Ecosystèmes COTS de développement et de vérification des logiciels critiques et temps réel

Version: v1.1.0
Référence: DLA-SF-0000000-194-QGP

Auteurs:
Julien Blond (julien.blond@ocamlpro.com)
Arthur Carcano (arthur.carcano@ocamlpro.com)
Pierre Villemot (pierre.villemot@ocamlpro.com)

Résumé

Ce rapport présente une étude des langages C, C++, Ada, SCADE, OCaml et Rust du point de vue de la sûreté. Il suit les clauses techniques [1] relatives au projet «COTS de qualité : logiciels critiques et temps réel» par et pour le CNES.

1. Introduction

1.1. Terminologie

Terme	Définition
COTS	*Commercial Off-The-Shelf* ou produit sur étagère
WCET	*Worst Case Execution Time* ou temps d'exécution du pire cas (*i.e.* maximal)

1.2. Organisation du document

Le document présente les langages C, C++, Ada, SCADE, OCaml et Rust d'un point de vue de la sûreté logicielle embarquée selon un plan organisé en trois axes:

1. une description du langage décrivant
   1. son paradigme;
   2. ses mécanismes de protection;
   3. ses compilateurs;
   4. son adhérence au système;
   5. ses gestionnaires de paquets;
   6. sa communauté.
2. l'outillage présent dans l'écosystème du langage :
   1. les débugueurs;
   2. les outils de tests;
   3. les outils de *parsing*;
   4. les capacités de méta-programmation;
   5. les possibilités de dérivation.
3. les aspects de sûreté logicielle :
   1. les outils d'analyse statique disponibles;
   2. les moyens de formalisation;
   3. le calcul statique du WCET;
   4. le calcul statique de la taille de pile maximale;
   5. les outils de qualité numérique;
   6. l'assurance générale de la qualité du code;
   7. l'utilisation dans le domaine critique.

1.3. Note méthodologique

La méthodologie utilisée pour l'étude des langages reprend les points abordés dans les clauses techniques [1]. De manière générale, l'étude est basée sur le contenu publique disponible sur internet, les brochures techniques ou commerciales et les connaissances propres des auteurs. Le volume d'information obtenu étant inégal suivant les outils, la complétude des informations fournies dans ce rapport n'est pas assurée. Toutefois, et sous réserve que les informations publiques soient à jour, elles sont *a priori* correctes.

Ce rapport étant lui même destiné à être *open source*, nous invitons le lecteur à participer à son amélioration continue en signalant toute erreur ou en contribuant via le dépôt https://github.com/OCamlPro/ppaqse-lang.

Pour des questions de lisibilité du document, certains points méthodologiques ou explications sont renvoyés en annexe.

2. C

2.1. Description

Le langage C est un langage de programmation créé en 1972 par Dennis Ritchie pour le développement du système d'exploitation Unix. Il est un des langages les plus utilisés dans le monde de l'informatique et est souvent utilisé pour écrire des systèmes d'exploitation, des compilateurs, des interpréteurs et des logiciels embarqués.

Le langage a été normalisé par l'ANSI[1] en 1989 [2] puis par l'ISO[2] en 1990 [3]. Le standard a été revu à plusieurs reprises jusqu'à la dernière version en 2018 [4].

Il existe par ailleurs plusieurs référentiels de programmation pour garantir une certaine qualité de code. Le plus connu est le référentiel MISRA-C [5] qui est utilisé dans l'industrie pour aider à fiabiliser les logiciels embarqués.

2.1.1. Paradigme

Le langage C est un langage de programmation essentiellement impératif. Il est possible, dans une certaine mesure, de programmer dans un style fonctionnel comme avec le code suivant qui calcule récursivement la longueur d'une liste chaînée.

```c
int length(struct list *l)
{
  return (nullptr == l) ? 0 : 1 + length(l->next);
}
```

mais cela reste limité par rapport aux langages explicitement fonctionnels. Le style idiomatique est plutôt procédural à la manière du Listing 1.

```c
int add_value(struct state *s, int v)
{
  int code = KO;
  if (nullptr != s) {
    s->acc += v;
    code = OK;
  }
  return code;
}
```

Listing 1: Exemple de code C idiomatique

Dans cet exemple, on déclare une fonction `add_value` qui prend en paramètre un pointeur vers une structure `state` et un entier `v`. Si le pointeur n'est pas `nullptr`, on ajoute la valeur `v` à l'accumulateur `acc` de la structure et on retourne `OK`. Sinon, on retourne `KO`.

