
    {
        A = v;
        B = v;
    }
}
```

Here, strong invariants will not be checked inside the annotated block (but will be immediately after).

[a] And *not* **`meta`**, since here the annotation is *inline*, as opposed to annotations in the global scope.

3.3.2 Typing Troubles

In Chapter 2, A-Lang locations all had the same type (they all yielded a *value* of the set $\mathcal{V}$). Hence, it was the same for meta-variables: they mapped to sets of locations of the same type.

However, in C, this is obviously not the case. For example, in Figure 3.7, the `\written` meta-variable of the **`\writing`** context is successively mapped to pointers of type **`int`**`*` and **`char`**`*`. Since this context accounts for all types of modifications, it could be mapped to types that are mutually incompatible: `\written` cannot be typed within the meta-predicate of a HILARE. Or rather, its type is the union of types of the mapped terms. Yet, this set of types is not known in advance when writing a HILARE.

Thus, nothing can be assumed for example about the type of `\written` when specifying a HILARE in the **`\writing`** context, except that it is always a pointer type (since it refers to the set of addresses that are modified).

Any other assumption would create a risk of typing error. For example, assuming there exists a C structure **`struct`** `S` with an `x` field, the presence of `\written->x` in a meta-predicate would make any instance of the HILARE related to an assignment to a location of another type ill-defined (see the

code below).

```
struct S { int x; };

/*@ meta \prop, \name(unsafe),
      \targets(main), \context(\writing),
    // \written is intended to be a struct S
    \written->x == 0;
*/

int main(...) {
    struct S u, v;
    v.x = 0;
    u = v; // Safe
    ...
    int i;
    i = 4; // Unsafe, i does not have an "x" field!
}
```

To deal with such vaguely typed meta-variables, the only usable ACSL operator is `\separated(l1, l2)`, described earlier, since its only requirement is that its parameters are pointers.

While this suffices to express interesting properties such as separation (see Figure 3.6), it does not allow reasoning about the value of the meta-variable. To address this issue, we introduce a construct to make assumptions about the type of a meta-variable while having a safeguard in case these assumptions were wrong.

More precisely, we add two predicates `\tguard` and `\fguard` that take an unsafe predicate (where a typing error might happen), behave as the identity if there is no error, and return respectively `\true` and `\false` otherwise. This allows the user to specify the previous example as `\tguard(\written->x == 0)`. If a particular instance of `\written` is of the expected type `struct S` (or any other structure with a field `x`) then its field is checked, else the property defaults to `\true` (i.e. we are only interested in modifications on locations of that type). Had `\fguard` been chosen instead of `\tguard`, any instantiation of that HILARE on a type that is not `struct S` would have defaulted to `\false`, effectively forbidding write operations to those types.

Intuitively, these functions should be used to guard any predicate that may be invalid for some instantiation of the HILARE. The default value to choose should reflect if these failures are expected in some cases or not: in

our example, `\tguard` allowed and ignored failures while `\fguard` did not.

> **Remark 11 (Handling unsafe function pointers).** As mentioned in Section 3.2.2, the `\called` meta-variable of the `\calling` context is mapped to the addresses of callees in target functions. These addresses are function pointers with unknown signatures.
>
> Although `\separated` is useful to distinguish two addresses, this construct does not handle function pointers: it's not possible to use `\separated(\called, my_func)` in a HILARE. The only way to identify which function is `\called` is to use an equality operator `==`, or a difference operator `!=`.
>
> However, the equality operators require both operands to have compatible types: while it will behave as expected for different functions with similar signatures, it will not type correctly with two functions of incompatible signatures.
>
> Hence, the common way to use the equality operator is wrapped inside a guard, as illustrated in property `no_backdoor` of Figure 3.6.

3.3.3 Labels in Meta-Properties

While the HILARE language allows specifying a property when some event defined by the context happens (e.g. a memory operation) and the safeguarding constructs enable that property to talk about the values of the meta-variables, sometimes we need to talk about the *effect* of the memory operation on these values.

For example, one may want to globally guarantee that some initially null global variable `G` is initialized only once to a strictly positive value. However, it is not possible to specify this without a means to refer to `G` *before* and *after* each modification, which would be needed to characterize our notion of initialization.

As mentioned in Section 3.1.2, one can use the `\at(expression, label)` construct to refer to the value of an expression *at a specific point of the program* identified by a *label*. An expression used without `\at` refers to its value at the point where it appears (which can be used explicitly with the `Here` label). There also exists two built-in labels `Pre` and `Post` referring respectively to the state before and after the current function. Furthermore, any previously defined C label can be used as a label in `\at`. The `\at` construct can naturally be used in the predicate P of an HILARE, with labels

`Pre`, `Post` and `Here` keeping their meanings. But these pre-defined labels do not allow us to link the values before and after a specific modification.

To tackle the aforementioned problem, we define two additional labels that are specific to the HILARE language and its contexts: `Before` and `After` that are used to refer to the states before and after the statements considered by the context, if any. These special labels may be mapped to actual ACSL labels by the contextualization function of a context when it makes sense to do so.

For example, the `\precond` context does not define them. The `\writing` context maps `Before` to `Here` (since by definition the edge returned by the context *precedes* a statement modifying the memory) and `After` to a C label inserted after the statement modifying the memory.

Example 19. With these labels, we can now write our previously problematic initialization meta-property:

```
meta \prop, \name(G_unique_initialization),
     \targets(\ALL), \context(\writing),
   \separated(\written, &G)
   || (\at(G, Before) == 0 && \at(G, After) > 0);
```

which means that each instruction either does not modify `G` or modifies it such that its value is `0` before the modification and strictly positive after it.

This extension makes HILARE requirements able to deal with very simple temporal properties, hence it overlaps with other existing Frama-C plugins specialized in specifying temporal properties such as Aoraï, or more generally other approaches similar to our own such as [CP05] and [TH02]. However, while this extension makes reasoning with local changes possible, it is definitely not a good tool for defining complex time-related requirements.

Remark 12 (Labels for function calling). Note that while Example 19 presents the usage of these labels within the `\writing` context, these can also be used within the `\calling` context, thus referring to the state before and after each call.

However, the labels do not make sense within other contexts where either nothing is modified (`\reading`) or the context do not correspond to a specific operation but are rather invariants.

3.3.4 Referring to Non-Global Values

As HILAREs are *global* properties that are not declared in the scope of any particular function, they can only refer to global variables and meta-variables. This is a strong limitation to the kind of properties that can be written, as some programs have few interesting objects declared in the global scope and typically pass them as arguments. To tackle this issue, we came up with two different mechanisms: the `\formal` construct and the notion of *local binding*.

Referring to function parameters Sometimes there is an object present in every target function of a meta-property as a consistently named function parameter, instead of a global variable. In these cases, we introduce the `\formal`[9] keyword to refer to such a parameter in the predicate of a HILARE.

When `\formal(some_param)` appears in a HILARE, each instantiation of the HILARE triggers the check that `some_param` is indeed a formal of the current target function. If it is, the `\formal` call is safely replaced by `some_param`. Otherwise, a typing error is triggered at the point where it is used.

Thus, `\formal` is best used when combined with the safeguarding constructs `\tguard` and `\fguard`, since it allows the specification engineer to *assume* that a formal is consistently defined in every target function and use it in a property, but to safely default to a conservative property if this assumption is wrong.

> **Example 20.** This HILARE specifies that if a function in the program takes a function pointer `pre_process` as a parameter, then it can only be called if it is distinct from a `do_not_call` function. If this parameter does not exist in a function, then the property defaults to `\true` since there is nothing to verify.

```
meta \prop, \name(pre_process_whitelist),
    \targets(\ALL), \context(\calling),
    ! \fguard(
        \called == \formal(pre_process)
        && \formal(pre_process) == do_not_call
    );
```

[9] A fuction parameter is also called a formal, hence the name.

Referring to bound names If it is not possible to rely on a consistent naming of formals across functions, we introduce a notion of *binding* to overcome this difficulty with some help from the user.

We introduce two special functions, `\bind` and `\bound`. The first one is to be used outside a HILARE, in the body of a C function, to *bind* a name to the value of a C expression at that point. This name is a new global identifier and must not exist in the source code before.

This name can then be used in a HILARE to formulate an interesting property about the value it refers to. A name can actually be bound multiple times to different values at different points of a program, meaning that the name inside a HILARE refers to the whole set of associated values.

Consequently, the same name can only be bound to values with mutually compatible types, and referring to that name with `\bound` in a HILARE will yield a value of the inferred type (unlike meta-variables where the type is not known in advance).

Example 21 (Bindings). The whole process is illustrated in Figure 3.8, where we use bindings to specify a property about all blocks allocated on the heaps by a single function.

We have a program with a function `create_cell` that allocates a new integer on the heap and returns its address each time it is called. We also have a global variable `lock`, that we assume is used in other parts of the program. We want to verify that all integers allocated by `create_cell` can only be modified when `lock` is not zero (which could be useful, for example in the context of a multithreaded application).

First, we use bindings to assign a new, global name to the set of all integers allocated with `create_cell`. Indeed, in the body of that function, we *bind* the temporary `c` variable, which contains a freshly allocated integer, with a new global name `cells`.

Then, we use that name in the HILARE at the end of the program, wrapped in a `\bound` function, as if it were a single variable. In this HILARE, `\bound(cells)` will refer to any variable allocated by `create_cell`, hence the HILARE must hold for every possible value.

Notice that the bound values are constant but may be pointers referring to changing memory. We are then specifying a property across all the memory states of the different instantiations.

```
int lock; // A global, imaginary lock

//A wrapper around malloc for single integers
//The name 'cell' will refer to every int allocated
//by this function
int* create_cell() {
    int* c = (int*) malloc(sizeof(int)); //assumed to succeed
    //@ imeta \bind(c, cells);
    return c;
}

//Modifies a cell if the lock is available
int safe_modify_cell (int* cell, int val) {
    if(!lock) {
        //Take the lock
        lock = 1;
        *cell = val;
        //Release the lock
        lock = 0;
        return 0;
    }
    else return -1;
}

//Modifies a cell arbitrarily
void unsafe_modify_cell (int* cell, int val) {
    *cell = val;
}

/*@ //Pointers returned by create_cell
    //can only be modified when the lock is taken
    meta \prop, \name(cell_modif_is_critical),
      \targets(\ALL), \context(\writing),
    \separated(\written, \bound(cells)) || lock;
*/
```

Figure 3.8: The usage of name binding, illustrated

3.3.5 Referring to Callee Parameters

When using the `\calling` context and targeting a specific callee, it is possible to refer to the value provided for a formal at every call site, using the `\called_arg(parameter_name)` term. It throws a type error when the called function does not have the specified formal, is indirectly called (through a function pointer) or is a variadic function. Thus, it should be surrounded by a guard (see Section 3.3.2).

Example 22. For example, the following HILARE states that parameter `x` of the function with signature `float sqrt(float x)` should never be provided with a negative value.

```
meta \prop,
    \name(sqrt_pos),
    \targets(\ALL),
    \context(\calling),
\tguard(\called == sqrt ==> \fguard(\called_arg(x) >= 0.));
```

In this contrived example, of course, the same goal could also be achieved by adding a precondition in the contract of `sqrt`, or even with the help of the `\precond` context.

Remark 13 (Fragile specification). For global-level specification to be feasible, it is necessary to have anchor points in the program that we can refer to in the specification. It is often global variables names, function names or in this case even function parameter names.

The more local the name we're referring to is, the more fragile to program changes the specification becomes. While MetAcsl is quick to alert that an assumption about names made in a HILARE is no longer valid, this can be silenced by guards. Hence, it is important to be extremely careful when using guards to avoid silent errors, for example by preferring defaulting to a false predicate rather than ignoring errors when possible.

3.4 The MetAcsl Plugin for Frama-C

The HILARE syntax and its extensions are not supported natively in ACSL. Hence, we introduced a new plugin in the Frama-C ecosystem: MetAcsl.

Any source code containing HILARE annotations (delimited by the `meta` or `imeta` keywords, depending on whether the annotation is global or inline) can then be correctly parsed by Frama-C, and other plugins can be invoked on the resulting source code.

This plugin is available to the public under the LGPL licence [10]. Its capabilities are not limited to just parsing HILARE specification: the next chapter describes in detail how it can be used to actually verify that a program is compliant with requirements specified through the HILARE language.

> **Remark 14 (On nomenclature).** Now is a good time to summarize the different names we've introduced and recall the differences between them:
>
> **High-level requirements** are properties that we want a program to uphold, expressed in natural language.
>
> **Meta-properties** are a formal class of high-level properties, presented in Chapter 2. They aim at being a general-purpose framework for expressing high-level requirements.
>
> **HILARE** is the name given to the extension of ACSL for specifying meta-properties, discussed in the current chapter. A meta-property specified using this syntax is called a HILARE as well.
>
> **MetAcsl** is the Frama-C plugin implementing the HILARE language and its verification strategy. It will be described at length in Chapter 4.

3.5 Complex High-level Requirements as HILARE

We present a small-scale case study where the HILARE language is useful to specify confidentiality and integrity properties such as memory access control. This earlier case study was imagined to evaluate the relevance of our approach on actual code and properties.

First, we describe the content of this case study and how useful properties about it are specified using HILAREs. In the next chapter, we will

[10] Available on Gitlab at the following location: https://git.frama-c.com/pub/meta.

present different assessment techniques to check the validity of the implementation with respect to these HILAREs.

See Chapter 5 for a full methodology for reasoning about the requirements of a program using the HILARE language and Chapter 7 for a detailed case study of a real, industrial-scale system.

3.5.1 Presentation of the Case Study

The case study, which was suggested by an industrial partner, deals with a confidentiality-oriented page management system, described as follows:

Definition 22 (Confidential page manager). The *manager* is a C system library allowing user processes to request the allocation of memory regions (pages) as well as requesting modifications, access or de-allocation of these regions.

It has the following basic properties:

1. Each page has a fixed size.
2. There is a maximal number of currently allocated pages. Above that number, allocation requests are denied.
3. The library does not directly expose pointers to the memory regions but instead exposes opaque structures.

as well as a number of confidentiality facilities:

4. We assume each user process (or *agent*) has a *confidentiality level* (an integer or any member of an ordered set), registered and exposed by the operating system.
5. Each allocated page has a confidentiality level as well, equal to the level of the agent that allocated it.

There are two basic confidentiality guarantees that such a system should offer: confidentiality of both *read* and *write* operations.

Requirement 3.1. An agent can never *read* from a page with a confidentiality level *higher* than its own (to preserve the confidentiality of the data written on the page).

> **Requirement 3.2.** An agent can never *write* to a page with a level *lower* than its own (to prevent the agent's data from being read by lower agents in the future).

There are several corollaries needed for requirements 3.1 and 3.2 to be useful in ensuring confidentiality (see Chapter 5 for a deeper discussion on how to ensure a set of requirements is robust enough):

> **Requirement 3.3.** The confidentiality level of an allocated page remains constant.

> **Requirement 3.4.** The allocation status of a page can only be modified by the allocation and de-allocation functions.

> **Requirement 3.5.** Non allocated pages are neither accessed nor modified.

> **Requirement 3.6.** Non allocated pages do not retain old data.

We also consider an extension of this system introducing *encryption* as a means to decrease the confidentiality level of a page. Two functions to encrypt and decrypt a page are added to the interface with a key based on the confidentiality level of the caller, and we weaken requirement 3.3 into:

> **Requirement 3.3b.** The confidentiality level of an allocated page remains constant, except in encryption/decryption functions.

Notice that these properties ensure the *confidentiality* but not the *integrity* of data – that is, the fact that data cannot be corrupted by non-privileged users – which is not considered here but could be similarly specified.

We wrote a simple implementation of this case study, where the system is modelled by a stateful interface of functions to allocate, free, write to or read from pages. The confidentiality level of the calling agent is represented by a global variable, which is assumed to be securely modified when the context changes. The interface of the library is listed in Figure 3.9, where the type `struct Page` is internally defined as:

```
enum allocation_status {PAGE_ALLOCATED, PAGE_FREE};
//Struct for page metadata
struct Page {
    char* data; //First address of the page
    enum allocation_status status; //Allocation status
    unsigned confidentiality_level; //Confidentiality of the page
    unsigned encrypted_level; //Confidentiality before encryption
};

struct Page;
enum result {PAGE_OK, PAGE_ERROR};

//Current confidentiality level of the agent
extern unsigned user_level;

// Must be called for the initial state to be valid
enum result init();

// Main functions
struct Page* page_alloc();
void page_free(struct Page* p);
enum result page_read(struct Page* from, char* buffer);
enum result page_write(struct Page* to, char* buffer);

// Encryption extension
enum result page_encrypt(struct Page* p);
enum result page_decrypt(struct Page* p);
```

Figure 3.9: Public programming interface of the page manager

3.5.2 Specification of the Requirements

All of these properties can be expressed using meta-properties, as illustrated in Figure 3.10 where requirements 3.1, 3.3b and 3.6 are specified.

> **Remark 15.** The `forall_page` predicate is a formula-shortening macro which quantifies over the globally-stored array of pages (both free and allocated), and `page_allocated`, `page_level`, `page_data` and `clean_page` are ACSL functions and predicates abstracting low-level operations for legibility purposes. Their definition is listed in Ap-

pendix A.1.

The first HILARE, `req_1`, makes use of the **`\reading`** context and applies to all functions of the library. It quantifies over all pages, binding variable `p`. It then states that if a page is allocated and has a higher confidentiality level than the current process, then the memory of that page cannot be read by the process (using the **`\separated`** predicate and the `\read` meta-variable presented in Section 3.2).

Property `req_3bis` uses a similar pattern with the **`\writing`** context to ensure that the confidentiality level field of a page cannot be written to while the page is allocated, except in the encryption and decryption functions (notice the exceptions in the target set made using the **`\diff`** operator).

The last requirement, `req_6`, states that at all times, free pages should not retain earlier data (the `clean_page` predicate checks that all bytes are null).

Remark 16 (Specification scope). Here, we are specifying the system library itself and not the potential user programs calling the interface. Should the library be verified against this specification, this should ensure that it cannot be misused to violate the confidentiality requirements.

The complete implementation and specification of this contrived example is available in Appendix A.2 and A.3 or online [11] for easier navigation. Refer to Chapter 5 for a detailed discussion about the design of a HILARE specification.

[11] https://git.frama-c.com/pub/meta/-/tree/master/case_studies/confidentiality

```
//Never read from a higher confidentiality page
meta \prop, \name(req_1), \targets(\ALL),
    \context(\reading),
        forall_page(p,
            page_allocated(p) && user_level < page_level(p) ==>
            \separated(page_data(p), \read)
        );
//The confidentiality of an allocated page
//is constant outside of encryption
meta \prop, \name(req_3bis),
    \targets(\diff(\ALL, {page_encrypt, page_decrypt})),
    \context(\writing),
        forall_page(p,
            page_allocated(p)
                ==> \separated(&p->confidentiality_level, \written)
        );

//The content of a free page is always null
meta \prop, \name(req_6), \targets(\ALL),
    \context(\strong_invariant),
        forall_page(p, !page_allocated(p) ==> clean_page(p));
```

Figure 3.10: Specification of confidentiality requirements with HILARE

Chapter 4

Assessing HILARE by Generating Code Annotations

While the two previous chapters introduce a specification approach for high-level requirements on C programs, nothing is said about the actual verification of such a program with respect to its HILARE specification.

This chapter bridges the gap by presenting a general principle for transforming a HILARE to a set of ACSL code annotations in Section 4.1. The subsequent sections widen that principle to the extensions to the HILARE language presented in the previous chapter and discuss performance and scalability considerations. Finally, the technique is used to enable the verification of the confidentiality-oriented case study described in the previous chapter, via both deductive verification and runtime assertion checking (see Chapter 1 for a description of these terms).

4.1 General Principle: Instantiation of Meta-Predicates

As mentioned in Chapter 3, several existing Frama-C plugins provide useful and efficient analysis of ACSL-annotated C code, such as deductive verification [Bau+20b] or runtime assertion checking [SKV17].

Following the usual Frama-C approach of *tool collaboration*, we wish to take benefit of existing analysers without re-implementing them for HILARE. To do that, we designed a way to transform HILAREs into plain ACSL annotations while keeping links between the original HILAREs and their ACSL translation. Hence, the existing tools can understand and anal-

yse the translation and their results for the translated ACSL annotations can then be interpreted in terms of HILARE.

The method is based on the initial definition of the HILARE language and meta-properties in Chapter 2: a meta-property is a combination of a target set, a context and a meta-predicate. The semantics given to a meta-property states that it is valid if and only if the meta-predicate holds for every path at every edge selected by the context, after meta-variables have been replaced by their local correspondence. We called this the instantiation of the meta-predicate (Definition 19) and the overall semantics was illustrated in Figure 2.5 in Section 2.5.

The idea is to generate a set of assertions corresponding to this semantics of a meta-property (hence of a HILARE):

Definition 23 (Equivalent Annotation Generation). For a given C program and a HILARE with the form

```
/*@ meta \prop, \name(...),
    \targets(S), \context(C), P; */
```

we use the following algorithm to *instantiate* the HILARE:

for all function $F \in S$ **do**
  for all program point p selected by C **do**
    $P' \leftarrow$ INSTANTIATE(P, p, C) ▷ Instantiate meta-variables in P
    ASSERT(P', p) ▷ Add `assert P';` at point p
  end for
end for

That is, we create an assertion for each necessary instantiation of the meta-predicate. We call the generated set of ACSL annotations the *instantiation* of the HILARE.

Remark 17 (Equivalence). According to the semantics of meta-properties, the HILARE holds if and only if all of its instantiations hold.

This needs no proof: it is correct by construction, since the generation algorithm is a literal mechanization of the semantics.

Remark 18 (Local naming conflicts). In some cases, the meta-predicate may bind a local logical variable, for example by quantifica-

tion: `\forall int i; ...`. However, meta-variables present in the predicate may be replaced with free C variables bearing the same name, provoking conflicts and soundness problems.

For example, the following predicate

```
\forall int* i; i != NULL ==> \separated(i, \written)
```

becomes nonsensical if `\written` is replaced by a local C variable named `i` (and even more so if that variable is not an `int*`).

To prevent these problems, we perform an α-conversion step for every predicate instantiation, if there is a conflict between any bound logic variable of the predicate and a C variable in scope with the same name, the bound variable is substituted by an appropriate fresh name.

Since meta-properties have more expressive power than ACSL, it is often impossible to transform a meta-property into a single ACSL annotation. In some cases, a meta-property is translated into function contract clauses (e.g. for weak invariants) but in most cases it has to be captured by assertions inserted directly into the body of a function.

