Formal Verification of an UAV autopilot: Static analysis and Verified Code Generation

Baptiste Pollien

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut Supérieur de l'Aéronautique et de l'Espace

---

Présentée et soutenue par

Baptiste POLLIEN

Le 16 novembre 2023

Vérification formelle du fonctionnement d'un autopilote: Analyse statique et Génération de code vérifiée

Formal Verification of an UAV autopilot: Static analysis and Verified Code Generation

---

École doctorale :
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité :
Informatique et Télécommunications

Unité de recherche :
ISAE-ONERA MOIS - Modélisation et ingénierie des systèmes

Thèse dirigée par
Xavier Thirioux

Jury
Sandrine Blazy, Université de Rennes, Rapporteure
Jean-Christophe Filliâtre, CNRS, Rapporteur
Timothy Bourke, Inria, Examinateur
Frédéric Dabrowski, Université d'Orléans, Examinateur
Christine Tasson, ISAE-SUPAERO, Examinatrice
Xavier Thirioux, ISAE-SUPAERO, Directeur de thèse

To my uncle Éric
December 1, 1979 - November 09, 2017

Remerciements

French acknowledgments

Je tiens tout d'abord à adresser mes sincères remerciements aux rapporteurs : Sandrine Blazy et Jean-Christophe Filliâtre, ainsi qu'aux membres de mon jury : Timothy Bourke, Frédéric Dabrowski et Christine Tasson, pour avoir accepté d'évaluer mes travaux de thèse, leurs retours positifs sur mon manuscrit et ma présentation, mais aussi pour leurs questions pertinentes lors de ma soutenance de thèse.

Je souhaite ensuite remercier mes nombreux encadrants pour le temps qu'ils ont pu m'accorder et leurs précieux soutiens. Gautier, même si ce n'était pas trop ton domaine, tu as toujours été présent et réactif pour répondre à mes questions, et j'espère que tu auras appris des choses. Pierre, malgré tes contraintes en termes de disponibilité et quelques délais administratifs (2 ans et demi pour que je puisse avoir un badge), j'ai sincèrement apprécié tous tes retours extrêmements intéressants et constructifs.

Je tiens tout particulièrement à remercier Christophe et Xavier que j'ai pu côtoyer quasi quotidiennement. Sans vous, ces 3 années de thèse n'auraient pas été les mêmes ! Je vous remercie d'avoir été disponible tout le long et d'avoir toujours pris le temps aussi bien pour discuter que pour m'aider face aux problèmes que j'ai pu rencontrer. Et même si le début de la thèse a été marqué par la pandémie de Covid-19, vous avez été présent et je ne me suis ni senti impacté (à part peut-être mes premières conférences en distancielles), ni isolé. Votre accompagnement m'a permis d'acquérir des connaissances solides en méthodes formelles et de produire un manuscrit présentant mes travaux de thèse dont je suis vraiment fier ! Vous m'avez également offert de nombreuses opportunités aussi bien en termes d'enseignement que des rencontres lors des différentes écoles d'été et conférences auxquelles vous m'avez permis d'assister. C'était vraiment un plaisir de partager ces 3 ans avec vous. Je garderai en souvenir que des bons moments, même si la dernière période de rédaction a été longue. Je retiendrai les pauses café aussi bien enrichissantes sur le plan scientifique que sur le plan humain. Merci encore pour ces 3 années !

Je tiens également à remercier mes collègues de l'équipe CASC mais aussi toutes les personnes que j'ai pu côtoyer au DISC. En particulier Alexis, Alfonso, Arthur, Benoit, Émilie, et Ghiles avec qui j'ai pu partager le bureau. Lelio pour nos longues discussions scientifiques et sportives. Odile pour son professionnalisme et sa réactivité pour toutes les démarches administratives. Mais aussi Cécile, Éric, Ombeline ainsi que tous les autres les doctorants et post-docs avec qui j'ai pu échanger et que j'aurais pu oublier dans cette liste non-exhaustive. Je remercie également mes deux stagiaires Thomas et Valentin qui ont dû me supporter et qui m'ont permit de faire avancer mes travaux pendant la longue période de rédaction.

Enfin, je tiens à remercier du fond du cœur tous mes proches qui m'ont soutenus de près ou de loin dans mon parcours scolaire et surtout lors de ma thèse. Nicolas pour m'avoir motivé à faire une thèse, en me présentant cette offre et en me vantant les qualités de la ville de Toulouse (tout ça pour finalement que tu partes à Madrid) mais aussi pour le soutien qu'on s'est mutuellement apporté au cours de la thèse et surtout lors de la période de rédaction. Théo et Philémon pour nous avoir permis avec Nicolas de sortir la tête de la thèse et de se changer les idées. Et Eva pour m'avoir remis régulièrement en état après mes nombreuses blessures au cours de la thèse.

Finalement, je tiens à dédier cette thèse à ma famille pour avoir toujours été là pour moi. En particulier, à mes oncles et tantes mais surtout à Éric dont j'aurais préféré qu'il soit encore parmi nous et qui me manque. Mes frère et sur, Lydie et Thomas, dont je suis fière de ce qu'ils sont devenus ! Mes parents, Christelle et Vincent, qui m'ont toujours soutenus et encouragés. Je ne vous remercierai jamais assez pour tout ce que vous m'avez apporté. Mes grands parents, Georgette, Marie-Christine, George (qui n'est malheureusement plus avec nous) et Henri, pour tous les moments qu'on a pu passer ensemble et tous vos encouragements. Enfin, la plus importante pour la fin, Salomé, que jai rencontré au cours de la thèse, qui a été un soutien à toute épreuve, et qui à surtout eu le courage de me supporter pendant mes périodes de stress. Merci infiniment d'avoir été là !

Abstract

Ensuring safety of critical systems is crucial and is often attained by extensive testing of the system. Formal methods are now commonly accepted as powerful tools to obtain guarantees on such systems, even if it is generally not possible to formally prove the safety and correctness of the whole system. This thesis deals with the formal verification of some software components of an autopilot. These components have been selected according to their criticality and should therefore be correct by construction and/or by verification. The goal of this thesis is first to review the formal verification and program proof methods that can be applied to such software in order to apply them on these components of the Paparazzi autopilot developed at ENAC. This thesis also aims to see if the analysis process using such techniques and associated tools can be applied on projects that already exist and are not designed for the verification tools. This thesis focuses on two techniques for the verification of two specific components of Paparazzi.

The first component is a Paparazzi mathematical library verified using the Frama-C platform. This library provides different UAV state representations and associated conversion functions that are used by the drone control system in order to take flight decisions. The Frama-C platform is used with the WP, EVA and RTE plugins to verify the absence of runtime errors in the library and some interesting functional properties on floating-point conversion functions. This verification work required the specification of the correctness properties in the form of function contracts (pre and post condition). The majority of the contracts were automatically verified by the SMT solvers except for some functional properties that must be manually helped using the Coq proof assistant.

The second component verified is a flight plan generator. Paparazzi has a domain specific language called FPL, used to specify flight plans. It is a programming and modelling language allowing the expression of complex missions. FPL missions are compiled into C code that is directly embedded into the autopilot code. The FPL to C code generator, currently written in OCaml, is therefore a critical component when addressing the drone safety. This thesis formally verifies the FPL compilation process. First, three-pass code generator, targeting the Clight intermediate language from the CompCert suite has been developed in Coq. Then, an operational semantics have been formally defined. Finally, the generator has been formally verified by manually proving a bisimulation relation between FPL semantics and Clight semantics. In the course of the formalization and verification process, we have also unveiled several problems in the original Paparazzi code generator.

Keywords: Formal methods, Verification and validation, Cyberphysical systems, Proof of program, Mechanized proof, Static analysis.

Contents

Résumé en Français **19**
i Introduction . . . 20
ii Analyse Statique de code en utilisant Frama-C . . . 22
ii.1 Absence d'erreurs à l'exécution . . . 23
ii.2 Vérification fonctionnelle de propriétés avec les prouveurs automatiques 24
ii.3 Vérification fonctionnelle de propriétés avec les prouveurs manuels . 25
ii.4 Discussion . . . 27
iii Verification de compilateur avec Coq . . . 28
iii.1 Langage de description de plan de vol . . . 29
iii.2 Générateur . . . 39
iii.3 Les leçons apprises . . . 46
iv Conclusion . . . 48
iv.1 Une bibliothèque mathématique pour la représentation d'états . . . 49
iv.2 Générateur de plan de vol . . . 50
v Conclusion générale et travaux futurs . . . 53

1 Introduction **57**
1.1 The Paparazzi UAV autopilot . . . 58
1.2 A mathematical library for state representation conversion . . . 60
1.3 Flight Plan Generator . . . 61

2 State of the art: formal verification **65**
2.1 Code verification . . . 66
2.1.1 Properties specification . . . 66
2.1.2 Code analysis . . . 69
2.2 Introduction to formal semantics . . . 71
2.2.1 `SIL`: Simple Imperative Language . . . 72
2.2.2 Operational semantics . . . 74
2.2.3 Denotational semantics . . . 78
2.2.4 Axiomatic semantics . . . 83
2.3 Proof assistants . . . 87
2.4 Related work . . . 90
2.4.1 Static code analysis . . . 91
2.4.2 Compiler verification . . . 92

I Static code analysis using Frama-C **95**

3 Introduction to the Frama-C plateform **97**
3.1 Frama-C platform . . . 97
3.1.1 Specifying with ACSL . . . 98
3.1.2 Verifying with EVA or WP . . . 99
3.2 How to verify code with Frama-C: A simple process . . . 101
3.2.1 Absence of runtime errors . . . 101
3.2.2 Functional verification . . . 103

4 Verifying the Mathematical Library of a UAV Autopilot with Frama-C **107**
4.1 Proving the absence of runtime errors . . . 107
4.2 Functional verification using automatic provers . . . 109
4.3 Functional verification using interactive provers . . . 112
4.4 Discussion . . . 113

II Verified compiler in Coq **117**

5 Verified compilation using Coq **119**
5.1 The concept of semantic preservation . . . 120
5.1.1 Generic notion of semantic preservation . . . 120
5.1.2 Simulation diagrams for transition semantics . . . 121
5.2 Introduction to Clight Semantics . . . 123
5.2.1 Clight Syntax . . . 123
5.2.2 Small-step semantics of Clight . . . 126
5.2.3 Execution of a Clight program . . . 134

6 Introduction to the Verified Flight Plan Generator **137**
6.1 FPL: Flight Plan Language . . . 138
6.2 Global architecture of the new generator . . . 139
6.2.1 Preprocessor . . . 140
6.2.2 Postprocessor . . . 141

7 Specification of the flight plan language **143**
7.1 Syntax of FPL . . . 143
7.1.1 Notations and informal definitions . . . 144
7.1.2 FPL definition . . . 145
7.1.3 Default block for FP . . . 147
7.1.4 Block index problem . . . 149
7.2 A flight plan execution example . . . 149
7.3 FP semantics . . . 151
7.3.1 States and traces . . . 152
7.3.2 Signatures of the semantics functions . . . 155

7.3.3 Inference rules describing flight plan semantics . . . 159

8 New features added to FPL 169
8.1 Syntax of the new features . . . 169
8.2 Semantics of the new features . . . 170
8.2.1 Forbidden deroutes semantics . . . 170
8.2.2 Updated semantics for block change . . . 172

9 Architecture of the Gallina generator 175
9.1 Extension pass . . . 177
9.1.1 Syntax of FPE . . . 178
9.1.2 FPE environment . . . 181
9.1.3 Well-formed extended flight plan . . . 183
9.1.4 FPE semantics . . . 187
9.2 Size verification pass . . . 199
9.2.1 Well-sized flight plan . . . 199
9.2.2 Size verification function . . . 201
9.3 Clight generation function . . . 202
9.3.1 Number of blocks in a global variable . . . 202
9.3.2 Auxiliary functions generated . . . 203
9.3.3 The auto_nav function . . . 205
9.4 Generator function . . . 208
9.4.1 Global Variables generation . . . 208
9.4.2 Forbidden deroute analysis . . . 210
9.4.3 Flight plan generator function . . . 211

10 Verification of the Flight Plan Generator 213
10.1 Generic definition of flight plans semantics . . . 213
10.2 Semantic preservation verification . . . 219
10.2.1 Framework for semantic preservation . . . 219
10.2.2 Extension pass verification . . . 221
10.2.3 Verification of the size verification pass . . . 223
10.2.4 Verification of the Clight generation pass . . . 224
10.2.5 VFPG correctness theorem . . . 228
10.3 Termination proof of the auto_nav function . . . 229
10.4 Verification hypotheses . . . 229
10.4.1 Models of the system . . . 230
10.4.2 New semantic rules . . . 231
10.4.3 Axioms used for the proof . . . 235

11 Discussion 237
11.1 Development methodology . . . 237
11.2 Technical remarks . . . 238
11.2.1 Design choices made during development . . . 238

11.2.2 Use of Clight 239
11.2.3 Arbitrary C code 239
11.2.4 MathComp Library 242
11.3 Applicability on real-world projects 242

12 Conclusion **245**
12.1 A mathematical library for state representation 245
12.2 Flight Plan Generator 246
12.3 General conclusion and future work 249

Appendixes **255**

A Flight Plan C code **255**
A.1 C code generated Example 255
A.1.1 XML structure of the flight plan 255
A.1.2 Corresponding `FP` structure 256
A.1.3 C code generated 257
A.2 Common flight plan functions 261
A.2.1 Header file for the common functions 261
A.2.2 Source file for the common functions 263

Bibliography **267**

List of Figures

1 La structure Gallina `Flight Plan` (`FP`) représentant `FPL`. 30
2 Exemple de l'exécution d'un plan de vol. 31
3 Règles d'inférence pour `FP`. 36
4 La fonction C `auto_nav` générée à partir d'un fichier `FPL` simple. 38
5 Architecture du générateur. 39
6 Nombre de lignes de code et annotations ajoutées. 50
7 Lignes de code pour les différents blocs du générateur. 51
8 Lignes de code Coq dans le générateur Gallina (commentaires et espaces exclus). 52

1.1 Architecture of the Paparazzi autopilot (embedded code) 60

2.1 Big-step semantics of `SIL` 75
2.2 Small-step semantics of `SIL` 76
2.3 Principle of abstract interpretation 80
2.4 The Floyd-Hoare inference rules 84
2.5 Example of two Coq proofs. 89

3.1 Diagram about Frama-C 100

4.1 (`float_mat_sum`, `sw/airborne/math/pprz_algebra_float.h:1340`) 108
4.2 (`float_rmat_of_quat`, `sw/airborne/math/pprz_algebra_float.h:639`) . . . 111

5.1 Bisimulation diagrams 122
5.2 Example of a C `for` loop converted into Clight. 125
5.3 Inference rules for sequence and assignment 129
5.4 Inference rules for conditional and loop statements 130
5.5 Inference rules for switch and goto 131
5.6 Inference rules for calls 132
5.7 Inference rules for the return statement 133

6.1 Current Generator integrated in Paparazzi 137
6.2 Simple example of an XML flight plan 138
6.3 Global architecture of the generator. 139

7.1 Example of an execution of a flight plan. 150
7.2 Gallina code for the *step* function 160
7.3 C code generated for the navigation *mode* 167

9.1 The C `auto_nav` function generated from a simple `FPL` plan 176
9.2 Architecture of the Coq Generator. 177
9.3 Example of stages extension. 180
9.4 Example of fp_stage_e list. 184
9.5 Modifications brought to the function `nav_goto_block`. 203
9.6 The C `on_enter_block` function generated from a simple flight plan. 204
9.7 The C `forbidden_deroute` function generated from a simple flight plan. . . 205
9.8 The C code generated for the evaluation of a exception. 208

12.1 Lines of code and annotations added. 246
12.2 Lines of Code for the different generator blocks. 247
12.3 Lines of Coq code in the Gallina generator (comments and blanks excluded). 248

Résumé en Français

This thesis begins with a summary in French. The rest of this document is written in English, and starts at page 57.

Abstract

Assurer la sécurité des systèmes critiques est crucial et est souvent atteint par des tests approfondis du système. Les méthodes formelles sont maintenant couramment acceptées comme des outils puissants pour obtenir des garanties fortes sur de tels systèmes, même s'il n'est généralement pas possible de prouver formellement la sûreté et la correction de l'ensemble du système. Cette thèse traite de la vérification formelle de certains composants logiciels d'un autopilote de drone. Ces composants ont été sélectionnés en fonction de leur niveau de criticité et doivent donc être corrects par construction et/ou par vérification formelle. L'objectif de cette thèse est de passer en revue différentes méthodes de vérification et de preuve de programmes qui peuvent être appliquées à de tels logiciels et de les utiliser sur un cas d'étude, l'autopilote Paparazzi développé à l'ENAC. Cette thèse vise également à déterminer si les processus de vérification utilisant ces techniques et les outils associés peuvent être appliqués sur des projets déjà existants et non conçus pour les outils de vérification. Cette thèse s'est concentrée sur l'application de deux techniques de vérification formelle sur deux composants spécifiques de Paparazzi.

Le premier composant est une bibliothèque mathématique que nous avons vérifiée partiellement à l'aide de la plateforme Frama-C. Cette bibliothèque fournit différentes représentations d'états de l'autopilote et des fonctions de conversions associées qui sont utilisées par le système de contrôle du drone afin de prendre des décisions de vol. La plateforme Frama-C est utilisée afin de vérifier à la fois l'absence d'erreurs à l'exécution dans les fonctions de la bibliothèque et quelques propriétés fonctionnelles intéressantes sur des fonctions de conversion en virgule flottante. La majorité des propriétés ont été vérifiées automatiquement et quelques propriétés fonctionnelles ont dû être prouvées manuellement avec l'assistant de preuve Coq.

Le deuxième composant vérifié est un générateur de plan de vol. Paparazzi propose un langage de spécification de plans de vol appelé FPL qui est Turing complet. Les missions exprimées en FPL sont compilées en code C qui est directement embarqué dans le code de l'autopilote. Le générateur de code FPL vers C, actuellement écrit en OCaml, est donc un composant critique pour la sureté des drones. Nous avons vérifié dans cette thèse le processus de compilation de FPL vers C en écrivant tout d'abord dans l'assistant de preuve Coq un générateur de code à trois passes, ciblant le langage intermédiaire Clight. La sémantique opérationnelle de FPL a été formellement définie et le générateur a été formellement vérifié en prouvant une relation de bisimulation entre la sémantique de FPL et celle de Clight toujours dans l'assistant de preuve Coq. Au cours du processus de formalisation et de vérification, nous avons également dévoilé plusieurs problèmes dans le générateur originel de Paparazzi et nous les avons résolus.

i Introduction

La vérification est une étape cruciale lors du processus de développement d'un système ou d'un programme, en particulier dans un contexte critique, pour prévenir des événements dramatiques tels que l'explosion de la fusée Ariane 5 [LLF+96]. Cette phase de vérification garantit que le système respecte sa spécification, et en particulier que des comportements indésirables ne se produiront jamais. Historiquement, la principale technique de vérification utilisée est le test : considérez plusieurs entrées pour le système et vérifiez que chacune d'entre elles produit les résultats attendus. Cependant, comme l'a écrit Edsger Disjktra [Dij72] :

«*Les tests de programme peuvent être un moyen très efficace de montrer la présence de bogues, mais ils sont complètement inadéquats pour démontrer leur absence.*»

Les méthodes formelles sont des théories, des techniques et des outils mathématiques permettant de prouver formellement les propriétés du matériel, des logiciels ou des modèles. Elles offrent des garanties plus fortes que les tests, car elles peuvent garantir que le système satisfait une propriété au lieu de simplement vérifier si certaines entrées respectent cette propriété. Il existe de nombreux domaines de méthodes formelles, et ils peuvent être classés en fonction des propriétés qu'ils peuvent aider à vérifier, des efforts nécessaires pour spécifier le système afin d'utiliser les outils de vérification, ou de leur niveau d'automatisation. Par exemple, l'*interprétation abstraite* est souvent utilisée pour prouver l'absence d'erreurs d'exécution et est un outil automatique. La *vérification déductive* est une autre technique qui peut être utilisée pour prouver des propriétés plus complexes, telles que la correction d'un programme par rapport à une spécification formelle, et elle utilise généralement des solveurs automatisés, bien qu'elle puisse parfois nécessiter l'utilisation d'un assistant de preuve et l'intervention humaine.

Les méthodes formelles sont aujourdhui largement acceptées comme des méthodes de verification complémentaires aux tests, en particulier pour les systèmes critiques tels que dans l'aérospatiale [KWN+10], l'automobile, le médical et la cybersécurité [Jae10]. Il existe désormais des normes de suretés qui recommandent lutilisation de méthodes formelles (par exemple, la norme DO-333 qui complète le document DO-178C utilisé par les autorités de certification pour les systèmes d'avions commerciaux). Cependant, la définition des processus de vérification pour les usages industriels est en général complexe en raison de leurs multiples contraintes. En effet, les processus doivent utiliser des outils de vérification qui peuvent s'adapter à de grands projets qui ne sont pas spécialement conçu pour ces outils. De plus, les processus peuvent être appliqués par des ingénieurs qui ne sont pas nécessairement des experts en méthodes formelles.

Paparazzi [HBG14, Pap21] est un logiciel libre de pilotage automatique pour drone sous licence GPL développée à l'ENAC (École Nationale de l'Aviation Civile) depuis 2003. Paparazzi est un système de contrôle complet pour véhicules autonomes. Paparazzi offre une partie logicielle de contrôle et quelques designs de composants matériels. Ce pilote automatique prend en charge différents types de drones (drones quadricoptères, avions à voilure fixe, rovers...) et permet de contrôler plusieurs d'entre eux simultanément. Paparazzi a également divers modes intégrés et offre la possibilité de créer des plans vols personnalisés.

Plusieurs composants essentiels de Paparazzi sont critiques car un dysfonctionnement de l'un d'eux peut entraîner des comportements indésirables voire même le crash du drone. Afin que les utilisateurs aient une grande confiance en Paparazzi, il est essentiel de s'assurer que ses composants sont corrects et qu'ils ne présentent pas de risques de défaillances.

L'objectif de cette thèse est de passer en revue différents processus de vérification en utilisant des outils formels afin de vérifier les propriétés de correction de certains composants critiques de Paparazzi. En plus de fournir des composants formellement vérifiés aux utilisateurs de Paparazzi, cette thèse vise à vérifier si les processus peuvent être utilisés sur des projets existants. En effet, Paparazzi constitue une excellente étude de cas, avec une grande base de code (350 000 lignes de code) écrite sans objectif de vérification par des programmeurs expérimentés qui utilisent des idiomes classiques du langage de programmation C (pointeurs, unions...).

Remarque

Ce chapitre est un résumé en français du manuscrit de thèse entièrement rédigé en anglais. Ce résumé est autosuffisant, mais certains éléments de la thèse ne seront pas détaillés ou ne seront pas présentés. En revanche, afin d'accéder à tous ces détails facilement, des références vers le contenu en anglais ont été ajoutées dans ce résumé. Dans la suite, nous utiliserons des conventions de numérotation afin de différencier les parties en français et en anglais. Les sections en chiffres romains minuscules (ex : Section ii) correspondent aux sections du résumé en français, et les sous-sections utilisent des chiffres arabes (ex : ii.2). Dans le reste du document en anglais, les chiffres romains majuscules sont utilisés pour la numérotation des parties (ex : Partie II), les chiffres arabes sont utilisés pour la numérotation des chapitres, des sections et des sous-sections (ex : Chapitre 9, Section 11.3...), et les lettres sont utilisées pour les annexes (ex : Annexe A.2).

Ce résumé est organisé comme suit. Il commence par deux sections indépendantes :

- La Section ii présente le travail réalisé pour vérifier une bibliothèque mathématique écrite en C en utilisant des techniques d'analyse statique de code.
- La Section iii présente la vérification formelle d'un générateur de plan de vol à l'aide d'un assistant de preuve.

Ces deux sections sont des résumés des deux parties du manuscrit, respectivement la Partie I et la Partie II. Enfin, la Section iv conclut le travail réalisé au cours de la thèse. Il est important de noter que différentes section et chapitres ne seront pas présentés dans ce résumé, dont: la Section 1.1 qui présente l'autopilote Paparazzi; le Chapitre 2 du manuscrit, qui présente un état de l'art sur les méthodes formelles avec une introduction sur les processus de vérification et sur les sémantiques formelles utilisées par les méthodes formelles; le Chapitre 5 qui présente les techniques de vérification utilisées dans la littérature pour vérifier formellement les compilateurs mais aussi la syntaxe et la sémantique de Clight[1],

[1]Clight est la représentation en Gallina du code C fournie par CompCert et utilisée par le nouveau

Remarque d'implementation

Chaque projet développé au cours de cette thèse est publiquement accessible dans les deux projets GitLab suivants :

- Bibliothèque math. : gitlab.isae-supaero.fr/b.pollien/paparazzi-frama-c
- Générateur de plan de vol : gitlab.isae-supaero.fr/b.pollien/vfpg

Tout au long de ce document, nous fournissons des liens cliquables qui font directement référence au fichier ou à la définition correspondant dans le projet GitLab. Des liens cliquables ont également été ajoutés pour chaque fonction, type ou opérateur qui font référence à sa définition ou notation originale.

ii Analyse Statique de code en utilisant Frama-C

Les pilotes automatiques de drones utilisent toute une gamme de capteurs pour collecter des données sur l'environnement du drone, comme par exemple le GPS, les accéléromètres ou les gyroscopes. Ensuite, le pilote automatique utilise ces données pour prendre des décisions sur la manière de contrôler le drone afin d'accomplir sa mission. La précision et la fiabilité des données sont donc cruciales, car les décisions prises par le pilote automatique doivent être cohérentes avec son environnement réel.

Dans Paparazzi, les données utilisées par le système de navigation sont acquises via une «Interface d'État» qui est connectée à des filtres d'estimation et à des capteurs. Cette interface d'état est une interface qui offre différentes représentations des données collectées et effectue automatiquement des conversions en fonction des besoins du pilote automatique. Les fonctions de représentation et de conversion sont définies dans une bibliothèque mathématique implémentée en langage C. Cette bibliothèque est donc souvent utilisée et ne doit pas contenir de bogues pouvant, par exemple, provoquer le plantage du programme ou générer des données invalides.

Dans cette thèse, nous nous sommes concentrés sur une partie importante de la bibliothèque mathématique C de Paparazzi qui fournit un modèle de l'attitude du drone à travers différentes représentations : matrices de rotation, angles d'Euler ou quaternions. La bibliothèque définit également des opérations élémentaires sur ces représentations. De plus, elle définit trois versions de chaque représentation et des fonctions associées : une utilisant des valeurs en `double`, une autre avec des valeurs en `float` et la dernière utilisant des valeurs en `int` pour représenter des valeurs en virgule fixe. Cette bibliothèque est une partie intéressante à vérifier car elle contient un code complexe mais critique.

Frama-C [KKP+15] est un outil d'analyse de code C qui nécessite l'ajout d'annotations dans le code, avec le langage ACSL (*ANSI C Specification Language*), pour spécifier les propriétés attendues. Ces annotations peuvent être par exemple des assertions ou des

générateur de plans de vol.

spécifications de contrats pour des fonctions. Un contrat définit les *pré-conditions* et les *post-condition* que doit respecter la fonction. Frama-C dispose de greffons comme WP (*Weakest Precondition* qui utilise des un calcul de plus faible préconditions) ou EVA (*Evolved Value Analysis* qui utilise l'interprétation abstraite) qui permettent de vérifier que les assertions ne sont pas violées et que les contrats sont respectés. Le Chapitre 3 présente plus en détail Frama-C et le processus utilisé dans cette thèse pour vérifier du code C.

Les erreurs à l'exécution, ou *runtime errors* en anglais, sont des erreurs qui provoquent généralement des comportements inattendus malgré un algorithme correct. Elles sont principalement liées à des effets de bord pendant l'exécution. Elles peuvent être causées par un accès mémoire illégal, un débordement lors d'un calcul ou par une division par 0. Le greffon RTE (*RunTime Errors*) de Frama-C ajoute des assertions qui correspondent à l'absence de ce type d'erreur. La preuve que les assertions ajoutées ne sont pas violées permet donc de garantir qu'il n'y aura pas d'erreurs à l'exécution. Par contre, il est important de noter que cette preuve n'offre aucune garantie fonctionnelle.

Dans le cadre de cette thèse, la vérification de la bibliothèque mathématique a été effectuée à l'aide de la plateforme Frama-C et a été réalisée principalement à l'aide de prouveurs automatiques. La Section ii.1 détaille l'analyse concernant l'absence d'erreurs d'exécution. La deuxième partie de l'analyse couvre la vérification des propriétés fonctionnelles de certaines fonctions de conversion entre les représentations d'état. La Section ii.2 détaille le processus de vérification en n'utilisant que des prouveurs automatiques. Enfin, comme les prouveurs automatiques n'ont pas pu prouver certaines de ces fonctions, la Section ii.3 présente comment nous avons prouvé de telles fonctions à l'aide du prouveur interactif Coq [The23]. Enfin, la Section ii.4 discute des leçons apprises pendant le travail et présente certaines perspectives.

ii.1 Absence d'erreurs à l'exécution

La bibliothèque définit des structures C pour représenter les données manipulées (matrices de rotation, quaternions, vecteurs...). Les fonctions de la bibliothèque travaillent uniquement par référence pour les entrées et pour les sorties. Afin d'éviter des erreurs liées à un déréférencement de pointeurs non valides, des préconditions garantissant la validité des références ont été ajoutées dans les contrats des fonctions. Il a aussi été nécessaire de spécifier les variants et invariants de boucle ainsi que les variables de sorties comme seuls espaces mémoire (hors pile) qui seront modifiées.

WP dispose de différents modèles arithmétiques qui prennent en compte de façon plus ou moins précise la sémantique de C. La vérification de la bibliothèque sur les entiers a été faite en utilisant le modèle réaliste de l'arithmétique machine des entiers. Avec le greffon RTE, il est alors nécessaire de vérifier qu'il n'y a pas de débordement de valeur (ou *overflow* en anglais) pour chaque opération arithmétique. Pour vérifier l'absence de dépassement, chaque fonction a été analysée dans l'objectif de déterminer les bornes maximales possibles des différentes variables. Lorsque ces bornes ont pu être déterminées, elles ont été ajoutées en préconditions dans les contrats des fonctions. Le processus utilisé pour déterminer les bornes est détaillé en Section 3.2.1.

Malheureusement, WP associé à des prouveurs automatiques n'est pas parvenu à vérifier ces nouveaux contrats. L'utilisation de références pour l'accès aux valeurs numériques surcharge les prouveurs et ce même en spécifiant en précondition que les structures en paramètres sont stockées à des emplacements mémoire séparés.

Pour pallier ce problème, nous avons décidé d'associer EVA à WP. EVA arrive à calculer des intervalles suffisament précis des valeurs possibles pour chaque variable. Ce résultat est ensuite transmis à WP par Frama-C ce qui permet de conclure plus facilement les preuves. Cette limitation de WP avait aussi été notée par Vassil Todorov durant sa thèse [Tod20] et il avait également utilisé un outil d'analyse statique par interprétation abstraite, Astrée, pour résoudre ce problème. En conclusion, lorsque l'on associe EVA avec WP, il est possible de vérifier l'absence d'erreur à l'exécution des fonctions de la bibliothèque sur les entiers.

WP dispose également d'un modèle arithmétique `real` qui correspond à l'arithmétique réelle au sens mathématique. Pour la vérification des versions de la bibliothèque travaillant sur des valeurs numériques à virgule flottante (aussi bien `float` ou bien `double`), nous avons décidé d'utiliser ce modèle. Il nous a permis de vérifier l'absence de division par 0 et que les variables ne prennent pas la valeur `NaN` (*Not A Number*). Pour effectuer ces vérifications, il a été seulement nécessaire d'ajouter comme préconditions que chaque valeur numérique passée en paramètre ne prend pas la valeur *NaN* et qu'elle n'est pas infinie. Là encore, l'absence de ces deux erreurs à l'exécution et la terminaison des fonctions ont été prouvées pour les versions `float` et `double` de la bibliothèque en utilisant WP et EVA. Par contre, notre vérification n'offre aucune garantie sur le risque de dépassement ou sur les erreurs liées aux arrondis. Cependant, ce modèle nous a été particulièrement utile pour la vérification des propriétés fonctionnelles présentée en section ii.2.

Remarque

La Section 4.1 présente en détail le travail réalisé pour vérifier l'absence d'erreur à l'exécution dans cette bibliothèque mathématique.

ii.2 Vérification fonctionnelle de propriétés avec les prouveurs automatiques

La vérification fonctionnelle permet de garantir les résultats attendus d'une fonction. Dans notre cas d'étude nous avons décidé de vérifier certaines propriétés fonctionnelles sur des fonctions comme `float_rmat_of_quat`. Cette fonction prend en paramètre un quaternion normalisé et retourne la matrice de rotation correspondante.

Afin de spécifier les propriétés fonctionnelles, des types et des prédicats ont été définis dans la logique fournie par le language ACSL. On retrouve la définition des types pour les matrices et les quaternions ainsi que des opérations élémentaires. Des lemmes ont ensuite été spécifiés et vérifiés pour garantir que ces opérations sont correctes. Une fonction logique qui convertit un quaternion en une matrice de rotation a également été définie, indépendamment du code C. Cette fonction est basée sur la formule mathématique de conversion d'un quaternion vers une matrice de rotation [Gru70, Klu76].

À partir de ces définitions, le contrat de la fonction a pu être établi. En supposant que le quaternion passé en paramètre est normalisé, nous avons voulu vérifier 2 propriétés

fonctionnelles. La première est que la matrice retournée correspond bien à la conversion du quaternion passé en paramètre: notre post-condition vérifie que la matrice de rotation générée par le code C est égale à la matrice de rotation générée par notre fonction logique. Comme dans la section précédente, nous utilisons le modèle `real` de WP pour la vérification de cette fonction, ce qui permet d'ignorer les différences de résultats entre la version C et la version mathématique qui auraient pu être liées à des erreurs d'arrondis. La seconde propriété vérifie que la matrice générée est bien une matrice de rotation, i.e. que la transposée de la matrice est son inverse.

Malgré l'utilisation du modèle `real` d'arithmétique, WP n'arrivait pas à vérifier le contrat. Il a donc été nécessaire d'analyser le code. Nous avons remarqué que le code C utilisait une constante `M_SQRT2` pour représenter $\sqrt{2}$ et qu'en simplifiant les calculs, la constante `M_SQRT2` était tout le temps multipliée à elle-même. Nous avons donc suggéré une modification du code en remplaçant ces opérations par une multiplication par 2. Cette modification ne change pas le nombre de multiplications mais permet de réduire les erreurs d'arrondis propagées par la fonction. Avec ces modifications de code et avec le modèle `real` pour l'arithmétique, WP vérifie le contrat de la fonction `float_rmat_of_quat`. Ce contrat garantit bien l'absence d'erreur à l'exécution et définit les propriétés fonctionnelles attendues. Ces propriétés permettent de vérifier uniquement le comportement idéaliste de la fonction sans considérer les erreurs potentielles de calcul.

Remarque

La Section 4.2 détaille cette étape de vérification fonctionnelle en utilisant les solveurs automatiques et donne un exemple de lemme utilisé pour s'assurer que les définitions mathématiques sont correctes.

ii.3 Vérification fonctionnelle de propriétés avec les prouveurs manuels

La vérification a été tentée pour la fonction inverse `float_quat_of_rmat`. Cette fonction convertit une matrice de rotation en un quaternion en utilisant quatre formules de la méthode de Shepperd [She78, ST19]. Ces formules sont déduites de la formule de conversion d'un quaternion en une matrice de rotation et dépendent de la trace de la matrice de rotation: l'une est définie lorsque la trace est strictement positive, les trois autres sont définies lorsque la trace est négative et correspondent aux choix possibles pour l'élément le plus élevé de la diagonale de la matrice. Chacune de ces formules a été définie par une fonction logique ACSL, avec des préconditions spécifiques. Ces préconditions garantissent que le quaternion calculé par la fonction C correspond au quaternion calculé en utilisant l bonne fonction logique correspondante. De plus, Frama-C permet de vérifier que les comportements sont disjoints et complets, assurant ainsi que pour chaque matrice d'entrée, il existe un seul comportement dont les préconditions sont satisfaites. Grâce à ce contrat, la fonction a été vérifiée pour produire un quaternion correspondant à la même rotation que la matrice d'entrée.

Désignons par `quat_of_rmat` la fonction mathématique qui renvoie le quaternion correspondant à une matrice de rotation donnée. Considérons ici le cas où la matrice de rotation

utilisée en entrée a une trace positive. Vérifier que `quat_of_rmat` est correct dans ce cas équivaut à vérifier que la propriété décrite dans le lemme suivant est vérifiée :

Lemme ii.1 - (`correctness_quat_of_rmat_pos`, `sw/airborne/math/pprz_algebra_float_func.h:644`).

$$\forall R \in M_{3,3}(\mathbb{R}),\ q \in \mathbb{H},\ \ \|q\| = 1 \ \wedge\ Tr(R) > 0 \\ \rightarrow (R = \texttt{rmat\_of\_quat}(q) \leftrightarrow q = \texttt{quat\_of\_rmat}(R))$$

Ce lemme peut être interprété comme suit : pour toutes les matrices de rotation R avec une trace positive et pour tous les quaternions unitaires q, R représente la matrice de rotation obtenue à partir de $\texttt{rmat\_of\_quat}(q)$, si et seulement si la fonction `quat_of_rmat` renvoie q lorsqu'elle est appliquée à R. Il a ensuite été traduit en un lemme ACSL, ainsi que les équations similaires découlant des trois autres cas.

Malheureusement, Frama-C n'est pas en mesure de prouver ces lemmes en utilisant uniquement des solveurs SMT automatiques, même après avoir considérablement augmenté la valeur du délai d'attente (*timeout* en anglais). La preuve nécessite des transformations spécifiques, telles que la factorisation, que les solveurs pourraient ne pas être en mesure de trouver. Nous avons donc dû utiliser le mode interactif de WP. Ce mode génère des scripts de preuve incomplets pour chaque objectif non prouvé. Les scripts contiennent toutes les définitions et les lemmes qui ont déjà été prouvés par Frama-C et les solveurs. Le théorème correspondant à un objectif non prouvé doit être vérifié avec un prouveur interactif, dans notre cas Coq (voir Section 2.3). L'implication selon laquelle si R est obtenu à partir de $\texttt{rmat\_of\_quat}(q)$, alors la fonction `quat_of_rmat` renvoie q a été vérifiée avec Coq pour les quatre lemmes (par exemple `lemma_impl_rmat_quat_trace_pos.v` pour le cas positif). L'implication inverse n'a pas encore été prouvée. Cependant, en considérant la vérification de la fonction `rmat_of_quat`, cette preuve est suffisante pour garantir que le résultat de la fonction `quat_of_rmat` décrit la même rotation que la matrice d'entrée.

Nous avons souhaité vérifier la fonction de conversion de la représentation d'Euler en matrice de rotation. Deux fonctions, `float_rmat_of_eulers_321` et `float_rmat_of_eulers_312`, effectuent cette conversion, différenciées par l'ordre des angles d'Euler. Les contrats définis garantissent la spéciale orthogonalité de la matrice, c'est-à-dire qu'elle est une matrice de rotation. Pour vérifier ces contrats, nous avons d'abord utilisé des solveurs SMT automatiques. Cependant, le code de conversion utilise les fonctions trigonométriques de la bibliothèque standard C, et les contrats intégrés de Frama-C pour ces fonctions ne fournissent pas suffisamment d'informations. Pour résoudre cela, nous avons ajouté une hypothèse au contrat pour établir l'équivalence entre les fonctions standard et mathématiques d'ACSL. Bien que cette hypothèse puisse ne pas être correcte, elle est justifiée par l'utilisation du modèle `real` pour tirer profit des propriétés des fonctions trigonométriques.

Remarque

La Section 4.3 explique cette étape de vérification fonctionnelle en utilisant des prouveurs interactifs, en détaillant plus précisément la vérification des fonctions `float_rmat_of_eulers_321` et `float_rmat_of_eulers_312`.

ii.4 Discussion

Dans ce chapitre, nous avons présenté un travail sur la vérification formelle d'une bibliothèque mathématique à l'aide de Frama-C. Nous nous sommes principalement concentrés sur la vérification de l'absence d'erreurs à l'exécution, mais nous avons également prouvé des propriétés intéressantes de fonctions plutôt complexes. Cependant, ce travail n'est pas achevé car la vérification de la bibliothèque mathématique n'est pas terminée. Plusieurs perspectives sont brièvement détaillées ci-dessous.

Tout d'abord, certaines preuves restent à réaliser, comme l'implication inverse du Lemme ii.1 présentée dans la Section ii.3 (et aussi dans la Section 4.3). Nous pourrions également vérifier des propriétés fonctionnelles sur d'autres fonctions, telles que la fonction d'initialisation de la matrice de rotation ou la composition de deux quaternions.

Deuxièmement, un point central que nous n'avons pas abordé est la présence d'erreurs d'arrondi potentielles. En fait, après avoir vérifié que le code se comporte de manière similaire à la spécification mathématique en utilisant le modèle arithmétique `real`, nous pourrions utiliser le modèle arithmétique `float` qui est conforme aux spécifications IEEE. Nous aurions pu prendre en compte les erreurs d'arrondi potentielles générées lors du calcul et obtenir une estimation de ces erreurs. L'idée est d'offrir des garanties que les résultats obtenus sont suffisamment proches du calcul mathématique attendu avec une différence négligeable. Bien entendu, il faudrait définir formellement le terme "négligeable" dans le contexte de l'autopilote Paparazzi. Une telle vérification aurait été possible, mais aurait pris du temps et aurait nécessité l'utilisation de *pilotes* supplémentaires ou de plugins Frama-C, car les solveurs ne prennent pas en charge pleinement la vérification des propriétés en virgule flottante.

Le but de ce travail principalement expérimental était de déterminer les limites de l'approche choisie et son applicabilité par les ingénieurs. La vérification des erreurs d'exécution à l'aide de WP nécessite au moins une connaissance de base des méthodes déductives pour les annotations de boucles avec des variantes de boucle et des invariants. La vérification des propriétés fonctionnelles est plus complexe. Ce type de vérification nécessite la capacité de définir formellement les propriétés attendues, c'est-à-dire de définir formellement les spécifications et, plus important encore, de décider quels modèles arithmétiques et de mémoire utiliser. Le chapitre 3 présente un example avec la fonction C `sqrt` fonction qui est impossible de prouver qu'une fonction renvoie toujours le résultat mathématique correct en utilisant la sémantique de la computation en virgule flottante de C. Les utilisateurs doivent donc comprendre les avantages et les inconvénients des différents modèles pour faire le choix le plus judicieux en fonction de leurs besoins. Ils peuvent également avoir besoin de connaître différents plugins pour les combiner afin de tirer le meilleur parti de leurs forces combinées.

Frama-C est un outil puissant, mais présente des limitations. Premièrement, l'interface graphique nécessite un redémarrage à chaque modification du code, sans possibilité de redémarrer les solveurs directement depuis celle-ci. Deuxièmement, lorsqu'il y a des erreurs, trois cas se présentent sans qu'il soit possible de déterminer lequel facilement : des erreurs d'utilisation nécessitant des corrections, des bugs de Frama-C nécessitant une correction,

ou des fonctionnalités non implémentées. La mise à jour fréquente de Frama-C pose des défis pour la confiance et la maintenance des preuves. Enfin, la vérification des erreurs d'exécution nécessite une connaissance de base des méthodes déductives, tandis que la vérification des propriétés fonctionnelles est plus complexe, exigeant la compréhension des spécifications et des modèles arithmétiques et de mémoire.

Remarque

La Section 4.4 discute plus en détail les limitations de l'utilisation de l'outil Frama-C.

iii Verification de compilateur avec Coq

Paparazzi offre la possibilité de définir des missions spécifiques. Ces missions, également appelées «plans de vol» (*flight plan* en anglais), sont exprimées à l'aide d'un langage spécifique basé sur XML, noté `FPL` dans cette thèse.

Les plans de vol sont des missions complexes, et `FPL` propose des fonctionnalités impératives communes pour les définir, telles que des variables mutables, des exceptions ou des boucles. `FPL` offre également des primitives de navigation spécifiques, telles que «aller à une position» ou «faire un cercle autour d'un emplacement». Par exemple, la mission suivante peut être décrite en `FPL` : «lorsque le drone démarre, il doit d'abord initialiser ses capteurs et attendre que la connexion GPS soit établie. Ensuite, le drone doit décoller et faire un cercle autour d'une certaine position GPS pour collecter des données. Pendant le vol, si le niveau de batterie descend en dessous de 20%, le drone doit automatiquement retourner à la position de départ (`Home`) et atterrir».

Actuellement, Paparazzi fournit un générateur de code qui prend en entré un plan de vol `FPL` et génère un fichier C qui doit être compilé avec le pilote automatique pour être intégré dans le drone. Le plan de vol ne peut donc pas être modifié en cours de vol. Cependant, l'opérateur du drone peut interagir avec le plan de vol et en changer l'ordre d'exécution. Le fichier de code C généré est principalement composé d'une fonction d'étape (*step function*), nommée `auto_nav`, qui est appelée périodiquement par le pilote automatique pour calculer les prochaines étapes du plan de vol à exécuter.

Le générateur est écrit en OCaml, et les utilisateurs de Paparazzi peuvent ne pas avoir entièrement confiance en le code généré. En particulier, on peut se demander si a) la fonction qui calcule les prochaines étapes à exécuter dans le plan de vol se termine toujours, et b) le code C généré se comporte comme décrit dans le plan de vol. Ces questions sont cruciales, car le code C produit est destiné à être directement intégré dans le drone.

Vérifier que le code C intégré est correct, c'est-à-dire qu'il se comporte comme prescrit par le plan de vol, peut être considéré comme un problème de «vérification de compilateur» où nous devons prouver que le compilateur traduisant les plans de vol `FPL` en code C garantit la correction du code intégré. Même si la vérification des compilateurs est un problème connu [Dav03, MCP67], des avancées récentes ont été réalisées en utilisant des assistants de preuve, en particulier l'assistant de preuve Coq. Coq permet d'écrire des programmes dans le langage fonctionnel Gallina, de prouver formellement des propriétés sur de tels programmes

à l'aide de puissantes tactiques, et d'extraire des programmes Gallina en programmes OCaml équivalents du point de vue sémantique [Let04,SBF+20,AAM+16]. Les principales étapes de la vérification d'un compilateur avec Coq peuvent être résumées comme suit : tout d'abord, exprimer la sémantique des langages source et cible en Gallina, puis écrire le compilateur en Gallina, et enfin prouver un théorème de préservation de la sémantique établissant que le code produit a le même comportement que le code source selon leur sémantique respective. Les compilateurs CompCert [LBK+16] C et Velus [BBP20,Bru20] Lustre sont des exemples récents de la faisabilité de cette approche pour de vastes sous-ensembles de langages de programmation réels.

Cette section, qui résume la partie II de la thèse,présente la vérification du générateur de plans de vol [PGH+23] en utilisant Coq et est organisée comme suit : La Section iii.1 présente FPL avec ses nouvelles fonctionnalités, donne un aperçu de sa sémantique à travers l'exécution d'un exemple, puis définit sa sémantique formelle. La Section iii.2 présente l'architecture du nouveau compilateur à trois passes et décrit l'ébauche de la preuve de préservation du compilateur ainsi que les choix faits pour en simplifier l'écriture. Enfin, la Section iii.3 présente les enseignements tirés et les problèmes rencontrés lors du développement du générateur.

iii.1 Langage de description de plan de vol

Paparazzi utilise un langage spécifique au domaine (DSL: Domain Specific Language) pour décrire les plans de vol, avec une syntaxe concrète en XML. FPL est une extension conservatrice de ce langage qui offre de nouvelles fonctionnalités, telles qu'un mécanisme de protection contre les comportements indésirables, qu'ils soient spécifiés dans le plan de vol ou par l'opérateur du drone.

Un fichier FPL[2] est divisé en deux sections : l'*en-tête du plan de vol* et la partie *core*. L'*en-tête du plan de vol* contient des définitions et des métadonnées. Il est composé de plusieurs sections : l'en-tête (code C arbitraire ajouté en tête du fichier C généré), les points de passage (une liste de points constants définis à l'aide de positions GPS ou de coordonnées relatives), les secteurs définis à l'aide de listes de points de passage, et les variables locales pouvant être utilisées dans le plan de vol[3].

Le *core* est la partie principale d'un plan de vol et définit les différentes parties de la mission. Dans la suite, lorsque nous parlons d'un plan de vol, nous nous référons uniquement à sa partie *core*. La section iii.1.1 présente la syntaxe de FPL et une description informelle de sa sémantique. La section iii.1.2 fournit un exemple intuitif de l'exécution d'un plan de vol. La section iii.1.3 présente les nouvelles fonctionnalités que nous avons ajoutées dans le langage. La section iii.1.4 introduit la sémantique formelle de FPL. Enfin, la section iii.1.5 présente la structure du code C généré.

[2] Le projet contient plusieurs exemples de plans de vol, tels que `full_example.xml` par exemple.
[3] La valeur de ces variables peut être mise à jour par les opérateurs pendant un vol.

iii.1.1 Syntaxe et sémantique informelle de FPL

La figure 1 présente la structure FP Gallina qui correspond à la grammaire FPL[4]. Nous désignons par *c_code* tout code C valide, par *c_value* toute expression C, par *c_cond* tout code C qui peut être évalué comme une expression booléenne, et par *var_name* le nom d'une variable C (voir la section 7.1.1). L'analyse syntaxique du langage FPL en FP est effectuée par un préprocesseur, présenté dans la section iii.2.1.

flight_plan ::= {| `excpts` : list *fp_exception*, `blocks` : list *fp_block* |}

fp_exception ::= {|
  `cond` : *c_cond*,
  `id` : *block_id*,
  `exec` : option *c_code*
|}

fp_block ::= {|
  `id` : *block_id*,
  `stages` : list *fp_stage*,
  `excpts` : list *fp_exception*
|}

fp_stage ::=
  `WHILE` (`cond`: *c_cond*)
    (`body`: list *fp_stage*)
  | **`SET`** (`variable`: *var_name*)
    (`valeur`: *c_value*)
  | **`CALL`** (`fun`: *c_code*)
  | **`DEROUTE`** (`idb`: *block_id*)
  | **`RETURN`** (`reset`: *bool*)
  | **`NAV`** (`mode`: *fp_nav_mode*)
    (`init`: *bool*)

Figure 1: La structure Gallina `Flight Plan` (FP) représentant FPL.

Un plan de vol est composé d'au moins un bloc et éventuellement d'exceptions. Un bloc décrit une partie de la mission (par exemple, «décollage» ou «initialisation des capteurs») et est composé d'un identifiant unique, d'exceptions locales potentielles et d'une liste d'instructions atomiques appelées *étapes* ou (*stages* en anglais). L'étape **`WHILE`** est une boucle impérative classique. L'étape **`SET`** assigne une `valeur` à une `variable`. L'étape **`CALL`** exécute le code C `fun`. L'étape **`DEROUTE`** modifie le bloc en cours d'exécution pour aller au bloc `idb` (la position dans le bloc et l'identifiant du bloc avant la déviation sont mémorisés comme dernière position). L'étape **`RETURN`** retourne au bloc qui a été exécuté avant la dernière étape de déviation ou d'exception. Son paramètre `reset` permet à l'utilisateur de choisir si l'exécution doit commencer au début du dernier bloc ou à sa dernière étape atteinte lorsque la déviation ou l'exception s'est produite. Enfin, l'étape **`NAV`** exécute le code de navigation correspondant à la primitve (par exemple, `GO` vers une position ou faire un `CIRCLE`) en fonction de la valeur prise par le paramètre `mode`[5].

Les exceptions constituent un mécanisme de protection pour éviter les comportements indésirables. Une exception est déclenchée lorsque sa condition `cond` est évaluée à `true`. L'exécution du plan de vol passe ensuite au bloc `id` et, si spécifié, le code `exec` est appelé. Par exemple, si le drone a moins de 20% de batterie restante, le plan de vol doit exécuter le bloc qui fait atterrir le drone à la position *Home*. Une exception peut être globale et sera testée à chaque appel de la fonction `auto_nav`, ou locale à un bloc et ne sera testée que lorsque le bloc est exécuté.

[4]Disponible dans le fichier `FlightPlan.v`

[5]Ce paramètre est défini par la somme *fp_nav_mode* qui représente chaque constructeur correspondant à une primitive de navigation, voir le fichier `FPNavigationMode.v`.

Finalement, notez qu'un plan de vol peut contenir des appels de code C arbitraire, par exemple, dans les conditions ou dans les étapes **CALL**. Nous discuterons des implications de la présence de ce code dans la section iii.1.4.

Remarque

La syntaxe de FPL présentée ici est une version simplifiée. La version complète est présentée en Section 7.1.2.

iii.1.2 Un exemple d'exécution d'un plan de vol

L'exécution des plans de vol est similaire à l'exécution de programmes écrits dans des langages synchrones typiques tels que Lustre [HCRP91] ou SCADE [Ber07] qui sont couramment utilisés dans le code embarqué. L'exécution du code synchrone est composée d'une phase d'initialisation suivie d'appels périodiques à une fonction d'étape. La phase d'initialisation, pour l'exécution du plan de vol, définit l'environnement afin de démarrer l'exécution au bloc 0. L'exécution du plan de vol consiste alors à appeler régulièrement la fonction `auto_nav`. Le premier appel à `auto_nav` exécute le premier bloc du plan et les étapes à l'intérieur du bloc doivent être exécutées. Il existe 2 types d'étapes : les étapes *continue* et les étapes *break*. Les formes des étapes sont définies statiquement et implicitement par la sémantique. L'exécution d'un bloc consiste à exécuter ses étapes séquentiellement dans leur ordre de définition tant qu'elles sont des étapes *continue*. L'exécution de la fonction `auto_nav` se termine lorsqu'une étape *break* est exécutée. Dans ce cas, l'exécution du plan de vol est mise en pause et reprise lors du prochain appel à `auto_nav`.

Exemple de plan de vol :

```
{| excpts : [],
    blocks : [
      {| id: 0, excpts: [],
          stages: [
            CALL "InitSensors()";
            WHILE "!GPSFixValid()" {};
            SET "home" "GPSPosHere()"]
      |};
      {| id: 1, excpts: [],
          stages: [
            NAV (TakeOff params) true;
            DEROUTE 10]
      |};
      ... {| id: 10, ... |} ...
    ]
|}
```

Une exécution possible de la fonction `auto_nav` :

Appel	Bloc actuel	Code exécuté
1	0	`InitSensors()` `!GPSFixValid()` ⇑ `true`
2 → 8	0	`!GPSFixValid()` ⇑ `true`
9	0	`!GPSFixValid()` ⇑ `false` `home = GPSPosHere()`
10	1	`StartMotors()`
11 → 19	1	`TakeOffDone()` ⇑ `false`
20	1	`TakeOffDone()` ⇑ `true` `Deroute` → 10
21	10	...
⋮	⋮	⋮

Figure 2: Exemple de l'exécution d'un plan de vol.

La figure 2 présente un exemple de plan de vol et l'une de ses exécutions possibles. Ce plan de vol ne contient aucune exception et plusieurs blocs dont seuls 3 sont représentés. Présentons brièvement l'exécution de la fonction `auto_nav` sur cet exemple. Le premier

bloc (bloc 0) est entré et sa première étape est exécutée. L'étape **CALL** est une étape *continue* : son exécution appelle simplement le code C qu'elle contient et l'étape suivante est exécutée. L'étape suivante est une étape **WHILE**, donc la condition de la boucle, c'est-à-dire "!GPSFixValid()", doit être évaluée. Supposons que cette évaluation renvoie true. Dans ce cas, l'étape **WHILE** est une étape *break* : elle met fin à l'exécution de la fonction auto_nav. Le plan de vol continuera son exécution lors de l'appel suivant à auto_nav, au cours duquel l'exécution du plan de vol reprend là où elle s'était arrêtée, c'est-à-dire à l'intérieur de la boucle **WHILE**. Comme le corps de la boucle est vide dans cet exemple, il n'y a aucune étape à exécuter et la condition de la boucle est réévaluée.

L'exécution de auto_nav continue à évaluer la condition de boucle jusqu'à ce qu'elle soit évaluée à false. Supposons que la condition de boucle soit évaluée à false lors de l'appel 9. Dans ce cas, l'étape **WHILE** est considérée comme *continue*. La dernière étape **SET** est donc exécutée et, étant donné qu'il s'agit d'une étape de continuation, son exécution met fin à l'exécution du bloc, arrête l'exécution de auto_nav et le bloc suivant (numéroté 1 ici) sera exécuté lors de la prochaine itération.

L'exécution de l'étape **NAV** est traduite en un appel à une fonction C qui définit les paramètres de navigation en fonction de la primitive de navigation utilisée. Ces paramètres seront utilisés par l'autopilote et traduits en ordres pour les moteurs. Dans cet exemple, l'étape **NAV** exécute la primitive de navigation TakeOff. Cette étape nécessite une étape d'initialisation car le paramètre init est défini sur true. Il y a donc un code spécifique qui sera appelé lors de la première exécution de l'étape (ici StartMotors qui n'apparaît pas dans le plan de vol mais qui est spécifié dans l'autopilote Paparazzi et ajouté par le générateur). Ensuite, le code C correspondant à la deuxième partie de la primitive TakeOff est une fonction TakeOffDone qui renvoie une valeur booléenne indiquant si le drone a décollé ou non. Si la fonction TakeOffDone renvoie false, alors l'étape est une étape de *break*, sinon **NAV** est considéré comme une étape de *continue*. Remarquez que la primitive de navigation TakeOff et les fonctions C appelées ont été créées pour cet exemple. Le vrai code C généré peut correspondre à plusieurs appels de fonctions qui peuvent avoir des noms non explicites. Ce code est généré à partir de fonctions présentes dans le fichier FPNavigationModeGen.v.

Finalement, lorsque le drone a décollé (appel 20), l'exécution de auto_nav se poursuit et l'étape **DEROUTE** est exécutée. Cette étape est une étape de *break* et le prochain appel à auto_nav exécutera le bloc correspondant dérouté (bloc 10 ici).

La grammaire présentée dans la figure 1 et utilisée dans l'exemple précédent définit une syntaxe simplifiée de FPL. Par exemple, l'étape *call* a plusieurs paramètres supplémentaires : il est possible de spécifier que le code doit être exécuté jusqu'à ce qu'il retourne true ou que l'étape doit s'arrêter après l'exécution. Nous omettons ces détails pour conserver les éléments clés de FPL et donner une présentation intuitive de sa syntaxe et de sa sémantique. La définition complète de FPL peut être trouvée en Section 7.1.2.

Les blocs et les étapes sont normalement exécutés les uns après les autres dans leur ordre de définition, mais les instructions **DEROUTE** ou les exceptions peuvent modifier le bloc actuellement en cours d'exécution. Notez qu'un opérateur humain peut également changer manuellement le bloc actuel pendant l'exécution du plan de vol sur la console de contrôle de Paparazzi.

Remarque

La Section 7.2 présente un exemple plus complet d'exécution avec, par exemple, des exceptions en plus.

iii.1.3 Nouvelles fonctionnalités

FPL apporte deux nouvelles fonctionnalités au langage actuellement utilisé dans Paparazzi. Tout d'abord, nous ajoutons un autre mécanisme de protection appelé *forbidden deroute*. Les détours interdits doivent être spécifiés par l'utilisateur pour empêcher l'exécution de changements de blocs «dangereux», tels que le passage direct à un bloc où les moteurs du drone sont coupés depuis un bloc où le drone est en vol. Un plan de vol peut contenir un nombre quelconque de détours interdits. La syntaxe des plans de vol est étendue comme suit :

fp_fb_deroute ::= {|
  `from` : *block_id*
  `to` : *block_id*
  `only_when` : option *c_cond*
|}

flight_plan ::= {|
  `fb_drtes` : list *fp_fb_deroute*
  `excpts` : list *fp_exception*
  `blocks` : list *fp_block*
|}

Un *forbidden deroute* décrit un changement d'un bloc `from` à un bloc `to` qui doit être soit interdit de manière inconditionnelle, soit surveillé par une condition `only_when`[6].

La deuxième fonctionnalité que nous avons ajoutée est une demande de l'utilisateur Paparazzi concernant l'exécution de code C lors de l'entrée ou de la sortie d'un bloc. Nous avons ajouté la possibilité pour les utilisateurs de spécifier du code `on_enter` et `on_exit` pour chaque bloc. Ce code est exécuté lors de l'entrée (respectivement de la sortie) du bloc. Comme pour d'autres détails du plan de vol, `on_enter` et `on_exit` ne sont pas présentés dans la syntaxe formelle du FPL afin de se concentrer sur les éléments les plus intéressants du FPL. Notez cependant que `on_enter` et `on_exit` sont effectivement gérés dans notre implémentation vérifiée.

Remarque

Le Chapitre 8 présente en détail les nouvelles fonctionnalités ajoutées au générateur.

iii.1.4 Sémantique

Les plans de vol décrivent le comportement souhaité du drone lorsqu'il vole de manière autonome. Dans cette section, nous définissons une sémantique pour FP décrivant l'évolution du système et les événements observables qui se produisent pendant l'exécution. Le système, modélisé comme un état d'exécution et des événements observables, est introduit dans une première section. Ensuite, nous présenterons quelques règles d'inférence pertinentes de la sémantique. Ces définitions seront ensuite utilisées dans la Section iii.2 pour vérifier le générateur. La définition de la sémantique est présentée en Section 7.3.

[6] Étant donné que les conditions sont des morceaux de code C arbitraire vus comme un type abstrait *c_cond*, nous utilisons un type d'option pour représenter un détour inconditionnel avec `None`, afin d'optimiser la génération de code.

États et traces

Une sémantique définit généralement comment un système évolue lors de son exécution, d'un état initial s à un état final s', mais décrit également les interactions avec le monde extérieur. Ces opérations externes sont modélisées par des *traces* ou des *outputs*. Par exemple, un langage de programmation impératif utilise souvent, en tant qu'état, une abstraction de la mémoire de l'ordinateur. La sémantique correspondante décrit comment la mémoire change lors de l'exécution du programme. Les traces peuvent donc être des séquences d'accès à une mémoire externe ou des messages reçus et envoyés via un réseau.

Définition iii.1 - (fp_state, src/semantics/FPEnvironment.v:41).

fp_state ::= {|
`idb`: *block_id*, `stages`: list *fp_state*,
`lidb`: *block_id*, `lstages`: list *fp_stage*
|}

Nous abstrayons l'état réel de la mémoire du plan de vol en nous concentrant sur les éléments clés tels qu'indiqués dans la Définition iii.1 : un environnement contient la position actuelle (resp. précédente) dans le plan de vol, représentée par le bloc actuel `idb` et les étapes restantes `stages` à exécuter dans ce bloc (resp. `lidb` et `lstages`). La position précédente est sauvegardée lorsqu'un déroutement ou une exception se produit.

Lors de l'exécution d'un plan de vol, plusieurs fonctions C spécifiques à chaque type de drone (aile fixe, rover, rotorcraft, etc.) sont appelées. Par exemple, la fonction `NavCircleWaypoint` qui définit les paramètres de navigation pour effectuer un cercle a des implémentations différentes pour les drones à aile fixe ou à rotor. L'exécution du plan de vol peut également exécuter du code C arbitraire défini par l'utilisateur. Ainsi, la sémantique produira des sorties qui représentent les appels à ce code C considérés comme des *appels externes*.

Définition iii.2 - (fp_trace, src/syntax/BasicTypes.v:92).

fp_event ::= **COND** (*c_cond* × *bool*)
| **C_CODE** *c_code*
| **SKIP**
fp_trace ::= list *fp_event*

Une seule trace est une valeur de *fp_event*, c'est-à-dire l'évaluation d'une condition ou d'un code C arbitraire. Le champ *bool* enregistre la valeur de la condition *c_cond*.

L'exécution du code C représenté dans la trace peut modifier l'environnement mémoire du drone. Des fonctions C spécifiques, comme les fonctions de navigation, sont définies en dehors du code du plan de vol, donc leur exécution ne modifie pas l'état du plan de vol. Cependant, du code C arbitraire peut être introduit par les utilisateurs et nous devons supposer que ce code ne modifie pas l'état du plan de vol, comme discuté dans la section iii.2.3. Ainsi, l'environnement mémoire du drone peut être décomposé en deux parties disjointes : a) l'espace mémoire abstrait par *fp_state* b) l'espace mémoire qui peut être modifié lors de l'exécution des appels de code C représentés par la trace *fp_trace*.

Définition iii.3 - (fp_env, src/semantics/FPEnvironment.v:50).

$fp\_env ::= (fp\_state \times fp\_trace)$

Un environnement contient l'état actuel du plan de vol ainsi qu'un historique des opérations externes qui ont eu lieu.

Le résultat de l'évaluation d'une condition est requis pour définir la sémantique, mais étant donné que ces conditions sont du code C arbitraire qui ne peut pas être interprété, nous supposons l'existence d'une fonction *evalc* spécifiée comme suit.

Paramètre iii.1 - (evalc, src/semantics/FPEnvironment.v:79).

$evalc\colon\ fp\_env\ \rightarrow c\_cond \rightarrow (bool\times\ fp\_env\ )$

evalc e c renvoie un couple $(b,\ e')$ tel que l'évaluation du code C c renvoie le booléen b. La trace de l'environnement e est mise à jour en ajoutant **COND**$(c,\ b)$ à celle-ci, donnant e'.

L'historique des traces contenu dans *fp_env* est essentiel pour l'évaluation des conditions. Prenons un exemple : évaluer la condition `"Battery() < 80"` deux fois peut donner des résultats différents car la batterie du drone se vide pendant le vol. Mais comme nous avons choisi de ne pas spécifier son environnement extérieur réel, l'utilisation d'un historique des traces nous permet de prendre en compte de tels cas et de représenter les appels de fonctions C avec des effets secondaires.

Règles d'inférence décrivant la sémantique des plans de vol

Nous avons défini un *fp_env* initial noté e_0 et une fonction *step* qui représente une évaluation sémantique en grands pas habituelle [NN07, Win93]. L'état e_0 contient une trace vide et fait référence au premier bloc du plan de vol. L'exécution du plan de vol consiste ensuite à appeler régulièrement la fonction `auto_nav`. La fonction *step* représente l'exécution d'un appel à `auto_nav`. Nous soulignons le fait que pendant cette exécution, le plan de vol n'est pas exécuté jusqu'à la fin, mais l'étape et/ou le bloc courant sont modifiés et du code C peut être appelé. L'environnement résultant sera prêt pour une autre étape d'exécution. Dans ce qui suit, en plus de chaque définition de notation, nous fournissons son nom Gallina et un lien cliquable vers sa définition dans le code source.

Notation iii.1 - (step, src/semantics/FPBigStep.v:263).
La fonction *step* décrit l'exécution d'un appel à la fonction de navigation :

$$step : flight\_plan \rightarrow fp\_env \rightarrow fp\_env$$

step $fp\ e = e'$ noté $e \xhookrightarrow[fp]{\mathsf{FP}} e'$ indique que e' est l'environnement résultant après l'exécution du plan de vol fp à partir de l'environnement e. La définition de la fonction utilisant cette notation est présentée dans la Figure 3[7]. Par la suite, nous manipulerons de manière interchangeable *fp_env* en tant que variable unique ou un couple de *fp_state* et *fp_trace*: (voir Définition iii.3).

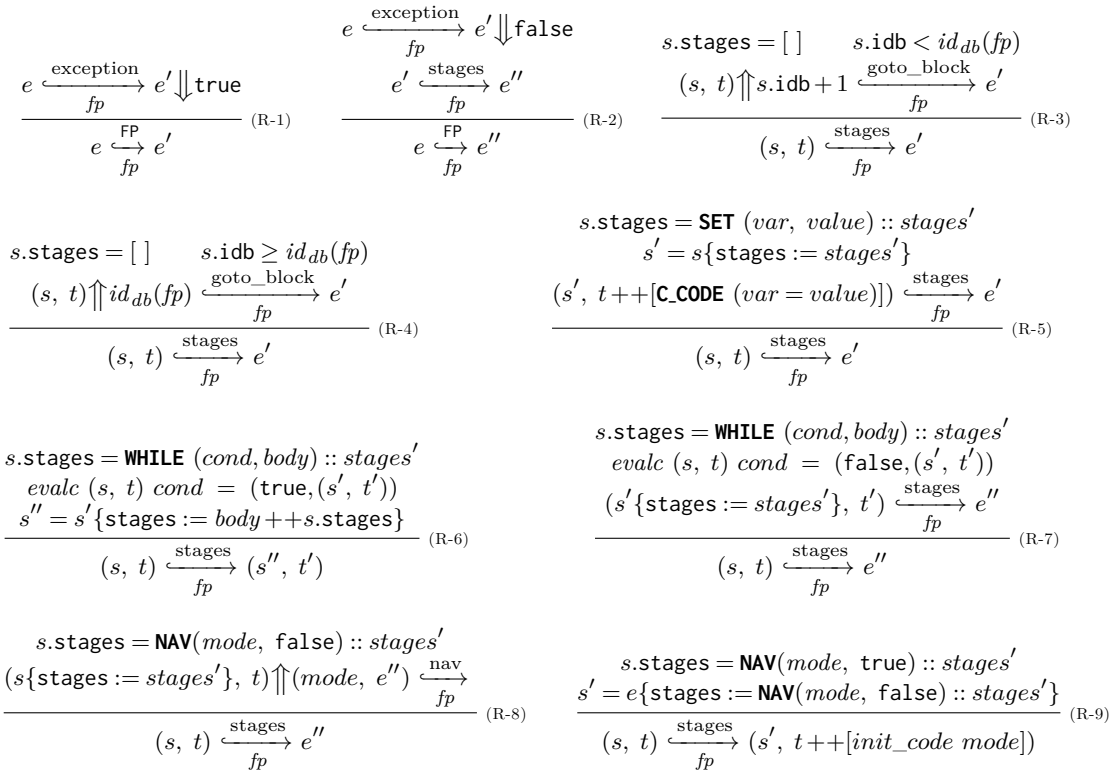

Figure 3: Règles d'inférence pour FP.

L'exécution d'un plan de vol peut être décomposée en deux phases. Tout d'abord, toutes les exceptions sont vérifiées et si l'une d'entre elles est soulevée, alors l'environnement est redirigé vers un bloc sûr. Sinon, les étapes restantes sont exécutées. Nous présentons donc d'abord la sémantique pour gérer les exceptions, pour rediriger vers un bloc et ensuite exécuter les étapes.

Notation iii.2 - (exception, src/semantics/FPBigStepGeneric.v:250).
$e \xhookrightarrow[fp]{\text{exception}} e' \Downarrow res$ est vrai si le test des exceptions fp démarrant de e se termine dans l'environnement e' et res est `true` si une exception globale ou locale a été déclenchée.

Notation iii.3 - (goto_block, src/semantics/FPBigStepGeneric.v:199).
$e \Uparrow id \xhookrightarrow[fp]{\text{goto\_block}} e'$ est vrai si le changement de l'environnement e vers le bloc id génère un nouvel environnement e'. Le nouvel environnement e' mémorise, en tant que position précédente, la position actuelle de e, et sa position actuelle pointe vers le bloc id si le changement n'est pas interdit. Si le changement est interdit, le nouvel environnement pointe vers la même position que l'environnement e, avec une *fp_trace* mise à jour si certaines conditions des changements interdites ont été évaluées.

[7] Il convient de noter que dans le développement Coq, la sémantique est implémentée en tant que fonction calculable (voir FPBigStep.v), mais nous la présentons sous la forme de règles d'inférence pour plus de lisibilité.

Notation iii.4 - (`run_step`, `src/semantics/FPBigStep.v:254`).
$(s,\ t) \xrightarrow[fp]{\text{stages}} e'$ est vrai si l'exécution de *fp* démarrant depuis l'environnement $(s,\ t)$ avec la liste $s.\texttt{stages}$ restant à exécuter se termine dans l'environnement e'. Si aucun stage ne reste à exécuter, l'exécution se poursuit avec le prochain bloc pour l'appel suivant de la fonction *step*.

Nous expliquons ci-dessous les règles d'inférence présentées à la figure 3. Nous désignerons par $var\{f := v\}$ un enregistrement similaire à var, mais avec la valeur v pour le champ f, et $id_{db}(fp)$ l'indice du bloc par défaut du plan de vol *fp* qui est le dernier bloc du plan de vol.

La règle d'inférence R-1 décrit ce qui se passe lorsqu'une exception est levée, et la règle R-2 décrit ce qui se passe lorsqu'aucune exception n'est levée.

La règle d'inférence R-3 décrit ce qui se passe lorsqu'il n'y a pas d'étape à exécuter mais que le bloc actuel n'est pas le dernier bloc du plan de vol. Dans ce cas, le bloc suivant à être exécuté est le bloc suivant dans le plan de vol. Sinon, le bloc actuel devient le bloc par défaut et nous restons là (cf. règle R-4).

La partie la plus intéressante de la sémantique concerne l'exécution des étapes. Lorsqu'il reste des étapes à exécuter dans l'environnement actuel, elles sont exécutées dans leur ordre de définition jusqu'à ce qu'une étape de *break* soit exécutée. Nous nous concentrons sur la présentation des différences pour les étapes **`SET`**, **`WHILE`** et **`NAV`** qui sont représentatives de toutes les étapes, donc sans perte de généralité.

L'étape **`SET`** (règle R-5) est une phase *continue* qui attribue simplement une valeur à une variable. La valeur attribuée est du code C arbitraire et ne peut pas être analysée. L'assignation est donc ajoutée dans la trace.

L'étape **`WHILE`** exécute une liste d'étapes tant qu'une condition est vraie. La condition est évaluée, et si elle est vraie, le corps de la boucle est ajouté au début de la liste des étapes à exécuter. L'exécution est alors interrompue, et l'environnement final est mis à jour (règle R-6). L'étape **`WHILE`** est conservé dans la liste des étapes afin d'évaluer la condition et éventuellement itérer la boucle après avoir exécuté son corps. Lorsque la condition de la boucle est `false`, l'étape **`WHILE`** est consommé, et l'exécution se poursuit (règle R-7).