Le code est clairement impératif avec la séquence d'instructions, le `if` et les effets de bord `s->acc += v` et `code = OK`.

[1] *American National Standard Insitute*, le service de standardisation des États-Unis.
[2] *International Organization for Standardization*, l'organisation internationale de standardisation.

Sauf pour les fonctions simples et totales comme avec `length`, le code de retour est généralement utilisé comme un code indiquant si l'appel a produit un résultat nominal ou s'il y a eu une erreur. En pratique, cela donne des appels en séquence avec un style dit *défensif* comme dans le Listing 2.

```c
int add_values(struct state *s) {
  int code = KO;
  if (KO == add_value(s, 42)) {
    /* traitement d'erreur 1 ...*/
  }
  if (KO == add_value(s, 24)) {
    /* traitement d'erreur 2 ...*/
  }
  code = OK;
  return code;
}
```

Listing 2: Exemple de code C défensif

2.1.2. Mécanismes intrinsèques de protection

Le C est dispose de très peu de mécanismes de protection. Il existe un système de type mais qui est très rudimentaire et ne permet pas de garantir la sécurité mémoire. Les pointeurs peuvent être manipulés de manière très libre et il est possible de déréférencer un pointeur n'importe où dans la mémoire.

Les mécanismes de protection disponibles pour le C viennent essentiellement des compilateurs qui ajoutent, ou pas, des analyses complémentaires. La plupart du temps, il est recommandé d'activer tous les avertissements du compilateur et de les traiter comme des erreurs pour assurer un maximum de vérifications (`-Wall` sur GCC par exemple).

2.1.3. Compilateurs

Pour des raisons historiques, il existe beaucoup de compilateurs pour le C car il est né a une époque où l'*open source* n'existait pas vraiment et les seules organisations capables de produire des compilateurs étaient des entreprises ou des grands laboratoires. Ces entreprises développaient souvent leur propre compilateur adapté à leur architecture ou au système qu'ils proposaient par ailleurs.

Nous ne citerons donc que les principaux compilateurs modernes qui sont les plus utilisés ou qui ont une caractéristique particulière qui peut les distinguer dans le contexte de l'embarqué critique.

Compilateur	Architectures	Licence
AOCC	AMD x86 (32 et 64 bits)	Propriétaire, Gratuit
Clang	AArch64, ARMv7, IA-32, x86-64, ppc64le	Apache 2.0
CompCert	x86, x86_64, ARM, PowerPC, SPARC	INRIA non commercial, Gratuit pour un usage non commercial
GCC	IA-32, x86_64, ARM, PowerPC, SPARC, ...[3]	GPLv3+
ICX (Intel C/C++ Compiler)	IA32, x86-64	Propriétaire, Gratuit
MSVC	IA-32, x86_64, ARM	Propriétaire
sdcc	microprocesseurs[4]	GPLv2

Notons que le compilateur AOCC est basé sur Clang/LLVM mais y ajoute des optimisations spécifiques aux processeurs AMD. Clang est la partie frontale de LLVM et les deux forment un compilateur modulaire initié par Apple pour remplacer GCC dans les années 2000. Aujourd'hui, ces deux compilateurs offrent des performances équivalentes.

GCC est le compilateur de référence pour le langage C depuis les années 1990. Il est très complet et supporte un grand nombre d'architectures dont seulement une petite partie est indiquée dans le tableau.

ICX est le compilateur d'Intel spécifique à aux processeurs et FPGA de la marque. Depuis 2021, il utilise le *backend* LLVM. Enfin, MSVC est le compilateur de Microsoft pour les systèmes Windows.