```
int* G;

/*@ assigns T[0 .. 40]; */
void bar(int* T);

void foo(int* p) {
    int i = *p;
    while(--i) {
        *p = i;
        bar(G);
    }
}
```

(a) original program

```
// \separated shortened to \sep
void foo(int* p) {
    int i = *p;
    while(1) {
        /*@assert \sep(&i, G);*/
        i = i - 1;
        if(i == 0) break;
        /*@assert \sep(p, G);*/
        *p = i;
        /*@assert
        \sep(G+(0..40), G);
        */
        bar(G);
    }
}
```

(b) instantiation on function `foo`

Figure 4.1: Basic generation strategy, illustrated

Example 23. Let us declare a HILARE, stating that in function `foo`, the global variable `G` should not be modified.

```
meta \prop, \name(G_is_constant),
    \targets({foo}), \context(\writing),
    \separated(\written, G);
```

In Figure 4.1 is a very simple (and useless) program on the left, and on the right the instantiation of property `G_is_constant` in function `foo`.

First, notice how the function has been syntactically de-sugared (the conditional loop is now an infinite loop with a conditional break; the side effect of the condition has its own instruction). This is a pre-processing performed by Frama-C itself, allowing a clear separation of program points.

Second, notice that assertions have been inserted *before* each instruction modifying the memory: this corresponds to the edges selected by the `\writing` context of the HILARE. Notice how the initialization of a variable is not considered a modification if performed during its declaration.

Each assertion is the original meta-predicate `\separated(\written, G)` where the meta-variable `\written` has been replaced by the actual address (or set of addresses) modified by the following instruction.

Finally, notice that while the `\writing` and `\reading` contexts normally only select *local* memory operations and not those performed by called functions (as described in Section 3.2.2), here an assertion is inserted before the call to `bar`. This is because this function is declared but not defined, which is an exception described in Remark 10 of Chapter 3. In this case, the *contract* of the function is used to over-approximate the effects on memory, hence the `G + (0 .. 40)` address range.

This annotation generation technique is implemented in the MetAcsl plugin presented in Chapter 3. We will describe the details of the plugin and its usage in Section 4.4.

Remark 19 (Another way...). Later in the thesis (in Chapter 6), we will describe another way HILARE requirements can be assessed without resorting to local assertion generation: by *deducing* them from other high-level requirements within a formal deduction system.

4.2 Automatic Simplification

While the proposed technique is simple, it entails that a predicate is instantiated for each selected edge in each target function. Thus, the number of instantiations can quickly become high enough (for example when using the *Strong Invariant* context) to become a problem for the Frama-C plugins that are expected to analyse the translated program, resulting in potentially long analysis times or loss of precision in the results.

However, we have observed that when a meta-property has been instantiated, there are a lot of cases where the resulting assertion is trivial to prove or disprove. For example if $\hat{P}$ is `\separated(\written, &A)`, and P is `\separated(&B, &A)` where `A` and `B` are different variables (thus separated by definition): this instantiation is trivially valid and its actual insertion can be skipped.

Thus, MetAcsl performs a *simplification phase* for assertions where trivially simple patterns such as the one mentioned above (with variations for potentially nested structures and arrays) are recognized and replaced by their truth value, which is then propagated through the property. Hence, the instantiations left in the code are those that could not be simplified, and for which other plugins should attempt a more thorough verification. The quantitative evaluation of this simplification will be discussed in Section 4.5.

4.3 Transformation of Extensions

Section 3.3 introduced several extensions to the original HILARE definition, in particular allowing for new means to refer to variables both local and global in the specification.

While some extensions do not affect the semantics of the HILARE language, for the others it is necessary to devise a verification mechanism compatible with the one presented in Section 4.1.

4.3.1 Labels in Meta-Properties

Section 3.3.3 presented the labels `Before` and `After`, that can be used with the ACSL construct `\at` to allow referring to the value of an expression respectively before and after the program points selected by the context.

To generate assertions that correctly convey these semantics when instantiating meta-predicates, we leverage the fact that `\at` can take nor-

mal C labels as input and appropriately generate C labels in the code surrounding the insertion point of a predicate when necessary.

Example 24. Let us consider the same original program from Example 23 and the following HILARE:

```
meta \prop, \name(G_increasing),
    \targets(\ALL), \context(\writing),
    \separated(\written, G)
    || (\at(*G, Before) <= \at(*G, After))
```

It states that the memory cell pointed by global variable `G` can only be modified if it increases its value.

We generate annotations as follows:

```
void foo(int* p) {
    int i = *p;
    while(1) {
        _before_1:
        i = i - 1;
        /*@assert \separated(&i, G) ||
            (\at(*G, _before_1) <= \at(*G, Here);
        */
        if(i == 0) break;
        _before_2:
        *p = i;
        /*@assert \separated(p, G) ||
            (\at(*G, _before_2) <= \at(*G, Here);
        */
        _before_3:
        bar(p);
        /*@assert \separated(G+(0..40), G) ||
            (\at(*G, _before_3) <= \at(*G, Here);
        */
    }
}
```

The assertions are inserted *after* the selected instructions rather than *before*, because we cannot refer to C labels that occur after the assertion. Hence, all values default to `After` (translated as `Here`) and `Before` is translated as a fresh C label inserted just before the instruction.

Remark 20 (Fragility of HILARE using labels). By default, since contexts select edges *preceding* interesting events, the value of all expressions in a meta-predicate is taken *before* the corresponding events (memory modifications, calls, etc.).

However, the current implementation of labels means that using a single *After* label will implicitly default every unspecified value to *After* as well, and *Before* must be used explicitly to restore the previous semantics.

This is a detail for the `\writing` context where at most one location (or localized set of locations) has different values before and after. However, the problem is more apparent with `\calling`, where many locations may have changed after a function call.

4.3.2 Referring to Bound Names

Section 3.3 also introduced a way to refer to some local variables or heap blocks using a notion of *bindings* with special functions `\bind` and `\bound`. They are used to respectively associate a local value to a global name and use that name in specification to refer to all values bound to that name.

To actually instantiate a HILARE with bindings, the program must be further instrumented using *ghost code*, presented in Section 3.1. As a reminder, ghost variables are declared for specification purposes only and cannot be used by the original C code, while ghost statements may only modify ghost variables. Thus, ghost code altogether cannot modify the original behaviour of the code but may facilitate verification.

For each bound name, we allocate an associated ghost global array whose role is to store the set of associated values. Consequently, each instance of `\bind(v, n)` is replaced by a ghost instruction adding `v` to the array `n_set` associated to `n` and every instance of a predicate $P(n)$ involving a bound name is replaced by a quantified predicate $\forall v \in$ `n_set`$, P(v)$.

This is illustrated in Figure 4.2, which is the translation of Figure 3.8 of Section 3.3. Notice that the type of the array is inferred from the `\bind` calls. As such, it is the responsibility of the user to ensure that every bound value is of the same type and to use the bound name appropriately (and MetAcsl will report an error otherwise).

Notice the two new ghost global variables `cells_set` and `cells_set_size` associated to the name `cells`, and how the initial call to `\bind` has been replaced by a call to ghost function `add_to_array`. This function manages the two variables, reallocating the `cells_set` as necessary and adding new values to it.

```
int lock;
//@ ghost int** cells_set = NULL;
//@ ghost size_t cells_set_size = 0;
int* create_cell() {
    int* c = (int*) malloc(sizeof(int));
    //@ ghost add_to_array(&cells_set, &cells_set_size, c);
    return c;
}
int safe_modify_cell(int* cell, int val) {
    if(!lock) {
        /*@ assert \forall size_t i; i < cells_set_size ==>
        \separated(&lock, cells_set[i]) || lock; */
        lock = 1;
        /*@ assert \forall i; ...
            \separated(cell, cells_set[i]) || lock; */
        *cell = val;
        /*@ assert \forall i; ...
            \separated(&lock, cells_set[i]) || lock; */
        lock = 0;
        return 0;
    }
    else return -1;
}
void unsafe_modify_cell(int* cl, int val) {
    //@ assert \forall i; ... \separated(cl, cells_set[i]) || lock;
    *cl = val;
}
```

Figure 4.2: Translation strategy for bindings on Figure 3.8

Finally, notice how the instantiations of the meta-predicate in `cell_modif_is_critical` are automatically wrapped in a quantification over the array.

> **Remark 21 (About the verification of bindings).** The instantiation of bindings introduces a lot of new code potentially spanning across the whole code base. As such, it is potentially very difficult for analysers to statically determine that a HILARE involving bindings is correct.
>
> One particular obstacle is that the arrays associated by bound names are allocated and re-allocated on the heap, something which

is notably difficult to analyse. To alleviate this problem, the plugin MetAcsl can be configured to use static arrays with fixed size instead. This might help the analysis but of course brings soundness issues.

Overall, this feature is more useful when associated with runtime analysis which actually executes the code (and the ghost code) or pseudo-execution such as symbolic execution or abstract interpretation. It allows specifying quickly a global fact about some particular values and ensure this is not violated during execution.

4.4 Practical Usage

In this section, we will quickly describe how the MetAcsl plugin can be used in practice to assess the validity of a program (partially) specified with HILARE. MetAcsl itself is a Frama-C plugin necessary to parse a source file containing HILARE specification, and able to perform the translation technique described in the beginning of the chapter.

If the plugin is installed (for example with `opam install frama-c-metacsl`, see the installation instructions of the release for more details[1]) along with Frama-C, it will by default parse HILARE parts of source files and remove them, doing nothing.

To enable the translation of these HILAREs, the `-meta` flag must be passed to Frama-C, which will create a new *project*[2] containing the same source file with all necessary code annotations inserted, equivalent to all specified HILAREs.

This translated program can then be processed by other tools, using the `-then-last` flag to chain plugins (see next section for details on analysis).

```
# Print the translation in output.c
frama-c input.c -meta -then-last -print -ocode output.c
# Try to run deductive verification on the translation
frama-c input.c -meta -then-last -wp
```

There are several configurations flags that can be passed to MetAcsl to customize the translation. The full list of options can be displayed using `frama-c -meta-h`.

[1] https://git.frama-c.com/pub/meta
[2] See Frama-C manual.

4.4.1 Reporting

As described in Section 3.1, in Frama-C each annotation embodies a property with a *validity status*, to which plugins can suggest changes. Hence, after using a plugin such as Wp, the project will be in a state where each generated assertion will have its own validity status (hopefully *Valid*).

The MetAcsl plugin consolidates this by generating a global, abstract annotation for each HILARE, that embodies its global status. If at some point the validity status of every annotation tied to a particular HILARE is *Valid*, then MetAcsl determines the HILARE is overall valid as well. This allows users to assess quickly whether a HILARE has been globally proven.

Conversely, sometimes some local assertions cannot be proven, in which case the global HILARE is deemed *Unknown* as well. In this case, MetAcsl makes it easy to identify which HILARE is potentially violated at that particular program point by prepending a different *name* to each assertion, containing the name of the HILARE it is tied to.

This also allows launching analysis from the command line and being quickly able to find the location of an offending assertion with just its unique name by browsing the source code.

4.4.2 Inline Verification Flags

While the MetAcsl flags allow altering the behaviour of the plugin globally, it is also possible to configure some properties on a HILARE basis, using the flagged form of HILARE briefly described in Section 3.2.

The `\flags` directive can be used to modify the behaviour of MetAcsl on a particular HILARE during translation.

Proof method. The `proof` flag determines how the HILARE is expected to be proved. Its default value, `local`, says that the HILARE should be considered valid if and only if every instantiated assertion is valid as well, as described earlier. The flag can also be set to `axiom`, in which case the HILARE is considered valid without verification. Lastly, the `deduce` value attempts to deduce the property from previous HILARE (the deduction mechanism will be described in Chapter 6).

Translation method. The `translate` flag toggles how the HILARE should be translated. The default is `yes`, which means "translate, using the

global parameters"[3]. One can also set this flag to `no` to disable the translation altogether, `check` to force the instantiation of the meta-predicate to use the **`check`** ACSL construct (see Section 3.1.2) or **`assert`** to force the usage of **`assert`**.

Having described the general verification technique for HILARE requirements and their extensions, let us now apply it to the small case study introduced and specified in the previous chapter, through different existing Frama-C plugins. This will serve as a practical example of HILARE verification.

4.5 Assessment of the Page Manager

At the end of the previous chapter, we presented a simple case study on a system called *page manager* (Section 3.5), along with a collection of high-level requirements it must uphold. We then described how to specify these requirements using HILARE (the full source and specification is in Appendix A).

We now want to evaluate the ability of Frama-C + MetAcsl to assess these requirements with the usual Frama-C tools. To that end, we wrote a correct C implementation of the different system functions. Then, to increase the sample size, we used a Frama-C plugin[4] to generate mutations of this correct implementation, providing a set of modified implementations, potentially invalid with respect to the requirements.

In this way, we obtain 126 implementation mutants. The mutations consist in the replacement of binary operators, the negation of conditions and the modification of numerical values. They simulate frequent programming errors in the code.

> **Remark 22.** Some mutants may be logically duplicate: reversing an equality condition and changing the equality to a difference give the same results).

The specification for the mutants remains the same as for the initial implementation.

For each mutant, we *manually* checked if the introduced mutation violates one of the requirements of the case study (i.e. if there is a possible

[3] By default, this means that every predicate is instantiated as an assertion. This is configurable via the global `-meta-asserts` and `-meta-checks` flags.

[4] See https://github.com/gpetiot/Frama-C-Mutation.

Functions	HILARE	Assertions	After simplification	Mutants (invalid/total)
11	11	408	273	42/126

Table 4.1: Statistics about the MetAcsl instrumentation of the page manager

	Wp	E-ACSL
False Positives	0/42	29/42
False Negatives	0/84	0/84
Interrupted (RTE)	N/A	19/126

Table 4.2: Automatic approaches compared to manual verification

input that invalidates a HILARE). If so, the mutant is considered invalid (this is the *ground truth*).

The proportion of invalid mutants is reported in Table 4.1 along with some quantitative information about the system and the instrumentation of the HILARE by MetAcsl. Here we can observe that the simplification phase described in Section 4.2 significantly reduces the number of generated assertions, thus easing the job of the tools that are subsequently run on the resulting translated programs.[5]

For each benchmark (initial version or one of the mutants, including all valid and invalid mutants), we first apply MetAcsl to generate an annotated C program. We then wish to investigate whether, thanks to the instrumentation with MetAcsl, different Frama-C tools are able to assess the validity of the benchmarks with respect to the meta-properties.

We test two existing assessment techniques, namely deductive verification with the Wp plugin and runtime verification with the E-ACSL plugin.

For both plugins, Table 4.2 indicates the number of false positives (cases where the mutant is invalid, but no violation was detected) and false negatives (cases where the mutant is valid, but flagged as violating a meta-property). The last row indicates that there was a runtime error during the execution of the mutant (only applicable for the second technique, since the first is static). We detail both techniques and comment this table in the rest of the section.

[5]For example, simplification saves 8 seconds on the deductive verification of the correct confidentiality implementation (for a total of 24 seconds).

4.5.1 Deductive Verification

We attempt to run Wp on each benchmark. While a proof success is definitive, a proof failure may have different causes: the property to be proved may be false, there could be insufficient assumptions available to the prover or it could simply exceed the capacity of the prover in its allocated time. Thus, if every proof failure is classified as a judgement of invalidity, false negatives are to be expected. To mitigate this phenomenon, we first manually annotated the case studies with partial function contracts for the correct implementations to be successfully proved. See Chapter 5 for a full discussion about troubleshooting a verification failure.

Every valid mutant was successfully proved as valid, and the proof failed for each invalid mutant (see Figure 4.2)[6], thus confirming the correctness of the transformation. These results demonstrate that the spec-to-spec translation with MetAcsl creates a convenient, fully automatic toolchain for deductive verification of global properties in Frama-C. As usual for deductive verification, some additional annotations were necessary to prove the different functions (40 lines of specification were needed, loops being the main point of effort) but their number was much smaller than the number of relevant assertions automatically generated from the HILARE specification.

4.5.2 Runtime Verification on Test Cases

We now wish to study if it is also possible to verify HILAREs at runtime—without any additional annotations—thanks to the E-ACSL [SKV17] plugin for runtime assertion checking. It automatically translates an ACSL-annotated C program into another program that fails at runtime if an annotation is violated.

Since our case study is a library without any particular point of entry, we wrote a small test suite of complete programs that can be actually compiled and executed. They contain simple functional tests and do not aim at covering every possible usage case. They feature a sequence of 40 calls to the library.

We then applied runtime assertion checking to the execution of every instrumented benchmark on all tests of the test suite, in order to detect potential mutation-induced violations of HILAREs at runtime.

The results are laid out in Figure 4.2 and are promising as well. The additional row refers to cases where the generated binary detected a vio-

[6]The last row is not relevant for deductive verification, see next subsection.

lation of a *safety* property [7], thus stopping the execution and preventing us to know if a HILARE violation would have been detected or not. In the future, it would be desirable to filter out safety-violating mutants, and only keep mutations simply modifying the semantics of the code.

There are no false negatives, confirming that the instrumentation of both MetAcsl and E-ACSL does not introduce any bug in the specification nor in the code.

> **Remark 23 (About false positives).** There is a significant number of false positives (incorrect mutants for which no test failed). There are several reasons for this. First, our initial test suites are not complete, and some mutants are not killed by these tests. This could be addressed by using the StaDy [Pet+18] plugin, which combines static and dynamic verification and allows the automatic generation of test cases that can exhibit counter-examples for invalid properties.
>
> The second reason is that E-ACSL only supports a subset of ACSL: some properties involving complex constructions such as the `\at` keyword are simply ignored by E-ACSL, thus they cannot possibly be violated at runtime. This support should be improved in the future.

This section demonstrates that it is easy to check HILAREs at runtime without extra annotation effort thanks to the combination of MetAcsl and E-ACSL, as long as the specified properties are supported by the tools. This is especially useful for properties that are not easily tractable with deductive verification: for example, a property using bindings (see Remark 21) might be very difficult to verify using Wp without writing extensive function contracts, while it can be immediately tested with E-ACSL.

[7] E-ACSL add checks to ensure that no runtime error (segmentation faults, overflow, ...) will occur and stops the program upon violation.

Chapter 5

Proposition of Validation Methodology with HILARE

The previous chapters introduce the concept of meta-properties for representing high-level requirements (Chapter 2), its application to the specification of C programs with the HILARE language (Chapter 3) and an approach for verifying these programs (Chapter 4). While there was a small case study based around a confidentiality-oriented page manager, we've yet to discuss a general methodology for tackling complex, realistic specification problems.

The purpose of this chapter is to present a methodology of specification and verification of a wide range of high-level requirements with HILARE and MetAcsl, and to illustrate it on several examples. The goal is to provide verification practitioners with detailed methodological guidelines for various common patterns of properties in order to facilitate their specification and verification. The provided patterns can be followed by less experienced verification engineers to avoid logical errors in the specification of a HILARE. We also emphasize some good practices showing how to avoid some frequent pitfalls.

The provided examples are inspired by very frequent kinds of properties and illustrated on a security-relevant use case: a microkernel of an operating system (OS), where the description of the system is intentionally left generic.

5.1 General Methodology

When confronted with a verification problem that seemingly involves large parts of a code base, using MetAcsl and its HILARE specification language might be an efficient and expressive way to encode the desired properties with reasonable effort. With experience, the specification and verification process with MetAcsl usually follows a recurring pattern, which might be useful for new users to know about.

Ensuring the problem is within the scope of HILARE

While it is possible to encode various categories of properties with the HILARE language, some of them are outside its scope and could be better addressed by more suitable tools.

Thus, one should check that a desired property:

- *does not relate multiple execution traces*. Indeed, if the specification involves the comparison of multiple execution traces, it is a relational property. MetAcsl is not intended to deal with such properties.[1]
- *is not overly concerned with the order of execution*. While the HILARE language can be used to write simple temporal properties (e.g. relating two consecutive states of the program), complex properties describing the behaviour of a program over time could be better treated with other Frama-C plugins.[2]
- *can somehow be reduced to a property on the global state*. Since the HILARE language is meant to express high-level requirements, properties can only be expressed over data that is visible in the global scope. While there are some facilities to pry into the state of individual functions when necessary, a property that is overly specific to a local state might be hard to verify, or even to specify, as a HILARE. In fact, a plain ACSL annotation would probably be sufficient in this case.

Identifying the working subset of the global state

As mentioned before, the HILARE language can only express properties over data that are visible at global level. Hence, it is necessary to identify where to anchor the properties in the global state. There are several such "anchor points":

[1] One could use e.g. the dedicated Rpp plugin for that purpose.
[2] Such as CaFE or Aoraï.

Global variables. This is the simplest and most desirable way. Global variables can just be referred to by name in any HILARE.

Common parameters. Sometimes data are not global but passed around as a parameter in numerous functions. If the naming of the formal parameter is consistent across those functions, it can be referred to. See `\formal` in Section 3.3.

Heap data. For memory dynamically allocated on the heap, the *binding* extension described in Section 3.3 allows tracking arbitrary pointers across a program but significantly complicates the subsequent proofs.

Once the relevant elements of the state of the program have been identified and made available, the properties themselves can be formulated as HILAREs. The set of variables (or more generally, memory locations) used to specify a property is referred to as its *memory footprint*.

Formulating the problem as HILARE

One should first identify the *target set* of the property: what set of functions should uphold the requirement. It is often helpful to define named function sets using C macros, and to compose them with set operators (see Section 3.2.1). Remember that by default a HILARE is not transitive: it does not apply to the callees of a function unless explicitly specified. Built-in operators can be used to refer to the sets of callees or callers of a function (or over-approximate them in case the code base contains indirect calls).

Reasoning with the HILARE language means reasoning with *conditions* on some *action*: ideally, one should try to express the requirement either as an *invariant* or as a constraint on:

- memory modifications;
- memory (reading) accesses;
- function calls.

In general, properties are easier to express when formulated as a *constraint* on some code operations that must hold under all or most circumstances, except maybe specific ones. This step will be detailed in Section 5.3 through a number of common specification patterns.

Ensuring the Specification is Complete

Once the main desired requirements are expressed, it is important to check that the specification is complete and overall consistent, lest the ensuing proof is useless.

Memory footprint closure. One of the main pitfalls of HILARE specification is forgetting to constrain the modifications of all variables in the footprint of every HILARE. It is important to ensure that such variables cannot be maliciously modified during the execution. Hence, for each location in a HILARE footprint, the practitioner should specify (e.g. as another HILARE) what parts of the code can modify it and under which constraints. This will be illustrated in Section 5.4.

Absence of undefined behaviours. Failing to account for potential undefined behaviours and other runtime errors is another trap to avoid. Indeed, even carefully written global specification (or any specification) will be rendered useless by undetected illegal behaviours, because their absence is one of the main assumptions of the verification tools. For example, the following assertion will happily be considered valid by Wp though it might be false (since pointer `p` to the character `A` is here used to write an integer).