L'étape de navigation est conçu pour encapsuler le comportement de toutes les primitives de navigation (`CIRCLE`, `GO`, etc.). Certaines d'entre elles nécessitent une étape d'initialisation, décrite par le paramètre *init* dans `FP`. Nous notons $e{\Uparrow}(mode,\ e') \xrightarrow[fp]{\text{nav}}$ la sémantique représentant l'exécution de l'étape **`NAV`** avec le paramètre *mode* de l'état e à e'. Nous notons *init_code mode* le code initial à exécuter pour le mode *mode*.

Lorsque *init* est défini sur `false`, le code de navigation est directement exécuté comme présenté dans la règle R-8. Dans l'autre cas, une étape d'initialisation est nécessaire avant d'exécuter le code de navigation. La règle R-9 décrit ce comportement : *init_code* est exécuté, mais l'étape de navigation n'est pas consommé et est modifié en définissant son paramètre *init* sur `false`.

Les autres règles d'inférence de la sémantique sont similaires et peuvent être trouvées à l'intérieur du fichier `FPBigStep.v` qui contient les définitions Gallina ou en Section 7.3.3.

iii.1.5 Code C généré

La Figure 4 présente le code C généré à partir d'un plan de vol FP simple avec un bloc et deux étapes **CALL**. Nous utilisons GEN_DEFAULT_C_CODE pour désigner le code généré pour le bloc par défaut ajouté par le prétraitement[8] à la fin de la liste des blocs. Les fonctions get_nav_block()/get_nav_stage() renvoient la valeur des variables nav_block/nav_stage, c'est-à-dire les identifiants du bloc/étape actuellement en cours d'exécution.

```
{|  excpts: [],
    fb_drtes: [],
    blocks: [
      {|
        id: 0,
        excpts: [],
        stages: [
          CALL "func1()";
          CALL "func2()"
        ]
      |}
    ]
|}
```

```
static inline void auto_nav(void) {
    switch (get_nav_block()) {
        case 0: // Block 0
            switch (get_nav_stage()) {
                case 0: // Stage 0
                    func1();
                case 1: // Stage 1
                    func2();
                default:
                case 3: // Default Stage
                    NextBlock();
                    break;
            }
            break;
        case 1: // Default Block
            GEN_DEFAULT_C_CODE()
    }
}
```

Figure 4: La fonction C auto_nav générée à partir d'un fichier FPL simple.

La fonction auto_nav est principalement composée d'une instruction switch dans laquelle chaque instruction case correspond au traitement d'un bloc. Chaque traitement de bloc consiste également en une instruction switch appelée *stage switch*. Chaque cas d'un stage switch correspond à l'exécution d'une étape. L'exemple précédent montre deux étapes **CALL** qui sont des étapes *continue*, donc les instructions case générées ne contiennent pas d'instruction break : si l'étape 0 est exécutée, lorsque la fonction func1 a été exécutée, la fonction func2 sera appelée. Sinon, les étapes *break* contiennent une instruction break, et le stage switch sera quitté lorsqu'une telle étape est rencontrée, mettant fin à l'exécution de auto_nav. Il est à noter que la structure du code C généré pour les étapes d'un bloc peut être différente de la structure FP. Dans l'exemple présenté sur la Figure 4, chaque instruction case C correspond à une étape FP, mais il peut y avoir des étapes qui peuvent produire plusieurs instructions case. Par exemple, les étapes de navigation peuvent nécessiter une étape d'initialisation, en fonction du paramètre init, qui est ajoutée en tant qu'instruction case supplémentaire dans le code généré.

[8] Le prétraitement est l'une des étapes de la génération de code qui sera détaillée dans la Section iii.2.1.

iii.2 Générateur

Cette section présente notre nouveau générateur et la manière dont nous l'avons vérifié. La Section iii.2.1 présente la nouvelle architecture du générateur vérifié. La Section iii.2.2 propose une esquisse du théorème de préservation de la sémantique qui a été prouvé. Enfin, nous discutons dans la Section iii.2.3 des hypothèses formulées et des axiomes utilisés pour vérifier le générateur.

iii.2.1 Architecture du générateur de code

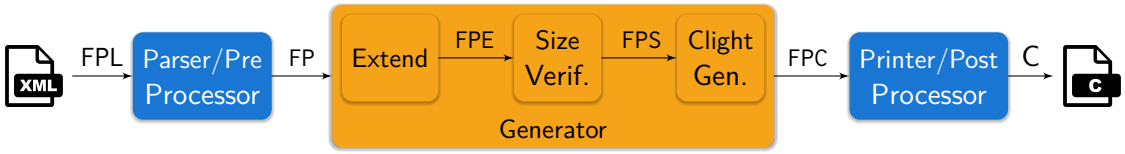

Figure 5: Architecture du générateur.

La Figure 5 présente l'architecture du générateur de code. Tout d'abord, le fichier d'entrée FPL est analysé par un préprocesseur écrit en OCaml, et certaines transformations sont effectuées pour générer la structure FP en Gallina représentant le plan de vol. Le cœur du générateur, écrit en Gallina, traduit ensuite FP en code Clight, qui est ensuite imprimé à l'aide d'un post-processeur OCaml pour obtenir le fichier de code C final à intégrer dans le pilote automatique. Le générateur FP vers Clight est divisé en trois passes : une passe d'*extension*, une passe de *size verification* et une passe de *generation*. La passe d'extension transforme la structure FP en une version *extended* FPE plus proche du code C à générer. La passe de *size verification* effectue une vérification de taille sur la structure FPE et peut renvoyer des erreurs. Si FPE est de taille correcte, cette passe renvoie une structure appelée FPS. Enfin, la passe de génération convertit FPS en code Clight.

Il serait idéal de vérifier que l'ensemble du générateur est correct. Malheureusement, certaines opérations telles que la lecture et l'écriture de fichiers sont difficiles à implémenter en Gallina. Par conséquent, la partie frontend et backend du générateur est actuellement écrite en OCaml. Cependant, nous sommes confiants que ces phases de prétraitement et de post-traitement sont correctes car elles ne réalisent que des transformations mineures. Le code de ces deux phases se trouve dans les fichiers OCaml `preproc.ml` et `postproc.ml` et ces deux passes sont détaillés dans les Sections 6.2.1 et 6.2.2.

Préprocesseur

Le premier rôle du préprocesseur est de convertir le fichier d'entrée FPL en une structure FP Gallina.

L'en-tête du plan de vol est prétraité avec des transformations simples, par exemple les coordonnées sont converties en différents formats (UTM, LLA, etc.). Les principales transformations se font sur la partie centrale du plan de vol : un bloc de sécurité est ajouté

et les étapes macros sont développées. Le bloc de sécurité est un bloc qui envoie le drone se poser à sa position de décollage. Il est ajouté à la fin du plan de vol afin d'être appelé uniquement lorsque le plan de vol original a été terminé. Les étapes macros sont du «sucre syntaxique» traduit en étapes atomiques : les boucles **FOR** sont converties en boucles **WHILE** et les étapes **PATH** qui prennent une liste de points de passage à suivre par le drone sont remplacées par une liste de primitives **NAV** GO.

La dernière transformation indexe les blocs initialement référencés dans FPL en utilisant des noms. Le préprocesseur vérifie que chaque référence de bloc correspond à une position d'un bloc défini dans le fichier FPL (en commençant à l'index 0) et remplace chaque nom de bloc par son index. Le préprocesseur génère ensuite un en-tête de fichier C composé de définitions (constantes pour les coordonnées des waypoints, les noms des blocs, la hauteur de sécurité, etc.) collectées pendant la phase de prétraitement.

Passe d'extension

Cette passe transforme les structures FP en FPE, un plan de vol étendu qui est une représentation intermédiaire plus proche de la structure finale du code C[9]. Comme présenté dans la Section iii.1.5, la structure FP contient des étapes qui correspondent à plusieurs instructions case dans le code C généré, comme l'étape **NAV** qui nécessite une étape d'initialisation. Certains constructs comme **WHILE** utilisent des listes d'étapes imbriquées pour représenter leurs corps, cependant le *stage switch* n'a lieu qu'une seule fois. De plus, toutes les étapes sont référencées par des indices dans le code C généré, mais FP ne contient pas de tels indices et sa sémantique procède en maintenant une séquence des étapes restantes à exécuter. FPE a été défini pour combler cet écart entre FP et le code C, et les différences entre FP et FPE concernent uniquement les étapes.

La passe d'extension est réalisée par la fonction extend_stages_default, qui numérote les étapes et linéarise la liste des étapes. Par exemple, l'étape **WHILE** sera dépliée en ajoutant le corps de la boucle à la liste principale des étapes. De nouvelles étapes seront également ajoutées, comme **NAV_INIT**$_e$ qui correspond au cas où un **NAV** nécessite une étape d'initialisation, ou **END_WHILE**$_e$ ajouté après le corps de la boucle pour générer un *case switch* dans le code C généré qui redémarrera la boucle.

Remarque

La Section 9.1 présente en détail cette phase d'extension et le langage FPE.

Passe de vérification de taille

Pendant la phase de prétraitement, les plans de vol qui sont syntaxiquement incorrects sont rejetés, mais aucune garantie n'est obtenue sur le processus de numérotation des blocs au niveau Coq. De plus, le code C généré vise à être compilé et exécuté sur des drones aux ressources limitées. Afin d'optimiser l'utilisation de la mémoire, les variables entières telles que nav_block, qui contient l'identifiant du bloc actuel, sont codées sur 8 bits, limitant ainsi

[9] Les définitions Coq sont disponibles dans le fichier FlightPlanExtended.v.

le nombre de blocs et d'étapes à 256. Nous avons donc défini une propriété `verified_fp_e`, notée H_{ws}, qui est satisfaite si un `FPE` est correct : le plan de vol ne contient pas plus de 256 blocs, chaque bloc est correctement numéroté et ne contient pas plus de 256 étapes, ainsi que d'autres propriétés non détaillées ici.

La passe de vérification de taille, réalisée avec la fonction `size_verification`, renvoie soit des messages d'erreur si le plan de vol est détecté comme incorrect, soit un `FPS`, un plan de vol de taille correcte. `FPS` est le sous-ensemble des plans de vol `FPE` qui respectent la propriété H_{ws}. La structure `FPS` transporte également une preuve de H_{ws} qui sera utilisée pour prouver le théorème de préservation de la sémantique présenté dans la Section iii.2.2.

La restriction de taille a été ajoutée après avoir identifié un problème dans l'ancien générateur, car il ne réalisait pas de vérification de taille et pouvait donc produire un plan de vol avec plus de 256 blocs tout en utilisant des variables de 8 bits pour stocker l'identifiant du bloc actuel. Certains blocs ne peuvent donc pas être accessibles, et le calcul de l'identifiant du bloc suivant peut provoquer un dépassement. Ce problème n'avait pas été soulevé auparavant car la limite de taille n'était pas atteinte dans des cas d'utilisation réels. Les plans de vol sont généralement courts car ils décrivent des missions réalisables par de petits drones avec des batteries limitées.

Remarque

La Section 9.2 détaile cette passe de *size verfication* et défini la propriété H_{ws}.

Fonction de génération Clight

La fonction finale de génération en Gallina est une fonction `global_definitions` prenant un plan de vol `FPS` en paramètre et renvoyant un programme Clight, noté `FPC`, contenant une variable globale `nb_block` représentant le nombre de blocs du plan de vol, ainsi que certaines fonctions : `forbidden_deroute` testant si une déviation entre deux blocs est interdite ou non, `auto_nav` et certaines fonctions auxiliaires qui ne seront pas détaillées dans cet article.

Les fonctions `extend_stages_default`, `size_verification` et `global_definitions` sont regroupées à l'intérieur de la fonction `generate_flight_plan`. Cette fonction prend un plan de vol `FP` et produit le code Clight final ou des messages d'avertissement et d'erreur, principalement générés lors de la passe de vérification de taille.

Remarque

La passe de génération qui convertit `FPS` en code Clight est détaillée dans la Section 9.3. La Section 9.4 présente comment les différentes passes de transformation sont liées ainsi que de nouvelles passes d'analyse qui fournissent des messages d'avertissement pour les utilisateurs.

Post-processeur

Le post-processeur produit un fichier de code C compilable à partir de la structure Clight générée. La première partie du fichier est l'en-tête générée par le préprocesseur. La deuxième

partie contient les fonctions produites par le générateur. Remarquez que nous voulons maintenir une compatibilité totale avec Paparazzi et donc être aussi proches que possible du code C généré par le générateur de code Paparazzi précédent et non vérifié. En effet, les utilisateurs de Paparazzi peuvent se fier à la structure du code généré, incidemment ou non, pour développer leurs propres composants. Le plan de vol n'est pas non plus exécuté de manière isolée et accède à un certain nombre de variables globales dans Paparazzi.

Le post-processeur utilise le module `PrintClight.ml` de CompCert qui traduit la structure Gallina Clight en pseudo code C. Malheureusement, ce module a été principalement développé à des fins de débogage et le code produit ne peut pas être compilé. Par exemple, lorsqu'un identifiant de variable Clight est imprimé, le symbole $ est ajouté devant l'identifiant. Le post-processeur effectue une dernière passe pour convertir le code C imprimé en un code compilable.

Le code C généré est très similaire au code généré par l'ancien générateur OCaml. Les seules différences remarquables sont dues aux limitations de Clight. Par exemple, l'opérateur booléen (`&&`) n'existe pas en Clight et doit être remplacé par des instructions conditionnelles. De même, les expressions comme `fun1(fun2())` ne peuvent pas être définies en Clight et doivent être scindées en deux instructions, en utilisant une variable temporaire pour stocker le résultat de l'appel à `fun2()`.

iii.2.2 Vérification du générateur

La vérification du générateur est divisée en deux parties : nous prouvons d'abord que le générateur se comporte correctement et préserve la sémantique, puis nous prouvons que le code généré pour la fonction `auto_nav` se termine toujours.

Préservation de la sémantique

Permettez-nous tout d'abord de présenter la sémantique abstraite d'un plan de vol.

Définition iii.4 - (`fp_semantics`, `src/semantics/FPBigStepGeneric.v:32`).

fp_semantics ::= {|
`env:` $Type$,
`init_env:` `env` $\rightarrow Prop$,
`step:` `env` $\rightarrow$ `env` $\rightarrow Prop$
|}

fp_semantics décrit la sémantique d'un plan de vol : l'exécution commence avec un environnement initial e_0 (c'est-à-dire que `init_env` e_0 est satisfait) et chaque appel à la fonction `auto_nav` est représenté par le prédicat `step`. Une structure *fp_semantics* a été définie pour `FP`, `FPE`, `FPS` et `FPC`. La fonction `step` pour `FPC` correspond à l'exécution d'un appel à `auto_nav` en utilisant la sémantique Clight [BL09]. L'environnement pour la sémantique `FPC` est composé d'une liste de traces Clight et d'un environnement mémoire qui contient au moins les mêmes informations que *fp_state*. Par la suite, nous noterons par "$semantics\ fp$" la *fp_semantics* pour le plan de vol fp et "$semantics_c\ cfp$" la *fp_semantics* pour le programme Clight cfp.

Remarque

La Section 10.1 définit formellement les sémantiques pour les différentes structures de plan de vol.

L'idée de préservation sémantique consiste à «exécuter» à travers la sémantique formelle correspondante à la fois le programme source et le programme généré en partant de deux environnements équivalents et de vérifier que les deux exécutions se terminent dans des états équivalents. Pour cela, nous définissons une relation d'appariement $\overset{\text{env}}{\sim}$ entre les environnements source et cible. Deux environnements sont appariés par la relation $\overset{\text{env}}{\sim}$ s'ils représentent le même environnement de plan de vol.

Nous formalisons notre problème de vérification comme un problème de *bisimulation* standard, c'est-à-dire que nous exhibons une simulation directe et une simulation inverse entre les deux sémantiques.

Définition iii.5 - (fp_fsim_properties, src/semantics/FPBigStepGeneric.v:51).

$$
\begin{aligned}
&\textit{simulation}\ FP_1\ FP_2\ (\backsim) ::= \{|\\
&\quad \texttt{match\_initial\_env}: \forall\ e_1,\ FP_1.\texttt{initial\_env}\ e_1\\
&\qquad\qquad\qquad \rightarrow \exists\ e_2,\ FP_2.\texttt{initial\_env}\ e_2 \wedge e_1 \backsim e_2,\\
&\quad \texttt{match\_step}: \qquad \forall\ e_1\ e_1',\ FP_1.\texttt{step}\ e_1\ e_1'\\
&\qquad\qquad\qquad \rightarrow \forall\ e_2,\ e_1 \backsim e_2\\
&\qquad\qquad\qquad \rightarrow \exists\ e_2',\ FP_2.\texttt{step}\ e_2\ e_2' \wedge e_1' \backsim e_2'\\
&|\}
\end{aligned}
$$

Définition iii.6 - (bisimulation, src/semantics/FPBigStepGeneric.v:92).

$$
\begin{aligned}
&\textit{bisimulation}\ FP1\ FP2\ (\sim_e) ::= \{|\\
&\quad \texttt{forward\_sim}: \textit{simulation}\ FP1\ FP2\ (\sim_e),\\
&\quad \texttt{backward\_sim}: \textit{simulation}\ FP2\ FP1\ (\sim_e)\\
&|\}
\end{aligned}
$$

Les définitions de *fp_semantics* et *bisimulation* sont présentées dans la Section 10.2 et se trouvent dans le fichier FPBigStepGeneric.v.

Enfin, nous définissons le théorème de préservation de la sémantique pour le générateur de plan de vol, c'est-à-dire une bisimulation entre la sémantique de FP et FPC. Nous notons $\sim$ la relation de correspondance entre *fp_env* et l'environnement de la sémantique de FPC.

Théorème iii.1 - (semantic_preservation, src/verification/VerifFPToFPC.v:118).

$$
\begin{aligned}
&\forall \textit{fp}\ \textit{cfp},\ \textit{cfp} = \textit{generate\_flight\_plan}\ \textit{fp}\\
&\quad \rightarrow \textit{bisimulation}\ (\textit{semantics}\ \textit{fp})\ (\textit{semantics}_c\ \textit{cfp})\ (\sim)
\end{aligned}
$$

Ce théorème énonce que si le code C *cfp* peut être généré à partir du plan de vol *fp*, alors il existe une bisimulation entre *fp* et *cfp*. La propriété de simulation directe stipule que toute exécution de *fp* peut être simulée par le code C généré : aucun comportement décrit par la sémantique n'est perdu pendant la génération. Réciproquement, la propriété de simulation inverse stipule que toute exécution de la fonction auto_nav en langage Clight

correspond à un comportement de la sémantique FP : toutes les exécutions sont autorisées par la sémantique.

La vérification du Théorème iii.1 a été divisée en preuves plus petites en prouvant la bisimulation pour les différentes étapes du générateur, c'est-à-dire entre FP-FPE (voir VerifFPToFPE.v), FPE-FPS (voir VerifFPEToFPS.v) et FPS-FPC (voir VerifFPSToFPC.v). Ces preuves ont finalement été combinées en utilisant la propriété de composition de la bisimulation. La preuve globale du Théorème iii.1 peut être trouvée dans le fichier Coq VerifFPToFPC.v.

Remarque

La Section 10.2 présente les différentes étapes intermédiaire et le théorème global.

Terminaison

Nous voulons nous assurer que le pilote automatique peut appeler la fonction auto_nav sans risque d'être bloqué en raison d'une boucle infinie. En utilisant le Théorème iii.1, la preuve de terminaison du code C équivaut à s'assurer que la fonction step de FP se termine (voir Section iii.1.4). Comme Coq n'autorise pas la définition de fonctions non-terminantes, la définition de la fonction de sémantique step de FP en Gallina prouve la terminaison de la fonction auto_nav. L'implémentation de cette fonction peut être trouvée dans le fichier Coq FPBigStep.v.

Remarque

La Section 10.3 détaille cette étape de vérification de terminaison de la fonction auto_nav.

iii.2.3 Hypothèses de vérification

La preuve du théorème de préservation de la sémantique repose sur une modélisation de notre système et certaines hypothèses. Le but de cette section est de présenter ces choix afin de donner confiance dans le nouveau générateur vérifié. La section iii.2.3 résume le modèle utilisé pour définir le théorème principal. La section iii.2.3 présente tous les axiomes définis pour vérifier ce théorème.

Remarque

Cette section est un résumé de la Section 10.4.

Modèles du système

L'autopilote de drone est un système complexe évoluant dans un environnement extérieur réel. La fonction auto_nav interagit avec l'autopilote de drone et indirectement avec l'environnement extérieur. Une propriété idéale que nous pourrions vouloir prouver est que le drone se comportera toujours comme défini dans le plan de vol. Malheureusement,

prouver cette propriété nécessiterait de représenter concrètement l'interaction entre le plan de vol, l'autopilote de drone et l'environnement extérieur, ce qui est complexe et prend du temps. À la place, nous avons prouvé que l'exécution de la fonction `auto_nav` interagira avec l'autopilote de drone comme défini dans le plan de vol, sans aucune garantie supplémentaire que l'autopilote se comportera correctement. Comme présenté dans la Section iii.1.4, nous modélisons l'environnement du drone par deux éléments : *fp_state* qui représente les états internes du plan de vol et *fp_trace* qui représente les interactions entre le plan de vol et l'autopilote. Ces interactions peuvent être des appels aux fonctions de l'autopilote, mais aussi à du code C arbitraire défini par l'utilisateur (par exemple, le code exécuté dans les étapes **`CALL`**). Nous supposons donc qu'ils ne modifieront pas l'état interne du plan de vol.

Nous ajoutons également une autre hypothèse pour prouver la terminaison de la fonction `auto_nav`. Comme l'exécution du plan de vol dépend de l'exécution d'un code C arbitraire qui ne peut pas être vérifié *a priori*, la terminaison de la fonction `auto_nav` est assurée à condition que le code C arbitraire se termine finalement, ce que nous considérons comme établi par d'autres moyens[10].

Axiomes

Nous utilisons une fonction de paramètre appelée `create_ident` qui prend une chaîne de caractères et produit un `ident` Clight lors de la génération de code. Les `idents` Clight sont utilisés dans la syntaxe Clight pour décrire les noms de variables ou de fonctions. Cette fonction est un `Parameter` Coq, c'est-à-dire qu'elle est définie en OCaml et liée lors de l'extraction. Elle doit être définie en OCaml car la chaîne de caractères correspondante est stockée dans une table de hachage. La fonction `PrintClight` utilise cette table de hachage pour imprimer le nom correspondant des variables et des fonctions. Comme cette fonction est définie en dehors de Coq, il n'est pas possible de prouver des propriétés à son sujet. Nous ajoutons donc un axiome indiquant que la fonction `create_ident` est injective, ce qui est raisonnable compte tenu de l'implémentation OCaml. La déclaration de la fonction et de l'axiome se trouve dans le fichier `ClightGeneration.v`.

La propriété `step` de la sémantique de `FPC` exécute de manière symbolique le code Clight de la fonction `auto_nav` à partir d'un état initial de la mémoire et se termine dans un état final de la mémoire. Pendant l'exécution du programme, la sémantique Clight produit une liste de `trace` qui correspond à la *fp_trace* de *fp_env*. La liste de `trace` doit donc être générée par l'exécution du code C arbitraire du plan de vol. Dans la sémantique Clight, les `traces` sont seulement générées pour les instructions qui exécutent des appels de fonctions externes ou intégrées. Cependant, le code C arbitraire dans Paparazzi n'est pas limité à ces constructions. Par exemple, un tel code peut apparaître dans la condition d'une instruction conditionnelle. De plus, il n'est pas possible d'utiliser les résultats de la fonction *evalc* dans la sémantique Clight. Nous définissons donc 4 axiomes qui sont disponibles dans les fichiers `ClightLemmas.v` et `CommonFPVerification.v`. En résumé, ces axiomes peuvent être vus comme une extension de la sémantique Clight qui prend en compte le modèle défini

[10] L'analyse WCET présentée dans [HBL+22a] montre que sur un exemple réel, la fonction `auto_nav` se termine toujours en moins de temps que sa période d'appel.

dans la section précédente en ajoutant principalement deux cas : 1) l'appel à une fonction vide produit une trace, 2) l'appel à une fonction avec un résultat booléen renvoie une valeur en utilisant *evalc* et cet appel produit également une trace.

En utilisant ces axiomes, nous n'avons pas besoin d'utiliser le mécanisme d'appel externe de Clight. Ainsi, nous définissons un autre axiome stipulant que l'exécution de code externe à partir du même état produit toujours la même trace (voir le fichier `FPBigStepClight.v`). Cet axiome permet de prouver que la sémantique de Clight est déterministe. Pour la preuve du Théorème iii.1, nous avons d'abord prouvé la simulation directe, puis nous avons utilisé la propriété selon laquelle Clight est déterministe pour prouver la simulation inverse.

Enfin, nous utilisons certains axiomes classiques de la bibliothèque standard de Coq, tels que le tiers exclu, l'irrélevance de la preuve (deux preuves sont égales si elles prouvent la même propriété) et l'extensionnalité fonctionnelle (l'égalité des fonctions est l'égalité point par point). Ces axiomes sont particulièrement nécessaires lors de la manipulation de types dépendants, voir Section iii.3.2.

iii.3 Les leçons apprises

Pendant le développement du projet, nous avons tiré plusieurs leçons que nous considérons précieuses pour ceux qui souhaitent prouver un compilateur avec Coq. La Section iii.3.1 présente quelques retours sur le processus de développement en Coq, et la Section iii.3.2 présente quelques remarques techniques sur l'utilisation de Coq, Clight et MathComp.

Remarque

Cette section est un résumé du Chapitre 11.

iii.3.1 Méthodologie de développement

La vérification d'un compilateur à l'aide d'un assistant de preuve comme Coq est une technique bien testée et éprouvée. Certains des projets l'utilisant sont suffisamment matures pour être utilisés dans le développement de logiciels critiques (voir par exemple CompCert [LBK+16]). Même si sa complexité ne peut pas être comparée à des projets comme CompCert ou Velus, le générateur de code FPL n'est pas un exemple anodin.

Le contexte de l'autopilote Paparazzi a restreint notre développement formel de plusieurs manières. Tout d'abord, le langage d'entrée FPL était fixé et déjà en cours d'utilisation, nous avons donc été obligés de conserver ses particularités[11] et de proposer uniquement des extensions conservatrices, au lieu d'une refonte globale qui aurait pu faciliter l'effort de vérification. Deuxièmement, nous étions également liés à une structure de code C spécifique sur laquelle les utilisateurs peuvent compter. Troisièmement, nous devions au plus tôt élaborer une sémantique exécutable et la présenter aux concepteurs originaux de FPL pour validation et feedback, ce qui nous a conduit à une sémantique FP simple sans modèle formel du code C externe.

[11] Par exemple, l'état *fp_state* ne stocke qu'une seule position précédente. Si nous exécutons deux **DEROUTE**, puis deux étapes **RETURN**, la première étape de retour revient de la deuxième déroute comme prévu, mais la deuxième étape de retour ne fait rien.

Le développement d'un tel projet avec Gallina nous a contraints à une clarification approfondie de certains détails sémantiques assez fastidieux et a révélé incidentellement un certain nombre de problèmes dans le compilateur d'origine. Par exemple, l'écriture de la preuve nous a permis de découvrir un bogue lorsqu'il y a plus de 256 blocs ou étapes. De plus, dans la sémantique d'origine de `FPL` basée sur le générateur précédent, nous avons constaté que l'étape **`DEROUTE`** a un comportement inattendu : lorsqu'elle est exécutée, la position actuelle est stockée dans `last_block` et `last_stage`, et le bloc actuel devient le bloc dérouté. Par conséquent, lorsqu'une étape **`RETURN`** est exécutée dans le bloc dérouté, l'exécution reprend à la position stockée, de sorte que la même étape **`DEROUTE`** est exécutée à nouveau et que le programme entre dans une boucle infinie. Ce problème n'a été découvert qu'au cours de la formalisation de la sémantique, car l'étape **`RETURN`** est rarement utilisée par les utilisateurs de Paparazzi, et quand elle l'est, c'est uniquement pour reprendre l'exécution du plan après une exception.

Enfin, la vérification du générateur en assurant la préservation de la sémantique est très chronophage, et les changements en cours de route doivent être évités autant que possible. En effet, le générateur de code a été divisé en trois passes indépendantes de manière à ce que l'effort de preuve pour chaque passe soit gérable, mais aussi de manière à ce que au moins les différentes passes ne sont pas susceptibles de changer, quelles que soient les difficultés de développement des autres passes.

iii.3.2 Remarques techniques

Pour définir la syntaxe de `FP` et `FPE`, nous avons naturellement souhaité réutiliser les types et les bibliothèques Coq, tels que list. Nous avons donc défini le plan de vol comme présenté dans la section iii.1.1, mais nous avons rencontré un problème lors de l'induction sur le type *fp_stage* récursif et imbriqué. Le principe d'induction généré automatiquement par Coq ne tient pas compte de l'interaction entre *fp_stage* et list. Nous avons donc dû définir notre propre principe d'induction : prouver une propriété P pour un *fp_stage* nécessite de prouver P pour toutes les étapes imbriquées différentes.

Dans la Section iii.1.4, dans un souci de lisibilité, la sémantique de `FP` est donnée sous forme d'une présentation relationnelle avec des règles d'inférence, mais nous avons décidé d'implémenter la sémantique en Coq sous forme d'une fonction *step* à des fins de validation précoce. Un interprète Ocaml est ensuite obtenu par simple extraction de la définition Coq. Le fait d'avoir une fonction permet également d'automatiser et de simplifier les preuves en utilisant la normalisation par évaluation au lieu d'appliquer manuellement de nombreuses règles de réécriture. Un autre point important est que nous souhaitions éviter d'adopter de nouveaux axiomes de manière gratuite concernant la correction des phases non-Gallina, telles que la préparation. Nous avons donc utilisé des types dépendants (c'est-à-dire un type de base avec certaines propriétés) associés à des fonctions de vérification qui injectent une structure à partir du type de base si sa propriété de correction est vérifiée, ou renvoient une erreur dans le cas contraire. Par exemple, nous avons défini le type des nombres naturels représentables sur 8 bits pour `FPS` ainsi que le type des plans de vol de taille et de numérotation bien définies (cf. propriété H_{ws} dans la Section iii.2.1).

Nous avons utilisé Clight comme langage de sortie car il présente de nombreux avantages. Tout d'abord, il offre une sémantique fiable pour le langage C. Cependant, la sémantique Clight pour les appels externes ne correspond pas à notre modèle. Nous avons donc dû définir notre propre sémantique Clight pour le code C arbitraire, comme présenté dans la Section iii.2.3. De plus, nos choix de modélisation séparent naturellement les états mémoire des plans de vol des états des drones (représentés par la trace des événements), ce qui nous évite de travailler avec la logique de séparation.

Deuxièmement, CompCert propose l'outil `clightgen` qui prend un fichier C et le convertit en une structure Clight. Cet outil peut aider l'utilisateur à comprendre Clight et ses différences avec C en traduisant des programmes C existants. Par exemple, l'opérateur booléen `&&` et certaines autres constructions n'existent pas en Clight (voir Section iii.1.5).

CompCert offre également deux sémantiques : une sémantique à petits pas et une sémantique à grands pas. D'une part, la sémantique à grands pas est la sémantique naturelle qui peut décrire le modèle d'exécution jusqu'à la fin d'un programme. Cette sémantique est plus facile à utiliser, mais malheureusement, elle ne définit pas le comportement de toutes les instructions Clight, telles que `goto`, qui est utilisée pour les boucles. D'autre part, la sémantique à petits pas décrit le comportement de toutes les instructions Clight. Cependant, la sémantique à petits pas ne décrit qu'une étape d'exécution, et le code restant à exécuter est stocké dans une continuation. L'exécution d'un programme complet correspond donc à une succession de petites étapes, ce qui peut être fastidieux pour la vérification de la préservation de la sémantique, car notre propre sémantique est directe et ne fait pas intervenir de continuations. Il est également possible de produire du code C avec le module `PrintClight` de CompCert. Cependant, comme présenté dans la section iii.1.5, le module ne produit pas de code C compilable et certaines transformations doivent être effectuées.

En plus de l'implémentation de CompCert, nous avons également utilisé MathComp, une bibliothèque Coq de mathématiques formalisées. Nous avons utilisé quelques lemmes MathComp sur l'arithmétique des nombres naturels, mais nous avons principalement utilisé `seq`, une bibliothèque sur les listes qui fournit de nombreuses fonctions et lemmes utiles, tels que la fonction `drop` qui supprime les premiers éléments d'une liste. Nous avons également utilisé SSReflect, un langage de preuve qui permet de simplifier les preuves par rapport aux tactiques de preuve standard de Coq. Le principal problème que nous avons rencontré en utilisant MathComp concerne l'extraction de programmes. En effet, la bibliothèque MathComp n'est pas conçue pour être extraite en code Ocaml car elle repose sur des constructions qui ne sont pas prises en charge. Nous avons donc dû spécifier manuellement les seules fonctions qui devaient être extraites de MathComp.

iv Conclusion

L'objectif de cette thèse était de passer en revue différents processus de vérification utilisant des outils formels pour vérifier des composants critiques du pilote automatique Paparazzi et garantir les propriétés de correction. Au cours de ces 3 années de doctorat, nous avons réussi à vérifier 2 composants critiques. Tout d'abord, nous avons vérifié l'absence d'erreurs

d'exécution et certaines propriétés fonctionnelles d'une bibliothèque mathématique en langage C. Ce travail est présenté dans la Partie I et résumé dans la Section iv.1. La deuxième partie de la thèse, résumée dans la Section iv.2 et détaillée dans la Partie II, concerne la vérification, à l'aide de Coq, du générateur de plan de vol qui traduit un plan de vol XML en code C embarqué.

En plus de fournir des composants vérifiés formellement aux utilisateurs de Paparazzi, cette thèse vise à déterminer si ces processus pourraient être utilisés dans des projets existants. Nous résumons par la suite les avantages et les inconvénients que nous avons découverts au cours de notre travail. Nous discutons également des travaux futurs possibles.

iv.1 Une bibliothèque mathématique pour la représentation d'états

L'autopilote Paparazzi est composé d'une *Interface État* qui est une boîte noire reliant les données acquises par les capteurs et les filtres d'estimation au système de contrôle en utilisant ces données pour prendre des décisions de navigation. La caractéristique principale de cette interface d'état est de proposer plusieurs représentations pour une donnée donnée et des fonctions de conversion si nécessaire. Cette interface est donc un composant critique de l'autopilote car des erreurs lors de la conversion peuvent entraîner de mauvaises décisions prises par le système de navigation, ce qui peut entraîner un crash du drone dans le pire des cas.

Au cours de cette thèse, nous nous sommes concentrés sur la vérification d'une bibliothèque mathématique C utilisée dans l'interface étatique pour la manipulation des représentations d'attitude ou de position. Cette bibliothèque fournit plusieurs représentations (angles d'Euler, matrices de rotation et quaternions) ainsi que des fonctions pour manipuler et convertir ces représentations. Les représentations sont également définies dans trois formats différents pour représenter les valeurs réelles : point fixe, virgule flottante ou double.

Pour vérifier cette bibliothèque C, nous avons utilisé la plateforme Frama-C associée à son plugin RTE (pour ajouter des assertions correspondant aux erreurs d'exécution, c'est-à-dire la division par zéro, le déréférencement de pointeur nul, le dépassement de capacité, etc.), son plugin WP (pour appliquer des méthodes déductives) et son plugin EVA (pour appliquer l'interprétation abstraite). Notre travail a d'abord consisté à vérifier l'absence d'erreurs d'exécution, en déterminant des contrats minimaux pouvant garantir les assertions ajoutées par le plugin RTE. Nous avons dû associer les plugins WP et EVA pour vérifier les assertions car ils sont complémentaires : EVA est capable de gérer les pointeurs et de calculer les intervalles de valeurs possibles, tandis que WP est capable de prouver les annotations de boucle. La deuxième partie de la vérification de la bibliothèque consistait à garantir les propriétés fonctionnelles de ces fonctions. Nous nous sommes principalement concentrés sur les fonctions de conversion plutôt complexes et nous avons vérifié que l'implémentation est mathématiquement correcte. En effet, comme nous avons utilisé le modèle arithmétique «réel» de WP, nous pouvons conclure que la fonction est correcte si nous ne tenons pas compte des erreurs d'arrondi dues aux calculs en virgule flottante. Pour prouver ces propriétés, nous avons principalement utilisé des solveurs

Implémentation	# lignes			
	Code	Annotations	Pourcentage	Preuves Coq
`int`	1150	903	44.0%	-
`double`	358	199	35.7%	-
`float`	1708	1557	47.7%	6472*
Total	3216	2659	45.3%	6472*

*Nombre total de lignes de code Coq, mais la majorité d'entre elles ont été générées automatiquement.

Figure 6: Nombre de lignes de code et annotations ajoutées.

automatiques tels que Alt-Ergo, CVC4 ou Z3, mais pour certaines propriétés spécifiques, nous avons dû les prouver manuellement à l'aide de Coq.

La figure 6 montre le nombre de lignes de code des fichiers initiaux (avec du code et des commentaires mais sans les lignes vides) ainsi que le nombre de lignes d'annotations ajoutées. L'ajout d'annotations double presque le nombre de lignes. Cette augmentation massive est principalement due aux spécifications mathématiques qui constituent en quelque sorte une deuxième définition des fonctions C. La figure 12.1 montre également le nombre de lignes de code des fichiers Coq, mais la majorité de ces fichiers sont générés automatiquement par Frama-C, et donc la preuve manuelle est beaucoup plus petite que 6472 lignes. Enfin, la figure 12.1 ne prend pas en compte les scripts WP générés automatiquement lors de la sauvegarde des tactiques/résolveurs appliqués aux objectifs vérifiés, représentant environ 8746 lignes.

Dans nos travaux futurs, nous prévoyons de compléter les preuves restantes présentées dans la section ii.3. Certaines propriétés fonctionnelles d'autres fonctions de la même bibliothèque mathématique de Paparazzi devront également être vérifiées. Nous souhaitons particulièrement nous concentrer sur la vérification des erreurs d'arrondi, et donc ne pas utiliser le modèle `real` de WP, mais plutôt un modèle qui représente précisément les nombres à virgule flottante. Nous devrons également comparer notre approche aux *preuves autoactives*, où les prouveurs SMT sont guidés par des assertions insérées par les développeurs pour les aider [BLK19, DM17].

iv.2 Générateur de plan de vol

Les plans de vol décrivent les missions spécifiques que le drone doit effectuer de manière autonome. Ces plans de vol sont exprimés à l'aide d'un langage spécifique au domaine basé sur XML de haut niveau. À l'origine, Paparazzi utilisait un générateur OCaml qui convertissait les missions en code C directement intégré dans le drone. Le générateur est un composant critique de Paparazzi car il doit produire un code qui se comporte exactement comme spécifié par l'utilisateur afin d'éviter des comportements indésirables.

La partie II, résume dans la Section iii, présente le développement et la vérification formelle d'un nouveau générateur de plans de vol pour l'autopilote Paparazzi. Ce nouveau générateur est entièrement compatible avec l'ancien générateur non prouvé mais prend en charge de nouvelles fonctionnalités telles que les déroutements interdits et détecte les

	Lignes de code	Pourcentage
Parser Coq	2789	13.1%
Préprocesseurr	2174	8.6%
Générateur Coq	20238	80.1%
Postprocesseur	60	0.3%
Total	25261	100%

Figure 7: Lignes de code pour les différents blocs du générateur.

plans de vol erronés qui étaient auparavant ignorés. Au cours des premières étapes de développement, nous avons effectué des tests montrant que le code C généré par les deux générateurs se comportent de manière similaire sur plusieurs exemples. Ce générateur est composé de 4 blocs principaux et la Figure 7 montre leurs différences de taille en termes de lignes de code[12]. Notez que le générateur OCaml d'origine comportait environ 1200 lignes de code et que les deux projets utilisent du code Paparazzi commun, représentant environ 3000 lignes de code. Le parseur vérifié, écrit en utilisant `menhir` avec l'option `-coq`, lit le fichier XML afin de produire une structure *Flight Plan Parsed*. Cette structure est ensuite légèrement transformée par un préprocesseur OCaml en une structure Coq `FP`. Ensuite, les transformations majeures sont effectuées dans le générateur Gallina vérifié, qui représente 80% du projet. Le générateur convertit la structure `FP` en code Clight correspondant. Enfin, le code Clight est imprimé et quelques transformations mineures sont effectuées par le postprocesseur OCaml, afin de produire du code C compilable. Le préprocesseur et le postprocesseur sont les seules parties du générateur qui ne sont pas formellement vérifiées. Cependant, nous sommes confiants que ces passes ne remettent pas en question la correction du générateur car elles ne font que des transformations mineures, et elles ne correspondent qu'à une petite partie du générateur.

Le générateur Gallina correspond à 80% du projet VFPG, car nous comptons le code générant le code Clight, ainsi que la spécification et la preuve de correction. Le générateur est un compilateur à trois passes. La première passe étend le plan de vol afin d'avoir une structure plus proche du code C qui sera généré. Ensuite, la deuxième passe vérifie que le plan de vol n'est pas trop grand pour éviter les débordements, mais vérifie également les transformations du préprocesseur, telles que la numérotation des blocs. Une fois vérifiées, ces propriétés sont conservées dans les étapes restantes car elles sont nécessaires pour mener à bien certaines preuves. Enfin, la troisième passe génère le code Clight. Nous avons prouvé indépendamment que chaque passe préserve la sémantique et avons utilisé un lemme de composition pour prouver la correction de l'ensemble du générateur Gallina. La principale difficulté rencontrée était de gérer le code C arbitraire que les utilisateurs peuvent ajouter aux plans de vol. Nous avons décidé d'ajouter une hypothèse : le code C arbitraire fourni par l'utilisateur se termine toujours et ne modifie pas l'état interne utilisé pour l'exécution du plan de vol. Sous cette hypothèse, le code C généré se comportera comme le plan de vol d'entrée.

[12]Nous ne comptons que les lignes du projet VFPG et pas les fichiers externes utilisés tels que `PrintClight.ml` de CompCert.

	Compilation	Spécification	Preuves	Total
Flight Plan (FP)	84	296	0	380 (1.9%)
Passe d'extension	416	1306	2239	3456 (19.5%)
Passe de vérif. taille	172	1124	2160	3456 (17.1%)
Génération Clight	1542	2239	4491	8272 (40.9%)
Commun	616	1689	1864	4169 (20.6%)
Total	2830 (14.0%)	6654 (32.9%)	10754 (53.1%)	20238 (100%)

Figure 8: Lignes de code Coq dans le générateur Gallina (commentaires et espaces exclus).

La figure 8 montre le nombre de lignes de code Coq pour FP, les différentes phases et le code commun. La plus grande partie concerne la génération Clight car la transformation de la structure du plan de vol en code Clight n'est pas directe et les preuves de préservation sémantique nécessitent d'exécuter manuellement le code Clight à travers sa sémantique. Dans la figure 8, nous divisons également le nombre de lignes de code en plusieurs catégories pour mettre en évidence que le code Coq fonctionnel ne représente que 14% du projet, le reste étant des spécifications ou des preuves. La preuve de propriétés générales sur le langage, telles que l'exécution correcte des primitives `on_enter` et `on_exit`, est comptée dans la catégorie «Commun».

En tant que travail futur, nous envisagerons de soutenir de nouveaux types de messages d'erreur et d'avertissement. Nous sommes actuellement en train de spécifier et de mettre en œuvre des messages d'avertissement lorsque des déroutes interdites bloquent le système. Nous voulons prévenir les utilisateurs lorsqu'il y a une `déroute` (ou une exception potentielle) vers un bloc explicitement interdit dans le plan de vol. Lorsque l'étape de `déroutement` est exécutée, en fonction de sa condition, le changement de bloc peut être interdit et le plan de vol peut rester bloqué dans une impasse.

Nous pouvons également exploiter davantage la propriété de bisimulation entre les langages d'entrée, intermédiaires et de sortie. Cela a déjà été fait pour la propriété selon laquelle les fonctions `on_enter` et `on_exit` sont correctement exécutées, mais nous pouvons l'étendre à d'autres propriétés, par exemple, s'il n'y a aucune alerte générée, alors il n'y a aucun risque de blocage. En utilisant la propriété de bisimulation, nous pouvons également améliorer la génération de code. Nous voulions initialement produire le même code que le générateur OCaml pour des raisons de compatibilité, mais aussi parce que la sémantique de FP a été définie en fonction du code généré par le générateur OCaml. Nous avons prouvé que la sémantique de FP est préservée par le code C correspondant, lui-même similaire au code C généré par le générateur OCaml d'origine, comme cela peut être attesté par l'examen du code et les tests. Nous pouvons donc concevoir un nouveau générateur produisant un code plus optimisé avec une structure complètement différente. Si nous sommes capables de prouver une bisimulation entre ce nouveau générateur et la sémantique de FP, nous avons donc une grande confiance que le code généré se comportera comme le générateur OCaml. De plus, nous avons plusieurs idées pour améliorer la confiance dans le générateur.

Premièrement, nous voulons réduire autant que possible le nombre de transformations non vérifiées, telles que les phases de prétraitement et posttraitement OCaml. Par exemple, nous pouvons envisager de connecter directement notre générateur à CompCert pour avoir un compilateur entièrement vérifié qui traduit `FPL` en code assembleur, à condition que l'architecture cible soit prise en charge par CompCert. Dans ce cas, le postprocesseur ne serait plus utilisé. Deuxièmement, le code généré est en réalité séparé du code C commun. Cependant, pour la preuve de vérification, nous utilisons une version Clight de ce code C commun. Comme Clight est un sous-ensemble strict du langage C, le code Clight utilisé pour la vérification est donc différent du code C commun utilisé pendant la compilation. Nous pourrions donc envisager de générer un seul programme contenant tout le code C commun en plus du code du plan de vol tel que `auto_nav`. Troisièmement, nous pourrions vouloir supprimer certains axiomes, en particulier ceux utilisés pour relier la sémantique opérationnelle de Clight à la sémantique dénotationnelle de `FP`, en particulier en ce qui concerne la gestion des appels externes. Nous pourrions envisager de redéfinir la sémantique de `FP` comme une sémantique opérationnelle qui utilise le même mécanisme que Clight pour les appels externes. Enfin, nous pourrions vouloir ajouter un analyseur syntaxique C pour un code C arbitraire. Cela pourrait être utilisé pour s'assurer que le code C est valide et restreindre les utilisateurs à certaines constructions spécifiques.

Enfin, nous voulons modulariser le générateur en séparant les modes de navigation du plan de vol. L'idée est d'offrir aux utilisateurs une interface générique pour spécifier leurs propres modes de navigation. S'ils prouvent que la sémantique est préservée pour leurs modes de navigation, alors ils disposent d'un générateur de code C personnalisé adapté à leurs besoins. Il sera alors possible de prendre en charge facilement de nouveaux pilotes automatiques ou d'utiliser le générateur dans un autre contexte qui utilise des programmes synchrones tels que les machines à états finis.

v Conclusion générale et travaux futurs

Grâce à cette thèse, nous avons eu l'occasion de passer en revue deux processus de vérification formelle différents. Tout d'abord, nous avons utilisé l'analyse statique pour vérifier du code existant en garantissant l'absence d'erreurs d'exécution et certaines propriétés fonctionnelles. Ensuite, nous avons développé un nouveau programme et prouvé son fonctionnement pour remplacer un ancien programme non vérifié. Les deux processus apportent des garanties solides au programme vérifié, mais ils présentent tous deux les mêmes limitations.

Tout d'abord, les garanties offertes sont basées sur des hypothèses. En effet, avec les outils actuels, il est impossible d'affirmer que lorsqu'un drone autonome avec une mission complexe est lancé, il se comportera toujours comme prévu. La grande question pour chaque processus de vérification formelle est de déterminer sous quels modèles et hypothèses raisonnables les activités de vérification doivent avoir lieu. Cette décision est cruciale et est trop souvent négligée, bien qu'elle puisse remettre en question de manière importante la pertinence du travail de vérification effectué si les hypothèses sont irréalistes ou contradictoires.

Ensuite, les deux outils nécessitent un certain niveau d'expertise en méthodes formelles,

même les outils «automatiques» tels que Frama-C et les solveurs SMT. Ce niveau d'expertise concerne à la fois la spécification et l'utilisation de l'outil. La spécification doit être rigoureusement élaborée pour capturer formellement la propriété exacte à vérifier, mais elle doit également être rédigée en utilisant des encodages et des représentations adaptées à l'outil. Par exemple, il existe plusieurs façons en Coq de définir une fonction spécifique, mais en fonction des besoins, certaines d'entre elles sont plus appropriées que d'autres. De plus, un niveau minimal d'expertise est requis pour tout outil formel. Par exemple, comme présenté dans la Partie I et résumé dans la Section ii, Frama-C utilisé avec des solveurs n'est pas capable de vérifier certains objectifs. La question est alors de déterminer si le problème vient d'une spécification incorrecte ou incomplète qui doit être modifiée, ou des solveurs qui *timeout* sur une propriété trop compliquée et doivent être aidés (e.g. avec des tactiques) ou si la preuve doit être réalisée manuellement avec Coq. C'est une question complexe qui peut prendre du temps à répondre pour les personnes sans expertise dans des outils tels que Frama-C.

Finalement, l'utilisation de méthodes et d'outils formels a un coût élevé en termes de maintenabilité. En effet, les deux projets ne fonctionnent actuellement pas avec la dernière version des outils Frama-C ou Coq. La vérification de la bibliothèque mathématique a été effectuée au début de la thèse et depuis, plusieurs versions de Frama-C et de leurs solveurs associés ont été publiées. Il existe maintenant des objectifs qui ne peuvent plus être vérifiés. Nous ne connaissons pas la raison de cela, et cela prendra certainement beaucoup de temps pour corriger les objectifs non prouvés. Le générateur vérifié a été mis à jour pour Coq 8.16 mais la dernière version actuellement disponible (Coq 8.17) n'est pas capable de terminer la preuve. Après enquête, il semble que la nouvelle version puisse automatiquement prouver certains objectifs et certains paramètres pour les prédicats qui n'ont plus besoin d'être spécifiés explicitement. La preuve pourrait donc être corrigée, mais cela prendra du temps. La maintenance des preuves est un problème essentiel pour l'adoption de méthodes formelles dans l'industrie. Même si ce problème est déjà étudié [Tal21] avec des solutions de réparation de preuves, c'est un point à prendre en compte lors de la sélection d'un outil de vérification formelle.

Dans les futurs travaux, il serait intéressant de vérifier d'autres composants critiques ou d'appliquer de nouveaux processus de vérification formelle. Par exemple, nous pourrions vérifier le générateur de code autopilote Paparazzi : Paparazzi propose un générateur de code C pour l'autopilote (disponible uniquement pour les rovers pour le moment). Les utilisateurs peuvent définir le comportement de leur autopilote en utilisant des machines à états décrites dans un fichier XML. De la même manière que la vérification du générateur de plan de vol, nous pourrions également prouver que la génération du code C est correcte. Dans ce cas, nous pourrions définir directement une sémantique opérationnelle comme le fait Clight et utiliser la logique de séparation. Il pourrait également être intéressant de vérifier d'autres parties du code C critique du système de contrôle ou du système d'exploitation en temps réel en utilisant la vérification de modèles avec des outils tels que CBMC [KST23].

Chapter 1

Introduction

Verification is a crucial stage during the development process of a system or a program, especially in a critical context, to prevent dramatic events such as the explosion of the Ariane 5 rocket [LLF+96]. This verification phase ensures that the system meets its specification, and in particular that unwanted behaviour will never occur. Historically, the main verification technique used is testing: consider several inputs for the system and verify that each one of them produces the expected results. However, as Edsger Disjktra [Dij72] wrote:

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

Formal methods are mathematical theories, techniques and tools to formally prove the properties of hardware, software, or models. They offer stronger guarantees than tests as they can ensure that the system satisfies a property instead of checking if some inputs respect this property. There are many formal methods domains and they can be classified by the properties they can help to verify, the efforts required to specify the system in order to use the verification tools or their automation level. For instance, *abstract interpretation* is often used to prove the absence of runtime errors and is an automatic tool. *Deductive verification* is another technique that can be used to prove more complex properties, such as the correctness of a program given a formal specification, and generally uses automated solvers, though sometimes needs to use a proof assistant and requires human intervention.

Formal methods are nowadays widely accepted as an efficient complement to testing, particularly for critical systems such as in aerospace [KWN+10], automotive, medical and cybersecurity [Jae10]. There are now safety standards that recommend the use of formal methods (for example, the DO-333 supplement to the DO-178C document used by certification authorities for commercial aircraft systems). However, the definition of verification processes for industrial uses is rather complex due to their multiple constraints. Indeed, the processes should use formal verification tools that can be scalable on large projects which are not specifically designed for these tools. Moreover, the processes might be applied by engineers that may not be experts in formal methods.

Paparazzi [HBG14, Pap21] is an open-source autopilot for *Unmanned Aerial Vehicle* (*UAV*) under a GPL license developed at ENAC, the French Civil Aviation University, since 2003. Paparazzi is a complete drone control system for autonomous vehicles that offers a control software part and some design of hardware components. This autopilot supports various types of drones (quadcopter drones, fixed-wings, rovers...) and permits the control of several of them simultaneously. Paparazzi has also various built-in modes and offers the ability to create customised flight plans. Several core components of Paparazzi are critical

as a malfunction in one of them can cause unwanted behaviour or the crash of the drone. If we want users to have strong confidence in Paparazzi, we must ensure that its components are correct and will not fail.

The goal of this thesis is to review different verification processes using formal tools in order to verify correctness properties on critical components of Paparazzi. In addition to providing formally verified components to Paparazzi users, this thesis aims to verify if the processes are worth being used on existing projects. Indeed, Paparazzi is a good case study containing a large code base (350,000 lines of code) written without verification purpose by good programmers who use classic idioms of the C programming language (pointers, unions, etc.).

This thesis is organised as follows. Chapter 2 presents a state of the art on formal verification with a general introduction on verification processes and formal semantics used by formal methods. The rest of the document is composed of two independent parts:

- Part I presents an effort to verify a mathematical library written in C using static analysis techniques,
- Part II presents the formal verification of a flight plan generator using a proof assistant.

Chapter 12 concludes the work presented in this document and gives some perspective on future work.

The remainder of the present introduction chapter briefly presents the Paparazzi autopilot in Section 1.1 and the work done in Part I (respectively in Part II) in Section 1.2 (respectively in Section 1.3).

Coq implementation

Every code developed in this thesis is publicly available in the two following GitLab projects:

- Mathematical library: `gitlab.isae-supaero.fr/b.pollien/paparazzi-frama-c`
- Flight plan generator: `gitlab.isae-supaero.fr/b.pollien/vfpg`

Throughout this document, we provide clickable links that directly refer to the corresponding file or definition in the GitLab project. We also add clickable links for every function, type or operator that refer to its original definition or notation.

1.1 The Paparazzi UAV autopilot

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

UAVs were originally developed by the army to be sent to risky missions instead of humans in order to protect their lives. Now, drones have become pervasive in several application domains: law enforcement, agriculture, construction, recreative purposes like taking photos or videos, etc. There are also future projects of delivery drones for medical supply or public transport. In order to be able to use drones for these large-scale projects, it will be necessary for drones to be autonomous and *reliable*, i.e. to be tested and validated rigorously by using, for example, formal verification techniques.

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

The case study chosen for this thesis is Paparazzi [HBG14], an open-source autopilot (under GPL license) whose development started in 2003 at ENAC. It is mainly designed for autonomous flight but supports manual control. It has also been developed to support different types of drones and is able to control multiple UAVs simultaneously from a unique system.

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

As shown on Figure 1.1, the different components of Paparazzi autopilot are separated between critical components (such as AHRS, IMU, INS and GPS) and non-critical components, with statically defined periodic time-slots in a sequential order. The latest operating system used is based on ChibiOS (an efficient and light RTOS for microcontrollers). The main part of the autopilot is handled in a single thread, but most of the low-level drivers have their own threads, with either high priority (communication with internal sensors) or low priority (Secure Digital logging) compared to the autopilot task. It is also possible to create threads for certain heavy tasks related to payload management or parameter estimation in dedicated modules. This ensures a proper timing for the core autopilot control stack.

In addition, the data between the sensors and the estimation filters are managed by a publisher/subscriber mechanism, where the estimated state is published in a blackboard type interface for the control and navigation loops. Conversion between the different attitude or position representations (Euler, rotation matrix, quaternion, latitude/longitude, ECEF, etc.) is automatically handled by this state interface.

The control loop is a core attitude control loop with the possibility of other control

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

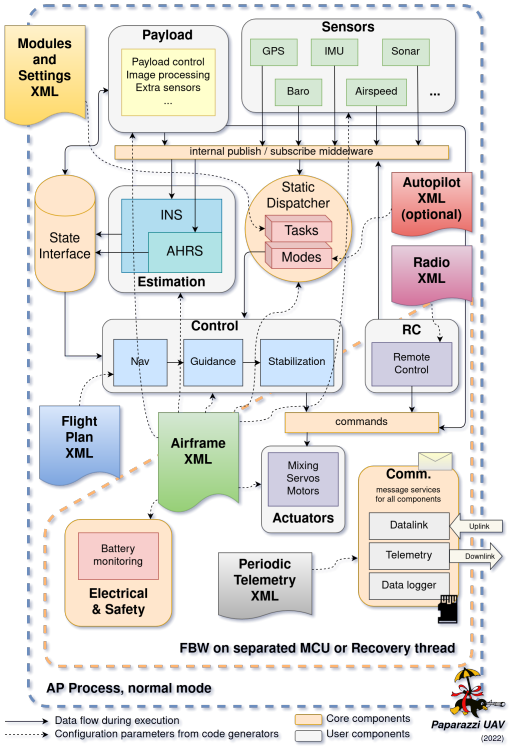

Figure 1.1: Architecture of the Paparazzi autopilot (embedded code)

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

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

1.2 A mathematical library for state representation conversion

Drone autopilots use a range of sensors to collect data about the drones environment, e.g. GPS, accelerometers or gyroscopes. The autopilot then uses this data to make decisions about how to control the drone, in order to achieve its mission. The correctness and the precision of the data are then crucial as the decisions taken by the autopilot should be consistent with its real environment.

[2]See http://wiki.paparazziuav.org/wiki/Flight_Plans for the documentation of flight plans.
[3]For more details on missions, see http://wiki.paparazziuav.org/wiki/Mission.

In Paparazzi, data employed by the navigation system is acquired through a *State Interface* that is connected to estimation filters and sensors. This state interface is a blackboard interface that offers different representations for the collected data and that automatically performs transformation depending on the needs of the autopilot. The representations and conversions functions are defined in a mathematical library implemented in C. This library is thus often used and must not contain bugs that can lead for instance to the crash of the program or produce invalid data.

In this thesis, we focused on an important part of Paparazzi C mathematical library that provides a model for the drone attitude through different representations: rotation matrices, Euler angles, or quaternions. The library also defines elementary operations on these representations. Moreover, it defines three versions of each representation and the associated functions: one using `double` values, another one with `float` values and the last one using `int` values to represent fixed point values. This library is an interesting part to verify as it contains non trivial but critical code.

Part I presents the verification of this mathematical library with static analysis techniques applied using the Frama-C [PGH+21, Pol21], a platform offering many plugins for the verification of C code. Chapter 3 presents in detail Frama-C and the verification processes used. In Chapter 4, we applied this process to prove the absence of runtime errors in the library and some interesting functional properties on floating-point conversion functions.

1.3 Flight Plan Generator

Paparazzi provides the possibility to define specific missions. These missions, or *flight plans*, are expressed using an XML-based domain specific language, denoted by `FPL` in this thesis.

Flight plans are complex missions and `FPL` offers common imperative features to define them, such as mutable variables, exceptions or loops. `FPL` also offers some specific navigation primitives like "go to a position" or "do a circle around a location". For instance, the following mission can be described in `FPL`: "when the drone is started, it first needs to initiate its sensors and wait for the GPS connection to be established. Then the drone should take off and circle around a certain GPS position to acquire data. During the flight, if the battery level falls below 20%, the drone must automatically go back to the `Home` position and land."

Paparazzi currently provides a code generator that takes as input a `FPL` flight plan and generates a C file that must be compiled together with the autopilot to be embedded in the drone. The flight plan thus cannot be changed during a flight. However, the drone operator can interact with the flight plan and change its execution order. The generated C code file is mainly composed of a step function, named `auto_nav`, which is called periodically by the autopilot to compute the next steps of the flight plan to be executed.

The generator is written in OCaml and Paparazzi users may not entirely trust the generated code. In particular, we may wonder if a) the function computing the next steps to be executed in the flight plan always terminates and b) the generated C code behaves as described by the flight plan. These questions are crucial as the produced C code is intended to be directly embedded in the drone.

Verifying that the embedded C code is correct, i.e. that it behaves as prescribed by the flight plan, can be seen as a *compiler verification* problem: we must prove that the compiler translating FPL flight plans into C code guarantees the correctness of the embedded code. Even if compiler verification is a known problem [Dav03, MCP67], some advances have recently been done using proof assistants, particularly the Coq proof assistant. Coq allows the user to write programs in the Gallina functional programming language, to formally prove properties on such programs using powerful tactics, and to extract Gallina programs into OCaml programs that are semantically equivalent [Let04, SBF+20, AAM+16]. The main steps of the verification of a compiler with Coq may be summarised as follows: first, express the semantics of both the source and the target languages in Gallina, then write the compiler in Gallina and finally prove a semantics preservation theorem establishing that the produced code has the same behaviour as the source code according to their respective semantics. The CompCert C compiler [LBK+16] and the Velus Lustre compiler [BBP20, Bru20] are recent *tours de force* showing that this approach can be applied to large subsets of real programming languages.

Part II of the thesis presents the verification of the flight plan generator [PGH+23] using Coq and is organised as follows: Chapter 5 presents verification techniques used in the literature to formally verify compilers. This chapter also presents the syntax and the semantics of Clight, the Gallina representation of C code provided by CompCert and used by the new flight plan generator. Chapter 6 shows an overview of the new generator and introduces the modifications we brought into the current generator to improve it. Chapter 7 defines the syntax and the semantics of FPL. Chapter 8 details the new features we added to FPL and the modifications implied in its semantics. Chapter 9 defines the specific architecture of the new 3-pass generator we implemented and the intermediate languages used to simplify the verification process. In Chapter 10, we prove that the new generator preserves the semantics under hypotheses that are explicitly presented. Finally, we discuss in Chapter 11 the lessons learned during this work and the applicability of this work on real-world projects.

Chapter 2

State of the art: formal verification

Contents

2.1 Code verification . . . 66
2.1.1 Properties specification . . . 66
2.1.2 Code analysis . . . 69
2.2 Introduction to formal semantics . . . 71
2.2.1 SIL: Simple Imperative Language . . . 72
2.2.2 Operational semantics . . . 74
2.2.3 Denotational semantics . . . 78
2.2.4 Axiomatic semantics . . . 83
2.3 Proof assistants . . . 87
2.4 Related work . . . 90
2.4.1 Static code analysis . . . 91
2.4.2 Compiler verification . . . 92

The verification of a system is a critical step during its development. The goal of this process is to ensure that the system meets its requirements, i.e. that it can run with the allowed resources, that all specified functionalities are implemented and that there is no bug. Verification is the part of the development process that takes most of the time, especially for embedded systems and critical systems. Indeed, any bugs in these types of systems can be costly or life-endangering, and it may not be possible to update the system easily during its life cycle.

There exist several more or less formal verification techniques: code reviewing, testing, applying formal methods, etc. Among them, formal methods are particularly used for critical systems, e.g. in the avionics industry. Formal methods are mathematically-based languages, techniques and tools to verify software systems [CW96]. There are several families of formal methods that are based on different approaches and each of them is adapted to verify different types of properties. Formal methods are mainly based on the different formal semantics that exist for programming languages. The formal semantics of a programming language L formally specifies the behaviour of any L program. These semantic approaches are used by automatic or semi-automatic formal methods or tools to capture potential execution of a program and to determine if it meets its specifications.

Section 2.1 introduces different verification processes used during the development of embedded and critical systems such as UAV autopilots. Section 2.2 presents different formal

approaches and techniques that exploit them. Then, Section 2.3 presents proof assistants, which are powerful software that help writing and verifying formal proofs and can be used to prove the correctness of complex software like compilers for instance. This section focuses on the Coq proof assistant that has been used for this thesis. Finally, Section 2.4 shows different projects that have been verified using formal methods.

2.1 Code verification

The development of a system or a program can generally be divided into 3 parts:

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

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

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

Verification is a critical step that often takes time. This step is highly dependent on the quality of the specification, especially for critical embedded systems that might require certification. Indeed, in order to ensure the correctness of a system, its behaviour should be correctly specified. Section 2.1.1 presents possible properties that can be defined for the specification of code. Section 2.1.2 presents available techniques that can be used in order to ensure these properties on code.

Remark

Note that the implementation step can take several forms such as code, hardware designs, large system architecture for distributed systems, etc. In this thesis, we only focus on **code specification and verification**.

2.1.1 Properties specification

Generally, the properties specified on code can be divided into two types:

General properties, i.e. properties describing errors or behaviours that any system wants to avoid. These properties generally state the absence of RTE (RunTime Error). Such errors can stop the application (e.g. division by 0, null pointer dereferencing) or produces unwanted behaviour (e.g. if an integer overflow occurs, the programs might give wrong results). Another example of unwanted behaviour is when the system becomes unresponsive. This can happen when a process is blocked in an infinite loop (*livelock*) or if two threads are interlocked (*deadlock*).

Specific properties or **functional properties** describing the expected behaviour of the system. They are generally given by the specification. For example, a property for an UAV might be: "The UAV should take off only if the battery level is sufficient to do at least an emergency landing".

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

We can also distinguish *safety* properties and *liveness* properties. Safety properties define error states of the system that should never be reachable. General properties most often fall in that category. An example might be that the autopilot of a quadcopter drone should never stop the four rotors when the drone is flying. Liveness properties refer to all states of the system that should be reached during its execution. For example, if a drone has been ordered to take off and all the necessary conditions to fly are satisfied, then the drone will eventually fly.

Remark

One type of properties that has not been discussed and which will not be detailed in this document are properties about numerical computation. As real numbers are usually represented with floating-point numbers in computers, algorithms have to be adapted to reduce and control errors generated by finite precision [Hig96]. In addition to the algorithms used, it is important to consider that the compiler may also introduce computation errors due to optimization choices [Mon08]. These computation errors generated by a compiler are also a problem for verification of computer-aided proofs of mathematical theorems (see Section 2.3 for the presentation of proof assistants). Thus, some rigorous verification methods have been developed to provide strong results about floating-point arithmetic [Rum10].