Le compilateur sdcc est un compilateur dédié aux microprocesseurs et supporte tous les standards C (même le C23 encore en brouillon).

Le compilateur CompCert est un peu à part car il s'agit d'un compilateur écrit en Coq et OCaml par l'INRIA et qui est formellement prouvé comme étant correct par rapport à la sémantique du langage C. Il offre des performances équivalentes à GCC avec un niveau d'optimisation léger (-O1). L'utilisation commerciale est pourvue par la société AbsInt. L'absence d'optimisations aggressives que l'on peut trouver dans GCC ou Clang est due au fait qu'il est difficile de démontrer que ces optimisations sont correctes d'un point de vue sémantique. Le projet se concentre sur les optimisations vérifiées afin de produire un code conforme à la spécification du langage C propre à une utilisation dans le domaine critique.

2.1.4. Interfaçage

Le C étant devenu *de facto* un langage de référence utilisé sur beaucoup de systèmes et avec un nombre important de bibliothèques, la plupart des langages modernes proposent une FFI[5] pour le C. Cela permet d'exporter du C dans ces langages mais également au C de les utiliser.

[3] La liste exhaustive est disponible à l'adresse https://gcc.gnu.org/install/specific.html.

[4] Intel MCS51, Maxims, Freescale, ... La liste est disponible sur le site du compilateur : http://sdcc.sourceforge.net/

[5] *Foreign Function Interface*, ou interface de programmation externe.

2.1.5. Adhérence au système

Bien qu'un programme C utilise généralement une interface POSIX via la `libc` dont il existe plusieurs implémentations, il est tout à fait possible de faire sans et d'utiliser le C sur un système nu. Naturellement, cela nécessite d'écrire toutes les interfaces systèmes nécessaires mais c'est justement un langage fait pour ça. Il est donc particulièrement adapté pour des systèmes embarqués.

2.1.6. Gestionnaire de paquets

Historiquement, le C repose plutôt sur les gestionnaires de paquets fournis par le système d'exploitation (`apt` pour Debian par exemple). Cependant, le besoin de créer des environnements isolés pour les projets a poussé à la création de gestionnaires de paquets spécifiques dans beaucoup de langages (`cabal` pour Haskell, `pip` pour Python, ...). Des gestionnaires de paquets pour le C ont donc été également créés pour répondre à ce besoin.

La plupart des gestionnaires de paquets proposent peu ou prou les mêmes fonctionnalités que l'on peut retrouver dans la Table 2.

Gestionnaire	Clib	Conan	vcpkg
Plateformes	Linux	Toutes	Linux, Windows, MacOS
Format	JSON	Python	JSON
Résolution des dépendances	✗	✓	✓
Cache binaire	✗	✓	✓
Nombre de paquets	350	1750	2500

Table 2: Gestionnaires de paquets C

Clib propose une gestion de paquet très rudimentaire qui consiste à simplement intégrer les sources d'un paquet dans les sources du projet. Cela peut être pratique pour des projets très petits ou pour des projets qui ne veulent pas dépendre d'un gestionnaire de paquet tiers.

Conan et vcpkg sont des gestionnaires de paquet plus complets qui proposent les fonctionnalités d'un gestionnaire de paquet moderne. Ce sont tous deux des outils matures et maintenus. Conan est plus simple d'utilisation et plus versatile car les descriptions de paquets (les *recipes*) sont écrites en Python. vcpkg est plus classique avec une description JSON mais plus de paquets disponibles.

Notons qu'il est aussi possible d'utiliser des gestionnaires de paquets agnostiques comme Nix. Ce denier dispose de toute les fonctionnalités qu'on peut attendre d'un gestionnaire de paquet moderne mais il est plus complexe à utiliser et nécessite une certaine connaissance de l'outil pour être efficace.