```
char A, B = 1;
int* p = &A; *p = 0; // Undefined behaviour: buffer overflow
//@ assert compiler_dependent: B == 1;
```

Hence, it is important to ensure that the `-wp-rte` option is passed to Frama-C. Thanks to it, Wp will try to prove that every memory access is valid (and report, as expected, a failure in this example).

Assessing the Properties

As described in Chapter 4, MetAcsl is a plugin that translates each HILARE into annotations in the target functions, that can then be assessed by existing Frama-C plugins, such as Eva, E-ACSL or Wp. We will focus here on the usage of Wp, that is, through deductive verification.

When running MetAcsl followed by Wp on a HILARE-specified program, most of the verification conditions are usually easily proved valid. Some guidelines about addressing proof failures are laid out in Section 5.5.

5.2 Presentation of an Illustrative Use Case

In this section, we present an illustrative use case: an imaginary microkernel of an OS dealing with various tasks. For the sake of clarity and simplicity, we intentionally give a very generic simplified description of the system, leaving out the low-level implementation. However, the architecture of the system mirrors that of real ones, and it realistically describes microcontroller targets, in which the CPU only uses physical addresses. In this context, we realistically assume that the numbers of tasks and regions are determined at compilation and do not change. However, it is not a hard limitation, and dynamic changes can also be supported.

The system has a number `NUM_TASKS` of applicative tasks, each one being identified by a task number `taskId` > 0 of type `uchar` (see Figure 5.1). Variable `CurTask` contains the number of the currently executed task. When the (privileged) microkernel services are executed, the variable `Context` is set to `SYSTEM_CTX`, otherwise execution proceeds with ordinary privileges.

The memory is structured in disjoint allocated memory areas that we call regions. The number of regions is given by `NUM_REGIONS`. These memory areas are modeled using two arrays, `RegionStart` and `RegionSize` indicating for each region respectively the pointer to the beginning of the region and its size in bytes. The owner of a region is modeled by the `RegionOwner` array. Thus, region `j` is owned by task `RegionOwner[j]`, starts at address `RegionStart[j]` and contains `RegionSize[j]` bytes. The region is owned by the microkernel if `RegionOwner[j]==SYSTEM_OWNER`, and by an applicative task otherwise. Region `j` is a code region if `RegionKind[j]==CODE_REGION`, and a data region otherwise.

Each task has a priority modeled by `TaskPriority` and a status modeled by `TaskStatus`. A task `i` is ready to be executed if `TaskStatus[i]` is equal to `READY_TSK`, and is waiting or sleeping otherwise. Task `i1` is of a higher priority than task `i2` if `TaskPriority[i1] < TaskPriority[i2]` (notice that the highest priority has the smallest value).

Lastly, the activation of a hardware Supervisor Mode Access Prevention (SMAP) feature is symbolized by a boolean `SMAP_enabled` variable. When enabled, the CPU should prevent the microkernel from accessing task memory at all when in privileged mode.

The next section recalls the HILARE syntax and introduces a bestiary of common specification patterns. Using that in Section 5.4, we will specify some desired properties over that system, such as *task memory isolation* (tasks should not read or write regions different from their own), *controlled privileged operations* (the aforementioned SMAP feature is enforced), *Write XOR execute* (ensure regions are never both writeable and executable at

```
typedef unsigned char uchar;
typedef unsigned int  uint;
// Number of tasks, defined at compilation
#define NUM_TASKS ...
// Number of regions, defined at compilation
#define NUM_REGIONS ...
#define SYSTEM_CTX 0
#define SYSTEM_OWNER 0
#define READY_TSK 0
#define CODE_REGION 0

char  Context;      // System (SYSTEM_CTX) or Task context
uchar CurTask;     // Current task
uint  SMAP_enabled; // 0 disabled, 1 enabled
uchar TaskPriority[NUM_TASKS]; // Priority of task i
uchar TaskStatus  [NUM_TASKS]; // Ready or waiting

char* RegionStart[NUM_REGIONS]; // Start of region i
uint  RegionSize [NUM_REGIONS]; // Size  of region i
uchar RegionOwner[NUM_REGIONS]; // Owner of region i
uchar RegionKind [NUM_REGIONS]; // Code or Data
```

Figure 5.1: Modeling tasks and memory regions in a microkernel

the same time) and *valid task scheduling* (ensure the scheduler respects all constraints of the system).

5.3 Common HILARE Specification Patterns

This section reminds the syntax of HILARE specification along with a set of commonly used patterns to specify usual integrity and confidentiality properties. It is intended to serve as a reference during a specification task and to support Section 5.4 where they are put together to form more complex properties. The process allowing the verification of HILARE specification is described in Section 5.5, and is useful to have a good understanding of the HILARE language.

A HILARE has the form illustrated in Figure 5.2, named *base pattern*. It specifies that Predicate must be valid in the given Context for all functions in Targets. Name denotes an user-defined name for the HILARE. Targets is a set of functions and Context is one of {`\strong_invariant`, `\weak_invariant`, `\precond`, `\postcond`,

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(Context),

Predicate;
```

Figure 5.2: The HILARE base pattern

\writing, \reading, \calling}. The precise semantics of these components are discussed at length in Chapter 3.

Predicate is an ACSL predicate (the reader can refer to the ACSL specification [Bau+20a] for the grammar) and can be a very general property, but is usually some form of validity check of different memory operations (hence manipulating memory locations) or a global invariant.

The following subsections give different possible combinations of Predicate and Context in the base pattern, illustrating how they can work together with the target set to specify interesting properties. Wherever it appears, symbol Location refers to a variable or a range of elements of an array, represented by their addresses.

Each pattern is a template refining the base pattern but still leaving some elements abstract. It is presented with an introductory sentence articulating these elements, followed by the raw syntax and possibly examples and explanations.

5.3.1 Simple Global Requirements

Pattern 1 (Global weak invariant). Name specifies that predicate Prop holds at the beginning and the end of each function in the Targets set.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\weak_invariant),

Prop;
```

Example 25. For example, the following property states that every function needs the variable `logic_state` to be correct (for some unspecified definition of `is_correct`) in the pre-condition and must ensure it is still correct in the post-condition. However, the state may be temporarily incorrect inside the function.

```
meta \prop,
    \name(state_always_valid),
    \targets(\ALL),
    \context(\weak_invariant),

is_correct(logic_state);
```

To disallow even brief violations of the invariant, one should use the `\strong_invariant` context, which ensures the invariant is valid at each *sequence point* of the program, as seen in Pattern 3.

Pattern 2 (Global post-condition). Name specifies that the predicate Prop holds at the end of each function in the Targets set.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\postcond),

Prop;
```

It is similar to Pattern 1 but omits the pre-condition on all functions. Both of them are simple ways to automatically add clauses to numerous function contracts.

Pattern 3 (Global strong invariant). Name specifies that the predicate Prop holds at every step of each function in the Targets set.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\strong_invariant),

Prop;
```

5.3.2 Memory Modification Requirements

Pattern 4 (No memory modification). Name specifies that no function in the Targets set directly writes[a] to Location.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\writing),

\separated(\written, Location);
```

[a] Indirect writes i.e. instructions hidden behind function calls are not considered.

Example 26. For example, the following HILARE means that the `logic_state` global variable can never be written to.

```
meta \prop,
    \name(state_never_changed),
    \targets(\ALL),
    \context(\writing),

\separated(\written, &logic_state);
```

As explained in Chapter 4, in practice when MetAcsl processes such a HILARE for further verification, it iterates through all target functions and write instructions, and adds an assertion of Predicate where `\written` has been replaced by the particular location being written to by the local instruction. The same process is used for all the following variations of this pattern.

Note that Location may still be written to by functions that are not in Targets, but that are *called by* functions in Targets. To automatically include these functions, the `\callees` operator is very useful.

As mentioned above, `\diff` is also very useful here, to indicate that only a fixed set (e.g. for initialization) of functions is allowed to write to some object. For instance, the following HILARE states that `private_key` can only be set in `enc_init`.

```
meta \prop,
    \name(only_init_allowed),
    \targets(\diff(\ALL, {enc_init})),
```

```
    \context(\writing),

\separated(\written, &private_key);
```

Pattern 5 (Conditional memory modification). Name restricts the situations when functions in the Targets set can write to Location. When Guard holds, the writing operation is allowed only if Constraint also holds just before the write operation.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\writing),

Guard &&
\overlaps(\written, Location)
==> Constraint;
```

This is one of the most used patterns. Note that Guard can be omitted to ensure that Constraint holds on *all* write accesses to Location. Recall (from Section 3.1.2) that `\overlaps` is the negation of `\separated`: it specifies that two locations overlap in memory.

Example 27. For example, the following HILARE is similar to the example in Pattern 4 but instead of simply forbidding memory modification, the HILARE allows it only if `has_privilege` is true.

```
meta \prop,
    \name(state_change_requires_privilege),
    \targets(\ALL),
    \context(\writing),

\true && // Can be omitted
\overlaps(\written, &logic_state)
==> has_privilege != 0;
```

Pattern 6 (Precise conditional memory modification). Name restricts the situations when functions in the Targets set can write to the global variable Var. When Guard holds, the writing operation is

allowed only if the relation between its previous and new value is valid according to the predicate Relation.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\writing),

Guard &&
\overlaps(\written, &Var)
==> Relation;
```

Relation is a particular instance of a constraint, that can refer to the value of Var before and after the writing operation using respectively `\at(Var, Before)` and `\at(Var, After)`.

Example 28. For example, the following HILARE only allows assigning increasing values to the global `var`.

```
meta \prop,
    \name(increasing_values),
    \targets(\ALL),
    \context(\writing),

\overlaps(\written, &var)
==> (\at(var, Before) <= \at(var, After));
```

5.3.3 Memory Access Requirements

Pattern 7 (No memory access). Name specifies that no function in the Targets set directly reads from Location.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\reading),

\separated(\read, Location);
```

Similarly to memory modification (Pattern 4), target operators can be very useful to build more complex properties.

Pattern 8 (Conditional memory access). Name restricts the situations when functions in the Targets set can directly read from Location. When Guard holds, the read is allowed only if Constraint also holds at the time of reading.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\reading),

Guard &&
\overlaps(\read, Location)
==> Constraint;
```

This is the dual of Pattern 5, and is used for similar purposes.

5.3.4 Call Graph Requirements

Pattern 9 (No function call). Name specifies that no function in the Targets set may directly call Function.

```
meta \prop,
    \name(Name),
    \targets(Targets),
    \context(\calling),

\tguard(\called != Function);
```

This pattern contains a necessary *guard* since `\called` and Function can have non-compatible prototypes, making the disequality ill-typed. In that case, `\called` is obviously not Function, so a `\tguard` is appropriate.

Pattern 10 (Conditional calling). Name specifies that functions in the Targets can call Function. However, when Guard holds, the call is only allowed if Constraint also holds at the time of calling.

```
meta \prop,
\name(Name),
    \targets(Targets),
    \context(\calling),
```

```
Guard &&
\fguard(\called == Function)
==> Constraint;
```

It is an extension of the previous pattern, similar to how Pattern 4 is extended by Pattern 5.

This time, we protect the equality with `fguard`, which evaluates to `\false` if `\called` and Function have incompatible prototypes, so that there is no need to enforce Constraint on such a call.

5.4 Combining Patterns to Express Complex Properties

This section demonstrates how the previous patterns can be articulated into a larger specification methodology on realistic confidentiality or integrity properties. To that end, we come back to the use case described in Section 5.2 and detail the specification process of some selected properties.

The simplified microkernel described in Section 5.2 is by nature a critical component of its host system, and as such should uphold several confidentiality and integrity properties pertaining to different aspects of its behaviour.

5.4.1 Task Memory Isolation

One such aspect is the strict compartmentalization of tasks: a task should only read from and write to the memory regions it owns. Moreover, it should not access regions containing code in any way. These two requirements can be specified with similar HILAREs using Patterns 5 and 8:

```
meta \prop,
    \name(region_integrity_task),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

\forall integer i; 0 <= i < NUM_REGIONS
&& Context != SYSTEM_CTX
&& \overlaps(\written,
             RegionStart[i] + (0 .. RegionSize[i] - 1))
==> RegionOwner[i] == CurTask && RegionKind[i] != CODE_REGION;
```

The above HILARE iterates on all regions and states that if a memory region is modified (that is, if the location targeted by a writing operation overlaps with a region) while in user land, then (i) the owner of that region should be the task itself, and (ii) it should not be a code region. Here, as the Location in Pattern 5, we use `RegionStart[i] + (0 .. RegionSize[i] - 1)`, which represents the *range* of addresses corresponding to region number `i`.

The complete specification is presented in Figure 5.3, where the second property `region_integrity_system` follows the same principles. Its Guard captures the modification of regions while in system context (when the privileged microkernel services are invoked from a task) and the Constraint forces the owner of the modified regions to be either the original task or the microkernel. Again, modifications of code regions are forbidden.

These two *integrity* properties are then mirrored using Pattern 8 to specify their *confidentiality* counterparts, that is restraining memory accesses instead of modifications. Figure 5.3 shows `region_confidentiality_task`, which is `region_integrity_task`'s confidentiality counterpart.

5.4.2 Controlled Privileged Operations

The SMAP feature mentioned in Section 5.2 should also be enforced: when enabled, code in privileged mode should not have any access whatsoever to task regions. Again, this can be specified with instances of Patterns 5 and 8 with a structure similar to the previous properties. Figure 5.4 illustrates such restriction of memory accesses, both for reading and writing. The Guard filters accesses in privileged mode to regions not owned by the microkernel, which can only happen if SMAP is disabled.

> **Remark 24 (Usage of macros).** Using C macros can improve the readability and portability of specifications by abstracting implementation details to simple predicates. Figure 5.4 demonstrates this. In particular, the `FORALL_REGION` macro hides the underlying array, bounds and indices, which are just noise, requirement-wise.

5.4.3 Write XOR Execute

Our microkernel discriminates memory regions by their kind: either code or data. While the previous properties ensure that code regions cannot be modified or accessed, executing instructions contained in data regions

```
meta \prop,
    \name(region_integrity_task),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

\forall integer i; 0 <= i < NUM_REGIONS
&& Context != SYSTEM_CTX
&& \overlaps(\written,
            RegionStart[i] + (0 .. RegionSize[i] - 1))
==> RegionOwner[i] == CurTask && RegionKind[i] != CODE_REGION;

meta \prop,
    \name(region_integrity_system),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

\forall integer i; 0 <= i < NUM_REGIONS
&& Context == SYSTEM_CTX
&& \overlaps(\written,
            RegionStart[i] + (0 .. RegionSize[i] - 1))
==>
(RegionOwner[i] == CurTask || RegionOwner[i] == SYSTEM_OWNER)
&& RegionKind[i] != CODE_REGION;

meta \prop,
    \name(region_confidentiality_task),
    \targets( \diff( \ALL, init ) ),
    \context(\reading),

\forall integer i; 0 <= i < NUM_REGIONS
&& Context != SYSTEM_CTX
&& \overlaps(\read,
            RegionStart[i] + (0 .. RegionSize[i] - 1))
==> RegionOwner[i] == CurTask && RegionKind[i] != CODE_REGION
```

Figure 5.3: Region integrity and confidentiality

```
#define FORALL_REGION(name, pred) \
    (\forall integer name; 0 <= name < NUM_REGIONS ==> pred)
#define CONTEXT_IS_SYSTEM (Context == SYSTEM_CTX)
#define REGION_OWNED_BY_SYSTEM(r) \
    (RegionOwner[r] == SYSTEM_OWNER)
#define REGION_RANGE(r) \
    (RegionStart[r] + (0 .. RegionSize[r] - 1))
#define SMAP_ENABLED (SMAP_enabled != 0)

/*@
meta \prop,
    \name(micro_kernel_confidentiality),
    \targets( \diff( \ALL, init ) ),
    \context(\reading),

FORALL_REGION(r,
  CONTEXT_IS_SYSTEM
  && ! REGION_OWNED_BY_SYSTEM(r)
  && \overlaps(\read, REGION_RANGE(r))
  ==> ! SMAP_ENABLED
);

meta \prop,
    \name(micro_kernel_integrity),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

FORALL_REGION( r,
  CONTEXT_IS_SYSTEM
  && ! REGION_OWNED_BY_SYSTEM(r)
  && \overlaps(\written, REGION_RANGE(r))
  ==> ! SMAP_ENABLED
);
*/
```

Figure 5.4: Supervisor Mode Access Prevention (SMAP)

must also be prevented. Furthermore, a task should not try to jump to another task's code. Finally, when in privileged mode, only microkernel code should be run. These three requirements can be materialized by the HILARE depicted in Figure 5.5.

```
meta \prop,
    \name(code_execution),
    \targets( \ALL ),
    \context(\calling),

\exists integer i; 0 <= i < NUM_REGIONS
&& RegionStart[i] <= (char*)\called
                   < RegionStart[i] + RegionSize[i]
&& RegionKind[i] == CODE_REGION
&&
((Context != SYSTEM_CTX && RegionOwner[i] == CurTask)
 ||
 (Context == SYSTEM_CTX && RegionOwner[i] == SYSTEM_OWNER)
);
```

Figure 5.5: Code/data exclusion and isolation

Rather than using a specific pattern, `code_execution` is a direct instantiation of Section 5.3's *base pattern*: it is a general validity check of the function call operation, where `\called` is the location of any function that may be called. It states that for every call during the execution, the call should land in a region that: (i) is a code region, (ii) is owned by the current task if executing in user mode, and (iii) is owned by the kernel if we are in privileged mode.

5.4.4 Valid Task Scheduling

The specification of the scheduling behaviour calls for different specification patterns: when switching contexts, the microkernel should not jump to any task that is ready to run, but to a task with the *highest priority*. Hence, whenever the current task changes, the first HILARE in Figure 5.6 puts a constraint on the *new task* instead of the old one. This is a use-case for Pattern 6. `schedule_priority` has no guard (so all modifications of `CurTask` are captured), but states that any new value of `CurTask` must represent a task that is both ready and of the highest priority (compared to other ready tasks). Notice the use of `\at(CurTask,After)` to refer to the new value of `CurTask`.

```
meta \prop,
    \name(schedule_priority),
    \targets( \diff( \ALL, init ) ),
    \context(\writing),

\overlaps(\written, &CurTask) ==>
TaskStatus[\at(CurTask,After)] == READY_TSK &&
( \forall integer j; 0 <= j < NUM_TASKS &&
TaskStatus[j] == READY_TSK ==>
TaskPriority[j] >= TaskPriority[\at(CurTask,After)] );

meta \prop,
    \name(current_always_ready),
    \targets( \diff( \ALL, init) ),
    \context(\strong_invariant),

TaskStatus[CurTask] == READY_TSK;
```

Figure 5.6: Requirement of the scheduler

While this captures the requirements related to context switch, there are others that our system should honour. One of them, `current_always_ready` of Figure 5.6, is that while the scheduler can only switch to ready tasks, the current task shall remain in a ready state for the entirety of its execution. This is an *invariant* that should never be broken, hence Pattern 3 is used to enforce that status.

Tying up loose ends

Although all the previous HILARE taken together form a consistent set that accurately formalizes some requirements for the microkernel, it is of the utmost importance to ensure the absence of loose ends, i.e. to check whether the memory footprint is properly constrained, as mentioned in Section 5.1. For example, properties in Figure 5.3 expect the data in `RegionOwner` to be correct. However, there is no safeguard preventing malicious code from spuriously changing region owners.

To resolve this issue, it is necessary to carefully track any variable in the footprint of every HILARE, and specify when, if ever, it is allowed to change, with the help of Pattern 4. This is done for some of the variables in Figure 5.7: for example, `context_modification` ensures that only a set $\mathcal{K}$

of functions[3] should be able to enable or disable privileged mode, and `region_owners_final` states that region owners should never change.

This should be done for all other relevant variables, such as `SMAP_enabled`, `TaskPriority`, `RegionKind`, etc.

```
meta \prop,
    \name(context_modification),
    \targets( \diff( \ALL, 𝒦 ) ),
    \context(\writing),

\separated(\written, &Context);

meta \prop,
    \name(task_status_modification),
    \targets( \diff( \ALL, {scheduler, init} ) ),
    \context(\writing),

\separated(\written, TaskStatus + (0 .. NUM_TASKS - 1));

meta \prop,
    \name(region_owners_final),
    \targets( \ALL ),
    \context(\writing),

\separated(\written,
          RegionOwner + (0 .. NUM_REGIONS - 1));
```

Figure 5.7: Selected footprint modification constraints

5.5 Verification Discussion

As explained in Chapter 4, the usual verification process of HILARE is to run MetAcsl, which transforms a HILARE into a set of code annotations, and then run Wp, the deductive verification plugin of Frama-C, which tries to prove the various annotations generated by MetAcsl.

Several guidelines and pitfalls were already presented in Section 5.1. We discuss here some additional ones regarding verification.

[3]Where $\mathcal{K}$ is defined as all kernel entry points such as system calls, exceptions and interrupts.

Proof failure analysis. If some annotations are not proved, this can be due to several reasons:

(i) The program is incorrect with respect to the HILARE: the code (or the HILARE) needs to be fixed. Given the name of the failing annotation, it is easy to trace back to the guilty HILARE.

(ii) There are insufficient pre-conditions on the function: the given context is not sufficient for the property to be valid. One should add pre-conditions needed for the proof.

(iii) The prover is not powerful enough: one should manually subdivide the proof into a few intermediate steps by writing annotations that are easier to prove and can help to deduce the required one.

(iv) Some functions that are called are not specified enough. It is common that, while a function clearly does not modify the state related to a property, this is not reflected in its specification and the prover cannot deduce that, after the call, the memory footprint of the property is unchanged. A first step consists thus in manually specifying the memory footprint of the callees. If this is not sufficient, one should specify the conditions needed as a conditional invariant on all potentially called functions using a HILARE.

Avoid overloading the proof context. When dealing with large functions, it may be useful to use the `-meta-checks` described in the previous chapter to instantiate a HILARE as a set of *checks* instead of assertions. The difference is subtle: a check simply tries to prove a predicate at a given point, while an assertion additionally adds it to the context for further proofs. Hence, at the end of a long function, provers might struggle with an overloaded proof context containing all previous assertions of the function.

High-level lemmas. When faced with problem (iii), a classic solution is to provide lemmas used to cut the goal. While it is possible to write lemmas in ACSL, this solution may not work for a HILARE as it can rely on specific local properties in the target functions or on variables non-available globally (e.g. a function parameter). An efficient solution is to write a HILARE used as a *lemma* to prove another HILARE or an ACSL annotation: it is just as efficient as using a regular lemma in a normal context. It can be a statement that follows from the hypothesis and directly implying the goal, or simply an additional hypothesis needed to prove the goal.

Example 29. Let us say we want to prove the following HILARE on a code base with hundreds of functions.

```
char array[1000];
int index = 0;

/* Hundreds of functions ... */

/*@ meta \prop,
        \name(end_of_array_untouched),
        \targets(\ALL),
        \context(\writing),

    \separated(\written, array + (900 .. 999));
*/
```

In other words, we want to prove that the last 100 cells of the global `array` are never modified. Let us assume that the code base only ever accesses `array` by indexing it with the global `index`. As is, even if `index` always stays below 900, the HILARE may not be verified statically.