Example of property specification

To illustrate the difference between these kinds of properties, let us define the specification of the `swap` function in C as follows:

```
void swap(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

To specify the properties of a program, we use the following semi-formal notation. M is a memory state at a certain point of the program execution. *Adresses* is the set of all existing addresses in M. *Values* is the set of all possible values that can be stored. We note $\&M$ the set of accessible addresses for the user in M ($\&M \subseteq Adresses$), $M[a]$ the stored value at the address a in M ($a \in \&M$ and $\{M[a] | a \in \&M\} \subseteq Values$). For a function `fun`

(or a portion of code), M^{init}_{fun} and M^{exec}_{fun} are respectively the state of the memory before and after executing the fun function.

Similarly to Hoare triples that will be detailed in Section 2.2, the specification of a function is composed of a precondition and postcondition specifying the properties verified by the state before and after the execution of the function.

Suppose we have two addresses a, $b \in \mathit{Addresses}$ and two values A, $B \in \mathit{Values}$. The specification of the statement $\mathsf{swap}(a, b)$ would be:

Precondition : $\varphi^{\mathsf{swap}}_{\text{functional}} \wedge \varphi^{\mathsf{swap}}_{\text{safety}} \wedge \varphi^{\mathsf{swap}}_{\text{security}}$

Postcondition : $\psi^{\mathsf{swap}}_{\text{functional}} \wedge \psi^{\mathsf{swap}}_{\text{safety}} \wedge \psi^{\mathsf{swap}}_{\text{security}}$

Both properties are thus divided into three sub-types of properties: *functional* properties stating the expected behaviour of the function, *safety* properties stating that no-runtime errors will occur and *security* properties that is a subset of general properties stating that users cannot access memory spaces to which they are not granted access.

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

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

$$\varphi^{\mathsf{swap}}_{\text{functional}} := (M^{init}_{\mathsf{swap}}(a) = A \wedge M^{init}_{\mathsf{swap}}(b) = B)$$

$$\psi^{\mathsf{swap}}_{\text{functional}} := (M^{exec}_{\mathsf{swap}}(a) = B \wedge M^{exec}_{\mathsf{swap}}(b) = A)$$

Remark

The identifiers a, b, A and B are constants that are defined in the formulae of the precondition and postcondition. They will not change during the execution of swap.

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

$$\varphi^{\mathsf{swap}}_{\text{safety}} := (a \in \&M^{init}_{\mathsf{swap}} \wedge b \in \&M^{init}_{\mathsf{swap}})$$

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

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

Security properties: we want to ensure that the `swap` function only modifies the elements pointed by the given parameters. The other elements of the memory should not be modified, and no memory allocation or free should be made. Such a property may be defined as follows:

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

$$\psi^{\mathsf{swap}}_{\text{security}} := \left\{ \begin{array}{c} \forall c \in Adresses : \Big(c \neq a \wedge c \neq b \wedge c \in \&M^{init}_{\mathsf{swap}}\Big) \\ \rightarrow \Big(c \in \&M^{exec}_{\mathsf{swap}} \wedge M^{init}_{\mathsf{swap}}(c) = M^{exec}_{\mathsf{swap}}(c)\Big) \end{array} \right\}$$

The postcondition verifies that all valid addresses that are not a or b (the only values pointed that have been modified) are still valid, after the execution of the function (no memory freed) and that the stored values at these memory addresses have not changed. This property is satisfied if we consider that the variable `tmp` will be stored in a CPU register and not in the memory. In this case, only the addresses a and b in the memory will be modified during the execution.

2.1.2 Code analysis

During the development, the verification is an important step to ensure that the implementation respects the specification. Usually, engineers and developers verify a program by either manually reviewing the written code or by using tests, i.e. the programs or the systems are executed with different set of inputs in order to verify that they behave as defined by the specification for these inputs without any errors. The inputs used in testing are often a small subset of all the possible inputs and it is not possible to test them exhaustively, except for very simple systems. This is generally sufficient for the majority of mainstream systems or software to verify the absence of functional bugs.

However, this method does not ensure that there are no bugs nor errors. Certain domains, especially for embedded and critical systems, need the guarantee of total absence of bugs. For example, we do not want an autopilot for drones to crash during a flight due to a division by 0 or a null pointer dereferencing. Generally, such errors cause unexpected behaviours of the program or, in the most serious cases, the system might stop.

Another type of errors that embedded and critical systems want to avoid is unexpected behaviour with respect to the specification due to either a developer implementation error or a misunderstanding of the specification. These errors are related to implementations that do not respect functional properties.

Code analysis is a family of methods that can verify properties on a program. These methods can analyse the code *statically* i.e. without executing it, or *dynamically* i.e. during execution of the code. In this report, we will focus on *static analysis* methods and particularly on *Deductive Methods* in Section 2.2.4 and *Abstract Interpretation* in Section 2.2.3.2.

The misunderstanding of the specification is a common problem on projects due to the use of natural language as a specification language, which introduces ambiguity. Moreover, with the increasing complexity of systems, it is often difficult to reason at the code level. To

avoid these problems, new development methods appear, in particular with the definition of *models*. Models represent the system and its architecture and are defined with formal or semi-formal languages that reduce ambiguity. This method, is called *Model Based Systems Engineering* (MBSE).

MBSE is a development approach that uses models to define the specification of a system. These models can describe different levels of complexity of the system: the different software and hardware components and how they communicate, but also different threads that can evolve in parallel, etc. Models can be represented using semi-formal languages (such as UML or SysML) or formal languages (for example finite state machines or Petri nets). These representations offer an accurate overview of the project and can sometimes reduce ambiguity.

In MBSE, in addition to the description of the models, different properties that the systems should satisfy can also be defined. These properties are similar to the type of properties defined in the previous section (*general* and *specific* properties, *safety* and *liveness* properties). In order to reduce ambiguity, natural language is avoided and formal languages are rather used. *Temporal logics* [SBB+99] are formal languages used to specify models with properties that describe the dynamic behaviour of the system. There are different tools that can formally verify that a model satisfies some given properties. For instance, Model Checking allows users to verify temporal properties on systems modeled with automata or Petri Nets (see Section 2.2.2.4).

Remark

The term "model" in Model Checking differs from "model" in MBSE, i.e. an abstract representation of systems. Models in Model Checking generally represent the set of all possible executions of the system. Verification of properties on a model corresponds to the verification that all possible executions of the system satisfy the property.

Once the model is defined and verified, the next step is to implement it. The implemented program must follow the specifications of the model in order to ensure that the properties are always respected. There are different methods to guarantee that the program matches the model. For instance, deductive methods on the generated program can be used when the specification uses contracts and Floyd-Hoare logic to define the properties (see Section 2.2.4). Another technique is to generate a program directly from a model.

Code generation is a process that translates a model into source code. A key point of generation tools is that they must be verified in order to guarantee that the code produced respects the model specified and the properties verified. For embedded systems, this process often generates C or Ada code, two programming languages mainly used to implement such systems. When possible, a MBSE tool has generally a code generation plugin. This is possible only if the description language used is sufficiently accurate (formally defined) and relatively close to the target programming language. For instance, the Ocarina tool can generate C or Ada code from a model described with AADL (Architecture Analysis and Design Language), a modelisation language mainly used in the avionics domain. Also, the SCADE suite, used by Airbus, allows them to generate C code and as a certified tool can be easily integrated in a development process following the DO-178 aeronautical standard.

2.2 Introduction to formal semantics

The semantics of a programming language describes the behaviour of a program written in this language and can be seen as a reference manual [NN07]. Semantics can be defined formally in order to reduce the ambiguity of natural language and also to reason on programs or the programming language itself. A complementary objective of this section is to present how formal semantics can be used to verify properties, through several "families" of formal methods, over a program but also the theory behind them, how they work and which kind of property they can establish.

We consider a program P written in the programming language L. We note $P \Downarrow B$ the semantics formula stating that the execution of P produces the observable behaviours B. B includes the operations occurring during the execution of the program but also information about termination, divergence (in case of an infinite loop) or even errors when executing an undefined computation. This notation is a generic definition showing the principle that the execution of a program will produce a result or terminate in a specific state.

Formal semantics used for the verification of programs is generally based on one of the three following approaches:

Operational semantics which describes the execution of a program by successive steps of a state machine. For example, an operational semantics for the execution of a sequence of two statements "$s_1; s_2$" could be:

$$\langle s_1; s_2, \rho \rangle \rightarrow \langle s_2, \rho' \rangle \rightarrow \rho''$$

where ρ, ρ' and ρ'' are states of the system (e.g. a memory environment). The rule presented above states that the execution of the sequence starts by executing in one small step the statement s_1 in the state ρ and results in the state ρ' with s_2 remaining to execute. Then the execution of the statement s_2 terminates in the state ρ''. The operational semantics presented through this example is also often called *structural operational semantics* (or small-step semantics). *Natural semantics* (or big-step semantics) is an alternative operational semantics that specifies how the final result of the execution is obtained and hides some internal execution details. In this case, the semantics for a sequence could be:

$$\langle s_1; s_2, \rho \rangle \Downarrow \rho''$$

Denotational semantics describes the results of the program execution as a mathematical object such as a function. For example, a denotational semantics can be described by a function $[\![s]\!](\rho)$ that produces a new state after the execution of the statement s from the initial state ρ. The execution of a sequence can thus be defined as follows:

$$[\![s_1; s_2]\!](\rho) = [\![s_2]\!]([\![s_1]\!](\rho))$$

Axiomatic semantics uses *Hoare triple* such as $\{\varphi\} s \{\psi\}$ stating that the execution of the statement s from a state that statisfies the property φ will terminate in a state that satisfy the property ψ. Axiomatic semantics describes the execution of the

program as logical assertions. The execution of a sequence can thus be described by the following inference rule:

$$\frac{\{\varphi\}\ s_1\ \{\gamma\} \qquad \{\gamma\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1; s_2\ \{\psi\}}\ (\text{Seq})$$

Throughout Section 2.2, a very simple imperative language noted SIL and presented in Section 2.2.1 is used to illustrate the different semantic approaches. This introduction to formal semantics is inspired by the book "Programming Language Foundations" [Stu13] and defines the semantics of SIL using the three previously introduced approaches respectively in Section 2.2.2, Section 2.2.3 and Section 2.2.4. In addition to the definition of SIL semantics, formal methods associated to the semantic approaches are briefly introduced. This presentation is extended in a state-of-the-art written at the beginning of this thesis [PTG+21].

2.2.1 SIL: Simple Imperative Language

This section introduces SIL, a very Simple Imperative Language. Programs in SIL can define variables and can only manipulate integer values. Variables are elements of the set $\mathbb{V}$ and a state of the memory is defined by a function ρ that associates variables to their values ($\rho : \mathbb{V} \to \mathbb{Z}$). The three basic arithmetic operations on integers are supported : $+, -$ and $\times$. Also, for Boolean condition, all arithmetic comparison operators are supported, i.e. $<, \leq, >, \geq, =$, and $\neq$. Note that $=$ is the equality comparison operator and $:=$ is reserved for assignment as presented below.

2.2.1.1 Syntax

An imperative program in SIL is a statement *stmt* of the following grammar:

stmt ::= `skip` | *stmt*`;` *stmt* | *v* `:=` *expr*
| `if` *cond* `then` *stmt* `else` *stmt* `fi`
| `while` *cond* `do` *stmt* `od`
cond ::= *expr* `<` *expr* | *expr* `<=` *expr*
| *expr* `>` *expr* | *expr* `>=` *expr*
| *expr* `!=` *expr* | *expr* `=` *expr*
| *expr* `&&` *expr* | *expr* `||` *expr*
| `True` | `False`
expr ::= *v* | *z* | *expr* `+` *expr* | *expr* `-` *expr* | *expr* `*` *expr*

with $v \in \mathbb{V}$ and as we consider integer variables, $z \in \mathbb{Z}$. Throughout this section, we note s_1 and s_2 two statements *stmt*, the symbol c denote a condition *cond* and e an expression *expr*.

Example 2.1 - Euclidean division.
The classical program for Euclidean division can be expressed in SIL by the following statement:

```
q := 0;
r := x;
while y <= r do
      q := q + 1;
      r := r - y
od
```

We suppose that the variables x and y are previously initialised in the environment. The program thus computes the Euclidean division of x by y and produces the quotient q and the remainder r as results.

To keep things as simple as possible for this introduction, SIL contains only the above constructs. It could be extended with functions calls, pointers, elaborate data structures, etc. However, this tiny language is already *Turing complete* (i.e. any Turing machine can be simulated as a program of this language) as we use $\mathbb{Z}$ as the domain of values for the integer variables. This put aside, SIL represents a commonly used basis of programming languages such as C or Rust. Additional features would not be useful for this presentation.

2.2.1.2 Informal Semantics

First, we informally define the semantics of SIL:

- skip is the statement that does not modify the state when executed.
- v := *expr* is the assignment operator. Classically, v:= 2 + 3 means "assign value of variable v to the result of the evaluation of $2+3$". Thus, the left operand of := must be a variable, and the right value of := must be a valid arithmetic expression.
- *stmt*; *stmt* is the sequence operator. s_1 ; s_2 means "execute s_1 then execute s_2".
- if *cond* then *stmt* else *stmt* fi is the selection operator. The statement "if c then s_2 else s_2 fi" means "if the Boolean condition c is true, execute s_1 else execute s_2".
- while *cond* do *stmt* od is the iteration operator. while c do s od means "execute repeatedly s until c is false".
- The *cond* and *expr* literals correspond respectively to the evaluation of the Boolean and arithmetic expressions.

In the following sections, we define an operational, denotational and axiomatic semantics for SIL. These different semantics only differ on how to evaluate statements. The evaluation for expressions and conditions is common to the different semantics and it is presented in the next section.

In the following, we note $\rho : \mathbb{V} \to \mathbb{Z}$ an environment and Σ the set of all possible environments. We also use the notation $\rho[v := z]$ to represent the same environment as ρ but where the variable v is now associated with the value z.

2.2.1.3 Formal semantics of expressions and conditions

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

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

$$
\begin{aligned}
&[\![v]\!](\rho) := \rho(v) \\
&[\![z]\!](\rho) := z \\
&[\![e_1 \texttt{ + } e_2]\!](\rho) := [\![e_1]\!](\rho) \ + \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ - } e_2]\!](\rho) := [\![e_1]\!](\rho) \ - \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ * } e_2]\!](\rho) := [\![e_1]\!](\rho) \ \times \ [\![e_2]\!](\rho)
\end{aligned}
$$

Remark

Note that we use different fonts and symbols in order to differentiate the character of the syntax (e.g. `*` character) and the mathematical operator (e.g. the $\times$ operator). We use the same principle for the definition of the condition semantics where we differentiate syntax of the Boolean values (e.g. `True`/`False` vs `true`/`false`) and operators (e.g. `<=` vs $\leq$).

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

The semantics $[\![c]\!](\rho) \in \mathbb{B}$ produces the Boolean value corresponding to the evaluation of the `SIL` condition c in the environment ρ and is formally defined as:

$$
\begin{aligned}
&[\![\texttt{True}]\!](\rho) &&:= \texttt{true} \\
&[\![e_1 \texttt{ \&\& } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ \wedge \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ < } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ < \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ > } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ > \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ = } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ = \ [\![e_2]\!](\rho)
\end{aligned}
\qquad
\begin{aligned}
&[\![\texttt{False}]\!](\rho) &&:= \texttt{false} \\
&[\![e_1 \texttt{ || } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ \vee \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ <= } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ \leq \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ >= } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ \geq \ [\![e_2]\!](\rho) \\
&[\![e_1 \texttt{ != } e_2]\!](\rho) &&:= [\![e_1]\!](\rho) \ \neq \ [\![e_2]\!](\rho)
\end{aligned}
$$

2.2.2 Operational semantics

As presented in the introduction of Section 2.2, an operational semantics for a language is a direct description of a program execution. The evaluation of a statement is naturally defined in a step-by-step fashion. In the following, we start by defining a *big-step* semantics that describes the execution of a statement from an initial state until reaching a final state. Then, we define a *small-step* semantics that specifies the evaluation of a statement of as a series of elementary steps.

2.2.2.1 SIL big-step semantics

We note $\langle s, \rho \rangle \Downarrow \rho'$ the property describing the big-step semantics of SIL. This property states that the execution of the statement s from an initial environment ρ terminates in the state ρ' and it is formally defined by the inference rules presented in Figure 2.1.

$$\frac{}{\langle \texttt{skip}, \rho \rangle \Downarrow \rho} \text{ (Skip)} \qquad \frac{}{\langle v := e, \rho \rangle \Downarrow \rho[v := [\![e]\!](\rho)]} \text{ (Assign)} \qquad \frac{\langle s_1, \rho \rangle \Downarrow \rho' \qquad \langle s_2; \rho' \rangle \Downarrow \rho''}{\langle s_1 ; s_2, \rho \rangle \Downarrow \rho''} \text{ (Seq)}$$

$$\frac{[\![c]\!](\rho) = \texttt{true} \qquad \langle s_1, \rho \rangle \Downarrow \rho'}{\langle \texttt{if } c \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}, \rho \rangle \Downarrow \rho'} \text{ (If}_1\text{)} \qquad \frac{[\![c]\!](\rho) = \texttt{false} \quad \langle s_2, \rho \rangle \Downarrow \rho'}{\langle \texttt{if } c \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}, \rho \rangle \Downarrow \rho'} \text{ (If}_2\text{)}$$

$$\frac{[\![c]\!](\rho) = \texttt{false}}{\langle \texttt{while } c \texttt{ do } s \texttt{ od}, \rho \rangle \Downarrow \rho} \text{ (While}_1\text{)} \qquad \frac{[\![c]\!](\rho) = \texttt{true} \quad \langle s, \rho \rangle \Downarrow \rho' \quad \langle \texttt{while } c \texttt{ do } s \texttt{ od}, \rho' \rangle \Downarrow \rho''}{\langle \texttt{while } c \texttt{ do } s \texttt{ od}, \rho \rangle \Downarrow \rho''} \text{ (While}_2\text{)}$$

Figure 2.1: Big-step semantics of SIL

The execution of the skip statement does not modify the environment (Rule (Skip)) unlike the execution of the assignment statement that computes the value of the expression and then updates the environment (Rule (Assign)). Rule (Seq) describes the execution of a sequence that consists in evaluating the first statement until it reaches a final state ρ' and then uses it as initial state for the evaluation of the second statement. The execution of a conditional statement consists in evaluating the condition and then executing the corresponding statement (see Rule (If$_1$) and Rule (If$_2$)). Finally, the execution of a while loop starts by evaluating the condition in order to decide if the loop should terminate (Rule (While$_1$)) or if we should execute the body before re-evaluating the condition (Rule (While$_2$)).

2.2.2.2 SIL small-step semantics

The small-step semantics, also called structural operational semantics simply defines the partial evaluation of a statement, i.e. how to evaluate a small part of a statement. The complete evaluation of a statement until a final state is reached consists of composing several small execution steps. This semantics uses two notations:

- $\langle s, \rho \rangle \rightarrow \langle s', \rho' \rangle$ states that the evaluation of one small step of the statement s in the environment ρ terminates in the intermediate statement s' and environment ρ'.
- $\langle s, \rho \rangle \rightarrow \rho'$ is special case from the previous notation, stating that the evaluation of one small step for the statement s in the environment ρ reaches a final state ρ'.

The small-step semantics is presented on Figure 2.2. Compared to the big-step semantics, the evaluation of the skip and assignment statements are similar. Indeed, they produce

$$\frac{}{\langle \texttt{skip}, \rho\rangle \to \rho}\ (\textsc{Skip}) \qquad \frac{}{\langle v := e, \rho\rangle \to \rho[v := [\![e]\!](\rho)]}\ (\textsc{Assign})$$

$$\frac{\langle s_1, \rho\rangle \to \rho'}{\langle s_1 ; s_2, \rho\rangle \to \langle s_2, \rho'\rangle}\ (\textsc{Seq}_1) \qquad \frac{\langle s_1, \rho\rangle \to \langle s_1', \rho'\rangle}{\langle s_1 ; s_2, \rho\rangle \to \langle s_1' ; s_2, \rho'\rangle}\ (\textsc{Seq}_2)$$

$$\frac{[\![c]\!](\rho) = \texttt{true}}{\langle \texttt{if } c \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}, \rho\rangle \to \langle s_1, \rho\rangle}\ (\textsc{If}_1) \qquad \frac{[\![c]\!](\rho) = \texttt{false}}{\langle \texttt{if } c \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}, \rho\rangle \to \langle s_2, \rho\rangle}\ (\textsc{If}_2)$$

$$\frac{[\![c]\!](\rho) = \texttt{false}}{\langle \texttt{while } c \texttt{ do } s \texttt{ od}, \rho\rangle \to \rho}\ (\textsc{While}_1) \qquad \frac{[\![c]\!](\rho) = \texttt{true}}{\langle \texttt{while } c \texttt{ do } s \texttt{ od}, \rho\rangle \to \langle s ; \texttt{while } c \texttt{ do } s \texttt{ od}, \rho\rangle}\ (\textsc{While}_2)$$

Figure 2.2: Small-step semantics of SIL

a final state and update the environment if necessary (Rule (Skip) and Rule (Assign)). The evaluation of the sequence is different from the big-step semantics as it is split into two cases: 1) if one small step of the first statement reaches a final state, then the second statement will be executed at the next step (Rule (Seq_1)). 2) if one small step reaches an intermediate state, then the statement executed at the next step would be the sequence of the intermediate statement produced and the second statement (Rule (Seq_2)). One small step for the `if-then-else` statement consists of updating the next statement to execute with the corresponding statement, depending on the result obtained from the evaluation of the condition (Rule (If_1) and Rule (If_2)). The evaluation of the `while` statement starts by evaluating the condition. In the case where the loop must be exited, then the small step reaches a terminal state (Rule (While_1)). In the other case, the next statement to execute is then the sequence of the body followed by the same `while` loop (Rule (While_2)). The loop condition will thus be re-evaluated after the body will finish being evaluated.

2.2.2.3 Relation between big-step and small-step semantics

The two operational semantics presented previously are two different specifications of the SIL program behaviour. We might be tempted to prove that they describe the same behaviour. However, the small-step semantics is more expressive as it also describes the case of non-terminating programs, also called *diverging programs*. Example 2.2 is an example of such a program: it is syntactically correct and does not terminate.

Example 2.2 - Non-terminating program.
An example of a non-terminating SIL program with an infinite loop:

```
x := 0;
while True do
        x := x + 1
od
```

The inference rules of the big-step semantics cannot be used to derive an execution of the diverging programs as it is impossible to define a final state. In the other case, the small-step semantics can be used to describe the execution of this kind of program. The successive execution of small steps will show the program behaviour even if it will never reach a final state.

If we really wanted to prove that the two semantics are equivalent, we could either restrict ourselves to non diverging programs or re-formulate the big-step semantics. Indeed, we could have another notation for big-step semantics such as $\langle s, \rho \rangle \Downarrow_n \langle s', \rho' \rangle$. This new definition would state that the execution from an initial state after n steps does not reach a final state and thus has been stopped in an intermediate state. The proof would also require defining new rules for multi-step reduction for the small-step semantics, that count the number of small steps executed from the initial state.

2.2.2.4 Verification applicability of operational semantics

In practice, operational semantics is used in different contexts such as verification of programs manipulating languages such as compilers and is used in the *model checking* technique.

Compiler verification

As a formal semantics provides a mathematical definition of the execution of a program, it can be manipulated in proof of programs when we want to prove a property on the program behaviour, especially when a proof assistant is used (see Section 2.3). For example, we might want to verify that the code in Example 2.1 performs correctly the Euclidean division and thus when it reaches a final state, we have the following equality: $x = q \times y + r$, and the inequality: $0 <= r < y$.

A major project that mechanises an operational semantics with a proof assistant is CompCert [LBK+16], a C compiler verified using the Coq proof assistant. It has been proven that the code produced by the compiler preserves the semantics of the input. The semantic preservation property is detailed in Section 5, but the important point is that operational semantics of the C language and of the assembly code generated were used to establish the proof.

Model Checking

Model checking [CES86] is a domain of formal verification that is generally not directly used on code. Model checking is a semantic method, i.e. it works on a representation of programs or systems using operational semantics. Model checking analyses a representation of a real system to verify properties, which generally implies computing all the possible executions of the system. This technique is used in different fields and at different stages of development to verify liveness and safety properties. For example, it can be used on the architecture of distributed systems to verify that there are no livelocks or on the specification of a hardware design to ensure that error states are not accessible. It is also used for embedded programs

to ensure that an unsafe state is never reached. The systems or programs analysed using model checking are usually represented using Petri Nets or automata.

Model checking needs a property specification language. This language must allow the user to describe properties about a state of the system, but also to describe the dynamic behaviour of the system. Most specification languages used for model checking are based on *temporal logic* [SBB+99]. This family of logic allows users to simply define, in a single property, states that can be true currently but also in the future. For example, if we suppose to have two logical properties φ and ψ about a system that have been defined previously, we can formally specify the following property using temporal logic: "At any time, if the property φ is true, then the property ψ will necessarily be true in the future".

There are different algorithms and techniques for model checking. The original algorithms simply enumerate explicitly all states [CES86]. Unfortunately, as the systems to verify became larger, these algorithms are limited by a known problem, called *state explosion problem*. In order to overcome this problem, there exists another category of model checking techniques called *symbolic model checking*. The main idea is to represent groups of states and transitions symbolically in order to simplify the verification of temporal properties and therefore reduce the computation time. There are different symbolic representations: binary decision diagrams or logical formulas that can be verified using SAT/SMT solvers [BSST09]. These algorithms will not be detailed in this thesis but they are introduced in the state-of-the-art written at the beginning of this thesis [PTG+21].

Several tools implement model checking algorithms. For example, there are the model checkers Java Pathfinder [HP00] for Java bytecode or CBMC [KST23] for C programs. CBMC can verify many kinds of properties such as memory safety, exceptions or some undefined behaviour. For non-imperative languages, there is Pkind [KT11] that is a model checker using k-induction to verify invariants on the synchronous language [BCE+03] Lustre, defined by N. Halbwachs et al. [HCRP91].

2.2.3 Denotational semantics

Denotational semantics is specified as a mathematical structure, such as a function, that manipulates the result of the program execution as a mathematical object. The denotational semantics for `SIL` programs is expressed here by a *total* mathematical function $[\![s]\!](\rho)$ which produces the final state reached after the execution of the statement s from the environment ρ. This approach is different from operational semantics for two reasons. First, big-step semantics describes valid execution from an initial state to a final state only for non diverging programs whereas the function $[\![s]\!](\rho)$ is total and will always produce a final state for every syntactically correct statement. Second, the small-step operational semantics only describes successive small execution steps of the program. The execution of diverging programs will thus never reach a final state. In denotational semantics, we expect the function $[\![s]\!](\rho)$ to return either the final state if the program converges or a special result $\perp$ for non-terminating programs. The $\perp$ symbol is a mathematical object different from any other environment stating that the program diverges. The result of the denotational semantics function is thus part of the set $\Sigma \cup \{\perp\}$ noted $\Sigma_\perp$, i.e. the union of all the possible environments and the special symbol describing diverging programs.

Let us formally define this mathematical function for SIL program and then we will see how denotational semantics can be used for the verification of programs using *Abstract interpretation*.

2.2.3.1 Semantics of statements

The definition of the semantics for skip and assignment statements is the following:

$$[\![\texttt{skip}]\!](\rho) ::= \rho$$
$$[\![v := e]\!](\rho) ::= \rho[v := [\![e]\!](\rho)]$$

As the skip statement does not modify the environment, its denotational interpretation is thus the identity function. The assignment statement is just the modification of the environment only for the variable assigned with the value computed from the expression.

$$[\![s_1 ; s_2]\!](\rho) ::= \begin{cases} \bot & \text{if } [\![s_1]\!](\rho) = \bot \\ [\![s_2]\!]([\![s_1]\!](\rho)) & \text{otherwise.} \end{cases}$$

The definition of the sequence starts by evaluating the first statement s_1 and then it is decomposed into two cases: 1) the first statement diverges, thus the sequence statement diverges 2) the first statement terminates in a final state, it is thus used for the evaluation of the second statement.

$$[\![\texttt{if } c \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}]\!](\rho) ::= \begin{cases} [\![s_1]\!](\rho) & \text{if } [\![c]\!](\rho) = \texttt{true} \\ [\![s_2]\!](\rho) & \text{otherwise.} \end{cases}$$

The evaluation of the conditional statement starts by computing the value of the condition. Depending on the result, the environment returned is the one produced from the corresponding statement, i.e. either s_1 if the condition has been evaluated to true or s_2 in the other case.

Finally, the definition of the while statement is more tricky to specify than in the operational semantics. Indeed, we cannot define the function recursively as this will not work for diverging programs that must return the $\bot$ value. The evaluation of the while statement is thus:

$$[\![\texttt{while } c \texttt{ do } s \texttt{ od}]\!](\rho) ::= \mu(F)$$
$$\text{where } F ::= w \mapsto \left(\rho \mapsto \begin{cases} \rho & \text{if } [\![c]\!](\rho) = \texttt{false} \\ \bot & \text{else if } [\![s]\!](\rho) = \bot \\ w([\![s]\!](\rho)) & \text{otherwise.} \end{cases} \right)$$

with μF the least fixpoint of function F. The function F takes as parameter a function w with the signature $(\Sigma \to \Sigma_\bot)$. This function thus takes an environment and produces a new environment or the $\bot$ value (see [Stu13] for more details, particularly on the existence of $\mu(F)$).

2.2.3.2 Abstract interpretation

Denotational semantics can be used for the static analysis of a program, i.e. the verification of a property that should be satisfied during the execution of the program without executing the program itself. Static analysis is often performed at compilation time in order to ensure the properties before being embedded in critical systems.

One of the goals of program analysis is to determine if specific states of the memory exist and are accessible. The basic idea is to compute for each variable and for each program point the set of possible values. A program point can be seen as a state of a finite-state machine. This state contains all possible values of the variables at this point. The transitions correspond to the execution of an instruction of the program. For example, if we have a program that contains the code of Example 2.1, we might want to ensure that for any execution, for every accessible state ρ before executing the Euclidean division program, the property P stating $\rho(\mathsf{y}) > 0$ holds. We note $\rho \models P$ if ρ satisfies this property. In the other case, we use the notation $\rho \not\models P$.

The principle consists in computing the set of accessible states in a *concrete domain*, i.e. in a *complete lattice*[1] $(\mathcal{D}, \sqsubseteq)$ (which is in this example $(\Sigma, \subseteq)$), using denotational semantics. Instead of taking an environment and producing a new environment after executing a statement, the semantics is extended to work on sets. Thus, $[\![s]\!](\rho)$ produces the set of all possible environments after the execution of the statement s from every state in ρ. Having this set of reachable states, we can then verify if it contains invalid states.

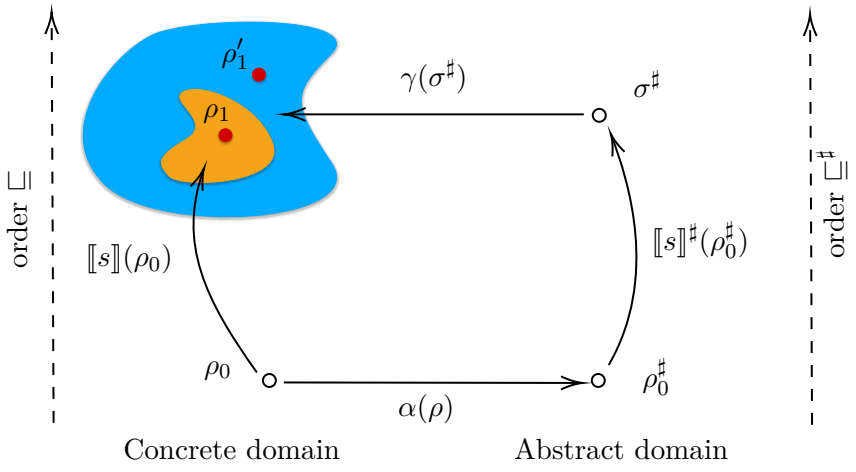

Figure 2.3: Principle of abstract interpretation

The left part of Figure 2.3 shows how this works. Suppose that s is the program that will be executed before executing our Euclidean division program and ρ_0 is every possible environment at the beginning of the program execution. We can compute the set

[1] A complete lattice is a partially ordered set where all the subsets have both a least upper bound and a greatest lower bound.

of reachable states $[\![s]\!](\rho_0)$ (the orange shape in the figure). In order to ensure that we can safely execute the Euclidean division, we want to verify that every reachable state satisfies the property P, i.e. $\forall \rho \in [\![s]\!](\rho),\ \rho \models P$.

This verification technique seems to be a perfect solution. Unfortunately, computing the exact set of reachable states is usually too complex, unless it is simply not computable, which is the general case. *Abstract interpretation* is although a convenient and versatile formal framework to define computable static analyses of programs [Cou99, CC77, CC92].

Abstract domains

The main principle of abstract interpretation is to use *abstract domains* to abstract the set of possible values, that are based on a *complete lattice* $\left(\mathcal{D}^\sharp, \sqsubseteq^\sharp\right)$[2]. We use the exponent $^\sharp$ to distinguish abstract objects from their concrete counterpart. Abstract domains are equipped with two functions to transform concrete elements to abstract elements, and vice-versa:

- $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$, the *abstraction function* that maps the concrete elements of $\mathcal{D}$ to abstract elements of $\mathcal{D}^\sharp$.
- $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$, the *concretisation function* that map the elements of the abstract domain to their equivalent concrete element.

Both functions should be monotone, e.g. the concretisation function should respect the property:

$$\forall \rho^\sharp, \sigma^\sharp \in \mathcal{D}^\sharp,\ \rho^\sharp \sqsubseteq^\sharp \sigma^\sharp \Rightarrow \gamma(\rho^\sharp) \sqsubseteq_{\mathcal{D}} \gamma(\sigma^\sharp).$$

There exist several abstract domains that can used in practice. Their usage depends on the property we want to verify or the precision needed. These different abstract domains can be divided into two types:

non-relational abstract domains is a type of domain that creates independent properties for all variables. For example, such domains are the *intervals domain* where each variable is associated with one interval, the *signs domain* which abstract the values taken by a variable by only keeping the information about its sign, or the *constants domain* where each variable is associated with a constant.

relational abstract domains is a type of domain where different variables can be associated in the same relation. For example, such domains are the *polyhedra domain* [CH78] that creates for instance relations of the form $2x + y < c$ where x and y are integer variables of a program and c an integer, the *Octogon* [Min01] which is a subset of the polyhedra domain restricted to linear constraints of the form $\pm x \pm y \leq c$, or the *zonotopes domain* [GGP09] that is also a subclass of polyhedra domains that is suitable for the study of numerical invariants and more particularly for real variables.

[2] A set $\mathcal{D}^\sharp$ with a relation order $\sqsubseteq^\sharp$ is a complete lattice if and only if it admits a least upper bound or a greatest lower bound for every subset of element [Mis14].

Abstract operators

Abstract domains are used to compute an overapproximation of the concrete states of the program. The computation of these sets for each state of the program depends on the instructions of the program. In order to make the calculations easier on these sets, *abstract operators* are defined. These operators can be seen as part of denotational semantics, but instead of manipulating sets of concrete states, it manipulates sets of abstract environments. We note $[\![s]\!]^\sharp(\rho)$ the set of reachable abstract environment of $\mathcal{D}^\sharp$ after the execution of the statement s from all the abstract environments in ρ. We call these operators *sound* when they satisfy the following condition:

$$\forall \rho^\sharp \in \mathcal{D}^\sharp, [\![s]\!]\Big(\gamma(\rho^\sharp)\Big) \subseteq \gamma\Big([\![s]\!]^\sharp(\rho^\sharp)\Big).$$

The utilisation of abstract domains and abstract operators by the abstract interpretation framework is represented in Figure 2.3. We start by computing the initial abstract state $\rho_0^\sharp$ using the abstraction function. Then, we can compute the abstract state $\sigma^\sharp$ reached after the execution of the statement s using the abstract operators ($[\![s]\!]^\sharp(\rho_0^\sharp)$). We can afterwards use the concretisation function to obtain an overapproximation of $[\![s]\!](\rho_0)$. Finally, we want to verify that $[\![s]\!](\rho_0)$ does not contain an invalid state i.e. a state ρ where $\rho \not\models P$. If there is an invalid state, it can either be a real invalid state reached by the program (see ρ_1 in the figure) or a state that is not reachable in reality by the program s as the abstract interpretation has computed an overapproximation (see ρ_1' in the figure). In the second case, we have a *false positive* and the user may think that there is a potential error although it cannot happen. In this case, we must use another abstract domain that may produce a more accurate result. In any case, abstract domains with sound operators are designed to never produce *false negative*. Indeed, this would mean that abstract interpretation would say that no error can occur when this is not the case.

Iteration techniques and applicability

Even with tools implementing abstract interpretation, the computation of sets when there are loops in the program might not converge. For this reason, some techniques like widening [FG10, and references therein] or iteration techniques such as *policy iterations* [AGG10, CGG+05, GGTZ07, GS10] are used. These methods are presented in the document [PTG+21].

Different tools using abstract interpretation are commercialised and used in the industrial world. For instance, Astrée is a static analyser commercialised by AbsInt [CCF+09]. Astrée is used by companies, especially Airbus, for the verification of critical systems. It analyses C programs and verifies the absence of runtime errors (RTE) such as division by zero or buffer overflows. This tool uses the combinations of several abstract domains: relational domains (such as octagons) are applied locally on a few variables of the analysed program, communicating their results back and forth to less expensive non-relational domains (such as intervals domain) or other instances of relational domains [CCF+06].

Another industrial tool using abstract interpretation is Polyspace from Mathworks. This

tool can verify C, C++ and Ada programs, ensuring that they respect some norms of safety and security.

Finally, applications of this technique may be found in compilers, which are used for the development of most programs in all domains and not just for embedded and critical systems, largely broadening its scope. For example, GCC, a C compiler, uses abstract interpretation to determine if it is possible to perform optimisations, such as the propagation of constant values.

2.2.4 Axiomatic semantics

Axiomatic semantics does not define explicitly the behaviour of a program like operational and denotational semantics do, but rather defines the meaning of a statement by specifying its effect on program state predicates. The predicates are logical assertions in which variables specify the program state. For example, we can have a predicate stating that the variable x in a specific program is not equal to 0.

Axiomatic semantics is closely related to Floyd-Hoare logic [Hoa69]. Floyd-Hoare logic is the foundation of *deductive program verification* [Fil11] that is another field of formal methods in which the problem of verifying the correctness of a program is translated into a problem of verifying the validity of a particular logic formula.

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

$$\{\varphi\}s\{\psi\}$$

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

In the following, we consider that the preconditions and postconditions are expressed here in first-order logic with equality, in which standard arithmetic for natural numbers is available. For example, the assertion "if the value of variable x is equal to 3, then the value of variable y is equal to the value of variable z plus 1" will be expressed as $(x = 3) \rightarrow (y = z + 1)$.

Example 2.3 - Euclidean division specification.
We note s the statement describing the program of the Euclidean division, presented in the Example 2.1. A Hoare triple associated with s is thus:

$$\{\mathsf{x} \geq 0 \wedge \mathsf{y} > 0\}\ s\ \{\mathsf{x} = \mathsf{q} \times \mathsf{y} + \mathsf{r} \wedge 0 \leq \mathsf{r} < \mathsf{y}\}$$

This triple is specifying s in the following way:

- We only consider the cases in which the input variables x and y verify $\mathsf{x} \geq 0$ and $\mathsf{y} > 0$.

$$\frac{}{\{\varphi\}\ \texttt{skip}\ \{\varphi\}}\ (\textsc{Skip}) \qquad \frac{}{\{\varphi[v/e]\}\ v\texttt{:=}e\ \{\varphi\}}\ (\textsc{Assign})$$

$$\frac{\{\varphi\}\ s_1\ \{\gamma\} \qquad \{\gamma\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1\texttt{;}s_2\ \{\psi\}}\ (\textsc{Seq}) \qquad \frac{\varphi \to \varphi' \qquad \{\varphi'\}\ s\ \{\psi'\} \qquad \psi' \to \psi}{\{\varphi\}\ s\ \{\psi\}}\ (\textsc{Cons})$$

$$\frac{\varphi \to \varphi' \qquad \{\varphi'\}\ s\ \{\psi\}}{\{\varphi\}\ s\ \{\psi\}}\ (\textsc{Str}) \qquad \frac{\{\varphi\}\ s\ \{\psi'\} \qquad \psi' \to \psi}{\{\varphi\}\ s\ \{\psi\}}\ (\textsc{Weak})$$

$$\frac{\{\varphi \wedge c\}\ s_1\ \{\psi\} \qquad \{\varphi \wedge \neg c\}\ s_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ c\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}}\ (\textsc{If})$$

$$\frac{\{\varphi \wedge c \wedge V = \omega \wedge V \in D\}\ s\ \{\varphi \wedge V \prec \omega \wedge V \in D\} \qquad (\mathrm{D}, \prec)\ \text{is wf}}{\{\varphi \wedge V \in D\}\ \texttt{while}\ c\ \texttt{do}\ s\ \texttt{od}\ \{\varphi \wedge \neg c \wedge V \in D\}}\ (\textsc{While})$$

Figure 2.4: The Floyd-Hoare inference rules

- If the preconditions are true, then s terminates and the output variables `q` and `r` are such that `q` is the quotient and `r` the remainder of the euclidean division of `x` by `y`.

2.2.4.1 Floyd-Hoare inference rules for `SIL`

Let us now present Floyd-Hoare logic [Hoa69] using the `SIL` language. The complete Floyd-Hoare formal system is presented on Figure 2.4. We describe the axioms and inference rules in the following. These rules describe how we can derive a valid Hoare triple from valid Hoare triples of simpler programs.

The rule for the skip statement (Skip) is one of the two axioms of the system with the assignment rule. Basically, as the `skip` statement does not modify the state, the postcondition φ holds if and only if the precondition φ is satisfied.

The assignment Rule (Assign) says that if postcondition φ holds after v `:=` e, then the precondition must be $\varphi[v/e]$, i.e. φ in which all free occurrences of v have been replaced by the expression e. It may seem counter-intuitive to have such an axiom in which the precondition is defined from the postcondition and not the contrary, but the reader can easily convince herself that this is a valid formulation. There is a version of this axiom defining the postcondition from the precondition, but it is more complicated:

$$\frac{}{\{\varphi\}\ v\texttt{:=}E\ \{\exists v'\ (v = E[v/v']) \wedge \varphi[v/v']\}}\ (\textsc{Assign}^*)$$

where v' is a new variable.

The sequence Rule (Seq) is rather intuitive: if you can prove the two Hoare triples $\{\varphi\}\ s_1\ \{\gamma\}$ and $\{\gamma\}\ s_2\ \{\psi\}$ then the precondition of $s_1\texttt{;}s_2$ is φ and its postcondition is ψ.

It may be the case that you cannot directly find the common precondition/postcondition γ which allows users to apply the rule. To solve the problem, the rule (Cons) and its two variants (Weak) and (Str) allow users to strengthen the preconditions or weaken the postconditions of a program.

The conditional Rule (If) is also intuitive: given a conditional program of the form `if` c `then` s_1 `else` s_2 `fi`, if you can prove the triple $\{\varphi \wedge c\}\ s_1\ \{\psi\}$ (meaning that you consider executions of s_1 in the cases where φ and the condition c hold) and the triple $\{\varphi \wedge \neg c\}\ s_2\ \{\psi\}$ (meaning that you consider executions of s_2 in the cases where φ holds and the condition c does not hold), then the triple $\{\varphi\}$ `if` c `then` s_1 `else` s_2 `fi` $\{\psi\}$ is proven. Notice again that you may have to use (Cons) and its variants to correctly establish the triples for s_1 and s_2.

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

$$\frac{\{\varphi \wedge c\}\ s\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\ c\ \texttt{do}\ s\ \texttt{od}\ \{\varphi \wedge \neg c\}}\ (\texttt{It}^*)$$

First, the premise of the rule is a Hoare triple $\{\varphi \wedge c\}\ s\ \{\varphi\}$ meaning that it should be the case that if the program is in a state verifying φ and c (the loop condition), then executing s brings to a state verifying φ. The formula φ is called a *loop invariant*. It is a formula specifying what is true and what should remain true at each loop iteration. Notice that c does not have to hold after executing s as we have to be able to exit the loop. The conclusion of the rule is the Hoare triple $\{\varphi\}$ `while` c `do` s `od` $\{\varphi \wedge \neg c\}$ meaning that the invariant should be true before entering the loop and is preserved when exiting the loop (in this case, c does not hold anymore). Finding a loop invariant is a difficult problem and there is no systematic method to find the formulas expressing a strong invariant of a loop, but some procedures like abstract interpretation may be used to find them. [FMV14] provides loop invariants for classical algorithms.

The complete Rule (While) for the loop adds some formula to the precondition and postcondition of its premise. In the Hoare triple $\{\varphi \wedge c \wedge V = \omega \wedge V \in D\}\ s\ \{\varphi \wedge V \prec \omega \wedge V \in D\}$, the expression V is called *the loop variant* and ω is a predefined variable that must not appear in φ, c, V nor s which is specific to the current loop (others variables will be initialised in the case of nested loops). The variant V is an expression based on the program variables whose value should decrease during the execution of the loop. The other premise of the rule, "$(D, \prec)$ is wf", means that $\prec$ is a well-founded relation on the set D and thus there is a minimal element in the set of possible values for V[3]. Therefore, as the value of V

[3] In general, we use $(\mathbb{N}, <)$ or $x \prec y := x < y \wedge 0 \leq y$.

decreases at each loop iteration, we can guarantee that we will exit the loop and that the program terminates. The postcondition $V \in D$ ensures that V is correctly typed.

The Floyd-Hoare system is sound and complete with respect to classic program semantics (see [NN07]). The formal system presented here is the simplest one. In particular, extensions for languages manipulating memory through pointers can be found for instance in [ORY01].

2.2.4.2 Weakest preconditions: a calculus for Floyd-Hoare logic

The Floyd-Hoare formal system allows users to build proofs on Hoare triples, and thus allows them to prove programs. But there is no mechanical method or algorithm to build such a proof. Dijkstra has introduced in [Dij75] the *predicate transformer semantics*, a particular semantics for imperative programs close to denotational semantics. This semantics is in fact a reformulation of Floyd-Hoare logic and can be used as a strategy to build deductions in Floyd-Hoare logic. Predicate transformer semantics associates with each programming statement a total function between two predicates representing the precondition and the postcondition of the statement. Depending on the domain and co-domain of the function, predicate transformers are *weakest preconditions* (from postconditions to preconditions) or *strongest postconditions* (from preconditions to postconditions). In the following, we only discuss about weakest preconditions.

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

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

The wp function is relatively easy to compute, when the loop variants and invariants are provided. It only requires to apply simple formulas that will be not presented here (they can be found in [PTG+21]). When the weakest precondition of a program is computed, the last step is to prove the final relation between the precondition of the program and the computed weakest precondition:

$$\{\varphi\}s\{\psi\} \iff \varphi \rightarrow wp(s, \psi)$$

In less formal words, the triple $\{\varphi\}s\{\psi\}$ is correct if and only if the formula φ is sufficient to logically imply the weakest-precondition needed to have ψ holding after executing the program s (that is $wp(s, \psi)$). The proof of this implication is not as immediate as the computation of wp. It can often be achieved by using automatic solvers like Alt-Ergo, Z3 or CVC4, but sometimes the problem is complex and it requires to manually prove it. In this case, proof assistants such as Coq or Isabelle/HOL can be used (see section 2.3).

2.2.4.3 Deductive methods and real-world languages

Weakest preconditions or *strongest postconditions* can often be automatically computed but not as the implication proof (i.e. $\varphi \rightarrow wp(s, \psi)$). There are various tools helping or automating the realisation of this proof, but they are frequently compatible with a unique specification language or target language, i.e. the language used to define the program to verify. For example, Frama-C[4] is a tool for the verification of C programs that uses ACSL as specification language, whereas GNATProve only works for the SPARK programming language.

2.3 Proof assistants

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

"Pencil-and-paper" mathematical proofs can be long and tedious, and thus are error-prone, making the proof questionable. Examples of proofs that have turned out to be incorrect after their publication are famous, for instance, Andrew Wiles first proof of Fermat's last theorem. Some proofs are also so long that it is difficult, if not impossible, for a human to check them, such as the proof of the Feit-Thompson theorem. For these reasons, computers started to be used to facilitate formal proofs. For instance, in 1976, Kenneth Appel and Wolfgang Haken were the first ones to prove a theorem with the help of a computer (the Four Color Theorem [AH76]). There are also several projects that have been developed to facilitate the writing of complete formal proofs and their verification.

The Automath project [DB94] was first created by Nicolaas de Bruijn, in 1967. The goal of this project was to create a language which can express all mathematical formulae, and then develop a system for the verification of theorems. This project was the precursor of current proof assistants. In the early 1970s, there was also Logic for Computable Functions (LCF) citegordon1979edinburgh interactive automated theorem prover that was developed at Stanford and Edinburgh. This theorem prover introduces theorem-proving tactics to help build the proof terms. However, it was not until the years 1990-2000 that proof assistants were developed further. Currently, the most developed are Isabelle/HOL developed at the University of Cambridge and the Technical University of Munich, Coq developed at Inria, and Lean [dMKA+] developed by Leonardo de Moura when he was at Microsoft Research in 2013. There are also smaller projects, such as KeYmaera X which is a proof assistant, using a specific logic for Cyber-Physical systems [Pla18]. Over time, proof assistants have been enriched: they have become more efficient, and allow users to prove more complex theorems, e.g. the Feit-Thompson theorem has been proven using Coq [GAA+13] or the Kepler conjecture using Isabelle/HOL [HAB+17].

Unfortunately, the use of a proof assistant can be disputed. Indeed, they are complex software and we cannot be sure that they do not have any bugs. In order to guarantee a

[4]Framework for Modular Analysis of C programs (https://frama-c.com). A C program analyser using for instance abstract interpretation and deductive method. See Chapter 3.

certain level of trust, some proof assistants, such as Coq, have an architecture that can be divided into two parts:

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

Despite their complexity, proof assistants are also used in "real-world" computer science projects. For instance, CompCert is a C compiler proven with Coq, used at Airbus. Sel4 is an operating system microkernel that have been proven with Isabelle.

Let us focus now on the Coq proof assistant as we used it to verify the two components of Paparazzi presented in this document.

The Coq proof assistant

Coq is a proof assistant developed by Inria based on the Gallina language. As presented earlier, it is a software that allows first users to define either mathematical structures or programs in a purely functional way. It also allows specifying properties on these structures and programs using the Gallina language. Coq offers also a powerful tactic language to build proofs about properties interactively. Finally, Coq proposes an extraction mechanism that converts Gallina programs into OCaml programs that are semantically equivalent [Let04, SBF+20, AAM+16]. The extracted programs thus preserve the properties verified on the corresponding Gallina code. All the concepts of Coq are well presented in the "Software Foundation" [PdAC+23] tutorial.

Coq is based on a natural deduction logical system [PM12, CH85, CP90] that follows the Curry-Howard isomorphism. This isomorphism states that there is a direct correspondence between functional programs expressed in Gallina and mathematical proofs. A proposition A to be proven can be viewed as a type in a program and an element $a \in A$ is a proof of A. The element a is also called a *witness* of A. Therefore, being able to write a program of type A is equivalent to establishing a proof of A. We will not detail the whole equivalence between the programming system and the logical system, but let us see a simple example.

Example 2.4 - Curry-Howard isomorphism for implication.
Let us consider two propositions A and B. The term "$A \to B$" is also a proposition. In order to prove this proposition in a functional way, we must define a function, also called a *proof term*, whose type is $A \to B$. This function thus must take a parameter of type A (or a witness of A) and it must produce an element whose type is B, i.e. a witness of B.

The Coq proof mechanism is thus based on defining functions that build the proof of a proposition, i.e. the type of the function defined should then be the same type as the proposition we want to prove. However, it is not always easy to build directly such a function. To face this problem, Coq provides a powerful tactic mechanism to build the proof interactively, i.e. the term which type is the proposition to be proven. Tactics can be seen as statements in a "pen-and-pencil" proof. During the development of a Coq proof, the user can see the remaining goals to prove and at the start of the proof, there is one goal that is thus the proposition to prove. When a tactic is evaluated, the goals are automatically updated. The tactics offer a "natural" way of writing proof for users but under the hood, the tactics are automatically translated into proof terms which Coq then checks to ensure that the proof is correct. This checking pass computes the type of the proof terms (i.e. the corresponding function) and ensures that the type corresponds to the proposition to prove.

```
Theorem implication:
  ∀ A B, A → B → A.
Proof.
  intros A B Ha Hb.
  apply Ha.
Qed.
```

(a) Proof using tactics

```
Definition implication: ∀ A B, A → B → A :=
  fun A B (Ha: A) (Hb: B) ⇒ Ha.
```

(b) Proof defined as a function

Figure 2.5: Example of two Coq proofs.

Figure 2.5 presents the two possible ways of proving the property ∀ A B, A → B → A in Coq. Code 2.5a uses Coq tactics to prove the property. The `intros` tactics introduces the different variables in the context: A, B that are two propositions and Ha, Hb which are respectively two witnesses (or proofs) of A and B. If you use the Coq interactive interface, after the execution of the tactic, the goal remaining is thus `A`. In order to prove `A`, we simply have to use `Ha` (a proof of `A`) by applying it on the goal using the `apply` tactics. Code 2.5b proves the same property, but directly defines instead the function which type corresponds to the proposition to be proven. This function is thus the *proof term* of the proposition. There is thus an equivalence between tactics and function definition. Indeed, we directly name the parameter of the function in the definition, instead of using the `intros` tactic, and we directly return the parameter `Ha` instead of using the `apply` tactic.

Remark

Note that the Coq proof in Figure 2.5a deliberately matches the explanation of the proof term presented in Figure 2.5. However, Coq has powerful tactics that can produce automatically this proof term. For example, this proof can be verified only using the `tauto` tactics which implements a decision procedure for intuitionistic propositional calculus (see `tauto` documentation).

Coq dependent types

Dependent types are types that depend on a value to be defined. For example, we can have the type $(Vect : nat \rightarrow Type)$ that is defined if a natural value is provided and thus $Vect\ n$ specifies the set of natural vectors of n elements. Dependent types are available in Coq and they have been used to verify the flight plan generator, presented in Part II.

In the following, we only use dependent types that define subsets of elements that satisfy properties. We note *type* a type and *prop* e is satisfied if e respects the property specified by *prop*. The type of *prop* is thus $type \rightarrow Prop$. We now present some notations that will be used later to manipulate dependent types describing subsets.

Notation 2.5 - Definition of dependent type or subset type.
$type_d ::= \{\ e\text{: } type \mid prop\ e\ \}$

We thus denote by $type_d$ the subset of *type* that only contains elements that respect the property *prop*.

Notation 2.6 - Instantiate a value of a dependent type.
$e_d ::= \texttt{exist}\ type_d\ e\ H$

The symbol e_d correspond to an element of $type_d$ that is defined by a dependent pair composed of a value e of *type* and a proof H that e satisfies the property *prop*.

Finally, we add the two notations presented below to access the value and the proof of an element that is from a subset type.

Notation 2.7 - Dependent type access.
If we consider an element e_d defined by `exist` $type_d\ e\ H$, we use the notation:

- (e_d).`value` to access the value e,
- (e_d).`proof` to get the proof H.

2.4 Related work

Formal methods are used in several domains [WLBF09], especially when failures may lead to money loss, mission loss or above all life loss. There are therefore a significant number of examples where formal methods are used in an industrial context. For instance, some significant railway signalling and train control projects have used formal methods: the RER Line A have applied retro-engineering and formal proof, the line 14 of the Paris subway for which safety-critical parts have been developed using B method [Abr10, Abr05]. The formal methods have also been employed in banking systems with for example the Mondex Smart Card [WSC+08], a smartcard-based electronic cash system that has been proven using Z. ANSSI developed the Wookey project [ANS19], a secure and trusted USB mass storage device that uses a secured microkernel implemented in Ada and verified using SPARK. Moreover, formal methods were also applied for web applications at AWS using TLA+ [NRZ+15]. In avionics, the Airbus A380 was developed using SCADE, Astrée and CAVEAT [ATB+95], a tool for the verification of C programs. These tools were used

to verify the absence of RTE in the primary flight-control software, and also to generate automatically 70% of the code, significantly decreasing coding errors. Formal methods are also applied on the Rosetta's Philae lander [Ver, VBF+11] that have been developed using TLA+ to completely rethink the RTOS (Real-time Operating System) used, yielding a code 5 to 10 times smaller.

We have previoulsy discussed projects using a proof assistant such as Sel4 [HE16], a verified OS microkernel or CompCert [Ler09b], a verified C compiler.

Formal methods are thus used in several domains and we can expect them to be used in new domains in a near future, such as drones or automotive. Automotive industry currently does not require certification to sell cars. However, with the increase of software embedded in cars and future autonomy functions for them, the authorities may eventually require certification. The PSA automotive group has explored with the thesis of Vassil Todorov [Tod20] different formal methods in order to determine how to apply them in an industrial context, especially in the automotive industry. This thesis proposes methodology for a non-expert to bring guarantees and robustness of safety-critical parts of the system. Todorov has used model-checking for the verification of a cruise controller function and have verified the absence of runtime errors and functional properties on a low-level function using deductive methods.

Autonomous drones may become common in the future as there are a number of possible applications such as the delivery of medicine or material in remote places, transportation or agriculture. It is thus essential to develop autopilot software that are safe. The work presented in this document tackles this problem of the verification of a drone autopilot using formal methods. We focus on critical components of Paparazzi that have never been verified and propose verification methods that can be applied on other drone autopilot or other domains where we need to have strong confidence in the software developed.

There are already several projects addressing this problem. The project Comp4Drone [Com21, LGB+22, HBL+22b, KOG+22], funded by the European Union's Horizon 2020 research and innovation program, works on defining a reference architecture for drone autopilot in order to provide development methods to reduce errors. Shi Zheng-Pu and al [ZPMGJG22] also formalised the propulsion subsystem of a flight control system using Coq. In this thesis, we focus on the formal verification of a mathematical library using static analysis techniques and the verification of a code generator.

2.4.1 Static code analysis

Software inspection techniques, now known as static code analysis, were first proposed by Michael E. Fagan in 1976 [Fag76]. Aurum et al. [APW02] published a review in 2002 describing many variations to the Fagan techniques created over the 25 years. There are several formal method domains that are based on static analysis techniques. For instance, deductive methods [Fil11], presented in Section 2.2.4, or abstract interpretation [Cou99, CC92], introduced in Section 2.2.3.2.

Several tools implement static code analysis. For instance, GNATProve for the verification of SPARK code or Frama-C [KKP+15] for C code verification, presented in Chapter 3.

Frama-C has been used on several case studies of embedded or critical systems to verify the absence of runtime errors or functional properties. Todorov [Tod20] verified a discrete-value function, used by PSA, calculating square root using a linear interpolation table and fixed-point numbers. Peyrard and al. [PKDR18] verified the absence of RTE of encryption-decryption modules of Contiki, an operating system for IoT. Pariente and Ledinot [PL10] uses Frama-C on a real critical embedded control program of Dassault Aviation. E Silva and al. [eSAB+16] have used abstract interpretation and deductive verification through Frama-C to verify embedded aerospace control software.

2.4.2 Compiler verification

The verification of a code generator, i.e. ensuring that the generated code behaves as the input code, can be seen as a *compiler verification* problem which has been a known problem for a long time. The first pen-and-paper proof of correctness for a compiler was published by McCarthy and Painter in 1967 [MCP67]. In 2003, there were already at least one hundred articles about compiler verification [Dav03]. There are now several usable verified compilers for many commonly used programming languages, for instance Jinja [KN06], a Java-like programming language with a verified compiler producing Java Virtual Machine byte-code proven in Isabelle/HOL, the Verisoft [LPP05] or CompCert [LBK+16] C compilers (respectively proven in Isabelle and Coq), or CakeML, a verified compiler for Standard ML proven in HOL4 [KMNO14, TMK+16].

Part II presents the work done about formal verification of the flight plan generator using Coq. Our approach is similar to the CompCert C compiler [LBK+16] and the Velus Lustre compiler [BBP20, Bru20] by ensuring that the compiler preserves the semantics. This approach is presented in Chapter 5. Notice that there is also an on-going work [BR19] using Coq to verify a compiler for Esterel, an imperative synchronous language, closer to `FPL`, the flight plan language, than the previously cited languages.

Part I

Static code analysis using Frama-C

Chapter 3

Introduction to the Frama-C plateform

Contents

3.1 Frama-C platform 97
3.1.1 Specifying with ACSL 98
3.1.2 Verifying with EVA or WP 99
3.2 How to verify code with Frama-C: A simple process 101
3.2.1 Absence of runtime errors 101
3.2.2 Functional verification 103

Frama-C (*Framework for Modular Analysis of C-programs*) is a tool developed by the CEA List and Inria to analyse C programs. Frama-C is able to parse a C program and generate an abstract syntax tree representing the program using CIL (C Intermediate Language). It also supports ACSL (*ANSI C Specification Language*) to add annotations to the abstract syntax tree. The architecture of Frama-C supports several plugins that can analyse the abstract syntax tree. For example, the *EVA* plugin applies abstract interpretation methods (see Section 2.2.3.2) and the *WP* plugin applies deductive methods through weakest precondition calculus (see Section 2.2.4.2).

This chapter is not a tutorial about Frama-C. Allan Blanchard's tutorial that introduces Frama-C and the WP plugin [Bla20] is a very good starting point. The official documentation of Frama-C available on its official site is a good reference to learn more deeply about Frama-C and particularly its plugins. However, we want to present the verification processes that have been applied to verify a mathematical library, presented in Chapter 4. Section 3.1 introduces the Frama-C platform and the plugins used. Then, Section 3.2 details the verification process.

3.1 Frama-C platform

Frama-C [KKP+15] is an analysis tool allowing users to formally verify C code using different techniques. It is a modular platform that supports several plugins implementing analysis techniques. For instance, the EVA plugin can be used for static analysis using abstract interpretation whereas the E-ACSL plugin focuses on dynamic analysis. In this thesis, we focus on *static analysis*.

The typical workflow with the Frama-C platform can be summarised in 3 steps:

1. The first step consists in *specifying* the properties that has to be verified. The specification is defined by adding ACSL annotations in the C code as special comments (see Section 3.1.1).
2. The second step is the *generation of the abstract syntax tree (AST)* of the analysed code by Frama-C. The generated tree also contains the previously added annotations.
3. The final step is the analysis of the AST by the plugins. Some plugins might add annotations which correspond to new properties to be verified. For instance, the RTE plugin adds assertions to verify the absence of runtime errors. Other plugins, like EVA or WP, use formal techniques to check the specification (see Section 3.1.2).

3.1.1 Specifying with ACSL

ACSL [BCF+21] is the specification language used by Frama-C to define the expected properties on C code. The ACSL annotations are added in C code as special comments. Using the ACSL specification language, different types of properties can be defined, such as:

- *contracts* for functions. A contract is composed of:
  - *preconditions*, i.e. assertions that specify the expected state of the program when entering the function. Preconditions must be verified by the caller of the function and are considered as holding by the function developer.
    Preconditions are specified using the `requires` special comment in C code.
  - *postconditions*, i.e. assertions that specify the guaranteed state of the program after the function execution.
    Postconditions are specified using the `ensures` special comment.
  - *frame specifications*, i.e. all the memory elements that will be modified during the execution of the function.
    Frame specifications are specified using the `assigns` special comment.
- loop annotations among which:
  - *invariants* that specify properties that hold when entering the loop, at the end of each iteration of the loop and when exiting the loop. Invariants are important properties as they are the only assertions that can be used after a loop for proving properties on a program.
    Invariants are introduced using the `loop invariant` special comment.
  - *variants*, expressions that are used to prove the termination of the loop (classically a strictly decreasing expression taking its values in $\mathbb{N}$).
    Variants are introduced using the `loop variant` special comment.
- *assertions* to specify properties that must be satisfied at a particular point of the program.
  Assertions are specified using the `assert` special comment.

Remark

ACSL offers also the possibility to define types, functions, lemmas and predicates to express functional properties in order to ease specification writing. See section 3.2.2 for more details.

In order to verify specific properties on a code, such as functional properties, annotations must be manually added to the code. However, some plugins can automatically add annotations corresponding to the verification of specific properties, such as the *RTE (RunTime Errors)* plugin that automatically adds assertions to verify the absence of runtime errors. These errors come generally from an incorrect implementation of a correct algorithm or borderline cases that were not taken into account. The RTE plugin supports common runtimes errors in C such as divisions by 0, overflows or dereferencing invalid pointers. The complete list of runtime errors taken into account by RTE can be found in the plugin documentation [dt21c]. As we used the RTE plugin, the Section 3.2.1 presents the process for the verification of such runtime errors.

3.1.2 Verifying with EVA or WP

As briefly presented above, Frama-C has many plugins (see [dt21b] for an exhaustive list). In this work, we focus only on three of them: RTE (already presented), EVA and WP. These three plugins can be combined together for a more efficient or complete analysis. For instance, RTE adds assertions about runtime errors that can then be proven by WP or EVA, or EVA can be run before WP to help verifying assertions. This verification process is detailed in Section 3.2.

The figure 3.1 is an overview on how Frama-C works with WP and EVA plugins. In the following, we detail how these plugins work.

3.1.2.1 EVA

EVA (*Evolved Value Analysis*) is a plugin that uses abstract interpretation [CC77] (see Section 2.2.3.2) to compute a domain of values for each variable in the program. This plugin is very efficient for verifying the absence of numerical runtime errors such as division by 0 or overflows. EVA is able to compute accurate intervals of potential values for the program variables or intermediate results. The verification then consists in checking if the faulty values are accessible in the computed intervals.

For full information about EVA plugin, see its documentation [dt21a].

3.1.2.2 WP

WP (*Weakest Precondition*) is a plugin that analyses the CIL abstract syntax tree generated by Frama-C. The analysis consists in computing the weakest preconditions [Dij75] of the program and generating the verification conditions (VC) from code annotations. VCs are pure logic formulas and proving these formulas guarantees the correctness of the code. The VCs are then converted into WhyML code which is a language provided by the Why3

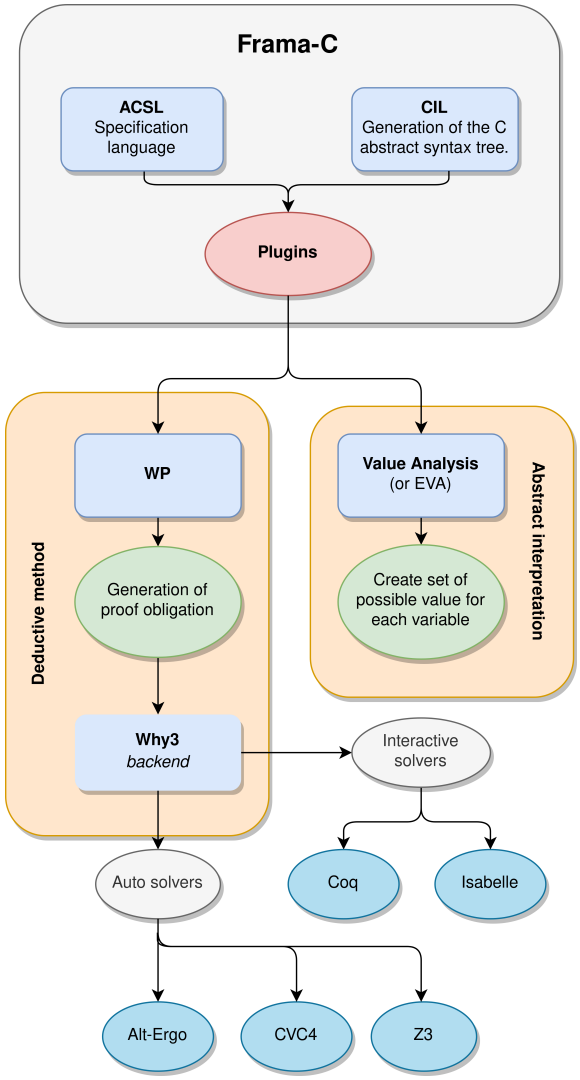

Figure 3.1: Diagram about Frama-C

platform. Why3 [Tea20] is a deductive verification platform developed by Jean-Christophe Filliâtre and others [Fil11]. Why3 is interfaced with solvers that are used to *discharge* VCs, i.e. prove them through theorem provers. Proofs can be done by:

- automatic solvers, like SMT solvers [BSST09] or automatic first-order theorem provers. Why3 is compatible with Alt-Ergo, Z3 or CVC4 SMT solvers.
- applying tactics that transform the VC if automatic provers fail to prove it (automatic provers cannot decide the truth value of all possible first-order formulas). For instance, you can apply a tactic that instantiates a bounded integer variable with all possible values and prove the property for each instantiated VC.

- manual proof assistants like Coq [The23] if automatic theorem provers and tactics fail. This is of course a more difficult exercise, but is sometimes necessary to convince yourself that the VC can be proven.

WP also provides several memory models for different levels of abstraction of C programs. The general process to conclude a proof using WP is detailed in Section 3.2 and can also be found in the official documentation of the plugin [dt21d].

3.2 How to verify code with Frama-C: A simple process

This section presents the verification process *we* used to verify parts of the mathematical library of the Paparazzi autopilot [PGH+21], presented in Chapter 4. For each function of the library, we have defined contracts that ensure the expected properties on the behaviour of the function. We focused on the verification of runtime errors (see Section 3.2.1) and the verification of functional properties (see Section 3.2.2) using the RTE, WP, and EVA plugins.

3.2.1 Absence of runtime errors

The first goal of our verification process is to prove the absence of runtime errors. We therefore want to find the minimal preconditions for the functions of the library that will ensure that no runtime errors that will occur. This process can be summarised into 3 steps:

1. Launch the analysis on the non annotated code with Frama-C using the RTE, WP, and EVA plugins. The RTE plugin will add assertions corresponding to potential runtime errors and the plugins EVA and WP will try to verify the assertions.
2. By analysing the assertions that are not proven by EVA nor WP, it is then possible to deduce the missing information in the contracts of the functions. It may also be necessary to specify loop variants and invariants.
3. Repeat the 2 previous steps until all assertions generated by the RTE plugin are verified.

Remark

Some preconditions that have been added to verify the absence of RTE during the process may not finally be needed. In order to have minimal contracts (i.e. the weakest precondition), it thus can be necessary to consider a fourth step to clean the contract by removing these unneeded preconditions.

The RTE plugin generates assertions that can be classified into 2 main categories: assertions corresponding to dereferencing potentially invalid pointers and assertions concerning possible overflows during arithmetic operations. The verification of the first category of assertions is straightforward as it only requires to add preconditions stating that the pointers passed as parameters of the function must be valid.

The second category requires more work from the specifier. Such assertions correspond to potential overflows that can occur during the computation of arithmetic operations. These computations often involve the function parameters. Bounds for the values passed as arguments when calling the function must therefore be computed in order to guarantee the absence of overflows.

In order to understand how to determine the bounds of variables, let us take a simple addition function as an example.

Example 3.1 - Simple C addition function.

```
int32_t dummy_add(int32_t a, int32_t b) {
    return a + b;
}
```

The function `dummy_add` takes two parameters `a` and `b` that are both signed integers coded on 32 bits and return a 32 bits signed integer value corresponding to the addition of `a` and `b`.

We denote by `INT_MAX` the maximum value that can be represented by 32 bits signed integers. If the function `dummy_add` is called with `a` = `b` = `INT_MAX`, then there will of course be an overflow as the result cannot be represented on a 32 bits integer (`INT_MAX` + `INT_MAX` > `INT_MAX`).

There are several ways to determine the bounds of the parameters and variables of the function depending on the information about the parameters that is known by the specifier:

- some values may already be bounded (e.g. $a \in [-1;1]$)
- some variables may be linked (e.g. $a = 2b$)
- no information on the function parameters may be known

Considering the verification of Paparazzi mathematical library, no information about the parameters of the functions were given. We thus decided to compute equal bounds for the parameters. For instance, for the previous `dummy_add` function, we search a bound M such that:

$$\begin{aligned} \forall a, b, \quad & -M \leq a \leq M \\ \wedge\, & -M \leq b \leq M \\ \Longrightarrow\, & INT\_MIN \leq a+b \leq INT\_MAX \end{aligned}$$

A simple bound can easily be deduced without further information: $M = \dfrac{INT\_MAX}{2}$, if we know that $INT\_MIN \leq -INT\_MAX$. The following contract guarantees therefore the absence of overflow:

```
/*@
    requires -INT_MAX/2 <= a <= INT_MAX/2;
    requires -INT_MAX/2 <= b <= INT_MAX/2;
*/
int32_t dummy_add(int32_t a, int32_t b){
    return a + b;
}
```

Remark

Such contract guarantees that there will be no overflows if the contract preconditions are respected when the function is called. It is therefore necessary to verify if all the call sites of the function respect its preconditions.

Depending on the code under analysis, it is possible to encounter specific problems. In some cases, it may be necessary to use a special *memory model* that is more fitted for the code to be proven correct. For example, Frama-C offers a memory model called `ref` that is particularly adapted when function parameters are passed by reference with pointers, i.e. using pointers, as WP struggles to manage pointers with its default memory model. The `ref` model creates aliases for the parameters passed by references allowing WP to manage these references as "simple" variables which simplifies verification. The different memory models are presented in detail in Frama-C documentation [dt21b] and in [PGH+22].

Remark

The functional specification of a function or a program, presented in the next section, is not always trivial and most of the time the proof is even more difficult. In massive programs not designed with correctness proof in mind, writing a specification and proving it are very time-consuming. Unfortunately, most companies cannot afford to spend this time on verification. A partial solution is to use WP, EVA and RTE as presented above. The developer has just to write minimal contracts for each function. The proofs are then easier and can be realised automatically. This technique allows any company to ensure the absence of RTE for a minimal cost.

3.2.2 Functional verification

Functional verification aims at offering guarantees on the behaviour or the result of a function. Let us present a simple example to detail what kind of properties can be verified and consider a function that takes a positive value x then computes and returns its square root. We denote by r the result of the function ($r := \texttt{sqrt}(x)$) that can either be an integer, a float or real value. Here are different properties that are expected to be verified on r:

1. The result is positive: $r \geq 0$,
2. The result is smaller than the argument of the function: $r \leq x$ (when $x \geq 1$),
3. The result squared is equal to the input: $r^2 = x$.

These three points are examples of what we call "functional properties". Depending on the algorithm used to compute r, we should expect to verify the first two properties more or less easily. The third property is mathematically correct, but is no more correct in real code using floating-point arithmetic due to potential rounding errors. For instance, we know "mathematically" that `sqrt`(2) is an irrational number and as computers are limited to represent floating-point values[1], the result returned by the function is an approximation of the mathematical result ($r \neq \sqrt{2}$). In this case, it is therefore be impossible to verify the property.

Rounding errors are often a difficult point for functional verification, implying that some mathematical properties cannot be proven because they are wrong with floating-point arithmetic. A workaround to this problem with Frama-C consists in using a WP arithmetic model called `real`. This model considers that floating-point operations are *mathematical* operations over *real numbers*. With this model, it is possible to verify mathematical properties on a C code, such as the third property expected for the square root function. This model was used to verify functional properties on floating-point functions of Paparrazi mathematical library.

Remark

The `real` model allows us to verify functional properties, but it is important to know that it is an uncorrect model of floating-point computations. Particularly, it does not respect the C programming language semantics and therefore, verification with this model does not guarantee any of the results. It is not possible to ensure that properties are verified, but at least it allows us to verify that the code has a correct meaning in a mathematical sense.

Functional properties verification process can then be separated into two parts: (i) specification of the expected properties and (ii) verification of these properties.

The first step requires a formal specification language with enough expressiveness. ACSL, the language used to define the contracts of functions can be used and allows users to define:

- *Types* that can describe mathematical types corresponding to types defined in the code, e.g. vectors, matrix, quaternions...
- *Elementary functions* that can operate on user defined types, e.g. addition of vectors.
- *Lemmas* to ensure that the mathematical definitions respect their mathematical properties, e.g. commutativity of the vectors addition. The verification of these lemmas is done when WP is launched. Chapter 4 presents an example of defined lemmas.
- *Predicates* that use elementary functions to express properties on the defined types, e.g. a predicate that states that a matrix M is a rotation matrix if and only if $M.M = I$, where I is the identity matrix and that the determinant of M is equal to 1.

[1] The floating-point values are only able to represent a finite number of digits and real value such as 0.1 cannot be correctly encoded.

Predicates can then be used in functions contracts to specify the functional properties that must be verified.

The second part of the process is the verification of the functional part of the functions contracts, but also of the lemmas, using WP. Lemmas must of course be verified to ensure that the functions are correct, but also can be used to facilitate the verification of the contracts. The WP plugin generates Verification Conditions (VC) that correspond to these contracts and lemmas that must be verified. Frama-C proposes different methods for the verification of the VC, which are presented below, from most automatic to least automatic.

3.2.2.1 Using SMT solvers

WP generates VC for the Why3 platform [Tea20]. As stated before, Why3 is a platform for deductive verification which allows using automatic SMT solvers such as Alt-Ergo, Z3 or CVC4. Each SMT solver has its pros and cons: for instance, Z3 is better for solving pure SAT problems whereas Alt-Ergo has a built-in floating-point library. It is therefore advised to attempt proving a VC with all SMT solvers in parallel.

3.2.2.2 Apply general tactics

WP has several predefined interactive tactics that can be applied on unproven goals. The idea of tactics is to "simplify" the current goal in order to help the SMT solvers. There are tactics to unfold definitions, which is useful when there are a great number of ACSL predicate definitions, or to split a conjunctive goal into two simple subgoals. These tactics are applied manually through the Frama-C graphical interface. When a goal is proven using tactics, it is possible to save the proof script. All applied tactics will be saved in a JSON file and when relaunching Frama-C, the tactics will be automatically applied on the saved goals.

3.2.2.3 Enable interactive mode to use Coq

Automatic solvers are sometimes unable to verify a goal, even with a large timeout and with the help of tactics. There are two cases:

- The goal cannot effectively be proven because it is false. This can either be that the property is not correct or the hypotheses are not correctly specified.
- Automatic solvers cannot prove the goal due to its complexity or their limitations.

We only consider here the second case, i.e. that the goal requires specific reasoning beyond the scope of automatic SMT solvers. During the verification of the mathematical library, we faced some examples that cannot be proven using automatic solvers and tactics, as seen in Section 4.3. The last method to verify this type of VC is to use the "interactive mode" of Frama-C. This mode enables the use of the Coq proof assistant to verify manually the VC. Similarly to the tactics, when Coq proof for a VC is done, it can be saved to be replayed when Frama-C will be relaunched.

The document [PGH+22] presents in details how to enable and use these methods in Frama-C.

Chapter 4

Verifying the Mathematical Library of a UAV Autopilot with Frama-C

Contents

4.1 Proving the absence of runtime errors 107
4.2 Functional verification using automatic provers 109
4.3 Functional verification using interactive provers 112
4.4 Discussion 113

The Paparazzi mathematical library studied in this thesis is an important component of the drone autopilot. Indeed, the library provides a model for drone attitude through different representations (rotation matrices, Euler angles, or quaternions) but also conversion and manipulation functions over these representations. The library is written in C with 3 implementations using double values, float values and fixed point values. This library is a critical component, as a part of the Paparazzi *State interface* that is connected to estimation filters and sensors that acquire data provided through different data representations. Crucially, the autopilot uses this data to take decisions.

We verified this library by applying the verification process presented in Section 3.2 using Frama-C. Section 4.1 details the analysis concerning the absence of runtime errors. The second part of the analysis covers the verification of functional properties for some state representation transformation functions. Section 4.2 details the verification process using only automatic provers. As automatic provers were not able to prove some of these functions, Section 4.3 presents how we proved such functions using the interactive prover Coq [The23]. Finally, Section 4.4 discuss lessons learned during the work and present some perspectives.

4.1 Proving the absence of runtime errors

The studied library defines C structures for the different representations (rotation matrices, quaternions, vectors, etc.). The library functions take only pointers on such structures as inputs and always return pointers. Preconditions ensuring the validity of pointers have been added in the contracts as the functions are not designed to work with invalid pointers. It has also been necessary to specify which variables will be modified during the execution of the function. Finally, invariants on `for` loop have been added to help the provers to ensure the absence of runtime errors.