2.1.7. Communauté

L'histoire du langage C et son rôle central dans l'évolution de l'informatique font qu'il existe une très grande communauté autour du langage. Cette communauté est à la fois privée et publique : privée car de nombreuses sociétés ont et continuent de développer des logiciels ou des produits utilisant le C et publique à travers toute la communauté *open source* qui continue de contribuer aux projets

existants (notamment les projets appuyés par la *FSF*[6]) et d'en proposer de nouveaux puisque le C reste dans les 10 langages les plus utilisés sur GitHub[7].

2.2. Outillage

2.2.1. Débugueurs

Comme pour les compilateurs, il existe une multitude de débugueurs en fonction des systèmes d'exploitation et des architectures. Pour simplifier la lecture de ce document, nous ne listons ici que les principaux débugueurs connus.

Debugueur	Architectures	License
Linaro DDT	x86-64, ARM, PowerPC, Intel Xeon Phi, CUDA	Propriétaire
GDB	x86, x86-64, ARM, PowerPC, SPARC, ...	GPLv3
LLDB	i386, x86-65, AArch64, ARM	Apache v2
TotalView	x86-64, ARM64, CUDA	Propriétaire
Undo	x86-64	Propriétaire
Valgrind	x86, x86-64, PowerPC, ARMv7	GPL
Visual Studio Debugger	x86, x86-64	Propriétaire
WinDBG	x86, x86-64	Gratuit
rr	x86, x86-64, ARM	GPLv2

Linaro DDT et TotalView sont plutôt dédiés aux systèmes de calculs intensifs qu'aux systèmes embarqués mais ils supportent les architectures CUDA qui peuvent être utilisés dans l'embarqués pour du traitement vidéo.

GDB est le déboggeur de référence pour le langage C car il va en général de pair avec l'utilisation de GCC. Il dispose des fonctionnalités classiques pour un déboggueur (breakpoints, pas à pas, ...). La version de base ne comporte qu'un outil en ligne de commande mais des interfaces graphiques existent pour le compléter, le plus connu étant probablement DDD.

LLDB est le déboggeur associé à Clang/LLVM comme GDB l'est à GCC. Etant plus jeune, il supporte moins d'architectures mais plus de systèmes d'exploitation (notamment MacOS et iOS).

Valgrind est un outil d'analyse dynamique qui permet de détecter des erreurs liées à l'utilisation de la mémoire. Il est particulièrement utile pour détecter les fuites mémoires sur les langages comme le C. Il fonctionne en compilant le *bytecode* à la volée en y ajoutant de l'instrumentation.

Undo est un débugueur récent compatible avec les commandes de GDB et proposant une interface graphique plus moderne que DDD. Il propose aussi une interface de navigation dans les historiques d'exécution en séquentiel ou parallèle en plus d'un débugueur mémoire. En revance, il n'est disponible que sous Linux et les architectures supportées ne sont pas clairement indiquées. Il est probable que l'outil fonctionne par compilation à la volée du *bytecode* et qu'il soit donc compatible avec toutes les architectures supportées par GCC.

[6] *Free Software Foundation*.
[7] https://www.blogdumoderateur.com/github-top-10-langages-utilises-developpeurs-2023/

Visual Studio Debugger est le débugueur associé à la suite de développement Visual Studio de Microsoft. Il est propriétaire et ne fonctionne que sous Windows mais offre un support avancé de tous les langages supportés par Visual Studio. Il ne faut pas confondre Visual Studio Debugger avec WinDBG qui est un débugueur plus bas niveau pour les plateformes Windows et qui est gratuit.

rr est un débugueur qui propose la même interface que GDB mais qui exécute le code dans une machine virtuelle et permet de remonter dans le temps et rejouer les exécutions de manière déterministe. Cela permet de débuguer plus facilement les erreurs aléatoires.

2.2.2. Tests

Nous avons comparés plusieurs outils de tests pour le langage C. Parmi ceux-ci, se dégagent Cantata, Parasoft, TPT et VectorCAST/C++ qui offrent un support de test étendu pour le C. Ils fournissent également :

- de la génération de test à des degrés divers ;
- une bonne gestion des tests à travers diverses configurations ;
- la génération de rapports nécessaires aux qualifications/certifications.