However, we can translate our assumption into a HILARE as an invariant to help the proof:

```
meta \prop,
    \name(index_below_900),
    \targets(\ALL),
    \context(\strong_invariant),

index < 900;
```

This will be translated by MetAcsl as a set of local assertions that will be useful to prove the assertions generated by `end_of_array_untouched`. Note that in this case, our lemma must *not* be translated into checks, lest the generated assertions cannot be used as hypotheses.

The next chapter will discuss other ways to exploit already established high-level requirements to prove new ones on a code base, without having to resort to generation of local assertions.

Chapter 6

Towards an Automated Framework for Deducing of Meta-Properties

The verification technique for the HILARE language presented in Chapter 4 is fundamentally *local*: it transforms a global requirement into a set of local properties in the source code. As discussed, this set of properties is sometimes quite large despite efforts to discard trivially true or false statements early. While this approach is not always perfect, it cannot be avoided for proving global requirements about a code base without further global information: the low-level code *must* be observed in some way.

However, when some global requirements have *already* been established on a code base (for example by specifying and proving HILARE statements the usual way), it feels natural to attempt *deducing* other global facts without having to resort to local analysis.

This chapter presents a motivation for deducing high-level requirements from others through some specific use cases, and describes an extensible framework for doing so in a provably sound manner. While the next chapter presents a large case study involving the concrete usage of this deduction technique, this chapter does not have the ambition to exhaustively explore the question of deducing meta-properties but rather establish a limited (but extensible) foundation for doing so.

Section 6.1 motivates the existence of a deduction framework by presenting several situations for which we want to make high-level deductions but currently cannot, and generalizes these situations to potential deduction patterns. Section 6.2 then discusses our methodology for designing an efficient and sound deduction framework able to handle such patterns and

presents the global architecture of our technical solution, while Sections 6.3 through 6.5 each describe a component of that architecture. Finally, Section 6.6 discusses the usability of this framework from the point of view of a regular user, and its extensibility by a knowledgeable user.

6.1 Motivation

When specifying and verifying properties in MetAcsl, there are some situations where a property that should be easy to verify given the already established HILAREs becomes tedious due to the limitations of the way MetAcsl works, that is by generating low-level annotations.

We motivate the development of a deduction framework with three simple illustrative use cases (inspired from a real system, see Chapter 7) that ought to be easy to prove and can be articulated into more complex properties. They are presented in the following Sections 6.1.1, 6.1.2 and 6.1.3 respectively. Each use case presents a concrete situation where we want to make a high-level deduction (and currently cannot), and generalizes the situation to a deduction pattern that we would like our deduction framework to handle.

While these use cases will give us an idea of what we want to *be able* to deduce, we also want to put the emphasis on performance. In line with our objective of easing the usage of formal methods discussed in Chapter 1, we want a deduction method to be as seamless as possible: at the price of completeness, we want a fully automated technique. Furthermore, it should easily scale to very large programs with hundreds of functions. In Chapter 7, we will evaluate these claims by testing the approach presented in this chapter on a large code base.

6.1.1 Use Case: Negative Memory Footprint

Consider the following program excerpt, defining two functions `f0` and `f1` (the body of `f1` is not shown):

```
int A, B;

void f1(); // A function which does not modify A

/* Potentially many other functions not modifying A */

void f0() {
    A = 42;
```

```
    f1();
    //@ assert GOAL: A == 42;
}
```

There are two global variables `A` and `B`. Suppose that we want to prove the `GOAL` assertion in `f0`, stating that after the call to `f1`, the variable `A` retains the same value.

With a HILARE, it is easy to specify that function `f1` and its callees do not write to variable `A` at all, with the `\writing` context:

```
meta \prop,
    \name(A_untouched_by_f1),
    \targets(\callees(f1)),
    \context(\writing),

\separated(\written, &A);
```

No matter the content of `f1` and its callees, it can also be assumed that the proof of `A_untouched_by_f1` is relatively easy given that `A` is indeed not modified in any of these functions. In particular, in general it is not necessary to write loop invariants to prove such a HILARE with this simple pattern (Pattern 4 in Chapter 5).

However, specifying and proving that HILARE does not immediately help us prove the `GOAL` assertion in `f0`. While one can easily mentally deduce that if a function and all its callees do not modify a memory location then its value does not change after calling it, the HILARE does not provide the local information necessary to automatically prove the assertion with deductive verification.

> **Remark 25 (Local deduction).** In many cases, the specification of a global HILARE *can* help prove local assertions. Indeed, with the method presented in Chapter 4, a HILARE is translated into a set of inline assertions to be proved by other means.
>
> When using deductive verification (and in particular the Wp plugin of Frama-C), an assertion in the code can be used to prove further assertions since its statement is stored in the proof context and assumed true. If another subsequent assertion is proved this way, it will be considered as true, with the explicit hypothesis that the first assertion is also true.
>
> Hence, a generated assertion of a HILARE can help prove other local properties, including assertions generated from other HILAREs.

> This means that in some cases, a HILARE can already be entirely deduced from another if all of their local assertions follow that pattern.
>
> This is not the case here.

In our current setting, we have two approaches for solving this problem, but they are flawed:

1. **Specify the whole memory footprint of `f1` with the `assigns` construct.** If `A` is not part of the footprint, then the `GOAL` assertion will be successfully proved by tools such as Wp. However, this is a huge undertaking: not only the whole memory footprint of `f1` must be identified, but also the footprint of all of its callees:

   ```
   int A, B, C, ...;

   /*@ assigns B; */
   void f2() { B = 1; }

   /*@ assigns C, ...; */
   void f3() { C = B; ... }

   // Same for all callees of f1

   /*@ assigns B, C, ...; */
   void f1() { f2(); f3(); ...}
   ```

   All of these footprints must be manually annotated and may be hard to prove: all loops need to be annotated, and the proof must care for all other memory locations of the footprint, whereas we are only interested in `A`.

2. **Specify that `A` keeps its value as part of `f1`'s contract.** This can be done by adding `ensures A == \old(A)` in the contract. However, for this statement to be proved, it again needs to be added to the contract of every callee. While the proof may be lighter than the previous approach, it is again potentially challenging, requiring loop annotations. Exotic code such as inline assembly may also be an important obstacle.

Both of the possible approaches look like important proof undertaking for what *should* be immediate to deduce. Hence, the first requirement of the deduction system is being able to deduce the following HILARE from the premise `A_untouched_by_f1` stated above.

```
meta \prop,
    \name(A_unchanged_by_f1),
    \targets(\callees(f1)),
    \context(\postcond),

\at(A, Pre) == \at(A, Post);
```

The local translation of this HILARE into function contracts, while not used to prove the HILARE, could then be used as hypotheses to prove the initial `GOAL` assertion.

Deduction pattern. **If** a variable v is not modified by a function f nor any of its callees, **then** the value of v is the same before and after any call of f.

6.1.2 Use Case: Propagating an Invariant to "Neutral" Functions

We now want to extend the above use case from simple variables to whole predicates. Consider the following program excerpt, containing again two global variables `A` and `B`, and a potentially large number of functions.

```
int A, B;

void f1() {
    if(A != B) { /* Error */ abort(); }
    A *= 2;
    B *= 2;
}

void f2(); // Function modifying neither A nor B
/* Many functions with the same property ... */
void fN(); // Idem
```

The `f1` function simply multiplies `A` and `B` by 2. It is trivially provable that predicate `A == B` is a *conditional* invariant of this function (if `A` and `B` have the same value before `f1`, it remains true afterwards). This invariant can be expressed with a HILARE (even though it only has one target function):

```
meta \prop,
    \name(A_B_same_in_f1),
```

```
    \targets( {f1} ),
    \context(\postcond),

\at(A, Pre) == \at(B, Pre) ==> \at(A, Post) == \at(B, Post);
```

Remark 26 (Weak and conditional invariants). Note that the above property is weaker than what we could specify with the `\weak_invariant` context: whereas this context specifies that a predicate *must* hold when the function is called (it is a pre-condition), here we only say that it *may* hold (and if it does, then it *must* also hold afterwards).

We define a new context named `\conditional_invariant` for that purpose. With it, the same property can be stated as follows:

```
meta \prop,
    \name(A_B_same_in_f1),
    \targets( {f1} ),
    \context(\conditional_invariant),

A == B;
```

Let us assume that the following HILARE is established, meaning that functions `f2` through `fN` do not (locally) modify variables `A` and `B`:

```
meta \prop,
    \name(A_B_untouched),
    \targets( \diff(\ALL, {f1}) ),
    \context(\writing),

\separated(\written, &A) && \separated(\written, &B);
```

Since functions `f2` through `fN` do not locally modify the two global variables, we would like to be able to deduce that `A == B` is also a conditional invariant of these functions, i.e. extend `A_B_same_in_f1` to all functions. This deduction is slightly more subtle than the previous one: the set of target functions must not call functions that may modify the variables.

In general, it is important to be able to deduce that if a predicate over a part of the state is a precondition of a function and that function does not modify this part of the state, then the predicate is also a postcondition.

Deduction pattern. **If** a function f does not modify the variables used in a predicate P and f only calls functions with the same property, **then** P is a conditional invariant of f.

6.1.3 Use Case: Propagating a Precondition to Callees

While the previous use case is quite simple (similar to the first use case), we also want to extend this use case to actual weak invariants. In other words, we want to deduce that a property is a precondition (and a postcondition) of the functions. Let us extend the previous program with the following excerpt:

```
/*@ requires A == B; */
void wrapper() {
    f2();
    f3();
    ...
    fN();
}

int main() {
    A = 1; B = 1;
    wrapper();
}
```

We can easily prove the precondition of the `wrapper` function since it is only called by `main`, which ensures that `A == B` before the call. This simple precondition can also be expressed with a HILARE:

```
meta \prop,
    \name(A_equal_B_before_wrapper),
    \targets( {wrapper} ),
    \context(\precond),

A == B;
```

Since we know that functions `f2` through `fN` do not modify `A` and `B`, we should be able to deduce that `A == B` is also a precondition for them (hence a weak invariant).

Deduction pattern. **If** a function f is only ever called by functions for which a predicate P is a precondition and that do not modify the variables used by P, **then** P is also a precondition of f.

6.2 Methodology and Architecture of the Deduction Framework

For the three use cases presented in the previous section, it can become useful to reason at higher level and allow the deduction of a HILARE from others. We want to construct a framework where the end user can just add the line `\flags(proof:deduce)` to a HILARE to try to automatically deduce it when MetAcsl is launched, rather than translating the HILARE to local assertions and delegating the task to other tools. To meet that objective of performance, the methodology we adopt is pragmatic: we want to be able to deduce simple things automatically and efficiently, while having a formal guarantee that the deduction is sound. Furthermore, we want to design the framework so that it is extensible: it should be possible to add new, potentially more complex deduction patterns later.

Our approach is threefold:

1. establish a mechanized formal model of the HILARE language using the Why3 [FP13] platform and prove that the deduction patterns highlighted in the previous section are valid within that model;

2. translate those deduction patterns into an actionable and efficient Prolog deduction engine that can apply them;

3. extend MetAcsl to automatically translate a HILARE-specified program to constraints (also called knowledge base) for the Prolog engine, which can then assess the validity of some properties using the previously established deduction patterns.

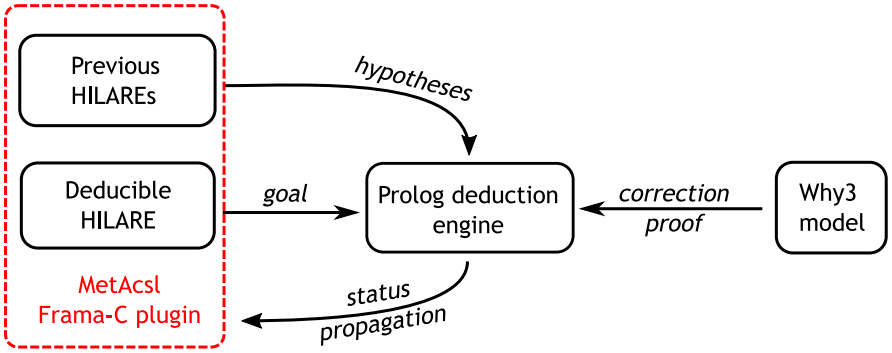

Figure 6.1: Illustration of the deduction methodology

This approach is illustrated in Figure 6.1. During the verification of a program annotated with HILARE where one of them has the `deduce` flag, the following happens:

1. both the HILARE to be deduced and the previous other HILAREs are translated into data that is understood by the external deduction engine (in Prolog);

2. the Prolog engine attempts to prove the desired HILARE based on the premise that the previous HILARE are valid, and on a set of high-level deduction patterns (such as the ones presented previously). It is guaranteed to terminate but is not complete: it handles only some specific HILARE structures and deduction patterns;

3. the deduction engine reports its result to MetAcsl, which propagates the status to the HILARE in Frama-C. If the translation to local assertions of that HILARE is enabled (with the `translation:yes` flag), then all the local assertions are considered valid if the deduction succeeded.

The Why3 component of the framework has no impact at "runtime" (during the proof of concrete programs) but is here to formally ensure that Step 2 is sound. In that way, each part of the framework can be trusted. Its weaker link is the interface between the Why3 model and the Prolog deduction engine, which is done manually: one must ensure that the deduction patterns encoded in the Prolog engine are exactly the ones proved within the Why3 model.

A proof of the deduction approach can only be made within a formal system describing the HILARE language and ACSL themselves. While this can be formalized on paper (as it is done in Chapter 2), it is preferable to be able to machine-check the proof if this approach is to be trusted. This could have been done entirely in ACSL, however the Why3 platform [FP13] and its WhyML language are particularly appropriate since one can express a pure formal system and theorems easily, without having to deal with C specificities. The Coq proof assistant [Coq21] was also a good candidate but the fact that Why3 can rely on external solvers to (try to) automatically prove theorems was a clear advantage, and we did not need the full expressiveness of Coq.

For the actual automated solving during the proof, Why3 could also have been leveraged to try to automatically apply the proved deduction theorems to facts translated by MetAcsl from HILARE annotations. However, we found that the performance of this solution did not scale when the

number of functions increased, hence the need for a dedicated deduction engine. Prolog was a good candidate to easily prototype an engine that can exhaustively explore the space of possibilities.

The next sections describe each part of the architecture in detail: a sketch of the soundness proof, the construction of the actual deduction engine and its communication with MetAcsl.

6.3 Using Why3 for Modelling Structural Deduction Patterns

To prove that the deduction patterns highlighted in Section 6.1 are sound using the Why3 platform, we first needed to transpose the formalization of meta-properties presented in Chapter 2 to that platform, before attempting to prove theorems within that formal system. We briefly describe that transposition, which follows the same structure as Chapter 2: formalizing the structure of the abstract programming language, its semantics, predicates over such programs, and then meta-properties.

```
type location

type function_name

type instruction =
    | Read location
    | Write location
    | Call function_name

type statement =
    | Instr instruction
    | Branch (statement, statement)
    | Loop statement
    | Block block

with block = L.list statement

type func = { def: block }

type program = M.fmap function_name func
```

Figure 6.2: Structural definition of a program in Why3

A small excerpt of the model can be seen in Figure 6.2. As expected, it closely matches the constructs defined in Chapter 2 but gives concrete, actionable types to each component of the model: starting from the bottom, a program is a map between names and functions, and each function is a block of statements that loosely matches C and ACSL semantics: a statement is either an instruction (read, write, or call) or a structural operator (conditional loop, nested block). Notice that it is sufficient for our purposes to reduce C programs to single reads from (resp. writes to) locations, as instructions performing multiple memory operations can be transformed to a set of instructions performing atomic operations.

> **Remark 27 (Recursivity).** Whereas loops are taken in account into the model, recursion is not allowed and its absence is a hypothesis of the following proofs. This is made to ease subsequent proofs.
>
> However, this hypothesis is perfectly sensible: should programs contain recursive calls, they can be transformed – at least in theory – into their iterative equivalent, for example by explicitly simulating the program stack [Zei].

Based on this structure, we define basic semantics of the different language constructs by describing their action over the state of a program: we define their footprint. Figure 6.3 illustrates the semantics of instructions (statements are omitted).

In particular, we define an inductive relation

```
state_instruction(p, i, s1, s2)
```

defining the constraints between the states preceding (`s1`) and following (`s2`) the execution of an instruction `i` on a given program `p`. As expected, reading does not modify the state (`s1` equals `s2`) but writing does (`s1` and `s2` must be equal on every location except the one written to during the instruction).

On that basis, still following the structure of Chapter 2, we formalize the concept of program property (assertion) and the concept of meta-property.

Within that model, we then formulate the three deduction patterns of Section 6.1 as theorems. The first theorem captures the use case of Section 6.1.1: negative assigns. It states that if a function and each of its callees do not modify a location l, then the value of that location must be the same before and after the execution of this function.

```
type value (* An abstract value type *)
(* The state maps locations to values *)
type state = M.fmap location value

(* Two states have the same value for a location *)
predicate same_state_on_location (s1 s2: state) (l: location) =
    forall v: value. M.mapsto l v s1 <-> M.mapsto l v s2

(* Two states have the same value for every location except one *)
predicate state_modification (l: location) (s s': state) =
    forall l'.  (l <> l') -> same_state_on_location s s' l'

(* Relation between the state before and after an instruction *)
inductive state_instruction program instruction state state =
| State_Read : (* no modification of state *)
    forall s l prog.
    state_instruction prog (Read l) s s
| State_Write : (* states differ on l *)
    forall l s s' prog.
    state_modification l s s' ->
    state_instruction prog (Write l) s s'
| State_Call : (* execute the function's block *)
    forall prog, f: function_name, s s': state, b: block.
    valid_program prog ->
    function_definition prog f b ->
    state_block prog b s s' ->
    state_instruction prog (Call f) s s'
```

Figure 6.3: Semantics of programs in Why3

Before stating the theorem itself, we need to formalize the notion of callees. We consider here again the A-LANG language introduced in Chapter 2 and a well-formed program in A-LANG (cf. Remark 4).

Definition 24 (Set of callees). Let $f \in F$ be a function. We define $\mathrm{callees}(f)$, the *set of callees* of f, as the smallest set of functions such that $f \in \mathrm{callees}(f)$ and

$$\forall g \in \mathrm{callees}(f), h \in F, v \in V_g, \quad \mathrm{instr}(v) = \textsc{Call}\ h \implies h \in \mathrm{callees}(f)$$

In other words, $\mathrm{callees}(f)$ is the smallest set that contains f and any

function called by its members. With this definition, we can now state our theorem.

Theorem 1 (Negative assigns). Let $f \in F$ be a function and $l \in L$ a location. We state that if

$$\forall g \in \text{callees}(f), \quad \forall v \in V_g, \quad l \notin W(v),$$

then for all states $\sigma_1, \sigma_2 \in \Sigma$ such that $\langle f, \sigma_1 \rangle \Downarrow^O \sigma_2$, we have that

$$\sigma_1(l) = \sigma_2(l).$$

That theorem is formally proved within Why3. Section 6.3.1 below gives a sketch of proof for this theorem as well as the two following ones.

Remark 28. In Why3, we write the statement of Theorem 1 as:

```
lemma negative_assigns:
    forall prog: program, l: location, f: block, calls: S.fset block,
    pre post: state, v: value.
    valid_program prog ->
    (* 'calls' is the set of f callees *)
    callees prog f calls ->
    (* f does not modify locally l *)
    untouched f l ->
    (* same for all callees *)
    (forall f'. S.mem f' calls -> untouched f' l) ->
    (* the value of l after and before f *)
    state_block prog f pre post ->
    (* is the same *)
    (M.mapsto l v pre <-> M.mapsto l v post)
```

For a given function f and location l, if it has been proved by a HILARE that for f and all of its callees, there is no local modification of l, then it can be deduced that the value associated to l does not change before and after a call of f.

Let us now formalize the second deduction pattern, which states that if a function f and its callees do not modify the set of variables over which a predicate P is defined, then that predicate is a conditional invariant of f, i.e. if P holds in the pre-state of f, it must also hold at the post-state.

Theorem 2 (Conditional invariants of neutral functions). Let $L \subseteq \mathcal{L}_F$ be a finite set of memory locations and $P \in \mathcal{D}(L)$ a predicate over L. We state that if

$$\forall g \in \text{callees}(f), \quad \forall v \in V_g, \quad W(v) \cap L = \emptyset$$

then for all states $\sigma_1, \sigma_2 \in \Sigma$ such that $\langle f, \sigma_1 \rangle \Big\Downarrow^{O} \sigma_2$, we have that

$$\sigma_1 \vDash P \implies \sigma_2 \vDash P$$

This theorem, which captures the second use case of Section 6.1, is also proved in Why3 within our metamodel.

Lastly, let us state the third deduction pattern. It states that if a function f is only called by functions for which a predicate P is a precondition and that do not modify the locations over which P is defined, then P must be a precondition for f as well. To state that theorem, it is easier to first define the notion of precondition.

Definition 25 (Precondition of a function). Let $f \in F$ be a function, L a set of locations and $P \in \mathcal{D}(L)$ a predicate over L. We say that P is a *precondition* of f and write $\text{precond}(f, P)$ if:

$$\forall g \in F, (v_1, v_2) \in E_g, \quad \text{instr}(v_2) = \textsc{Call}\ f \implies (v_1, v_2) \vDash_g P$$

In other words, P is a precondition of f if at every call site of f, P must hold for every possible execution of the caller (see Definition 18 in Chapter 2).

With this definition, we can now state our final theorem.

Theorem 3 (Propagation of preconditions). Let $f \in F$ be a function, $L \subseteq \mathcal{L}_F$ a finite set of memory locations and $P \in \mathcal{D}(L)$ a predicate over L. We state that if

$$\begin{aligned} &\forall g \in F, (\exists v \in V_g, \text{instr}(v) = \textsc{Call}\ f) \implies \\ &\qquad \text{precond}(g, P) \wedge (\forall h \in \text{callees}(g), \forall v \in V_h, W(v) \cap L = \emptyset) \end{aligned}$$

then $\text{precond}(f, P)$.

In other words, if for every caller g of f, P is a precondition of g and g *and its callees* do not modify the variables over which P is defined, then P

is a precondition of f as well.

It is important to take into account the callees of g (that is, including f), because between the beginning of g (where P is true) and the call to f, there may be calls to other functions for which P is not a postcondition.

With this, we have generalized and formalized every use case from the first section, which is a contribution in and of itself. The next subsection gives an idea of how these three theorems were proved, while Section 6.4 describes how these deduction theorems are actually applied in our framework to deduce HILAREs from others.

6.3.1 Details of the Proof Process

First, we have to notice that Theorems 1 through 3 are stated for any program O. To help us prove these theorems, we would like to exhibit an induction principle on programs. To that end, we first define the notion of *call graph* of a program, which will be conveniently structured for subsequent proofs.