```
/*@
  requires valid_float_mat(o, m, n);
  requires rvalid_float_mat(a, m, n);
  requires rvalid_float_mat(b, m, n);
  assigns o[0..m-1][0..n-1];
*/
void float_mat_sum(float **o, float **a, float **b, int m, int n)
{
  int i, j;
  /*@
    loop invariant 0 <= i <= m;
    loop assigns o[0..m-1][0..n-1], i, j;
    loop variant m - i;
  */
  for (i = 0; i < m; i++) {
    /*@
      loop invariant 0 <= j <= n;
      loop assigns o[i][0..n-1], j;
      loop variant n - j;
    */
    for (j = 0; j < n; j++) { o[i][j] = a[i][j] + b[i][j]; }
  }
}
```

Figure 4.1:(`float_mat_sum`, sw/airborne/math/pprz_algebra_float.h:1340)

Figure 4.1 shows an example of C code annotated for the function `float_mat_sum` that computes the addition of two matrices pointed by `a` and `b`. The result is written in the matrix referenced by the pointer `o`. The contract states as preconditions that all elements of matrices of `a` and `b` must be accessible for reading (predicate `rvalid_float_mat`) and that the elements of `o` should be accessible for reading and writing (predicate `valid_float_mat`). The loops are also annotated with variants and invariants that are classic for such `for` loops. Every loop of the library is annotated similarly as they are mainly used to go through all the elements of matrices.

The WP plugin of Frama-C offers different models of arithmetic that take into account more or less precisely C semantics. The verification of the `int` flavor of the functions was made using 32-bit integer arithmetic with overflows. When using the RTE plugin to verify the absence of runtime errors, assertions are automatically added to check that there is no overflow for each arithmetical operation. To verify this, each function was manually analysed to determine the maximum possible value for each variable. When bounds of the variables have been determined, they were added as preconditions in the function contracts. Unfortunately, WP associated with automatic provers is not able to verify these new contracts. Even if the complete memory separation of structures used in a function is specified as precondition and if the `ref` memory model presented in Section 3.2.1 is used, the solvers are unable to prove that the modification of a field in a structure does not change any other part of the memory. We found that the WP plugin is not really adapted

when functions used pointers as parameters, even with the `ref` memory model. WP seems to not always use `ref` model even when it is manually enabled. WP is then "overloaded" by accesses to values by reference in the code and VC generated by WP concerning the absence of overflows cannot be discharged anymore by automatic solvers. To overcome this problem, we decided to associate the EVA plugin with WP. EVA has no issue dealing with pointers nor aliasing and is able to compute accurate intervals of possible values for each variable. The result produced by EVA makes it easier to conclude some proofs. We cannot use only EVA as it cannot verify loop variants and invariants unlike WP. We thus verify the absence of runtime errors for the functions using `int` values by associating EVA and WP.

Remark

This WP limitation when pointers are extensively used as input and output parameters was also found by Vassil Todorov during his PhD thesis [Tod20]. He also used a static analysis tool using abstract interpretation, Astrée [CCF+09] associated with WP, to solve the same problem.

WP has also an arithmetical model `real` for real arithmetic. We decided to use this model for the verification of the library functions working on floating-point values. Using the same precondition used for the `int` version of the functions, such a model allows us to verify the absence of division by zero and more generally any occurrence of the `NaN` value during computation. To perform these verifications, it was only necessary to add as preconditions the fact that each pointer refers to a valid address. The absence of these two kinds of runtime errors as well as the termination of the functions have been proven, using WP and EVA, for the `float` and `double` versions of the library. Unfortunately, our verification does not offer any guarantee on the risk of floating-point overflow or on rounding errors. Moreover, the properties proven for the `real` model can only serve as an hint, but not a guarantee, that they hold for floating-point values perhaps in some approximate sense. However, this model was particularly useful to verify functional properties as presented in section 4.2. Indeed, even if the model is semantically incorrect and we cannot get functional guarantees during execution, it ensures at least that the code is correct in the mathematical sense.

4.2 Functional verification using automatic provers

The goal of functional verification is to ensure properties about the behaviour of functions. We first focus here on the function `float_rmat_of_quat` to explain the process used. This function takes a normalised quaternion as input and returns the corresponding rotation matrix.

In order to specify the functional properties of such a function, types and predicates have been defined in the logic provided by ACSL [BCF+21], the language used to express Frama-C annotations. We defined types for matrices and quaternions, as well as elementary algebraic operations. We specified lemmas and then verified them to ensure that these operations are correct (e.g. we verified that matrix transposition is idempotent). Then, a logical function

that converts a quaternion to a rotation matrix has been defined independently of the C code from the library. This function is based on the mathematical equation that expresses the conversion of a quaternion to a rotation matrix [Gru70, Klu76]. In the following, we note `rmat_of_quat` the function that represents this conversion. The `rmat_of_quat`'s semantics, noted `l_RMat_of_FloatQuat` in the C development, is expressed in ACSL as a mathematical specification. This function takes as parameter a unitary quaternion q and returns a rotation matrix. Frama-C is able to verify automatically that for any given unitary quaternion q, the rotation matrix computed by `rmat_of_quat` corresponds to the same rotation as described by the quaternion. Verification of this property has required to verify the following lemma:

Lemma 4.1 - (`correctness_rmat_of_quat`, `sw/airborne/math/pprz_algebra_float_func.h:520`).
$\forall q \in \mathbb{H}, v \in \mathbb{R}^3,\ q(0,v)q^* = (0, \texttt{rmat\_of\_quat}(q).v)$

This lemma states that given a quaternion q and a vector v, applying the rotation with the quaternion q on vector v is equivalent to applying the rotation matrix obtained from q by `rmat_of_quat` on v.

As shown in Figure 4.2, the contract for the function `float_rmat_of_quat` has then been established using these ACSL functions. Assuming that the quaternion passed as parameter is normalised, we wanted to verify two functional properties. The first one is that the returned matrix does indeed correspond to the conversion of the quaternion passed as a parameter: our post-condition verifies that the rotation matrix generated by the C code is equal to the rotation matrix generated by our logical function `rmat_of_quat`. Note that we use the function `l_RMat_of_FloatRMat` that convert the C structure representing a rotation matrix into the structure defined in the ACSL logic. As presented in the previous section, we use the WP `real` model for the verification of this property, thus ignoring the differences in the results between the mathematical version and the C version which could have been introduced by rounding errors. The second verified property is that the generated matrix is indeed a rotation matrix, i.e. the transpose of the matrix is its inverse, and its determinant is equal to 1.

Despite the use of the `real` arithmetic model, WP could not verify this contract. It was therefore necessary to manually review the code. We noticed that the C code used a constant `M_SQRT2` to represent $\sqrt{2}$. By analysing the calculations done in the code, we realised that the constant `M_SQRT2` was always multiplied by itself. We therefore suggested a code modification that replaces `M_SQRT2 * M_SQRT2` by 2 (Figure 4.2 shows the modified version of the code). This modification does not change the number of multiplications in the C code but cancels the rounding errors propagated by the function. With this code change and the arithmetic model `real`, WP verifies the contract of the function `float_rmat_of_quat`.

Remark

Notice that the code presented in Figure 4.2 contains several ACSL assertion of the form "`\at(val, Pre) == val`" that ensures that the value of the parameter `val` has not changed since the start of the execution of the function. These assertions were needed to help the solvers to verify the contracts when using WP. As explained previously, WP is overloaded when functions use pointers.

```
#define RMAT_ELMT(_m, _row, _col) ((_m).m[(_row)*3+(_col)])
/*@
    requires \valid(rm);
    requires rvalid_FloatQuat(q);
    requires unitary_quaternion(q);
    requires \separated(rm, q);
    ensures l_RMat_of_FloatRMat(rm) ==  l_RMat_of_FloatQuat(q);
    ensures rotation_matrix(l_RMat_of_FloatRMat(rm));
    assigns *rm;
*/
  void float_rmat_of_quat(struct FloatRMat *rm, struct FloatQuat *q){
    const float _a = q->qi;
    const float _2a = 2.f * _a;
    const float _b = q->qx;
    const float _2b = 2.f * _b;
    const float _c = q->qy;
    const float _d = q->qz;
    const float _2d = 2.f * _d;
    const float a2_1 = _2a * _a - 1;
    const float ab = _2a * _b;
    const float ac = _2a * _c;
    const float ad = _2a * _d;
    const float bc = _2b * _c;
    const float bd = _2b * _d;
    const float cd = _c * _2d;
    //@ assert \at(q->qi, Pre) == q->qi;
    //@ assert \at(q->qx, Pre) == q->qx;
    //@ assert \at(q->qy, Pre) == q->qy;
    //@ assert \at(q->qz, Pre) == q->qz;
    RMAT_ELMT(*rm, 0, 0) = a2_1 + _2b * _b;
    RMAT_ELMT(*rm, 0, 1) = bc + ad;
    RMAT_ELMT(*rm, 0, 2) = bd - ac;
    RMAT_ELMT(*rm, 1, 0) = bc - ad;
    RMAT_ELMT(*rm, 1, 1) = a2_1 + 2.f * _c * _c;
    //@ assert \at(q->qi, Pre) == q->qi;
    RMAT_ELMT(*rm, 1, 2) = cd + ab;
    RMAT_ELMT(*rm, 2, 0) = bd + ac;
    RMAT_ELMT(*rm, 2, 1) = cd - ab;
    RMAT_ELMT(*rm, 2, 2) = a2_1 + _2d * _d;
    //@ assert \at(q->qi, Pre) == q->qi;
    //@ assert \at(q->qx, Pre) == q->qx;
    //@ assert \at(q->qy, Pre) == q->qy;
    //@ assert \at(q->qz, Pre) == q->qz;
}
```

Figure 4.2:(float_rmat_of_quat, sw/airborne/math/pprz_algebra_float.h:639)

4.3 Functional verification using interactive provers

The same verification has been attempted for the inverse function `float_quat_of_rmat` that converts a rotation matrix into a quaternion. There are different equations to perform this conversion in the literature but we use the four formulae of Shepperd's method [She78,ST19]. These equations are directly deduced from the formula that converts a quaternion into a rotation matrix and are defined according to the diagonal values of the rotation matrix: one is defined when the trace is strictly positive, the three other ones are defined when the trace is negative and correspond to the possible choices for the greatest element of the diagonal of the matrix. We defined each of these formulae by an ACSL logical function. When defining postconditions of C functions, we use ACSL `behaviour` feature to specify one sub-contract per Shepperd's case, specifying as preconditions in the sub-contract the conditions for which the corresponding Shepperd's function is defined. For instance, we defined a behaviour that requires as precondition that the trace of the input matrix is positive. These behaviours then ensure, as postconditions, that the quaternion computed by the C function is equal to the quaternion computed using the corresponding logical function. Another feature offered by Frama-C is the possibility to verify that the behaviours are disjoint and complete, i.e. for every input matrix, there is one and only one behaviour such that its preconditions are fulfilled by the matrix. With this contract, we were able to verify that the function returns a quaternion that corresponds to the same rotation as the matrix used as input.

Let us denote by `quat_of_rmat` the mathematical function that returns the quaternion corresponding to a given rotation matrix. Let us consider here the case where the rotation matrix used as input has a positive trace. Verifying that `quat_of_rmat` is correct in this case is equivalent to verifying that the property described on the following lemma holds:

Lemma 4.2 - (`correctness_quat_of_rmat_pos`, `sw/airborne/math/pprz_algebra_float_func.h:644`).

$$\forall R \in M_{3,3}(\mathbb{R}),\ q \in \mathbb{H},\ \ \|q\| = 1 \ \wedge\ Tr(R) > 0 \\ \rightarrow (R = \texttt{rmat\_of\_quat}(q) \leftrightarrow q = \texttt{quat\_of\_rmat}(R))$$

This lemma can be read as follows: for all matrices R with a positive trace and for all unitary quaternions q, R represents the rotation matrix obtained from $\texttt{rmat\_of\_quat}(q)$, if and only if the function `quat_of_rmat` returns q when applied to R. It has then been translated into an ACSL lemma, as well as the similar equations resulting from the three other cases.

Unfortunately, Frama-C is not able to prove these lemmas using only automatic SMT solvers, even after extending the timeout value considerably. The proof requires specific transformations, such as factorization, that the solvers might not be able to find. We therefore had to use the interactive mode of WP. This mode generates incomplete proof scripts for each unproven goal. The scripts contain all the definitions and lemmas that have already been proven by Frama-C and the solvers. The theorem corresponding to an unproven goal needs to be verified with some interactive prover, in our case Coq (see Section 2.3). The implication that if R is obtained from $\texttt{rmat\_of\_quat}(q)$, then the function `quat_of_rmat` returns q has been verified with Coq for the four lemmas (for example

lemma_impl_rmat_quat_trace_pos.v for the positive case). The reverse implication has not been proven yet. However, by considering the verification of the function `rmat_of_quat`, this proof is sufficient to guarantee that the result of the function `quat_of_rmat` describes the same rotation than the input matrix.

We also wanted to verify the function implementing the conversion from the Euler representation of a rotation to a rotation matrix. In the library, there are two functions, `float_rmat_of_eulers_321` and `float_rmat_of_eulers_312`, that implement this conversion. These two functions differ on the order of Euler angles (given the $(\vec{z},\vec{y},\vec{x})$ axis for the `321` function and given the $(\vec{z},\vec{x},\vec{y})$ axis for the `312` function). The contracts defined for these functions ensure that the matrix must be special orthogonal, i.e. a rotation matrix. In order to verify these contracts, we started using only automatic SMT solvers. The first problem we faced was that the code for the conversion uses the `cosf` and `sinf` trigonometric functions from the C standard library. Frama-C equips these built-in functions with contracts, but these contracts do not provide enough information. We decided to add an hypothesis in the contract stating that the result of these functions was equal to the result obtained with the corresponding mathematical trigonometric function of ACSL (defined by `\cos` and `\sin`). This hypothesis might not be correct. However, as we use the `real` model, this hypothesis enables the use of properties of trigonometric functions, for instance:

Lemma 4.3 - (`cos_sin_square`, `sw/airborne/math/pprz_algebra_float_func.h:715`).

$$\forall a \in \mathbb{R},\ \cos^2(a)+\sin^2(a)=1$$

Unfortunately, Frama-C was not able to prove the postcondition of the conversion functions, even with this hypothesis and WP tactics. Even by using tactics, there were remaining unproven subgoals. Instead of using Coq to verify the whole postcondition, we decided to define generic lemmas in ACSL which correspond to the subgoals unproven by SMT solvers and verify them in Coq. The following lemma is an example of such a subgoal, which can easily be proven by hand using factorization and properties of trigonometric functions.

Lemma 4.4 - (`float_rmat_of_eulers_321_1`, `sw/airborne/math/pprz_algebra_float_func.h:721`).

$$\forall a,\ b,\ c \in \mathbb{R},\ \sin^2(a).\cos^2(b)+(\sin(a).\sin(b).\cos(c)-\sin(c).\cos(a))^2 \\ +(\cos(c).\cos(a)+\sin(a).\sin(b).\sin(c))^2=1$$

In order to facilitate the proof in Coq, we define some subproperties in ACSL that are verified by the solvers and then used in the Coq proof. For example, the Lemma 4.3 has been verified by the solvers and then be used to verify the Lemma 4.4. This technique has also been used for the verification of the Lemma (4.2).

4.4 Discussion

We have presented in this Chapter a work on the formal verification of a mathematical library using Frama-C. We mainly focused on ensuring the absence of runtime errors, but

we also proven interesting properties of rather complex functions. However, this work is not complete as the verification of the mathematical library can be improved. There are several perspectives briefly detailed in the following.

First, some proofs are remaining, such as the reverse implication of Lemma 4.2 presented in Section 4.3. We could also verify functional properties on other functions, such as the initialization function for rotation matrix or the composition of two quaternions.

Second, a central point we do not tackle is the presence of potential rounding errors. In fact, after verifying that the code behaves similarly to the mathematical specification using the `real` arithmetical model, we could use the `float` arithmetical model that is correct with respect to IEEE specifications. We could have taken into account potential rounding errors generated during the computation and therefore obtained a bound on such errors. The idea is to offer guarantees that the results obtained are close enough to the expected mathematical calculation with a negligible difference. We should of course define formally the term "negligible" in the Paparazzi autopilot context. Such verification would have been possible but it would have taken time and would require the use of additional *driver* or Frama-C plugins as solvers do not fully support the verification of floating-point properties.

The aim of this mainly experimental work was to work out the limitations of the chosen approach and its applicability by engineers. The verification of runtime errors using WP requires at least some basic knowledge about deductive methods for loop annotations with loop variants and invariants. The verification of functional properties is more complex. This type of verification requires the capacity to formally define the expected properties, i.e. formally define the specifications and more importantly, decide which arithmetic and memory models to use. Indeed, as presented in Chapter 3 on the square root example, it is impossible to prove that a `sqrt` C function always returns the correct mathematical result, using the semantics of C floating-point computation. Users must thus understand the pros and cons of the different models to make the most judicious choice based on their needs. They may also need to know different plugins to combine them in order to get the most of their combined strengths.

Frama-C is a powerful tool, but we find some limits to its usage. First, the graphical interface had to be restarted each time the code was modified. It is not possible to edit the code and restart the solvers directly from the GUI. We do not know if the new Ivette graphical interface fixes this problem, but it wasted a lot of time. Second, when errors are encountered using Frama-C, there are three possible cases: 1) You may incorrectly use the tools, so you have to fix the annotations or the parameters used for the solver. 2) It may be a bug of Frama-C, a plugin or a dependent application (e.g. Why3). You can either directly fix the issue by looking into the code as it is an open-source project or submit an issue and wait for a fix. 3) You may use a feature that is unimplemented despite being presented in the documentation (it typically concerns ACSL annotations). In order to understand if the feature is not implemented, you can either look at the corresponding code or ask to the Frama-C mailing list. Frama-C developers plan to have a more precise documentation in the future to prevent users to use unimplemented features. When encountering such problems, the main issue is to understand which one of these three cases is currently happening. The

universal solution is thus to ask in the mailing list as they are very responsive. Finally, the last issue that may discourage the use Frama-C is that it continuously evolves with a new major version every 6 months. A new version might bring new features for the verification of new properties, but might also break previous proofs. Indeed, features used for the verification of properties may be removed, or solvers may be updated and can no longer prove some goals (due to a change in heuristics or a the correction of a soundness bug). In any case, having a program proven in a version of the tool that is not proven in the newest version raises some trust issue, and we can wonder if in the first place the program was indeed proven correct. In addition to this trust issue, maintaining the proof for the different versions thus costs time. We faced this version issue as the mathematical library was initially verified using Frama-C 23.0 (Vanadium) launched in July 2021, but some goals are no longer proven with the latest version currently available, Frama-C 27.1 (Cobalt). It would be nice to update the project to support the latest version and further investigate on why some goals cannot be proven anymore, which remains to be done.

Part II

Verified compiler in Coq

Chapter 5

Verified compilation using Coq

Contents

5.1 The concept of semantic preservation 120
5.1.1 Generic notion of semantic preservation 120
5.1.2 Simulation diagrams for transition semantics 121
5.2 Introduction to Clight Semantics 123
5.2.1 Clight Syntax 123
5.2.2 Small-step semantics of Clight 126
5.2.3 Execution of a Clight program 134

Compilers are softwares that translate an input language into another language, such as GCC that converts C code into machine code. These softwares may also perform some verifications by refusing incorrect input code (e.g. the code is not well-typed). Compilers are widely used and have been around for many years. Even if being known for their performance to produce efficient code, they are complex pieces of software that may introduce bugs [ER08, SLS16]. Indeed, they might perform some optimisation producing very efficient code. However, the semantics of the optimised code, i.e. how it behaves, might be different from the input code. Ensuring that a compiler is correct by verifying that the *semantics is preserved*, is thus an important topic that must be tackled, especially for the compilation of embedded code for critical systems.

Formally verifying the correctness of a compiler can be achieved by three main techniques:

Verified compilers This type of compiler is accompanied by a formal proof stating that if it can produce an output (there is no error generated), then the semantics is preserved.

Verified validator The compiler includes a validator which verifies whether the semantics of the produced code corresponds to the input, after the compilation. This technique is also called *translation validation* [PSS98, Nec00, BGGT23].

Certifying compilers In addition to the code generated, the compiler produces a proof of correctness, also called a *certificate*, that an external checker can verify. This technique corresponds to the *proof-carrying* approach [Nec97, FNSG07].

These techniques might also be associated such as CompCert [LBK+16] which is a verified compiler for the C input language to assembly code and provides a verified validator called *Valex* that verifies the assembling and linking stages.

The goal of this chapter is to present all the needed notions that we used for the verification of the flight plan generator. Our verification approach, which is detailed in Chapter 6, is similar to the one used for Velus [BBP20, Bru20], a verified compiler for Lustre proven in Coq. Indeed, we proved that our generator preserves the semantics between an input flight plan and the produced Clight code. Clight is a Gallina structure representing C code, defined in CompCert.

The notion of semantic preservation is explained in Section 5.1. The Clight semantics used for the semantic preservation proof is introduced in Section 5.2 and corresponds to the Clight semantics defined in CompCert.

5.1 The concept of semantic preservation

Verified compilers correction consists in proving that the semantics of the source program is preserved during the code generation. This notion of *semantic preservation* is well presented by Leroy [Ler09a, Ler09b] and it is summarised in this section.

5.1.1 Generic notion of semantic preservation

We consider a source program S and the resulting program C after the compilation with the compiler $Comp$ without any errors ($Comp(S) = \texttt{OK}(C)$). In order to prove that the compiler is correct, we want to prove that S semantics is preserved. We thus define the formal semantics for both input and target languages, noted $S \Downarrow B$ and $C \Downarrow B$, where B is a behavior observed by semantics (e.g. external calls, memory access, etc.).

Bisimulation is the strongest notion of semantic preservation stating that both S and C programs have the same observable behaviours.

Definition 5.1 - Bisimulation.
$\forall\ B,\ S \Downarrow B \ \Leftrightarrow\ C \Downarrow B$

However, this definition is often too strong for compiler verification to be used in practice. Some input languages are underspecified and in this case the compiler implementer is authorised to choose one of the possible behaviours of the source program. For example, the C standard does not formally define the evaluation order for expressions and C compilers can select one in particular. The compiled program C has thus less behaviours than the source program S. We thus define semantic preservation as a refinement property [AL88], also called a backward simulation property:

Definition 5.2 - Backward simulation.
$\forall\ B,\ C \Downarrow B \ \Rightarrow\ S \Downarrow B$

The backward simulation property states that any behaviour of C correspond to a behaviour of S. In other words, the execution of a compiled program cannot have behaviours that are undefined in the semantics of the input language.

The dual notion for backward simulation is the *forward simulation* property.

Definition 5.3 - Forward simulation.
$\forall\ B,\ S \Downarrow B\ \Rightarrow\ C \Downarrow B$

The forward simulation property states that any behaviour described by the input language semantics can happen during the execution of the compiled program. The produced program thus does not contain less possible behaviours than the input but the program C can have other behaviours not defined in S semantics.

Based on the experience of CompCert [Ler09a], the forward simulation property is generally easier to prove. Nevertheless, this property is not sufficient to guarantee the semantic preservation of the compiler. But in the case where the compiled program C is deterministic ($C \Downarrow B_1 \wedge C \Downarrow B_2 \Rightarrow B_1 = B_2$), the forward simulation proof is sufficient to prove the backward simulation property and thus that the semantic is preserved. Indeed, let us suppose that we have $C \Downarrow B$, and we want to prove that we have $S \Downarrow B$. As the semantics is defined for S, it exists B' such that $S \Downarrow B'$. Using the forward simulation property, we thus have $C \Downarrow B'$ and as C is deterministic, $B = B'$ which concludes the proof.

5.1.2 Simulation diagrams for transition semantics

Semantics for synchronous languages like Lustre or for the flight plans, which is defined in Chapter 7, are generally described by state transitions, starting from an initial state. We consider two languages L_1 and L_2 with P_1 and P_2 that are possible programs in their respective languages and are deterministic. We suppose that P_2 was produced by the compiler *Comp* from the program P_1. A transition semantics for both programs is defined by the two following properties:

initial_state $P_1\ S_1$ specifies that the state S_1 is an initial environment for the program P_1,

$G_1 \vdash S_1 \xrightarrow{t} S_1'$ specifies that the execution from the state S_1 to the state S_1' that produces the trace t (a history of possible external events that occurred during the execution). The execution corresponds to one semantic step of the program P_1 represented by its global environment G_1.

The semantics for the program P_2 is defined similarly. Assuming that L_2 is deterministic, the semantics is preserved if we can prove a forward simulation property. Initially, we must construct the relation $S_1 \backsim S_2$ stating that both states of the programs P_1 and P_2 are *matching*, i.e. they represent the same state of the system. The forward simulation property for transition semantics can thus be divided into two sub-properties: the initial states should match and the states should match after an execution step.

The first sub-property, presented in Definition 5.4, states that for any initial state of P_1 there exists a matching state in P_2 that is also an initial state.

Definition 5.4 - Matching initial state.
$\forall\ S_1,$ *initial_state* $P_1\ S_1$
$\Rightarrow \exists\ S_2,$ *initial_state* $P_2\ S_2 \wedge S_1 \backsim S_2$

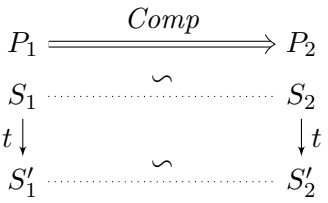

Figure 5.1: Bisimulation diagrams

The second sub-property states that the environments resulting from a step execution for P_1 and P_2 with two matching environments should be still matching. This property is depicted in diagram in Figure 5.1.

Notice that this diagram shows a bisimulation property. In our case, the property that guarantees the semantic preservation corresponds to a lock-step simulation property, shown in Definition 5.5, as the programs are deterministic.

Definition 5.5 - Lock-step simulation.

$$\begin{aligned}&\forall\ S_1\ S'_1\ t,\ G_1 \vdash S_1 \xrightarrow{t} S'_1\\ &\quad\Rightarrow \forall\ S_2,\ S_1 \backsim S_2\\ &\quad\Rightarrow \exists\ S'_2,\ G_2 \vdash S_2 \xrightarrow{t} S'_2 \wedge S'_1 \backsim S'_2\end{aligned}$$

The lock-step property states that for any execution of P_1 from the state S_1 to S'_1, and any matching state S_1 and S_2, then there is a possible execution in P_2 from S_2 to S'_2 such that S'_1 and S'_2 are matching.

An important benefit of this approach is that verified compilers can be composed. Indeed, many compilers such as CompCert are divided into several passes that communicate through intermediate languages. For instance, let us build a 2-pass verified compiler from two verified compilers $Comp_1$ and $Comp_2$ that have respectively for input languages L_1 and L_2 and produces respectively L_2 and L_3. We thus can define the compiler $Comp$ that takes the language L_1 as input and generate L_3 as follow:

Definition 5.6 - 2-Pass compiler.

$$\begin{aligned}&Comp\ P_1 ::= \texttt{match}\ Comp_1\ P_1\ \texttt{with}\\ &\qquad\qquad \mid \texttt{ERROR}\ \Rightarrow \texttt{ERROR}\\ &\qquad\qquad \mid \texttt{OK}(P_2) \Rightarrow Comp_2\ P_2\\ &\qquad\qquad \texttt{end}\end{aligned}$$

Having established the proof that the semantics is preserved by $Comp_1$ and $Comp_2$ we can easily build the proof that $Comp$ preserves the semantics between L_1 and L_3. This composition of property allows us to split the proof of the generator into several sub-proofs of the different passes that are generally easier to prove.

The approach used for the verification of the flight plan generator, presented in Chapter 10, is adapted from the approach presented in this section for the need of the project but retains the same general principles such as the definition of a matching relation, using the lock-step simulation property and the composition of passes.

5.2 Introduction to Clight Semantics

Clight is an intermediate language used in the CompCert C compiler. Clight is a subset of the C programming language that supports lots of C features like pointers arithmetic, `switch` statements, loops, or even `goto` statements. In this section, we present a simplified version of Clight syntax and its semantics, ignoring all details concerning types in order to simplify the notations and focus on key elements of the Clight semantics. The complete definition of Clight can be found in the Coq source code of CompCert.

In the following, we use a notation for records similar to the one used in Coq:

Notation 5.7 - Record.

Notation used in the manuscript	Corresponding Coq notation

record_type ::= {|
  field1: *type*1,
  field2: *type*2
|}
	Record record_type := {
	  field1: type1;
	  field2: type2;
	}.

If the element r is an element of *record_type*, we access to its `field1` value using the notation "r.`field1`". We denote by "$r\{\texttt{field2} := v\}$" a record similar to r but with value v for field `field2`. We also use the following notation to instantiate a record, i.e. the definition of the variable *record_val* whose type is *record_type*:

Notation 5.8 - Instantiation of record.

record_val ::= {\|
  `field1` := *val*1, `field2` := *val*2
\|}

Note that in the following, we use the record to define logical proposition as in Coq. If consider the same example presented above with both *type*1 and *type*2 specifying a *Prop*, then the definition in Notation 5.7 is equivalent to *record_type* ::= *type*1 ∧ *type*2.

5.2.1 Clight Syntax

Definition 5.9 introduces Clight expressions. We use the same notations as in the Coq source code but we provide their corresponding C expressions. We do not present type information, and typecast, `sizeof` and `alignof` operators are not represented here but they are also provided.

Definition 5.9 - (expr, cfrontend/Clight.v:49).

expr ::= **Econst** (value: *value*)	value
| **Evar** (var: *ident*)	var
| **Etempvar** (var: *ident*)	var
| **Ederef** (e: *expr*)	* e
| **Eaddrof** (e: *expr*)	& e
| **Eunop** (◇: unary_operation) (e: *expr*)	◇ e
| **Ebinop** (⊕: binary_operation) (e1: *expr*) (e2: *expr*)	e1 ⊕ e2
| **Efield** (expr: *expr*) (field: *ident*)	e->field

A Clight expression can be a constant value. In the Coq source code, there is a **Econst** constructor specific for every type of value supported (32/64 bits, float, etc.). In this simplified version, we gather all supported values through the *value* set, cf. Definition 5.10. An expression can also be a variable that can either be stored in memory or as a temporary variable. Temporary variables are a different class of local variables that are stored in registers and thus cannot be addressed. There are then operations specific to pointer arithmetic such as dereferencing a pointer or getting the address of a variable. Finally, we have unary and binary operations on expressions and access to the field of a structure.

Definition 5.10 - (val, common/Values.v:37).

value ::= `int` (`value`: *int*) | `undef`
| `float` (`value`: *float*) | `ptr` (`value`: *location*)

Remark

Variables in Clight are referenced using *ident* that are positive numbers, like all identifiers used in Clight (variables, function names or function parameters). Section 9.3 presents how these number values are converted into readable strings if we want to print Clight code as C code.

Clight expressions thus correspond to the pure subset of C expressions as they do not contain assignment operators (`=` or `++`) nor function calls. Indeed, both assignment operators or function calls are considered only as statements in Clight unlike in C. This avoids having side effects in expressions and forces to have a specific evaluation order for assignments (which is an undefined behaviour in C). For similar reason, there is no boolean `and` (`&&`). Instead, an `if-then-else` statement with a temporary variable should be used.

Definition 5.11 presents Clight statements and their equivalent C statements. A Clight *statement* can be the no-operation statement, assignment to a variable[1] or a temporary variable, a function call, a sequence, a conditional, an infinite loop with the associated `break` and `continue` statements, a `switch` statement, and a `goto` statement with the possibility to set labels. The different traditional loops in C (`for`, `while`, `do-while`) can be expressed with "**Sloop** `s1 s2`" statements. This statement repeats the execution of `s1` then `s2` until a `break` statement is executed. Example 5.2 shows the Clight code generated for a C `for` loop. The *labeled_statement* either corresponds to a sequence of C `case` if the **LScons** contains an integer value or to `default`.

The function name in a **Scall** statement is defined through an expression that must be a variable or a function pointer. In the semantics, presented in Section 5.2.2, we will explain how we can access the function definition from the variable name. In Clight there are two types of functions, internal functions and external functions, that are described by *fundef* defined as follows.

Definition 5.12 - (fundef, cfrontend/Clight.v:150).

fundef ::= **Internal** *function*
| **External** *external_function*

[1] The **Sassign** statement has an expression for the left value of the assignment as it can be a dereferenced pointer or a field of a structure.

Definition 5.11 - (statement, cfrontend/Clight.v:96).

statement ::= **Sskip**	skip
| **Sassign** (var: *expr*) (expr2: *expr*)	var = expr1
| **Sset** (tmp: *ident*) (expr: *expr*)	tmp = expr1
| **Scall** (res: option *ident*) (fun: *expr*) (args: list *expr*)	[res =] fun(args)
| **Sbuiltin** (res: option *ident*) (fun: *external_function*) (args: list *expr*)	[res =] fun(args)
| **Ssequence** (s1: *statement*) (s2: *statement*)	s1; s2
| **Sifthenelse** (cond: *expr*) (s1: *statement*) (s2: *statement*)	if (cond) s1 else s2
| **Sloop** (s1: *statement*) (s2: *statement*)	loop{s1; s2}
| **Sbreak**	break
| **Scontinue**	continue
| **Sreturn** (value: option *expr*)	return [value]
| **Sswitch** (value: *expr*) (lbs: *labeled_statement*)	switch (value) lbs
| **Slabel** (label: *label*) (s: *statement*)	label: s
| **Sgoto** (label: *label*)	goto label
labeled_statement ::= **LSnil**	
| **LScons** (val: option $\mathbb{Z}$) (s: *statement*) (sl: *labeled_statement*)	

Example of C Code:

```
for(int i = 0; i < 10; i++) {
    a = fun();
}
```

Equivalent Clight code:

```
Ssequence (
  Sassign (Evar(i), Econst(0)),
  Sloop (
    Ssequence (
      Sifthenelse( Ebinop(<, Evar(i), Econst(10)), Sskip, Sbreak),
      Scall(Evar(a), Evar(fun), [])
    ),
    Sassign (Evar(i), Ebinop(+, Evar(i), Econst(1)))
  )
)
```

Figure 5.2: Example of a C `for` loop converted into Clight.

The internal functions are functions whose implementation is known unlike external functions. The external functions can be built-in functions, called with the statement **Sbuiltin**, or functions defined in external libraries. Both internal functions or external library functions are called using the **Scall** statement. We show in Section 5.2.2 how the **Scall** statement manages to distinguish whether the function is external or internal. Note that for a C file an internal function corresponds to a function defined in the file (this is

similar for Clight). External functions are functions whose signature is provided by the `include` macro. The compilation of this file thus produces the assembly code of the internal functions and the code of the external functions is added later during the linking phase.

The Clight definition of external functions is defined by the `external_function` inductive type that we will not detail in this document. It specifies the type of external functions (built-in, external functions, external memory access, etc.) and their signatures. On the other hand, internal functions are described by the *function* record.

Definition 5.13 - (`function`, `cfrontend/Clight.v:135`).

function ::= {|
  `params`: list *ident*, `vars`: list *ident*,
  `body`: *statement*, `temps`: list *ident*
|}

A *function* is composed of a list of parameters, a list of local variables that must be stored in the memory or as temporary variables and its body is defined as a statement. The name of the functions and their definitions are associated in a global definition, i.e. a *gdef* element, that are all stored in the global environment.

Definition 5.14 - (`globdef`, `common/AST.v:299`).

globdef ::= **Gvar** *globvar*
| **Gfun** *fundef*
gdef ::= *ident* × *globdef*

In addition to function definition, a global definition can also be a global variable defined with **Gvar**. Finally, with all these elements, we can define a program.

Definition 5.15 - (`program`, `cfrontend/Ctypes.v:1518`).

prog ::= {|
  `main`: *ident*, `defs`: list *gdef*
|}

A Clight program is composed of the name of the `main` entry function and a list of global definitions that corresponds to all elements defined and known: the global variables, the functions defined locally and the known functions that are defined externally.

5.2.2 Small-step semantics of Clight

We now present the semantics of Clight. CompCert provides two different semantics, a big-step operational semantics [BL09] and a small-step continuation-based operational semantics [AB07] that are respectively defined in `ClightBigstep.v` and `Clight.v` files. The big-step semantics is easier to use but it does not support `goto` statements. As we use Clight to generate code containing `goto` statements, we therefore focus on the small-step semantics of Clight. The inference rules presented here are an extension of [Bru20] that presents a subset of Clight semantics. We use the notation from [Ler09a] that describes the small-step semantics of Cminor, the intermediate language used in CompCert after Clight.

Clight small-step semantics rules for the evaluation of expression or statement are parameterised by a global environment G, a local environment E, a temporary environment L and a memory state M.

A global environment G is an element of *genv* that maps global variables and function names to their locations and function locations to their definitions. The following functions concerns global environments.

Definition 5.16 - Operations over global environments, defined in `Globalenvs.v`.

find_def $G\ l = \texttt{Some}\ d$	Returns the global definition d referenced by the location l in G,
symbol $G\ id = \texttt{Some}\ l$	Returns the location l referenced by the name id in G,
globalenv $P = G$	Returns the global environment G initialised with the *gdef* of P,
init_mem $P = M$	Returns the initial memory M for the program P

Similarly to the global environment, a local environment E of *env* maps local variables to their location in memory. The temporary environments *tmp_env* are direct maps from temporary variables to their values. Two functions manipulating temporary environments are defined.

Definition 5.17 - Operations over temporary environments defined in `Clight.v`.

$$create\_undef\_temps\ \overrightarrow{vars} = L$$

This function initialises a temporary environment with all the variables $\overrightarrow{vars}$ set to the `undef` value.

$$bind\_parameters\_tmp\ \overrightarrow{vars}\ \overrightarrow{values}\ L = L'$$

This function updates the temporary environment L for all the variables of $\overrightarrow{vars}$ with the values contained in $\overrightarrow{values}$. This function returns `None` if $\overrightarrow{vars}$ and $\overrightarrow{values}$ do not contain the same number of elements. To ease readability, we denote by $L\{id \leftarrow val\}$ the environment resulting from updating the variable id in environment L with the value val.

The set *mem* describes the memory states of the program. The memory model used in Clight is presented in [LB08]. We use here a simplified version of the memory model where a memory state M stores at every location one *value*, i.e. one memory location can either store 8 bits or 64 bits value[2]. Four operations over memory states are defined as follows.

Definition 5.18 - Operations over memory, defined in `Memory.v`.

store $M\ l\ v = \texttt{Some}\ M'$	Produces the updated memory state M' containing v at position l,
load $M\ l = \texttt{Some}\ v$	Returns the value contained at position l in M,
alloc $M\ n = \texttt{Some}(M',\ l)$	Returns a new memory state where at position l, n bytes have been allocated,
free $M\ E = \texttt{Some}\ M'$	Returns a new memory state where the variables of E have been freed.

In the Coq source code, the evaluation of expressions is defined by the mutually inductive predicates *eval_expr* and *eval_lvalue*. These predicates respectively evaluate an expression in r-value or l-value position.

[2]The memory model used in CompCert is similar to real memory where every location can store only 8 bits values. The operations over memory states have thus extra parameters to specify the size of the value that we want to access.

Signature 5.19 - (eval_expr, cfrontend/Clight.v:368).

$$eval\_expr\colon genv \to env \to tmp\_env \to mem \to expr \to value \to Prop$$

The property "*eval_expr G E L M e v*", noted $G, E, L \vdash e, M \Rightarrow v$, states that the evaluation of the expression e in the current environment (G, E, L and M) produces the value v.

We extend *eval_expr* with the notation $G, E, L \vdash \vec{e}, M \overset{list}{\Longrightarrow} \vec{v}$ that evaluates a list of expressions and thus produces a list of values.

Signature 5.20 - (eval_lvalue, cfrontend/Clight.v:410).

$$eval\_lvalue\colon genv \to env \to tmp\_env \to mem \to expr \to location \to Prop$$

The property "*eval_lvalue G E L M e l*", noted $G, E, L \vdash e, M \Leftarrow l$, states that the evaluation of the expression e as a left value produces the memory location l.

The evaluation of expressions does not modify the memory state as all expressions are pure in Clight. In this thesis, we will not present the inference rules for the evaluation of expressions in order to focus on the evaluation of statements that do modify the memory. The semantics of expressions can be found directly in the Coq source code but also in several articles, cf. [Bru20, BL09, Ler09a].

Signature 5.21 - (step, cfrontend/Clight.v:559).

$$step\colon genv \to state \to trace \to state \to Prop$$

The property "*step G S t S′*", noted $G \vdash S \xrightarrow{t} S'$, states that the small-step execution of the Clight semantics from a state S produces the trace t and terminates in the state S'.

A `trace` is a list of events that occur during the execution of a Clight program. An `event` can either be an external call or an access to external memory. These events are thus generated when external calls are executed. The execution of a step may also not produce events, the generated trace will then be an empty trace denoted by ϵ.

Definition 5.22 - (state, cfrontend/Clight.v:490).

state ::=	$\mathcal{S}\ (F,\ s,\ k,\ E,\ L,\ M)$	Regular state
	$\mid \mathcal{C}\ (F_d,\ \overrightarrow{v_{args}},\ k,\ M)$	Call state
	$\mid \mathcal{R}\ (v,\ k,\ M)$	Return state

The *step* predicate of the small-step semantics describes the evolution of the program state. Definition 5.22 presents the different forms that a state can take. The regular state $\mathcal{S}$ holds the information that we are currently executing the function F and we have to execute the statement s in the current context composed of the environment E, the temporary environment L and the current memory state M. The call state $\mathcal{C}$ holds the information needed to execute the function whose definition is F_d with the parameters values $\overrightarrow{v_{args}}$ in the memory state M. Finally, the return state $\mathcal{R}$ carries information that the function has finished executing in the memory state M and the function returned the value v. All the different states carry the element k which is a continuation containing information about the local context (inside the block of a switch, a loop or a function) and the remaining statement to execute.

Definition 5.23 - (cont, cfrontend/Clight.v:458).

$cont ::=$ **Kstop**: $cont$
| **Kseq**: $statement \to cont \to cont$
| **Kloop**$_1$: $statement \to statement \to cont \to cont$
| **Kloop**$_2$: $statement \to statement \to cont \to cont$
| **Kswitch**: $cont \to cont$
| **Kcall**: option $ident \to function \to env \to tmp\_env \to cont \to cont$

The continuation **Kstop** is the initial continuation and the others are inductive continuations: **Kseq** carries the statement that will be executed next, **Kloop**$_1$ and **Kloop**$_2$ carry the statement of the loop being executed and they respectively describe which one of the two statements is being executed (cf. Definition 5.11), **Kswitch** states that the execution is inside a switch statement and **Kcall** states that we are inside a function and carries the information of the calling function. Two functions that manipulate continuation are defined: `is_call_cont` states whether a continuation is a *call continuation*, i.e. a **Kstop** or a **Kcall**; `call_cont` removes the local context of the continuation, i.e. unfolds the continuation until reaching a *call continuation*.

The transition rules of the small-step semantics corresponding to the `step` predicate are described in figures 5.3 to 5.7.

$$\frac{}{G \vdash \mathcal{S}(F,\ \textbf{Ssequence}(s_1,\ s_2),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s_1,\ \textbf{Kseq}(s_2,\ k),\ E,\ L,\ M)}\ (\text{C-1})$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \textbf{Sskip},\ \textbf{Kseq}(s,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s,\ k,\ E,\ L,\ M)}\ (\text{C-2})$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \textbf{Scontinue},\ \textbf{Kseq}(s,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \textbf{Scontinue},\ k,\ E,\ L,\ M)}\ (\text{C-3})$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \textbf{Sbreak},\ \textbf{Kseq}(s,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \textbf{Sbreak},\ k,\ E,\ L,\ M)}\ (\text{C-4})$$

$$\frac{G, E, L \vdash\ a_1, M \Leftarrow l \qquad G, E, L \vdash\ a_2, M \Rightarrow v \qquad store\ M\ l\ v = \texttt{Some}\ M'}{G \vdash \mathcal{S}(F,\ \textbf{Sassign}(a_1,\ a_2),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \textbf{Sskip},\ k,\ E,\ L,\ M')}\ (\text{C-5})$$

$$\frac{G, E, L \vdash\ a, M \Rightarrow v}{G \vdash \mathcal{S}(F,\ \textbf{Sset}(id,\ a),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \textbf{Sskip},\ k,\ E,\ L\{id\ \leftarrow v\},\ M)}\ (\text{C-6})$$

Figure 5.3: Inference rules for sequence and assignment

Figure 5.3 presents the inference rules for sequence and assignment. The evaluation of a sequence of statements starts by computing the first statement and the second is stored in the continuation (rule C-1). The statement stored in the continuation is executed when there is no statement to execute locally, i.e. the current statement is **Sskip** (rules C-2). The `continue` and `break` statements, described by rules C-3 and C-4 ignore all statements stored in the continuation until they reach the end of a block, i.e. the end of a loop or a

`switch` statement. The assignment, described by the rules C-5 and C-6, starts by evaluating the value of the right expression, stores the data and then terminates the execution by setting **Sskip** as the statement in the produced state. The rules differ on how the data are stored: the **Sassign** statement computes the location referenced by the left value and then updates the memory state while the **Sset** simply updates the temporary environment.

$$\frac{G, E, L \vdash\ c, M \Rightarrow v \qquad \mathit{istrue}\ v}{G \vdash \mathcal{S}(F,\ \mathbf{Sifthenelse}(c,\ s_1,\ s_2),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s_1,\ k,\ E,\ L,\ M)} \text{(C-7)}$$

$$\frac{G, E, L \vdash\ c, M \Rightarrow v \qquad \mathit{isfalse}\ v}{G \vdash \mathcal{S}(F,\ \mathbf{Sifthenelse}(c,\ s_1,\ s_2),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s_2,\ k,\ E,\ L,\ M)} \text{(C-8)}$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \mathbf{Sloop}(s_1,\ s_2),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s_1,\ \mathbf{Kloop}_1(s_1,\ s_2,\ k),\ E,\ L,\ M)} \text{(C-9)}$$

$$\frac{s = \mathbf{Sskip} \vee s = \mathbf{Scontinue}}{G \vdash \mathcal{S}(F,\ s,\ \mathbf{Kloop}_1(s_1,\ s_2,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s_2,\ \mathbf{Kloop}_2(s_1,\ s_2,\ k),\ E,\ L,\ M)} \text{(C-10)}$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \mathbf{Sskip},\ \mathbf{Kloop}_2(s_1,\ s_2,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Sloop}(s_1,\ s_2),\ k,\ E,\ L,\ M)} \text{(C-11)}$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \mathbf{Sbreak},\ \mathbf{Kloop}_1(s_1,\ s_2,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ M)} \text{(C-12)}$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \mathbf{Sbreak},\ \mathbf{Kloop}_2(s_1,\ s_2,\ k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ M)} \text{(C-13)}$$

Figure 5.4: Inference rules for conditional and loop statements

Figure 5.4 introduces the rules about the conditional and loop statements. We use the predicate *istrue* and *isfalse* stating whether the value corresponds respectively to the boolean value `true` and `false`. The conditional statement, described by the rules C-7 and C-8, updates the statement to execute depending on the result of the evaluation of the condition. The evaluation of the loop statement starts by executing the first statement s_1 and then updating the continuation with $\mathbf{Kloop}_1$ (rule C-9). The second loop statement s_2 is executed and the continuation is updated to $\mathbf{Kloop}_2$ when the s_1 has finished running or a `continue` statement has been executed (rule C-10). The loop is restarted when s_2 has finished to be executed (rule C-11) or is exited when a `break` is encountered (rules C-12 and C-13).

The inference rules about the `switch` and `goto` statements are shown in Figure 5.5. The function *lbl_stmt* [3] takes an integer value v and a *labeled_statement* *sl*. The function returns the statement corresponding to the matching case in a `switch` statement. The statement returned is the first *labeled_statement* matching the value v (or the default case if the value

[3]The function "*lbl_stmt* v *sl*" is equivalent to `seq_of_labeled_statement (select_switch` v *sl*`)` in the Coq source code.

$$\frac{G, E, L \vdash\ e, M \Rightarrow v \qquad \mathit{lbl\_stmt}\ v\ \mathit{sl} = s}{G \vdash \mathcal{S}(F,\ \mathbf{Sswitch}(e,\ \mathit{sl}),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s,\ \mathbf{Kswitch}(k),\ E,\ L,\ M)} \text{(C-14)}$$

$$\frac{s = \mathbf{Sskip} \vee s = \mathbf{Sbreak}}{G \vdash \mathcal{S}(F,\ s,\ \mathbf{Kswitch}(k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ M)} \text{(C-15)}$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \mathbf{Scontinue},\ \mathbf{Kswitch}(k),\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Scontinue},\ k,\ E,\ L,\ M)} \text{(C-16)}$$

$$\frac{}{G \vdash \mathcal{S}(F,\ \mathbf{Slabel}(\mathit{lbl},\ s),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s,\ k,\ E,\ L,\ M)} \text{(C-17)}$$

$$\frac{\mathit{findlabel}\ \mathit{lbl}\ F.\mathsf{body}\ (\mathit{callcont}\ k) = \mathsf{Some}\ (s', k')}{G \vdash \mathcal{S}(F,\ \mathbf{Sgoto}(\mathit{lbl}),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ s',\ k',\ E,\ L,\ M)} \text{(C-18)}$$

Figure 5.5: Inference rules for switch and goto

is not matched) in *sl*. Every branch following the matching `case` are concatenated and linearized, i.e. **LScons** are replaced by **Ssequence**. The execution of the `switch` statement thus consists in evaluating the given expression and then the returned state is composed of the statement produced by *labeled_statement* and the continuation updated with **Kswitch** (rule C-14). The `switch` statement is exited either if a `break` statement is executed or if the execution reaches the end of the block. (rule C-15). The `continue` statement also exits the `switch` block but stays in the produced state as it searches its corresponding loop in the continuation (rule C-16).

The execution of a **Slabel** statement, described in rule C-17 simply consists in ignoring the label and execute its statement. However, the evaluation of the `goto` statement, presented in rule C-18, requires a call to the *findlabel* function which produces a new statement to execute and a new continuation. The function "`find_label` *lbl s k*" produces the couple (s', k') where s' is the sub-statement in s labeled by *lbl* and k' is the continuation k updated with the context around s'. In the context of the `goto` statement, the *findlabel* function is used to find in the body of the current function the statement corresponding to the label. The continuation k is the initial continuation when entering the current function, computed with the function *callcont*. The continuation produced is thus the continuation which would have been produced when the execution, through the Clight semantics, reaches the statement s'.

Figure 5.6 presents the inference rules concerning function calls. The evaluation of a **Scall**(*optid*, *e*, $\vec{v}$), showed in rule C-19, starts by evaluating the expression e which produces a reference to the called function and all the expressions of $\vec{v}$ that produce the values of the function parameters. The function definition should then be found in the global environment using the function reference. Finally, *step* produces a **Kcall** state containing the information of the function called and where the information of the current call state is saved in the continuation.

$$\frac{G, E, L \vdash\ e, M \Rightarrow\ vf \qquad G, E, L \vdash\ \vec{v}, M \overset{list}{\Longrightarrow} \overrightarrow{v_{args}} \qquad find\_def\ G\ vf = \mathsf{Some}\ (\mathbf{Gfun}\ F_d)}{G \vdash \mathcal{S}(F,\ \mathbf{Scall}(optid,\ e,\ \vec{v}),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{C}(F_d,\ \overrightarrow{v_{args}},\ \mathbf{Kcall}(optid,\ F,\ E,\ L,\ k),\ M)} \text{(C-19)}$$

$$\frac{G, E, L \vdash\ \vec{v}, M \overset{list}{\Longrightarrow} \overrightarrow{v_{args}} \qquad external\_call\ ef\ G\ \overrightarrow{v_{args}}\ M\ t\ v_{res}\ M' \qquad optid = \mathsf{Some}\ id}{G \vdash \mathcal{S}(F,\ \mathbf{Sbuiltin}(optid,\ ef,\ \vec{v}),\ k,\ E,\ L,\ M) \xrightarrow{t} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L\{id \leftarrow v_{res}\},\ M')} \text{(C-20)}$$

$$\frac{G, E, L \vdash\ \vec{v}, M \overset{list}{\Longrightarrow} \overrightarrow{v_{args}} \qquad external\_call\ F_e\ G\ \overrightarrow{v_{args}}\ M\ t\ v_{res}\ M' \qquad optid = \mathsf{None}}{G \vdash \mathcal{S}(F,\ \mathbf{Sbuiltin}(optid,\ F_e,\ \vec{v}),\ k,\ E,\ L,\ M) \xrightarrow{t} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ M')} \text{(C-21)}$$

$$\frac{function\_entry\ F\ \overrightarrow{v_{args}}\ M\ E\ L\ M'}{G \vdash \mathcal{C}(\mathbf{Internal}(F),\ \overrightarrow{v_{args}},\ k,\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ F.\mathsf{body},\ k,\ E,\ L,\ M')} \text{(C-22)}$$

$$\frac{external\_call\ F_e\ G,\ \overrightarrow{v_{args}}\ M\ t\ v_{res}\ M'}{G \vdash \mathcal{C}(\mathbf{External}(F_e),\ \overrightarrow{v_{args}},\ k,\ M) \xrightarrow{t} \mathcal{R}(v_{res},\ k,\ M')} \text{(C-23)}$$

Figure 5.6: Inference rules for calls

The execution of the **Sbuiltin** starts by evaluating the value of the parameters and executes the function call through the predicate `external_call`. The predicate "$external\_call\ F_e\ G\ \overrightarrow{v_{args}}\ M\ t\ v_{res}\ M$"' states that the execution of the function F_e with the parameters $\overrightarrow{v_{args}}$ in the current context $(G,\ M)$ terminates in the memory state M', returns the value v_{res} and produces the trace t containing the information about this external call. The returned value is stored in the temporary environment depending on whether the **Sbuiltin** state has an assignment (rules C-20 and C-21).

The transition from a call state depends on whether the function definition corresponds to an internal or external call. The internal call, described by the rule C-22 is defined by the *function_entry* predicate. This predicate[4] initialises the context with the local variables and the values of the parameters. This context is then used in the regular state produced from which the body statement of the called function is executed. The external calls, presented in rule C-23, are executed similarly to the **Sbuiltin** statement using the *external_call* statement.

Finally, Figure 5.7 introduces the rules concerning the `return` statement. The execution of an explicit return statement (rule C-24) or a **Sskip** statement when the continuation is a *call continuation* (rule C-26), terminates the execution of the current function. The memory used by the local variables should thus be freed. However, if the `return` statement contains an expression (rule C-25), it should be evaluated before the memory is freed. The state returned when the execution of a function terminates is a return state that contains the value returned, the continuation unfolded using *callcont* and the memory state freed. The

[4]The *function_entry* predicate is formally presented later in definition 5.24

$$\frac{free\ M, E = \mathsf{Some}\ M'}{G \vdash \mathcal{S}(F,\ \mathbf{Sreturn}(\mathsf{None}),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{R}(\mathsf{undef},\ (callcont\ k),\ M')} \text{(C-24)}$$

$$\frac{G, E, L \vdash\ e, M \Rightarrow v \qquad free\ M\ E = \mathsf{Some}\ M'}{G \vdash \mathcal{S}(F,\ \mathbf{Sreturn}(\mathsf{Some}\ e),\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{R}(v,\ (callcont\ k),\ M')} \text{(C-25)}$$

$$\frac{iscallcont\ k \qquad free\ M\ E = \mathsf{Some}\ M'}{G \vdash \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ M) \xrightarrow{\epsilon} \mathcal{R}(\mathsf{undef},\ k,\ M')} \text{(C-26)}$$

$$\frac{optid = \mathsf{Some}\ id}{G \vdash \mathcal{R}(v,\ \mathbf{Kcall}(optid,\ F,\ E,\ L,\ k),\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L\{id\ \leftarrow v\},\ M)} \text{(C-27)}$$

$$\frac{optid = \mathsf{None}}{G \vdash \mathcal{R}(v,\ \mathbf{Kcall}(optid,\ F,\ E,\ L,\ k),\ M) \xrightarrow{\epsilon} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ M)} \text{(C-28)}$$

Figure 5.7: Inference rules for the return statement

execution of a transition from a return state is only defined if the continuation is a **Kcall** containing the context of the calling function. The transition thus restores the context and updates the temporary environment with the returned value if the call contains an assignment (rules C-27 and C-28).

The predicate *function_entry* used in the semantics for the **Scall** statement (see Figure 5.6), initialises the context of the function called. In particular, among other things, it describes how the parameter values are stored. CompCert provides two variants for the Clight semantics using different strategies to store these parameter values. The first one considers the parameters as local variables (`function_entry1`, `cfrontend/Clight.v:703`). The second one uses temporary variables to store the values of the parameters (`function_entry2`, `cfrontend/Clight.v:715`). In the following, we only consider the second semantics as it is easier to manipulate temporary environments than to store values in the memory. The *function_entry* predicate for the second semantics is thus defined in the Definition 5.24.

Definition 5.24 - (function_entry2, `cfrontend/Clight.v:715`).

$$\frac{\begin{array}{c} norepet\ F.\mathsf{vars} \\ norepet\ F.\mathsf{params} \qquad F.\mathsf{params} \cap F.\mathsf{temps} = \emptyset \qquad alloc\_variable\ G\ M\ F.\mathsf{vars}\ E\ M' \\ bind\_parameters\_tmp\ F.\mathsf{params}\ \overrightarrow{v_{args}}\ (create\_undef\_temps\ F.\mathsf{temps}) = \mathsf{Some}\ L \end{array}}{function\_entry\ F\ \overrightarrow{v_{args}}\ M\ E\ L\ M'} \text{(C-29)}$$

The predicate *function_entry* is satisfied if several criteria are met. First, the names of the variables and parameters should be unique. This property is described using the *norepet* predicate. Secondly, as the values of the parameters are stored in the temporary environment, there should be no name conflict between $F.\mathsf{params}$ and $F.\mathsf{temps}$. The memory is then initialised through `alloc_variables` predicate: a memory location is allocated for every variable of $F.\mathsf{vars}$ and the local environment E is built to map variable names to

their respective locations. Finally, the temporary environment is instantiated, all temporary variables are associated with the `undef` value and the parameters are set to the input values contained in $\overrightarrow{v_{args}}$.

5.2.3 Execution of a Clight program

In the previous section, we defined the predicate *step* describing transition rules of the small-step semantics of Clight. We now want to define the semantics for the execution of a Clight program *prog*. The CompCert compiler is composed of several intermediate languages such as Clight or Cminor. In order to have common definitions to specify the semantic preservation property between all these languages, they must implement the same signature shown in Definition 5.25.

The *semantics* record of CompCert intermediate languages is defined through several elements that must be provided: the type used to represent the execution state of the program (`state`), the type that stores all information about the global environment (`genvtype`), the predicate representing an execution step between two states that may produce a trace (`step`), a predicate representing an initial state for the program (`initial_state`), a predicate stating whether a state and the value returned correspond to a final state (`final_state`), and finally type of global environments used to execute a program (`globalenv`).

Definition 5.25 - (semantics, common/Smallstep.v:528).

semantics ::= {|
  `state:` *Type*,
  `genvtype:` *Type*,
  `step:` `genvtype` → `state` → *trace* → `state` → *Prop*,
  `initial_state:` `state` → *Prop*,
  `final_state:` `state` → *int* → *Prop*,
  `globalenv:` `genvtype`
|}

In Section 5.2.2, almost all elements to define the *semantics* have been introduced except the predicates describing initial and final states. Definition 5.26 presents the inference rule for the definition of the predicate "*initial_state* P S" stating that S is an initial state for the Clight program P. Definition 5.27 shows the inference rule for the predicate "*final_state* S r" stating that S is a final state of a Clight program that returns the value r.

Definition 5.26 - (initial_state, cfrontend/Clight.v:684).

$$\frac{\begin{array}{cc} \mathit{globalenv}\ P = G & \mathit{init\_mem}\ P = \texttt{Some}\ M_0 \\ \mathit{symbol}\ G\ P.\texttt{main} = \texttt{Some}\ l & \mathit{find\_def}\ G\ l = \texttt{Some}\ (\textbf{Gfun}\ F_d) \end{array}}{\mathit{initial\_state}\ P\ \mathcal{C}(F_d\ \texttt{[]}\ \textbf{Kstop}\ M_0)}\ \text{(C-30)}$$

An initial state for a program P, described by the rule C-30, is a call state that will execute the `main` function of P. The initial continuation should be a **`Kstop`**. The global environment and the initial memory state should have been initialised using the program P.

Definition 5.27 - (final_state, cfrontend/Clight.v:695).

$$\frac{}{\mathit{final\_state}\ \mathcal{R}(\texttt{int}(r)\ \textbf{Kstop}\ M)\ r}\ \text{(C-31)}$$

The final state for a program returning the value r, defined by the rule C-31, should be a return state with the value r and an empty continuation. Finally, using all previous definitions, we can define the *clight_semantics* function that produces the small-step *semantics* for a specific Clight program P.

Definition 5.28 - (semantics2, cfrontend/Clight.v:732).

clight_semantics: *prog* → *semantics*
clight_semantics P ::= {|

state	:=	*state*,	step	:=	*step*,
genvtype	:=	*genv*,	initial_state	:=	*initial_state* P,
globalenv	:=	*globalenv* P,	final_state	:=	*final_state*

|}

The *step* predicate only describes a transition for the execution of a Clight program. A complete execution is thus composed of successive step executions. We note $G \vdash S \xrightarrow{t}{}^{*} S'$ the predicate, which corresponds to a transitive closure, stating that it exists a sequence of inference rules when the Clight small-step semantics exhibits an execution starting in the state S and terminating in the state S' while producing the trace t. This predicate is defined inductively by a base case (rule C-32) and a recursive case (rule C-33). The rule C-34 presents the case where successive executions might not terminate (e.g. when executing an infinite loop). In this case, the generated trace could be infinite.

Definition 5.29 - Defined in Smallstep.v file.

$$\frac{}{G \vdash S \xrightarrow{\epsilon}{}^{*} S}\ \text{(C-32)} \qquad \frac{G \vdash S \xrightarrow{t_1} S_1 \qquad G \vdash S_1 \xrightarrow{t_2}{}^{*} S_2}{G \vdash S \xrightarrow{t_1 +\!+ t_2}{}^{*} S_2}\ \text{(C-33)}$$

$$\frac{G \vdash S \xrightarrow{t} S' \qquad G \vdash S' \xrightarrow{T}{}^{*} \infty}{G \vdash S \xrightarrow{t +\!+ T}{}^{*} \infty}\ \text{(C-34)}$$

Finally, we can define a *correct_execution* predicate describing the correct execution of a Clight program P: the execution should start from an initial environment of P and eventually reach a final state in a finite number of steps.

Definition 5.30 - Correct execution of a Clight program.

$$\frac{\mathit{globalenv}\ P = G \qquad \mathit{initial\_state}\ P\ S \qquad G \vdash S \xrightarrow{t}{}^{*} S' \qquad \mathit{final\_state}\ S'\ v}{\mathit{correct\_execution}\ P\ S\ t\ S'\ v}\ \text{(C-35)}$$

Chapter 6

Introduction to the Verified Flight Plan Generator

Contents

6.1 FPL: Flight Plan Language 138
6.2 Global architecture of the new generator 139
6.2.1 Preprocessor 140
6.2.2 Postprocessor 141

Paparazzi Flight Plan generator is a critical component of Paparazzi autopilot. Figure 6.1 shows an overview of its integration inside the build chain of Paparazzi. The XML file describing the mission that has been defined by the user is translated by the generator into C code. This C code is then compiled with the rest of the autopilot and directly embedded in the drone.

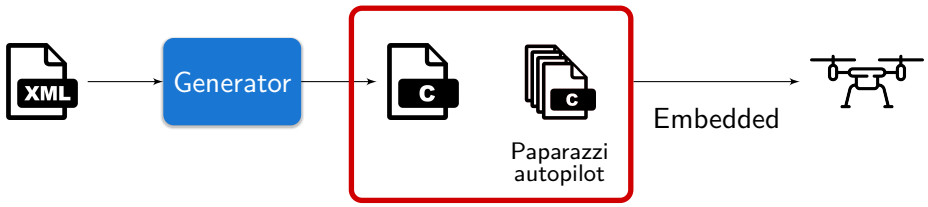

Figure 6.1: Current Generator integrated in Paparazzi

The current Paparazzi generator is implemented in OCaml but has never been formally verified. The first version of the OCaml generator was implemented in January 2005 and has been thoroughly tested over the last 18 years. The generator is now well-trusted by Paparazzi users as a major issue never occurred yet. However, this is not a formal argument for the absence of bugs in the generator and not an admissible argument if we wish for instance to qualify the tool for critical applications like autonomous flying drones operating in cities. We must have stronger guarantees that the produced code corresponds to the flight plan defined by the user without adding any extra unwanted behaviours.

We thus decided to develop a new *Verified Flight Plan Generator* (VFPG) using an approach presented in Chapter 5, similar to CompCert. This chapter gives the reader a general view of the input flight plan language supported (cf Section 6.1) and the global VFPG architecture (cf Section 6.2).

6.1 FPL: Flight Plan Language

Paparazzi uses a Domain Specific Language (DSL) to describe flight plans, with an XML concrete syntax. FPL is a conservative extension of this language that offers new features such as a protection mechanism against unwanted behaviours, either specified in the flight plan or by the drone operator. The new generator is fully compatible with the previous flight plans and so those written for the current Paparazzi generator are accepted by VFPG.

A FPL file is divided into two sections: the *flight plan header* and the *core* part. The *flight plan header* contains definitions and meta-information. It is composed of several sections: the header (arbitrary C code added in the header of the generated C file), waypoints (a list of constant points defined using GPS positions or relative coordinates), sectors defined using lists of waypoints, and local variables that can be used by the flight plan. The value of these variables can be updated by operators during a flight. The *core* is the main part of a flight plan and defines the different parts of the mission.

```
<!DOCTYPE flight_plan SYSTEM "../flight_plan.dtd">

<flight_plan alt="260" ground_alt="185" lat0="43.565624" lon0="1.475071"
             max_dist_from_home="1500" name="Thesis"
             security_height="25" qfu="1.0">
    <waypoints>
        <waypoint name="HOME" x="0.0" y="0.0"/>
    </waypoints>

    <exceptions>
        <exception cond="Battery() @LT 20" deroute="Wait GPS"
                   exec="LowBatteryMode()"/>
    </exceptions>

    <blocks>
        <block name="Wait GPS">
            <set var="time" value="GetTime()"/>
            <while cond="!GPSFixValid()">
                <call fun="InitSensors()" break="FALSE" loop="FALSE"/>
            </while>
        </block>
    </blocks>
</flight_plan>
```

Figure 6.2: Simple example of an XML flight plan

Figure 6.2 is a simple example of an XML file describing a flight plan. In this example, the *flight plan header* is only composed of `waypoints` and the *core* section contains `exceptions` and `blocks`. The `blocks` section contains `block` that are sub-parts of the mission. Every block is referenced by a unique name and is divided into several atomic statements called `stages`. In this example, three stages are used: `set`, `while` and `call`. The VFPG and

Paparazzi repositories contain several flight plan examples such as `full_example.xml` for instance[1].

Remark

In the following, when speaking of a flight plan we only refer to its *core*.

The previous FPL example is written in the XML concrete syntax. In order to prove properties on the generator in Coq, we first wrote a Gallina structure FP that is a representation of FPL getting rid of the XML structure. The next chapters detail this Gallina structure: Chapter 7 introduces the syntax and the formal semantics of FP without the new features and Chapter 8 updates FP syntax and semantics to support the new features brought by FPL.

6.2 Global architecture of the new generator

The current generator is fully written in OCaml and its architecture is composed of one pass (see Figure 6.1). The new generator architecture presented in Figure 6.3 is composed of three passes of OCaml code and two of Gallina code[2].

Figure 6.3: Global architecture of the generator.

The new flight plan generator takes as input an XML file describing an FPL instance. This file is read and a *parser* [WPGT23] produces a *flight plan parsed* (FPP) structure. This structure reflects the hierarchy of the XML file but can be easily manipulated. The parser is generated by Menhir [Men] using the `-coq` option. This option also produces a correctness proof for the parser in Coq [JPL12]. The *preprocessor* block written in OCaml performs some transformations on the FPP structure and produces the FP Gallina structure representing the flight plan (cf. Section 7.1 for more details). The transformations realised by the preprocessor are presented in Section 6.2.1.

The *Gallina generator* is the core part of the new generator and translates FP into Clight code, the Gallina structure representing C code (cf. Section 5.2). The Gallina generator, presented in details in Chapter 9, is a 3-pass compiler which performs the major transformations in the generator and is thus the part we should formally verify as detailed in Chapter 10. The Gallina generator, in addition than the FP structure, also takes a list of variables that must be defined in the code generated. Indeed, during the transformation

[1] Paparazzi provides also online documentation for the definition of flight plans: https://wiki.paparazzi-uav.org/wiki/Flight_Plans

[2] The orange block corresponds to our own Gallina code, blue correspond to our own OCaml code and the green box is OCaml code provided by CompCert.

performed by the preprocessor, new variables may be introduced. The C code generated should thus define these variables. The variables produced are detailed in Section 6.2.1 presenting the preprocessor.

The *printer* translates afterwards the produced Clight into C code using the OCaml CompCert `PrintClight.ml` module. Finally, the *post-processor*, presented in section 6.2.2, performs some minor transformations on the printed C code to obtain the final C code file to be embedded in the autopilot.

Notice that the new generator should ideally be written only in Gallina and we should verify that the semantics is preserved from the XML input file to the produced C code. Nevertheless, some actions cannot be realised in Gallina such as the manipulation of GPS coordinates and operation on files occuring during the preprocessing and postprocessing. However, we are confident that the phases written in OCaml are correct as they only perform minor transformations on the flight plan structure. Indeed, the *printer* and the preprocessor perform some direct translation from different languages and may expand some notations (see Section 6.2.1). The postprocessor only performs slight modifications on the C code produced (cf Section 6.2.2). The code for these three phases can be verified by a simple code review process and can be found in the OCaml files `PrintClight.ml`, `preproc.ml` and `postproc.ml`. We discuss in Chapter 12 the confidence that can be given to these unverified parts.

The generated C file contains the flight plan *header* (constants and variables needed by the autopilot), a main `auto_nav` function and auxiliary functions, presented in Section 9.3.2. The `auto_nav` function is the core component of the autopilot and is called at a frequency of 20Hz. At each call, `auto_nav` sets the navigation parameters to execute the flight plan (current stage, etc.). A control loop is executed in parallel with a higher frequency to translate navigation parameters into commands to the drone actuators. Chapter 9 contains some extracts of an example of a `auto_nav` function, Appendix A.1 contains a full example of an XML flight plan, the corresponding `FP` structure and the produced C code, and the VFPG project contains several examples of generated C files[3].

6.2.1 Preprocessor

The first role of the preprocessor is to convert the `FPP` input with a XML structure into a `FP` Gallina structure. The main transformations happen on the core part of the flight plan: an init block is added, macro stages are expanded, arbitrary C codes are protected and blocks are numbered.

The `init` block is an empty block appended at the beginning of the blocks list. This block has been added to ensure that the first block defined by the user is entered normally through the execution of the flight plan.

A new feature, presented in Chapter 8, is the possibility for users to define `on_enter` code that must be executed when the flight plan execution enters a block. The execution of the flight plan is detailed precisely when presenting the semantics of `FP` in Chapter 7, but let us briefly describe it in the following. The flight plan execution starts in the first

[3] https://gitlab.isae-supaero.fr/b.pollien/vfpg/-/tree/thesis/tests/regression-tests

block. It is thus not entered and its `on_enter` code is not executed. The solution found to execute the `on_enter` code of the first block without overcomplicating the current structure is to add an `init` block that is an empty block appended at the beginning of the blocks list. It will be executed first but as it is empty it does not change the UAV state nor produce effects. Then, the execution will continue to the next block and therefore the first block defined by the user will be entered and the `on_enter` code will be executed.

Macro stages are "syntactic sugar" that are translated to core stages[4]: `path` stages that take a list of waypoints to be followed by the drone are replaced by a list of `go` primitives of `nav` stages. `for` loops are also converted into `while` loops. The `for` stage in Paparazzi defines a variable with an initial value which is incremented at every loop execution until reaching the final value. As every `for` stage is removed, its variables declaration is also removed. The list of variables produced by the preprocessors that must be declared in the C code produced is thus the list of variables defined in the `for` stages.

An important point about flight plans that is discussed during this thesis is the possible presence of arbitrary C code specified by the user. This code can be function calls or conditions to evaluate but the user can also add any valid C code. Depending on the context, we decided to consider these codes either as function or variable names and the produced C code will include the code entered by the user. In order to produce valid C code, the arbitrary C code should be "protected" by adding parenthesis around it to facilitate its macro-expansion. These code generation details are presented in Section 9.3.

The final transformation indexes the blocks. They are initially referenced in `FPP` using names. The preprocessor verifies that every block reference corresponds to a position of a defined block in `FPP` (starting at index 0) and replaces each block name with its index.

In addition to the production of the `FP` Gallina structure, the preprocessor also generates the header of the produced C file. This header is appended at the beginning of the C file generated by the printer. The header is composed of definitions collected during the preprocessing phase (constants for the waypoints coordinates, block names, security height, etc.). For example, the preprocessing phase performs simple transformations, such as coordinates being converted into different formats (UTM, LLA, etc.).

Notice that the indexation pass of the blocks is currently part of the OCaml preprocessor as the relation between names and indices is needed for the generation of the C header file.

6.2.2 Postprocessor

The post-processor produces a compilable C code file from the printed Clight structure. The first part of the file is the header generated by the preprocessor. The second part contains the functions produced by the generator. Notice that we want to keep full compatibility with Paparazzi and therefore be as close as possible to the C code generated by the previous and unverified flight plan generator. Indeed, Paparazzi users may rely on the generated code structure, incidentally or not, to develop their own components. The flight plan is also not run in isolation and accesses a number of global variables in Paparazzi.

[4] All core stages are presented in Section 7.1 along with the `FP` syntax.

The CompCert `PrintClight` module used by the new generator translates the Gallina Clight structure into pseudo C code. Unfortunately, this module has mainly been developed for debugging purposes and the produced code cannot always be compiled. For instance, when a Clight variable identifier is printed, the $ symbol is added in front of the identifier. The post-processor makes a final pass to convert the printed C code into a compilable one.

Remark

We could have connected the generator directly to CompCert, as demonstrated in projects like Velus [BBP20], to avoid using the `PrintClight` module. However, Paparazzi is designed to work with various drone types, including some that might have unconventional or unsupported architectures in CompCert. Consequently, we have prefered to use the `PrintClight` module to maximize compatibility. Section 11.2.2 discuss the possibility of directly plugging the generator to CompCert.

The generated C code is very similar to the code generated by the old OCaml generator. The only noticeable differences are due to Clight limitations. For instance, the boolean operator (`&&`) does not exist in Clight and must be replaced by conditional statements. Similarly, expressions like `fun1(fun2())` cannot be defined in Clight and must be split into two statements, using a temporary variable to store the result of calling `fun2()`. These generation details are presented in Section 9.3.

Chapter 7

Specification of the flight plan language

Contents

7.1 Syntax of FPL 143
7.1.1 Notations and informal definitions 144
7.1.2 FPL definition 145
7.1.3 Default block for FP 147
7.1.4 Block index problem 149
7.2 A flight plan execution example 149
7.3 FP semantics 151
7.3.1 States and traces 152
7.3.2 Signatures of the semantics functions 155
7.3.3 Inference rules describing flight plan semantics 159

This chapter provides a formal specification for FP, the Gallina structure that represents the flight plan language FPL. First, Section 7.1 defines the syntax of FP. Then, Section 7.2 presents an example of a flight plan execution to give an intuition of its semantics. Finally, Section 7.3 describes the formal semantics of FP.

All definitions in this chapter are linked to the corresponding Coq definitions in the project repository through clickable links. Note that some definitions may slightly differ from the Coq source code as this chapter presents the original flight plan language from Paparazzi. Chapter 8 presents the language changes on FPL we bring into the new flight plan generator and definitions in this chapter are identical to the Coq source code.

7.1 Syntax of FPL

In order to present the syntax of FPL, Section 7.1.1 first introduces some notations. Section 7.1.2 defines the syntax of FPL. Then, Section 7.1.3 introduces functions concerning the default block that are essential for FPL semantics definition. Finally, Section 7.1.4 discusses a problem around blocks numbering.

7.1.1 Notations and informal definitions

A particularity of FP is that it can contain arbitrary C code provided by the autopilot user. The following notation is used to designate different sets of arbitrary C code.

Notation 7.1 - Informal specification of a set.

set ::= { *type* | "*descr*" }

This notation states that all elements of *set* are values of *type*. The informal description *descr* express what the elements of *set* should represent.

We now define informally several sets representing different types of arbitrary C code used in the FP definition.

Definition 7.2 - Types for C code (defined in BasicTypes.v).

c_code ::= { *String* | "Any valid C statement" }
c_value ::= { *String* | "Any valid C expression" }
c_cond ::= { *String* | "Any C code that can be evaluated as a boolean expression" }
var_name ::= { *String* | "Any C variable indentifier" }

We thus consider that these two informal properties hold:

- $c\_cond \subseteq c\_value \subseteq c\_code$,
- $var\_name \subseteq c\_value$.

For example, the code "`var == 1`" is a valid boolean expression in C but it can also be considered as a valid C statement if we add a semicolon. It is important to note that these notations are used to specify precisely the *expected* type of external C code in FPL but do not offer guarantees about the validity of such C code. We discuss in Section 7.3.1 the implications of the presence of such code on the generator verification process.

Coq implementation

We use the Gallina *String* type to represent arbitrary C code for two reasons. First, the OCaml parser directly produces a *String* value for the different fields of the XML flight plan that contain C code. The second reason is related to the method we use to produce Clight code. Currently, arbitrary C code is used as a variable identifier and the produced Clight code is an expression that contains only this variable. This generation process is detailed precisely in Section 9.3.

In future work, we may want to use the CompCert parser to convert arbitrary C code into Clight in order to ensure that it is indeed correct C code. This change has not been currently done as it would require large code modification in the project, in particular for the verification of semantic preservation. This point will be discussed in Chapter 10.

7.1.2 FPL definition

Using the introduced notations, the flight plan syntax of FP can now be defined as follows.

Definition 7.3 - (flight_plan, src/syntax/FlightPlanGeneric.v:146).

flight_plan ::= {|
  excpts: list *fp_exception*,
  blocks: list *fp_block*
|}

This record states that a flight plan is composed of a list of blocks and possibly exceptions.

Definition 7.4 - (fp_block, src/syntax/FlightPlan.v:48).

fp_block ::= {|

name:	*String*,	excpts:	list *fp_exception*,
id:	*block_id*,	stages:	list *fp_stage*,
pre_call:	option *c_code*,	post_call:	option *c_code*

|}

A block describes a part of the mission (e.g. "take off" or "initialise sensors"). It is composed of a name (used to pretty print error or warning messages), a unique id (generated during the preprocessing), potential local exceptions, a list of atomic instructions called *stages* and potentially code that must be executed before/after the stages execution.

Coq implementation

As presented in Section 6.2.1, blocks are numbered during the preprocessing pass as they are used to generate the C file header needed for Paparazzi. Blocks are numbered in their definition order starting from 0. For example, the second block of the flight plan will have an id of 1.

Definition 7.5 - (fp_exception, src/syntax/FlightPlanGeneric.v:34).

fp_exception ::= {|
  cond: *c_cond*,
  id: *block_id*,
  exec: option *c_code*
|}

Exceptions constitute a protection mechanism to avoid unwanted behaviours. An exception is raised when its condition cond is evaluated to true. The execution of the flight plan then proceeds to the block id and, if specified, the code exec is called. For example, an exception could be: "if the drone has less than 20% of battery left, the flight plan must execute the block that lands the drone at the *Home* position". An exception can be global and will be tested at every call to the auto_nav function or local to a block and will be tested only when the block is executed.

Definition 7.6 - (fp_stage, src/syntax/FlightPlan.v:37).

fp_stage ::=

| **WHILE** (cond: *c_cond*) (body: list *fp_stage*)
| **SET** (var: *var_name*) (value: *c_value*)
| **NAV** (mode: *fp_nav_mode*) (until: option *c_cond*) (init: *bool*)
| **DEROUTE** (idb: *block_id*)
| **RETURN** (reset: *bool*)
| **CALL** (fun: *c_code*) (until: option *c_cond*) (loop: *bool*) (break: *bool*)

Coq implementation

Section 9.1 presents an extended version of the flight plan language that adds new stages. In order to have a modular Coq development process, the definition of the flight plan is divided into two files: FlightPlanGeneric.v and FlightPlan.v. The first file contains the generic definitions of flight plans and exceptions. These definitions are common to both flight plan versions. The second file thus contains the specific definitions for the input flight plan (blocks and stages). The same principle of dividing the definitions into two files is used in the following.

Let us first present informally the semantics of *stages* which are the atomic building blocks of flight plans (the formal semantics of FP and *stages* are presented in section 7.3).

WHILE stage allows the user to repeat the execution of the stages contained in the body while the condition cond holds. It is important to note that this stage is not a classic imperative loop. This is explained with the formal definition of the semantics.

SET stage assigns value to var.

CALL stage executes the C code fun. The other parameters will be explained in the formal definition of the semantics.

DEROUTE stage changes the block being currently executed to block idb. The position in the current block (i.e. the next *stage* after the **DEROUTE** being executed) and the id of the current block are memorised as the *last* position. Note that there is only one *last* position memorised.

RETURN stage returns to the *last* position memorised, i.e. to the block that was executed prior to the last deroute stage or exception. Its reset parameter allows the user to choose whether the execution should start at the beginning of the *last* block or at its latest stage reached when the deroute or exception occurred.

NAV stage executes the navigation code corresponding to a primitive (e.g. GO to a position or do a CIRCLE) depending on the value taken by the parameter mode. This parameter is defined by the *fp_nav_mode* type, introduced in Definition 7.7, that gathers all navigation modes. A description of these different modes is provided in Paparazzi

documentation. The parameter `until` allows the user to set a termination condition in order to stop the execution of the navigation mode. The `init` field describes if the stage requires an initialisation phase, i.e. a specific code that must be executed once at the beginning of the execution of the navigation stage.

Remark

The `init` field needs to be set to `true` only for some navigation modes. These modes are defined through a function (`nav_need_init`, src/syntax/FPNavigationMode.v:235) that takes a parameter *fp_nav_mode* and returns a Boolean value stating if the mode requires an initialisation phase. Currently, the preprocessing uses the `nav_need_init` function to set the `init` parameter. However, in the Gallina generator, we have no guarantee that they are correctly set.

We could add in future work an initial pass in the Coq generator to verify that all navigation stages requiring an initialisation have their `init` parameter fields set to `true`.

Definition 7.7 - (`fp_navigation_mode`, src/syntax/FPNavigationMode.v:157).

fp_nav_mode ::=

| `HEADING` (`params`: *params_heading*)	| `ATTITUDE` (`params`: *params_attitude*)
| `MANUAL` (`params`: *params_manual*)	| `GO` (`params`: *params_go*)
| `XYZ` (`params`: *params_xyz*)	| `CIRCLE` (`params`: *params_circle*)
| `STAY` (`params`: *params_stay*)	| `FOLLOW` (`params`: *params_follow*)
| `EIGHT` (`params`: *params_eight*)	| `OVAL` (`params`: *params_oval*)
| `GUIDED` (`params`: *params_guided*)	| `HOME`
| `SURVEY_RECTANGLE` (`params`: *params_survey_rectangle*)	

Coq implementation

Navigation modes are specific to Paparazzi. We could imagine to adapt this code generator to other autopilots or other code generation problems for synchronous languages. For this reason, navigation modes are defined separately from the flight plan. It would only be necessary to modify the navigation functions, their associated semantics and some lemmas in order to adapt the generator while preserving verified properties.

7.1.3 Default block for FP

In Definition 7.3, the flight plan may contain no blocks and in Definition 7.4 the blocks might have no stages. We thus need a default block that contains at least one stage. This block is executed when there are no blocks nor stages left to execute and takes the drone to the `HOME` position[1]. The default block is always implicitly positioned as the last block of

[1]This behaviour has been chosen to be consistent with the old generator.

the flight plan, i.e. at the end of the blocks defined by the user. The `id` of the default block thus depends on the length of the flight plan. We first define a default block function that takes a *block_id* and returns a default block with the corresponding id.

Definition 7.8 - (default_block, src/syntax/FlightPlan.v:67).