Definition 26 (Call graph). The *call graph* of a program is a directed graph (V, E) such that:

- The set of nodes is the set of functions in the program: $V = F$;
- There is an edge from f to g if and only if f **is called** by g:

$$\forall f, g \in F, \exists v \in V_g, \mathrm{instr}(v) = \textsc{Call}\ f \Leftrightarrow (f, g) \in E.$$

The no-recursion assumption discussed in Remark 27 means that the call graph of any program is a *directed acyclic graph*, that is, there is no direct or indirect call from a function to itself.

This implies the existence of a *topological ordering* [HLC09] $\leq_T$ of the vertices of the call graphs (i.e. the functions of the program). The topological ordering is a total order that respects the following property: for all functions $f, g \in F$, $f \leq_T g$ if f is (directly or indirectly) called by g. In particular, every (direct or indirect) callee of a function comes before that function in the order:

$$\forall f \in F, \forall g \in \mathrm{callees}(f), \quad g \leq_T f.$$

Notice that for two functions f and g, $f \leq_T g$ does not necessarily mean that the former is a callee or the latter. Indeed the call graph can be

a forest: there can exist functions f, g such that neither f is a callee of g nor g is a callee of f.

Since $\leq_T$ is a *well-founded* order, we can deduce a *Noetherian induction principle* over functions of a call graph, allowing us to easily prove that a property holds on a whole program.

> **Theorem 4 (Induction principle on functions of a call graph).** Let $\mathcal{P}$ be a property of functions. Assume that for any function $g \in F$, we can deduce $\mathcal{P}(g)$ from the following premise
>
> $$\forall h \in F, h \leq_T g \implies \mathcal{P}(h).$$
>
> Then $\mathcal{P}(f)$ is true for all $f \in F$.

This means that we can prove functional properties of our metamodel by choosing a particular function g, and proving that the property holds on g, assuming that the property already holds for every callee h of g (which is a weaker premise than the one stated in the principle). In particular, it means that the property must be immediately true of a function without callees.

Given that induction principle, small lemmas about what can be deduced within the model are incrementally built from the ground up, naturally leading to the proof of Theorems 1 and 2.

Theorem 1, sketch of proof. We reason by induction over functions of a call graph. For leaf functions, since we know that none of their instructions modify the location l, we can apply the semantics of execution to prove that the states before and after their execution must agree on the value of l.

For other functions, we do the same but use the induction hypothesis whenever a call appears. Indeed, we can deduce that the execution of callees does not change the value of l since our no-modification assumption subsumes the necessary condition to apply the induction hypothesis.

Theorem 2, sketch of proof. We use a similar proof structure, applying execution semantics on leaf functions to deduce that they must not change the variables over which our predicate is defined. We then deduce that the predicate must hold at the end of the function if it was already the case at the beginning. We do the same for other functions, simply applying the induction hypothesis for functions calls.

Theorem 3, sketch of proof. For this theorem, we must use a symmetrical variation of the induction principle exhibited in Theorem 4, where the topological ordering is *reversed* i.e. $f \leq_{T'} g$ if f *calls* g. Since that order is still a well-founded order, we can deduce a new induction principle where we can assume a property is true for every *caller* of a function to prove it on that function.

Using that reversed induction principle, we reason using the same proof structures as for the two previous theorems. For functions without callers, any predicate is immediately a precondition (according to Definition 25) and for other functions it is enough to simply apply the induction hypothesis and the semantics of execution.

Most of the proofs are fully automatic, thanks to the SMT solvers used by Why3. Out of the 41 proofs, 7 had to be written manually using the Coq proof assistant. The whole model and the accompanying proof is available inside the MetAcsl release. [1]

6.4 Using Prolog for Applying the Deduction Patterns

While the Why3 model in the previous section is here to formally prove that deductions made with the deduction patterns of Section 6.1 are sound, the Prolog deduction engine is here to actually use these patterns to make the deductions.

Example 30 (A bit of Prolog). A Prolog program essentially is a list of *facts* and *rules*. The following code contains two facts stating that predicate `likes` is always true for the combination of parameters `me` and `pizza`, and that `is_junk_food` is true for parameter `burger`.

```
likes(me, pizza).
is_junk_food(burger).
```

One can then state rules for deducing new facts from existing ones:

```
% If FOOD is junk food, then 'me' likes it
likes(me, FOOD) :- is_junk_food(FOOD).
```

[1] https://git.frama-c.com/pub/meta

A rule can have multiple conditions, in which case they are separated by commas.

The engine can then be *queried* to check if a given fact can be deduced from the known ones. For example, the query `likes(me, burger)` will yield `true`.

Furthermore, we can pass *unbound* variables (_) in a query to check if there exists an instantiation of it for which the fact can be deduced. For example, the query `likes(me, _)` will yield `true` since the anonymous variable can be replaced by `burger` to make the predicate hold.

The rationale for choosing Prolog to write the deduction engine is twofold:

- Prolog is a natural choice for the design and the usage of small CLP (Constraint Logic Programming [JM94]) languages, such as {log}. Hence, transposing the wording of the Why3 lemma is easy.
- If built correctly, an ad-hoc deduction engine written in Prolog is highly efficient. An early attempt was to directly use Why3 as a deduction engine, but the highly general nature of this platform made deduction on our large yet simple samples frustratingly inefficient.

The solver is structured into three layers: the ground, intermediate, and high-level layers.

Ground layer. The deduction engine expects a set of *facts* about the program provided by the user: the list of functions, the call graph and a list of HILAREs that are already proved. The next section describes how the translation from a C program to Prolog facts for this layer is operated. It mainly contains facts of the form `meta_ground(C, P, S)` where `C` is the representation of a HILARE context, `P` its predicate and `S` its target set. A fact of this form states that the HILARE with these components is already established in the program. By nature, these facts are not hard-coded into the solver but generated at runtime from a concrete C program.

Intermediate layer. Based on these known facts, there is a first layer in the engine that lay down very basic deduction *rules*, which we call the *intermediate layer*. It can essentially infer that a HILARE is valid if a stronger form of it is already established, or if it is simply the combination of multiple identical properties with different target sets.

```
% meta_inter(?Context, ?Predicate, ?Set)
% Means that a HILARE with these components
% can be deduced by the intermediate layer

% If HILARE already proved, it is immediately deduced
meta_inter(C, P, S) :- meta_ground(C, P, S).

% Weak invariant implies postcondition
meta_inter("Postcond", P, S) :-
    meta_inter("Weak invariant", P, S).

% Weak invariant implies precondition
meta_inter("Precond", P, S) :-
    meta_inter("Weak invariant", P, S).

% Strong invariant implies weak invariant
meta_inter("Weak invariant", P, S) :-
    meta_inter("Strong invariant", P, S).

% HILARE is valid if the same HILARE
% is valid for a larger set of functions
meta_inter(C, P, S) :- subsumes(C, P, S, _).

% HILARE is valid if its set of targets can be partitioned
% into two sets for which the property is already proved valid
meta_inter(C, P, S) :- merge_props(C, P, S, _, _).
```

Figure 6.4: The intermediate layer of deduction

This layer is illustrated in Figure 6.4. Each rule encodes a different trivial deduction pattern, commented in the code. This layer makes use of a predicate `subsumes(+C, +P, +S1, -S2)` which checks if the same HILARE with a larger set (`S2`) of functions is known to be valid, and a predicate `merge_props(+C, +P, +S, -S1, -S2)` which checks if the target set `S` of a HILARE can be partitioned into two sets `S1` and `S2` for which the HILARE is known to be valid. Their definition is omitted from the listing.

> **Remark 29 (Set operations in Prolog).** When the programs are large, the target sets have a considerable size: for the solver to remain efficient and to easily manipulate sets, we use a Prolog[a] extension

called {log} [Dov+96] which provides easy syntax and efficient deduction facilities for finite sets.

This slightly changes the usual Prolog syntax. Notably, premises of rules are separated by ampersands (&) instead of commas.

[a] More specifically, SWI-Prolog

High-level layer. On top of this simple layer, the engine defines rules for making less trivial high-level deductions based on the theorems established in the previous section. A small excerpt of this layer is provided in Figure 6.5. In particular, the first deduction lemma is transposed.

It states that a negative assign of location `L` for the target set `S` (that is, the HILARE stating that the value of `L` does not change after the execution of each function in `S`) is considered valid if it is known (through a previous HILARE) that these functions do not modify `L` locally *and* that `S` is closed by call (functions within that set do not call functions outside of it).

```
% Try to deduce directly with simple rules
meta_valid(C, P, S) :-
    meta_inter(C, P, S).

% Try to deduce as a negative_assigns
meta_valid("Postcond", negative_assigns(L), S) :-
    % Functions do not modify L
    meta_inter("Writing", not_written(L), S) &
    % Callees are within S
    callees_restrict(S, S).
```

Figure 6.5: Part of the high-level layer of deduction

The engine can then be queried to efficiently check if a particular HILARE can be deduced based on the known facts of the programs, or even list all deducible facts.

Remark 30 (Absence of circular dependencies). As described, the engine is separated in three layers: ground (facts), intermediate (simple deductions) and high-level (complex deductions).

This is because deduction rules from a given level must only depend on conditions from lower levels or otherwise more restrictive predicates, to avoid circular dependencies, which would cause the engine to run indefinitely.

While this allows for a performant deduction engine, it also means that it cannot perform a chain of high-level deductions to reach a goal: the chain must be broken up into separate deductions (and so separate HILAREs).

6.5 Automatic Deduction of HILARE from MetAcsl

The next step is to allow requesting the deduction of a HILARE from within MetAcsl. In the source specification, the verification practitioner must explicitly mark a property with the flag `proof:deduce` to disable its local verification (see Chapter 4) and instead attempt its deduction.

When encountering such a HILARE, there is a fully automatic (but limited) translation of the program environment into Prolog facts that can be used by the deduction engine. In particular, the translation exports the list of functions in the program as a set, the call graph as a list of calls, and the list of HILAREs appearing before the property under proof.

Remark 31 (Hypotheses). Since all HILAREs preceding the goal in the source specification are exported as *facts*, they are considered valid by the deduction engine and are therefore *hypotheses* of the goal, should it be proven valid.

Hence, one should always ensure all HILAREs are valid before concluding that a deduced HILARE is definitely valid. MetAcsl helps by automatically declaring used hypotheses as dependencies.

In presence of multiple HILAREs to be deduced, each one is deduced independently of the others, with multiple exports of the knowledge and multiple runs of the Prolog engine.

Export of the function set. The MetAcsl plugin lists all the functions of the C program and exports them as a set in a `targets({f, g, h, ...})` fact.

Export of the call graph. The MetAcsl plugin syntactically detects all functions calls in the C program (function pointers are ignored, a known limitation). Every edge of the call graph is then exported as a single `calls(f, g)` fact.

Export of HILAREs. When translating HILAREs, each one is exported into a `meta_ground(C, P, S)` where `C` is the name of the context, `S` is the target set and `P` the predicate.

For the predicate, pattern matching is used to syntactically identify some known forms of the HILARE language that can be further exploited by the deduction engine. If the pattern is not recognized at all (which is the case in most properties), the HILARE is exported with an abstract predicate about which nothing is known.

In particular, the exporter recognizes the `\separated(\written, &X)` pattern where `X` is a variable or the field of a variable (if it's a structure). In this case the variable is assigned a unique name and the whole predicate is exported as the `not_written(X)` special form.

Export of structures. If a program declares a structure type `str` with fields `foo` and `bar`, MetAcsl will export this type information to the solver:

```
field(str, foo).
field(str, bar).
```

And the Prolog solver knows that if it can establish that no field of a `str` variable is modified, then the whole variable can be considered untouched, and the other way around:

```
% If a struct is not modified, all of its fields aren't
meta_inter("Writing", not_written(field(V, F)), S) :-
    meta_inter("Writing", not_written(V), S).
```

> **Example 31.** In Figures 6.6 and 6.7, there are respectively a C program annotated with two HILAREs, and the translation of this program into Prolog knowledge, given that we want to deduce the second property (this wish is materialized by the `proof:deduce` flag). This illustrates the translation of different C elements into knowledge for the deduction engine.
>
> In this case, the property can successfully be deduced using the first proved lemma.

6.6 Usability and Extensibility

How to use. From the point of view of a regular user of MetAcsl and the HILARE language, the usage of our deduction framework is easy, as

```
int A, B;
void f3() {
    B = 42;
}
void f2() {
    B = 12;
    f3();
}
void f1() {
    B = 0;
    f3();
    f2();
}
void f0() {
    A = 42;
    f1();
    //@ assert A == 42;
}
/*@ meta \prop,
        \name(untouched),
        \targets(\callees(f1)),
        \context(\writing),

\separated(\written, &A); */

/*@ meta \prop,
    \name(nega_correct),
    \targets(\callees(f1)),
    \context(\postcond),
    \flags(proof:deduce, translate:yes),

A == \old(A); */
```

Figure 6.6: An example input program and specification

```
% Export of the set of functions
targets({f_f3, f_f2, f_f1, f_f0}).

% Export of the call graph
calls(f_f1, f_f3).
calls(f_f2, f_f3).
calls(f_f1, f_f2).
calls(f_f0, f_f1).


% Translation of untouched
meta_ground("Writing", not_written(a_22), {f_f3, f_f2, f_f1}).

% Translation of nega_correct. This is what we want to prove.
go :- meta_valid("Postcond", negative_assigns(a_22),
                 {f_f3, f_f2, f_f1}).
```

Figure 6.7: Knowledge base for the Prolog engine (automatically generated from Figure 6.6)

knowledge about its inner workings is not necessary. It is sufficient to add the `proof:deduce` flag to attempt deducing a HILARE from others with MetAcsl, seamlessly.

It is useful, however, to know about which kinds of deductions can be made by this solver. Hence, it would be advisable for a regular user to read Section 6.1 from this chapter, in order to know in which situations it might be useful to consider deduction as an option.

Furthermore, due to the current limitations of the pattern matcher described in Section 6.5, knowing about the handled predicate patterns will help a regular user to write HILAREs that will be understood by the system.

How to extend. Currently, the deduction framework handles the three high-level use cases described in Section 6.1, as well as trivial deductions between similar HILAREs handled by the intermediate layer described in Section 6.4.

In order to extend the scope of what can be deduced, one must:

1. Identify an example of deduction that should be handled.

2. Generalize that example to a deduction pattern.

3. Formalize and prove that deduction pattern within the existing Why3 model. It might be necessary to extend the existing model itself in order to reason about some language features.

4. Manually translate the statement of that deduction pattern to the Prolog solver. Again, it might be necessary to add new constructs to the solver for the statement to make sense.

5. Ensure that the exporter within MetAcsl can identify the patterns of HILAREs relevant to the deduction pattern.

This is exactly what was done for the three existing deduction patterns, and it is not an easy undertaking. However, we believe that the more patterns are added to the framework, the easier it will be to add new ones since the Prolog and Why3 models will already be complete enough to simply focus on the use case.

Chapter 7

The Wookey Case Study

While one of the purposes of the HILARE language (Chapter 3) and its associated plugin MetAcsl (Chapter 4) is to specify and verify high-level requirements, another purpose is to ease the specification task over large code bases. Hence, it was a natural step to tackle a larger, realistic use case, to both stress-test our approach and demonstrate its merits. Hence, this chapter presents a real case study on a large low-level code base, and applies the techniques and methodology described in previous chapters to specify and verify a set of key security requirements.

WooKey [Ben+19] is an open hardware and software project developed by the French National Cybersecurity Agency (ANSSI) aiming to build a custom USB thumb drive from the ground up focusing on security through built-in user data authentication and strong authentication. An example of an assembled WooKey device is shown in Figure 7.1.

Figure 7.1: An assembled WooKey device.

As seen on the figure, WooKey is a small device with a USB port, a touch screen and a smart card slot (on the photo, a white card is currently inserted). The device contains physical memory as a normal USB flash drive would. However, to access the stored data, any user must first insert her personal smart card then type a PIN code using the touch screen to unlock the storage. She can then use the device as a normal USB flash drive, then lock the memory again.

In this chapter, Section 7.1 presents the WooKey project and focuses on the low-level architecture and security model of the bootloader. Section 7.2 outlines multiple high-level security requirements that we want to verify on the bootloader. These requirements are refined into concrete HILAREs in Section 7.3 which describes our specification approach. Finally, Section 7.4 discusses how these properties have been formally verified via deductive verification, and thanks to the deduction framework we describe in Chapter 6.

7.1 Architecture

WooKey has multiple software components (mostly drivers) to manage the hardware key, such as SDIO[1], USB, cryptography management, etc. All components rely on the kernel called EwoK which contains them and enforces various security rules such as strict isolation between tasks and drivers or least privilege principle. Furthermore, the kernel is implemented in Ada/SPARK, often used in applied formal method domains such as avionics or railway systems to build safe software.

However, as every kernel must, the EwoK microkernel relies on the firmware's[2] bootloader (called the *loader* from now on), by which it is initially executed. Indeed, the loader has a critical role in WooKey's architecture: it not only boots to the kernel but also contains a number of security safeguards.

The WooKey device has a built-in flash memory that contains both the loader and the firmware. It allows the firmware to be updated by a mechanism called DFU (Device Firmware Upgrade), which is triggered if a physical button is pushed during the boot sequence. Due to technical limitations, firmware upload and verification have to be performed in-place, where it will be executed. This led to the adoption of a *flip-flop* mechanism

[1] Secure Digital Input Output, to interact with the mass storage.
[2] Firmware is the name given to software that drives low-level functions of hardware.

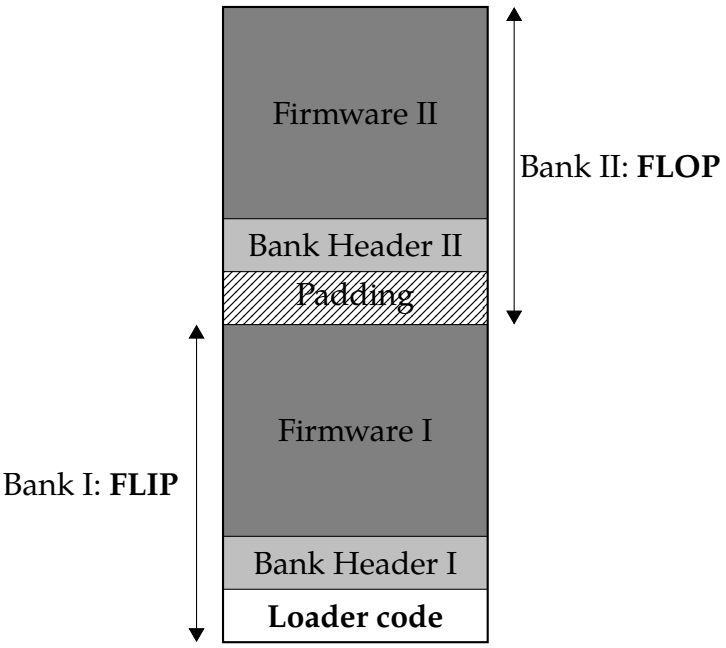

Figure 7.2: Flash memory layout with flip-flop redundancy. Memory addresses increase upwards.

which enforces software redundancy in order to handle different kinds of file corruption. This mechanism is illustrated in Figure 7.2.

The flash memory is separated into two redundant *banks* called respectively *flip* and *flop*. Each bank contains a (possibly different) copy of the firmware (the kernel and all its tasks) and the tasks responsible for firmware upgrade. Furthermore, each bank has a *Bank Header* section containing meta-data about the installed firmware in the bank such as hash sums, signatures and version numbers. There is only one copy of the loader, which cannot be updated, at the beginning of the memory. The loader is responsible for verifying each firmware against its meta-data section (to detect corruption or malicious modifications) and then choosing a bank to boot on based on the version numbers.

Overall, it is critical to ensure that the loader is safe and secure, since it (i) cannot be updated after device assembly, (ii) has every privilege during its execution, and (iii) can thwart every security measures of the upper layers.

7.2 Defining Important Requirements

In collaboration with WooKey's developers from the ANSSI, we established a set of high-level requirements of which the proof would both be useful to increase trust in the loader and be able to evaluate the relevance of the HILARE language and MetAcsl over such a large project.

> **Requirement 7.1 (Persistent bank choice).** Once a bank is chosen as a valid boot target, no further operations are realized on the other one.

As explained above and pictured in Figure 7.2, after performing various checks to determine if each bank – *flip* and *flop* – is securely bootable, the loader has to choose a bootable bank to boot on. It is essential that once the target bank is determined, every subsequent operation is realized on that bank (in particular, the boot itself!). Violating this requirement could allow booting to a corrupted firmware, for example via bypassing the subsequent integrity checks by performing them on a different bank.

> **Requirement 7.2 (Booting sequence enforcement).** Each step of the boot sequence is executed only once, and in the correct order described in Figure 7.3.

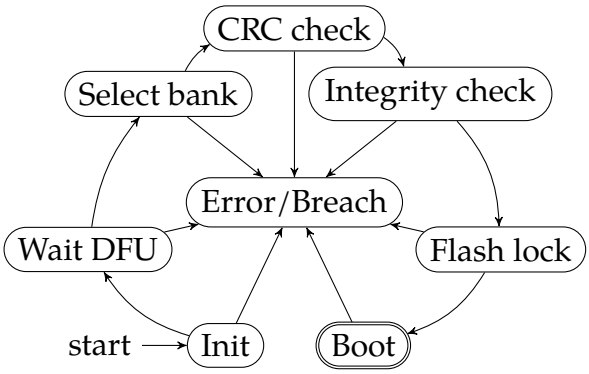

Figure 7.3: Boot sequence automaton

The outline of the whole operation of the loader can be described by a finite-state automaton, represented in Figure 7.3. Indeed, the functionalities of the loader can be grouped in 7 different steps, from initialization to the final boot. The semantics and implementation details of the intermediate steps do not matter to the requirement, which is that they must be

executed linearly one after the other, except if an error (safety or security) occurs, which causes the system to enter a sink failed state.

Violating this requirement could allow completely bypassing the security checks performed in the intermediate states of the automaton, or recovering from a security breach by avoiding sink states of the automaton.

> **Requirement 7.3 (Principle of least memory access).** Each step of the boot sequence only accesses the memory it strictly needs.

Since the loader has every privilege to access the memory, it is important to check that it does not abuse these privileges. Hence, it is necessary to establish the maximal memory footprint each step should have and ensure that these footprints are not overstepped. These expected footprints are detailed in Table 7.1, referring to the memory regions of Figure 7.2. Violating this requirement could allow the loader to spuriously modify the firmware, potentially corrupting it or adding a malicious payload.

Module	Can read	Can write	Can execute
Init	×	Protection flags	Loader code
Wait DFU	×	×	
Select bank	Bank Header I ∧ II	×	
CRC check	Bank Header I ∨ II	×	
Integrity check	Firmware I ∨ II	×	
Flash lock	×	Protection flags	Loader code
Boot	×	×	Firmware I ∨ II

Table 7.1: Expected footprints of the different modules on the flash memory.