```
default_block : block_id → fp_block
default_block id ::= {|
  name   := "HOME",                  excpts    := [],
  id     := id,                      pre_call  := None,
  stages := [NAV HOME None false],   post_call:= None
|}
```

Remark

The presentation of Coq functions respects the following pattern in the manuscript. First, we provide the name of the function and a clickable reference of its Gallina definition in the GitLab project. The signature is then provided in Coq style. Eventually, the function is defined.

The default block index is the smallest index available, which thus corresponds to the number of blocks in the flight plan.

Definition 7.9 - (get_default_block_id, src/syntax/FlightPlanGeneric.v:172).

```
get_default_block_id : flight_plan → block_id
get_default_block_id fp ::= length fp.blocks
```

We denote by $id_{db}(fp)$ the default block index of the flight plan *fp*.

The *get_block* function that gets the nth block in a flight plan can now be defined: if the block does not exist, then the default block is returned.

Definition 7.10 - (get_block, src/syntax/FlightPlanGeneric.v:182).

get_block : *flight_plan* → *block_id* → *fp_block*

$$get\_block\ \ fp\ \ id ::= \begin{cases} b & \text{if } b \in fp.\texttt{blocks} \ \wedge\ b.\texttt{id} = id \\ default\_block\ id_{db}(fp) & \text{otherwise.} \end{cases}$$

We denote by $block_{fp}(id)$ the result of "*get_block fp id*".

Coq implementation

The implementation of the *get_block* function simply gets the *id*th block in the plan or the default block if *id* is too big. This supposes of course that the blocks are well-numbered, i.e. that the set of ids for a set of n blocks is exactly $\{0,\ldots,n-1\}$. Section 9.2 details how we ensure this property.

7.1.4 Block index problem

The *get_block* function returns a block for any id. We define a subset of *block_id* that contains all block ids corresponding to a real block in a flight plan.

Definition 7.11 - Correct block index.
$\forall\ id\ \in\ block\_id, fp\ \in\ flight\_plan,$
The index *id* is a *correct block index* of the flight plan *fp* iff $block_{fp}(id).\texttt{id} = id$.

Throughout the execution of the flight plan, the semantics may change *id*s that are not correct. The flight plan generated with the old generator has an execution mechanism to normalise block ids in order to manipulate only *correct block indexes* during execution. We want FP to have a similar mechanism. We thus define a normalisation function that transforms any block id into a *correct block index* for a specific flight plan. This function is used in the semantics as we also want to maintain *correct states* throughout the execution, i.e. execution states of the flight plan that only store *correct block index*.

Definition 7.12 - (normalise_block_id, src/syntax/FlightPlanGeneric.v:207).
$normalise\_id : flight\_plan \rightarrow block\_id \rightarrow block\_id$

$$normalise\_id\ \ fp\ \ id ::= \begin{cases} id_{db}(fp) & \text{if } id_{db}(fp) < id \\ id & \text{otherwise.} \end{cases}$$

We use the notation $\|id\|_{fp}$ to describes the result of "$normalise\_id\ \ fp\ \ id$".

If the blocks are well-numbered (as supposed in section 7.1.3), the following property holds: $\forall\ id,\ \ block_{fp}(id).\texttt{id} = \|id\|_{fp}$.

7.2 A flight plan execution example

Execution of flight plans is similar to execution of programs written in typical synchronous languages such as Esterel [BR19] or Lustre [HCRP91]. The execution of synchronous code is composed of an initialisation phase followed by periodic calls to a step function. Here, the initialisation phase sets the environment in order to start the execution at block 0. Executing the flight plan then consists in calling regularly the auto_nav function. This function starts by testing all global and local exceptions. If an exception is raised, the *last* position is updated with the current position, then the execution of the flight plan is derouted to the block referred by the exception, and the function terminates. Otherwise, the execution of the flight plan continues. If the environment has been correctly initialised, the first call to auto_nav starts by the execution of the first block of the plan, and stages inside this block are to be executed. There are 2 types of stages: *continue* stages and *break* stages. The type of a stage is defined by the language semantics as the same stage can be of both types depending on the context. Executing a block consists in executing its stages sequentially in their definition order as long as they are *continue* stages. The execution of the auto_nav function terminates when a *break* stage is executed or when the last stage of a block has been executed. The execution of the flight plan pauses and is resumed by the next call to auto_nav.

Example of a flight plan:

```
{| excpts := [
    {| cond:= "Battery() < 20",
        id:= 15,
        exec:= "LowBatteryMode()"
    |}
  ],
  blocks := [
    {| name:= "Init_GPS", id:= 0, excpts:= [],
       stages:= [
        CALL "InitSensors()" None false false;
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"],
      pre_call:= None, post_call: None
    |};
    {| name:= "Take-off", id:= 1, excpts:= [],
       stages:= [
        NAV (TakeOff params) None true;
        DEROUTE 10],
      pre_call: None, post_call: None
    |};
    ...
    {| id:= 10, ... |};
    ...
    {| name:= "HOME", id:= 15, excpts:= [],
      stages:= [
        NAV HOME None false],
      pre_call:= None, post_call:= None
    |}
  ]
|}
```

Possible execution of the `auto_nav` function:

Call	Current Block ID	Code Executed

1	0	`Battery() < 20` ⇑ `false`
		`InitSensors()`
		`!GPSFixValid()` ⇑ `true`

2 → 8	0	`Battery() < 20` ⇑ `false` `!GPSFixValid()` ⇑ `true`

9	0	`Battery() < 20` ⇑ `false`
		`!GPSFixValid()` ⇑ `false`
		`home = GPSPosHere()`

10	1	`Battery() < 20` ⇑ `false` `StartMotors()`
11 → 19	1	`Battery() < 20` ⇑ `false` `TakeOffDone()` ⇑ `false`

20	1	`Battery() < 20` ⇑ `false`
		`TakeOffDone()` ⇑ `true`
		`Deroute` → 10

21	10	...
⋮	⋮	⋮
1000	4	`Battery() < 20` ⇑ `true` `LowBatteryMode()`
1001	15	`home()`
⋮	⋮	⋮

Figure 7.1: Example of an execution of a flight plan.

Figure 7.1 presents an example of a flight plan and one of its possible executions. This flight plan contains one global exception and several blocks of which only 4 are represented. Let us present briefly the execution of the `auto_nav` function on this example. From the call 1 to 999, the exception is always tested first (the expression `Battery() < 20` is evaluated) but it always returns `false`. The exception is thus not raised. We detail later how the exception is handled when raised, but at first, we focus on the execution of the blocks and stages of the plan.

During the first call, block 0 is entered and its first stage is executed. The **`CALL`** stage is a *continue* stage: its execution simply calls the C code it contains and the next stage is executed. The next stage is a **`WHILE`** stage, therefore the condition of the loop, i.e. `"!GPSFixValid()"`, must be evaluated. We assume that this evaluation returns `true`. In this case, the **`WHILE`** stage is a *break* stage: it ends the execution of the `auto_nav` function. The flight plan will continue its execution at the next call to `auto_nav` during which the execution of the flight plan is resumed where it was left, i.e. inside the **`WHILE`** loop. As the loop body is empty in this example, there is no stage to execute and the condition of the loop is reevaluated. The execution of `auto_nav` continues to evaluate the loop condition until it is evaluated to `false`. This is of course a possible execution among many as the

system that will be executing the flight plan is a drone evolving in a real environment, and thus, the time to set the GPS can vary. We assume that the loop condition evaluates to `false` in call 9. In this case, the **`WHILE`** stage is considered as *continue*. The last stage **`SET`** is therefore executed and as it is a *continue* stage, its execution terminates the execution of the block, stopping `auto_nav` execution and the next block (numbered 1 here) will be executed at next iteration.

The **`NAV`** stage execution is translated into a call to a C function that sets navigation parameters depending on the navigation primitive used. These parameters will be used by the autopilot and translated into orders for the motors. In this example, the **`NAV`** stage runs the navigation primitive `TakeOff`. This stage requires an initialisation step as the `init` parameter is set to `true`. There is thus some specific code that will be called at the first execution of the stage (here `StartMotors` that does not appear in the flight plan but is specified in Paparazzi autopilot and added by the generator). Then, the C code corresponding to the second part of the `TakeOff` primitive is a function `TakeOffDone` that returns a boolean value depending on whether the drone has taken off or not. If the `TakeOffDone` function returns `false`, then the stage is a *break* stage, otherwise **`NAV`** is considered as a *continue* stage. Notice that the navigation primitive `TakeOff` and the C functions called have been forged for this example. The real generated C code may correspond to calls to several functions that might have non-explicit names. This code is generated from functions found in the `FPNavigationModeGen.v` file.

Finally, when the drone has taken off (call 20), `auto_nav` execution continues and the **`DEROUTE`** stage is executed. This stage is a *break* stage and the next call to `auto_nav` will execute the corresponding derouted block (block 10 here).

The execution of the function `auto_nav` will continue following the same principle for the blocks and stages until call 1000 when the C expression `Battery() < 20` is evaluated to `true` and thus the exception is raised. In this case, the code `exec` of the exception is executed (here `LowBatteryMode()`), and the current block to execute is modified. At the next call to the `auto_nav` function, block 15 will then be executed.

Blocks and stages are thus normally executed one after the other in their definition order, but **`DEROUTE`** statements or raised exceptions can change the currently executed block. Note that human operators can also manually change the current block during the flight plan execution on Paparazzi control console.

In this example, the flight plan does not contain `pre_call` and `post_call` code. This code is specific to a block and it is always executed when a block is executed no matter what stage is being proceeded. The `pre_call` is executed after global exceptions and before the execution of local exceptions. The `post_call` code is always executed at the end of the execution of the stages, just before the termination of the `auto_nav` function.

7.3 FP semantics

Flight plans describe how the drone should behave when flying autonomously. In this section, we define a big-step semantics for `FP`. This semantics is composed of a initial environment and a *step* function representing a usual semantics evaluation function [NN07, Win93].

The *step* function describes both the system evolution and observable events that happen during execution of the `auto_nav` function. The system modeled as an execution state and observable events is introduced in Section 7.3.1. Section 7.3.2 presents the signature of semantics functions with their associated notation and also defines functions manipulating, for example, states or navigation modes. Finally, Section 7.3.3 specifies inference rules defining the *step* function.

7.3.1 States and traces

A semantics defines how a system evolves during its execution from an initial state s to a final state s', but also describes the interactions with the outside world. These external operations are modeled by *traces* or *outputs*. For instance, an imperative programming language often uses as a state an abstraction of the computer memory. The corresponding semantics describes how the memory changes during the program execution. Traces thus can be sequences of accesses to an external memory or messages received and sent over a network.

Definition 7.13 - (`fp_state`, `src/semantics/FPEnvironment.v:41`).

fp_state ::= {|
  `idb`: *block_id*, `stages`: list *fp_stage*,
  `lidb`: *block_id*, `lstages`: list *fp_stage*
|}

We abstract the real memory state of the flight plan by focusing on key information for the execution, as shown in Definition 7.13. An *fp_state* contains the current position in the flight plan, represented by `idb`, the current block id, and `stages`, the remaining stages to be executed within this block. The state cannot store an id for the stages as they are not numbered. The flight plan state also contains the previous position similarly represented by `lidb` and `lstages`. The previous position is saved when a deroute or an exception occurs and used with the **`RETURN`** stage to return to the previous position.

Remark

The state can only store one previous position and it is thus only possible to do a single return. For example, let us suppose the following execution. The flight plan executes in block A a deroute stage to block B. During the execution of block B, another deroute stage to block C is executed. If a return stage is executed in block C, then the flight plan will resume the execution in block B. However, if there is a second return stage in block B, the flight plan will not resume the execution in block A but resume in the last memorised position which is block B.

This limitation is present in the current version of Paparazzi and there are two reasons for this restriction. First, the previous stage memorised in the state is used only with the return stage. In practice, this stage is rarely used by Paparazzi users. Secondly, the Paparazzi UAV autopilot is executed on limited resources and it is not possible to have dynamic allocation to memorize the complete positions history.

When executing a flight plan, several C functions specific to each type of drone (fixed wing, rover, rotorcraft, etc.) are called. For instance, the `NavCircleWaypoint` function setting the navigation parameters to perform a circle has different implementations for fixed wing or rotocraft drones. Moreover, the flight plan execution can also execute arbitrary C code defined by the user. The behaviour of such code cannot be formally defined. The semantics of the flight plans will therefore produce outputs that represent calls to such C code that are considered as *external calls*.

Definition 7.14 - (fp_event, src/syntax/BasicTypes.v:86).

fp_event ::= **COND** (*c_cond* × *bool*)
| **C_CODE** *c_code*
| **SKIP**

A single trace is a value of *fp_event*, i.e. the evaluation of a C expression representing a condition or arbitrary C code. The *bool* field records the resulting value of the condition *c_cond*. We also consider a **SKIP** event that does nothing, useful for functions returning *fp_event* only in some cases. Notice that defining **SKIP** avoids using an option type to manage such functions. A *trace* is defined as a list of *fp_event*.

Definition 7.15 - (fp_trace, src/syntax/BasicTypes.v:92).

fp_trace ::= list *fp_event*

The execution of C code represented in the trace may modify the memory environment of the drone. Specific C functions like navigation functions are defined outside the flight plan code, therefore their execution does not modify the state of the flight plan. However, arbitrary C code may be introduced by users and we must assume that such code does not modify the state of the flight plan as discussed in section 10.4.1. The drone memory environment can thus be decomposed into two disjoint parts: a) the memory space abstracted by *fp_state* b) the memory space that can be modified when executing C code calls represented by *fp_trace*.

Definition 7.16 - (fp_env, src/semantics/FPEnvironment.v:50).

fp_env ::= {|
`state`: *fp_state*,
`trace`: *fp_trace*
|}

An environment thus contains the current state of the flight plan and a history of the external operations that occurred. Example 7.17 presents an environment produced during the execution of a flight plan.

Example 7.17 - An environment during a flight plan execution.

The environment e_1 is the resulting *fp_env* after call 1 in the flight plan example presented in Figure 7.1.

```
e1 ::= {| state := {| idb  := 0, stages  := [ WHILE "!GPSFixValid()" [];
                                              SET "home" "GPSPosHere()" ],
                      lidb := 0, lstages := [ CALL "InitSensors()" None false false;
                                              WHILE "!GPSFixValid()" [];
                                              SET "home" "GPSPosHere()" ] |},
          trace := [ COND ("Battery() < 20", false);
                     C_CODE "InitSensors()";
                     COND ("!GPSFixValid()", true) ] |}.
```

In order to define the semantics inference rules, we often have to access the elements of the state from a given *fp_env*. To ease readability, we denote by $e.\texttt{f}$ the value of $e.\texttt{state.f}$ to access the field `f` of the underlying *fp_state* in the following.

The execution of arbitrary C code in an environment results in appending new events to the trace. We define a specific function for appending a new event to the trace as this operation is often used in the semantics definition.

Definition 7.18 - (app_trace, src/semantics/FPEnvironment.v:76).

app_trace : *fp_env* → *fp_trace* → *fp_env*
app_trace *e* *t* ::= $e\{\texttt{trace} := e.\texttt{trace} +\!\!+ t\}$.

We denote by $e(\!|t|\!)$ the expression *app_trace e t*. We use the same notation for the execution of arbitrary C code as they are semantically equivalent.

Notation 7.19 - Execution of arbitrary C code.

$\forall$ *code* $\in$ *c_code*, $e(\!|code|\!) ::= e\{\texttt{trace} := e.\texttt{trace} +\!\!+ [\texttt{C\_CODE}\ code]\}$

The result of the evaluation of a condition is required to define the semantics, but since such conditions are arbitrary C code that cannot be interpreted, we assume the existence of an *eval* function specified as follows.

Parameter 7.20 - (eval, src/semantics/FPEnvironmentGeneric.v:11).

eval : *fp_trace* → *c_cond* → *bool*

The result of the evaluation of arbitrary C code depends on its execution environment which is, as previously explained, represented by its trace history. The function *eval* thus takes as parameters an *fp_trace*, the code condition to evaluate and returns a boolean value. Using this parameter function, we can now define the *evalc* function that evaluate a arbitrary C condition for a specific environment.

Definition 7.21 - (evalc, src/semantics/FPEnvironment.v:79).

evalc : *fp_env* → *c_cond* → (*bool* × *fp_env*)
evalc *e* *cond* ::= `let` *res* := *eval* *e*.`trace` *cond* `in`
(*res*, $e\{\texttt{trace} := e.\texttt{trace} +\!\!+ [\texttt{COND}\ (cond,\ res)]\}$)

The function *evalc e c* returns a couple $(b,\ e')$ such that evaluating the C code *c* returns the boolean *b*. The trace of the environment *e* is updated by appending **COND** $(c,\ b)$ to it, yielding e'.

The trace history contained in *fp_env* is essential for evaluating conditions. Let us consider an example: evaluating the condition "`Battery()` < 80" twice may produce different results as the drone battery is emptying during flight. But as we chose to not specify the real outside environment, considering a trace history allows us to support such cases and to represent C function calls with side effects.

Finally, we define the function *init_env* that produces an initial environment for a specific flight plan.

Definition 7.22 - (`init_env`, `src/semantics/FPEnvironment.v:121`).

$$\begin{aligned} &\mathit{init\_env} : \mathit{flight\_plan} \rightarrow \mathit{fp\_env} \\ &\mathit{init\_env}\ \ \mathit{fp} ::= \{|\ \texttt{state} := \{|\ \texttt{idb} := 0, \quad \texttt{stages} := \mathit{block}_{\mathit{fp}}(0), \\ &\qquad\qquad\qquad\qquad\quad \texttt{lidb} := 0, \quad \texttt{lstages} := \mathit{block}_{\mathit{fp}}(0)\ |\}, \\ &\qquad\qquad\quad \texttt{trace} := \texttt{[]}\ |\} \end{aligned}$$

The initial environment contains an empty trace and a state refering to the first block of the flight plan (block 0). The list of stages remaining to be executed (the fields `stages` and `lstages`) are thus all the stages of block 0.

7.3.2 Signatures of the semantics functions

The definition of the flight plan semantics can be divided into several functions. This section introduces the signature of these functions and gives an informal description of their behaviour (Section 7.3.3 defines them formally).

Executing the flight plan consists in calling regularly the `auto_nav` function. The *step* semantics function represents the execution of a call to `auto_nav`. We emphasize the fact that during this execution, the flight plan is not run to completion, but the current stage and/or the current block are modified and C code may be called. The resulting environment will be ready for another execution step.

Signature 7.23 - (`step`, `src/semantics/FPBigStep.v:263`).

$$\mathit{step} : \mathit{flight\_plan} \rightarrow \mathit{fp\_env} \rightarrow \mathit{fp\_env}$$

The property "$\mathit{step}\ \mathit{fp}\ e = e'$", noted $e \xhookrightarrow[\mathit{fp}]{\mathsf{FP}} e'$, states that e' is the resulting environment after the execution of the flight plan *fp* starting from the environment e.

The execution of a flight plan can be decomposed into two phases. First, all exceptions are checked and if one of them is raised, then the environment is derouted to a safe block. Second, the current stages of the flight plan are evaluated.

7.3.2.1 Exceptions

Executing exceptions consists of testing if there are global or local exceptions that are raised.

Signature 7.24 - (exception, src/semantics/FPBigStepGeneric.v:250).

$$exception : flight\_plan \to fp\_env \to (bool \times fp\_env)$$

The property "$exception\ fp\ e = (res, e')$", noted $e \xrightarrow[fp]{\text{exception}} e' \Downarrow res$, states that the test of fp exceptions starting from environment e terminates in environment e' and res is `true` iff a global or local exception has been raised.

The definition of the *exception* function uses *test_exception* that verifies if a single exception is raised. *test_exceptions* is a straightforward extension that checks a list of *fp_exception*.

Signature 7.25 - (test_exception, src/semantics/FPBigStepGeneric.v:217).

$$test\_exception : flight\_plan \to fp\_env \to fp\_exception \to (bool \times fp\_env)$$

The property "$test\_exception\ fp\ e\ ex = (res,\ e')$", noted $e \Uparrow ex \xrightarrow[fp]{\text{test\_exception}} e' \Downarrow res$, states that the test of the ex exception starting from e terminates in environment e' and res is `true` iff the exception ex has been raised.

Signature 7.26 - (test_exceptions, src/semantics/FPBigStepGeneric.v:236).

$$test\_exceptions : flight\_plan \to fp\_env \to list\ fp\_exception \to (bool \times fp\_env)$$

The property "$test\_exceptions\ fp\ e\ exs = (res,\ e')$", noted $e \Uparrow exs \xrightarrow[fp]{\text{test\_exceptions}} e' \Downarrow res$, states that testing the exs exceptions starting from e terminates in environment e' and res is `true` iff an exception of exs has been raised.

7.3.2.2 Run stages

If there are no raised exceptions, then the stages of the flight plan can be executed. The execution of the remaining stages is described by the *run_step* function.

Signature 7.27 - (run_step, src/semantics/FPBigStep.v:254).

$$run\_step : flight\_plan \to fp\_env \to fp\_env$$

The property "$run\_step : fp\ e = e'$", noted $e \xrightarrow[fp]{\text{stages}} e'$, states that the execution of fp starting from the environment e with the list $e.\texttt{stages}$ remaining to be executed terminates in environment e'. If there are no remaining stages, the execution stops and at the next call to the *step* function, the next block will be executed.

The definition of the semantics for some stages requires specific functions. This is the case for the navigation stages, and stages that change the block to execute, such as **DEROUTE** and **RETURN** stages.

7.3.2.3 Navigation code

The navigation stage regroups all navigation modes of Paparazzi. The 13 navigation modes, presented in Definition 7.7, have specific C code to be executed. The function *nav_code_exec* describes the execution of the specific code of the navigation stages.

Signature 7.28 - (nav_code_sem, src/semantics/FPBigStep.v:160).

nav_code_exec : *flight_plan* → *fp_env* → *fp_nav_mode* → option *c_cond* → *fp_env*

The property "*nav_code_exec fp e mode until* = e'", noted $e \Uparrow (mode,\ until) \xrightarrow[fp]{\text{nav}} e'$, states that the execution of the **NAV** stage with parameters *mode until* from the state e terminates in the state e'.

The definition of the semantics of the navigation stage and *nav_code* requires functions that produce the C code that must be executed for each navigation mode. Some functions return *fp_trace* as some navigation modes do not require executing code and others require calls to several C instructions.

As presented in Example 7.1, the execution of navigation stages may start with an initialisation phase if the `init` parameter is set to `true`. The code that must be executed during this phase is described by the *nav_init_code* function.

Definition 7.29 - (init_code_nav_sem, src/semantics/FPNavigationModeSem.v:54).

nav_init_code : *fp_nav_mode* → *fp_trace*

The different navigation modes have specific behaviour when they are executed after the initialisation phase. In the initial OCaml generator, they were generated independently as every navigation mode was considered as a single stage. However, we want now to gather all navigation modes under a single navigation stage in order to separate explicitly the specific code of Paparazzi from the generator. We will also show that it also simplifies the verification of the generator. The drawback of this approach is that the semantics must take into account all navigation functions cases: functions that have specific conditions, functions that execute specific C code, etc. We thus need to define all the following functions that generate conditions, traces or events for the different navigation cases. We present precisely the meaning of these different functions when defining the inference rules in Section 7.3.3.

The *fp_trace* generated by *nav_code* corresponds to the specific C code of the navigation modes.

Definition 7.30 - (gen_fp_nav_code_sem, src/semantics/FPNavigationModeSem.v:310).

nav_code : *fp_nav_mode* → *fp_trace*

The execution of all the different navigation modes could be expressed using only the *nav_code* function except for the `GO` mode. This mode has a different execution structure as it has to test a specific condition to terminate (the drone has reached its destination) and has a specific code to execute. In order to have a generic semantics for all the navigation modes, we define the two following functions.

First, the *nav_cond* function returns the condition if the navigation mode requires it otherwise it returns `None`. The condition is therefore only returned for the `GO` mode.

Definition 7.31 - (nav_cond_sem, src/semantics/FPNavigationModeSem.v:88).

nav_cond : *fp_nav_mode* → option *c_cond*

Secondly, the *last_wp* function returns an event that corresponds to C code that must be executed for the `GO` navigation mode. This code is executed when the mode finishes its execution and stores in the `last_wp` variable the waypoint corresponding to the GPS

position reached. The **SKIP** event is returned for the modes that have no code to execute, i.e. all the other modes.

Definition 7.32 - (last_wp_exec, src/semantics/FPEnvironment.v:136).

last_wp : *fp_nav_mode* → *fp_event*

The code generated by these functions for the navigation modes is very specific to the Paparazzi UAV autopilot and is not interesting in the context of the presentation of the verified generator. We thus not define formally these functions but their implementations can be found in the source code repository using the provided links.

Finally, we define functions corresponding to arbitrary C code added by the user that must be executed before or after the execution of the navigation code. This code is stored as a parameter of the navigation mode and added during the preprocessing.

Definition 7.33 - (pre_call_nav_sem, src/semantics/FPNavigationModeSem.v:109).

pre_call_nav : *fp_nav_mode* → *fp_event*

Definition 7.34 - (post_call_nav_sem, src/semantics/FPNavigationModeSem.v:119).

post_call_nav : *fp_nav_mode* → *fp_event*

7.3.2.4 Change block

During the execution of the flight plan, an operation that often occurs is changing the block that must be executed. The *goto_block* function describes how the drone environment evolves during this operation.

Signature 7.35 - (goto_block, src/semantics/FPBigStepGeneric.v:199).

$$goto\_block : flight\_plan \to fp\_env \to block\_id \to fp\_env$$

The property "$goto\_block\ fp\ e\ id = e'$", noted $e \Uparrow id \xrightarrow[fp]{\text{goto\_block}} e'$, states that the deroute from environment e to block id generates a new environment e'. The new environment e' memorizes, as the previous position, the current block position of e and its current position points to block id.

The *goto_block* function executes the C code that must be evaluated before changing the block[2] and then modifies the state to change the block that must be executed. The state modification is realised by the *change_block* function. This function modifies the current position and the previous position is only modified if the new block to be executed is different from the current block. It is important to note that the *change_block* function normalises the block id passed as a parameter to reproduce the behaviour of the previous generator who makes sure to always be in a stable state with *correct block index*.

[2]The code executed is presented in section 7.3.3 and sets the internal variables of the autopilots that must be initialised when the flight plan will execute a new block.

Definition 7.36 - (change_block, src/semantics/FPEnvironment.v:86).

change_block : *flight_plan* → *fp_env* → *block_id* → *fp_env*

$$
\begin{aligned}
\mathit{change\_block}\ \mathit{fp}\ e\ \mathit{id}\ ::= \{|\ \mathtt{state} := \{|\ \mathtt{idb} &:= \|\mathit{id}\|_{\mathit{fp}},\\
\mathtt{stages} &:= \mathit{block}_{\mathit{fp}}(\|\mathit{id}\|_{\mathit{fp}}).\mathtt{stages},\\
\mathtt{lidb} &:= \begin{cases} e.\mathtt{lidb} & \text{if } e.\mathtt{idb} = \mathit{id}\\ e.\mathtt{idb} & \text{otherwise.}\end{cases}\\
\mathtt{lstages} &:= \begin{cases} e.\mathtt{lstages} & \text{if } e.\mathtt{idb} = \mathit{id}\\ e.\mathtt{stages} & \text{otherwise.}\end{cases}\\
|\},\\
\mathtt{trace} := \quad & e.\mathtt{trace}\ |\}
\end{aligned}
$$

Finally, we define the *return_block* function that modifies the environment passed as parameter to resume the execution in the state memorised as the last position. The new block to be executed is then the last block id saved. The *reset* parameter states whether the execution must be resumed at the beginning of the block or at the memorised position.

Definition 7.37 - (return_block, src/semantics/FPEnvironment.v:155).

return_block : *flight_plan* → *fp_env* → *bool* → *fp_env*

$$
\begin{aligned}
\mathit{return\_block}\ \mathit{fp}\ e\ \mathit{reset}\ ::= \{|\ \mathtt{state} := \{|\ \mathtt{idb} &:= e.\mathtt{lidb},\\
\mathtt{stages} &:= \begin{cases} \mathit{block}_{\mathit{fp}}(e.\mathtt{lidb}).\mathtt{stages} & \text{if } \mathit{reset}\\ e.\mathtt{lstages} & \text{otherwise.'}\end{cases}\\
\mathtt{lidb} &:= e.\mathtt{lidb},\\
\mathtt{lstages} &:= e.\mathtt{lstages}\ |\},\\
\mathtt{trace} := \quad & e.\mathtt{trace}\ |\}
\end{aligned}
$$

7.3.3 Inference rules describing flight plan semantics

In this section we precisely define the inference rules describing flight plans semantics. This semantics is defined by the *step* function that describes an execution of the `auto_nav` function.

Definition 7.38 - (step, src/semantics/FPBigStep.v:263).

$$\frac{e \xrightarrow[\mathit{fp}]{\text{exception}} e' \Downarrow \mathtt{true}}{e \xhookrightarrow[\mathit{fp}]{\mathsf{FP}} e'}\ \text{(FP-1)} \qquad \frac{e \xrightarrow[\mathit{fp}]{\text{exception}} e' \Downarrow \mathtt{false} \quad e' \xrightarrow[\mathit{fp}]{\text{stages}} e'' \quad \mathit{block}_{\mathit{fp}}(e'.\mathtt{idb}).\mathtt{post\_call} = \mathtt{None}}{e \xhookrightarrow[\mathit{fp}]{\mathsf{FP}} e''}\ \text{(FP-2)}$$

$$\frac{e \xrightarrow[\mathit{fp}]{\text{exception}} e' \Downarrow \mathtt{false} \quad e' \xrightarrow[\mathit{fp}]{\text{stages}} e'' \quad \mathit{block}_{\mathit{fp}}(e'.\mathtt{idb}).\mathtt{post\_call} = \mathtt{Some}\ \mathit{code}}{e \xhookrightarrow[\mathit{fp}]{\mathsf{FP}} e''(\!|\mathit{code}|\!)}\ \text{(FP-3)}$$

The execution of the *step* function starts by testing the possible exceptions. Inference rule FP-1 describes what happens when an exception is raised and rules FP-2 and FP-3 describe what happens when no exception is raised. Inference rule FP-3 presents more specifically the case when the current block contains a `post_call` code to execute.

Notice that in our Coq implementation, the semantics is implemented as a computable *function*. However, we decided to present it as a set of inference rules for readability in the present manuscript. Figure 7.2 presents the Gallina code corresponding to the *step* function. Notice that the function behaves similarly as the inference rules of Definition 7.38, the only noticeable difference being the function `get_code_block_post_call` that converts the post call of the current block into a trace. If the post call code is None, then the function returns a trace with only a **`SKIP`** event.

```
Definition step (e: fp_env): fp_env :=
  match exception fp e with
  | (true, e') ⇒ e'
  | (false, e') ⇒
    let e'' := run_step e' in
    let post := get_code_block_post_call (get_current_block fp e')
    in
      app_trace e'' post
  end.
```

Figure 7.2: Gallina code for the *step* function

Coq implementation

Using functions instead of propositions simplifies the proof as the function can be computed, but also allows us to prove that the function terminates. Section 10.3 discusses more precisely the question of the semantics and `auto_nav` function termination.

7.3.3.1 Inferences rules for exceptions

Execution of exceptions, showed in Definition 7.39, consists of testing all global exceptions of the flight plan and local exceptions of the current block.

Inference rule FP-4 specify the case when a global exception is raised. If there is no global exception raised, the `pre_call` code is executed, if the current block contains any, and then the local exceptions are tested. Inference rules FP-5 and FP-6 characterise respectively what happens when there is no pre call code to execute and when there is some.

Definition 7.39 - (exception, src/semantics/FPBigStepGeneric.v:250).

$$\frac{e \Uparrow fp.\texttt{excpts} \xhookrightarrow[fp]{\text{test\_exceptions}} e' \Downarrow \texttt{true}}{e \xhookrightarrow[fp]{\text{exception}} e' \Downarrow \texttt{true}} \text{(FP-4)}$$

$$\frac{\begin{array}{c} e \Uparrow fp.\texttt{excpts} \xhookrightarrow[fp]{\text{test\_exceptions}} e' \Downarrow \texttt{false} \qquad block_{fpe}(e'.\texttt{idb}).\texttt{pre\_call} = \texttt{None} \\ e' \Uparrow block_{fpe}(e'.\texttt{idb}).\texttt{excpts} \xhookrightarrow[fp]{\text{test\_exceptions}} e'' \Downarrow res \end{array}}{e \xhookrightarrow[fp]{\text{exception}} e'' \Downarrow res} \text{(FP-5)}$$

$$\frac{\begin{array}{c} e \Uparrow fp.\texttt{excpts} \xhookrightarrow[fp]{\text{test\_exceptions}} e' \Downarrow \texttt{false} \qquad block_{fpe}(e'.\texttt{idb}).\texttt{pre\_call} = \texttt{Some}\ code \\ e' \langle\!| code |\!\rangle \Uparrow block_{fpe}(e'.\texttt{idb}).\texttt{excpts} \xhookrightarrow[fp]{\text{test\_exceptions}} e'' \Downarrow res \end{array}}{e \xhookrightarrow[fp]{\text{exception}} e'' \Downarrow res} \text{(FP-6)}$$

Definition 7.40 - (test_exceptions, src/semantics/FPBigStepGeneric.v:236).

$$\frac{}{e \Uparrow [] \xhookrightarrow[fp]{\text{test\_exceptions}} e \Downarrow \texttt{false}} \text{(FP-7)} \qquad \frac{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e' \Downarrow \texttt{true}}{e \Uparrow ex :: exs \xhookrightarrow[fp]{\text{test\_exceptions}} e' \Downarrow \texttt{true}} \text{(FP-8)}$$

$$\frac{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e' \Downarrow \texttt{false} \qquad e' \Uparrow exs \xhookrightarrow[fp]{\text{test\_exceptions}} e'' \Downarrow res}{e \Uparrow ex :: exs \xhookrightarrow[fp]{\text{test\_exceptions}} e'' \Downarrow res} \text{(FP-9)}$$

The *test_exceptions* function tests individually all given exceptions by structural induction on the exceptions list (see Definition 7.40). Inference rule FP-7 illustrates the base case when the list is empty. If the list contains at least one element, the exception is tested. Inference rule FP-8 corresponds to the case when the exception is raised and inference rules FP-9 describes the recursive call when the first exception of the list is not raised.

The execution of an exception, described in Definition 7.41, consists first of disabling the exception with an `idb` that is the same that the current block (see inference rule FP-10). We do not want a raised exception to deroute the execution to the same block as the one being currently executed. This behaviour was specified by the previous generator and Paparazzi users may rely on it, therefore we cannot change it. In the other cases, the condition of the exception is evaluated using *evalc* function. Inference rule FP-11 describes the case when the exception is not raised. In the case where the exception is raised, the `exec` code is executed if there is one, and then the block is derouted (c.f. rules FP-12 and FP-13).

Definition 7.41 - (test_exception, src/semantics/FPBigStepGeneric.v:217).

$$\frac{e.\texttt{idb} = ex.\texttt{id}}{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e \Downarrow \texttt{false}} \text{(FP-10)}$$

$$\frac{e'.\texttt{idb} \neq ex.\texttt{id} \qquad evalc\ e\ ex.\texttt{cond} = (\texttt{false}, e')}{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e' \Downarrow \texttt{false}} \text{(FP-11)}$$

$$\frac{\begin{array}{c} e'.\texttt{idb} \neq ex.\texttt{id} \\ evalc\ e\ ex.\texttt{cond} = (\texttt{true}, e') \qquad ex.\texttt{exec} = \texttt{None} \qquad e' \Uparrow id \xhookrightarrow[fp]{\text{goto\_block}} e'' \end{array}}{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e'' \Downarrow \texttt{true}} \text{(FP-12)}$$

$$\frac{\begin{array}{c} e'.\texttt{idb} \neq ex.\texttt{id} \qquad evalc\ e\ ex.\texttt{cond} = (\texttt{true}, e') \\ ex.\texttt{exec} = \texttt{Some}\ code \qquad e' \langle\!| code |\!\rangle \Uparrow ex.\texttt{id} \xhookrightarrow[fp]{\text{goto\_block}} e'' \end{array}}{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e'' \Downarrow \texttt{true}} \text{(FP-13)}$$

Definition 7.42 - (goto_block, src/semantics/FPBigStepGeneric.v:199).

$$\frac{change\_block\ fp\ e\ id = e' \qquad e' \langle\!| \texttt{reset\_time; init\_stage} |\!\rangle = e''}{e \Uparrow id \xhookrightarrow[fp]{\text{goto\_block}} e''} \text{(FP-14)}$$

We denote by `init_stage` the event corresponding to the execution of the C code "`nav_init_stage();`" and by `reset_time` the event that refers to the C code "`block_time = 0;`". This code is used to re-initialise the parameters of the autopilot before executing a new stage. These parameters may for instance measure the angular increment when a `CIRCLE` is run and will not modify the internal state of the flight plan. Inference rule FP-14, presented in Definition 7.42, details the execution of the *goto_block* function that uses the *change_block* function and then executes the initialisation code.

7.3.3.2 Run stages

The function *run_step* executes all the remaining stages one after the other until reaching a *break* stage or the last stage to execute. We first detail the case where there is no remaining stage to execute in the current block. Then we detail the execution of a non-empty list of stages for each possible `FPL` construct.

Remark

Stages will not always be explicitly classified as *break* or *continue* stages. Indeed, there are stages that are of both types depending on their parameters or on the evaluation of a C condition. For example, the **CALL** stage has a boolean parameter called `break` that defines if the stage should break after its execution. However, it is easy to deduce from the inference rules if a stage behaves as a *break* or *continue* stage depending on whether there is a recursive call to the *run_step* function.

Definition 7.43 - (`run_step`, `src/semantics/FPBigStep.v:254`).

$$\frac{e.\texttt{stages} = [] \qquad e.\texttt{idb} < id_{db}(fp) \qquad e \Uparrow e.\texttt{idb} + 1 \xhookrightarrow[fp]{\text{goto\_block}} e'}{e \xhookrightarrow[fp]{\text{stages}} e'} \text{(FP-15)}$$

$$\frac{e.\texttt{stages} = [] \qquad e.\texttt{idb} \geq id_{db}(fp) \qquad e \Uparrow e.\texttt{idb} \xhookrightarrow[fp]{\text{goto\_block}} e'}{e \xhookrightarrow[fp]{\text{stages}} e'} \text{(FP-16)}$$

An interesting point to notice is that we want to manage the possibility to recover from incorrect states, i.e. state containing block indices that are not correct. Inference rule FP-15 describes what happens when there is no stage to execute but the current block id is smaller than $id_{db}(fp)$. In that case, the current state is correct and the execution will deroute to the next block which is the one following the current block in the flight plan. The next block index is computed by incrementing the current block index. As we ensure that it is strictly smaller than $id_{db}(fp)$, there is no risk of producing an incorrect block index. Otherwise, for any other block index we consider being in the default block and thus we need to normalise the block id and the execution will be derouted to the current block index (c.f. rule FP-16). The *goto_block* function will normalise the block index, producing a correct state referencing the default block and the last position will not be updated. We cannot deroute the current state directly to $id_{db}(fp)$ otherwise the last position will be updated with the current incorrect position (see Definition 7.36).

Remark

Note that this point is one of the properties we verified on the flight plan semantics [WPGT23]. Section 11.3 discusses how we can verify this type of property and what are the guarantees that can be obtained on the generated code.

Definition 7.44 - (while_sem, src/semantics/FPBigStep.v:72).

$$\frac{e.\texttt{stages} = \textbf{WHILE}\ (cond, body) :: stages \qquad evalc\ e\ cond\ =\ (\texttt{true},\ e') \qquad e'\{\texttt{stages} := body +\!\!+ e.\texttt{stages}\}(\!|\texttt{init\_stage}|\!) = e''}{e \xrightarrow[fp]{\text{stages}} e''} \text{(FP-17)}$$

$$\frac{e.\texttt{stages} = \textbf{WHILE}\ (cond, body) :: stages \qquad evalc\ e\ cond\ =\ (\texttt{false}, e') \qquad e'\{\texttt{stages} := stages\} \xrightarrow[fp]{\text{stages}} e''}{e \xrightarrow[fp]{\text{stages}} e''} \text{(FP-18)}$$

The **WHILE** stage executes a list of stages while a condition holds. The condition is evaluated and if it holds, the body of the loop is added at the beginning of the stages list to be executed and the execution is stopped and the final environment is updated (rule FP-17). The **WHILE** stage is kept in the stages list in order to evaluate the condition and possibly iterate the loop after having executed its body. When the loop condition is `false`, the **WHILE** stage is consumed and the execution continues (rule FP-18).

Remark

The `init_stage` code used in rule FP-17 is often used in the following rules. `init_stage` initialises stage parameters and should be used before execution of a new stage. In the semantics, we thus decide to execute this code when a stage finishes its execution and we therefore suppose that before the first execution of the flight plan, this code is executed. However, you can notice that there are some rules, like rule FP-18, that do not execute this code. The placements of `init_stage` code are based on the code generated by the previous generator. As future work, we might consider modifying the generated code and then prove that the `init_stage` code is always executed before the first execution of any stages.

Definition 7.45 - (set_sem, src/semantics/FPBigStep.v:88).

$$\frac{e.\texttt{stages} = \textbf{SET}\ (var,\ value) :: stages \qquad e\{\texttt{stages} := stages\} = e' \qquad e'(\!|var = value;\ \texttt{init\_stage}|\!) \xrightarrow[fp]{\text{stages}} e''}{e \xrightarrow[fp]{\text{stages}} e''} \text{(FP-19)}$$

The **SET** stage (rule FP-19) is a *continue* stage that simply assigns a value to a variable. The assigned value is arbitrary C code and cannot be analyzed. The assignment is thus added in the trace.

Definition 7.46 - (call_sem, src/semantics/FPBigStep.v:94).

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ \mathit{until},\ \texttt{false},\ \texttt{false}) :: \mathit{stages} \qquad e\{\texttt{stages} := \mathit{stages}\} = e' \qquad e'(\!|\mathit{fun};\ \texttt{init\_stage}|\!) \xrightarrow[\mathit{fp}]{\text{stages}} e''}{e \xrightarrow[\mathit{fp}]{\text{stages}} e''} \text{(FP-20)}$$

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ \mathit{until},\ \texttt{false},\ \texttt{true}) :: \mathit{stages} \qquad e\{\texttt{stages} := \mathit{stages}\}(\!|\mathit{fun};\ \texttt{init\_stage}|\!) = e'}{e \xrightarrow[\mathit{fp}]{\text{stages}} e'} \text{(FP-21)}$$

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ \mathit{until},\ \texttt{true},\ \texttt{false}) :: \mathit{stages} \qquad \mathit{evalc}\ e\ \mathit{fun}\ =\ (\texttt{false},\ e') \qquad e'\{\texttt{stages} := \mathit{stages}\} = e'' \qquad e''(\!|\texttt{init\_stage}|\!) \xrightarrow[\mathit{fp}]{\text{stages}} e'''}{e \xrightarrow[\mathit{fp}]{\text{stages}} e'''} \text{(FP-22)}$$

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ \mathit{until},\ \texttt{true},\ \texttt{true}) :: \mathit{stages} \qquad \mathit{evalc}\ e\ \mathit{fun}\ =\ (\texttt{false},\ e') \qquad e'\{\texttt{stages} := \mathit{stages}\}(\!|\texttt{init\_stage}|\!) = e''}{e \xrightarrow[\mathit{fp}]{\text{stages}} e''} \text{(FP-23)}$$

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ \texttt{None},\ \texttt{true},\ \mathit{break}) :: \mathit{stages} \qquad \mathit{evalc}\ e\ \mathit{fun}\ =\ (\texttt{true},\ e')}{e \xrightarrow[\mathit{fp}]{\text{stages}} e'} \text{(FP-24)}$$

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ (\texttt{Some}\ \mathit{cond}),\ \texttt{true},\ \mathit{break}) :: \mathit{stages} \qquad \mathit{evalc}\ e\ \mathit{fun}\ =\ (\texttt{true},\ e') \qquad \mathit{evalc}\ e'\ \mathit{cond}\ =\ (\texttt{true},\ e'') \qquad e''\{\texttt{stages} := \mathit{stages}\}(\!|\texttt{init\_stage}|\!) = e'''}{e \xrightarrow[\mathit{fp}]{\text{stages}} e'''} \text{(FP-25)}$$

$$\frac{e.\texttt{stages} = \textbf{CALL}\ (\mathit{fun},\ (\texttt{Some}\ \mathit{cond}),\ \texttt{true},\ \mathit{break}) :: \mathit{stages} \qquad \mathit{evalc}\ e\ \mathit{fun}\ =\ (\texttt{true},\ e') \qquad \mathit{evalc}\ e'\ \mathit{cond}\ =\ (\texttt{false},\ e'')}{e \xrightarrow[\mathit{fp}]{\text{stages}} e''} \text{(FP-26)}$$

Remember that the call stage is of the form **CALL** (*fun*, *until*, *loop*, *break*). First, let us suppose that the *loop* parameter is set to `false`. The execution of the stage then consists of executing the code *fun* and thus adding the code to the trace. The *break* parameter describes if the stage should be a *break* or *continue* stage. Inference rules FP-20 and FP-21 detail this behaviour.

In the case when the *loop* parameter is set to `true`, the **CALL** stage is executed until the code *fun* returns `false`. Inference rules FP-22 and FP-23 express the case when *fun* returns

`false` and take into account the *break* parameter as previously. Inference rule FP-24 defines the case where *fun* returns `true` and there is no *until* condition. In this case, the stage behaves has a *break* stage and the **`CALL`** stage is not consumed (i.e. at the next call to the `auto_nav` function, the execution will be resumed to the **`CALL`** stage). Finally, if the code *fun* returns `true` and the *until* parameter contains a condition, the **`CALL`** stage will be consumed depending on the result of the evaluation of the condition (c.f. rules FP-25 and FP-26).

Definition 7.47 - (`deroute_sem`, `src/semantics/FPBigStep.v:136`).

$$\frac{e.\texttt{stages} = \textbf{DEROUTE}\ id :: stages \qquad e\{\texttt{stages} := stages\}(\!|\texttt{init\_stage}|\!) = e' \qquad e' \Uparrow id \xhookrightarrow[fp]{\text{goto\_block}} e''}{e \xhookrightarrow[fp]{\text{stages}} e''}\ \text{(FP-27)}$$

The **`DEROUTE`** stage (rule FP-27) is a *break* stage that uses the *goto_block* function to modify the block that must be executed.

Definition 7.48 - (`return_sem`, `src/semantics/FPBigStep.v:142`).

$$\frac{e.\texttt{stages} = \textbf{RETURN}\ reset :: stages \qquad return\_block\ fp\ e\ reset\ =\ e' \qquad e'(\!|\texttt{reset\_time}|\!) = e''}{e \xhookrightarrow[fp]{\text{stages}} e''}\ \text{(FP-28)}$$

The **`RETURN`** stage, defined by the rule FP-28, is a *break* stage that call the *return_block* function to resume the execution into the previously memorised state.

7.3.3.3 Navigation stage

The navigation stage is designed to encapsulate the behaviour of all navigation primitives (`CIRCLE`, `GO`, etc.), as showed in Definition 7.49. Some of them require an initialisation step, described by the *init* parameter in `FP`.

When *init* is set to `true`, the stage requires an initialisation step first, before executing the navigation code. Rule FP-29 describes this behaviour: *nav_init_code* is executed but the navigation stage is not consumed and is modified by setting its *init* parameter to `false`. In the next call to the *step* function, the execution of the navigation stage will continue using the other rules.

In the other case, when *init* is set to `false`, the navigation code can be executed. This execution depends on the presence of a condition to test in the navigation mode. Figure 7.3 presents schematically the C code that must be executed in both cases. The `nav_code_exec` function corresponds to the execution of *nav_code_exec* function. The function `NextStageAndBreak()` computes the next stage id and stops the execution of the *step* function and the other functions correspond to the execution of the trace generated by

Definition 7.49 - (nav_sem, src/semantics/FPBigStep.v:182).

$$\frac{e.\texttt{stages} = \textbf{NAV}\ (mode,\ until,\ \texttt{true}) :: stages \qquad nav\_init\_code\ mode = init \qquad e\{\texttt{stages} := \textbf{NAV}\ (mode,\ until,\ \texttt{false}) :: stages\}(\!|init;\ \texttt{init\_stage}|\!) = e'}{e \xhookrightarrow[fp]{\text{stages}} e'} \text{(FP-29)}$$

$$\frac{e.\texttt{stages} = \textbf{NAV}\ (mode,\ until,\ \texttt{false}) :: stages \qquad nav\_cond\ mode = \texttt{None} \qquad e(\!|pre\_call\_nav\ \ mode|\!)\Uparrow(mode,\ until) \xhookrightarrow[fp]{\text{nav}} e'}{e \xhookrightarrow[fp]{\text{stages}} e'} \text{(FP-30)}$$

$$\frac{e.\texttt{stages} = \textbf{NAV}\ (mode,\ until,\ \texttt{false}) :: stages \qquad nav\_cond\ mode = \texttt{Some}\ cond \qquad evalc\ e(\!|pre\_call\_nav\ mode|\!)\ cond\ =\ (\texttt{true},\ e') \qquad e'\{\texttt{stages} := stages\} = e'' \qquad e''(\!|\ post\_call\_nav\ mode;\ last\_wp\ mode;\ \texttt{init\_stage}|\!) = e'''}{e \xhookrightarrow[fp]{\text{stages}} e'''} \text{(FP-31)}$$

$$\frac{e.\texttt{stages} = \textbf{NAV}\ (mode,\ until,\ \texttt{false}) :: stages \qquad nav\_cond\ mode = \texttt{Some}\ cond \qquad evalc\ e(\!|pre\_call\_nav\ mode|\!)\ cond\ =\ (\texttt{false},\ e') \qquad e'\Uparrow(mode,\ until) \xhookrightarrow[fp]{\text{nav}} e''}{e \xhookrightarrow[fp]{\text{stages}} e''} \text{(FP-32)}$$

If *nav_cond mode* = None:

```
pre_call_nav(mode);
nav_code_exec(mode);
```

If *nav_cond mode* = Some cond:

```
pre_call_nav(mode);
if (cond) {
    last_wp(mode);
    post_call_nav(mode);
    NextStageAndBreak();
}
else {
  nav_code_exec(mode);
}
```

Figure 7.3: C code generated for the navigation *mode*

the function with the same names as defined previously. The post_call_nav function is not always represented in the figure as the *nav_code_exec* function will execute the post navigation code. Inference rule FP-30 presents the first case where there is no condition to

test. As explained in Section 7.3.2, the second case is only used for the execution of the GO mode. Inference rule FP-31 describes the case when the condition is evaluated to true, i.e. the drone reached its destination, and the execution of the stage ends. The rule FP-32 is the other case when the drone has not yet reached its final destination.

Definition 7.50 - (nav_code_sem, src/semantics/FPBigStep.v:160).

$$\frac{e.\mathtt{stages} = \mathbf{NAV}\ (mode,\ until,\ \mathtt{false}) :: stages \qquad until = \mathsf{None} \qquad e(\!|nav\_code\ mode;\ post\_call\_nav\ mode|\!) = e'}{e{\Uparrow}(mode,\ until) \xhookrightarrow[fp]{\text{nav}} e'} \text{(FP-33)}$$

$$\frac{\begin{array}{c}e.\mathtt{stages} = \mathbf{NAV}\ (mode,\ until,\ \mathtt{false}) :: stages \\ until = \mathsf{Some}\ cond \qquad evalc\ e(\!|nav\_code\ mode|\!)\ cond\ =\ (\mathtt{true},\ e') \\ e'\{\mathtt{stages} := stages\} = e'' \qquad e''(\!|post\_call\_nav\ mode;\ \mathtt{init\_stage}|\!) = e'''\end{array}}{e{\Uparrow}(mode,\ until) \xhookrightarrow[fp]{\text{nav}} e'''} \text{(FP-34)}$$

$$\frac{\begin{array}{c}e.\mathtt{stages} = \mathbf{NAV}\ (mode,\ until,\ \mathtt{false}) :: stages \qquad until = \mathsf{Some}\ cond \\ evalc\ e(\!|nav\_code\ mode|\!)\ cond\ =\ (\mathtt{false},\ e') \qquad e'(\!|post\_call\_nav\ mode|\!) = e''\end{array}}{e{\Uparrow}(mode,\ until) \xhookrightarrow[fp]{\text{nav}} e''} \text{(FP-35)}$$

Inference rules FP-33 define the execution of the *nav_code_exec* function when the stage has no *until* condition. The remaining rules specify the execution of the navigation code when there is an *until* condition. If the condition is evaluated to true (rule FP-34), the navigation stage is consumed and the execution of the *step* function stops. In the other case for which the condition is evaluated to false (rule FP-35), the execution stops but the stage is not consumed.

Chapter 8

New features added to FPL

Contents

8.1 Syntax of the new features 169
8.2 Semantics of the new features 170
8.2.1 Forbidden deroutes semantics 170
8.2.2 Updated semantics for block change 172

During the development of the new generator, two new features were added to the language described in Chapter 7. First, another protection mechanism called *forbidden deroute* has been added. Forbidden deroutes are specified by the user to prevent the execution of "dangerous" block changes such as jumping directly to a block where the drone motors are cut off from a block where the drone is airborne. The second feature is a Paparazzi user request about the execution of C code when entering or leaving a block.

Section 8.1 presents the modification made on the FP Gallina structure in order to support these new features. Section 8.2 defines the semantics of the resulting language by detailing the differences from the old semantics and by presenting the new inference rules.

Remark

This chapter describes the version of FPL supported by the new generator. In the following chapters, any references to FPL or the FP structure and its semantics implicitly targets this extended version.

8.1 Syntax of the new features

In order to add the previously presented new features to FPL, the FP structure has been modified. In the following, we present the changes on some flight plan structures presented in Chapter 7.

Definition 8.1 - (flight_plan, src/syntax/FlightPlanGeneric.v:146).

flight_plan ::= {|
  fb_deroutes: list *fp_fb_deroute*
  excpts: list *fp_exception*,
  blocks: list *fp_block*
|}

A flight plan may now contain an arbitrary number of forbidden deroutes through the fb_deroutes field which is a list of forbidden deroutes.

Definition 8.2 - (fp_forbidden_deroute, src/syntax/FlightPlanGeneric.v:23).

fp_fb_deroute::= {|
  from: *block_id*
  to: *block_id*
  only_when: option *c_cond*
|}

A forbidden deroute describes a deroute from a block from to a block to that must be either unconditionally forbidden or watched by a only_when condition. Since conditions are pieces of arbitrary C code seen as the abstract type *c_cond*, we use an option type to represent unconditional deroute with None in order to optimise code generation[1].

Definition 8.3 - (fp_block, src/syntax/FlightPlan.v:48).

fp_block ::= {|

name:	*String*,	excpts:	list *fp_exception*,
id:	*block_id*,	stages:	list *fp_stage*,
pre_call:	option *c_code*,	post_call:	option *c_code*
on_enter:	option *c_code*,	on_exit:	option *c_code*

|}

Finally, we redefine *fp_block* to support the second feature. We add the possibility for users to specify on_enter and on_exit code for every block. This code is executed when entering (resp. exiting) the block during a change of block. The pre_call and post_call C code blocks are different as they are executed at every execution of the function auto_nav when the corresponding block is the current block.

8.2 Semantics of the new features

The modifications made to support the new features only affect the semantics of block change. Indeed, forbidden deroutes prevent some changes of block that may be dangerous. In addition, when the execution is derouted to a new block, the on_exit and on_enter code must be executed. In this section, we first define semantics rules to evaluate if a deroute is forbidden and then we update the rules FP semantics in order to take into account these new features.

8.2.1 Forbidden deroutes semantics

An operation that changes the block being executed may occur through the execution of the auto_nav function. We then want to verify that the flight plan does not forbid the deroute between the current block to the new block to execute. We thus define the *forbidden_deroute* function that verifies if this deroute is forbidden. The presentation of the semantics follows the one used in Chapter 7.

[1] Having a None as the only_when condition, we can avoid the generation of a conditional statement. See Section 9.3.2.2 presenting the C code generated for the test of the forbidden deroutes.

Signature 8.4 - (forbidden_deroute, src/semantics/FPBigStepGeneric.v:193).

$$forbidden\_deroute : flight\_plan \to fp\_env \to block\_id \to (bool \times fp\_env)$$

The property "$forbidden\_deroute\ fp\ e\ to = (res,\ e')$", noted $e \Uparrow to \xhookrightarrow[fp]{\text{fb\_deroute}} e' \Downarrow res$, states that, from an environment e, the test that a deroute from block $e.\texttt{idb}$ to block to is forbidden or is not a *correct block index*, terminates in the state e'. This verification returns the boolean value res that characterises if the deroute is indeed forbidden.

Signature 8.5 - (test_forbidden_deroutes, src/semantics/FPBigStepGeneric.v:173).

$$test\_fb\_deroutes : fp\_env \to block\_id \to block\_id \to \text{list } fp\_fb\_deroute \to (bool \times fp\_env)$$

The property "$test\_fb\_deroutes\ e\ from\ to\ fbds = (res, e')$", noted $e \Uparrow fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}} e' \Downarrow res$, states that the test of the forbidden deroutes list $fbds$, for a deroute from block $from$ to block to starting from environment e terminates in e'. The value res is `true` iff there is an element in $fbds$ that forbids this deroute.

Signature 8.6 - (test_forbidden_deroute, src/semantics/FPBigStepGeneric.v:159).

$$test\_fb\_deroute : fp\_env \to block\_id \to block\_id \to fp\_fb\_deroute \to (bool \times fp\_env)$$

The property "$test\_fb\_deroute\ e\ from\ to\ fbd = (res, e')$", noted $e \Uparrow fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}} e' \Downarrow res$, states that the test regarding the forbidden deroute fbd of a deroute from block $from$ to block to starting from e terminates in environment e'. The value res is `true` iff fbd corresponds to the deroute and if it is enabled (if there is a condition `only_when`, it must be evaluated to `true`).

Definition 8.7 - (forbidden_deroute, src/semantics/FPBigStepGeneric.v:193).

$$\frac{e \Uparrow fp.\texttt{fb\_deroutes} \xhookrightarrow[(e.\texttt{idb},\ to)]{\text{test\_fb\_deroutes}} e' \Downarrow res}{e \Uparrow to \xhookrightarrow[fp]{\text{fb\_deroute}} e' \Downarrow res}\ \text{(FP-36)}$$

Inference rule FP-36 describes the verification when the deroute from the current block to the block to is prevented by one of the forbidden deroutes of the flight plan.

Definition 8.8 - (test_forbidden_deroutes, src/semantics/FPBigStepGeneric.v:173).

$$\frac{}{e \Uparrow [] \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}} e \Downarrow \texttt{false}}\ \text{(FP-37)} \qquad \frac{e \Uparrow fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}} e' \Downarrow \texttt{true}}{e \Uparrow fbd\ ::\ fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}} e' \Downarrow \texttt{true}}\ \text{(FP-38)}$$

$$\frac{e \Uparrow fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}} e' \Downarrow \texttt{false} \qquad e' \Uparrow fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}} e'' \Downarrow res}{e \Uparrow fbd\ ::\ fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}} e'' \Downarrow res}\ \text{(FP-39)}$$

As with exceptions, the analysis of a deroute will go through all the list of deroutes *fp_fb_deroute* by structural induction. Inference rule FP-37 describes the base case when there is no remaining forbidden deroute to test. It thus returns `false`, stating that this deroute is not forbidden. For the recursive case, we test individually the forbidden deroutes in the list. Rule FP-38 presents the case when the deroute is detected as forbidden. We therefore do not have to verify the other forbidden deroutes. Rule FP-39 defines the case when the tested forbidden deroute does not forbid the deroute being verified and then calls recursively the *test_fb_deroutes* function on the rest of the list.

Definition 8.9 - (test_forbidden_deroute, src/semantics/FPBigStepGeneric.v:159).

$$\frac{from \neq fbd.\texttt{from} \vee to \neq fbd.\texttt{to}}{e{\Uparrow}fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}} e{\Downarrow}\texttt{false}} \text{(FP-40)}$$

$$\frac{from = fbd.\texttt{from} \wedge to = fbd.\texttt{to} \qquad fbd.\texttt{only\_when} = \texttt{None}}{e{\Uparrow}fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}} e{\Downarrow}\texttt{true}} \text{(FP-41)}$$

$$\frac{\begin{array}{c} from = fbd.\texttt{from} \wedge to = fbd.\texttt{to} \\ fbd.\texttt{only\_when} = \texttt{Some}\ cond \qquad evalc\ e\ cond = (res,\ e') \end{array}}{e{\Uparrow}fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}} e'{\Downarrow}res} \text{(FP-42)}$$

In order to test if a deroute is forbidden by a *fp_fb_deroute*, we first need to verify if the block id of the *from* and the *to* blocks correspond to the deroute. In the case where they are not equal, the forbidden deroute does not correspond and the deroute is not prevented (c.f. rule FP-40). In the other case where the deroute corresponds, we have to ensure that the forbidden deroute is enabled. If there is no condition (see rule FP-41), it is automatically enabled and the deroute is forbidden. On the other hand, if there is a condition, it must be evaluated. The result of the evaluation thus characterises whether the forbidden deroute is enabled or not (c.f. inference rule FP-42).

8.2.2 Updated semantics for block change

The new features added have an impact when there is a block change. We thus present the updated semantics for the *goto_block* function that changes the block being executed and the inference rule to evaluate the **RETURN** stage.

The *goto_block* function must initially verify if the deroute is allowed. In this case, the function will modify the block executed and will evaluate the appropriate code. First, we define the code that must be executed when the block being executed changes. In order to simplify the notation as `on_enter` and `on_exit` are optional codes, we define the following function:

Definition 8.10 - (opt_c_code_to_trace, src/syntax/FlightPlanGeneric.v:79).

$opt\_c\_code\_to\_trace$: option $c\_code \rightarrow fp\_trace$

$$opt\_c\_code\_to\_trace\ opt\_code ::= \begin{cases} [\texttt{C\_CODE}\ code] & \text{if } opt\_code = \texttt{Some}\ code \\ [\,] & \text{otherwise.} \end{cases}$$

This function transforms any optional C code into the corresponding trace. In case there is no code to execute, then the trace is empty. In the other case, the trace is a singleton list containing the C code. We can now define the *c_change_block* function which describes the code that must be evaluated when the block to execute is changed.

Definition 8.11 - (c_change_block, src/semantics/FPBigStepGeneric.v:148).

c_change_block : $flight\_plan \rightarrow block\_id \rightarrow block\_id \rightarrow fp\_trace$

$$\begin{aligned} c\_change\_block\ fp\ from\ to\ ::=\ & (opt\_c\_code\_to\_trace\ block_{fp}(from).\texttt{on\_exit}) \\ & +\!+ [\texttt{reset\_time}; \texttt{init\_stage}] \\ & +\!+ (opt\_c\_code\_to\_trace\ block_{fp}(to).\texttt{on\_enter}) \end{aligned}$$

The code executed when the block to execute is changed from the block *from* to the block *to* can be divided into 3 parts. First, the execution will exit the block *from*, and thus the `on_exit` code from the *from* block must be evaluated. Then, the code that initialises Paparazzi parameters when the block is changing is executed. Finally, the execution of the flight plan will enter the *to* block. The corresponding `on_enter` code is therefore executed.

The new semantics for the *goto_block* function can now be defined using all the functions previously introduced.

Definition 8.12 - (goto_block, src/semantics/FPBigStepGeneric.v:199).

$$\frac{e \Uparrow id \xrightarrow[fp]{\text{fb\_deroute}} e' \Downarrow \texttt{true}}{e \Uparrow id \xrightarrow[fp]{\text{goto\_block}} e'} \text{(FP-43)}$$

$$\frac{\begin{array}{c} e \Uparrow id \xrightarrow[fp]{\text{fb\_deroute}} e' \Downarrow \texttt{false} \\ change\_block\ fp\ e'\ id\ =\ e'' \qquad e'' (\!| c\_change\_block\ e.\texttt{idb}\ id |\!) = e''' \end{array}}{e \Uparrow id \xrightarrow[fp]{\text{goto\_block}} e'''} \text{(FP-44)}$$

Inference rule FP-43 defines the case when the deroute is forbidden. In this case, the deroute is not proceeded and the block being executed is not modified. However, the environment might have been modified as some external C code, with potential side effect, may have been executed to test the forbidden deroute. Inference rule FP-44 describes the case where the deroute is not forbidden. The environment is modified to change the block that will be executed. The corresponding C code is then executed.

Definition 8.13 - (`return_sem`, `src/semantics/FPBigStep.v:142`).

$$\frac{e.\texttt{stages} = \textbf{RETURN}\ reset :: stages \qquad return\_block\ fp\ e\ reset\ =\ e' \qquad e'(\!|c\_change\_block\ e.\texttt{idb}\ e'.\texttt{idb}|\!) = e''}{e \xrightarrow[fp]{\text{stages}} e''}\ \text{(FP-45)}$$

Inference rule FP-45 describes the new semantics of the **RETURN** stage. The main difference concerns the `on_enter` and `on_exit` code blocks which are now executed.

Coq implementation

During the execution of the flight plan, the user can modify the block being executed through the Ground Control Station (GCS). When the user asks for a block change, the order is sent to the autopilot that executes the *goto_block* function which updates the current state. The main reason of the introduction of the forbidden deroute feature is to prevent user mishandling. For this reason, only the *goto_block* function verifies if the block change is forbidden, not the function concerning the execution of the **RETURN** stage. This behaviour could be added in future work if it is a feature requested by users.

Chapter 9

Architecture of the Gallina generator

Contents

9.1 Extension pass . . . 177
9.1.1 Syntax of FPE . . . 178
9.1.2 FPE environment . . . 181
9.1.3 Well-formed extended flight plan . . . 183
9.1.4 FPE semantics . . . 187
9.2 Size verification pass . . . 199
9.2.1 Well-sized flight plan . . . 199
9.2.2 Size verification function . . . 201
9.3 Clight generation function . . . 202
9.3.1 Number of blocks in a global variable . . . 202
9.3.2 Auxiliary functions generated . . . 203
9.3.3 The auto_nav function . . . 205
9.4 Generator function . . . 208
9.4.1 Global Variables generation . . . 208
9.4.2 Forbidden deroute analysis . . . 210
9.4.3 Flight plan generator function . . . 211

The new flight generator, written in Gallina, translates a FP structure into Clight code. The architecture of the generator is divided into three passes, making semantics preservation proof easier to establish by defining three independent sub-problems. FP and the generated Clight code have several structural differences. The passes of the generator thus perform transformations on the FP structure to reduce these differences before generating the Clight code. In order to understand the transformations applied to the flight plan, we first describe these structural differences.

Figure 9.1 presents the C code generated from a simple FP flight plan with a single block and three stages[1]. The get_nav_block()/get_nav_stage() functions return the value of the nav_block/nav_stage variables, i.e. the ids of the current block/stage being executed. The functions NextBlock and NextStage update the block (resp. stage) index by computing

[1]We present directly the generated C code generated instead of the Clight code as it is easier to read for this informal presentation.

their next value. The implementation of these functions can be found in the common_flight_plan.c file and are presented in Appendix A.2. The other functions are made up just for this example.