Notice how after the bank selection, the modules can read either the first or the second bank, but not both, echoing Requirement 7.1.

7.3 Specification Approach

This work is based on the first release of WooKey (a subsequent one was released during our work, fixing a number of security issues). Some statistics about WooKey's loader and its dependencies' code base are highlighted in Table 7.2. It has a large size, which we deemed representative enough to evaluate our approach, despite the highly specialized code it contains.

Files	Functions	Lines of code	Loops	Global variables	Assignments
82	581	5250	58	29	1694

Table 7.2: Some statistics about the loader's code base.

Setting up an environment where Frama-C can parse the whole code base was relatively pain-free thanks to tools such as Bear [Bear] and documentation by the Frama-C team [Mar18]. At first, slight modifications where needed due to unsafe pointer casts in the code. Later, these were fixed directly by the WooKey team.

File structure. Most of the loader's logic is defined in a `main.c` file, while most of the other files contain helper functions to interface with the hardware and implementations of some common algorithms such as cyclic redundancy checks.

This `main.c` file defines a global structure `ctx` (context) which stores information gathered during the boot sequence, such as if the button for DFU was pushed, what bank was selected for boot, the final address to boot on, etc.

Another noteworthy file is `automaton.c`, which defines the automaton of Figure 7.3 and stores the current state of the sequence using a global variable `state` (an enumeration).

> **Remark 32 (Preservation of the original source).** A major requirement of the specification approach is that it *must not* modify the original source code in any way.
>
> Modifying the structure of the program while preserving the original behaviour would certainly ease the specification of some requirements. However, the goal of this chapter is to demonstrate that the HILARE language and MetAcsl are tools that are flexible enough to adapt to various forms of programs.
>
> Hence, we have only allowed ourselves to annotate the existing code and possibly add ghost code.

The following Sections 7.3.1 and 7.3.2 respectively describe how Requirements 7.1 and 7.2 were refined into enforceable high-level properties, using the methodology outlines in Chapter 5. By lack of time, Requirement 7.3 was not refined during the thesis.

7.3.1 Persistent Bank Choice

To refine Requirement 7.1 into high-level code properties, it is necessary to first refine the meaning of bank being "chosen" as a valid boot target, and what an operation on a bank is.

Identifying the relevant variables. Helpfully, the boot sequence has a single function where the choice is made, called `loader_exec_req_selectbank` (which we will call `selectbank` from now on), representing the associated state in Figure 7.3.

Furthermore, the `ctx` global contains boolean fields `boot_flip` and `boot_flop`. It is intended that after the choice function, one and only one of these two fields must be set to true, representing the chosen bank. Additionally, metadata associated with the chosen bank must be stored into the `fw` (firmware) field of the same global.

Hence, a bank can be considered as chosen after the execution of `selectbank` without errors, and the choice is embodied by the three aforementioned fields. Lastly, a fourth field `next_stage` is set at a later stage with the final boot address, which strictly depends on the earlier choice.

Operations on banks. While the previous paragraphs define what it means for a bank to be "chosen" as a boot target, Requirement 7.1 also mentions the notion of "further operations" on a bank. We can conservatively define an operation on a bank (flip or flop) as any access (read or write) to:

1. its metadata i.e. the members of `flip_shared_vars` or `flop_shared_vars` (both global structures containing information such as the status and signature of the bank);
2. its data i.e. the whole memory region starting from the base address `FLIP_BASE` or `FLOP_BASE` and of size `FLIP_SIZE` (which are compile-time macros).

From requirements to properties. With the clarifications made in the above paragraphs, we can refine the requirements into the following set of high- and low-level properties.

> **Property 1 (Function assumption).** The `selectbank` function is partially correct: it either enters an error state or returns in a state where

exactly one of `boot_flip` and `boot_flop` is true and `fw` is set accordingly.

Property 2 (Persistent choice). No function other than `selectbank` shall modify the above three fields.

Property 3 (Enforced choice). No function shall access or modify any location between `FLIP_BASE` and `FLIP_BASE + FLIP_SIZE` nor `flip_shared_vars` when `boot_flip` is set to false. Symmetrically for the *flop* bank.

Lastly, as mentioned previously, the `next_stage` field is set at a later stage (namely in function `loader_exec_req_flashlock`, or `flashlock` for short) to the final boot address, depending on the earlier bank choice and the potential press of the DFU button. Hence, it is important to check that this field is correctly set.

Property 4 (Continuation of choice – global). No function other than `flashlock` shall modify the `next_stage` field.

Property 5 (Continuation of choice – local). The `flashlock` function is partially correct: it correctly considers the earlier bank choice and the press of the physical button to decide where to boot.

As every location used in these properties is defined globally, it is easy to translate each property into its HILARE counterpart, using the *writing* and *reading* contexts.

For example, Property 2 can be specified with the macros and HILAREs listed in Figure 7.4. Notice how the usage of macros makes the actual specification much more legible, even by people without HILARE and MetAcsl background. We use a macro `UNTOUCHED` for encapsulating Pattern 4 from our methodology in Chapter 5 and another macro `ALL_EXCEPT_SELECTBANK` for specifying the set of targets.

As for Property 1, since it is simply a postcondition on the bank selection function, we can simply add an **`ensures`** clause to its contract or write a HILARE with the **`\postcond`** context, with the following predicate:

```
ensures \result == LOADER_REQ_ERROR ||
  (ctx.boot_flip == sectrue && ctx.fw == &flip_shared_vars.fw) ||
  (ctx.boot_flop == sectrue && ctx.fw == &flop_shared_vars.fw);
```

```
#define UNTOUCHED(title, tset, loc) \
    meta \prop, \
    \name(title), \
    \targets(tset), \
    \context(\writing), \
    \separated(\written, loc);
#define ALL_EXCEPT_SELECTBANK \
    (\diff(\ALL, loader_exec_req_selectbank))

/*@ UNTOUCHED(bank_ro_1, ALL_EXCEPT_SELECTBANK, &ctx.fw) */
/*@ UNTOUCHED(bank_ro_2, ALL_EXCEPT_SELECTBANK, &ctx.boot_flip) */
/*@ UNTOUCHED(bank_ro_3, ALL_EXCEPT_SELECTBANK, &ctx.boot_flop) */
```

Figure 7.4: Specification of Property 2 with HILAREs

> **Remark 33.** In the above predicate, the operator between the two last statements should be an exclusive disjunction to correctly reflect the property. However since the `ctx.fw` field cannot have two different values, a simple disjunction is enough here.

In Section 7.4, we will discuss how using the deduction framework presented in Chapter 6 was useful for the verification of this property.

7.3.2 Booting Sequence Enforcement

Refining Requirement 7.2 to code properties is not quite as easy, as it needs a bit more knowledge of the underlying code.

Identifying the relevant variables. As mentioned in the beginning of the Section, the main anchor point for identifying the current state of the sequence is the global `state` variable from `automaton.c`. Furthermore, the automaton of Figure 7.3 is explicitly defined as a structure listing permitted transitions from one state to another, in a global constant structure called `loader_automaton`.

Manipulation of the automaton. The `automaton.c` file contains a list of facilities for manipulating the raw variables, such as `loader_set_state` or `loader_is_valid_transition`. These functions are then used in `main.c` to manage the control flow.

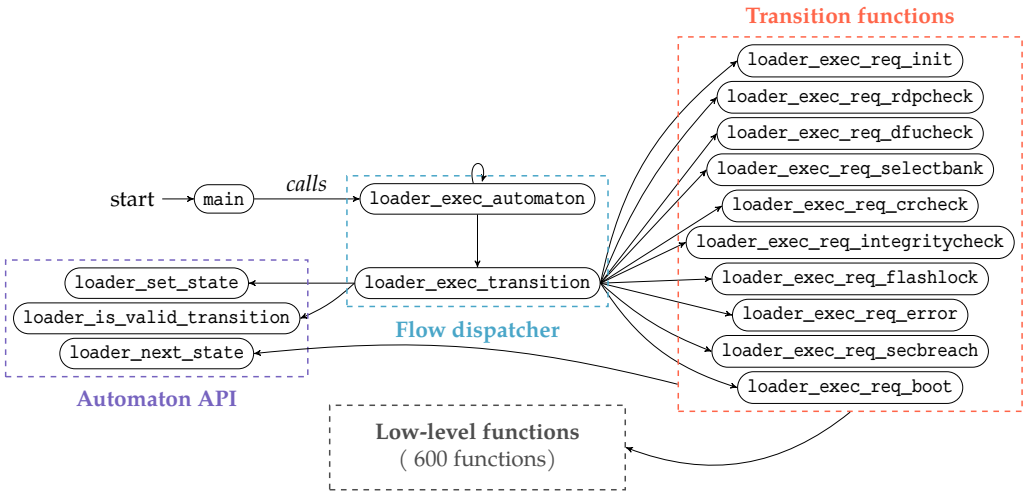

Figure 7.5: Outline of the loader's call graph.

Transition functions. The general outline of the loader's call graph is illustrated in Figure 7.5. There are a number of transition functions that contains the main logic of each step. For example, the previously mentioned `selectbank` contains the code that selects the bank, embodying the transition from "Wait DFU" to "Select bank" in Figure 7.3's automaton.

These transition (or *request*) functions must return the expected next state in the sequence. This return value is carried back to `loader_exec_automaton` which calls `loader_exec_transition` to check that a transition to the requested next state exists, and then call the appropriate transition function.

> **Remark 34 (Automaton hypothesis).** One assumption that we must make is that the automaton described by the `loader_automaton` mentioned earlier is correct with respect to the intended boot sequence order of Figure 7.3. This must be carefully verified manually.

Refinement of the requirement. Based on the hypothesis of Remark 34, it is enough to specify that the loader strictly abides by the automaton to refine Requirement 7.2. The code can be divided into four main components (as seen in Figure 7.5), of which three manage the general operation of the sequence. This allows to divide the specification itself into three problems:

(i) *the flow dispatcher* must correctly use the automaton API and dispatch calls to the right transition functions based on the automaton;

(ii) *the automaton API* must correctly manipulate the raw data structures (`state` and `loader_automaton`). Furthermore, no other function can manipulate these structures;

(iii) *the transition functions* must comply with the `nextstate` parameter (read on).

While specifying local correction is important, it is also crucial to specify and verify our assumptions about the control flow of the program: any unexpected call from one component to another (for example, from the automaton API to a transition function) could compromise the boot order.

In particular, the bulk of the hundreds of functions used by the loader is included in the so-called "Low-level" component, where it is plausible for a call back to another component to go undetected. This is especially true since the hundreds of functions are scattered across dozens of files in a complex file tree. Hence, the last problem:

(iv) there must be *no calls* between the components other than those illustrated in Figure 7.5.

The next four small subsections each tackle one of the four presented specification problems.

The flow dispatcher

While specifying (i) is the cornerstone of the whole specification, this is relatively straightforward. Function `loader_exec_automaton` is simply a `while` loop repeatedly calling `loader_exec_transition` with the requested next state. There is not much to specify:

```
static void loader_exec_automaton(loader_request_t req) {
    while (true) {
        req = loader_exec_automaton_transition(req);
    }
}
```

As for the main dispatch function, `loader_exec_automaton_transition`, it is essentially a large `switch` with safety checks. It is partially listed in Figure 7.6 (the first comment is part of the original source).

This function takes a request to a next state and calls the automaton API to check that the transition from the current state to the requested

```
static loader_request_t loader_exec_automaton_transition(
    const loader_request_t req) {
    loader_state_t state = loader_get_state();
    if (! loader_is_valid_transition(state, req)) {
        loader_set_state(LOADER_ERROR);
        goto end_transition;
    }
    loader_state_t nextstate = loader_next_state(state, req);
    /* nextstate must always be valid,
     * considering the automaton defined
     * in automaton.c */

    switch(req) {
        case LOADER_REQ_INIT:
        return loader_exec_req_init(nextstate);
        // One case for each possible state, omitted
        // ...
    }
}
```

Figure 7.6: The main dispatch function of WooKey's bootloader

one is valid. It then calls the appropriate transition function using a large `switch`, passing the next state to the transition function.

Notice that this function does not change the state of the automaton. It is the responsibility of each transition function to set the state according to the parameter it receives.

The correction of this function is mainly based on the correction of the automaton API itself (which is called to ensure that the transition is valid). The only thing about this function that must be verified is that for each possible request, the right function is called by the `switch`. This is best done visually.

The Automaton API

The automaton API and implementation, contained in `automaton.c`, is responsible for performing operations on the automaton and ensuring it remains in a valid state. It contains the four following functions:

```
// Get the current state
loader_state_t loader_get_state(void);
// Set the current state
```

```
void          loader_set_state(const loader_state_t new_state);
// Return the next state given the current one and a transition request
loader_state_t loader_next_state(const loader_state_t current_state,
                                 const loader_request_t request);
// Check that a transition request is valid given the current state
secbool loader_is_valid_transition(const loader_state_t current_state,
                                   const loader_request_t request);
```

The first two functions manipulate a global variable `state`, which contains the current state of the automaton. The two last functions `loader_next_state` and `loader_is_valid_transition` perform lookups in the global `loader_automaton` structure, which contains a representation of all valid transitions.

> **Remark 35 (Secure booleans).** The `secbool` type is a special type from WooKey intended to implement booleans that are resistant to fault injections[Voa97]. It has the following definition:
>
> ```
> /* Secure boolean against fault
>    injections for critical tests */
> typedef enum
>   {secfalse = 0x55aa55aa, sectrue = 0xaa55aa55} secbool;
> ```
>
> The values are chosen so that it is hard to transform a `true` into a `false` simply by inverting bits.

This small API and implementation of automaton is not only used in the bootloader of WooKey but in other components as well. Hence, verifying the behaviour of each function is important in terms of safety and security.

> **Property 6 (Correction of automaton functions).** Each function of the automaton API has the expected behaviour.

Function contracts. To that end, we simply annotate each function with a classic ACSL contract that reflects its behaviour. We give an example of contract in Figure 7.7, where the behaviour of `loader_is_valid_transition` is specified.

In this contract, we make use of two ACSL predicates defined elsewhere: `is_cell_accessor_index(i, s)` and `is_next_state_index(i, s, r)`. The first checks that `s` is an existing state (more precisely, the $\texttt{i}^{\text{th}}$ element

of the state enumeration). The second one checks that `r` is a valid request from state `state` in the automaton (more precisely, it is the $\texttt{i}^{\text{th}}$ valid transition from that state).

The postcondition states that the parameter should be a valid state. We then have two possible behaviours:

1. If the request is valid for the current state, then the function should return `sectrue` (see Remark 35), and the current state shouldn't change.

2. Else, the function should return `secfalse`.

The same kind of contract is laid out for the three other functions.

> **Remark 36 (Centralization of specification).** In order to avoid dispersing the different contracts and annotations throughout the dozens of files, and in an effort to always avoid modifying the original source (even for annotations), we use the ACSL Importer [Bau] Frama-C plugin to write all the specification in one place. The plugin then handles dispatching the different annotations everywhere needed.

Validity of state. Since all functions in the bootloader should only manipulate the automaton through the dedicated API, and that API ensures

```
/*@ requires \exists integer i;
        is_cell_accessor_index(i, current_state);
    behavior found:
        assumes \exists integer i;
            is_next_state_index(i, current_state, request);
        ensures \result == sectrue;
        ensures logic_state == \old(logic_state);
    behavior not_found:
        assumes \forall integer i;
            !is_next_state_index(i, current_state, request));
        ensures \result == secfalse;
    complete behaviors;
    disjoint behaviors; */
secbool loader_is_valid_transition(... current_state, ... request);
```

Figure 7.7: The contract of a function from the automaton API

the state remains valid at all times, we can specify the following high-level property.

> **Property 7 (State remains valid).** At every point of every function of the bootloader, the `state` global variable (and hence the result of `loader_get_state`) should be a valid state (a known member of the enumeration).

In the next section, we will see how this property can again be *deduced* from others.

As stated in (ii), we also want to ensure that all the global structures are only manipulated through the API. More precisely, the `state` global variable can only be modified by its setter.

> **Property 8 (State encapsulated in setter).** The `state` global variable is not modified by any function, except `loader_set_state`.

This property is easily specified with a HILARE in the `\writing` context.

Transition functions

As described in Section 7.3.2, transition functions receive a parameter from the main dispatcher. This parameter, called `nextstate` in all transition functions, contains the state that the automaton is expected to have at the end of the transition. The transition function is supposed to call `loader_set_state` at some point to enact the transition. We want to verify that.

> **Property 9 (Enactment of transition).** At the end of every transition function (except the ones which never return), the current state is `nextstate` or an error state.

In order to specify this property with a HILARE, we can leverage the `\formal` construct described in 3.3.4 to refer to the `nextstate` parameter, common to our targets. The resulting HILARE is listed in Figure 7.8.

We use a macro to define the set of all transition functions (the macro is not only used here). From this set, we exclude the three non-returning functions: two sink error states, and the final boot state (which just jumps to the boot address to launch the kernel).

```
#define TRANSITION_FUNCTIONS ({          \
    loader_exec_req_init,                \
    [...]                                \
    loader_exec_req_boot                 \
})

/*@ meta \prop, \name(enactment_of_transition),
    \targets(\diff(TRANSITION_FUNCTIONS,
                \union(loader_exec_req_boot,
                       loader_exec_error,
                       loader_exec_secbreach))),
    \context(\postcond),
    \fguard(
        state == \formal(nextstate) ||
        state == LOADER_ERROR
    );
*/
```

Figure 7.8: Specification of the expected state change for transitions

We then state that all of these functions should ensure that after their execution, the current state is either the `nextstate` parameter or an error.

Isolation of components

The last specification problem, (iv), is to ensure that the interfaces between the different components are those illustrated in Figure 7.5: there must be no spurious calls between unrelated components. We can separate this requirement into multiple properties.

> **Property 10 (Flow dispatcher calls everyone else).** Functions from the flow dispatcher component are only called by themselves or `main`.

Indeed, the role of this component is to *orchestrate* the others. Any call back to that component would radically alter the flow of control.

> **Property 11 (Automaton state modification).** The `loader_set_state` function is only called by transition functions, except to go to an error state or in `main`.

As seen in the previous component, the state of the automaton is actually changed in the transition functions (always to the next state of the transition). It is of the utmost importance that no other function (especially in the low-level component) changes the state.

> **Property 12 (Dispatch of transition functions).** The transition functions are only called by the flow dispatcher, namely `loader_exec_transition`.

Finally, since the call of transition functions is orchestrated by the flow dispatcher, no other component should be able to call them.

All isolation properties can simply be stated using Pattern 9 from Chapter 5 to restrict the callees of some sets of functions.

Property 11 is slightly more complicated, as we must refer to the potential value passed to `loader_set_state` to allow for errors. It can be specified with the following HILARE:

```
meta \prop,
    \name(state_wrapper_only_called_in_transitions),
    \targets(\diff(\ALL,
                \union(TRANSITION_FUNCTIONS,
                       main))),
    \context(\calling),
    \tguard(\called == loader_set_state
        ==> \called_arg(new_state) == LOADER_ERROR );
```

We state that in all functions except transition functions and `main`, if `loader_set_state` is called then it can only be to set the error state. We use the `called_arg` construct presented in Section 3.3.5 to refer to the value passed to the callee.

7.4 Verification Process

In the previous section, we refined two of the three main requirements presented in Section 7.2 into a set of properties (1 through 12). As illustrated for some of these properties, each one correspond to either a HILARE (a high-level property) or a low-level annotation such as a function contract.

All the HILAREs were centralized into a single `specification.h` file, which was passed to Frama-C after every other source file. As explained by Remark 36, the low-level annotations were centralized into a separated file

handled by the ACSL Importer plugin. The full specification is available within the release of MetAcsl[3].

Deductive verification. We used the Wp plugin in combination with MetAcsl to attempt proving all the above properties. The total number of inline assertions generated by MetAcsl is 183, which is relatively small compared to the size of the code base. This can be explained by the fact that most of the functions do not manipulate pointers, which allows MetAcsl to automatically discard most of the assertion candidates as trivially true or false.

Proving the remaining 183 assertions and the various contracts took a fair amount of effort. In some cases, it was necessary to manually write partial contracts and loop invariants. For example, it was necessary to add loop invariants in all transition functions for Property 9, since it specifies postconditions for these functions.

In several instances, it was useful to add new HILAREs as high-level lemmas to specify global properties that help prove the local assertions. Some of them were able to be immediately deduced from others (see Section 7.4.1).

Overall, every property has been proved by Wp and its SMT solvers without having to manually write proofs (using Coq for example), but instead simply adding specification locally or globally. The final specification takes approximately 6 minutes to be fully proved on a single 3 *Ghz* CPU core, and fully verify Requirements 7.1 and 7.2 on the bootloader.

7.4.1 Relevance of the Deduction Framework

The proof of several properties was greatly eased by the deduction framework built in Chapter 6. It was particularly useful in functions where we wanted to establish a postcondition (notably for Properties 1 and 9) but which called potentially many functions unrelated to that property.

Deducing "negative footprints". For example, Property 1 is a postcondition on the `selectbank` function, stating that the global fields related to the choice of banks end up in a correct state. Between the instructions which set these fields in a correct way and the end of the function, there are multiple calls to other functions, mainly for debugging purposes. These

[3]In folder `case_studies/wookey/loader` at https://git.frama-c.com/pub/meta.

functions themselves call lower-level functions to interact with the system. However, none of them modify the variables related to the choice of banks.

Normally we would have been forced to specify and prove that each of these unrelated functions do not modify the relevant variables, either by stating that as a postcondition or as part of their frame clause. In both cases, as discussed in Section 6.1.1, this would have been a tedious work, especially for the functions involving assembly language.

```
#define DEDUCED_NEG_ASSIGNS(title, tset, val) \
    meta \prop, \
    \name(title), \
    \targets(tset), \
    \context(\postcond), \
    \flags(proof:deduce, translate:yes), \
    val == \old(val);
#define CALLED_BY_SELECTBANK \
    (\diff(\callees(loader_exec_req_selectbank), loader_exec_req_selectbank))

/*@ UNTOUCHED(ctxbflip_not_written_by_utils,
        CALLED_BY_SELECTBANK, &ctx.boot_flip) */
/*@ DEDUCED_NEG_ASSIGNS(ctxbflip_not_changed_by_utils,
        CALLED_BY_SELECTBANK, ctx.boot_flip) */

/*@ UNTOUCHED(ctxbflop_not_written_by_utils,
        CALLED_BY_SELECTBANK, &ctx.boot_flop) */
/*@ DEDUCED_NEG_ASSIGNS(ctxbflop_not_changed_by_utils,
        CALLED_BY_SELECTBANK, ctx.boot_flop) */

/*@ UNTOUCHED(ctxfw_not_written_by_utils,
        CALLED_BY_SELECTBANK, &ctx.fw) */
/*@ DEDUCED_NEG_ASSIGNS(ctxfw_not_changed_by_utils,
        CALLED_BY_SELECTBANK, ctx.fw) */
```

Figure 7.9: High-level deduction in WooKey

Instead, we were able to specify the HILAREs illustrated in Figure 7.9. For each of the three global fields `ctx.boot_flip`, `ctx.boot_flop` and `ctx.fw`, we first specify that no function called by `selectbank` modify them. This is easily proved with Wp and MetAcsl, and does not involve manual annotations.

Then we use the deduction framework to simply deduce that the value of these three fields must not have changed during the execution of these

functions. This deduction is made under two seconds in each case. Since we use the `translate:yes` flag in the deduced properties, MetAcsl still generates local assertions for them, but they are immediately considered correct. Using them, Wp is able to automatically infer that calling low-level functions in `selectbank` does not modify the relevant variables, and thus that our desired postcondition is true. The same deduction pattern is used for the proof of Property 9.

Deducing a global invariant. In Property 7, we specify that the state of the automaton must remain valid at all times. This can be deduced from Property 8 stating that no function other than `loader_set_state` modifies the state, and from the fact that we proved this function maintains a valid state.

Using the deduction patterns illustrated in Sections 6.1.2 and 6.1.3 from Chapter 6, we are able to successively deduce that the valid state is a *conditional invariant* of all functions, and then a *weak invariant*. This helped prove assertions related to the call of the automaton API, which requires that the states passed as parameters are valid.

This demonstrates the relevance of the deduction framework for real applications and its ability to scale on large codebase, where it is most useful.

Chapter 8

Related Work

This chapter presents other works related to our domain and the stance of our approach compared to them. Section 8.1 discusses previous efforts to specify and verify high-level properties while Section 8.2 describes work on systems similar to our case studies.

8.1 High-Level Specification and Verification

This Section relates to Chapter 2 through 4, presenting the main objective and solution of the thesis.

Previous works have proposed various approaches to make high-level requirements amenable to specification and verification.

8.1.1 Extensions of Specification Languages

The new specification language presented in Chapter 3 partially overlaps with previous extensions of contract-based specification languages such as JML [LBR99], a behavioural specification language for Java programs.

Indeed, JML has been extended by Cheon and Perumandla [CP05] to specify protocols (properties pertaining to the order of call sequences), and by Trentelman and Huisman [TH02] to express temporal properties. While protocols may be expressible with our work as well as a subset of temporal properties, it may not be as simple as the syntax provided in their works, since such properties are not our main focus. On the other hand, while these works achieve their specific goals efficiently, they cannot be used to specify arbitrary high-level requirements on programs easily.

The general idea of defining a high-level concept in the global scope and then *weaving it* into the program has been partially explored before: the work of Pavlova *et al.* [Pav+04] enables enforcing high-level security properties by performing a code transformation weaved throughout the implementation. But as mentioned in the previous chapter, the properties considered by the authors are very specific to the Java Card programming language, both in terms of specification (only some pre-defined categories of properties can be expressed) and verification: the code transformation is based on the assumption that there are a set of *core* functions acting as the main interfaces to the smart card. While their verification mechanism is quite similar to what we present in Chapter 4, it lacks generality at the specification level.

Similarly, ACSL [Bau+20a], the specification language we are choosing to extend in this work, has been extended in different ways in the past. To cite only a few, Stouls and Groslambert [SG11] introduced Aoraï [1] to allow the specification and verification of complex temporal properties (very partially supported by our approach) while Blatter *et al.* [Bla+18] explored the specification and verification of relational properties, a different class of interesting properties, with Rpp [2]. Again, these efforts have been very successful but target different requirements than this work.

8.1.2 Link with Aspect Oriented Programming

At the same time, this concept of cross-cutting global object is analogous to the Aspect-Oriented Programming (AOP) [Kic+97] paradigm. This paradigm allows writing code (called *concerns*) at the global level while specifying a set of control flow points (called *pointcuts*) where the code needed by the concern should be inserted. For example, this may be useful for easily adding logging to an already existing implementation.

Our approach can be seen as writing cross-cutting concerns at specification level rather than code level. Indeed, contexts, as introduced in Chapters 2 and 3, can be related to AOP's pointcuts: they are a criterion for specifying a set of control flow points and performing a local specification action. This comparison is continued within Chapters 2 and 3.

Several other works explore what they identify as *Aspect-Oriented Specification*, in different ways than ours: Zhao and Rinard [ZR03] explore how to specify programs written in AOP with a notion of invariant, and propose a translation of that specification to JML. Bagherzadeh et al. [Bag+11]

[1] Available at https://frama-c.com/fc-plugins/aorai.html.
[2] Available at https://frama-c.com/fc-plugins/rpp.html.

propose another way to reason about AOP programs. Compared to our work, these efforts focus only on reasoning about programs that are already written in an AOP style, i.e. where some pervasive programming logic is *already* centralized at a single point, whereas we want to tackle arbitrary program and only centralize at the *specification level*.

Closer to our work, Yamada and Watanabe [YW06] propose another specification approach that can be translated to JML, allowing the specification of generalized weak invariants over arbitrary classes and methods, by specifying so-called *assertion aspects* inspired by AOP concepts, which translate to local annotations. While our work has the same inspiration (trying to centralize similar assertions spanning many functions), this approach lack the expressiveness we are aiming for, especially in terms of memory management.

8.2 Case Studies

This Section relates to the simple case study of Chapters 3 and 4 as well as the more comprehensive case studies of Chapters 5 and 7.

Verification of kernels and firmwares. Defining and verifying security properties of OS kernels down to the concrete source code was done in several projects, including notably [Ric10; Mur+13; CSG16; Dam+13; Jom+18; Les15], with different strategies. Compared to these projects, which target microprocessors with MMU-based memory isolation, the security properties shown in this thesis are just simple examples for the sake of illustration and better match microcontroller setups, which usually have no MMU and thus do not implement virtual memory. For instance, a model of the hardware including the MMU, the processor's privilege levels and the program counter is required to show that the MMU configuration matches the kernel's internal control structures and that the control flow only enters the kernel privilege level at the defined entry points for system calls, exception, and interrupt handling.

Moreover, some projects prove higher-level security policy abstractions like information flow enforcement [Mur+13; Dam+13; CSG16], from which confidentiality and integrity properties similar to the ones introduced in this thesis can be derived, such as memory isolation in Pip [Jom+18], as defined for a separation kernel [Rus81].

Compared to these projects, the contribution illustrated in this thesis is the ability to both define and mechanically verify global security properties directly at the level of the C source code. Indeed, all previous approaches

first define and prove the security properties on an abstract model of the OS kernel and the underlying hardware and second show that the concrete, low-level code executed refines the abstract model and still verifies the security properties defined on the abstract model. It is argued that for verification purposes, it is easier to reason on an abstract model than on low-level concrete source code [Mur+13]. On the other hand, proving that the source code refines the model can be very difficult if the code has not been explicitly designed for that purpose: rather than designing and bending the abstract model to fit on existing code, it is sometimes easier to directly reason on it.

Similarly, in seL4, an abstract model in Isabelle/HOL and the C source code are written separately and all mechanized proofs of security properties rely on a first mechanized proof of correctness that the C source code refines the abstract model [Mur+13]. This proof of correctness initially cost 25 person-years [Kle+09] and the team is still working on how to improve its maintainability [And19].

To make this approach more scalable, in CertiKOS, the OS kernel and the hardware are modeled as a set of small stackable layers (also called *deep specifications*), whose interfaces and observable behaviours are defined in Coq [Gu+16]. In each layer, the C or assembly implementation is verified to be a refinement of the layer's upper interface (called *overlay*) assuming that the layer's lower interface (called *underlay*) is correctly implemented by lower layers. The verification of each layer implementation is done using Coq tactics [Gu+15]. Global properties can then be proved using only the Coq model of the layers' interfaces.

The drawback of such a modularized approach is that it makes it difficult to obtain an efficient implementation of a microkernel, because in microkernels targeting performance the implementations of the required features are entangled. Indeed, by design such software must only implement the bare minimal features required at the kernel privilege level and the performance of the microkernel is critical for the performance of the whole software system.

In PROSPER [Dam+13], refinement steps are bypassed using a HOL4 model of the ARMv7 instruction set architecture [FM10]. The information flow properties are defined and verified on an abstract model of idealized ARMv7 machines representing the user-level execution contexts and the kernel execution context. In the verification process, HOL4 is used to generate pre- and post-conditions for atomic executions of the kernel at boot-time and from entry to exit to/from the kernel execution context (that is, for hypercall, exception and interrupt handling).

The verification of the concrete kernel code thus consists in verifying

these pre- and post-conditions and is done at the ARMv7 binary code level using the BAP suite [Bru+11]. Although this approach has not been tested on a unified full-featured OS microkernel, it is likely to be more scalable than in seL4 and more performance-compatible than in CertiKOS. However, it forces to reason on a model of the target machine architecture, making the source code an implementation detail, while MetAcsl allows us to reason on the C source code and its adaptations to the target machine architecture.

In Pip and ProvenCore, the C source code is automatically generated from an executable specification, in Coq for Pip [Jom+18] and in a proprietary language called Smart for ProvenCore [Les15], on which all properties and mechanized proofs are written. The extension of the proof to the C code relies on the code generation process, whose proof of correctness is ongoing for Pip [Jom+18] and is not public for ProvenCore. While generating C code from an abstract executable specification makes it easier to prove the actual C code, this adds the open challenge of being able to generate efficient code and certifying such a generator. Ongoing efforts study Coq tactics [Pit+20] that implement proven heuristics to generate efficient code.

A few security properties of WooKey's bootloader were defined in [ANS+]. Some of the authors used the Frama-C GUI and the EVA plugin [BBY17] to understand the undocumented code by performing value analysis via abstract interpretation. With this approach, they infer the transitions of the automaton from the code and state that since the automaton is simple, it is enough to verify visually that there is no unexpected sequence of transitions, but suggest the usage of MetAcsl for more complex properties. They also check for runtime errors and dead code with EVA. Then the authors proceed to use dynamic symbolic execution with KLEE [Klee] to check a functional property of the bootloader: ensuring that the dual-bank bootloader cannot boot from previous firmware versions (the anti-rollback mechanism). Compared to our work in Chapter 7, the authors of [ANS+] only very lightly apply the approach of global specification, instead focusing on specific functions and completely automatic verification rather than deductive verification.

Bootloader verification was attempted for SABLE [Con+18] and a simplified variant of a RISC-V first-stage bootloader (FSBL) [Str20]. In both cases, the bootloaders ensure integrity and authenticity of the loaded software and the platform state, although with different strategies. In SABLE, the dynamic root of trust measurement facility of Intel and AMD processors as well as a hardware TPM are used to prevent system image decryption in case of failure and otherwise allow an external user to do

remote attestation of the successful boot process. More traditionally and like in WooKey, the RISC-V FSBL studied in [Str20] is the ROM-located first piece of software executed at boot and checks the cryptographic signature of the loaded software.

In both projects the verification follows techniques already used for microkernels but is reported as partially achieved. SABLE follows the strategy of seL4 [Kle+09] while the RISC-V FSBL follows the code generation strategy down to RISC-V binary code using Bedrock2 [EG], an imperative language and compiler written in Coq, and the riscv-coq implementation of the RISC-V specification in Coq. Interestingly the RISC-V FSBL verification covers the use of the hardware DMA to load the software image.

8.3 Industrial Application of HILARE

The HILARE based approach and the MetAcsl tool presented in this thesis have already been used in a large-scale industrial verification project [DHK21].

It was performed for a security-critical smart card product in order to perform its rigorous Common Criteria based certification at level EAL6. Security of a smart card strongly relies on the requirement that the underlying JavaCard virtual machine ensures necessary isolation properties.

In that project, formal verification of a JavaCard Virtual Machine implementation was performed by Thales using the Frama-C verification tool set. It strongly relied on the MetAcsl tool. The target security properties include integrity and confidentiality. The target implementation contains over 7 000 lines of C code. In particular, based on only 36 HILAREs manually specified by verification engineers, approximately 400 000 lines of ACSL annotations (leading to over 27 400 proof goals) were automatically generated by MetAcsl and formally proved by Wp. This industrial verification project illustrates the potential of industrial applications of the contributions proposed in this work.

Chapter 9

Conclusion

9.1 Summary

In this thesis, we have provided a solution to the problem of specifying high-level requirements on large programs, and applied this solution to the security of a large C program.

The original problem that we are addressing is the lack of specification formalism for specifying requirements that are not specific to small code units but rather pertain to large components of the system. This problem was especially apparent for security requirements such as memory isolation, access control, confidentiality and privacy. There was originally no means of specifying and verifying these requirements in an expressive, maintainable and traceable way.

We proposed a new class of properties called meta-properties, formalized on a small, abstract programming language: A-Lang. These language-independent properties are essentially global predicates that are woven across the source code of large components, according to a criterion called context. Contexts are rules to automatically select code points where the predicate should hold and allow predicates to refer to local information: for example, a predicate can check whether an instruction is modifying a particular variable or not. Several base contexts are highlighted for interacting with memory operations and function calls.

In order to make these meta-properties actionable, we refined their formal framework into a concrete way to express and validate them on a real programming language, namely C. We designed a concrete syntax, called the HILARE language, for specifying meta-properties on C programs. This syntax is an extension of ACSL, a pre-existing contract-based specification language for C programs. We showed how the formal con-

texts translate for C and how the quirks of the C language interact with high-level requirements. In particular, we developed several extensions on top of the initial formalized features to ease the specification of structured C programs.

Specification written in ACSL can be assessed with a variety of tools in the Frama-C framework's ecosystem. We devised an automatic technique to translate global HILARE requirements to local ACSL specification. This allows the existing tools to be reused in order to build a complete tool chain for HILARE verification. This automatic translation was implemented in a new Frama-C plugin called MetAcsl.

While the translation from global to local requirements makes verification possible, it does not always scale well when the code base is large, especially when using verification techniques where manual effort is needed. To alleviate the problem, we designed the foundations for a sound framework which enables deducing global requirements from others, without a local intermediate step. We provided proofs of the framework's soundness machine-checked with Why3 and implemented a limited number of deduction patterns in an efficient Prolog solver, integrated seamlessly within MetAcsl.

Given all these tools, we devised a general methodology for users wanting to tackle the specification and verification of high-level requirements on real C programs. This methodology articulates the notions of the thesis into clear steps and patterns that can be reused across different specification problems. Numerous examples are given, from the specification of memory confidentiality to security properties of an artificial micro-kernel. Lastly, we applied this methodology to a concrete program found in the wild: the bootloader of the WooKey project. Using the concepts and tools presented in this thesis, we were able to specify several security requirements on the bootloader and verify that the code is valid with respect to these properties, with little manual effort.

The MetAcsl tool has been successfully applied by Thales in an industrial context for formal verification of security properties of a JavaCard Virtual Machine [DHK21]. We believe that it has a strong potential for other industrial applications for verification of security-critical software.

9.2 Future Work

There are several interesting research avenues that build upon the work presented in this document.

9.2.1 Extending and Improving the Deduction Framework

The deduction framework presented in Chapter 6 is still experimental and has several limitations.

Currently, the automatic translation from MetAcsl to Prolog handles only simple cases. In particular, only non-pointer global variables are considered by the pattern matcher: arrays and pointers in general are not handled and nothing can be deduced about HILAREs referencing them.

Furthermore, the pattern-matcher only recognizes simple forms of properties and do not handle composition well. For example, it will recognize that the predicate

```
\separated(\written, &A)
```

is stating that `A` is not modified, but not that

```
\separated(\written, &A) && \separated(\written, &B)
```

is the same for two variables.

This could be improved by either improving the pattern-matcher to handle logical operators or include a subset of ACSL and its semantic inside the Prolog engine, and a corresponding proof in the Why3 model.

Furthermore, as discussed in Section 6.6, there are only three simple deduction patterns currently implemented in the solver and proved in Why3: there is a lot of room for extending the scope of the deduction. One obvious extension is the addition of more deduction patterns to cover more situations where deduction would be desirable.

More generally, the fact that the framework currently only deals with ad-hoc patterns is a limitation. It would be desirable for the framework the have a more general purpose. For example, should a sizeable subset of ACSL be formalized in Why3, the deduction framework could be able to deduce high-level requirements from others only based on their semantics. However, it would be a challenge to design such a general approach while retaining its efficiency on large code bases.

9.2.2 Higher-Level Specification Language

While they enable the specification of high-level requirements, meta-properties and HILAREs are still quite close to the code: the user must precisely qualify the set of targets, the kind of operations targeted by the context and correctly refer to the global state. This allows specifying properties in the global scope but can sometimes feel verbose, and hard to proofread for a domain expert with little knowledge of the code base.

This can be alleviated with macros, as demonstrated in this thesis: C macros can abstract away lists of targets and HILARE patterns (as seen for example in Figure 7.4) and even parts of predicates. However, this is a limited approach as it relies on the C preprocessor, which only manipulates text instead of semantic tokens and is notoriously hard to work with.

A macro system, not documented in this thesis, has been implemented within MetAcsl to serve as a potential future foundation for a more robust means to abstract low-level operations in a way that domain experts can understand. This budding macro system was used extensively in the specification of the confidentiality use case presented in Chapter 3. The actual macro definitions are listed in Figure A.1 in the appendix. Currently, it does not offer more features than the C preprocessor allows.

A good way of improving upon this approach could be to develop this abstraction system by specializing it: there could be abstract sets of functions, predicates, HILARE patterns that are statically type-checked instead of simply replacing text. Using this, a set of useful built-in macros could be aggregated into a library of macros.

One could then imagine that the specification of an application should only be written using this higher-level library, only punctually adding new domain-specific ones, so that the specification remains legible for all. This could also enable the generation of specification from other sources such as abstract models from other applications, where the "glue" between the higher-level specification and the concrete code is materialized by a set of macros defined afterwards.

Appendix A

Implementation and Specification of the Page Manager

This appendix contains the complete specification and implementation of the page manager case study, which serves as an illustration of the usefulness of the HILARE language to express concrete high-level requirements such as confidentiality in Chapter 3.

Listing A.1 lists the macros and logical functions used to express these requirements in Listing A.3 while Listing A.2 contains the whole C implementation of this case study. In Chapter 4, we semi-automatically verify that this implementation is valid with respect to its specification.

```
meta \macro,
    \name(\forall_page),
    \arg_nb(2),
    \forall int i; 0 <= i < MAX_PAGE_NB ==>
        \let \param_1 = pages + i; \param_2;

meta \macro,
    \name(page_data),
    \arg_nb(1),
    \param_1->data + (0 .. PAGE_SIZE - 1);

meta \macro,
    \name(\constant),
    \arg_nb(1),
    \separated(\written, \param_1);

meta \macro,
    \name(\hidden),
    \arg_nb(1),
    \separated(\read, \param_1);
```

```
meta \macro,
    \name(not_called),
    \arg_nb(1),
    !\fguard(\called == \param_1);


predicate valid_page(struct Page* p) =
    \valid(p) && \valid(p->data + (0 .. PAGE_SIZE - 1)) &&
    0 <= p - pages < MAX_PAGE_NB;

//The given page is filled with zeroes
predicate clean_page(struct Page* p) =
       \forall int i; 0 <= i < PAGE_SIZE ==>
           p->data[i] == 0;

logic enum allocation_status page_status(struct Page* p) = p->status;
logic unsigned page_level(struct Page* p) = p->confidentiality_level;
predicate page_allocated(struct Page* p) = p->status == PAGE_ALLOCATED;
predicate page_lower(struct Page* p, unsigned user_level) =
    user_level > p->confidentiality_level;
```

Listing A.1: Definition of macros in page manager specification (Chapter 3)

```
/**
 * Memory initialization. Every page has a NULL address
 * and 0 conf level.
 * Should be called at the beginning of main (or ideally before).
 */
int init() {
    if(1 + 1 == 2) {
        pages = (struct Page*) malloc(MAX_PAGE_NB * sizeof(struct Page));
    }
  if(pages == NULL)
    return 0;
  /*@
        loop invariant 0 <= i <= MAX_PAGE_NB;
    loop invariant \forall unsigned j; 0 <= j < i ==>
      pages[j].status == PAGE_FREE && pages[j].confidentiality_level == 0;
    loop assigns i, *(pages + (0 .. MAX_PAGE_NB - 1));
  */
  for(unsigned i = 0 ; i < MAX_PAGE_NB ; ++i) {
    char* p = malloc(PAGE_SIZE * sizeof(char));
    if(p == NULL)
      return 0;
    else {
```

```
      pages[i].status = PAGE_FREE;
      memset(p, 0, PAGE_SIZE * sizeof(char));
      pages[i].confidentiality_level = 0;
      pages[i].data = p;
    }
  }
  user_level = 0;
  return 1;
}

/**
 * Returns the first free page it finds or NULL
 */
struct Page* find_free_page() {
  /*@
    loop invariant 0 <= i <= MAX_PAGE_NB;
    loop assigns i;
  */
  for(int i = 0 ; i < MAX_PAGE_NB ; ++i)
    if(pages[i].status == PAGE_FREE)
      return pages + i;
  return NULL;
}

/**
 * Allocates a new page (if there is memory still available) and returns it
 * Its confidentiality level is the current level of the caller.
 */
struct Page* page_alloc() {
  struct Page* fp = find_free_page();
  /*@
    loop invariant 0 <= i <= MAX_PAGE_NB;
    loop invariant \forall unsigned j; 0 <= j < i ==>
      pages[j].status != PAGE_FREE;
    loop assigns i;
  */
  for(unsigned i = 0 ; i < MAX_PAGE_NB ; ++i)
    if(pages[i].status == PAGE_FREE) {
      pages[i].confidentiality_level = user_level;
      pages[i].status = PAGE_ALLOCATED;
      return pages + i;
    }
  return NULL;
}

/**
 * Free a page and erase its content
 * (replacing it by zeroes).
 * No effect if the page is already free.
```

```
 */
void page_free(struct Page* p) {
  if(p != NULL && p->status == PAGE_ALLOCATED) {
    memset(p->data, 0, PAGE_SIZE * sizeof(char));
    p->status = PAGE_FREE;
  }
}

/**
 * Copies PAGE_LENGTH bytes from 'from's data to the buffer
 * if the confidentiality conditions are met
 */
int page_read(struct Page* from, char* buffer) {
  if(from != NULL && from->status == PAGE_ALLOCATED &&
    from->confidentiality_level <= user_level) {
    memcpy(buffer, from->data, PAGE_SIZE);
    return PAGE_OK;
  } else return PAGE_ERROR;
}

/**
 * Copies PAGE_LENGTH bytes from the buffer data to 'to's data
 * if the confidentiality conditions are met
 */
int page_write(struct Page* to, char* buffer) {
  if(to != NULL && to->status == PAGE_ALLOCATED &&
    to->confidentiality_level >= user_level) {
    memcpy(to->data, buffer, PAGE_SIZE);
    return PAGE_OK;
  } else return PAGE_ERROR;
}

/**
 * Raise the confidentiality level if the correct key is passed
 */
unsigned raise_conf_level() {
  return ++user_level;
}

/**
 * Lower the confidentiality level
 */
unsigned lower_conf_level() {
  if(user_level > 0)
    --user_level;
  return user_level;
}

/**
```

```
 * Encrypt the given page in place (using the encrypt primitive)
 * if the confidentiality conditions are met, effectively lowering
 * its confidentiality level to 0
 */
int page_encrypt(struct Page* p) {
  if(p != NULL && p->confidentiality_level > 0
      && p->confidentiality_level == user_level
      && p->status == PAGE_ALLOCATED) {
    encrypt(p->data, user_level, PAGE_SIZE);
    p->encrypted_level = user_level;
    p->confidentiality_level = 0;
    return PAGE_OK;
  }
  else return PAGE_ERROR;
}

/**
 * Decrypt the given page in place (using the decrypt primitive)
 * if the confidentiality conditions are met, effectively restoring
 * its previous confidentiality level
 */
int page_decrypt(struct Page* p) {
  if(p != NULL && p->confidentiality_level == 0
      && p->encrypted_level == user_level
      && p->status == PAGE_ALLOCATED) {
    p->confidentiality_level = user_level;
    decrypt(p->data, user_level, PAGE_SIZE);
    return PAGE_OK;
  }
  else return PAGE_ERROR;
}
```