```
{|  excpts:= [],
    fb_drtes:= [],
    blocks:= [
      {|
        name:=  "Init_GPS",
        id:= 0,
        excpts:= [],
        stages:= [
          SET "time" "GetTime()";
          WHILE "!GPSFixValid()" [
            CALL "InitSensors()" None
                  false false;
          ]
        ]
        pre_call:= None,
        post_call:= None
      |},
    ]
|}
```

```
void auto_nav(void) {
  switch (get_nav_block()) {
    case 1: // Block 0
      switch (get_nav_stage()) {
        case 0:  time = GetTime(); // SET stage
        case 1:  while_0_1: // WHILE stage
                 if (!(!(GPSFixValid()))) {
                   goto endwhile_0_3;
                 } else {
                   NextStage();
                   break;
                 }
        case 2:  InitSensors(); // CALL stage
        case 3:  goto while_0_1; // WHILE stage
                 endwhile_0_3:
        default: NextBlock();
                 break;
      }
      break; // End Block 0
      ⋮
```

Figure 9.1: The C auto_nav function generated from a simple FPL plan

Remark

This example is not complete as the default block that must be positioned at the end of plan is missing. There are also some statements that have been removed in order to have a minimal example to emphasize the key differences between the original flight plan and the generated C code. The complete C code generated from a similar flight plan is available in Appendix A.1.

The auto_nav function is mainly composed of a switch statement in which every case statement corresponds to the treatment of a block. Each block treatment also consists of a switch statement called *stage switch*. Each case of a stage switch is numbered and represents a part of a stage execution. The previous example uses a **SET** and a **CALL** stages which are *continue* stages, therefore the generated case statements do not contain a break statement. For example, if stage 0 is executed, when the assignment "time = GetTime()" terminates then flight plan execution continues and the **WHILE** stage will be run. Otherwise, *break* stages have a break statement and the stage switch will be exited when encountering such a stage, ending auto_nav execution. This is the case for the **WHILE** stage when the condition is evaluated to true. The next stage is computed with the NextStage function and then the auto_nav function is exited. At the next iteration, the execution will be

resumed in the **CALL** stage, so inside the loop. It is worth noting that the structure of the C code generated for the stages of a block is different from the corresponding FP structure. In the example presented on Figure 9.1, the FP **WHILE** stage which has a nested list of stages with one element is converted into three cases: a case to test the condition, a case that corresponds to the call stage of the loop body and finally a case corresponding to the end of the while statement to (possibly) restart its execution. Another important point to notice is that there is a default case in the *stage switch* which computes the next block that does not correspond to any stage of FP. We thus do not have a direct correspondence between FP stages and the *stage switch* case statements. Notice also that the case statements are numbered while the stages are not.

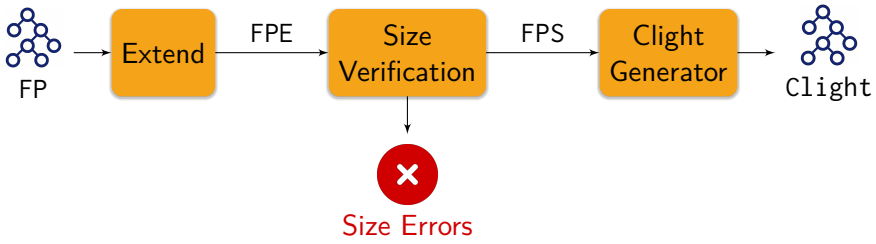

Figure 9.2: Architecture of the Coq Generator.

Example 9.1 shows that there are differences between the FP structure and the generated C code. Figure 6.3 presents the architecture of the new Gallina-based generator that is split into three passes: an *extension* pass, a *size verification* pass and a *generation* pass. The extension pass transforms the FP structure into an *extended* version FPE closer to the C code to be generated. For example, every stage of FPE is numbered and corresponds to a switch case of the C code generated. Section 9.1 presents the new structure FPE with its semantics. This section also presents the transformation realized to transform an FP structure into FPE. The *size verification* pass performs some size verification on the FPE structure and may return errors. One of the goals of this pass is to ensure that the flight plan does not contain more than 256 blocks or stages as the block and stage indices are encoded on 8 bits. If the FPE is well-sized, this pass returns a structure called FPS. The verification pass and FPS are presented in Section 9.2. The generation pass that is detailed in Section 9.3 converts FPS into Clight code. Section 9.4 presents how the different transformation passes are linked and also new analysis passes that provide warning messages for users.

9.1 Extension pass

The extension transforms FP into FPE, an extended flight plan which is an intermediate representation closer to the final C code structure. This structure is defined in the FlightPlanExtended.v Coq file. As presented in the introduction of this chapter, the FP structure contains stages that correspond to several case statements in the generated C

code. Moreover, all stages are referenced with indices in the generated C code but FP does not contain such indices and its semantics proceeds by maintaining a sequence of remaining stages to be executed. FPE has thus been defined to fill this gap between FP and C code.

Section 9.1.1 defines the syntax of the extended flight plan in order to understand the differences with FP. Section 9.1.2 presents the new environments that take advantage of the extended flight plan. Section 9.1.3 describes the transformation function and the dependent type describing a well-formed extended flight plan. Finally, Section 9.1.4 then defines the semantics of well-formed FPE.

9.1.1 Syntax of FPE

The definition of FPE is similar to the definition of FP. The only distinctions are relative to stages which must correspond to the case statements of the C code to be generated.

Definition 9.1 - (flight_plan, src/syntax/FlightPlanGeneric.v:146).

flight_plan$_e$ ::= {|
  fb_deroutes: list *fp_fb_deroute*
  excpts: list *fp_exception*,
  blocks: list *fp_block*$_e$
|}

Coq implementation

As previously explained in Section 7.1, FP and FPE definitions share parts that are defined in files with the suffix Generic such as FlightPlanGeneric.v. The generic files specify module types that must be instantiated to access common definitions. The instantiation of FP and its specific definitions are in files that do not have prefixes, for example, FlightPlan.v file. Similarly, FPE is instantiated in files with the suffix Extended like FlightPlanExtended.v. This convention is used to define the flight plan, as well as the corresponding environments and semantics.

FP and FPE therefore share similar definitions as there are minimal syntactic differences between them. The Coq definitions have thus often the same name. We index the definitions concerning FPE by e to distinguish them from the ones concerning FP. For instance, *structure*$_e$ is the FPE definition of a *structure* defined for FP. We only emphasize the differences between the two versions in the following.

Definition 9.2 - (fp_block, src/syntax/FlightPlanExtended.v:69).

fp_block$_e$ ::= {|

name:	*String*,	excpts:	list *fp_exception*,
id:	*block_id*,	stages:	list *fp_stage*$_e$,
pre_call:	option *c_code*,	post_call:	option *c_code*
on_enter:	option *c_code*,	on_exit:	option *c_code*

|}

The extended version of the block is similar to the FP block with the exception that the stages fields contains fp_stage_e stages.

Definition 9.3 - (fp_stage, src/syntax/FlightPlanExtended.v:40).

fp_stage_e ::=
- | **WHILE**$_e$ (id: *stage_id*) (params: *params_while*)
- | **END_WHILE**$_e$ (id: *stage_id*) (params: *params_while*) (body: list fp_stage_e)
- | **SET**$_e$ (id: *stage_id*) (var: *var_name*) (value: *c_value*)
- | **DEROUTE**$_e$ (id: *stage_id*) (idb: *block_id*)
- | **RETURN**$_e$ (id: *stage_id*) (reset: *bool*)
- | **CALL**$_e$ (id: *stage_id*) (fun: *c_code*) (until: option *c_cond*) (loop: *bool*) (break: *bool*)
- | **NAV_INIT**$_e$ (id: *stage_id*) (mode: *fp_nav_mode*)
- | **NAV**$_e$ (id: *stage_id*) (mode: *fp_nav_mode*) (until: option *c_cond*)
- | **DEFAULT**$_e$ (id: *stage_id*)

Definition 9.4 - (params_while, src/syntax/FlightPlanExtended.v:31).

params_while ::= {|
  while_block_id: *block_id*, cond: *c_cond*,
  while_id: *stage_id*, end_while_id: *stage_id*
|}

fp_stage_e is an inductive type describing the extended flight plan stages. The extended flight plan is obtained using the extend_stages_default function that converts a list of *fp_stage* into a list of fp_stage_e.

Figure 9.3 presents an example of such a transformation. Let us briefly explain the transformations made by the function extend_stages_default. First, all stages are numbered with respect to their declaration order. The stages **CALL**$_e$, **SET**$_e$, **DEROUTE**$_e$ and **RETURN**$_e$ are not modified apart from the id field which has been added. **NAV** stages are converted into **NAV**$_e$ stages with the same mode and until parameters. In the generated C code, the FP navigation stages are translated into a case that executes the navigation code but also another case that executes the initialisation code if the init parameter is set to true. Therefore, the init field of navigation stages is removed by adding a **NAV_INIT**$_e$ stage before **NAV**$_e$ when necessary.

Throughout the transformation, the **WHILE** stages are flattened: they are converted into a list of stages composed of a **WHILE**$_e$ stage, then the body of the loop converted into a fp_stage_e list and finally a **END_WHILE**$_e$ stage. Both **WHILE**$_e$ and **END_WHILE**$_e$ are parametrised with *params_while*, introduced in the Definition 9.4, which contains the loop condition, the id of the corresponding **WHILE**$_e$ and **END_WHILE**$_e$ stages and the id of the block which contains these stages. The stage ids are used in the semantics to know the index of the

A list of FP stages:

```
CALL code None false true;
NAV m1 (Some cond1) true;
WHILE cond2 [
      NAV m2 None false;
];
DEROUTE 1;
```

The generated FPE stages:

```
CALL_e 0 code None false true;
NAV_INIT_e 1 m1 (Some cond1);
NAV_e 2 m1 (Some cond1);
WHILE_e 3
  {| while_block_id:= 1, cond:= cond2,
     while_id:= 3,       end_while_id:= 5 |};
NAV_e 4 m2 None;
END_WHILE_e 5
  {| while_block_id:= 1, cond:= cond2,
     while_id:= 3,       end_while_id:= 5 |}
  [NAV_e 4 m2 None];
DEROUTE_e 6 1;
DEFAULT_e 7;
```

Figure 9.3: Example of stages extension.

stage to execute in order to continue or exit the loop. For similar reasons, the block and stage indices are also used during the code generation to produce unique labels as presented in figure 9.1. For verification purposes (cf. Section 10.2.2), the extended body of the loop is also passed as the body parameter of **END_WHILE**$_e$. Indeed, it is used during the verification of the first pass to recover the body of loop when looking to a **END_WHILE**$_e$ stage and it is not used in the semantics (see Section 9.1.4) or for the code generation.

Finally, the function `extend_stages_default` appends a default stage to the end of the stages list. This stage corresponds to the default case of the *stage switch* that jumps to the next block as shown in figure 9.1.

In order to express properties of the generator and verify them, we often have to access the index of a *fp_stage*$_e$. We thus define the following function.

Definition 9.5 - (get_stage_id, `src/syntax/FlightPlanExtended.v:55`).

get_stage_id : *fp_stage*$_e$ → *stage_id*

This function returns the corresponding stage index. We note *s*.id the result of "*get_stage_id s*".

Remark

In the remaining of the chapter, we use the following convention: the *fp* variable refers to a *flight_plan* and the *fpe* variable refers to a *flight_plan*$_e$.

9.1.2 FPE environment

This section defines the environment structure and some auxiliary functions that will be used when defining FPE semantics. Definition 9.6 presents fp_env_e, an environment that uses the same representation of the drone environment than fp_env by having strictly disjoint parts. We reuse fp_trace to represent the memory space that can be modified when external C code is executed. However, we define a new structure fp_state_e to represent the execution state of the flight plan using fp_stage_e.

Definition 9.6 - (fp_env, src/semantics/FPEnvironmentExtended.v:44).

fp_env_e ::= {|
  state: fp_state_e,
  trace: fp_trace
|}

Definition 9.7 introduces fp_state_e, the state representation for the extended flight plan. As stages are numbered in $flight\_plan_e$, fp_state_e only contains the id of the current stage and the memorised position in the last executed block, instead of a list of remaining stages as in fp_state.

Definition 9.7 - (fp_state, src/semantics/FPEnvironmentExtended.v:35).

fp_state_e ::= {|
  idb: $block\_id$, stage: $stage\_id$,
  lidb: $block\_id$, lstage: $stage\_id$
|}

The initial environment is an empty trace with a state where every index is set to 0.

Definition 9.8 - (init_env, src/semantics/FPEnvironmentExtended.v:108).

$init\_env_e$: $flight\_plan_e$ → fp_env_e
$init\_env_e$ fpe ::= {| state := {| idb :=0, stage := 0,
lidb := 0,lstage := 0 |},
trace := [] |}

Coq implementation

The definition of $init\_env_e$ is independent of the flight plan contrary to fp_env. However, every instantiation of the generic environment, defined in FPEnvironmentGeneric.v file, requires the flight plan in the signature of this function.

We define the $evalc_e$ function that evaluates a condition in a FPE environment. Its definition is equivalent to the $evalc$ function for FP.

Definition 9.9 - (evalc, src/semantics/FPEnvironmentExtended.v:73).

$evalc_e$: $fp\_env_e \rightarrow c\_cond \rightarrow (bool \times fp\_env_e)$
$evalc_e\ e\ cond ::= \texttt{let}\ res := eval\ e.\texttt{trace}\ cond\ \texttt{in}$
$(res, e\{\texttt{trace} := e.\texttt{trace} +\!\!+ [\textbf{COND}\ (cond, res)]\})$

The environment of FPE is close to *fp_env*. We use the same notation, $e(\!|c|\!)$ for the execution of arbitrary C code. As a reminder, this notation only appends the code c being executed to the trace of the environment e. We will also use the same notation $block_{fpe}(n)$ that returns the nth block of the extended flight plan *fpe*. If n is greater than the number of blocks, then we return the default block defined as follows:

Definition 9.10 - (default_block, src/syntax/FlightPlanExtended.v:86).

$default\_block_e$: $block\_id \rightarrow fp\_block_e$
$default\_block_e\ id ::= \{|$
`  name      := "HOME",              id         := id,`
`  stages    := [NAV`$_e$ `0 HOME None],  excpts     := [],`
`  pre_call  := None,                post_call  := None,`
`  on_enter  := None,                on_exit    := None`
$|\}$

The default block for the extended flight plans is similar to the default block for FP as it contains only a stage that is a navigation stage bringing the drone home.

As extended stages are numbered, we define several functions in order to get a specific stage in a flight plan. These functions are similar to the functions we previously defined to access blocks. First, we define the default stage. For now, we suppose that all stages are correctly indexed (in their declaration order) and the default stage is positioned at the end of every block. In section 9.1.3, we will see that this supposition is correct by construction if the the flight plan stages have been extended using the `extend_stages_default` function. The default stage id is therefore the id of the last stage and is computed with the $default\_stage\_id_e$ function.

Definition 9.11 - (default_stage_id, src/syntax/FlightPlanExtended.v:107).

$default\_stage\_id_e$: $flight\_plan_e \rightarrow block\_id \rightarrow stage\_id$
$default\_stage\_id_e\ fpe\ idb ::= (length\ block_{fpe}(idb).\texttt{stages}) - 1$

The default stage index is specific to every block as they may not contain the same number of stages. We now define the function returning the default stage.

Definition 9.12 - (default_stage, src/syntax/FlightPlanExtended.v:109).

$default\_stage_e$: $flight\_plan_e \rightarrow block\_id \rightarrow fp\_stage_e$
$default\_stage_e\ fpe\ idb ::= \textbf{DEFAULT}_e\ (default\_stage\_id_e\ fpe\ idb)$

Definition 9.13 - (get_stage, src/syntax/FlightPlanExtended.v:113).

$$get\_stage_e : flight\_plan_e \to block\_id \to stage\_id \to fp\_stage_e$$

$$get\_stage_e\ \ fpe\ \ idb\ \ ids ::= \begin{cases} \text{s} & \text{if} \quad \begin{array}{l} s \in block_{fpe}(s).\texttt{stages} \\ \quad \wedge\ s.\texttt{id} = ids \end{array} \\ default\_stage_e\ \ fpe\ \ idb & \text{otherwise.} \end{cases}$$

Coq implementation

Similarly to the *get_block* function, the implementation of get_stage_e returns the nth element as we suppose that the stages are correctly numbered.

Finally, we define the $get\_current\_stage_e$ function that gets the current stage of the flight plan *fpe* for the environment e. We note the call to this function $stage^{fpe}(e)$.

Definition 9.14 - (get_current_stage, src/semantics/FPEnvironmentExtended.v:126).

$$get\_current\_stage_e : flight\_plan_e \to fp\_env_e \to fp\_stage_e$$
$$get\_current\_stage_e\ \ fpe\ \ e ::= get\_stage_e\ fpe\ \ e.\texttt{idb}\ \ e.\texttt{ids}$$

9.1.3 Well-formed extended flight plan

The type definition of extended flight plans allows us to express a wide range of possible flight plans. For example, the position of FPE stages and their fields are not constrained. Figure 9.4a presents a possible list of stages in an extended flight plan: the stages indices are in arbitrary orders, the loop stages ($\textbf{WHILE}_e$ and $\textbf{END\_WHILE}_e$) and the default stage are not correctly positioned, and the fields of loop stages are not properly initialised. Figure 9.4b describes an expected list of stages. In this example, the stages are correctly numbered, the loop stages are correctly defined and there is only one default stage that is positioned at the end of the list. We called *well-formed* flight plans the $flight\_plan_e$ that only contains correctly defined lists of stages.

To define FPE semantics, we must only consider plans that are well-formed (for instance, the loop stages must be correctly positioned). Moreover, to generate Clight code the flight plans must be correctly numbered. Therefore, the flight plans must be constrained and flight plans such as the one described on Figure 9.4a should not be considered.

We want to formally specify the set of *well-formed* flight plans defined as a dependent type over $flight\_plan_e$ that respects a wf_fp_e property. This property is satisfied when the extended flight plan is *well-formed*. In the following, we first formally define the property wf_fp_e and then the subset of *well-formed* flight plans. Then we specify the extension function we used to produce these specific flight plans from *flight_plan* plans.

9.1.3.1 Well-formed *while*

The first property that describes a well-formed extended flight plan states that for any $\textbf{WHILE}_e$ stage in the flight plan, its parameter p should be correctly initialised: the block

```
CALL_e 10 code None false true;
NAV 2 m1 (Some cond1);
DEFAULT_e 7;
NAV_e 4 m2 None false;
END_WHILE_e 8
  {| while_block_id:= 1, cond:= cond2,
     end_while_id:= 7,   while_id:= 5|}
  [NAV_e 5 m3 None false];
NAV_INIT_e 1 m4 (Some cond3);
DEROUTE_e 8 1;
WHILE_e 3
  {| while_block_id:= 2, cond:= cond4,
     end_while_id:= 9,   while_id:= 8|};
```

(a) Possible FPE stages

```
CALL_e 0 code None false true;
NAV_INIT_e 1 m1 (Some cond1);
NAV 2 m1 (Some cond1);
WHILE_e 3
  {|while_block_id:= 1, cond:= cond,
    end_while_id:= 5,   while_id:= 3|};
NAV_e 4 m2 None;
END_WHILE_e 5
  {|while_block_id:= 1, cond:= cond,
    end_while_id:= 5,   while_id:= 3|}
  [NAV_e 4 m2 None];
DEROUTE_e 6 1;
DEFAULT_e 7;
```

(b) Expected FPE stages

Figure 9.4: Example of fp_stage_e list.

and stage indices, describing the position of the **WHILE**$_e$ stage, should correspond to the indices stored in p (i.e. p.while_block_id and p.while_id) and the field end_while_id of p is correctly set. The index p.end_while_id should also refer to a correct **END_WHILE**$_e$ stage in the flight plan. This stage should be positioned after the **WHILE**$_e$ stage and its *body* field should contain the correct subsequence of the stage list that corresponds to the body of the loop.

Definition 9.15 - (wf_while, src/syntax/FlightPlanExtended.v:190).

$$
\begin{aligned}
&wf\_while : flight\_plan_e \to Prop\\
&wf\_while\ \ fpe ::= \forall\ idb\ \ ids\ \ ids'\ \ params,\\
&\qquad get\_stage_e\ \ fpe\ \ idb\ \ ids = \textbf{WHILE}_e\ \ ids'\ \ params\\
&\qquad \to\ \ \ ids = params.\texttt{while\_id}\\
&\qquad\quad \wedge\ ids = ids'\\
&\qquad\quad \wedge\ block_{fpe}(idb).\texttt{id} = params.\texttt{while\_block\_id}\\
&\qquad\quad \wedge\ params.\texttt{end\_while\_id} > ids\\
&\qquad\quad \wedge\ get\_stage_e\ \ fpe\ \ idb\ \ params.\texttt{end\_while\_id}\\
&\qquad\qquad = \textbf{END\_WHILE}_e\ \ params.\texttt{end\_while\_id}\ \ params\\
&\qquad\qquad\qquad block_{fpe}(idb).\texttt{stages}[params.\texttt{while\_id}+1\\
&\qquad\qquad\qquad\qquad ::params.\texttt{end\_while\_id}]
\end{aligned}
$$

This definition uses a specific notation for the extraction of a subsequence in a list that can be defined as follows.

Notation 9.16 - Factor of a list.
For any list of elements l, we note $l[n :: m]$ the sublist of l from the $n+1$th element to the mth element. For example, if $l := [1,2,3,4,5]$ then $l[1 :: 3] = [2,3]$.

9.1.3.2 Well-formed *end while*

Definition 9.17 - (wf_end_while, src/syntax/FlightPlanExtended.v:204).

$$
\begin{aligned}
&wf\_end\_while : flight\_plan_e \to Prop \\
&wf\_end\_while\ \ fpe ::= \forall\ idb\ \ ids\ \ ids'\ \ params\ \ block, \\
&\qquad get\_stage_e\ \ fpe\ \ idb\ \ ids = \mathbf{END\_WHILE}_e\ \ ids'\ \ params\ \ block \\
&\qquad \to\ \ \ ids = params.\texttt{end\_while\_id} \\
&\qquad\quad \wedge\ ids = ids' \\
&\qquad\quad \wedge\ block_{fpe}(idb).\texttt{id} = params.\texttt{while\_block\_id} \\
&\qquad\quad \wedge\ get\_stage_e\ \ fpe\ \ idb\ \ params.\texttt{while\_id} \\
&\qquad\qquad = \mathbf{WHILE}_e\ \ params.\texttt{while\_id}\ \ params
\end{aligned}
$$

The second property, called *wf_end_while*, is the dual of *wf_while*. The property specifies that for any $\mathbf{END\_WHILE}_e$ stage in a flight plan, its parameter p must carry correct information. Similarly for the $\mathbf{WHILE}_e$ stage, in the block p.while_block_id of the flight plan, p.end_while_id should be the index of the $\mathbf{END\_WHILE}_e$ stage and p.while_id should refer to the corresponding $\mathbf{WHILE}_e$ of the loop.

9.1.3.3 Well-positioned default stage

The three following properties state that every block of a well-formed flight plan should only contain one $\mathbf{DEFAULT}_e$ stage and that it should be in last position in the block. As a reminder, the function call "$default\_stage\_id_e\ \ fpe\ \ idb$" returns the default index, i.e. the index to the last stage of the block *idb*. The first property *wf_no_default* specifies that any stage that is not the last stage in a block must not be a $\mathbf{DEFAULT}_e$ stage.

Definition 9.18 - (wf_no_default, src/syntax/FlightPlanExtended.v:215).

$$
\begin{aligned}
&wf\_no\_default : flight\_plan_e \to Prop \\
&wf\_no\_default\ \ fpe ::= \forall\ idb\ \ ids, \\
&\qquad ids < default\_stage\_id_e\ \ fpe\ \ idb \\
&\qquad \to \forall\ id,\ get\_stage_e\ \ fpe\ \ idb\ \ ids \neq \mathbf{DEFAULT}_e\ id
\end{aligned}
$$

The second property *wf_default_last* states that the last stage of every block should be the default stage defined by the function $default\_stage_e$.

Definition 9.19 - (wf_default_last, src/syntax/FlightPlanExtended.v:222).

$$
\begin{aligned}
&wf\_default\_last : flight\_plan_e \to Prop \\
&wf\_default\_last\ \ fpe ::= \forall\ idb,\ \ get\_stage_e\ \ fpe\ \ idb\ \ (default\_stage\_id_e\ \ fpe\ \ idb) \\
&\qquad\qquad = default\_stage_e\ \ fpe\ \ idb
\end{aligned}
$$

The third property *wf_stage_gt_0* specifies that every block should contain at least one stage. This property ensures that the default block id computed with the function $default\_stage\_id_e$ is not a negative value, but most importantly, associated with the two previous properties, it guarantes that every block have at least a default stage.

Definition 9.20 - (wf_stages_gt_0, src/syntax/FlightPlanExtended.v:228).

wf_stage_gt_0 : $\mathit{flight\_plan}_e \to \mathit{Prop}$
wf_stage_gt_0 fpe ::= $\forall$ *idb*, *length* $(\mathit{block}_{fpe}(\mathit{idb}).\texttt{stages}) > 0$

9.1.3.4 Well numbered stages

The final property, called *wf_numbering*, states that all the stages that are different from the $\texttt{DEFAULT}_e$ stage, should be well-numbered, i.e. the *id* of each stage should be its index in the stage list of a block. The property that the last default stage is well-numbered can be proven using the *wf_default_last* property.

Definition 9.21 - (wf_numbering, src/syntax/FlightPlanExtended.v:233).

wf_numbering : $\mathit{flight\_plan}_e \to \mathit{Prop}$
wf_numbering fpe ::= $\forall$ *idb ids*,
$(\forall\ n,\ \mathit{get\_stage}_e\ \mathit{fpe}\ \mathit{idb}\ \mathit{ids} \neq \texttt{DEFAULT}_e\ n)$
$\to (\mathit{get\_stage}_e\ \mathit{fpe}\ \mathit{idb}\ \mathit{ids}).\texttt{id} = \mathit{ids}$

9.1.3.5 Well-formed flight plan

We can now define the $\mathit{wf\_fp}_e$ property that defines the well-formedness of a flight plan, gathering the 6 previously defined properties.

Definition 9.22 - (wf_fp_e, src/syntax/FlightPlanExtended.v:239).

$\mathit{wf\_fp}_e$: $\mathit{flight\_plan}_e \to \mathit{Prop}$
$\mathit{wf\_fp}_e$ *fpe* ::= {|

while:	*wf_while fpe*,	end_while:	*wf_end_while fpe*,
no_default:	*wf_no_default fpe*,	default_last:	*wf_default_last fpe*,
stage_gt_0:	*wf_stage_gt_0 fpe*,	numbering:	*wf_numbering fpe*

|}

Finally, we can define $\mathit{wf\_flight\_plan}_e$, the dependent type that corresponds to the subset of $\mathit{flight\_plan}_e$ that respects the property $\mathit{wf\_fp}_e$. This subset describes all extended flight plans that are well-formed.

Definition 9.23 - (flight_plan_wf, src/syntax/FlightPlanExtended.v:248).

$\mathit{wf\_flight\_plan}_e$ ::= { *fpe*: $\mathit{flight\_plan}_e$ | $\mathit{wf\_fp}_e$ *fpe* }

We use the same notation to access the fields of the flight plan even if it is a dependent type. For example *fpe*.blocks returns the list of blocks contained in the flight plan *fpe*.

9.1.3.6 Extension pass

The goal of this section is to define the first pass of the generator that converts any flight plan into a well-formed extended flight plan. The extension pass is realized by the extend_flight_plan function.

Definition 9.24 - (extend_flight_plan, src/generator/FPExtended.v:1313).

extend_flight_plan : *flight_plan* → $wf\_flight\_plan_e$

The extend_flight_plan function is implemented using the extend_stages_default function presented in Section 9.1.1 which numbers the stages and linearizes the list of stages. For instance, the **WHILE** stages are unfolded by appending the loop body to the main list of stages. New stages are also added such as $\textbf{NAV\_INIT}_e$, $\textbf{END\_WHILE}_e$ or $\textbf{DEFAULT}_e$. The stages of the generated flight plan therefore have a direct correspondence to the *switch case* of the generated C code.

An important point to note is that the extend_flight_plan function, returning a $wf\_flight\_plan_e$, produces a $flight\_plan_e$ but also the proof that this flight plan extended is well-formed. This presents two advantages: 1) the extended flight plan produced comes with the proof that is *well-formed*, we thus do not have to carry the hypothesis in the proofs that the flight plan is produced by extend_flight_plan to know for example that it is well-numbered 2) the definition of FPE semantics, presented in the next section, will thus use the specificity of *well-formed* flight plan, e.g. as the default stage is positioned at the end, it will be used to update the states in order to execute the next block.

Remark

The wf_fp_e property does not force any specific positioning for the navigation stages, i.e. a $\textbf{NAV\_INIT}_e$ stage does not have to be followed by a $\textbf{NAV}_e$ stage. Indeed, we considered that the positions of both navigation stages will have no impact on the semantics execution as they will be evaluated independently but also for the Clight generation. This case is different from the loop stages where the semantics suppose that they are correctly initialised or that the default stage is positioned at the end of every block. Obviously, for the semantic preservation proof of the first pass (cf. Section 10.2.2), we have to ensure that the extend_flight_plan function correctly generates the $\textbf{NAV\_INIT}_e$ stage. However, this property is not necessary for the proof of the other passes, unlike the fact that loop stages are correctly initialised or that the stages are correctly numbered.

9.1.4 FPE semantics

FPE semantics reuses some concepts of FP semantics. However, as we only consider *well-formed* extended flight plan, stages are indexed, and thus the evaluation of FPE semantics is closer to the execution of the expected C code. Similarly to the *step* function for FP, FPE semantics is defined by the $step_e$ function that represents the execution of the auto_nav function.

Signature 9.25 - (step, src/semantics/FPBigStepExtended.v:377).

$$step_e : wf\_flight\_plan_e \to fp\_env_e \to fp\_env_e$$

The property "$step_e\ fpe\ e = e'$", noted $e \xhookrightarrow[fpe]{\text{FPE}} e'$, states that e' is the resulting environment after the execution of the extended flight plan *fpe* starting from the environment e.

As FP and FPE have a common structure (except for stages) and share similar functions, we do not represent the functions signature as the only difference is that they use the extended version of flight plans and environments.

Remark

In the following, we use the notation *fpe* that is either an $flight\_plan_e$ or an $wf\_flight\_plan_e$, in order to reuse the function defined previously. Indeed, we introduced functions taking $flight\_plan_e$ that we want to use in the semantics definitions without having to overload the notations with for example (*fpe*).value. However, we explicitly specify in function signatures if the definition requires a *well-formed* flight plan or only an extended flight plan.

Definition 9.26 - (step, src/semantics/FPBigStepExtended.v:377).

$$\frac{e \xhookrightarrow[fpe]{\text{exception}_e} e' \Downarrow \texttt{true}}{e \xhookrightarrow[fpe]{\texttt{FPE}} e'} \text{(FPE-1)}$$

$$\frac{e \xhookrightarrow[fpe]{\text{exception}_e} e' \Downarrow \texttt{false} \qquad e' \xhookrightarrow[fpe]{\text{stages}_e} e'' \qquad block_{fpe}(e'.\texttt{idb}).\texttt{post\_call} = \texttt{None}}{e \xhookrightarrow[fpe]{\texttt{FPE}} e''} \text{(FPE-2)}$$

$$\frac{\begin{array}{c} e \xhookrightarrow[fpe]{\text{exception}_e} e' \Downarrow \texttt{false} \\ e' \xhookrightarrow[fpe]{\text{stages}_e} e'' \qquad block_{fpe}(e'.\texttt{idb}).\texttt{post\_call} = \texttt{Some}\ code \end{array}}{e \xhookrightarrow[fpe]{\texttt{FPE}} e''(\!|code|\!)} \text{(FPE-3)}$$

The rules FPE-1, FPE-2 and FPE-3 are similar to the rules of FP (c.f. rules FP-1, FP-2 and FP-3). These rules state that the execution of the auto_nav function starts by testing the exceptions and, if there is none raised, then the stages are executed.

9.1.4.1 Inferences rules for exceptions

During the extension pass, exceptions are not modified and their semantics is the same as for FP. The following rules about the exceptions are thus the same as the ones defined for FP except that they use the $evalc_e$ and $goto\_block_e$ functions.

Definition 9.27 - (exception, src/semantics/FPBigStepGeneric.v:250).

$$\frac{e \Uparrow fpe.\texttt{excpts} \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e' \Downarrow \texttt{true}}{e \xhookrightarrow[fpe]{\text{exception}_e} e' \Downarrow \texttt{true}} \text{(FPE-4)}$$

$$\frac{e \Uparrow fpe.\texttt{excpts} \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e' \Downarrow \texttt{false} \qquad block_{fpe}(e'.\texttt{idb}).\texttt{pre\_call} = \texttt{None} \qquad e' \Uparrow block_{fp}(e'.\texttt{idb}).\texttt{excpts} \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e'' \Downarrow res}{e \xhookrightarrow[fpe]{\text{exception}_e} e'' \Downarrow res} \text{(FPE-5)}$$

$$\frac{e \Uparrow fpe.\texttt{excpts} \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e' \Downarrow \texttt{false} \qquad block_{fpe}(e'.\texttt{idb}).\texttt{pre\_call} = \texttt{Some}\ code \qquad e' \langle\!| code |\!\rangle \Uparrow block_{fp}(e'.\texttt{idb}).\texttt{excpts} \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e'' \Downarrow res}{e \xhookrightarrow[fpe]{\text{exception}_e} e'' \Downarrow res} \text{(FPE-6)}$$

Definition 9.28 - (test_exceptions, src/semantics/FPBigStepGeneric.v:236).

$$\frac{}{e \Uparrow [] \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e \Downarrow \texttt{false}} \text{(FPE-7)}$$

$$\frac{e \Uparrow ex \xhookrightarrow[fpe]{\text{test\_exception}_e} e' \Downarrow \texttt{true}}{e \Uparrow ex :: exs \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e' \Downarrow \texttt{true}} \text{(FPE-8)}$$

$$\frac{e \Uparrow ex \xhookrightarrow[fp]{\text{test\_exception}} e' \Downarrow \texttt{false} \qquad e' \Uparrow exs \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e'' \Downarrow res}{e \Uparrow ex :: exs \xhookrightarrow[fpe]{\text{test\_exceptions}_e} e'' \Downarrow res} \text{(FPE-9)}$$

Definition 9.29 - (test_exception, src/semantics/FPBigStepGeneric.v:217).

$$\frac{e.\texttt{idb} = ex.\texttt{id}}{e \Uparrow ex \xrightarrow[fpe]{\text{test\_exception}_e} e \Downarrow \texttt{false}} \text{(FPE-10)}$$

$$\frac{e'.\texttt{idb} \neq ex.\texttt{id} \qquad evalc_e \;\; e \;\; ex.\texttt{cond} \;=\; (\texttt{false},\, e')}{e \Uparrow ex \xrightarrow[fpe]{\text{test\_exception}_e} e' \Downarrow \texttt{false}} \text{(FPE-11)}$$

$$\frac{\begin{array}{c} e'.\texttt{idb} \neq ex.\texttt{id} \\ evalc_e \;\; e \;\; ex.\texttt{cond} \;=\; (\texttt{true},\, e') \qquad ex.\texttt{exec} \;=\; \texttt{None} \qquad e' \Uparrow id \xrightarrow[fpe]{\text{goto\_block}_e} e'' \end{array}}{e \Uparrow ex \xrightarrow[fpe]{\text{test\_exception}_e} e'' \Downarrow \texttt{true}} \text{(FPE-12)}$$

$$\frac{\begin{array}{c} e'.\texttt{idb} \neq ex.\texttt{id} \qquad evalc_e \;\; e \;\; ex.\texttt{cond} \;=\; (\texttt{true},\, e') \\ ex.\texttt{exec} \;=\; \texttt{Some}\; code \qquad e' (\!| code |\!) \Uparrow ex.\texttt{id} \xrightarrow[fpe]{\text{goto\_block}_e} e'' \end{array}}{e \Uparrow ex \xrightarrow[fpe]{\text{test\_exception}_e} e'' \Downarrow \texttt{true}} \text{(FPE-13)}$$

9.1.4.2 Inference rules for block changing

The inferences rules for block changing are also similar to the ones defined for FP semantics. We just have to define an extended version of the *change_block*$_e$ function. This function is semantically equivalent to the *change_block* function but the field stage is now initialised to 0 (instead of the list of all the stages of the new block to be executed).

Definition 9.30 - (change_block, src/semantics/FPEnvironmentExtended.v:80).

change_block$_e$: *flight_plan*$_e$ $\rightarrow$ *fp_env*$_e$ $\rightarrow$ *block_id* $\rightarrow$ *fp_env*$_e$

$$\begin{array}{rl} change\_block_e\ fpe\ e\ id\ ::= \{|\ \texttt{state} := \{|\ \texttt{idb} & := \|id\|_{fpe}, \\ & \texttt{stage} \;:= 0, \\ & \texttt{lidb} \;:= e.\texttt{lidb}, \\ & \texttt{lstage} := \begin{cases} e.\texttt{lstage} & \text{if } e.\texttt{idb} = id \\ e.\texttt{stage} & \text{otherwise.} \end{cases} \\ & |\}, \\ \texttt{trace} := & e.\texttt{trace}\ |\} \end{array}$$

Like in FP semantics, the *goto_block* function verifies first if the deroute is forbidden (c.f. rule FPE-14). When the deroute is not forbidden, the environment is updated with the new block that must be executed (c.f. rul FPE-15). Similarly to the test of exceptions, the test of forbidden deroutes is equivalent to the one defined for FP with the *evalc* function replaced by *evalc*$_e$.

Definition 9.31 - (goto_block, src/semantics/FPBigStepGeneric.v:199).

$$\frac{e \Uparrow id \xhookrightarrow[fpe]{\text{fb\_deroute}_e} e' \Downarrow \texttt{true}}{e \Uparrow id \xhookrightarrow[fpe]{\text{goto\_block}_e} e'} \text{(FPE-14)}$$

$$\frac{\begin{array}{c} e \Uparrow id \xhookrightarrow[fpe]{\text{fb\_deroute}_e} e' \Downarrow \texttt{false} \\ \textit{change\_block}_e\ fpe\ e'\ id\ =\ e'' \qquad e'' (\!| \textit{c\_change\_block}_e\ e.\texttt{idb}\ id |\!) = e''' \end{array}}{e \Uparrow id \xhookrightarrow[fpe]{\text{goto\_block}_e} e'''} \text{(FPE-15)}$$

Definition 9.32 - (forbidden_deroute, src/semantics/FPBigStepGeneric.v:193).

$$\frac{e \Uparrow fpe.\texttt{fb\_deroutes} \xhookrightarrow[(e.\texttt{idb},\ to)]{\text{test\_fb\_deroutes}_e} e' \Downarrow res}{e \Uparrow to \xhookrightarrow[fpe]{\text{fb\_deroute}_e} e' \Downarrow res} \text{(FPE-16)}$$

Definition 9.33 - (test_forbidden_deroutes, src/semantics/FPBigStepGeneric.v:173).

$$\frac{}{e \Uparrow [] \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}_e} e \Downarrow \texttt{false}} \text{(FPE-17)}$$

$$\frac{e \Uparrow fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}_e} e' \Downarrow \texttt{true}}{e \Uparrow fbd\ ::\ fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}_e} e' \Downarrow \texttt{true}} \text{(FPE-18)}$$

$$\frac{e \Uparrow fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}_e} e' \Downarrow \texttt{false} \qquad e' \Uparrow fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}_e} e'' \Downarrow res}{e \Uparrow fbd\ ::\ fbds \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroutes}_e} e'' \Downarrow res} \text{(FPE-19)}$$

Definition 9.34 - (test_forbidden_deroute, src/semantics/FPBigStepGeneric.v:159).

$$\frac{from \neq fbd.\texttt{from} \vee to \neq fbd.\texttt{to}}{e{\Uparrow}fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}_e} e{\Downarrow}\texttt{false}} \text{(FPE-20)}$$

$$\frac{from = fbd.\texttt{from} \wedge to = fbd.\texttt{to} \qquad fbd.\texttt{only\_when} = \texttt{None}}{e{\Uparrow}fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}_e} e{\Downarrow}\texttt{true}} \text{(FPE-21)}$$

$$\frac{\begin{array}{c}from = fbd.\texttt{from} \wedge to = fbd.\texttt{to} \\ fbd.\texttt{only\_when} = \texttt{Some}\ cond \qquad evalc_e\ e\ cond = (res,\ e')\end{array}}{e{\Uparrow}fbd \xhookrightarrow[(from,\ to)]{\text{test\_fb\_deroute}_e} e'{\Downarrow}res} \text{(FPE-22)}$$

9.1.4.3 Run stages

The main difference between FP and FPE semantics pertains to stages execution. In FP semantics, the remaining stages in the current environment are executed until a *break stage* is reached. The FPE semantics is different as every fp_env_e e environment refers only to an atomic stage that corresponds to a case statement in the generated C code. The $stage^{fpe}(e)$ function returns the corresponding stage in the flight plan *fpe*. The execution thus consists in running this corresponding stage through the *run_stage* function. This function, in addition to executing the current stage and therefore updating the current stage, returns a boolean value that states whether the execution of the flight plan should continue or break.

Signature 9.35 - (run_stage, src/semantics/FPBigStepExtended.v:274).

$$run\_stage : wf\_flight\_plan_e \to fp\_env_e \to (bool \times fp\_env_e)$$

The property "$run\_stage\ fpe\ e = (res, e')$", noted $e \xhookrightarrow[fpe]{\text{stage}}_e id{\Downarrow}e'$, states that e' is the final state after executing the stage $stage^{fpe}(e)$ of the flight plan *fpe*. The result *res* states whether the stage executed is a continue stage or not.

We can now define the inference rules of the *run_step* function that executes the stages until reaching a break stage.

Definition 9.36 - (run_step, src/semantics/FPBigStepExtended.v:289).

$$\frac{e \xhookrightarrow[fpe]{\text{stage}}_e e'{\Downarrow}\texttt{false}}{e \xhookrightarrow[fpe]{\text{stages}_e} e'} \text{(FPE-23)}$$

$$\frac{e \xhookrightarrow[fpe]{\text{stage}}_e e'{\Downarrow}\texttt{true} \qquad e' \xhookrightarrow[fpe]{\text{stages}_e} e''}{e \xhookrightarrow[fpe]{\text{stages}_e} e''} \text{(FPE-24)}$$

The rules FPE-23 describes the case when the `auto_nav` function should break, after having executed the current stage. Otherwise, the rule FPE-24 presents what happens when the execution should continue after the execution of the current extended stage. The *run_step* function thus only executes the stages and does not verify if the execution reaches the end of the block. Indeed, the block change is now performed during the execution of the **DEFAULT**$_e$ stage.

The implementation of the function *run_stage* identifies the current stage and executes the corresponding semantics. For example, if the current stage is a **WHILE**$_e$ stage, it will execute the *while_sem* function that returns the new environment after the execution of this stage and the corresponding boolean indicating whether the execution should terminate. Every extended stage has its own semantics function. We present now the inference rules for each stage.

Definition 9.37 - (while_sem, src/semantics/FPBigStepExtended.v:70).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{WHILE}_e\ (ids,\ params) \\ evalc_e\ \ e\ \ params.\texttt{cond}\ =\ (\texttt{true},\ e') \qquad e'\{\texttt{ids} := ids + 1\}(\!|\texttt{init\_stage}|\!) = e'' \end{array}}{e \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{false}} \text{(FPE-25)}$$

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{WHILE}_e\ (ids,\ params) \\ evalc_e\ \ e\ \ cond\ =\ (\texttt{false}, e') \qquad e'\{\texttt{ids} := params.\texttt{end\_while\_id} + 1\} = e'' \end{array}}{e \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{true}} \text{(FPE-26)}$$

Definition 9.38 - (end_while_sem, src/semantics/FPBigStepExtended.v:85).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{END\_WHILE}_e\ (ids,\ params,\ body) \\ e\{\texttt{ids} := params.\texttt{while\_id}\} = e' \qquad e' \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow res \end{array}}{e \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow res} \text{(FPE-27)}$$

As the stages are correctly extended and numbered in `FPE`, the **WHILE**$_e$ and **END_WHILE**$_e$ stages both contain their own position stored in parameter *params*. When the condition is evaluated to `true`, the rule FPE-25 shows that the execution stops in the next stage that corresponds to the first stage of the loop body. When the condition is evaluated to `false` it jumps at the end of the loop (the stage following the **END_WHILE**$_e$ stage of this loop), and continues the execution (rule FPE-26). When an **END_WHILE**$_e$ stage is reached, the execution continues from the **WHILE**$_e$ stage and thus the condition will be evaluated using the **WHILE**$_e$ inference rules (c.f. rule FPE-27).

Remark

The semantics supposes that *params* is correctly initialised, otherwise the loop will not execute correctly. This is one case where it is required to evaluate the semantics on a *well-formed* flight plan extended.

Definition 9.39 - (set_sem, src/semantics/FPBigStepExtended.v:91).

$$\frac{stage^{fpe}(e) = \mathbf{SET}_e\ (ids,\ var,\ value) \qquad e(\!|var = value;\ \texttt{init\_stage}|\!) = e' \qquad e'\{\texttt{ids} := ids + 1\} = e''}{e \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{true}} \text{(FPE-28)}$$

The rule FPE-28 corresponds to the execution of the $\mathbf{SET}_e$ stage. The main difference between FP and FPE semantics is how the next stage is computed. FP uses a list of the remaining stages to be executed whereas FPE simply increments the current stage index.

Definition 9.40 - (call_sem, src/semantics/FPBigStepExtended.v:97).

$$\frac{stage^{fpe}(e) = \mathbf{CALL}_e\ (ids,\ fun,\ until,\ \texttt{false},\ \texttt{false}) \qquad e\{\texttt{ids} := ids + 1\} = e' \qquad e'(\!|fun;\ \texttt{init\_stage}|\!) = e''}{e \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{true}} \text{(FPE-29)}$$

$$\frac{stage^{fpe}(e) = \mathbf{CALL}_e\ (ids,\ fun,\ until,\ \texttt{false},\ \texttt{true}) \qquad e\{\texttt{ids} := ids + 1\}(\!|fun;\ \texttt{init\_stage}|\!) = e'}{e \xrightarrow[fpe]{\text{stage}}_e e' \Downarrow \texttt{false}} \text{(FPE-30)}$$

$$\frac{stage^{fpe}(e) = \mathbf{CALL}_e\ (ids,\ fun,\ until,\ \texttt{true},\ \texttt{false}) \qquad evalc_e\ \ e\ \ fun\ =\ (\texttt{false},\ e') \qquad e'\{\texttt{ids} := ids + 1\}(\!|\texttt{init\_stage}|\!) = e'''}{e \xrightarrow[fpe]{\text{stage}}_e e''' \Downarrow \texttt{true}} \text{(FPE-31)}$$

$$\frac{stage^{fpe}(e) = \mathbf{CALL}_e\ (ids,\ fun,\ until,\ \texttt{true},\ \texttt{true}) \qquad evalc_e\ \ e\ \ fun\ =\ (\texttt{false},\ e') \qquad e'\{\texttt{ids} := ids + 1\}(\!|\texttt{init\_stage}|\!) = e''}{e \xrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{false}} \text{(FPE-32)}$$

$$\frac{stage^{fpe}(e) = \texttt{CALL}_e\ (ids,\ fun,\ \texttt{None},\ \texttt{true},\ break) \qquad evalc_e\ \ e\ \ fun\ =\ (\texttt{true},\ e')}{e \xhookrightarrow[fpe]{\text{stage}}_e e' \Downarrow \texttt{false}}\ \text{(FPE-33)}$$

$$\frac{\begin{array}{c} stage^{fpe}(e) = \texttt{CALL}_e\ (ids,\ fun,\ (\texttt{Some}\ cond),\ \texttt{true},\ break) \\ evalc_e\ \ e\ \ fun\ =\ (\texttt{true},\ e') \\ evalc_e\ \ e'\ \ cond\ =\ (\texttt{true},\ e'') \qquad e''\{\texttt{ids} := ids + 1\}(\!|\texttt{init\_stage}|\!) = e''' \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e e''' \Downarrow \texttt{false}}\ \text{(FPE-34)}$$

$$\frac{\begin{array}{c} stage^{fpe}(e) = \texttt{CALL}_e\ (ids,\ fun,\ (\texttt{Some}\ cond),\ \texttt{true},\ break) \\ evalc_e\ \ e\ \ fun\ =\ (\texttt{true},\ e') \qquad evalc_e\ \ e'\ \ cond\ =\ (\texttt{false},\ e'') \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{false}}\ \text{(FPE-35)}$$

The semantics rules for the $\texttt{CALL}_e$ stage are similar to those for the FP semantics.

Definition 9.41 - (deroute_sem, src/semantics/FPBigStepExtended.v:141).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \texttt{DEROUTE}_e\ (ids,\ idb) \\ e\{\texttt{ids} := ids + 1\}(\!|\texttt{init\_stage}|\!) = e' \qquad e' \Uparrow idb \xhookrightarrow[fpe]{\text{goto\_block}_e} e'' \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e e'' \Downarrow \texttt{false}}\ \text{(FPE-36)}$$

The rule FPE-36 presents the semantics of the $\texttt{DEROUTE}_e$ stage. This stage computes the next stage by incrementing the current stage index and then calling the $goto\_block_e$ function to change the block being executed. Finally, the function returns the newly computed environment and the boolean value `false` as it is a break stage.

Remark

The computation of the next stage is important as during the deroute, the current position (i.e. the block and stage indices) is memorised as the last position. This last position is used when a $\texttt{RETURN}_e$ stage is executed to resume the execution in the memorised position. If the next stage is not computed before the deroute, then after a $\texttt{RETURN}_e$, the execution will be resumed on the initial $\texttt{DEROUTE}_e$ that will re-deroute the flight plan instead of continuing the execution of the current block.

Definition 9.42 - (return_sem, src/semantics/FPBigStepExtended.v:149).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{RETURN}_e\ (ids,\ reset) \\ return\_block_e\ \ fp\ \ e\ \ reset\ =\ e' \qquad e'(\!|c\_change\_block_e\ e.\texttt{idb}\ e'.\texttt{idb}|\!) = e'' \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e e''\Downarrow\texttt{false}}\ \text{(FPE-37)}$$

The execution of the **RETURN**$_e$ stage presented in rule FPE-37 is identical to the FP semantics. We just have to adjust the definition of the $return\_block_e$ function for the fp_env_e.

Definition 9.43 - (return_block, src/semantics/FPEnvironmentExtended.v:185).

$return\_block_e : flight\_plan_e \to fp\_env_e \to bool \to fp\_env_e$

$$\begin{array}{l} return\_block_e\ fpe\ e\ reset\ ::= \{|\ \texttt{state} := \{|\ \texttt{idb} := e.\texttt{lidb}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{stage} := \begin{cases} 0 & \text{if } reset \\ e.\texttt{lstage} & \text{otherwise.} \end{cases}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{lidb} := e.\texttt{lidb}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{lstage} := e.\texttt{lstage}\ |\}, \\ \qquad\qquad\qquad\qquad\qquad \texttt{trace} := \quad e.\texttt{trace}\ |\} \end{array}$$

As there is a specific init stage, the execution of the navigation stage for FPE is splitted into the execution of the initalisation code and the execution of the navigation code.

Definition 9.44 - (nav_init_sem, src/semantics/FPBigStepExtended.v:158).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{NAV\_INIT}_e\ (ids, mode) \\ nav\_init\_code\ mode = init \qquad e\{\texttt{ids} := ids + 1\}(\!|init;\ \texttt{init\_stage}|\!) = e' \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e e'\Downarrow\texttt{false}}\ \text{(FPE-38)}$$

The inference rule FPE-38 describes the execution of the initialisation code for the navigation stage. The nav_init_code, as for the other functions that generate navigation code, are strictly identical to for the FP semantics as they only depend on the navigation mode that has not been modified during the flight plan extension pass.

The inference rules for the **NAV**$_e$ stage require the definition of $nav\_code\_exec_e$. This function behaves like the nav_code_exec function but also returns the boolean value stating whether the execution should continue.

Signature 9.45 - (nav_code_sem, src/semantics/FPBigStepExtended.v:165).

$$nav\_code\_exec_e : flight\_plan_e \to fp\_env_e \to fp\_nav\_mode \to \text{option } c\_cond \to (bool \times fp\_env_e)$$

The property "$nav\_code\_exec_e\ fpe\ e\ mode\ until = (res, e')$", noted $e{\Uparrow}(mode,\ until) \xrightarrow[fpe]{\text{nav}}_e e'{\Downarrow}res$, states that the execution of the $\mathbf{NAV}_e$ stage with parameters *mode until* from the state e terminates in the state e'. The returned value is equal to `false` if the execution of the `auto_nav` function should break.

The rest of the inference rules for $\mathbf{NAV}_e$ stage and $nav\_code\_exec_e$ function are equivalent to the `FP` semantics.

Definition 9.46 - (nav_sem, src/semantics/FPBigStepExtended.v:188).

$$\frac{stage^{fpe}(e) = \mathbf{NAV}_e\ (ids,\ mode,\ until) \qquad nav\_cond\ mode = \mathsf{None} \qquad e(\!|pre\_call\_nav\ \ mode|\!){\Uparrow}(mode,\ until) \xrightarrow[fpe]{\text{nav}}_e e'{\Downarrow}res}{e \xrightarrow[fpe]{\text{stage}}_e e'{\Downarrow}res}\ \text{(FPE-39)}$$

$$\frac{\begin{array}{c}stage^{fpe}(e) = \mathbf{NAV}_e\ (ids,\ mode,\ until) \qquad nav\_cond\ mode = \mathsf{Some}\ cond \\ evalc_e\ \ e(\!|pre\_call\_nav\ mode|\!)\ \ cond\ =\ (\mathsf{true},\ e') \qquad e'\{\mathsf{ids} := ids + 1\} = e'' \\ e''(\!|post\_call\_nav\ mode;\ last\_wp\ mode;\ \mathsf{init\_stage}|\!) = e'''\end{array}}{e \xrightarrow[fpe]{\text{stage}}_e e'''{\Downarrow}\mathsf{false}}\ \text{(FPE-40)}$$

$$\frac{\begin{array}{c}stage^{fpe}(e) = \mathbf{NAV}_e\ (ids,\ mode,\ until) \qquad nav\_cond\ mode = \mathsf{Some}\ cond \\ evalc_e\ \ e(\!|pre\_call\_nav\ mode|\!)\ \ cond\ =\ (\mathsf{false},\ e') \\ e'{\Uparrow}(mode,\ until) \xrightarrow[fpe]{\text{nav}}_e e''{\Downarrow}res\end{array}}{e \xrightarrow[fpe]{\text{stage}}_e e''{\Downarrow}res}\ \text{(FPE-41)}$$

Definition 9.47 - (nav_code_sem, src/semantics/FPBigStepExtended.v:165).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{NAV}_e\ (ids,\ mode,\ until) \\ until = \mathsf{None} \qquad e(\!|nav\_code\ mode;\ post\_call\_nav\ mode|\!) = e' \end{array}}{e{\Uparrow}(mode,\ until) \xhookrightarrow[fpe]{\text{nav}}_e\ e'{\Downarrow}\mathsf{false}}\ \text{(FPE-42)}$$

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{NAV}_e\ (ids,\ mode,\ until) \\ until = \mathsf{Some}\ cond \qquad evalc_e\ \ e(\!|nav\_code\ mode|\!)\ \ cond\ =\ (\mathsf{true},\ e') \\ e'\{\mathsf{ids} := ids + 1\} = e'' \qquad e''(\!|post\_call\_nav\ mode;\ \mathsf{init\_stage}|\!) = e''' \end{array}}{e{\Uparrow}(mode,\ until) \xhookrightarrow[fpe]{\text{nav}}_e\ e'''{\Downarrow}\mathsf{false}}\ \text{(FPE-43)}$$

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{NAV}_e\ (ids,\ mode,\ until) \qquad until = \mathsf{Some}\ cond \\ evalc_e\ \ e(\!|nav\_code\ mode|\!)\ \ cond\ =\ (\mathsf{false},\ e') \qquad e'(\!|post\_call\_nav\ mode|\!) = e'' \end{array}}{e{\Uparrow}(mode,\ until) \xhookrightarrow[fpe]{\text{nav}}_e\ e''{\Downarrow}\mathsf{false}}\ \text{(FPE-44)}$$

Definition 9.48 - (default_sem, src/semantics/FPBigStepExtended.v:213).

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{DEFAULT}_e\ ids \qquad e\{\mathsf{ids} := default\_stage\_id_e\ \ fpe\ \ e.\mathsf{idb}\} = e' \\ e'.\mathsf{idb} < id_{db}(fpe) \qquad e'{\Uparrow}e.\mathsf{idb} + 1 \xrightarrow[fpe]{\text{goto\_block}_e} e'' \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e\ e''{\Downarrow}\mathsf{false}}\ \text{(FPE-45)}$$

$$\frac{\begin{array}{c} stage^{fpe}(e) = \textbf{DEFAULT}_e\ ids \qquad e\{\mathsf{ids} := default\_stage\_id_e\ \ fpe\ \ e.\mathsf{idb}\} = e' \\ e'.\mathsf{idb} \geq id_{db}(fpe) \qquad e'{\Uparrow}e.\mathsf{idb} \xrightarrow[fpe]{\text{goto\_block}_e} e'' \end{array}}{e \xhookrightarrow[fpe]{\text{stage}}_e\ e''{\Downarrow}\mathsf{false}}\ \text{(FPE-46)}$$

Finally, the $\textbf{DEFAULT}_e$ stage has only to compute the next block that should be executed. Inference rule FPE-45 corresponds to the case when the next block index can be computed by incrementing the current index while preserving a correct state. Otherwise, inference rule FPE-46 deals with the case when the current block index is either the default block index or an incorrect one. In both cases, the execution is derouted to the current block index and the function *goto_index* normalises the block index, similarly to the FP semantics.

9.2 Size verification pass

During the preprocessing phase, flight plans that are syntactically incorrect are rejected, but no guarantee is obtained on the block numbering process at the Coq level. Moreover, the generated C code is aimed to be compiled and executed on drones with limited resources. In order to optimise memory usage, integer variables like `nav_block` that contains the current block id are coded on 8 bits, therefore limiting the number of blocks and stages to 256. In general, this limitation is not reached in a real use case, but it is important to have errors raised when this happens. We thus want to define a new subset of the well-formed extended flight plans. The plans belonging to this subset are called *well-sized* flight plans. These flight plans are produced during this size verification pass and should respect the size restriction imposed in order to prevent overflows or errors in the Clight code generated. First, we formally define the well-sized flight plan. We then present the size verification function.

9.2.1 Well-sized flight plan

All navigation variables in the C code used for the reference of block or stage index are coded on 8 bits. We therefore want to manipulate only natural numbers that can be represented on 8 bits, i.e. that there are smaller than 256. The property *is_nat8* is thus defined to specify this type of value.

Definition 9.49 - (`is_nat8`, `src/common/CommonLemmasNat8.v:26`).

$is\_nat8 : nat \to Prop$
$is\_nat8\ \ n ::= n < 256$

In some cases, the block index number should be more restricted. As presented in Chapter 6, in the XML flight plan the user refers to blocks using their names and the preprocessor converts them into indices. This initial flight plan does not contain a default block. All the references to block indices in the flight plan should thus be *user block indices*, i.e. *correct block indices*[2] except the index of the default block, as the user cannot refer to this block. We thus define the property *is_user_id* stating whether an index is a *user block index*.

Definition 9.50 - (`is_user_id`, `src/syntax/FlightPlanSized.v:32`).

$is\_user\_id : flight\_plan_e \to nat \to Prop$
$is\_user\_id\ \ fpe\ \ n ::= n < id_{db}(fpe)$

The *user block index* property is a stronger constraint than the 8 bits value as the well-sized flight plans are limited to 256 blocks ($user\ block\ index < number\ of\ blocks < 256$). This restriction will be defined later in definition 9.52.

The definition of the property stating the *well-sized* property for flight plan, requires first to describe the well-sized blocks specified by the property *well_sized_block*.

[2]Correct block index is defined in definition 7.11 and states that the index corresponds exactly to the index of a block.

Definition 9.51 - (well_sized_block, src/syntax/FlightPlanSized.v:84).

well_sized_block: $fp\_block_e \rightarrow Prop$
well_sized_block $b ::= \{|$
  nb_stage8: *is_nat8* (*length* b.stages),
  correct_local_excpts: $\forall\ e \in b$.excepts, *is_user_id* e.id,
  correct_deroutes: $\forall\ s \in b$.stages *ids idb*, $s =$ **DEROUTE**$_e$ *ids idb* $\rightarrow$ *is_user_id idb*
$|\}$

The property *well_sized_block* is satisfied for a fp_block_e if it has less than 256 stages (fields nb_stage8), the local exceptions and also deroutes stages contain block indices that are *user block indices* (cf. respectively fields correct_local_excpts and correct_deroutes).

Definition 9.52 - (verified_fp_e, src/syntax/FlightPlanSized.v:103).

verified_fp$_e$: $flight\_plan_e \rightarrow Prop$
verified_fp$_e$ *fpe* $::= \{|$
  nb_block8: *is_nat8* ((*length fpe*.blocks) + 1)
  correct_global_excpts: $\forall\ e \in fpe$.excepts, *is_user_id* e.id,
  correct_fbds: $\forall\ fbd \in fpe$.fb_deroutes,
    is_user_id fbd.from $\wedge$ *is_user_id* fbd.to
  well_size_blocks: $\forall\ i$, *well_sized_block* ($block_{fpe}(i)$),
  well_numbered_blocks: $\forall\ i, i \leq id_{db}(fpe) \rightarrow block_{fpe}(i)$.id $= i$
$|\}$

We can now define the *verified_fp*$_e$ property stating whether a $wf\_flight\_plan_e$ is well-sized. First, a flight plan is well-sized if there is less than 256 accessible blocks (field nb_block8). As a reminder, the default block is not stored in the flight plan but it is implicitly positioned at the end of the blocks list. The number of blocks is thus the number of blocks stored in the flight plan plus 1. Similarly to the local exceptions, the id of the global exceptions should be *user block indices* (field correct_global_excpts). Furthermore, the field correct_fbds states that all the block indices stored in forbidden deroutes should also be *user block indices*. Finally, every block must be well-sized and also well-numbered as expected by the semantics (see Section 7.1.3) i.e. the id of the nth block is n, if we start counting from 0 (respectively field well_size_blocks and well_numbered_blocks).

Coq implementation

The initial blocks for the XML file are referenced by a unique name stored as a *String* that the preprocessor translates into an index. At the Coq level, we do not have any guarantee that the translation has been correctly done. The numbering step cannot be easily moved into the Gallina generator as it uses a hashtable that can be tricky to manipulate and the original names are used for the generation of the C file header. Instead of adding an axiom stating that the blocks are correctly numbered and the references are correctly converted, we added the property well_numbered_blocks and *is_user_id* for the *block_id* references to the general property stating that the flight plan is well-sized. The size verification pass will thus verify that the preprocessing correctly numbered the blocks and that the block id contained in the flight plan are *user block indices*. However, we cannot ensure that the converted indices correspond to the same block referenced in the XML file.

Finally, we can define $flight\_plan_s$ the dependent type specifying the *well-sized* flight plans, i.e. the subset of $wf\_flight\_plan_e$ that respect the property $verified\_fp_e$.

Definition 9.53 - (flight_plan_sized, src/syntax/FlightPlanSized.v:115).

$flight\_plan_s ::= \{\ fpe\text{:}\ wf\_flight\_plan_e \mid verified\_fp_e\ fpe\ \}$

9.2.2 Size verification function

The size verification pass goes through an extended flight plan and verifies that it respects the $verified\_fp_e$ property. The result produced by the pass is described by *res_size_analysis*.

Definition 9.54 - (res_size_analysis, src/syntax/FlightPlanSized.v:123).

$$\begin{aligned} res\_size\_analysis ::= {} & \texttt{OK}\ flight\_plan_s \\ & \mid \texttt{ERR}\ (\text{list}\ err\_msg) \end{aligned}$$

The analysis will thus return a well-sized flight plan if the original FPE satisfies the property. In the other case, an error is returned with a list of *err_msg* describing error messages.

Definition 9.55 - (err_msg, src/syntax/BasicTypes.v:80).

$$\begin{aligned} err\_msg ::= {} & \texttt{WARNING}\ String \\ & \mid \texttt{ERROR}\ String \end{aligned}$$

The Gallina generator can produce two types of messages. The **ERROR** variant describes errors detected in the flight plan which prevents the generation of a correct C code. This size-verification pass only produces error messages when an incorrect flight plan is detected. The second type of message is specified by the **WARNING** variant which describes behaviours that the user might not expect. For example, such warning messages are produced by the generator, when there are forbidden deroutes that might be blocking. This analysis is presented in Section 9.4.2.

The size verification pass is realized with the function `size_verification`. It thus either returns error messages if the flight plan is detected as incorrect or a well-sized flight plan.

Definition 9.56 - (size_verification, src/generator/FPSizeVerification.v:509).

$size\_verification : flight\_plan_e \to res\_size\_analysis$

Coq implementation

The size restriction has been added after finding an issue in the old generator as it did not perform size verification and thus can produce a flight plan with more than 256 blocks while using 8 bits variables to store the current block id. Some blocks cannot therefore be accessed and the computation of the next block id may create an overflow. This problem was not raised before because the upper bound is not reached in real use cases. Flight plans are in general short as they describe missions that can be realized by small drones with limited batteries.

9.3 Clight generation function

The C code that executes the flight plan can be split into two different parts. First, there is a common C code defined in `common_flight_plan.c` file and presented in Appendix A.2. This file contains the definition of the global variable storing the state of the drone, i.e. the current block and stage indices, but also several functions manipulating this state. For example, `get_nav_stage` returns the current stage index or `NextBlock` updates the current state to execute the next block. The second part corresponds to the C code produced by the generator and consequently to the Clight code produced by the Gallina generation function. This code contains various definitions that are specific to the flight plan considered. This Clight code is generated by the *global_definition* function.

Definition 9.57 - (global_definitions, src/generator/Generator.v:234).
global_definition: $flight\_plan_s \rightarrow$ list *gdef*

This function takes a well-sized flight plan as a parameter and returns a Clight program, noted `FPC`, containing a list of global definitions. This list is composed of the definitions of the global variable `nb_block`, of auxiliary functions used by the common C code and obviously of the `auto_nav` function. The different parts of `FPC` are generated by independent functions.

9.3.1 Number of blocks in a global variable

The function *gvar_nb_blocks* takes as a parameter a `FPS` flight plan and produces the definition of the global variable `nb_block`. This definition states that the variable is an integer value coded on 8 bits that cannot be modified and which is initialised with the number of blocks in the flight plan, computed with the function *get_nb_blocks*.

Definition 9.58 - (gvar_nb_block, src/generator/Generator.v:218).
gvar_nb_blocks : $flight\_plan_s \rightarrow$ *globvar*

Coq implementation

In the previous generator, the number of blocks value was defined as a constant using the `#define` preprocessor macro. This macro was then used in the common C code, for example, to verify that a block id is a *correct block index*. However, we had to switch to a global variable for several reasons. First, the Clight code corresponds to C code after the C preprocessor has been run. The Gallina generator cannot produce macro, and this macro should have been generated by the `VFPG` preprocessor which produces the flight plan headers. Moreover, the verification of the generator requires that we convert the common C code into its corresponding Clight code using the `clightgen` tool of CompCert (cf. Section 10.2.4). Before conversion, the `clightgen` tool runs the C preprocessor and thus all the macros should be defined. At the Coq level, we cannot have the information, without using axioms, that the value in the generated Clight code, obtained after the expansion of the `#define` macro that has been computed in the `VFPG` preprocessor, corresponds to the number of blocks in the flight plan. Having now the Gallina generator producing this global variable, we can easily get information on its value without using any axiom.

9.3.2 Auxiliary functions generated

In the previous generator, the common C code was totally independent of the generated flight plan and thus the generated C file only contained the definition of the `auto_nav` function. However, we extend the generator with new features which forces us to modify the common C code. As presented in Chapter 8, we added the possibility to execute specific code when entering or leaving a block or to prevent a block change which could be dangerous. These new features thus concern changing the block being executed that is realised by the `nav_goto_block` function. This function takes the new block index and computes the new state after the block being executed is changed. Figure 9.5 presents the initial version of the `nav_goto_block` function and the modifications we brought.

```
void nav_goto_block(uint8_t b) {


  if (b != nav_block) {
    last_block = nav_block;
    last_stage = nav_stage;
  }

  nav_block = b;
  nav_init_block();
}
```

(a) Original code

```
void nav_goto_block(uint8_t b) {
  if (forbidden_deroute(nav_block, b)) {
    return;
  }
  if (b != nav_block) {
    last_block = nav_block;
    last_stage = nav_stage;
  }
  on_exit_block(nav_block);
  nav_block = b;
  nav_init_block();
}
```

(b) New version of the function

Figure 9.5: Modifications brought to the function `nav_goto_block`.

Similarly to the *goto_block* or the $goto\_block_e$ functions, the initial version of `nav_goto_block` updates the previous position (`last_block` and `last_stage` variable) only if we are not already in the derouted block. The function `nav_init_block` normalises the block index, initialises the `nav_stage` variable and executes the initialisation code[3]. In the new version of the `nav_goto_block` function, we first verify if the deroute is forbidden or not through the `forbidden_deroute` function. Moreover, we execute the `on_exit` code using `on_exit_block` function as we leave the current block. The `on_enter` code is executed using the `on_enter_block` function that is used in the new version of `nav_init_block`.

[3]The initialisation code corresponds to the code executed in the `FP` semantics of the `goto_block` function (see Definition 8.12). The complete definition of the `nav_init_block` that contains the initialisation code can be found in Appendix A.2.

Remark

The code presented in Figure 9.5b is different from the real common C code (see `common_flight_plan.c` or Appendix A.2). We wanted to protect state variables, for example `nav_block`. In order to be compatible with the previous generator, we kept all these variables that can be accessible outside the `common_flight_plan.c` file as the autopilot might access to these variables. However, we define private variables (for example `private_nav_block`) that store the state and that cannot be accessed outside. The private variables can thus only be modified through the common C code. This restriction prevents the plan to be in an incorrect state. The new version of `nav_goto_block` therefore only modifies the private variables and then updates the public variables.

9.3.2.1 On enter and on exit code

The Gallina functions *gen_fp_on_enter_block* and *gen_fp_on_exit_block* produce respectively the Clight code of the functions `on_enter_block` and `on_exit_block`.

Definition 9.59 - (gen_fp_on_enter_block, src/generator/AuxFunctions.v:60).

$gen\_fp\_on\_enter\_block : flight\_plan_s \rightarrow function$

Definition 9.60 - (gen_fp_on_exit_block, src/generator/AuxFunctions.v:80).

$gen\_fp\_on\_exit\_block : flight\_plan_s \rightarrow function$

These generated functions take a block id and execute the C code corresponding to the block. Figure 9.6 presents the C code generated for the function `on_enter_block` that corresponds to a flight plan containing two blocks of which only one contains code in the `on_enter` field.