Listing A.2: Full implementation of the manager, with inline annotations

---

```
/*@
  ensures \result ==> \forall int i; 0 <= i < MAX_PAGE_NB ==>
        \valid(pages + i) &&
    pages[i].status == PAGE_FREE && pages[i].confidentiality_level == 0;
  ensures \result ==> user_level == 0;
*/
int init();

struct Page* page_alloc();

/*@
  behavior valid:
```

```
    assumes p != \null;
    requires valid_page(p);
*/
void page_free(struct Page* p);

/*@
  behavior valid:
    assumes from != \null;
    requires valid_page(from);
    requires \valid(buffer + (0 .. PAGE_SIZE - 1));
    //Buffer unrelated to page fields (status_constant)
    requires \forall int i; 0 <= i < MAX_PAGE_NB ==>
      \separated(pages + i, buffer + (0 .. PAGE_SIZE - 1));
    //Buffer unrelated to pages data (memcpy)
    requires \forall int i; 0 <= i < MAX_PAGE_NB ==>
      \separated(pages[i].data + (0 .. PAGE_SIZE - 1),
          buffer + (0 .. PAGE_SIZE - 1));
*/
int page_read(struct Page* from, char* buffer);

/*@
  behavior valid:
    assumes to != \null;
    requires valid_page(to);
    requires \valid(buffer + (0 .. PAGE_SIZE - 1));
    //'to' data unrelated to page fields (status_constant)
    requires \forall int i; 0 <= i < MAX_PAGE_NB ==>
      \separated(pages + i, to->data + (0 .. PAGE_SIZE - 1));
    //Buffer unrelated to page data (memcpy)
    requires \forall int i; 0 <= i < MAX_PAGE_NB ==>
      \separated(pages[i].data + (0 .. PAGE_SIZE - 1),
          buffer + (0 .. PAGE_SIZE - 1));
*/
int page_write(struct Page* to, char* buffer);

/*@
  assigns *(data + (0 .. size - 1)) \from *(data + (0 .. size - 1));
*/
void encrypt(char* data, unsigned key, unsigned size);

/*@
  assigns *(data + (0 .. size - 1)) \from *(data + (0 .. size - 1));
*/
void decrypt(char* data, unsigned key, unsigned size);

/*@
  behavior valid:
    assumes p != \null;
    requires valid_page(p);
```

```
*/
int page_encrypt(struct Page* p);
/*@
  behavior valid:
    assumes p != \null;
    requires valid_page(p);
*/
int page_decrypt(struct Page* p);

/*@ // Reasonable memory hypotheses as axioms
  axiomatic memory_separation {
    axiom all_sep: \forall struct Page* p; valid_page(p) ==>
      \forall int i; 0 <= i < MAX_PAGE_NB &&  pages + i != p ==>
        \separated(p->data + (0 .. PAGE_SIZE - 1),
            pages[i].data + (0 .. PAGE_SIZE - 1));

    axiom local_sep: \forall int i,j;
      0 <= i < MAX_PAGE_NB ==>
      0 <= j < MAX_PAGE_NB ==> \let p = pages + j;
      \separated(&pages[i].confidentiality_level,
                p->data + (0 .. PAGE_SIZE - 1)) &&
      \separated(&pages[i].status,
                p->data + (0 .. PAGE_SIZE - 1)) &&
      \separated(&(pages[i]).data,
                p->data + (0 .. PAGE_SIZE - 1));

    axiom local_sep2: \forall int i, j;
      0 <= i < MAX_PAGE_NB ==>
      0 <= j < MAX_PAGE_NB ==> \let p = pages + j;
      \separated(&pages[i].encrypted_level,
                p->data + (0 .. PAGE_SIZE - 1));
  }


//========================== METAPROPERTIES ====================

  //Page status is only modified in page_alloc/init/free
  meta \prop,
    \name(status_constant),
    \targets(\diff(\ALL, {init, page_alloc, page_free})),
    \context(\writing), \forall_page(p, \constant(&p->status));

  //Never write to a lower confidentiality page outside of free
  meta \prop,
    \name(confidential_write),
    \targets(\diff(\ALL, {page_free, init})),
    \context(\writing),
      \forall_page(p,
        page_allocated(p) && user_level > page_level(p) ==>
```

```
      \constant(page_data(p))
    );

//Never read from a higher confidentiality page
meta \prop,
  \name(confidential_read),
  \targets(\diff(\ALL, init)),
  \context(\reading),
    \forall_page(p,
      page_allocated(p) && user_level < page_level(p) ==>
      \hidden(page_data(p))
    );

//Free pages are not written upon
meta \prop,
  \name(constant_free_pages),
  \targets(\diff(\ALL, init)),
  \context(\writing),
    \forall_page(p,
      !page_allocated(p) ==> \constant(page_data(p))
    );

//Free pages are not read from
meta \prop,
  \name(hidden_free_page),
  \targets(\diff(\ALL, init)),
  \context(\reading),
    \forall_page(p,
      !page_allocated(p) ==> \hidden(page_data(p))
    );

  //Current confidentiality is only modified
  //through raise/lower_conf_level
meta \prop,
  \name(curconf_wrapped),
  \targets(\diff(\ALL, {raise_conf_level, lower_conf_level, init})),
  \context(\writing), \constant(&user_level);

//Confidentiality modifiers are not called within the library
meta \prop,
  \name(curconf_wrapped_2),
  \targets(\ALL),
  \context(\calling),
    not_called(raise_conf_level) &&
    not_called(lower_conf_level);

//The content of a free page is always null
meta \prop,
  \name(free_page_null),
```

```
    \targets(\diff(\ALL, init)),
    \context(\strong_invariant),
      \forall_page(p, !page_allocated(p) ==> clean_page(p));

  //The confidentiality of an allocated page
    //is constant outside of encrypt/decrypt
    meta \prop,
    \name(constant_conf_level),
    \targets(\diff(\ALL, {init, page_encrypt, page_decrypt})),
    \context(\writing),
      \forall_page(p,
        page_allocated(p) ==> \constant(&p->confidentiality_level)
      );

  //The encryption level is constant outside of encrypt/decrypt
  meta \prop,
    \name(constant_enc_level),
    \targets(\diff(\ALL, {init, page_encrypt, page_decrypt})),
    \context(\writing),
      \forall_page(p, \constant(&p->encrypted_level));

  //The encryption/decryption primitives are
    //only called within page_encrypt and page_decrypt
  meta \prop,
    \name(encdec_uncalled),
    \targets(\diff(\ALL, {page_encrypt, page_decrypt})),
    \context(\calling),
      not_called(encrypt) && not_called(decrypt);

    meta \prop,
        \name(pages_array_allocated),
        \targets(\diff(\ALL, init)),
        \context(\strong_invariant),
        \forall integer i; 0 <= i < MAX_PAGE_NB ==> \valid(pages + i);
*/
```

Listing A.3: Full specification of the manager: contracts and HILAREs

Bibliography

[Gen92] General Accounting Office Washington DC Information Management And Technology Division. "PATRIOT MISSILE DEFENSE: Software Problem Led to System Failure at Dhahran, Saudi Arabia". In: 1992 (cit. on p. 1).

[SJ97] T.H. Soereide and E Jersin. *Sleipner A GBS Loss, Report 16, Quality Assurance*. 1997 (cit. on p. 1).

[Lio96] Jacques-Louis Lions. "ARIANE 5 Flight 501 Failure: Report by the Enquiry Board". In: 1996 (cit. on p. 1).

[Ric53] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Trans. Amer. Math. Soc.* 74 (1953), pp. 358–366 (cit. on p. 2).

[Chu36] Alonzo Church. "A note on the Entscheidungsproblem". In: *Journal of Symbolic Logic* 1.1 (1936), pp. 40–41 (cit. on p. 2).

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265 (cit. on p. 2).

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969) (cit. on p. 3).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of*

Programming Languages. POPL '77. 1977, pp. 238–252 (cit. on p. 3).

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263 (cit. on p. 3).

[Kin76] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394 (cit. on p. 3).

[CR06] Lori A. Clarke and David S. Rosenblum. "A Historical Perspective on Runtime Assertion Checking in Software Development". In: *SIGSOFT Softw. Eng. Notes* 31.3 (2006), pp. 25–37 (cit. on p. 4).

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. 2005, pp. 263–272 (cit. on p. 4).

[Mey92] B. Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51 (cit. on p. 4).

[GDPR16] *Regulation of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)*. European Commission, 2016 (cit. on p. 6).

[Pav+04] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. "Enforcing High-Level Security Properties for Applets". In: *International Conference on Smart Card Research and Advanced Applications*. 2004 (cit. on pp. 7, 162).

[Rob+19a] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. "MetAcsl: Specification and Verification of High-Level Properties". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 11427. LNCS. 2019, pp. 358–364 (cit. on p. 11).

[Rob+19b] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. "MetAcsl : spécification et vérification de propriétés de haut niveau". In: *Actes des 18èmes journées d'Approches Formelles dans l'Assistance au Développement de Logiciels*. AFADL. 2019 (cit. on p. 11).

[Rob+19c] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. "Tame Your Annotations with MetAcsl: Specifying, Testing and Proving High-Level Properties". In: *Tests and Proofs (TAP)*. Vol. 11823. LNCS. 2019, pp. 167–185 (cit. on p. 11).

[Rob+21a] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. "Methodology for Specification and Verification of High-Level Requirements with MetAcsl". In: *IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 2021, pp. 54–67 (cit. on p. 11).

[Rob+21b] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. "Méthodologie pour la validation d'exigences globales appliquée à la sécurité". In: *Actes des 20èmes journées d'Approches Formelles dans l'Assistance au Développement de Logiciels*. AFADL. 2021 (cit. on p. 12).

[DHK21] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. "Formal verification of a JavaCard virtual machine with Frama-C". In: *Formal Methods*. LNCS. 2021, pp. 427–444 (cit. on pp. 12, 166, 168).

[All70] Frances E. Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (1970), pp. 1–19 (cit. on p. 17).

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In: *European Conference on Object-Oriented Programming*. Vol. 1241. LNCS. 1997, pp. 220–242 (cit. on pp. 29, 47, 162).

[Men09] Elliott Mendelson. *Introduction to Mathematical Logic*. 5th. 2009. Chap. 1 (cit. on pp. 37, 39).

[IsoC] *The ANSI C standard (C99)*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf. International Organization for Standardization (ISO) (cit. on pp. 49, 52).

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609 (cit. on p. 50).

[Bau+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. "The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform". In: *Commun. ACM* 64.8 (2021), pp. 56–68 (cit. on p. 50).

[Cuo+09] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. "Experience Report: OCaml for an Industrial-strength Static Analysis Framework". In: *SIGPLAN Not.* 44.9 (2009), pp. 281–286 (cit. on p. 50).

[Bau+20a] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filiâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2020 (cit. on pp. 50, 51, 99, 162).

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. 1997 (cit. on p. 50).

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "JML: A Notation for Detailed Design". In: *Behavioral Specifications of Businesses and Systems*. Vol. 523. The Kluwer International Series in Engineering and Computer Science. 1999, pp. 175–188 (cit. on pp. 50, 161).

[OBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. "Polynomial Invariants by Linear Algebra". In: *Automated Technology for Verification and Analysis*. 2016, pp. 479–494 (cit. on p. 56).

[SG11] Nicolas Stouls and Julien Groslambert. *Vérification de propriétés LTL sur des programmes C par génération d'annotations*. Research Report (French). 2011 (cit. on pp. 56, 162).

[Bau+20b] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *WP plugin manual*. https://frama-c.com/fc-plugins/wp.html. 2020 (cit. on pp. 56, 57, 79).

[SKV17] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. "E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper)". In: *Runtime Verification (RV)*. 2017, pp. 164–173 (cit. on pp. 56, 58, 79, 91).

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 – Where Programs Meet Provers". In: *ESOP'13 22nd European Symposium on Programming*. Vol. 7792. LNCS. 2013 (cit. on pp. 57, 122, 123).

[AltErgo] OCamlPro. *Alt-Ergo, An SMT Solver For Software Verification* (cit. on p. 57).

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. 2008, pp. 337–340 (cit. on p. 57).

[Coq21] The Coq Development Team. *The Coq Proof Assistant*. Version 8.13. 2021 (cit. on pp. 57, 123).

[CP05] Yoonsik Cheon and Ashaveena Perumandla. "Specifying and Checking Method Call Sequences in JML". In: *International Conference on Software Engineering Research and Practice*. 2005, pp. 511–516 (cit. on pp. 68, 161).

[TH02] Kerry Trentelman and Marieke Huisman. "Extending JML Specifications with Temporal Logic". In: *International Conference on Algebraic Methodology and Software Technology*. Vol. 2422. LNCS. 2002, pp. 334–348 (cit. on pp. 68, 161).

[Pet+18] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. "How testing helps to diagnose proof failures". In: *Formal Aspects of Computing* (2018), pp. 629–657 (cit. on p. 92).

[Zei] Steven J. Zeil. *Converting Recursion to Iteration*. URL: https://www.cs.odu.edu/~zeil/cs361/latest/Public/recursionConversion/index.html (cit. on p. 125).

[HLC09] Chung-Yang (Ric) Huang, Chao-Yue Lai, and Kwang-Ting (Tim) Cheng. "CHAPTER 4 - Fundamentals of algorithms". In: *Electronic Design Automation*. 2009, pp. 173–234 (cit. on p. 129).

[JM94] Joxan Jaffar and Michael J. Maher. "Constraint logic programming: a survey". In: *The Journal of Logic Programming* 19-20 (1994). Special Issue: Ten Years of Logic Programming, pp. 503–581 (cit. on p. 132).

[Dov+96] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi. "{log}: A language for programming in logic with finite sets". In: *The Journal of Logic Programming* 28.1 (1996), pp. 1–44 (cit. on p. 134).

[Ben+19] Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. "WooKey: Designing a Trusted and Efficient USB Device". In: *Annual Computer Security Applications Conference (ACSAC)*. 2019, pp. 673–686 (cit. on p. 141).

[Bear] László Nagy. *Build EAR* (cit. on p. 146).

[Mar18] André Maroneze. *Analysis scripts: helping automate case studies*. 2018 (cit. on p. 146).

[Voa97] J. Voas. "Fault injection for the masses". In: *Computer* 30.12 (1997), pp. 129–130 (cit. on p. 153).

[Bau] Patrick Baudin. *ACSL Importer*. URL: https://frama-c.com/fc-plugins/acsl-importer.html (cit. on p. 154).

[Bla+18] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto, and Guillaume Petiot. "Static and Dynamic Verification of Relational Properties on Self-composed C Code". In: *International Conference on Tests and Proofs*. 2018 (cit. on p. 162).

[ZR03] Jianjun Zhao and Martin Rinard. "Pipa: A Behavioral Interface Specification Language for Aspect". In: *Fundamental Approaches to Software Engineering*. 2003, pp. 150–165 (cit. on p. 162).

[Bag+11] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. "Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces". In: *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*. AOSD '11. 2011, pp. 141–152 (cit. on p. 162).

[YW06] Kiyoshi Yamada and Takuo Watanabe. "An Aspect-Oriented Approach to Modular Behavioral Specification". In: *Electronic Notes in Theoretical Computer Science* 163.1 (2006). Proceedings of the First Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2005), pp. 45–56 (cit. on p. 163).

[Ric10] Raymond J. Richards. "Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. 2010, pp. 301–322 (cit. on p. 163).

[Mur+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. "seL4: from General Purpose to a Proof of Information Flow Enforcement". In: *IEEE Symposium on Security and Privacy*. 2013, pp. 415–429 (cit. on pp. 163, 164).

[CSG16] David Costanzo, Zhong Shao, and Ronghui Gu. "End-to-End Verification of Information-Flow Security for C and Assembly Programs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. PLDI '16. 2016, pp. 648–664 (cit. on p. 163).

[Dam+13] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. "Formal Verification of Information Flow Security for a Simple Arm-Based Separation Kernel". In: *ACM Conference on Computer & Communications Security (CCS)*. CCS '13. 2013, pp. 223–234 (cit. on pp. 163, 164).

[Jom+18] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud, and Samuel Hym. "Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base". In: *International Workshop on Automated Verification of Critical Systems (AVOCS)*. 2018 (cit. on pp. 163, 165).

[Les15] Stéphane Lescuyer. "ProvenCore: Towards a Verified Isolation Micro-Kernel". In: *International Workshop on MILS: Architecture and Assurance for Secure Systems, MILS@HiPEAC*. 2015 (cit. on pp. 163, 165).

[Rus81] J. M. Rushby. "Design and Verification of Secure Systems". In: *ACM Symposium on Operating Systems Principles (SOSP)*. SOSP '81. 1981, pp. 12–21 (cit. on p. 163).

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *ACM Symposium on Operating Systems Principles*. 2009, pp. 207–220 (cit. on pp. 164, 166).

[And19] June Andronick. "A Million Lines of Proof About a Moving Target (Invited Talk)". In: *International Conference on Interactive Theorem Proving (ITP)*. Vol. 141. LIPIcs. 2019 (cit. on p. 164).

[Gu+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, 2016, pp. 653–669 (cit. on p. 164).

[Gu+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. "Deep Specifications and Certified Abstraction Layers". In: *ACM Symposium on Principles of Programming Languages (PoPL)*. POPL '15. 2015, pp. 595–608 (cit. on p. 164).

[FM10] Anthony Fox and Magnus O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *International Conference on Interactive Theorem Proving (ITP)*. ITP'10. 2010, pp. 243–258 (cit. on p. 164).

[Bru+11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification (CAV)*. CAV'11. 2011, pp. 463–469 (cit. on p. 165).

[Pit+20] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. "Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs". In: *Automated Reasoning*. Vol. 12167. LNCS. 2020, pp. 119–137 (cit. on p. 165).

[ANS+] ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, and Trusted Labs. "Inter-CESTI: Methodological and Technical Feedbacks on Hardware Devices Evaluations". In: *SSTIC 2020* (cit. on p. 165).

[BBY17] Sandrine Blazy, David Bühler, and Boris Yakobowski. "Structuring Abstract Interpreters Through State and Value Abstractions". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 10145. LNCS. 2017, pp. 112–130 (cit. on p. 165).

[Klee] *KLEE*. URL: https://klee.github.io/ (cit. on p. 165).

[Con+18] Scott D. Constable, Rob Sutton, Arash Sahebolamri, and Steve Chapin. *Formal Verification of a Modern Boot Loader*. Electrical Engineering and Computer Science - Technical Reports 183. Syracuse University, 2018 (cit. on p. 165).

[Str20] Zygimantas Straznickas. "Towards a Verified First-Stage Bootloader in Coq". MA thesis. Massachusetts Institute of Technology, 2020 (cit. on pp. 165, 166).

[EG] Andres Erbsen and Samuel Gruetter. *Language and compiler for verified low-level programming* (cit. on p. 166).

université PARIS-SACLAY | ÉCOLE DOCTORALE Interfaces: matériaux, systèmes, usages

Titre : Spécifier et vérifier des exigences de haut niveau sur des programmes importants : application à la sécurité des programmes C

Mots clés : Méthodes formelles, Vérification déductive, Frama-C, Langage de spécification

Résumé : La spécification et la vérification d'exigences haut niveau (comme des propriétés de sécurité, telles que l'intégrité des données ou la confidentialité) reste un défi pour l'industrie, alors que les cahiers des charges en sont remplis. Cette thèse présente un cadre formel pour les exprimer appelé les *meta-propriétés*, décrites pour un langage de programmation abstrait, et centrées sur les propriétés liées aux manipulations de la mémoire et les invariants globaux. Ce cadre formel est appliqué au langage C avec *HILARE*, une extension d'ACSL, qui permet la spécification d'exigences haut niveau sur des programmes C de grande taille avec facilité.

Des techniques de vérification pour HILARE, basées sur la génération d'assertions locales et la réutilisation des analyseurs de Frama-C existants, sont présentées et implantées dans le greffon *MetAcsl* pour Frama-C. Une méthodologie pour l'évaluation des propriétés de grands programmes est détaillée, articulant les méta-propriétés, les techniques de vérification et les particularités du C. Cette méthodologie est illustrée par un cas d'étude complexe : le bootloader de Wookey, un périphérique de stockage chiffré. Enfin, nous explorons une autre manière de vérifier une exigence de haut niveau en la déduisant à partir d'autres, via un système formel prouvé en Why3 et intégré dans MetAcsl.

Title : Specifying and Verifying High-Level Requirements on Large Programs : Application to Security of C Programs

Keywords : Formal methods, Deductive verification, Frama-C, Specification language

Abstract : Specification and verification of high-level requirements (such as security properties like data integrity or confidentiality) remains an important challenge for the industrial practice, despite being a major part of functional specifications. This thesis presents a formal framework for their expression called *meta-properties*, supported by a description on an abstract programming language, focusing on properties related to memory and global invariants. This framework is then applied to the C programming language, introducing the HILARE extension to ACSL, to allow easy specification of these requirements on large C programs.

Verification techniques for HILARE, based on local assertion generation and reuse of the existing Frama-C analyzers, are presented and implemented into the *MetAcsl* plugin for Frama-C. A complete methodology for assessing large programs is laid out, articulating meta-properties, verification techniques and quirks specific to the C programming language. This methodology is illustrated to a complex case study involving the bootloader of WooKey, a secure USB storage device. Finally, we explore another way to verify a high-level requirement deducing it from others, through a formal system proven in Why3 and integrated in MetAcsl.