```
{|  excpts:= [],
    fb_drtes:= [],
    blocks:= [
      {| id:= 0, on_enter:= None, ... |};
      {| id:= 1,
         on_enter:= Some "enter_1()",
         ... |}
    ]
|}
```

```
void on_enter_block(uint8_t block) {
  switch (block) {
    case 1:
      enter_1();
      break;
    default:
      break;
  }
}
```

Figure 9.6: The C `on_enter_block` function generated from a simple flight plan.

The C code generated for the `on_exit_block` function has the same structure but executes the code of the `on_exit` field. These functions are simply composed of a `switch` statement and every case corresponds to the index of the blocks containing code to execute.

After the execution of the C code, the `break` statement is executed which stops the execution of the function.

9.3.2.2 Forbidden deroute function

The Clight code of the `forbidden_deroute` function is generated by the Gallina function *gen_fp_forbidden_deroute*.

Definition 9.61 - (gen_fp_forbidden_deroute, src/generator/AuxFunctions.v:155).

gen_fp_forbidden_deroute : $flight\_plan_s \rightarrow function$

The `forbidden_deroute` function takes a `from` and `to` block id that corresponds to a deroute between 2 blocks and returns a boolean value stating if the deroute is forbidden by the flight plan. Figure 9.7 presents an example of these function generated for a flight plan containing three forbidden deroutes.

```
{|  excpts:= [],
    fb_drtes:= [
        {| from:= 1, to:= 5,
           only_when:= None |};
        {| from:= 1, to:= 6,
           only_when:= None |};
        {| from: 2, to: 1,
           only_when:= Some "cond()" |}
    ],
    blocks:= [...]
|}
```

```
_Bool forbidden_deroute(uint8_t from,
                        uint8_t to) {
  switch (from) {
    case 1:   if (to == 5)
                return (_Bool) 1;
              if (to == 6)
                return (_Bool) 1;
              break;
    case 2:   if (to == 1)
                if (cond())
                  return (_Bool) 1;
              break;
    default:  break;
  }
  return (_Bool) 0;
}
```

Figure 9.7: The C `forbidden_deroute` function generated from a simple flight plan.

The function is composed of a first `switch` statement where every `case` regroups all the forbidden deroutes with the same `from` block id. The function then checks the `to` parameter to determine if there are matching forbidden deroutes. Finally, the function returns the boolean value `true` if and only if there is a corresponding forbidden deroute that is enabled, based on the `only_when` field.

9.3.3 The auto_nav function

The `auto_nav` function is generated by the Gallina function *gen_fp_auto_nav*. The C code generated for a flight plan was already briefly presented in Figure 9.1 and a larger example is detailed in the Appendix A.1.

Definition 9.62 - (gen_fp_auto_nav, src/generator/Generator.v:204).

gen_fp_auto_nav : $flight\_plan_s \rightarrow function$

The different functions used to produce the Clight code are not presented here as this process is a direct translation from the well-sized flight plan to Clight Code. Indeed, each block is converted into a `case` statement of the *block switch* and similarly, each stage is converted into a `case` statement of the *stage switch*. The implementations of the Clight generation functions can be found in `Generator.v` file.

As presented in Chapter 6, we wanted the new generator to produce a C code similar to the previous generator for compatibility reasons. Even if the overall implementation of the functions is not presented, we want to focus on some details and choices made to produce this constrained Clight code.

First, Clight uses *ident*s, which are positive numbers, to represent all C code identifiers. When `PrintClight` translates the Clight structure into C code, the *ident*s are either converted into the corresponding *String* value stored in an hashtable if it exists or else directly printed as a numerical value. As we want to have human-readable and compilable code, we want then that all identifiers present in the final C code of the autopilot are correctly printed. For example, if we generate Clight code manipulating the `nav_block` variable, we should generate an expression where the *ident* value is "*id*" and the couple (*id*, `nav_block`) is stored in the hashtable used by `PrintClight`. We thus have decided to use the function *create_ident* to generate the *ident* for function names such as `get_nav_stage` or other used variables and to update the hashtable used by `PrintClight`.

Parameter 9.63 - (create_ident, src/generator/ClightGeneration.v:47).

create_ident : *String* → *ident*

The *create_ident* function is a parameter linked to the OCaml function `intern_string`. This function looks in the global hashtable used by `PrintClight` and either returns the *ident* if the *String* value passed as a parameter is already present or generates a new *ident* value and updates the hashtable. We could have implemented this function in Gallina [BJM13], but we would have gained little confidence in the generator for a significant cost in terms of performance and development time as it is not easy to manipulate mutable global variables in Coq.

Remark

As presented in Section 6.2.2, even using the hashtable, `PrintClight` add a $ symbol before the printed identifier. This symbol is thus removed with the post-processor as the C standard does not support the usage of this symbol for identifiers.

The second choice we had to make is how to deal with arbitrary C code. This code can either provide a function call, an expression or even a statement. We thus decided to generate an *ident* for the *String* describing arbitrary C code through a function called *arbitrary_ident*.

Parameter 9.64 - (`arbitrary_ident`, `src/generator/ClightGeneration.v:54`).

arbitrary_ident : *String* → *ident*

Technically, this parameter function is linked to the same OCaml function as *create_ident*. However, we decided to define two separate functions in order to distinguish both codes for documentation purposes. As such, it is easier to understand if an *ident* corresponds to a C identifier or arbitrary C code. Moreover, for the verification of the generator presented in Chapter 10.4.2, we will have to define axioms to manipulate arbitrary C code. These axioms mainly translate the effect of executing arbitrary C code into updating a trace. Having two different functions that generate *ident*, we can restrict the axioms by specializing it to only the expressions and statements that use *arbitrary_ident* to produce *ident*.

Remark

In future work, we could add a parser for the arbitrary C code, in order to check syntax errors and produce correct Clight code without using the *arbitrary_ident* function. However, we should then reformulate or remove the axioms concerning the C code.

Arbitrary C code, translated into an *ident* using the *arbitrary_ident* function is either considered as a variable when it is positioned as an expression or as the name of a void function in a call statement. In both cases, the printed code is therefore the arbitrary C code correctly positioned in the final C code of the autopilot. However, this choice has a drawback for statements. Indeed, we defined a call to a void function without parameters and thus `PrintClight` produces parenthesis with no parameters. For example, for the stage "`CALL "exec1()"`" the statement produced by `PrintClight` will be "`(exec1())();`". One of the roles of the postprocessor previously presented in Section 6.2.2 is thus to remove spurious parentheses. We had to use a call to represent the statement as we cannot have a statement that uses an expression such as a variable in Clight even if this is possible in C[4].

Remark

The preprocessor presented in Section 6.2.1 adds parentheses around arbitrary C code in order to facilitate the suppression of the parentheses during the postprocessing. However, these parentheses restrained the possible C code users can provide, as a code such as "`exec1(); exec2()`" cannot be written anymore while it was possible with the previous generator.

Finally, the remaining choices are due to the particularities of Clight as it is a subset of legit C code (see Section 5.2). For example, a problem we faced is that the boolean operator && does not exist in Clight. We should instead use a `if-then-else` statement with a temporary variable. Similarly, function calls cannot be expressions but only statements with assignment. For example, the condition of the `if` statement cannot be a function call and it is also not possible to write nested functions calls. In both cases, we also have to use

[4] In C we could write the statement `var;` with `var` a variable but this cannot be expressed in Clight.

a temporary variable that stores the intermediate result before being used as an expression. In order to tackle this problem we had to define several temporary variables that are used in different cases. All these variables definitions are regrouped and documented in the TmpVariables.v file. Figure 9.8 presents the C code generated to evaluate an exception, showing how the temporary variables are used.

```
{|  excpts:= [
      {| cond:= "exception1()",
         id:= 4,
         exec:= "test_exec1()" |}
    ],
    fb_drtes:= [...],
    blocks:= [
      ... {| id:= 4, ... |}; ...
    ]
|}
```

```
tmp_nav_block = get_nav_block();
if (tmp_nav_block != 4) {
  tmp_cond = (_Bool) exception1();
} else {
  tmp_cond = (_Bool) 0;
}
if (tmp_cond) {
  test_exec1();
  nav_goto_block(4);
  return;
}
```

Figure 9.8: The C code generated for the evaluation of a exception.

Similarly to the FP semantics presented in the Definition 7.39, an exception is raised only if the current block does not correspond to a potential derouted block and the exception condition is raised. The generated C code thus first gets the index of the current block and stores it in the temporary variable `tmp_nav_block`. The condition "`tmp_nav_block != 4 && exception1()`" is then evaluated using the `if-then-else` statement and by storing the result in the temporary variable `tmp_cond`. Finally, the condition is tested. If the exception is raised, the code in the `exec` field is evaluated and then the execution is derouted before terminating the `auto_nav` function.

9.4 Generator function

We defined in the previous sections the intermediate passes of the generator, and we now define the global generation function. As presented in Section 6.2, this generation function takes the flight plan to convert and a list of variables that must be defined. This function should produce either the Clight definitions with potential warnings or error messages. Section 9.4.1 presents how the global variables are generated. Then, Section 9.4.2 introduces the pass that produces the warning messages. For now, we warn the user when forbidden deroutes might block the execution. Finally, Section 9.4.3 presents the main function that regroups all the previously defined generating functions.

9.4.1 Global Variables generation

Initially, the function generating the Clight global variables was pretty straightforward using the *gen_globaldef_var* function.

Definition 9.65 - (gen_globaldef_var, src/generator/ClightGeneration.v:241).

gen_globaldef_var : *String* → *gdef*

This function takes the name of the variables and generates the definition of the corresponding global variables. The variables produced with this function are defined as mutable and contain integer values stored on 32 bits. These definitions are then added to the global environment along the definition of the Clight functions (`auto_nav`, `forbidden_deroute`...).

However, while verifying the generator, we faced problems when we wanted to prove properties concerning access to elements of the global environment. Indeed, when executing a function call in the Clight semantics, we need to search its definition in the global environment. At the Coq level, we do not have any guarantee that the names of the global variables defined do not clash with the name of the functions we are looking for in the global environment. We thus define *correct_gvars*, the minimal property that the global variables must satisfy in order to avoid all problems related to access to the global environment.

Definition 9.66 - (correct_gvars, src/generator/GvarsVerification.v:43).

correct_gvars : list *gdef* → *Prop*

This property states that the list of global variables should be correctly defined (i.e. they are indeed global integer variables stored on 32 bits) and with no clash between names. A global variable is therefore considered as correctly defined if it is a 32 bits mutable variable and the *ident* used to refer to this variable has been generated with the *create_ident* function, in order to have a reference to its *String* name. The no-name clash part of the *correct_gvars* property states that there should not be multiple definitions with the same name in the list of global variables, but most of all that they must not override the definitions of the functions and variables used by the flight plan. These definitions can be the common definitions, presented in Appendix A.2 or generated definitions (`nb_block`, `auto_nav`...). The variable `reserved_idents` contains the *ident* of reserved identifiers which must not be redefined by any global variable.

Using this property, we thus define the subset *cgvars* describing the set of global definitions that are correctly defined.

Definition 9.67 - (cgvars, src/generator/GvarsVerification.v:48).

cgvars ::= {*vars*: list *gdef* | *correct_gvars vars* }

As we want that global variables definitions should respect the *correct_gvars* property, the function generating these definitions will therefore produce two types of results. This choice is described by *res_gvars_analysis* that either produces a *cgvars* element or a list of errors if it is not possible to generate a correct list of global definitions.

Definition 9.68 - (res_gvars_analysis, src/generator/GvarsVerification.v:64).

res_gvars_analysis ::= **GVARS** *cgvars*
| **ERR_GVARS** (list *err_msg*)

Finally, we define *gvars_definition* that takes a list of variables names and produces a *res_gvars_analysis*.

Definition 9.69 - (gvars_definition, src/generator/GvarsVerification.v:170).

gvars_definition : list *String* → *res_gvars_analysis*

The function *gvars_definition* now ensures us that if it returns a **GVARS** results then we have the list of global variables definitions and a proof that they are correctly defined. This allows us to avoid using axioms stating that the input names do not clash with the `reserved_idents`.

9.4.2 Forbidden deroute analysis

The new generator brings a new feature about forbidden deroutes. As a reminder, the user can define some deroutes between different blocks that could be dangerous and which must be forbidden. For example, passing from a flying block to a block that stops the motors should be blocked. Moreover, we found two cases where the forbidden deroutes might cause unexpected behaviour as they cancel the deroute without any warnings (see the `nav_goto_block` presented in Figure 9.5b).

The first case occurs if all the stages of a block have been executed, then the execution should continue in the next block. The next block state is computed using the `nav_goto_block` function. Unfortunately, this function will not modify the state if the deroute is forbidden and thus the current state will still refer to the end of the previous block. This will result in an infinite loop as every execution of the `auto_nav` function will unsuccessfully try to go to the next block.

The other case occurs when there is a **DEROUTE** stage from the current block id_{from} to the block id_{to}, and that this deroute is forbidden. The execution of the deroute stage will have no effect. Indeed, when a deroute is executed, the state is updated to refer to the next stage (see rule FP-27 or rule FPE-36) and then the `nav_goto_block` function is executed. In this case, the block index in this state will not be modified as the deroute is forbidden. At the next call of the `auto_nav` function, the execution will thus continue starting from the stage following the **DEROUTE** stage. Thus, we must warn the user that the deroute stage he has defined might be ignored during the execution.

These behaviours might be misleading and we want to warn the user if these cases arise. Therefore, we define the *fb_deroute_analysis* function that takes a flight plan and produces messages warning the user in case some forbidden deroute can provoke an unexpected execution.

Definition 9.70 - (fb_deroute_analysis, src/generator/FBDerouteAnalysis.v:181).

fb_deroute_analysis : *flight_plan* → list *err_msg*

The forbidden deroute analysis currently provides warning messages if two successive blocks are forbidden or if there is a deroute in a block that might be blocked. The analysis only produces warnings and not error messages as the user might develop a flight plan based on this unexpected behaviours. Moreover, the analysis does not produce errors as

it does not take into account the `only_when` condition as we cannot evaluate it statically. The analysis may therefore produce warnings about forbidden deroute in cases where the condition `only_when` will never be `true`.

Remark

In future work, we could imagine adding a new warning message about local exceptions that are forbidden. Take for example a flight plan where in the block id_{from} there is a local exception that should deroute the execution to the block id_{to} when it is raised. If this flight plan forbids the deroute from id_{from} to id_{to}, then the exception will have no effect when it is raised. Also in this case the `auto_nav` function will terminate every time before executing any stage (see the code in Figure 9.8).

9.4.3 Flight plan generator function

We finally define the global function that generates the Clight code corresponding to a flight plan. As we saw previously, the size verification pass or the generation of the global variables can produce errors. The result of the generation function is described by *res_gen* that produces either the Clight code with potential warnings or errors.

Definition 9.71 - (res_gen, src/generator/Generator.v:243).

res_gen ::= **CODE** (*prog* × list *err_msg*)
| **ERRORS** (list *err_msg*)

The generation function, called *generate_flight_plan*, produces the global environment containing the definitions of the global variables passed as parameters and the functions generated that correspond to the flight plan.

Definition 9.72 - (generate_flight_plan, src/generator/Generator.v:249).

```
generate_flight_plan : flight_plan → list String → res_gen
generate_flight_plan fp name_gvars ::=
    let fpe := extend_flight_plan fp in
    let deroute_analysis := fb_deroute_analysis fp in
    match gvars_definition name_gvars with
      | GVARS gvars ⇒
        match size_verification fpe with
          | OK fps ⇒
            let prog := {|auto_nav, gvars ++ (global_definition fps) |} in
            CODE (prog, deroute_analysis)
          | ERR errors ⇒ ERRORS (deroute_analysis ++ errors)
        end
      | ERR_GVARS errors ⇒ ERRORS (deroute_analysis ++ errors)
    end
```

The function starts by extending the flight plan using the `extend_flight_plan` function presented in Section 9.1.3. As presented previously, this extension pass transforms the flight plan into a version closer to Clight code, in particular, every stage is numbered and there is a direct match between stages in the flight plan and `case` in the *stage switch* of the Clight code. Then, the two passes that may produce errors and stop the code generation are executed. In the first pass, we try to produce correct global variables (see Section 9.4.1). The second pass corresponds to the size verification of the flight plan realised by the `size_verification` function introduced in Section 9.2.2. This pass ensures, among other things, that the flight plan does not contain more than 256 blocks or stages to avoid overflow errors. Finally, the `global_definitions` function, described in Section 9.3, generates the Clight code of the `auto_nav` and auxiliary functions and the definition of the global variables. In any case, whether the code generation is stopped by errors or not, the warning messages produced by the forbidden deroute analysis are also returned (see Section 9.4.2).

Chapter 10

Verification of the Flight Plan Generator

Contents

10.1 Generic definition of flight plans semantics 213
10.2 Semantic preservation verification 219
10.2.1 Framework for semantic preservation 219
10.2.2 Extension pass verification 221
10.2.3 Verification of the size verification pass 223
10.2.4 Verification of the Clight generation pass 224
10.2.5 VFPG correctness theorem 228
10.3 Termination proof of the auto_nav function 229
10.4 Verification hypotheses 229
10.4.1 Models of the system 230
10.4.2 New semantic rules 231
10.4.3 Axioms used for the proof 235

The verification of the generator can be divided into two parts. First, we should verify that the generator behaves correctly by proving that the FP semantics is preserved when generating the C code, as presented in Section 5.1. Second, the code generated for the function auto_nav must always terminate as both the Paparazzi code that calls regularly the auto_nav function and the main control loop of the autopilot are executed in a single thread program. We thus want to ensure that the autopilot can call the auto_nav function without the risk of being blocked because of an infinite loop.

The new 3-passes generator is composed of several languages (FP, FPE, FPS and FPC) that have different semantics. Section 10.1 presents a common semantic framework and instantiates this definition for the four languages used. These definitions are used in Section 10.2 to specify the semantic preservation property and to verify it. Section 10.3 then focuses on proving termination of the auto_nav function. Finally, Section 10.4 presents the hypotheses used for the verification of the generator.

10.1 Generic definition of flight plans semantics

The new generator is composed of three passes that consume and produce different representations of the flight plan provided by the languages presented in the previous chapters.

These languages have their own semantics. In order to define a generic framework for the manipulation of semantics and therefore to be able to specify the semantic preservation property, we define a record type *fp_semantics* corresponding to a generic definition of a flight plan semantics.

Definition 10.1 - (fp_semantics, src/semantics/FPBigStepGeneric.v:32).

fp_semantics ::= {|
  env: *Type*,
  initial_env: env → *Prop*,
  step: env → env → *Prop*
|}

fp_semantics is defined by three fields:

- env, the type representing (somehow abstracted) drone environments;
- a initial_env property, initial_env e stating that the environment e is an initial drone environment;
- a step property, step e e' stating that the execution of a flight plan step, which corresponds to a auto_nav function call, from the environment e will eventually terminate and yield in the environment e'.

Having this specification of a flight plan semantics, we can now instantiate it for the different flight plan representations used in the generator.

FP semantics

The *fp_semantics* of FP can be simply defined with what has been introduced in Chapter 7 and 8.

Definition 10.2 - (semantics_fp, src/semantics/FPBigStep.v:275).

semantics : *flight_plan* → *fp_semantics*
semantics fp ::= {|
  env := *fp_env*,
  initial_env := λ e, *init_env fp* = e,
  step := λ e e', $e \xhookrightarrow[fp]{\text{FP}} e'$
|}

Remark

Notice that we use the classic notation $\lambda e, expr(e)$ to define anonymous function taking a parameter e and returning the value of $expr(e)$.

FPE semantics

The definition of FPE can be presented similarly as its semantics has already been defined in Section 9.1.

Definition 10.3 - (semantics_fpe, src/semantics/FPBigStepExtended.v:558).

$semantics_e$: $wf\_flight\_plan_e \to fp\_semantics$
$semantics_e$ fpe ::= {|
  env:= fp_env_e,
  initial_env:= λ e, $init\_env_e$ fpe = e,
  step:= λ e e', $e \xrightarrow[fpe]{\text{FPE}} e'$
|}

FPS semantics

Well-sized flight plans ($flight\_plan_s$) are defined as a dependent type over FPE. The semantics of FPS will use the same semantics as FPE because they share the same structure, but with a restricted environment: as the flight plan is well-sized (e.g. we have guarantees on the number of blocks), we want to only use environments with block and stage indices restricted to 8 bits values. We thus define the property $fp\_env\_on\_8$ describing such environments.

Definition 10.4 - (fp_env_on_8, src/semantics/FPEnvironmentSized.v:42).

$fp\_env\_on\_8$: $fp\_env_e \to Prop$
$fp\_env\_on\_8$ e ::= {|
  nav_block8: is_nat8 e.idb, nav_stage8: is_nat8 e.ids,
  last_block8: is_nat8 e.lidb, last_stage8: is_nat8 e.lids
|}

This property states that a fp_env_e can be encoded on 8 bits only if every index of the environment contains 8 bits values. The definition of fp_env8, the environment for FPS semantics, is therefore a dependent type of fp_env_e that satisfies property $fp\_env\_on\_8$.

Definition 10.5 - (fp_env8, src/semantics/FPEnvironmentSized.v:49).

fp_env8 ::= { e: fp_env_e | $fp\_env\_on\_8$ e }

The definition of FPS semantics is based on the semantics of FPE and thus the definition of FPS initial environment uses the function $init\_env_e$.

Definition 10.6 - (init_env8, src/semantics/FPBigStepSized.v:1138).

$init\_env_s$: $flight\_plan_s \to fp\_env8$
$init\_env_s$ fps ::= exist fp_env8 ($init\_env_e$ (fps).proof) is_init_env8

The initial environment for FPS is an element of fp_env8 which is a dependent type. We must thus provide an initial value (provided by the $init\_env_e$ function) and a proof (is_init_env8) that this environment satisfies the property $fp\_env\_on\_8$.

The step function for FPS is based on the step function of FPE but takes a FPS flight plan as parameter.

Definition 10.7 - (step, src/semantics/FPBigStepSized.v:1124).

$step_s$: $flight\_plan_s \to fp\_env8 \to fp\_env8$

The property "$step_s\ fps\ e = e'$", noted $e \xhookrightarrow[fps]{\mathsf{FPS}} e'$, states that the execution of the flight plan fps from the initial environment e eventually terminates in the states e'. The definition of the FPS function can be summarised by the following inference rule.

$$\frac{fpe = (fps).\mathtt{value} \qquad (e).\mathtt{value} \xhookrightarrow[fpe]{\mathsf{FPE}} (e').\mathtt{value}}{e \xhookrightarrow[fps]{\mathsf{FPS}} e'}\ (\text{FPS DEF})$$

Inference rule (FPS def) states that $e \xhookrightarrow[fps]{\mathsf{FPS}} e'$ is defined if the execution of the flight plan fpe extracted from fps from the environment of e using the semantics of FPE terminates in the environment e'. We have the guarantee that e' is a correct fp_env8 through the following lemma.

Lemma 10.8 - (exec_step8, src/semantics/FPBigStepSized.v:1080).

$\forall\ fpe\ e\ e',\ fp\_env\_on\_8\ e$
$\to e \xhookrightarrow[fpe]{\mathsf{FPE}} e'$
$\to fp\_env\_on\_8\ e'$

The lemma exec_step8 states that any execution of the FPE semantics from an environment respecting the $fp\_env\_on\_8$ property will terminate in an environment also respecting the property $fp\_env\_on\_8$. The FPS step function is thus correctly defined, i.e. the execution of the step function from any initial fp_env8 will terminate in a fp_env8 environment.

We can finally define the semantics of FPS.

Definition 10.9 - (semantics_fps, src/semantics/FPBigStepSized.v:1204).

$semantics_s$: $flight\_plan_s \to fp\_semantics$
$semantics_s\ fps ::= \{|$
  env := fp_env8,
  initial_env := $\lambda\ e,\ init\_env_s\ fps = e$,
  step := $\lambda\ e\ e',\ e \xhookrightarrow[fps]{\mathsf{FPS}} e'$
$|\}$

Clight semantics for flight plan

We now define the semantics for FPC describing the execution of the Clight code for a flight plan. Let us first define the environment used. As for the other flight plan semantics, the

environment contains the current state and a history of external calls that occurred during the execution.

Definition 10.10 - (fp_cenv, src/semantics/FPEnvironmentClight.v:46).

fp_env_c ::= {|
  mem: *mem*, trace: *trace*
|}

In the case of FPC semantics, the variables representing the flight plan states (e.g. nav_stage storing the idb value) are global variables stored in the memory. The state of fp_env_c is thus the Clight memory and the history is the Clight trace.

We can now define the first property of the flight plan semantics, $init\_env_c$, describing the initial environment of fp_env_c.

Definition 10.11 - (initial_state, src/semantics/FPBigStepClight.v:47).

$init\_env_c$: *prog* → fp_env_c → *Prop*
$init\_env_c$ *P e* ::= {|
  init_m_env: *init_mem P* = Some *e*.mem,
  init_trace: *e*.trace = []
|}

The initial environment property is parametrised by *prog*, similarly to the definitions for the other semantics. Indeed, the program contains all the functions definitions of the flight plan. The initial environment is thus composed of a memory state that is the initial memory for the program passed as a parameter and the trace is empty as the execution of the flight plan has not started.

The definition of the step function for the FPC semantics is based on the Clight small-step operational semantics introduced in Section 5.2.2 and for this reason, the FPC step function is directly defined as an operational semantics instead of denotational semantics with a computable step function like for FP, FPE and FPS.

Signature 10.12 - (step, src/semantics/FPBigStepClight.v:34).

$$step_c\text{: } prog \to fp\_env_c \to fp\_env_c \to Prop$$

The property "$step_c$ *P e e′*", noted $e \xhookrightarrow[P]{\text{FPC}} e'$, states that the execution of the program *P* for the state *e* terminates in the state e'.

Definition 10.13 - (step, src/semantics/FPBigStepClight.v:34).

$$\frac{\begin{array}{c} G = \mathit{globalenv}\ P \\ G \vdash \mathcal{S}(F,\ s,\ k,\ E,\ L,\ e.\mathsf{mem}) \xrightarrow{t}{}^{*} \mathcal{S}(F,\ \mathbf{Sskip},\ k,\ E,\ L,\ e'.\mathsf{mem}) \\ s = \mathbf{Scall}(\mathsf{None},\ \mathbf{Evar}\ \mathsf{auto\_nav},\ []) \\ E[\mathsf{auto\_nav}] = \mathsf{None} \qquad e'.\mathsf{trace} = e.\mathsf{trace}\ +\!\!+\ t \end{array}}{e \xhookrightarrow[P]{\mathsf{FPC}} e'}\ (\text{FPC DEF})$$

The rule (FPC def) defines the $step_c$ property. This property states that the FPC step function corresponds to the execution of a call statement to the auto_nav function through Clight semantics. This property is parametrised by the global environment G generated from the program P and enforces the following points:

- The initial state is a regular state containing the information that the execution is currently in the function F, the next statement to execute is a **Scall** for the auto_nav function, the code remaining to execute is stored in the continuation k, the environment is E, the local environment is L and the memory environment is stored in e.
- The execution terminates in a regular state with the same information for the current function F, the continuation k, the environment E and the local environment L but where the remaining statement to execute is **Sskip** and the final memory state is the memory stored in e'.
- The final trace stored in e' appends the trace t produced by the semantics to the trace of the initial environment e.

Finally, we restrict the environment E to not contain definitions for the auto_nav identifier in order to force the use of the auto_nav definition that must be defined in the global environment.

The semantic of FPC can finally be defined using the $init\_env_c$ and $step_c$ properties.

Definition 10.14 - (semantics_fpc, src/semantics/FPBigStepClight.v:54).

$semantics_c$: $prog \to fp\_semantics$
$semantics_c\ P ::= \{|$
  env $:= fp\_env_c$,
  initial_env $:= init\_env_c\ P$,
  step $:= \lambda e\ e', e \xhookrightarrow[P]{\mathsf{FPC}} e'$
$|\}$

10.2 Semantic preservation verification

The idea of semantic preservation presented in Section 5.1 consists in "executing" through the corresponding formal semantics the source program and the generated program starting from two matching environments and verifying that both executions terminate in matching states. In order to do so, a relation $(\backsim)$ matching environments in the source language and environments in the target language needs thus to be defined. Section 10.2.1 introduces the formalisation of the semantic preservation property between flight plans languages specified using the generic definition. This property is defined so that the proof of the global VFPG generator can be decomposed into the proof of the semantic preservation of the different passes. Section 10.2.2, 10.2.3 and 10.2.4 corresponds to the verification of the 3-passes of the generator by mainly specifying the matching relation between environments. Finally, Section 10.2.5 presents the global theorem describing the correctness of the VFPG generator.

10.2.1 Framework for semantic preservation

We consider two flight plans semantics FP_1 and FP_2 equipped with a matching relation $(\backsim)$. Definition 10.15 formalises the *simulation* property stating that FP_2 can simulate every behaviour of FP_1. This property is defined if two proofs are provided:

1. A proof of the `match_initial_env` property, that corresponds to Definition 5.4, stating that for every initial environments of FP_1 there is a matching initial environment of FP_2.

2. A proof of the `match_step` property, which is the flight plan version of the Lock-step simulation, shown in Definition 5.5. This property states that for every execution of FP_1 there are matching FP_2 environments that describe the same execution in FP_2 semantics.

Definition 10.15 - (`fp_fsim_properties`, `src/semantics/FPBigStepGeneric.v:51`).

$$
\begin{aligned}
&\mathit{simulation}\text{: } (FP_1\text{: } \mathit{fp\_semantics})\\
&\qquad\qquad \to (FP_2 : \mathit{fp\_semantics})\\
&\qquad\qquad \to (\mathit{match\_envs}\text{: } FP_1.\mathsf{env} \to FP_2.\mathsf{env} \to \mathit{Prop})\\
&\qquad\qquad \to \mathit{Prop}\\
&\mathit{simulation}\ FP_1\ FP_2\ (\backsim) ::= \{|\\
&\quad \mathtt{match\_initial\_env}\text{: } \forall\ e_1,\ FP_1.\mathsf{initial\_env}\ e_1\\
&\qquad\qquad\qquad\qquad \to \exists\ e_2,\ FP_2.\mathsf{initial\_env}\ e_2 \wedge e_1 \backsim e_2,\\
&\quad \mathtt{match\_step}\text{: } \qquad \forall\ e_1\ e_1',\ FP_1.\mathsf{step}\ e_1\ e_1'\\
&\qquad\qquad\qquad\qquad \to \forall\ e_2,\ e_1 \backsim e_2\\
&\qquad\qquad\qquad\qquad \to \exists\ e_2',\ FP_2.\mathsf{step}\ e_2\ e_2' \wedge e_1' \backsim e_2'\\
&|\}
\end{aligned}
$$

The property describing the simulation of flight plan semantics is then used to define a bisimulation property corresponding to the semantics preservation property we want to prove.

Definition 10.16 - (fp_bisim_properties, src/semantics/FPBigStepGeneric.v:81).

bisimulation_prop: $(FP_1$: *fp_semantics*$)$
$\rightarrow (FP_2 : \textit{fp\_semantics})$
$\rightarrow (\textit{match\_envs}$: FP_1.env $\rightarrow$ FP_2.env $\rightarrow$ *Prop*$)$
$\rightarrow$ *Prop*
bisimulation_prop FP_1 FP_2 $(\backsim) ::= \{|$
forward_simulation: *simulation* FP_1 FP_2 $(\backsim)$,
backward_simulation: *simulation* FP_2 FP_1 $(\backsim)$
$|\}$

The *bisimulation_prop* property for two flight plans semantics is defined if the proof of the forward simulation (simulation between FP_1 and FP_2) and the proof of the backward simulation (simulation between FP_2 and FP_1) are provided. If we consider having a generator with input flight plans from FP_1 producing flight plans of FP_2, the forward simulation property states that every behaviour of the input flight plans exists in the produced code. On the other hand, the backward simulation property states that the produced flight plans do not bring new behaviours that are not described in FP_1 semantics.

Coq implementation

Note that in Definition 10.16 of *bisimulation_prop*, there is a type error for the matching environment of the backward simulation property. Indeed, to simplify the notation we use the same *simulation* property for forward and backward simulation, but in the Coq source code, two definitions for the simulation property are needed (fp_fsim_properties and fp_bsim_properties).

Definition 10.17 - (bisimulation, src/semantics/FPBigStepGeneric.v:92).

bisimulation: *fp_semantics* $\rightarrow$ *fp_semantics* $\rightarrow$ *Prop*
bisimulation FP_1 FP_2 ::=
| Bisimulation (*match_envs*: FP_1.env $\rightarrow$ FP_2.env $\rightarrow$ *Prop*)
(*props*: *bisimulation_prop* FP_1 FP_2 *match_envs*)

Finally, we can define the *bisimulation* property that is satisfied for two semantics if a matching environment relation and a proof of the *bisimulation_prop* property are provided. This bisimulation property is well-defined as we use it with only deterministic semantics.

We can also easily define and prove the compose_bisimulations lemma stating that the bisimulation between flight plans semantics can be composed. This property of the bisimulation is used to prove the correctness theorem of the generator as presented in Section 10.2.5.

Lemma 10.18 - (compose_bisimulations, src/semantics/FPBigStepGeneric.v:97).

$\forall\ FP_1\ FP_2\ FP_3,$
$\quad bisimulation\ FP_1\ FP_2$
$\quad \rightarrow bisimulation\ FP_2\ FP_3$
$\quad \rightarrow bisimulation\ FP_1\ FP_3$

10.2.2 Extension pass verification

This section presents the verification of the first pass of the generator, i.e. the FP to FPE. We therefore need to specify the matching relation between *fp_env* and fp_env_e in order to define the correctness property.

As environments in both languages are composed of a state and a trace, the matching relation can be decomposed into two matching relations. As they use the same definition for traces, the traces matching relation is simply equality on lists. On the other hand, the matching relation between states must be satisfied when both environments describe the same execution state of a flight plan. First, Definition 10.19 specifies a matching relation between a FP stage s and a FPE stage s_e, noted $s \overset{\text{stage}}{\sim} s_e$.

Definition 10.19 - (match_stage, src/verification/MatchFPwFPE.v:46).

$$\frac{}{\texttt{SET}\ (var,\ val) \overset{\text{stage}}{\sim} \texttt{SET}_e\ (ids,\ var,\ val)}\ (\textsc{Set})$$

$$\frac{}{\texttt{CALL}\ (fun,\ until,\ loop,\ break) \overset{\text{stage}}{\sim} \texttt{CALL}_e\ (ids,\ fun,\ until,\ loop,\ break)}\ (\textsc{Call})$$

$$\frac{}{\texttt{DEROUTE}\ idb \overset{\text{stage}}{\sim} \texttt{DEROUTE}_e\ (ids,\ idb)}\ \textsc{Deroute} \qquad \frac{}{\texttt{RETURN}\ reset \overset{\text{stage}}{\sim} \texttt{RETURN}_e\ (ids,\ reset)}\ (\textsc{Return})$$

$$\frac{}{\texttt{NAV}\ (mode,\ until,\ \texttt{false}) \overset{\text{stage}}{\sim} \texttt{NAV}_e\ (ids,\ mode,\ until)}\ (\textsc{Nav})$$

Next, we define equivalence between a list l of FP stages and a list l_e of FPE stages, noted $l \overset{\text{stages}}{\sim} l_e$. We note $|l|$ the length of list l and $(l)[s ::]$ the suffix of l starting from index s. The relation $\overset{\text{stages}}{\sim}$ is specified on Definition 10.20 and follows a common induction scheme over lists. Axiom (Nil) states that an empty FP list is equivalent to a FPE list containing the **DEFAULT**$_e$ stage. The (Cons) rule allows to add equivalent stages in front of equivalent lists of stages. The (Nav init) rule treats the case of the **NAV** FP stage when an initialisation step is required. Finally, the **WHILE** FP stage needs to be treated more carefully as the list of stages corresponds to the remaining stages to be executed: we might be inside a loop or encounter new loops. Rule (End While) describes the case when we are already in a loop, stating that we should find an **END_WHILE**$_e$ stage in the FPE list. Rule (While) represents the other case where we should find the whole loop: a **WHILE**$_e$ stage, a body equivalent to the body of **WHILE** and then a **END_WHILE**$_e$.

We can now define an equivalence relation between environments *fp_env* and fp_env_e through the $\overset{\text{env}}{\underset{fpe}{\sim}}$ relation, called match_env in the Coq implementation. We denote by $e \overset{\text{env}}{\underset{fpe}{\sim}} e'$ the expression *match_env e e'*.

Definition 10.20 - (match_stages, src/verification/MatchFPwFPE.v:60).

$$\frac{}{[] \overset{\text{stages}}{\sim} [\texttt{DEFAULT}_e\ id]}\ (\textsc{Nil}) \qquad \frac{s \overset{\text{stage}}{\sim} s_e \qquad stages \overset{\text{stages}}{\sim} stages_e}{s :: stages \overset{\text{stages}}{\sim} s_e :: stages_e}\ (\textsc{Cons})$$

$$\frac{stages \overset{\text{stages}}{\sim} stages_e}{\texttt{NAV}\ (mode,\ until,\ \texttt{true}) :: stages \overset{\text{stages}}{\sim} \texttt{NAV\_INIT}_e\ (ids, mode) :: \texttt{NAV}_e\ (ids',\ mode,\ until) :: stages_e}\ (\textsc{Nav init})$$

$$\frac{cond = p_e.\texttt{cond} \qquad body \overset{\text{stages}}{\sim} body_e \qquad stages \overset{\text{stages}}{\sim} stages_e}{\texttt{WHILE}\ (cond, body) :: stages \overset{\text{stages}}{\sim} \texttt{END\_WHILE}_e\ (ids,\ p_e,\ body_e) :: stages_e}\ (\textsc{End While})$$

$$\frac{\begin{array}{l} stages_e = body' +\!\!+ [\texttt{END\_WHILE}_e\ (n',\ p'_e,\ body')] +\!\!+ stages'_e \qquad cond = p_e.\texttt{cond} = p'_e.\texttt{cond} \\ |body'| = p_e.\texttt{end\_while\_id} - p_e.\texttt{while\_id} + 1 \qquad body \overset{\text{stages}}{\sim} body' \qquad stages \overset{\text{stages}}{\sim} stages'_e \end{array}}{\texttt{WHILE}\ (cond, body) :: stages \overset{\text{stages}}{\sim} \texttt{WHILE}_e\ (ids,\ p_e) :: stages_e}\ (\textsc{While})$$

Definition 10.21 - (match_env, src/verification/MatchFPwFPE.v:104).

match_env: $fp\_env \to fp\_env_e \to Prop$
match_env $e\ e' ::= \{|$

match_block:	$e.\texttt{idb} = e'.\texttt{idb}$,
match_stages:	$e.\texttt{stages} \overset{\text{stages}}{\sim} ((block_{fpe}(e'.\texttt{idb}).\texttt{stages})[e'.\texttt{ids} ::])$,
match_last_block:	$e.\texttt{lidb} = e'.\texttt{lidb}$,
match_last_stages:	$e.\texttt{stages} \overset{\text{stages}}{\sim} ((block_{fpe}(e'.\texttt{lidb}).\texttt{stages})[e'.\texttt{lids} ::])$,
gmatch_trace:	$e.\texttt{trace} = e'.\texttt{trace}$

$|\}$

Two matching environments must thus share the same current trace and matching states. As we cannot compare a list of FP stages with a FPE stage index, we compare the lists of remaining stages to be executed. The list of FPE stages remaining to be executed is obtained by removing the $e'.\texttt{ids} - 1$ first stages of the current block. The *fpe* flight plan is required to define the simulation relation as fp_env_e describes a stage position specific to this flight plan.

Having defined matching relations and the formal semantics of the flight plans, the following semantic preservation theorem can be established and then proven.

Theorem 10.22 - (semantic_preservation, src/verification/VerifFPToFPE.v:1817).

$\forall\ fp\ fpe,\ extend\_flight\_plan\ fp = fpe$
$\quad \to bisimulation\ (semantics\ fp)\ (semantics_e\ fpe)$

Sketchily, as the number of stages to execute before reaching a break stage is unknown, we have to prove the semantics preservation theorem by induction over the list of stages. However, we faced a problem when performing this induction as the list is a nested recursive *fp_stage* type, i.e. the list of *fp_stage* may contain a **WHILE** stage that also contains a list of *fp_stage*. As we naturally wanted to use Coq types and libraries, the definitions of FP and

FPE, presented in Section 7.1 and 9.1.1, use the Coq native list type and thus the induction principle generated automatically by Coq does not take into account the interplay between *fp_stage* and list. We thus had to define our own induction principle: proving a property P for a *fp_stage* requires proving P for all the different nested stages (see the `fp_stage_ind'` induction principle).

10.2.3 Verification of the size verification pass

As a reminder of Section 9.2, the second pass performs some size verification on $wf\_flight\_plan_e$. In the case where no errors are found, the produced $flight\_plan_s$ describes the same flight plan as the input but as it is a dependent type, it contains a proof that the flight plan is well-sized. In addition, the FPS semantics is based on the semantics of FPE but using a *fp_env8* environment, i.e. a *fp_env* that only contains 8 bits values. In order for the $step_s$ function to be well-defined, we proved that any execution of the $step_e$ function from an *fp_env8* environment terminates in an *fp_env8*.

As FPE and FPS semantics have common definitions, the matching relation between environment for the definition of the semantic preservation theorem is thus the equality of the environments and ignores the proof contained in the *fp_env8* dependent type.

Definition 10.23 - (match_env, src/semantics/FPEnvironmentSized.v:118).

$match\_env8$: $fp\_env_e \rightarrow fp\_env8 \rightarrow Prop$
$match\_env8\ e\ e' ::= e = (e')$.value

We can now easily define and prove the semantic preservation of the second pass as they use the same semantics functions. The proof simply consists of extracting the FPE definitions from FPS semantics. However, the theorem 10.24 is proven under the condition that the verification pass does not produce errors as it is a necessary condition for the FPS semantics to be well-defined.

Theorem 10.24 - (semantic_preservation, src/verification/VerifFPEToFPS.v:98).

$\forall\ fpe\ fps,\ size\_verification\ fpe = \mathbf{OK}\ fps$
$\rightarrow bisimulation\ (semantics_e\ fpe)\ (semantics_s\ fps)$

Remark

An important information provided by the theorem, is that if the flight plan is well-sized then any execution of FPE (sequence of calls to the $step_e$ function), starting from an correct initial environment will always terminate in an environment with 8 bits values. This information is essential for the verification of the third pass, because it guarantees that there will be no overflow while storing the environments in memory.

10.2.4 Verification of the Clight generation pass

The verification of the first generator pass allows us to prove that FP and FPE semantics behave similarly despite having different execution structures. The second pass shows that FPE and FPS have similar execution structures and thus that both flight plan semantics will only produce 8 bits environment. The final pass is quite different as we want to ensure the semantic preservation between FPS and FPC that share a similar execution structure (e.g. by storing the current block and stage indices) with totally different representations, both for flight plans (*flight_plan*$_s$ vs Clight code) and for environments used in semantics (*fp_env8* vs *mem*).

In order to specify the semantic preservation theorem we need to first define the matching relation between these two different representations of environments. Similarly to the verification of the extension pass, the matching relation is decomposed into two parts: a matching relation between traces and a matching relation between states.

A flight plan trace t and a Clight trace t_c are matching through the *match_trace* relation (see Definition 10.25), if they contain the same information. We use the notation $t \overset{\text{trace}}{\sim} t_c$ to denote this matching relation.

Definition 10.25 - (match_trace, src/verification/MatchFPSwFPC.v:49).

match_trace : *fp_trace* → *trace* → *Prop*

The *match_trace* relation is defined by the inferences rule presented in Definition 10.26. These rules are defined inductively on the trace list and by ignoring **SKIP**. Two elements are matching if they respect the match_event property, noted $\overset{\text{event}}{\sim}$.

Definition 10.26 - (match_trace, src/verification/MatchFPSwFPC.v:49).

$$[] \overset{\text{trace}}{\sim} [] \ (\textsc{Nil}) \qquad \frac{t \overset{\text{trace}}{\sim} t_c}{(\textbf{SKIP} :: t) \overset{\text{trace}}{\sim} t_c} \ (\textsc{Skip}) \qquad \frac{e \overset{\text{event}}{\sim} e_c \qquad t \overset{\text{trace}}{\sim} t_c}{(e :: t) \overset{\text{trace}}{\sim} (e_c :: t_c)} \ \textsc{Cons}$$

The match_event relation, presented in Definition 10.27 simply states that any condition evaluation or C code execution described in a *fp_event* should correspond to an equivalent *event* (an element of a *trace*).

Definition 10.27 - (match_event, src/verification/MatchFPSwFPC.v:41).

match_event: *fp_event* → *event* → *Prop*

This property is simply defined by the two following inference rules, (Cond) and (Code):

$$\frac{}{\textbf{COND}\ (cond,\ b) \overset{\text{event}}{\sim} (cond\_event\ cond\ b)} \ (\textsc{Cond}) \qquad \frac{}{\textbf{C\_CODE}\ code \overset{\text{event}}{\sim} (code\_event\ code)} \ (\textsc{Code})$$

These rules are based on the two following functions producing respectively an *event* that corresponds to the execution of an arbitrary C code or the evaluation of a boolean condition.

Definition 10.28 - (code_event, src/semantics/FPEnvironmentClight.v:101).

code_event: *c_code* → *event*
code_event code ::= **Event_annot** *code* [].

Definition 10.29 - (cond_event, src/semantics/FPEnvironmentClight.v:98).

cond_event: *c_cond* → *bool* → *event*
cond_event cond res ::= **Event_syscall** *cond* [] (int_of_bool *res*).

Both **Event_annot** and **Event_syscall** are event constructors. The function int_of_bool converts a Boolean Coq value into an integer value of CompCert.

Let us now define the relation between the state of *fp_env8* and the memory. In the generated C code, the *state values* of the flight plan (e.g. the variable nav_stage storing the value of the stage index ids) are global variables stored in the memory at specific locations. The global environment G contains all locations and definitions of these global variables. The matching relation will thus ensure that the values stored in the memory correspond to the element of the *fp_env8* state. We thus define the property *correct_mem_access* specifying that a global variable contains a specific value.

Definition 10.30 - (correct_mem_access, src/verification/MatchFPSwFPC.v:102).

correct_mem_access: *genv* → *mem* → *ident* → *value* → *Prop*
correct_mem_access G M *id* v ::= $\forall$ l, *symbol* G *id* = Some l
→ writable8_mem M l $\wedge$ *load* M l = Some v

The property "*correct_mem_access* G M *id* v" states that the memory M contains at the location of the global variable *id*, defined in G, the value v. This memory M should also satisfy the property writable8_mem stating that the location of the global variable is readable and writable. This property is mandatory as the state will be modified during the execution of the auto_nav function.

The execution of the flight plan also requires the usage of constant variables such as nb_blocks storing the number of blocks. We thus define the property *correct_const_access* stating that a constant variable still contains its initial value, defined in the global definition. Indeed, we do not explicitly define the element *globvar* in Section 5.2 that contains the definition of a global variable, but a global variable can be defined as constant, with a flag read-only set to true, and has also a flag init setting the initial value of the variable.

Definition 10.31 - (correct_const_access, src/verification/MatchFPSwFPC.v:111).

correct_const_access: *genv* → *mem* → *ident* → *Prop*
correct_const_access G M *id* ::= $\forall$ l d, *symbol* G *id* = Some l
→ *find_def* G l = Some (**Gvar** d)
→ *load* M l = Some d.init

We can define the relation "*match_menv* G e M", noted $e \overset{\text{menv}}{\underset{G}{\sim}} M$, that states whether an *fp_env8* environment e is matching the memory state M with the global environment

G, containing the definition of the global variables.

Definition 10.32 - (match_menv, src/verification/MatchFPSwFPC.v:123).

match_menv: *genv* → *fp_env8* → *mem* → *Prop*
match_menv G e M ::= {|
  correct_nav_block: *correct_mem_access* G M nav_block e.idb,
  correct_nav_stage: *correct_mem_access* G M nav_stage e.ids,
  correct_last_block: *correct_mem_access* G M last_block e.lidb,
  correct_last_stage: *correct_mem_access* G M last_stage e.lids,
  correct_nb_blocks: *correct_const_access* G M nb_blocks
|}

The *match_menv* relation specifies that all global variables corresponding to the state variables contain the same values in the memory M as in e. This relation also specifies that the global constant nb_blocks is well-defined.

Remark

As explained in Section 9.3, there are also private variables that store the value of the current state, and thus the property in the Coq source also defines cases for all the private variables.

Finally, the main matching relation "*match_env*$_c$ G e e'", noted $e \overset{\text{env}_c}{\underset{G}{\sim}} e'$, is defined by composing the two matching properties *match_menv* and *match_trace*.

Definition 10.33 - (match_env, src/verification/MatchFPSwFPC.v:155).

match_env$_c$: *genv* → *fp_env8* → *fp_env*$_c$
match_env$_c$ G e e' ::= $(e \overset{\text{menv}}{\underset{G}{\sim}} e'.\text{mem})$
$\wedge\ (e.\text{trace} \overset{\text{trace}}{\sim} e'.\text{trace})$

As presented in Section 9.3, the generated Clight code depends on common code presented in Appendix A.2. This common code contains functions to access and modify the state of the flight plan. In order to prove the semantic preservation theorem, we thus need to consider the common C code and not only the generated code. We therefore define the following function.

Definition 10.34 - (gen_full_context, src/verification/GeneratorProperties.v:39).

gen_full_context: *flight_plan*$_s$ → *cgvars* → *prog*
gen_full_context *fps* *gvars* ::= {|
  main := auto_nav,
  defs := *Common*.global_definitions ++ *gvars* ++ (*global_definition* *fps*)
|}

The *gen_full_context* function generates a program with the global variables provided by the preprocessor, the definitions produced by the generator (auto_nav, on_enter_-

`block`...) corresponding to the input flight plan and the Clight code of common definitions contained in `CommonFPSimplified.v`. The Clight code has been produced from `common_flight_plan.c` file using the `clightgen` option of CompCert. `CommonFP.v` is the resulting file containing the Clight code of all common definitions that has been generated using `clightgen`[1]. Moreover, this file also contains some definitions from the standard C library (e.g. `___builtin_fabs` or `___compcert_i64_umod`). The module `CommonFPSimplified.v` regroups in `global_definitions` a simplified list of the common definitions, which contains only the functions and variables useful for the generator[2]. The verification time of the proof is significantly reduced when there are only the required global definitions.

Remark

We decided to set the `main` field of the program produced by the *gen_full_context* function to `auto_nav` for arbitrary reason. Indeed, this field is not used in `FPC` semantics.

We can define the Theorem 10.35, using the *gen_full_context* function, stating that the third pass preserves the semantic for any well-sized flight plan and any correct list of global variables (see Section 9.4.1).

Theorem 10.35 - (`semantic_preservation`, `src/verification/VerifFPSToFPC.v:2986`).

$$\forall\ fps\ gvars\ P,\ gen\_full\_context\ fps\ gvars = P \rightarrow bisimulation\ (semantics_s\ fps)\ (semantics_c\ P)$$

We proven the theorem by first establishing the forward simulation, executing the `FPS` semantics and the Clight semantics and applying our own induction principle over the stages, as presented in Definition 10.36.

Definition 10.36 - (`run_step_ind8`, `src/semantics/FPBigStepSized.v:1208`).

$$\frac{\forall e\ e', e \xrightarrow[fps]{\text{stage}_s} e' \Downarrow false \rightarrow P\ e\ e' \qquad \forall e\ e'\ e'', e \xrightarrow[fps]{\text{stage}_s} e' \Downarrow true \rightarrow e' \xrightarrow[fps]{\text{stages}_s} e'' \rightarrow P\ e'\ e'' \rightarrow P\ e\ e''}{\forall e\ e',\ e \xrightarrow[fps]{\text{stages}_s} e' \rightarrow P\ e\ e'}\ \text{IND}$$

This induction principle states that in order to prove a property about an initial state and the state resulting from the execution of stages, we should prove first the base case when the current state refers to a *break* stage and then the inductive case when the current

[1]The `CommonFP.v` file has been modified to use *create_ident* and *arbitrary_ident* functions for the definitions of *ident* in order to be correctly printed and the temporary variable "`_t'1`" produced has been replaced from a specific positive value (128) to "`#"t'1"`".

[2]We removed the definitions of the standard C library as we have lemmas which guarantee that a definition is in the global environment.

state refers to a *continue* stage with the induction hypothesis that the property holds for the remaining execution steps.

Due to the semantic differences between FPS and FPC and the fact that the generated C code contains arbitrary user provided code that could not be executed through the Clight semantics, we decided to use axioms to translate such arbitrary C code into a trace. Indeed, as presented in Section 9.3.3, we do not use external calls, but instead, we produce statements where the *ident* are generated using the function *arbitrary_ident*. These axioms can thus be seen as new semantic rules for Clight and are presented in Section 10.4.2. Section 11.2.3 discusses other possibilities we could use to manage this arbitrary code.

We then used the fact that Clight is deterministic when external calls are not used (see `step_deterministic_gen` lemma) to prove the forward simulation property and thus Theorem 10.35 [WPGT23].

10.2.5 VFPG correctness theorem

After having defined and proven the semantic preservation theorems for the different passes, we can finally define the correctness theorem for the flight plan generator as a whole. We need to define first the generation function producing the Clight program corresponding to the input flight plan and the global variable names.

Definition 10.37 - (generator, src/verification/VerifFPToFPC.v:53).

```
generator : flight_plan → list String → res_gen
generator fp name_gvars ::=
    let fpe := extend_flight_plan fp in
    let deroute_analysis := fb_deroute_analysis fp in
    match gvars_definition name_gvars with
      | GVARS gvars ⇒
        match size_verification fpe with
          | OK fps ⇒
            let P := gen_full_context fps gvars in
            CODE (P, deroute_analysis)
          | ERR errors ⇒ ERRORS (deroute_analysis ++ errors)
        end
      | ERR_GVARS errors ⇒ ERRORS (deroute_analysis ++ errors)
    end
```

The *generator* function is slightly different from *generate_flight_plan*, presented in Definition 9.72, which is the generation function used in VFPG. The *generator* function uses the *gen_full_context* function that adds the definitions of the common functions. This is mandatory as to prove the semantic preservation between FPS and FPC, the definitions of these functions must be known to interpret function calls to them. Despite not being the same implementation, we can trust this function to produce a program that corresponds to the program that will be executed by the autopilot. Indeed, when the C code generated will be executed, it will have in this context these common definitions.

Coq implementation

In order to improve the confidence in the proof, a future work is to generate a program using only the *generator* function and thus the C file produced will contain the generated function and the common C code. We will thus still need a common file containing the definition of the macros as they cannot be expressed in Clight. However, for readability and modularity reasons, we prefer for now to have the common C code in a separate file.

Theorem 10.38 specifies the correctness of the generator by stating that if the generator does not produce errors, then the semantics is preserved between the input flight plan and the produced program. This theorem is simply proven using the composition lemmas and the intermediate proof of the semantic preservation of the 3 different passes. The semantics preservation theorem holds under several hypotheses and axioms that are presented in Section 10.4

Theorem 10.38 - (semantic_preservation, src/verification/VerifFPToFPC.v:118).

$$\forall\ fp\ gvars\ P\ warnings,\ generator\ fp\ gvars = \texttt{CODE}\ (P,\ warnings) \\ \rightarrow\ bisimulation\ (semantics\ fp)\ (semantics_c\ P)$$

10.3 Termination proof of the auto_nav function

The auto_nav function is called at a frequency of 20Hz by the single thread autopilot program. It is thus essential to ensure that there are no risk of being blocked because of an infinite loop in the auto_nav function in order to prevent unwanted and dangerous behaviour of the drone.

The termination proof consists in ensuring that the execution will always reach a *break* stage or the end of the current block. Using the bisimulation relation provided by Theorem 10.38, the termination proof of the generated C code is equivalent to ensure that the FP *step* function terminates (see Section 7.3). Since Coq does not allow users to define non-terminating functions, the definition of the *step* semantics function of FP in Gallina proves the termination of the auto_nav function.

An important point to notice is that the execution of the flight plan depends on execution of arbitrary C code (for example the code executed in **CALL** stages). As we cannot verify C code added by users, the termination of the auto_nav function holds under the condition that all pieces of arbitrary C code eventually terminate[3].

10.4 Verification hypotheses

The proof of the semantic preservation theorem is based on a modelling of our system and some hypotheses. The goal of this section is to present these choices in order to give

[3]We will discuss this point in Section 10.4.1.

confidence in the new verified generator. Section 10.4.1 summarises the model used to define the main theorem. Section 10.4.2 presents the new Clight semantic rules defined as axioms. Finally, Section 10.4.3 presents all other axioms defined and used to verify this theorem.

10.4.1 Models of the system

The drone autopilot is a complex system evolving in a real outside environment. The `auto_nav` function is a function that interacts with the drone autopilot and indirectly with the outside environment. An ideal property we may want to prove is that the drone will always behave as defined in the flight plan. Unfortunately, proving this property would require to represent concretely the interaction between the flight plan, the drone autopilot, and the outside environment which is complex and time-consuming. Instead, we proved that the execution of the `auto_nav` function will interact with the drone autopilot as defined in the flight plan without any further guarantees that the autopilot will behave correctly.

This modelling choice is represented in the proof by the definition of the drone environment in the semantics and the use of the *eval* parameter function (see Parameter 7.20). Indeed, as presented in Section 7.3.1, we model the drone environment by two elements: *fp_state* that represents the internal states of the flight plan and *fp_trace* that represents the interactions between the flight plan and the autopilot. These interactions can be calls to functions of the autopilot but also to arbitrary C code defined by the user (for example through **`CALL`** stages). We thus assume that they do not modify the internal state of the flight plan. We also assume another hypothesis about these interactions for proving the termination of the `auto_nav` function. As the execution of the flight plan depends on the execution of arbitrary C code that cannot be verified *a priori*, the termination of the `auto_nav` function holds under the condition that arbitrary C code eventually terminates, which we consider established by other means. For instance, the WCET analysis presented in [HBL+22a] shows that on a real-world example, the function `auto_nav` always terminates and in particular, in less time than its call period.

The *eval* function describes the interaction with the outside environment that must produce a result. It takes a history t of the code already executed in the environment, that may have modified the autopilot state, and the C code c to evaluate. The function produces a deterministic result corresponding to the execution of c in an environment altered by the known history t. The *eval* function can thus be seen as an abstraction of any autopilot environment (excluding the flight plan state). The use of a history allows us to describe how this environment has evolved since the autopilot started.

This approach is similar to the partial function evaluating external calls, used to prove the determinism of the target language in CompCert [Ler09a]. A function W is used to describe the outside world. W takes the identifier of an external call with its parameter and produces the result value v and an updated world W'. The semantics for the execution of an external call is thus to use the W function to produce the updated world W' and the trace containing the external function called and its result. In our case, instead of a new function after each evaluation, the *eval* function takes the trace history to "build" the current execution environment.

10.4.2 New semantic rules

The `step` property of `FPC` semantics (see Definition 10.13) executes symbolically the Clight code of the `auto_nav` function from an initial memory state and terminates in a final memory state. During program execution, the Clight semantics produces a *trace* that corresponds to the *fp_trace* of *fp_env*. The *trace* must thus be generated by the execution of the arbitrary C code of the flight plan. In Clight semantics, the *trace*s are only generated for statements executing calls of external or built-in functions. However, arbitrary C code in Paparazzi is not restrained to these constructions. For instance, such code may appear in the condition of a conditional statement. Moreover, there is no way to use results of the *evalc* function in the Clight semantics. We thus define axioms that can be seen as an extension of the Clight semantics that take into account the model defined in the previous section by mainly adding two cases: 1) the call to a void function producing a trace, 2) the call to a function with a boolean result that returns a value using *evalc* and producing a trace.

The goal of these axioms is thus to convert some specific expressions or statements, that are generated during the third pass, into *trace*s that are equivalent to a *fp_trace* that may be produced by a flight plan semantics (`FP`, `FPE` or `FPS`). For example, as presented in Section 9.3.3, the code generated for the stage "**`CALL`** *code*" produce the Clight statement "**`Scall`**(`None`, **`Evar`**($*code*), `[]`)" and the execution of `FP` semantics will produce the trace [**`C_CODE`** *code*]. The following inference rule should thus produce from this Clight code generated, the correct equivalent *trace*.

$$\frac{t = [code\_event\ code]}{G \vdash \mathcal{S}(F,\ \textbf{Scall}(\texttt{None},\ \textbf{Evar}(\$code), \texttt{[]}),\ k,\ E,\ L, M) \xrightarrow{t} \mathcal{S}(F,\ \textbf{Sskip},\ k,\ E,\ L,\ M)}\ (\text{A:Ex})$$

However, the axioms we defined are more complex than the one presented above. This is mainly due to how we generate the code for navigation stages. Indeed, the navigation code produced for the different navigation stages is known, e.g., if we take the example presented in Section 7.2, with the `TakeOff` navigation stage, the functions called are `StartMotors()` and `TakeOffDone()`. We thus decided to produce the correct corresponding Clight expression or statement. For example, if the previous generator produced for a specific navigation stage the C code "`x = fun(a, b)`", then VFPG would generate the Clight statement "**`Scall`**(`Some#"x"`, **`Evar`**($`"fun"`), [`#"a"`, `#"b"`])". Nevertheless, the implementation of these functions are not known at generation time as they are specific to the target drone (see Section 7.3.1) and thus this type of code should appear in the *trace* when it is evaluated by Clight semantics. Unfortunately, with this type of generated code, we cannot use the inference rule (A:Ex) as it expects a Clight code of the form "**`Scall`**(`None`, **`Evar`**($`"x = fun(a, b)"`), `[]`)".

Remark

Note that in the semantics defined for `FP`, `FPE` and `FPS`, the *fp_trace* generated for the execution of this navigation stage example would thus be [**`C_CODE`** `"x = fun(a, b)"`].

In the following, we will thus present the 4 axioms used in the correctness proof that are compatible with our generation of arbitrary code and navigation code. Section 11.2.3 discusses how we could have generated the navigation code differently in order to simplify the axioms.

In order to present these axioms, we first need to define functions to generate code *event* for C code calls and a relation between elements of `FPS` semantics and Clight expressions, we also use the following two notations for the generation of Clight *ident*:

Definition 10.39 - Notation for the generation of *ident*.

$\#name ::= create\_ident \quad name$
$\$name ::= arbitrary\_ident \quad name$

First, the function *code_event_optassign* defined below is a generic function returning Clight events that either correspond to function calls with a potential assignment or to C code to execute. The events generated should be equivalent to a `FPS` trace and are thus produced using the *code_event* function (see Definition 10.28).

Definition 10.40 - (`code_event_optssgn`, `src/semantics/FPEnvironmentClight.v:119`).

code_event_optassign: option *ident* → *c_code* → option (list *c_code*) → *event*

This function takes a potential variable, the name of the function or the arbitrary C code and potential parameters. The function then produces the corresponding event. We will not formally define this function but here are some examples of produced events:

- *code_event_optassign* `None "a + 1" None` = *code_event* `"a + 1"`
- *code_event_optassign* (`Some #"x"`) `"a + 1" None` = *code_event* `"x = a + 1"`
- *code_event_optassign* (`Some #"x"`) `"fun" (Some [])` = *code_event* `"x = fun()"`
- *code_event_optassign* `None "fun" (Some ["a"; "b"])` = *code_event* `"fun(a, b)"`

Second, we define the relation *match_param* between a Clight expression and an `FPS` *event* that is a *String* describing the code added in the *fp_trace*.

Definition 10.41 - (`match_param`, `src/verification/ClightLemmas.v:150`).

match_param: *expr* → *String* → *Prop*
match_param e s, noted $e \overset{\text{param}}{\sim} s$ is defined by the following inference rules:

$$\frac{}{\texttt{Evar}(\$code) \overset{\text{param}}{\sim} code}\ (\textsc{Evar}) \qquad \frac{string\_of\_int\ v = s}{\texttt{Econst}(\texttt{int}\ v) \overset{\text{param}}{\sim} s}\ (\textsc{EconstI}) \qquad \frac{string\_of\_float\ v = s}{\texttt{Econst}(\texttt{float}\ v) \overset{\text{param}}{\sim} s}\ (\textsc{EconstF})$$

$$\frac{e_1 \overset{\text{param}}{\sim} s_1 \qquad e_2 \overset{\text{param}}{\sim} s_2 \qquad s_1 ++ "*" ++ s_2 = s}{\texttt{Ebinop}(\texttt{Mult}, e_1, e_2) \overset{\text{param}}{\sim} s}\ (\textsc{Ebinop})$$

Rule (Evar) states that an **Evar** expression whose identifier is produced by the *arbitrary_ident* function from the string *code*, is equivalent to the *code* event. Similarly, the expressions containing integer or floating-point values are equivalent to the corresponding string representing the numerical value (rules (EconstI) and (EconstF)). The **Ebinop** expression, described by the rule (Ebinop), is equivalent to the evaluation of the two expressions concatenated with the string character describing the binary operation. Note that the definition of *match_param* is defined not exhaustively (e.g. in the **Ebinop** case, the equivalence relation is only defined for the multiplication) as it is used by the axioms we do not want them to be too permissive. We thus only define the necessary cases for the verification of the generator.

We then extend this property with the relation *match_params* for the analysis of an expression list.

Definition 10.42 - (match_params, src/verification/ClightLemmas.v:165).

match_params: list *expr* → list *String* → *Prop*

The property *match_params e s*, noted $e \overset{\text{params}}{\sim} s$ is defined by a classical induction principle (see the following inference rules).

$$\frac{}{[] \overset{\text{params}}{\sim} []}\ (\textsc{Nil}) \qquad \frac{e \overset{\text{param}}{\sim} s \qquad le \overset{\text{params}}{\sim} ls}{e :: le \overset{\text{params}}{\sim} s :: ls}\ (\textsc{Cons})$$

Finally, we define a special relation when there are optional parameters.

Definition 10.43 - (match_optparams, src/verification/ClightLemmas.v:176).

match_optparams list *expr* → option (list *String*) → *Prop*

match_params e s, noted $e \overset{\text{optparams}}{\sim} s$, is defined by the two following inference rules:

$$\frac{}{[] \overset{\text{optparams}}{\sim} \texttt{None}}\ (\textsc{None}) \qquad \frac{le \overset{\text{params}}{\sim} ls}{le \overset{\text{optparams}}{\sim} (\texttt{Some}\ ls)}\ (\textsc{Some})$$

Axioms can be divided into two categories: 1) axioms representing rules for call producing traces, 2) axioms for statements producing a boolean result.

Axiom 1 describes the execution of arbitrary C code or function call. This axiom states that the execution of a **Scall** produces the corresponding trace using the *code_event_optassign* function. The *match_optparams* relation is used to recover the string version of the parameter from the list of parameter expressions.

Axiom 1 - (step_arbitrary_call_gen, src/verification/ClightLemmas.v:194).

$$\frac{\overrightarrow{v_{args}} \overset{\text{optparams}}{\sim} optparams \qquad t = [code\_event\_optassign\ optvar\ func\ optparams]}{G \vdash \mathcal{S}(F, \textbf{Scall}(optvar, \textbf{Evar}(\$func), \overrightarrow{v_{args}}), k, E, L, M) \xrightarrow{t} \mathcal{S}(F, \textbf{Sskip}, k, E, L, M)}\ (\text{A:}\textsc{Call-1})$$

Coq implementation

We could wonder if the *match_optparams* relation could be directly implemented as a function producing the corresponding trace. However, in the proof of the generation pass, the values in the trace are already known (as we proved first the forward simulation property) and thus the *optparams* value is automatically obtained when applying this axiom. We just then have to prove that they are matching though the relation *match_optparams*.

Axiom 2 is similar to the previous axiom and states that the **Sassign** statement can be directly converted into a trace. The **Sassign** statement is mainly used for the code generation for the **SET** stage that contains arbitrary code for the variables to set.

Axiom 2 - (step_arb_assign_C_code, src/verification/ClightLemmas.v:291).

$$\frac{expr \overset{\text{param}}{\sim} value \qquad t = [code\_event\_optassign\ var\ value\ \texttt{None}]}{G \vdash \mathcal{S}(F,\ \texttt{Sassign}(\texttt{Evar}(var),\ expr),\ k,\ E,\ L,\ M) \xrightarrow{t} \mathcal{S}(F,\ \texttt{Sskip},\ k,\ E,\ L,\ M)}\ (\text{A:ASSIGN})$$

The next two axioms belong to the second category of axioms and allow code evaluation with the *eval* function. They both follow the same principle, the Clight semantics transition from state S to S'' with the trace t is defined if several requirements are satisfied. First, the state S should contain a statement with an expression that requires the evaluation of arbitrary C condition c and the memory state should be the field `mem` of the fp_env_c env. Then, the evaluation of the code c with the function *eval* and the trace t (that should be equivalent through the relation $\overset{\text{trace}}{\sim}$ to $env.\texttt{trace}$) produces the boolean value b. We note env' the same environment as env where the event corresponding to the evaluation of the code is appended to the trace. We note also S' the same state that S except for the expression containing the arbitrary C code in the statement that has been replaced by a constant expression with b, but also the memory state is now $env'.\texttt{mem}$. Finally, it must exist a relation in the Clight semantics describing the execution from the state S' to the state S'' that produces an empty trace.

Axiom 3 is a generic version for the evaluation of arbitrary conditions as presented previously. Indeed, it is defined for every function f_stmt that takes an expression and produces a statement. This axiom is used for the evaluation of condition inside conditional statement or **Sset**. Notice that we use the same notation $env(\!|T|\!)$ than for *fp_env* which appends the trace T in the environment env.

Axiom 3 - (eval_c_code_cond, src/verification/CommonFPVerification.v:83).

$$\frac{\begin{array}{c}(f\_stmt : expr \to statement) \quad (env,\ env',\ env'' : fp\_env_c) \\ t \overset{\text{trace}}{\sim} env.\texttt{trace} \qquad eval\ t\ fun = b \qquad T = [cond\_event\ fun\ b] \qquad env' = env(\!|T|\!) \\ G \vdash \mathcal{S}(F,\ f\_stmt\ \texttt{Econst}(int\_of\_bool\ b),\ k,\ E,\ L,\ env'.\texttt{mem}) \xrightarrow{\epsilon} \mathcal{S}(F,\ \texttt{Sskip},\ k',\ E',\ L',\ env''.\texttt{mem})\end{array}}{G \vdash \mathcal{S}(F,\ f\_stmt\ \texttt{Evar}(\$fun),\ k,\ E,\ L,\ env.\texttt{mem}) \xrightarrow{T} \mathcal{S}(F,\ \texttt{Sskip},\ k',\ E',\ L',\ env''.\texttt{mem})}\ (\text{A:ECond})$$

Despite being generic, the previous axiom does not work with **Scall** statements as the expressions in these statements are the function name or its parameters. Replacing this axiom by a constant value will not have the expected behaviour as the Clight semantics will not have any rule to interpret this new statement. To address this problem, we define Axiom 4 for the evaluation of arbitrary boolean condition in **Scall** statements. These statements are produced mainly during code generation of navigation stages, as explained previously.

Axiom 4 - (eval_call_res, src/verification/CommonFPVerification.v:209).

$$\frac{\begin{array}{c}(env,\ env',\ env'' : fp\_env_c) \\ optid = \texttt{Some}\ var \qquad expr = \texttt{Evar}(\$fun) \\ code\_event\ call = code\_event\_optassign\ \texttt{None}\ fun\ (\texttt{Some}\ \overrightarrow{params}) \qquad t \overset{\text{trace}}{\sim} env.\texttt{trace} \\ eval\ t\ call\ =\ b \qquad T = [cond\_event\ call\ b] \qquad \overrightarrow{v_{args}} \overset{\text{params}}{\sim} \overrightarrow{params} \qquad env' = env(\!|T|\!) \\ G \vdash \mathcal{S}(F,\ \texttt{Sset}(var,\ \texttt{Econst}(int\_of\_bool\ b)),\ k,\ E,\ L,\ env'.\texttt{mem}) \xrightarrow{\epsilon} \mathcal{S}(F,\ \texttt{Sskip},\ k',\ E',\ L',\ env''.\texttt{mem})\end{array}}{G \vdash \mathcal{S}(F,\ \texttt{Scall}(optid,\ expr,\ \overrightarrow{v_{args}}),\ k,\ E,\ L,\ env.\texttt{mem}) \xrightarrow{T} \mathcal{S}(F,\ \texttt{Sskip},\ k',\ E',\ L',\ env''.\texttt{mem})}\ (\text{A:CallR})$$

Coq implementation

It is important to note that the presented version of the Clight syntax does not contain any type information and thus the corresponding axioms do not verify that the expressions are well-typed. However, in the Coq source code there are extra hypotheses to ensure well-typedness of expressions. For example, Axiom 4 verifies that the type of the returned value is a boolean value.

Using these axioms, we do not need to use the external call mechanism of Clight. We then were able to use the principle that Clight is deterministic for code without external code (see `step_deterministic_gen` lemma [WPGT23]) to prove the backward simulation property and thus the correctness Theorem 10.38.

10.4.3 Axioms used for the proof

We use a parameter function called *create_ident* that takes a *String* and produces a Clight *ident* during the code generation pass. Clight *idents* are used in the Clight syntax to describe variable or function names. This function is a Coq `Parameter`, i.e. it is defined in OCaml and linked during the extraction. As explained in Section 9.3, it has to be defined

in OCaml as the corresponding *String* is stored in a hashtable. The `PrintClight` function uses the hashtable to print the corresponding name of variables and functions. As this function is defined outside Coq, it is not possible to prove properties on it. We thus add the Axiom 5 stating that the *create_ident* function is injective which is reasonable knowing the OCaml implementation.

Axiom 5 - (`create_ident_injective`, `src/generator/ClightGeneration.v:63`).

$$\forall\ s_1\ s_2,\ \mathit{create\_ident}\ s_1\ =\ \mathit{create\_ident}\ s_2 \rightarrow\ s_1\ =\ s_2$$

A similar axiom has been added for *arbitrary_ident* function. A similar solution was proposed to be added directly in CompCert[4]. However, it was not kept because the equivalent function of *create_ident*, noted `ident_of_string` is not computed within Coq as it is an OCaml function, but also the fact that different runs of CompCert or of `clightgen` can use different mappings of identifiers to strings.

We also use some classical axioms from Coq standard library such as proof irrelevance stating that two proofs are equal if they prove the same property $(\forall\ (P : \mathit{Prop})\ (p_1\ p_2 : P),\ p_1 = p_2)$ and functional extensionality stating the equality of functions is pointwise equality $(\forall\ A\ B\ (f\ g : A \rightarrow B),\ (\forall x,\ f\ x\ =\ g\ x) \rightarrow\ f\ =\ g)$. These axioms are particularly necessary when manipulating dependent types, see Section 11.2.

Finally, the proof of the semantic preservation theorem depends on other Coq axioms[5] that are not used for the verification of VFPG but comes from the CompCert project as we use Clight semantics. These axioms are for instance the excluded middle axiom $(\forall P,\ P\ \vee\ \neg P)$ or axioms from the library `ClassicalDedekindReals`.

[4] See the following pull request in CompCert project: https://github.com/AbsInt/CompCert/pull/222

[5] This can be observed using the Coq command: `Print Assumptions semantic_preservation.`

Chapter 11

Discussion

Contents

11.1 Development methodology 237
11.2 Technical remarks 238
11.2.1 Design choices made during development 238
11.2.2 Use of Clight 239
11.2.3 Arbitrary C code 239
11.2.4 MathComp Library 242
11.3 Applicability on real-world projects 242

The main goal of this thesis is to review formal verification processes in order to verify critical components of the Paparazzi autopilot. We also want to see if the verification processes can be applied on a real-world application. In the previous chapter, we presented how we formalised and verified the new flight plan generator using compiler verification techniques. This chapter aims to discuss the verification process and its applicability.

Section 11.1 summarises our Coq development approach. Then, Section 11.2 comments on some technical choices made and possible alternatives. Finally, Section 11.3 presents the different steps of the verification process and discusses its applicability.

11.1 Development methodology

The verification of a compiler using a proof assistant such as Coq is a well-tested and tried technique. Some of the projects using it are mature enough to be used in critical software development (see CompCert [LBK+16] for instance). Even if its complexity cannot be compared to projects like CompCert or Velus, the FPL code generator is not a toy example.

The context of the Paparazzi autopilot has constrained our formal development in several ways. First, the FPL input language was fixed and already in use, so we were obliged to keep its quirks[1] and propose only conservative extensions, instead of a global redesign that could have been helpful in order to ease the verification effort. Second, we were also tied to a specific C code structure that users may rely upon. Third, we needed as soon as possible to devise an executable semantics and present it to the original FPL designers for validation and feedback, which led us to a simple FP semantics without a formal model of external C code.

[1] For instance, the *fp_state* only stores one previous position. If we execute two **DEROUTE** then two **RETURN** stages, the first return rolls back from the second deroute as expected, but the second return does nothing.

Developing such a project with Gallina forced us to a deep clarification of some rather tedious semantic details and incidentally unveiled a number of issues in the original compiler. For instance, writing down the proof allowed us to find a bug when there are more than 256 blocks or stages. Also, in the original semantics of FPL based on the previous generator, we found that the **DEROUTE** stage has an unexpected behaviour: when it is executed, the current position is stored in `last_block` and `last_stage`, and the current block becomes the derouted block. Therefore, when there is a **RETURN** stage executed in the derouted block, the execution resumes at the stored position, so the same **DEROUTE** stage is executed again and the program enters an infinite loop. This issue was only found during the formalisation of the semantics as the **RETURN** stage is rarely used by Paparazzi users and when so, it is only to resume the plan execution after an exception was raised.

Finally, verification of the generator by ensuring semantic preservation is really time-consuming and on-the-fly changes must be avoided as much as possible. Indeed, slight modifications in functions or in the specifications can impact several proofs that must be updated. However, we try to reduce and contain this problem by splitting the code generator into three independent passes so that the proof effort for each pass is manageable, but also so that they are independent, i.e. the different passes are unlikely to all change, whatever the development hazards of the other passes.

11.2 Technical remarks

The purpose of this section is to discuss various technical decisions made throughout the development. These decisions concern specific design choices, the use of Clight, our modelisation of arbitrary C code and the MathComp library we used.

11.2.1 Design choices made during development

In Section 7.3, for the sake of readability, the semantics of FP is given through a relational presentation with inference rules, but we decided to implement the semantics in Coq as a *step* function. Having a function makes it possible to automate and simplify proofs by using normalization by evaluation instead of manually applying many rewriting rules. An OCaml interpreter can also be obtained by mere extraction of the Coq function. The interpreter can then be used for early validation purposes. Indeed, it can be executed on example to show its behaviour in order to be validated by Paparazzi users and we can compare the execution with the code produced by the current OCaml generator.

Another important point is that we wanted to avoid gratuitously assuming new axioms about the correctness of non-Gallina phases, such as preprocessing. For example, blocks are numbered during preprocessing but no guarantee is obtained on the result at the Coq level. Therefore, we used dependent types associated with verification functions that inject a structure from the base type if its correctness property holds, or return an error otherwise. For instance, the size verification pass (cf. Section 9.2) ensuring that the flight plan is well-sized and well-numbered or the analysis of global variables with the *gvars_definition* function ensuring that the variable names to define do not clash with the functions of the flight plans (cf. Section 9.4.1).

11.2.2 Use of Clight

We used the Clight language from CompCert as the output language because it has many advantages. First, it provides a trustworthy semantics for C. Indeed, as presented in Section 5.2, CompCert offers two semantics: a small-step one and a big-step one. On the one hand, the big-step semantics is the natural semantics describing the run-to-completion model of a program. This semantics is easier to work with, but unfortunately, it does not define the behaviour of all Clight statements such as `goto`, which is used for loops. On the other hand, the small-step semantics, that we used, describes the behaviour of all Clight statements. However, the small-step semantics only describes one step of execution and the remaining code to execute is stored in a continuation. The execution of a whole program then corresponds to a succession of small steps which was tedious for verifying semantic preservation, as our own semantics is direct and do not involve continuations.

Second, CompCert offers the `clightgen` tool that takes a C file and converts it into a Clight structure. This tool can help the user to understand Clight and its differences with C, by translating existing C programs. For instance, the `&&` boolean operator and some other constructions do not exist in Clight (see Section 9.3.3).

Finally, the Clight code generator can be plugged into CompCert, such as in the Velus project [BBP20], in order to have a complete verified compiler toolchain without too much effort. In our case, as Paparazzi is compatible with several types of drones, the target architecture may be exotic and not supported by CompCert. We thus prefer to produce C code with the `PrintClight` module of CompCert. Any cross-compiler toolchains of Paparazzi can thus be used in order to produce the embedded binary. However, as presented in Section 9.3.3, the module does not produce compilable C code and some transformations have to be performed during post-processing (cf. Section 6.2.2).

11.2.3 Arbitrary C code

After the development of the new generator in Coq and in order to verify it, we started by specifying the formal semantics of `FP` before having precisely studied Clight semantics. We thus defined a denotational semantics and we decided to manage external calls using the *eval* function (cf. Parameter 7.20). However, Clight semantics for external calls does not correspond to our model. We thus have to define our own Clight semantics for arbitrary C code and external functions, as presented in Section 10.4.2. Also, our modelisation choices naturally split the flight plan memory states from the drone states (represented by the trace of events), thus we avoid dealing with separation logic.

This solution allows us to prove the semantic preservation of the generator using the original `FP` semantics. However, this solution is mainly based on our choice to have an *eval* function for the evaluation of arbitrary C code and on the hypothesis that the execution of this code will terminate and not modify the state of the flight plan. In the following, we first justify the design choice of the *eval* function, and then we discuss possible alternatives for the evaluation of arbitrary C code. Finally, we present potential solutions to ensure the hypotheses we have on the arbitrary code.

We initially defined a semantics in which the *eval* function takes a *time* parameter (a

natural integer value) instead of the trace. This function thus produces a new *time* in addition to the result value. The *fp_env* was therefore composed of a state and a variable storing the current *time* and the *evalc* function updates the *time* instead of the trace. The idea was to represent the fact that the execution of the same code twice might not produce the same result (e.g. the function returning the battery level might not return the same value). Note that despite having this *time* value, the semantics will still produce a trace that is not stored in the environment, like in Clight semantics. However, when we wanted to verify semantic preservation of the `FPS` to `FPC` pass, we faced some problems while trying to define a relation between the *time* variable and the memory. Indeed, it was not possible to store *time* safely in the memory as the value is unbounded. We could have defined an environment like fp_env_c that is composed of a memory state and *time*, but this would have required to define a matching relation for the states, the *time* value and also the traces (as they would have been generated by `FPS` and `FPC` semantics). We thus prefer our solution consisting in storing a history of external calls.

A point that may be questionable is whether the *eval* function is a good abstraction for the evaluation of C code in a non-deterministic environment such as the drone autopilot, as Paparazzi allows hardware interruptions that might modify the drone memory by updating, for instance, the battery voltage value. **On one hand**, we can consider that *eval* is not a good abstraction function as it is a Coq function, and thus is deterministic. However, the *eval* function is a Coq Parameter and thus its definition is not known. At the Coq level, we cannot predict or prove any property about its behaviour. As presented in Section 10.4.1, the *eval* function can thus be seen as an abstraction function describing any initial drone environment and the external call history describes its evolution in a non-deterministic environment. Indeed, an element of the history is an external call that may modify non-deterministically the drone environment. For example, an external call that updates the internal memory with a new value acquired by a sensor is not deterministic. This is a similar approach that the one presented in Section 10.4.1 and used in CompCert [Ler09a] that defines a W function which describes the outside environment and after every execution, a new environment is produced, updated by the effect of the external call. We could have modelled the addition of external events in the trace at any time during the execution. These events would correspond to the modifications of the memory state caused by potential hardware interruptions. This modelisation would have been more realistic but more complex to set up and it would not have improved the guarantees. Indeed, in the flight plan semantics, we made sure that the history is updated at every call to *eval* function. If the flight plan contains two following **`CALL`** stages with the same abstract code, they will be evaluated with different histories and at the Coq level, we cannot prove that they produce the same result. In future work, we may want to ensure this property, i.e. the execution of the `auto_nav` function through the `FP` semantics will never call twice the *eval* function with the same history. **On the other hand**, we might find inconsistency problems while having a Coq function (which is deterministic) that is a Parameter and thus its implementation can be non-deterministic (if the function is linked to a non-deterministic function). Depending on the implementation of the *eval* function and if we use the property that a Coq function is deterministic, we can thus expect to prove `false`. However, we have at least two arguments

which allow us to have confidence in our model: 1) as explained previously, at the Coq level, the implementation parametric function is not known, and thus we cannot predict its behaviour 2) we can have inconsistency if the proof relies on the determinism of the *eval* function (i.e. using property such as "$eval\ t\ c\ =\ b \wedge eval\ t\ c\ =\ b' \rightarrow b = b'$") when it is not the case. For example, we cannot use this property in proof of lemmas that compare two executions of the same flight plan. Indeed, despite having the same trace, they might eventually diverge at a certain point as they may have not the same outside environment. However, if we only consider one execution of a flight plan, such as during the semantics preservation proof, then we can suppose that the *eval* function is deterministic.

The current axioms used to extend the new Clight semantics for the evaluation of arbitrary code can seem complex and may reduce confidence in the proof for external readers. In order to simplify the axioms and increase the trustworthiness of the proof, we might want to generate differently the Clight code for the navigation stages. We actually consider them as real statements, but in `FP` semantics the navigation function directly produces a trace corresponding to the function to call. In other words, as presented in Section 10.4.2, the `FP` semantics may produce the trace "**`C_CODE`** `"x = fun(a, b)"`" for a navigation stage but the code generated would be "**`Scall`**(`Some#"x"`, **`Evar`**($`" fun"`), [#`"a"`, #`"b"`])". We could then modify the code generation and instead produce a statement similar to the abstract code such as "**`Scall`**(`None`, **`Evar`**($`"x = fun(a, b)"`), `[]`)". We therefore can have only two axioms one comparable to (A:Ex) and one similar for the evaluation of conditions.

An alternative solution we think about for the evaluation of arbitrary C code is to use the same principle as the Clight semantics for the evaluation of external calls. In this case, we would have evaluated the arbitrary C code using a property $eval_{prop}$ defined by the following inference rule if we want to use the same definition of *fp_env*.

$$\frac{e\{\texttt{trace} := e.\texttt{trace} +\!+ [\textbf{COND}\ (cond,\ b)]\} = e'}{eval_{prop}\ e\ cond\ b\ e'}\ \text{(NA:Cond)}$$

This property states that the evaluation of the code *cond* from the environment e terminates in the environment e' and produce the output b if e' is the same environment than e where the trace "[**`COND`** $(cond,\ b)$]" has been appended. Using this definition, we could expect using directly the Clight semantics without having to append axioms as they would have similar definitions. However, we thought about this solution too late and it would have required a massive work to translate the denotational semantics of the flight plan into an operational semantics, similar that the one presented in this document, and mainly we would have to redo or adapt the whole semantic preservation proof.

A key point of our modelisation is that we consider that the evaluation of arbitrary C code will always terminate and will not modify the state of the flight plan. Despite supposing this hypothesis, we have several ideas to ensure this property or to increase its trustworthiness.

The first idea is to use separation logic by redefining the axioms corresponding to the

extension of the Clight semantics. This allows us to formally separate the memory into two distinct elements: the state of the flight plan and the memory environment of the drone autopilot. The new axioms thus state that the execution of the abstract code modify only the drone environment part and produce a new memory state. In order to ensure that this is not just a modelisation choice but that the executed code only modifies the drone environment, we can execute the abstract code in a contained environment that has only access to specific parts of the memory.

The second idea is to verify the abstract C code. We can first add a parsing function for the abstract C code to Clight code, in order to ensure that the code provided by the user is semantically correct. During the parsing, we can also restrict the code provided to some Clight construction such as the one supported by the extension of the Clight semantics or to new construction by adding new axioms. Then, we can also verify the Clight code produced by using static analysis techniques. We could thus ensure several properties: 1) that the code does not contain any call to common flight plan functions (defined in Appendix A.2) and does not modify the state of the flight plan 2) that the code terminates and 3) that the code cannot have runtime errors.

11.2.4 MathComp Library

In addition to CompCert implementation, we also used MathComp, a Coq library of formalised mathematics. We used some MathComp lemmas about arithmetics on natural numbers but we mainly used `seq`, a library about lists that provides a lot of useful functions and lemmas such as the `drop` function that removes the first elements of a list. We also use SSReflect, a proof language that allows us to simplify proofs compared to standard Coq proof tactics. The main problem we faced using MathComp concerns program extraction. Indeed, the MathComp library is not designed to be extracted into OCaml code as it depends on constructions that are not supported, e.g. the extraction of tuple.v that deal with a specific kind of inductive type that has logical parameters, is not supported in the Ocaml extraction. We thus have to manually specify the only functions that need to be extracted from MathComp.

11.3 Applicability on real-world projects

Our compiler project is not just another example of a verified code generator proven in Coq. In addition to having designed and formally verified a critical component of Paparazzi we wanted also to determine if such a development could be applied in an industrial context, i.e. if the technique presents a number of advantages for a minimal cost.

With this project, we came to the conclusion that completely developing and formally proving a code transformer in Coq implies mastering three main competences that correspond to the three development steps of the project: 1) developing a project using Coq, 2) formalising a programming language and its semantics, and 3) verifying a program with a proof assistant. Notice that these three competences can be used alone or associated according to the needs and constraints of the project.

First, developing a project with Gallina allows us to reduce the number of errors. As Gallina is a pure functional language that does not have side effects (for example, global variables), code development is made easier. Notice also that there are no risk of infinite loop as Coq does not authorize the definition of non terminating functions.

Secondly, the formalisation of the semantics requires a good understanding of the input language. This allows the detection of unwanted or unspecified behaviour. For example, when we have defined the semantics of `FPL` based on the previous generator, we found that the **`DEROUTE`** stage has an unexpected behaviour: when it is executed, the current block and current stage (not the next stage) are stored as `last_block` and `last_stage`, and the current block becomes the derouted block. Therefore, when there is a **`RETURN`** stage executed in the derouted block, the execution will continue at `last_block` and `last_stage`, so the same **`DEROUTE`** stage will be executed again and the program enters an infinite loop[2]. In addition to the detection of unwanted behaviour, formally defining a language semantics in Coq forces the language designer to carefully consider all the possible cases and therefore avoid undefined behaviour. Finally, even if the semantics is not used to verify the generator, it can at least be used as a clear documentation or even as a model to verify properties about the programs.

Finally, the verification of the generator by ensuring semantic preservation is really time-consuming and requires some advanced skills in proof theory and Coq. Indeed, it is necessary to formalise the problem such that the proof effort will be minimized for the user. Then the proof can be written in Coq. For instance, in our project, the main difficulty was to find a relation between the `FPL` structure and the generated C code. We can of course find bugs when writing the proof. In our case, writing the proof allowed us to find a bug when there is more than 256 blocks or stages. In addition, having a semantic preservation proof for a multiple-passes compiler ensures that the semantics of the different languages (input, output and intermediate languages) of the generator behave similarly. This can be useful to verify properties on one of these languages and to have the guarantee that the output code will also satisfy the property. We use this principle to have guarantee on the generated code by verifying properties on `FPS`. For instance, we verified [WPGT23] that we can recover from incorrect states (see Section 7.3.3) or that the `on_enter` and `on_exit` code is always correctly executed when the block being executed changes. All the properties verified on `FPS` are gathered inside the `FPSProp.v` file. Finally, when the generator is proven, its users have more confidence in the program and it may facilitate the certification process for critical programs.

An interesting alternative to verifying a compiler, that might take less time to develop, is to develop a verified validator in the spirit on what we have done in the size verification pass. The idea would be to specify the expected property about a language, then develop a program that tests if the property is satisfied and finally prove that if the program does not return errors, then the property is guaranteed for the input code.

[2]Infinite loop in the sense of the stage executed. The `auto_nav` function will always terminate (after the execution of the **`DEROUTE`** or **`RETURN`** stages).

Chapter 12

Conclusion

The objective of this PhD thesis was to review various verification processes that use formal tools to verify critical components of the Paparazzi autopilot and ensure correctness properties. During these 3 PhD-years, we have been able to verify 2 critical components. First, we have verified the absence of runtime errors and some functional properties of a C mathematical library. This work is summarised in Section 12.1. The second part of the thesis, resumed in Section 12.2, is the verification, using Coq, of the flight plan generator translating an XML flight plan into embedded C code.

Along with providing formally verified components to Paparazzi users, this thesis seeks to determine whether these processes could be used on existing projects. We summarise in Section 12.3 the pros and cons we have discovered during our work. We also discuss possible future works.

12.1 A mathematical library for state representation

Paparazzi autopilot is composed of a *State Interface* which is a black box linking data acquired by sensors and estimation filters to the control system using these data to make navigation decisions. The main feature of this state interface is to proposes several representations for a given data and conversion functions when needed. This interface is thus a critical component of the autopilot as errors during the conversion can result in bad decisions made by the navigation system which can lead to a crash of the drone in the worst case.

During this thesis, we have focused on the verification of a C mathematical library used in the state interface for the manipulation of attitude or position representations (cf. Part I). This library provides several representations (Euler angles, rotation matrices and quaternions) as well as functions for manipulating and converting these representations. The representations are also defined in three different formats to represent real values: fixed point, floating point or double.

In order to verify this C library, we have used the Frama-C platform associated with its RTE plugin (to add assertions corresponding to runtime errors, i.e. division by 0, null pointer dereferencing, overflow, etc.), its WP plugin (to apply deductive methods) and its EVA plugin (to apply abstract interpretation). Our work has first consisted in verifying the absence of runtime errors, by determining minimal contracts that can ensure the assertions added by the RTE plugin. We had to associate both WP and EVA plugins to verify the assertions as they are complementary: EVA is able to manage pointers and compute intervals of possible values while WP is able to prove loop annotations. The second part of the verification of the library was to ensure functional properties about these functions. We

Implementation	# lines			
	Code	Annotations	Percentage	Coq proofs
`int`	1150	903	44.0%	-
`double`	358	199	35.7%	-
`float`	1708	1557	47.7%	6472*
Total	3216	2659	45.3%	6472*

*Total lines of Coq, but the majority of them was automatically generated.

Figure 12.1: Lines of code and annotations added.

have mainly focused on the rather complex conversion functions and we have ensured that the implementation is mathematically correct. Indeed, as we have used the `real` arithmetic model of WP, we can conclude that the function is correct if we do not take into account rounding errors due to floating-point arithmetics. In order to prove these properties, we have mainly used automatic solvers such as Alt-Ergo, CVC4 or Z3, but for some specific properties, we had to manually prove them using Coq.

Figure 12.1 shows the number of code lines of initial files (with code and comments but without blank lines) and the number of annotation lines added. Adding annotations almost doubles the number of lines. This massive increase is mainly due to the mathematical specifications which are in some way a second definition of the C functions. Figure 12.1 also shows the number of code lines for Coq files, but the majority of these files are automatically generated by Frama-C, and thus the manual proof is significantly smaller than 6472 lines. Finally, Figure 12.1 does not take into account WP scripts automatically generated when saving tactics/solvers applied for verified goals, representing around 8746 lines.

In future work, we plan to complete the remaining proofs presented in Section 4.3. Some functional properties of other functions from the same mathematical library of Paparazzi should also be verified. We want especially to focus on verifying rounding errors, and therefore not to use WP `real` model but rather a model that accurately represents floating-point numbers. We should also compare our approach to *autoactive proofs*, where interactive provers are not used, but SMT solvers are guided by assertions inserted by developers to help the provers [BLK19, DM17].

12.2 Flight Plan Generator

Flight plans describe specific missions that the drone must perform autonomously. These flight plans are expressed using a high-level, XML-based domain specific language. Originally, Paparazzi used an OCaml generator that converts the missions into C code directly embedded in the drone. The generator is a critical component of Paparazzi as it must produce a code that behaves exactly as specified by the user to avoid unwanted behaviours.

Part II presents the development and the formal verification of a new flight plan generator for the Paparazzi autopilot. This new generator is fully compatible with the

	Lines of code	Percentage
Coq Parser	2789	13.1%
Preprocessor	2174	8.6%
Coq generator	20238	80.1%
Postprocessor	60	0.3%
Total	25261	100%

Figure 12.2: Lines of Code for the different generator blocks.

previous unproven generator but supports new features such as forbidden deroutes and detects erroneous flight plans which were previously unnoticed. During the early development stages, we developed some tests showing that the C code generated by the two generators behave similarly on several examples. This generator is composed of 4 main blocks and Figure 12.2 shows their size differences in terms of lines of code[1]. Note that the original OCaml generator was roughly 1200 lines of code and both projects use common Paparazzi code amounting to approximately 3000 lines of code. The verified parser, written using `menhir` with the `-coq` option, reads the XML file in order to produce a *Flight Plan Parsed* structure. This structure is then slightly transformed by an OCaml preprocessor into the `FP` Coq structure. Then the major transformations occur in the verified Gallina generator, which correspond to 80% of the project. The generator converts the `FP` structure into the corresponding Clight code. Finally, the Clight code is printed, and some minor transformations are performed by the OCaml postprocessor, in order to produce compilable C code. Both preprocessor and postprocessor are the only parts of the generator that are not formally verified. However, we are confident that these passes do not question the correctness of the generator as they only perform minor transformations, and they correspond to only a small part of the generator.

The Gallina generator corresponds to 80% of the VFPG project as we count the code generating the Clight code, but also the specification and the correctness proof. The generator is a 3-pass compiler. The first pass extends the flight plan in order to have a structure closer to the C code that will be generated. Then, the second pass ensures that the flight plan is not too big, to avoid overflow, but also verifies preprocessor transformations, such as the numbering of the blocks. Once verified, these properties are carried on in the remaining stages as they are required to complete some proofs. Finally, the third pass generates the Clight code. We have proven independently that each pass preserves the semantics, and we have used a composition lemma to prove the correctness of the whole Gallina generator. The main difficulty encountered was to manage the arbitrary C code that users can add into the flight plans. We have decided to add an hypothesis: the arbitrary C code provided by the user always terminates and does not modify the internal state used for the flight plan execution. Under this assumption, the C code generated will behave as the input flight plan.

[1]We only count the lines of the VFPG project and not the external file used such as `PrintClight.ml` from CompCert.

	Compilation	Specification	Proofs	Total
Flight Plan (FP)	84	296	0	380 (1.9%)
Extension pass	416	1306	2239	3456 (19.5%)
Size Verification pass	172	1124	2160	3456 (17.1%)
Clight generation	1542	2239	4491	8272 (40.9%)
Common	616	1689	1864	4169 (20.6%)
Total	2830 (14.0%)	6654 (32.9%)	10754 (53.1%)	20238 (100%)

Figure 12.3: Lines of Coq code in the Gallina generator (comments and blanks excluded).

Figure 12.3 shows the number of Coq code lines for FP, the different passes and the common code. The biggest part concerns the Clight generation as the transformation of the flight plan structure into Clight code is not straightforward and the semantic preservation proofs require to manually execute the Clight code through its semantics. In Figure 12.3, we also split the number of code lines into several categories to highlight that the working Coq code only corresponds to 14% of the project, everything else being specifications or proofs. The proof of general properties about the language, such as the correct execution of `on_enter` and `on_exit` primitives, are counted in the "Common" line.

As future work, we will consider supporting new types of error and warning messages. We are currently specifying and implementing warning messages when forbidden deroutes are blocking. We want to warn the users when there is a `deroute` (or a potential exception) to an explicitly forbidden block in the flight plan. When the `deroute` stage is executed, depending on its condition, the change of block may be forbidden and the flight plan may be stuck in a deadlock.

We can also exploit further the bisimulation property between input, intermediate and output languages. This has already been done for the property that `on_enter` and `on_exit` are correctly executed, but we can extend that for other properties, e.g. if there is no warning generated, then there is no risk of deadlock. Using the bisimulation property, we can also improve code generation. We wanted initially to produce the same code as the OCaml generator for compatibility reasons but also because the semantics of FP was defined based on the code generated by the OCaml generator. We have proven that the semantics of FP is preserved by corresponding C code, itself similar to the C code generated by the original OCaml generator, as can be asserted by code reviewing and testing. We can therefore design a new generator producing a more optimised code with a completely different structure. If we are able to prove a bisimulation between this new generator and FP semantics, we thus have a strong confidence that the generated code will behave as the OCaml generator. Moreover, we have several ideas to improve confidence in the generator. First, we want to reduce as much as possible the number of unverified transformations, such as the OCaml preprocessing and postprocessing phases. For example, we can consider connecting directly our generator to CompCert to have a fully verified compiler that translates FPL to assembly code, provided the target architecture is supported by CompCert.

In this case, the postprocessor would no longer be used. Second, the generated code is actually separated from the common C code. However, for the verification proof, we use a Clight version of this common C code. As Clight is a strict subset of the C language, the Clight code used for the verification is thus different from the common C code that is used during the compilation. We could thus imagine generating only one program containing all the common C code in addition to the flight plan code such as `auto_nav`. Third, we might want to remove some axioms, especially those used to connect Clight operational semantics to `FP` denotational semantics, in particular as regards the management of external calls. We could imagine redefining `FP` semantics as an operational semantics that uses the same mechanism as Clight for external calls. Lastly, we may want to add a C parser for arbitrary C code. This could be used to ensure that the C code is valid and restrict users to some specific constructions.

Finally, we want to modularize the generator by separating the navigation modes from the flight plan. The idea is to offer users a generic interface to specify their own navigation modes. If they prove that the semantics is preserved for their navigation modes, then they have a custom C code generator suiting their needs. It will then be possible to support easily new autopilots or to use the generator in another context that uses synchronous programs such as finite-state machine.

12.3 General conclusion and future work

Through this thesis, we have had the chance to review 2 different formal verification processes. First, we have used static analysis to verify existing code by ensuring the absence of runtime errors and some functional properties. Second, we have developed a new program and proven it, to replace an old unverified program. Both processes bring some strong guarantees to the verified program, but they have both the same limitations.

First, the guarantees offered are based on hypotheses. Indeed, with actual tools, it is impossible to state that when an autonomous drone with a complex mission is launched, it will always behave as expected. The big question for every formal verification process is to determine under which reasonable models and hypotheses verification activities should take place. This decision is crucial and is too easily overlooked, although it can dramatically question the relevance of the verification work carried out if the hypotheses are unrealistic or contradictory.

Then, both tools require a certain level of expertise in formal methods, even “automatic” tools such as Frama-C and SMT solvers. This level of expertise concerns both the specification and the tool usage. The specification must be rigorously worked out to formally capture the the exact property to verify but it must also be written using encodings and representations that are adapted to the tool. For example, there are several ways in Coq to define a specific function but depending on the need, some of them are more suitable than others. Also, a minimal level of expertise is required for any formal tool. For example, as presented in Part I, Frama-C used with solvers is not able to verify some goals. The question is then to sort out whether the problem comes from a wrong or incomplete specification that must be modified or from solvers choking on a too complicated property so that they

must be helped (e.g. with tactics) or the proof must be done manually with Coq. This is a complex question that can take time to answer for people with no expertise on tools such as Frama-C.

Finally, the use of formal methods and tools has a high cost in terms of maintainability. Indeed, both projects currently do not work on the latest version of the Frama-C or Coq tools. The verification of the mathematical library has been done at the beginning of the thesis and since then, several versions of Frama-C and their associated solvers have been released. There are now goals that can no longer be verified. We do not know the reason why it is so, and it will certainly take a lot of time to fix the unproven goals. The verified generator has been updated to Coq 8.16 but the latest version currently available (Coq 8.17) is not able to complete the proof. After investigation, it seems that the new version can automatically prove some goals and some parameters for predicates which no longer need to be explicitly specified. The proof could thus be fixed, but it will take time. Proof maintenance is an essential issue for the adoption of formal methods in industry. Even if this problem is already studied [Tal21] with proof repair solutions, this is a point to consider when selecting a formal verification tool.

In future works, it would be interesting to verify other critical components or apply new formal verification processes. For instance, we could have verified Paparazzi autopilot generator: Paparazzi proposes a C code generator for autopilot (only available for rovers at the moment). Users can define their autopilot behaviour using state machines described in an XML file. Similarly to the verification of the flight plan generator, we could also prove that the generation of the C code is correct. In this case, we could directly define an operational semantics as Clight does and use separation logic. It might also be interesting to verify other parts of critical C code from the control system or the real-time operating system using model checking with tools such as CBMC [KST23].

Appendixes

Appendix A

Flight Plan C code

A.1 C code generated Example

This section presents an example of the C code generated for a simple flight plan. Section A.1.1 presents the XML example. Section A.1.2 presents the corresponding FP Gallina structure. Finally, Section A.1.3 shows the C code generated.

A.1.1 XML structure of the flight plan

```
<!DOCTYPE flight_plan SYSTEM "../flight_plan.dtd">

<flight_plan alt="260" ground_alt="185" lat0="43.565624" lon0="1.475071"
             max_dist_from_home="1500" security_height="25" qfu="1.0"
             name="Thesis">
  <waypoints>
    <waypoint name="HOME" x="0.0" y="0.0"/>
  </waypoints>

  <exceptions>
    <exception cond="Battery() @LT 20" deroute="Wait GPS"
               exec="LowBatteryMode()"/>
  </exceptions>

  <blocks>
    <block name="Wait GPS">
        <set var="time" value="GetTime()"/>
        <while cond="!GPSFixValid()">
          <call fun="InitSensors()" break="FALSE" loop="FALSE"/>
        </while>
    </block>
  </blocks>
</flight_plan>
```

A.1.2 Corresponding FP structure

```
{|
  excpts:= [
    {|
      cond:= "Battery() < 20",
      id:= 2,
      exec:= "LowBatteryMode()"
    |}
  ],
  fb_drtes:= [],
  blocks:= [
      {|
        name:= "INIT_BLOCK",
        id:= 0,
        excpts:= [],
        blocks:= [],
        pre_call:= None,
        post_call:= None
      |}
      {|
        name:=  "WaitGPS",
        id:= 0,
        excpts:= [],
        stages:= [
          SET "time" "GetTime()";
          WHILE "!GPSFixValid()" [
            CALL "func1()" None false false;
          ]
        ]
        pre_call:= Some "pre_call()",
        post_call:= Some "post_call()"
      |},
      {|
        name:= "HOME", id:= 15, excpts:= [],
        stages:= [NAV HOME None false],
        pre_call:= None,
        post_call:= None
      |}
    ]
|}
```

A.1.3 C code generated

A.1.3.1 Header Generater by the pre-processor

```
/* This file has been generated by the VFPG from tests/regression-tests/example-thesis.xml */
/* Version v6.0_unstable-none--dirty */
/* Please DO NOT EDIT */

#ifndef FLIGHT_PLAN_H
#define FLIGHT_PLAN_H

#include "std.h"
#include "generated/modules.h"
#include "modules/core/abi.h"
#include "autopilot.h"


#define FLIGHT_PLAN_NAME "Small1"
#define NAV_DEFAULT_ALT 260 /* nominal altitude of the flight plan */
#define NAV_UTM_EAST0 360285
#define NAV_UTM_NORTH0 4813595
#define NAV_UTM_ZONE0 31
#define NAV_LAT0 434622300 /* 1e7deg */
#define NAV_LON0 12728900 /* 1e7deg */
#define NAV_ALT0 185000 /* mm above msl */
#define NAV_MSL0 51850 /* mm, EGM96 geoid-height (msl) over ellipsoid */
#define QFU 1.0
#define WP_dummy 0
#define WP_HOME 1
#define WAYPOINTS_UTM { \
  {0.0, 0.0, 260},\
  {0.0, 0.0, 260},\
};
#define WAYPOINTS_ENU { \
  {0.00, -0.00, 75.00}, /* ENU in meters  */ \
  {0.00, -0.00, 75.00}, /* ENU in meters  */ \
};
#define WAYPOINTS_LLA { \
  /* 1e7deg, 1e7deg, mm (above NAV_MSL0, local msl=51.85m) */
  {.lat=434622299, .lon=12728900, .alt=260000}, \
  /* 1e7deg, 1e7deg, mm (above NAV_MSL0, local msl=51.85m) */
  {.lat=434622299, .lon=12728900, .alt=260000}, \
};
#define WAYPOINTS_LLA_WGS84 { \
  /* 1e7deg, 1e7deg, mm (above WGS84 ref ellipsoid) */
  {.lat=434622299, .lon=12728900, .alt=311850}, \
  /* 1e7deg, 1e7deg, mm (above WGS84 ref ellipsoid) */
  {.lat=434622299, .lon=12728900, .alt=311850}, \
};
#define WAYPOINTS_GLOBAL { \
  FALSE, \
  FALSE, \
};
```

```
#define NB_WAYPOINT 2
#define FP_BLOCKS { \
  "INIT_BLOCK" , \
  "Wait_GPS" , \
  "HOME" , \
}
#define NB_BLOCKS 3
#define GROUND_ALT 185.
#define GROUND_ALT_CM 18500
#define SECURITY_HEIGHT 25.
#define SECURITY_ALT 210.
#define HOME_MODE_HEIGHT 25.
#define MAX_DIST_FROM_HOME 1500.


#ifndef FBW


#endif

#ifdef NAV_C


static inline void auto_nav_init(void) {
}

// ----------------------------------------------------------------------
// C CODE GENERATED BY THE GENERATOR, PRESENTED IN THE FOLLOWING SECTION
// ----------------------------------------------------------------------

#endif // NAV_C

#endif // FLIGHT_PLAN_H
```

A.1.3.2 C code generated by the new generator

```
unsigned char const nb_blocks = 3;

void on_enter_block(unsigned char block)
{
  switch (block) {
    default:
      break;
  }
}

void on_exit_block(unsigned char block)
{
  switch (block) {
    default:
      break;
  }
}

_Bool forbidden_deroute(unsigned char from, unsigned char to)
{
  switch (from) {
    default:
      break;
  }
  return (_Bool) 0;
}

static inline void auto_nav(void)
{
  unsigned char tmp_nav_block;
  unsigned char tmp_nav_stage;
  _Bool tmp_cond;
  float tmp_RadOfDeg;
  float tmp_AltitudeMode;
  _Bool tmp_NavCond;
  tmp_nav_block = get_nav_block();
  if (tmp_nav_block != 1) {
    tmp_cond = (_Bool) (Battery()<20);
  } else {
    tmp_cond = (_Bool) 0;
  }
  if (tmp_cond) {
    (LowBatteryMode());
    nav_goto_block(1);
    return;
  }
  tmp_nav_block = get_nav_block();
  switch (tmp_nav_block) {
    case 0:
      tmp_nav_stage = get_nav_stage();
      switch (tmp_nav_stage) {
        default:
```

```
          set_nav_stage(0);
          NextBlock();
          break;
      }
      break;
    case 1:
      (pre_call_1());
      tmp_nav_stage = get_nav_stage();
      switch (tmp_nav_stage) {
        case 0:
          set_nav_stage(0);
          time = GetTime();
          NextStage();
        case 1:
          while_1_1:
          set_nav_stage(1);
          if (!(!(GPSFixValid()))) {
            goto endwhile_1_3;
          } else {
            NextStage();
            break;
          }
        case 2:
          set_nav_stage(2);
          (InitSensors());
          NextStage();
        case 3:
          set_nav_stage(3);
          goto while_1_1;
          endwhile_1_3:
          set_nav_stage(4);
        default:
          set_nav_stage(4);
          NextBlock();
          break;
      }
      (post_call_1());
      break;
    default:
      tmp_nav_stage = get_nav_stage();
      switch (tmp_nav_stage) {
        case 0:
          set_nav_stage(0);
          nav_home();
          break;
        default:
          set_nav_stage(1);
          NextBlock();
          break;
      }
      break;
  }
}
```

A.2 Common flight plan functions

This section shows the common C code that is a set of global variables storing the states of the flight plan and functions used by the generated code. This common C code is regrouped in a C header file and a C source file shown respectively in Section A.2.1 and Section A.2.2.

A.2.1 Header file for the common functions

```
/*
* Copyright (C) 2007-2011  The Paparazzi Team
*
* This file is part of paparazzi.
*
* paparazzi is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2, or (at your option)
* any later version.
*
* paparazzi is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with paparazzi; see the file COPYING.  If not, write to
* the Free Software Foundation, 59 Temple Place - Suite 330,
* Boston, MA 02111-1307, USA.
*/

/**
* @file subsystems/navigation/common_flight_plan.h
* Common flight_plan functions shared between fixedwing and rotorcraft.
*/

#ifndef COMMON_FLIGHT_PLAN_H
#define COMMON_FLIGHT_PLAN_H

#include <stdint.h>

/** In s */
extern uint16_t stage_time, block_time;

/* The modification of the following variables do not affect
* the behavior of the flight plan
*/
extern uint8_t nav_stage, nav_block;
extern uint8_t last_block, last_stage;

/* function to initiallize the vcisl */
void init_function(void);
```

```
/** needs to be implemented by fixedwing and rotorcraft seperately */
void nav_init_stage(void);

void nav_init_block(void);
void nav_goto_block(uint8_t block_id);

/* Getter */
uint8_t get_nav_block();
uint8_t get_nav_stage();
uint8_t get_last_block();
uint8_t get_last_stage();

/* Setter */
void set_nav_block(uint8_t b);
void set_nav_stage(uint8_t s);

#define InitStage() nav_init_stage();

#define Block(x) case x: set_nav_block(x); // Modified
void NextBlock(); // Modified
#define GotoBlock(b) nav_goto_block(b)

#define Stage(s) case s: set_nav_stage(s); // Modified
void NextStage(); // Modified
#define NextStageAndBreak() { NextStage(); break; }
#define NextStageAndBreakFrom(wp) { last_wp = wp; NextStageAndBreak(); }

#define Label(x) label_ ## x:
#define Goto(x) { goto label_ ## x; }
void Return(uint8_t x); // Modified

#define And(x, y) ((x) && (y))
#define Or(x, y) ((x) || (y))
#define Min(x,y) (x < y ? x : y)
#define Max(x,y) (x > y ? x : y)
#define LessThan(_x, _y) ((_x) < (_y))
#define MoreThan(_x, _y) ((_x) > (_y))

/** Time in s since the entrance in the current block */
#define NavBlockTime() (block_time)

/** Functions generated by the flight plan */
void on_enter_block(unsigned char);
void on_exit_block(unsigned char);
_Bool forbidden_deroute(unsigned char, unsigned char);

// constant define in the flight_plan.h
extern const uint8_t nb_blocks;

#ifndef GEN
#include "flight_plan.h"
#endif
```

```
#endif /* COMMON_FLIGHT_PLAN_H */
```

A.2.2 Source file for the common functions

```
/*
 * Copyright (C) 2007-2009  ENAC, Pascal Brisset, Antoine Drouin
 *
 * This file is part of paparazzi.
 *
 * paparazzi is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * paparazzi is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with paparazzi; see the file COPYING.  If not, write to
 * the Free Software Foundation, 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/**
  * @file subsystems/navigation/common_flight_plan.c
  * Common flight_plan functions shared between fixedwing and rotorcraft.
  */

#include "common_flight_plan.h"


/** In s */
uint16_t stage_time, block_time;

uint8_t nav_stage, nav_block;

/** To save the current block/stage to enable return */
uint8_t last_block, last_stage;

static uint8_t private_nav_stage, private_nav_block;
static uint8_t private_last_block, private_last_stage;

void init_function(void) {
    private_nav_stage = 0;
    nav_stage = 0;
    private_nav_block = 0;
    nav_block = 0;
    private_last_stage = 0;
    last_stage = 0;
    private_last_block = 0;
```

```
    last_block = 0;
}

uint8_t get_nav_block() {
  return private_nav_block;
}
uint8_t get_nav_stage() {
  return private_nav_stage;
}
uint8_t get_last_block() {
  return private_last_block;
}
uint8_t get_last_stage() {
  return private_last_stage;
}

void set_nav_block(uint8_t b){
  if (b >= nb_blocks) {
    private_nav_block = nb_blocks - 1;
  }
  else {
    private_nav_block = b;
  }
  nav_block = private_nav_block;
}

void set_nav_stage(uint8_t s) {
  private_nav_stage = s;
  nav_stage = s;
}

void NextBlock() {
  if (private_nav_block < nb_blocks - 1) {
      nav_goto_block(private_nav_block + 1);
  } else {
      nav_goto_block(private_nav_block);
  }
}

void NextStage() {
  private_nav_stage++;
  nav_stage = private_nav_stage;
  InitStage();
}

void Return(uint8_t x) {
  on_exit_block(private_nav_block);
  private_nav_block = private_last_block;
  nav_block = private_nav_block;
  if (x == 1) {
    private_nav_stage = 0;
  } else {
    private_nav_stage = private_last_stage;
```

```
  }
  nav_stage = private_nav_stage;
  block_time = 0;
  InitStage();
  on_enter_block(private_nav_block);
}

void nav_init_block(void)
{
  if (private_nav_block >= nb_blocks) {
    private_nav_block = nb_blocks - 1;
    nav_block = private_nav_block;
  }
  private_nav_stage = 0;
  nav_stage = private_nav_stage;
  block_time = 0;
  InitStage();
  on_enter_block(private_nav_block);
}

void nav_goto_block(uint8_t b)
{
  /* If the deroute is forbidden */
  if (forbidden_deroute(private_nav_block, b)) {
    /* The goto is not taken */
    return; // FIXME: Go to an error block?
  }
  if (b != private_nav_block) {
    /* To avoid a loop in a the current block */
    private_last_block = private_nav_block;
    last_block = private_last_block;
    private_last_stage = private_nav_stage;
    last_stage = private_last_stage;
  }
  on_exit_block(private_nav_block);
  private_nav_block = b;
  nav_block = private_nav_block;
  nav_init_block();
}
```

Bibliography

[AAM+16] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Z. Weaver.
"CertiCoq : A verified compiler for Coq".
2016.
Direct link.
[Cited on pages 29, 62, and 88.]

[AB07] Andrew W. Appel and Sandrine Blazy.
"Separation Logic for Small-step Cminor".
CoRR, abs/0707.4389, 2007.
Direct link, arXiv:0707.4389.
[Cited on page 126.]

[Abr05] Jean-Raymond Abrial.
"The B-book - assigning programs to meanings".
Cambridge University Press, 2005.
Direct link.
[Cited on page 90.]

[Abr10] Jean-Raymond Abrial.
"Modeling in Event-B - System and Software Engineering".
Cambridge University Press, 2010.
doi:10.1017/CBO9781139195881.
[Cited on page 90.]

[AGG10] Assalé Adjé, Stéphane Gaubert, and Éric Goubault.
"Coupling Policy Iteration with Semi-definite Relaxation to Compute Accurate Numerical Invariants in Static Analysis".
In *ESOP*, pages 23–42, 2010.
Direct link, doi:10.1007/978-3-642-11957-6_3.
[Cited on page 82.]

[AH76] Kenneth Appel and Wolfgang Haken.
"Every planar map is four colorable".
Bulletin of the American mathematical Society, 82(5):711–712, 1976.
Direct link.
[Cited on page 87.]

[AL88] Martin Abadi and Leslie Lamport.
"The Existence of Refinement Mappings".

In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 165–175, July 1988.
LICS 1988 Test of Time Award.
Direct link, doi:10.1109/LICS.1988.5115.
[Cited on page 120.]

[ANS19] ANSSI.
"The WooKey project", 2019.
Direct link.
[Cited on page 90.]

[APW02] Aybuke Aurum, Håkan Petersson, and Claes Wohlin.
"State-of-the-art: software inspections after 25 years".
Software Testing, Verification and Reliability, 12(3):133–154, 2002.
Direct link, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.243, doi:10.1002/stvr.243.
[Cited on page 91.]

[ATB+95] C Antoine, A Trotin, P Baudin, JM Collart, and J Raguideau.
"Caveat: a formal proof tool to validate programs".
1995.
Direct link.
[Cited on page 90.]

[BBP20] Timothy Bourke, Lélio Brun, and Marc Pouzet.
"Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset".
In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages*, volume 4 of *POPL'20*, page 29, New Orleans, LA, USA, January 2020. ACM.
Direct link, doi:10.1145/3371112.
[Cited on pages 29, 62, 92, 120, 142, and 239.]

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone.
"The synchronous languages 12 years later".
In *Proceedings of The IEEE*, pages 64–83, 2003.
Direct link, doi:10.1109/JPROC.2002.805826.
[Cited on page 78.]

[BCF+21] Patrick Baudin, Pascal Coq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto.
"ACSL: ANSI/ISO C specification language".
2021.
Direct link.
[Cited on pages 98 and 109.]

[Ber07] Gérard Berry.
"Scade: Synchronous design and validation of embedded control software".
In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, Dordrecht, 2007. Springer Netherlands.
[Cited on page 31.]

[BGGT23] Lélio Brun, Christophe Garion, Pierre-Loïc Garoche, and Xavier Thirioux.
"Equation-directed axiomatization of lustre semantics to enable optimized code validation".
22(5s), sep 2023.
doi:10.1145/3609393.
[Cited on page 119.]

[BJM13] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux.
"Implementing and reasoning about hash-consed data structures in Coq".
CoRR, abs/1311.2959, 2013.
Direct link, arXiv:1311.2959.
[Cited on page 206.]

[BL09] Sandrine Blazy and Xavier Leroy.
"Mechanized semantics for the Clight subset of the C language".
Journal of Automated Reasoning, 43(3):263–288, 2009.
Direct link, doi:10.1007/s10817-009-9148-3.
[Cited on pages 42, 126, and 128.]

[Bla20] Allan Blanchard.
"Introduction à la preuve de programmes C avec Frama-C et son greffon WP", 2020.
Direct link.
[Cited on page 97.]

[BLK19] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov.
"Towards Full Proof Automation in Frama-C Using Auto-active Verification".
In *NFM 2019 - 11th Annual NASA Formal Methods Symposium*, pages 88–105, Houston, TX, United States, May 2019. Springer.
Direct link, doi:10.1007/978-3-030-20652-9_6.
[Cited on pages 50 and 246.]

[BR19] Gérard Berry and Lionel Rieg.
"Towards Coq-verified Esterel Semantics and Compiling".
CoRR, abs/1909.12582, 2019.
Direct link, arXiv:1909.12582.
[Cited on pages 92 and 149.]

[Bru20] Lélio Brun.
"Mechanized semantics and verified compilation for a dataflow synchronous language with reset".
Theses, Université Paris sciences et lettres, July 2020.
Direct link.
[Cited on pages 29, 62, 92, 120, 126, and 128.]

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli.
"Satisfiability Modulo Theories".
In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
doi:10.3233/978-1-58603-929-5-825.
[Cited on pages 78 and 100.]

[CC77] Patrick Cousot and Radhia Cousot.
"Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints".
In *POPL*, pages 238–252, 1977.
Direct link, doi:10.1145/512950.512973.
[Cited on pages 81 and 99.]

[CC92] Patrick Cousot and Radhia Cousot.
"Abstract Interpretation Frameworks".
J. Log. Comput., 2(4):511–547, 1992.
Direct link, doi:10.1093/logcom/2.4.511.
[Cited on pages 81 and 91.]

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival.
"Combination of Abstractions in the Astrée Static Analyzer".
In M. Okada and I. Satoh, editors, *ASIAN*, pages 272–300. Springer, 2006.
Direct link, doi:10.1007/978-3-540-77505-8_23.
[Cited on page 82.]

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival.
"Why does Astrée scale up?".
Formal Methods in System Design, 35(3):229–264, 2009.
Direct link, doi:10.1007/s10703-009-0089-6.
[Cited on pages 82 and 109.]

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla.
"Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications".
ACM Trans. Program. Lang. Syst., 8(2):244263, April 1986.

Direct link, doi:10.1145/5397.5399.
[Cited on pages 77 and 78.]

[CGG+05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot.
"A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs".
In *CAV*, pages 462–475, 2005.
Direct link, doi:10.1007/11513988_46.
[Cited on page 82.]

[CH78] Patrick Cousot and Nicolas Halbwachs.
"Automatic Discovery of Linear Restraints Among Variables of a Program".
In *POPL*, pages 84–96, 1978.
Direct link, doi:10.1145/512760.512770.
[Cited on page 81.]

[CH85] Thierry Coquand and Gérard Huet.
"Constructions: A higher order proof system for mechanizing mathematics".
In Bruno Buchberger, editor, *EUROCAL '85*, pages 151–184, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
doi:10.1007/3-540-15983-5_13.
[Cited on page 88.]

[Com21] Comp4Drones.
"D3.3 Implementation of Integrated and Modular Architecture for Drones first version".
Technical report, Comp4Drones project, 2021.
Direct link.
[Cited on page 91.]

[Cou99] Patrick Cousot.
"The Calculational Design of a Generic Abstract Interpreter".
In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam, 1999.
Direct link.
[Cited on pages 81 and 91.]

[CP90] Thierry Coquand and Christine Paulin.
"Inductively defined types".
In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
doi:10.1007/3-540-52335-9_47.
[Cited on page 88.]

[CW96] Edmund M. Clarke and Jeannette. M. Wing.
"Formal Methods: State of the Art and Future Directions", 1996.
Direct link, doi:10.1145/242223.242257.
[Cited on page 65.]

[Dav03] Maulik A. Dave.
"Compiler Verification: a Bibliography".
ACM SIGSOFT Software Engineering Notes, 28, 2003.
Direct link, doi:10.1145/966221.966235.
[Cited on pages 28, 62, and 92.]

[DB94] Nicolaas Govert De Bruijn.
"A survey of the project AUTOMATH".
In *Studies in Logic and the Foundations of Mathematics*, volume 133, pages 141–161. Elsevier, 1994.
Direct link, doi:10.1016/S0049-237X(08)70203-9.
[Cited on page 87.]

[Dij72] Edsger W. Dijkstra.
"The Humble Programmer".
Commun. ACM, 15(10):859–866, 1972.
Direct link, doi:10.1145/355604.361591.
[Cited on pages 20 and 57.]

[Dij75] Edsger W. Dijkstra.
"Guarded Commands, Nondeterminacy and Formal Derivation of Programs".
Commun. ACM, 18(8):453–457, 1975.
Direct link, doi:10.1145/360933.360975.
[Cited on pages 86 and 99.]

[DM17] Claire Dross and Yannick Moy.
"Auto-active proof of red-black trees in SPARK".
In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods*, pages 68–83, Cham, 2017. Springer International Publishing.
Direct link, doi:10.1007/978-3-319-57288-8_5.
[Cited on pages 50 and 246.]

[dMKA+] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer.
"The Lean Theorem Prover (System Description).".
In *CADE*.
Direct link, doi:10.1007/978-3-319-21401-6_26.
[Cited on page 87.]

[dt21a] The Frama-C development team.
"EVA documentation", 2021.

Direct link.
[Cited on page 99.]

[dt21b] The Frama-C development team.
"Frama-C documentation", 2021.
Direct link.
[Cited on pages 99 and 103.]

[dt21c] The Frama-C development team.
"RTE documentation", 2021.
Direct link.
[Cited on page 99.]

[dt21d] The Frama-C development team.
"WP documentation", 2021.
Direct link.
[Cited on page 101.]

[ER08] Eric Eide and John Regehr.
"Volatiles Are Miscompiled, and What to Do about It".
In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, page 255264, New York, NY, USA, 2008. Association for Computing Machinery.
Direct link, doi:10.1145/1450058.1450093.
[Cited on page 119.]

[eSAB+16] Rovedy Aparecida Busquim e Silva, Nanci Naomi Arai, Luciana Akemi Burgareli, Jose Maria Parente de Oliveira, and Jorge Sousa Pinto.
"Formal Verification With Frama-C: A Case Study in the Space Software Domain".
IEEE Transactions on Reliability, 65(3):1163–1179, 2016.
Direct link, doi:10.1109/TR.2015.2508559.
[Cited on page 92.]

[Fag76] M. E. Fagan.
"Design and code inspections to reduce errors in program development".
IBM Systems Journal, 15(3):182–211, 1976.
doi:10.1147/sj.153.0182.
[Cited on page 91.]

[FG10] Paul Feautrier and Laure Gonnord.
"Accelerated Invariant Generation for C Programs with Aspic and C2fsm".
Electr. Notes Theor. Comput. Sci., 267(2):3–13, 2010.
Direct link, doi:10.1016/j.entcs.2010.09.014.
[Cited on page 82.]

[Fil11] Jean-Christophe Filliâtre.
"Deductive Program Verification".
Thèse d'habilitation, Université Paris-Sud, December 2011.
Direct link.
[Cited on pages 83, 91, and 100.]

[FMV14] Carlo Furia, Bertrand Meyer, and Sergey Velder.
"Loop invariants: Analysis, Classification and Examples".
ACM Computing Surveys, 46(3), 2014.
Direct link, doi:10.1145/2506375.
[Cited on page 85.]

[FNSG07] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo.
"An Open Framework for Foundational Proof-Carrying Code".
In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 6778, New York, NY, USA, 2007. Association for Computing Machinery.
Direct link, doi:10.1145/1190315.1190325.
[Cited on page 119.]

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry.
"A machine-checked proof of the odd order theorem".
In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
Direct link, doi:10.1007/978-3-642-39634-2_14.
[Cited on page 87.]

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot.
"The Zonotope Abstract Domain Taylor1+".
In *CAV*, pages 627–633, 2009.
Direct link, doi:10.1007/978-3-642-02658-4_47.
[Cited on page 81.]

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou.
"Static Analysis by Policy Iteration on Relational Domains".
In *ESOP*, pages 237–252, 2007.
Direct link, doi:10.1007/978-3-540-71316-6_17.
[Cited on page 82.]

[Gru70] Carl Grubin.
"Derivation of the quaternion scheme via the Euler axis and angle".
Journal of Spacecraft and Rockets, 7(10):1261–1263, 1970.

doi:10.2514/3.30149.
[Cited on pages 24 and 110.]

[GS10] Thomas Martin Gawlitza and Helmut Seidl.
"Computing Relaxed Abstract Semantics w.r.t. Quadratic Zones Precisely".
In *SAS*, pages 271–286, 2010.
Direct link, doi:10.1007/978-3-642-15769-1_17.
[Cited on page 82.]

[HAB+17] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dand, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, and et al.
"A Formal Proof of the Kepler Conjecture".
Forum of Mathematics, Pi, 5:e2, 2017.
Direct link, doi:10.1017/fmp.2017.1.
[Cited on page 87.]

[HBG14] Gautier Hattenberger, Murat Bronz, and Michel Gorraz.
"Using the Paparazzi UAV System for Scientific Research".
In *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, pages pp 247–252, Delft, Netherlands, August 2014.
Direct link, doi:10.4233/uuid:b38fbdb7-e6bd-440d-93be-f7dd1457be60.
[Cited on pages 20, 57, and 59.]

[HBL+22a] Gautier Hattenberger, Fabien Bonneval, Matheus Ladeira, Emmanuel Grolleau, and Yassine Ouhammou.
"Micro-drone autopilot architecture for efficient static scheduling".
In G. de Croon and C. De Wagter, editors, *13th International Micro Air Vehicle Conference*, pages 175–182, Delft, the Netherlands, Sep 2022.
Paper no. IMAV2022-21.
Direct link.
[Cited on pages 45 and 230.]

[HBL+22b] Gautier Hattenberger, Fabien Bonneval, Matheus Ladeira, Emmanuel Grolleau, and Yassine Ouhammou.
"Micro-drone autopilot architecture for efficient static scheduling".
In *13th International Micro Air Vehicle Conference*, pages 175–182, Delft, Netherlands, September 2022.
Direct link.
[Cited on page 91.]

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud.
"The synchronous dataflow programming language Lustre".
Proceedings of the IEEE, 79(9):1305–1320, September 1991.
Direct link, doi:10.1109/5.97300.
[Cited on pages 31, 78, and 149.]

[HE16] Gernot Heiser and Kevin Elphinstone.
"L4 Microkernels: The Lessons from 20 Years of Research and Deployment".
ACM Transactions on Computer Systems, 34(1):1:1–1:29, apr 2016.
Direct link, doi:10.1145/2893177.
[Cited on page 91.]

[Hig96] Nicholas J. Higham.
"Accuracy and Stability of Numerical Algorithms".
Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.
Direct link.
[Cited on page 67.]

[Hoa69] C. A. R. Hoare.
"An Axiomatic Basis for Computer Programming".
Commun. ACM, 12(10):576–580, 1969.
doi:10.1145/363235.363259.
[Cited on pages 83 and 84.]

[HP00] Klaus Havelund and Thomas Pressburger.
"Model checking java programs using java pathfinder".
International Journal on Software Tools for Technology Transfer, 2:366–381, 2000.
doi:10.1007/s100090050043.
[Cited on page 78.]

[Jae10] Éric Jaeger.
"Study of the Benefits of Using Deductive Formal Methods for Secure Developments. (Etude de l'apport des méthodes formelles déductives pour les développements de sécurité)".
PhD thesis, Pierre and Marie Curie University, Paris, France, 2010.
Direct link.
[Cited on pages 20 and 57.]

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy.
"Validating LR(1) Parsers".
In *ESOP 2012 - Programming Languages and Systems - 21st European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416, Tallinn, Estonia, March 2012. Springer.
Direct link, doi:10.1007/978-3-642-28869-2_20.
[Cited on page 139.]

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski.
"Frama-C: A software analysis perspective".
Formal aspects of computing, 27:573–609, 2015.

doi:10.1007/s00165-014-0326-7.
[Cited on pages 22, 91, and 97.]

[Klu76] Allan R. Klumpp.
"Singularity-free extraction of a quaternion from a direction-cosine matrix".
Journal of Spacecraft and Rockets, 13(12):754–755, 1976.
doi:10.2514/3.27947.
[Cited on pages 24 and 110.]

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens.
"CakeML: A Verified Implementation of ML".
In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179191, New York, NY, USA, 2014. Association for Computing Machinery.
Direct link, doi:10.1145/2535838.2535841.
[Cited on page 92.]

[KN06] Gerwin Klein and Tobias Nipkow.
"A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler".
ACM Trans. Program. Lang. Syst., 28(4):619695, jul 2006.
Direct link, doi:10.1145/1146809.1146811.
[Cited on page 92.]

[KOG+22] Soulimane Kamni, Yassine Ouhammou, Emmanuel Grolleau, Antoine Bertout, and Gautier Hattenberger.
"A Reverse Design Framework for Modifiable-off-the-Shelf Embedded Systems: Application to Open-Source Autopilots".
In *11th International Conference on Model and Data Engineering (MEDI 2022)*, volume 13761 of *Lecture Notes in Computer Science*, pages 133–146, Cairo, Egypt, November 2022. Springer Nature Switzerland.
Direct link, doi:10.1007/978-3-031-21595-7_10.
[Cited on page 91.]

[KST23] Daniel Kroening, Peter Schrammel, and Michael Tautschnig.
"CBMC: The C Bounded Model Checker", 2023.
Direct link, arXiv:2302.02384.
[Cited on pages 54, 78, and 250.]

[KT11] Temesghen Kahsai and Cesare Tinelli.
"PKind: A parallel k-induction based model checker".
In *PDMC*, pages 55–62, 2011.
Direct link, doi:10.4204/EPTCS.72.6.
[Cited on page 78.]

[KWN+10] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival.
"Astree: Proving the Absence of Runtime Errors".
In J.C. Laprie, editor, *Embedded real time software and systems - ERTS2 2010*, Toulouse, France, May 2010. AAAF, SEE, SIA.
Direct link.
[Cited on pages 20 and 57.]

[LB08] Xavier Leroy and Sandrine Blazy.
"Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations".
J. Autom. Reason., 41(1):131, jul 2008.
doi:10.1007/s10817-008-9099-0.
[Cited on page 127.]

[LBK+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand.
"CompCert - A Formally Verified Optimizing Compiler".
In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE.
Direct link.
[Cited on pages 29, 46, 62, 77, 92, 119, and 237.]

[Ler09a] Xavier Leroy.
"A formally verified compiler back-end".
CoRR, abs/0902.2137, 2009.
Direct link, arXiv:0902.2137.
[Cited on pages 120, 121, 126, 128, 230, and 240.]

[Ler09b] Xavier Leroy.
"Formal Verification of a Realistic Compiler".
Commun. ACM, 52(7):107115, jul 2009.
Direct link, doi:10.1145/1538788.1538814.
[Cited on pages 91 and 120.]

[Let04] Pierre Letouzey.
"Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq".
Thèse de doctorat, Université Paris-Sud, July 2004.
Direct link.
[Cited on pages 29, 62, and 88.]

[LGB+22] Matheus Ladeira, Emmanuel Grolleau, Fabien Bonneval, Gautier Hattenberger, Yassine Ouhammou, and Yuri Hérouard.
"Scheduling Offset-Free Systems Under FIFO Priority Protocol".
In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume

231 of *Dagstuhl Artifacts Series*, Modena, Italy, July 2022.
Direct link, doi:10.4230/DARTS.8.1.4.
[Cited on page 91.]

[LLF+96] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin OHalloran.
"Ariane 5 flight 501 failure report by the inquiry board", 1996.
Direct link.
[Cited on pages 20 and 57.]

[LPP05] D. Leinenbach, W. Paul, and E. Petrova.
"Towards the formal verification of a C0 compiler: code generation and implementation correctness".
In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 2–11, 2005.
Direct link, doi:10.1109/SEFM.2005.51.
[Cited on page 92.]

[MCP67] John Mac Carthy and James Painter.
"Correctness of a compiler for arithmetic expressions".
In *Proceedings of Symposia in Applied Mathematics*, Mathematical aspects of Computer Science 1, 1967.
Direct link.
[Cited on pages 28, 62, and 92.]

[Men] "Menhir".
http://gallium.inria.fr/~fpottier/menhir/.
Accessed: 2023-07-05.
[Cited on page 139.]

[Min01] Antoine Miné.
"The Octagon Abstract Domain".
In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
Direct link, doi:10.1007/s10990-006-8609-1.
[Cited on page 81.]

[Mis14] Jayadev Misra.
"Knaster-Tarski Theorem", 2014.
Direct link.
[Cited on page 81.]

[Mon08] David Monniaux.
"The pitfalls of verifying floating-point computations".
ACM Trans. Program. Lang. Syst., 30(3), 2008.

Direct link, doi:10.1145/1353445.1353446.
[Cited on page 67.]

[Nec97] George C. Necula.
"Proof-Carrying Code".
In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106119, New York, NY, USA, 1997. Association for Computing Machinery.
Direct link, doi:10.1145/263699.263712.
[Cited on page 119.]

[Nec00] George C. Necula.
"Translation Validation for an Optimizing Compiler".
In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 8394, New York, NY, USA, 2000. Association for Computing Machinery.
Direct link, doi:10.1145/349299.349314.
[Cited on page 119.]

[NN07] Hanne Riis Nielson and Flemming Nielson.
"Semantics with Applications: An Appetizer".
Undergraduate Topics in Computer Science. Springer, 2007.
doi:10.1007/978-1-84628-692-6.
[Cited on pages 35, 71, 86, and 151.]

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff.
"How Amazon Web Services Uses Formal Methods".
58(4):66–73, 2015.
Direct link.
[Cited on page 90.]

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang.
"Local Reasoning about Programs that Alter Data Structures".
In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2001.
Direct link, doi:10.1007/3-540-44802-0_1.
[Cited on page 86.]

[Pap21] Paparazzi UAV Team.
"Paparazzi – the free autopilot", 2021.
Direct link.
[Cited on pages 20 and 57.]

[PdAC+23] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Ctlin Hricu, Vilhelm Sjöberg, and Brent

Yorgey.
"Logical Foundations", volume 1 of *Software Foundations*.
Electronic textbook, 2023.
Direct link.
[Cited on page 88.]

[PGH+21] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux.
"Verifying the Mathematical Library of an UAV Autopilot with Frama-C".
In *26th International Conference on Formal Methods for Industrial Critical Systems - FMICS 2021*, Paris, France, August 2021.
Direct link, doi:10.1007/978-3-030-85248-1_10.
[Cited on pages 61 and 101.]

[PGH+22] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux.
"A gentle introduction to C code verification using the Frama-C platform".
Research report, ISAE-SUPAERO ; ONERA – The French Aerospace Lab ; ENAC, March 2022.
Direct link.
[Cited on pages 103 and 105.]

[PGH+23] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux.
"A Verified UAV Flight Plan Generator".
In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 130–140, Melbourne, Australia, May 2023.
Direct link, doi:10.1109/FormaliSE58978.2023.00021.
[Cited on pages 29 and 62.]

[PKDR18] Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza.
"Towards formal verification of Contiki: Analysis of the AES–CCM* modules with Frama-C".
In *RED-IOT 2018-Workshop on Recent advances in secure management of data and resources in the IoT*, 2018.
Direct link.
[Cited on page 92.]

[PL10] Dillon Pariente and Emmanuel Ledinot.
"Formal verification of industrial C code using Frama-C: a case study".
Formal Verification of Object-Oriented Software, page 205, 2010.
Direct link.
[Cited on page 92.]

[Pla18] André Platzer.
"Logical Foundations of Cyber-Physical Systems".
Springer, 2018.
doi:10.1007/978-3-319-63588-0.
[Cited on page 87.]

[PM12] Christine Paulin-Mohring.
"Introduction to the coq proof-assistant for practical software verification", pages 45–95.
Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
Direct link, doi:10.1007/978-3-642-35746-6_3.
[Cited on page 88.]

[Pol21] Baptiste Pollien.
"Vérification d'une bibliothèque mathématique d'un autopilote avec Frama-C".
In *Approches formelles dans l'assistance au développement de logiciels - AFADL 2021*, pages 31–35, Virtual event, France, June 2021.
Direct link.
[Cited on page 61.]

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman.
"Translation Validation".
In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, page 151166, Berlin, Heidelberg, 1998. Springer-Verlag.
Direct link.
[Cited on page 119.]

[PTG+21] Baptiste Pollien, Xavier Thirioux, Christophe Garion, Gautier Hattenberger, and Pierre Roux.
"Formal Verification for Autopilot - Preliminary state of the art".
Research report, ISAE-SUPAERO ; ONERA – The French Aerospace Lab ; ENAC, 2021.
Direct link.
[Cited on pages 72, 78, 82, and 86.]

[Rou13] Pierre Roux.
"Analyse statique de systèmes de contrôle commande : synthèse d'invariants non linéaires".
PhD thesis, 2013.
Thèse de doctorat dirigée par Wiels, Virginie et Garoche, Pierre-Loïc Sureté de logiciel et calcul de haute performance Toulouse, ISAE 2013.
Direct link.
[Cited on page 80.]

[Rum10] Siegfried M. Rump.
"Verification methods: Rigorous results using floating-point arithmetic".
Acta Numerica, 19:287–449, May 2010.
Direct link, doi:10.1017/S096249291000005X.
[Cited on page 67.]

[SBB+99] Ph. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit.
"Vérification de logiciels : techniques et outils du model-checking".
Vuibert, April 1999.
Direct link.
[Cited on pages 70 and 78.]

[SBF+20] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter.
"Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq".
Proceedings of the ACM on Programming Languages, pages 1–28, January 2020.
Direct link, doi:10.1145/3371076.
[Cited on pages 29, 62, and 88.]

[She78] Stanley W. Shepperd.
"Quaternion from Rotation Matrix".
Journal of Guidance and Control, 1(3):223–224, 1978.
doi:10.2514/3.55767b.
[Cited on pages 25 and 112.]

[SLS16] Chengnian Sun, Vu Le, and Zhendong Su.
"Finding Compiler Bugs via Live Code Mutation".
In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 849863, New York, NY, USA, 2016. Association for Computing Machinery.
Direct link, doi:10.1145/2983990.2984038.
[Cited on page 119.]

[ST19] Soheil Sarabandi and Federico Thomas.
"Accurate computation of quaternions from rotation matrices".
In Jadran Lenarcic and Vincenzo Parenti-Castelli, editors, *Advances in Robot Kinematics 2018*, pages 39–46, Cham, 2019. Springer International Publishing.
Direct link, doi:10.1007/978-3-319-93188-3_5.
[Cited on pages 25 and 112.]

[Stu13] Aaron Stump.
"Programming Language Foundations".

Wiley, 2013.
Direct link.
[Cited on pages 72 and 79.]

[Tal21] Ringer Talia.
"Proof Repair".
PhD thesis, University of Washington, 2021.
Direct link.
[Cited on pages 54 and 250.]

[Tea20] The Why3 Development Team.
"Why3 documentation", 2020.
Direct link.
[Cited on pages 100 and 105.]

[The23] The Coq Development Team.
"The Coq Proof Assistant, version 8.8.0", June 2023.
Direct link, doi:10.5281/zenodo.8161141.
[Cited on pages 23, 101, and 107.]

[TMK+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish.
"A New Verified Compiler Backend for CakeML".
In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 6073, New York, NY, USA, 2016. Association for Computing Machinery.
Direct link, doi:10.1145/2951913.2951924.
[Cited on page 92.]

[Tod20] Vassil Todorov.
"Automotive embedded software design using formal methods".
Phd thesis, Université Paris-Saclay, December 2020.
Direct link.
[Cited on pages 24, 91, 92, and 109.]

[VBF+11] Eric Verhulst, Raymond Boute, José Faria, Bernhard Sputh, and Vitaliy Mezhuyev.
"Formal Development of a Network-Centric RTOS".
01 2011.
doi:10.1007/978-1-4419-9736-4.
[Cited on page 91.]

[Ver] Eric Verhulst.
"Main mission of Rosetta's Philae lander accomplished".
Direct link.
[Cited on page 91.]

[Win93] Glynn Winskel.
"The formal semantics of programming languages – an introduction".
MIT Press, 1993.
Direct link.
[Cited on pages 35 and 151.]

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald.
"Formal Methods: Practice and Experience".
41(4):1–40, 2009.
Direct link, doi:10.1145/1592434.1592436.
[Cited on page 90.]

[WPGT23] Valentin Wasquel, Baptiste Pollien, Christophe Garion, and Xavier Thirioux.
"Improving a Verified Flight Plan Generator: New Properties and Code Modifications".
Technical report, ISAE - Institut Supérieur de l'Aéronautique et de l'Espace, August 2023.
Direct link.
[Cited on pages 139, 163, 228, 235, and 243.]

[WSC+08] Jim Woodcock, Susan Stepney, David Cooper, John Clark, and Jeremy Jacob.
"The Certification of the Mondex Electronic Purse To ITSEC Level E6".
20:5–19, 2008.
doi:10.1007/s00165-007-0060-5.
[Cited on page 90.]

[ZPMGJG22] SHI Zheng-Pu, CUI Min, XIE Guo-Jun, and CHEN Gang.
"Coq Formalization of Propulsion Subsystem of Flight Control System for Multicopter".
Journal of Software, 33(6):2150–2171, 2022.
Direct link.
[Cited on page 91.]

Summary for the general public

Critical systems are systems whose failure could have catastrophic consequences, such as the destruction of expensive equipment or the death of people. These systems are found in the automotive, medical and drone autopilot systems. It is essential to ensure that these systems do not present a risk of failure. Formal methods are powerful verification techniques for obtaining strong guarantees on such systems, even if it is generally not possible to formally verify the entire system. This thesis deals with the formal verification of certain critical software components of a drone autopilot. It made it possible to review different methods of verification and proof of programs that can be applied to such software and use them on a case study, the Paparazzi autopilot developed at ENAC.

Keywords: Formal methods, Verification and validation, Cyberphysical systems, Proof of program, Mechanized proof, Static analysis.

Résumé pour le grand public

Les systèmes critiques sont des systèmes dont la défaillance peut avoir des conséquences catastrophiques, telles la destruction de matériel coûteux ou le décès de personnes. On retrouve ces systèmes aussi bien dans l'automobile, le médical ou bien dans les systèmes de pilotage automatique de drone. Il est essentiel de s'assurer que ces systèmes ne présentent pas de risque de défaillance. Les méthodes formelles sont des techniques de vérification puissantes permettant d'obtenir des garanties fortes sur de tels systèmes, même s'il n'est généralement pas possible de vérifier formellement l'ensemble du système. Cette thèse traite de la vérification formelle de certains composants logiciels critique d'un autopilote de drone. Elle a permis de passer en revue différentes méthodes de vérification et de preuve de programmes qui peuvent être appliquées à de tels logiciels et de les utiliser sur un cas d'étude, l'autopilote Paparazzi développé à l'ENAC.

Mots clés : Méthodes formelles, Vérification et validation, Systèmes cyber-physique, Preuve de programme, Preuve mécanisée, Analyse statique.

This work is supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CONCORDE N 2019 65 0090004707501)