Criterion, LibCester, NovaProva et Opmock tiennent plus des usuels cadres logiciels de tests et offrent une génération de test plus limitée. Cependant, ils offrent du *mocking* ou un support pour des interfacages avec des outils tiers qui peuvent être utiles dans les cas d'intégration.

Outil	Tests	Generation	Gestion	*mocking*
Cantata	UIRC	+++	✓	
Criterion	U	+		
LibCester	U	+	✓	✓
NovaProva	U	+	✓	✓
Opmock	U	++		✓
Parasoft	UC	++	✓	
TPT	UINC	+++	✓	
VectorCAST/C++	UC	++	✓	✓

Table 4: Comparaison des outils de tests pour le langage C

La Table 4 utilise la nomenclature indiquée dans l'Annexe H.

2.2.3. Parsing

On peut distinguer les parseurs en fonction du type de langage considéré

- les langages réguliers
- les langages algébriques
- les grammaires booléennes déterministes
- les langages algébriques avec des grammaires conjonctives ou booléennes

Nom	Code	Plateforme	License
Flex	Mixte	Toutes	Libre, BSD
Quex	Mixte	Toutes	Libre, LGPL
Re2c	Mixte	Toutes	Libre, Domaine public
Ragel	Mixte	Toutes	Libre, LGPL, MIT

Table 5: Parsers de langages réguliers

Les parseurs de langages réguliers sont des outils utilisés pour parser des expressions régulières dans les langages de programmation ou pour servir de *lexer* en découpant un texte en *tokens* qui servent ensuite de termes à un parseur plus complexe.

Flex est l'outil de référence pour générer des *lexers* en C. Les autres lexers sont des alternatives plus modernes qui offrent des performances un peu meilleures ou des fonctionnalités supplémentaires comme la gestion de l'unicode.

Nom	Algorithme	Grammaire	Code	Plateforme	License
ANTLR	LL(k) adaptatif	EBNF	Mixte	Java	Libre, BSD
Byacc	LALR(1)	Yacc	Mixte	Toutes	Libre, Domaine public
Hyacc	LR(1), LALR(1), LR(0)	Yacc	Mixte	Toutes	Libre, GPL
Lemon	LALR(1)	Lemon	Mixte	Toutes	Libre, Domaine public
LLgen	LL(1)	LLgen	Mixte	POSIX	Libre, BSD
LLnextgen	LL(1)	LLnextgen	Mixte	Toutes	Libre, GPL
Yacc	LALR(1)	Yacc	Mixte	Toutes	Libre, CPL & CDDL
Tree-sitter	LR(1), GLR	Javascript, DSL, JSON	Séparé	Toutes	Libre, MIT
Styx	LALR(1)	Styx	Séparé	Toutes	Libre, LGPL
Coco/R	LL(1)	Wirth	Mixte	Toutes	Libre, GPL
SableCC	LALR(1)	SableCC	Séparé	Java	Libre, GPL
Bison	LALR(1), LR(1), IELR(1), GLR	Yacc	Mixte	Toutes	Libre, GPL
Unicc	LALR(1)	EBNF	Mixed	POSIX	Libre, BSD
Kmyacc	LALR(1)	Yacc	Mixte	Toutes	Libre, GPL
APG	Récursif descendant avec backtracking	ABNF	Séparé	Toutes	Libre, BSD
GOLD	LALR(1)	EBNF	Séparé	Toutes	Libre, zlib modifiée

Table 6: Parsers de langages algébriques

Nom	Algorithme	Grammaire	Code	Plateforme	License
DParser	GLR sans scanner	EBNF	Mixte	POSIX	Libre, BSD
YAEP	Earley	EBNF	Mixte	Toutes	Libre, LGPL
Bison	LALR(1), LR(1), IELR(1), GLR	Yacc	Mixte	Toutes	Libre, GPL
GDK	LALR(1), GLR	Yacc	Mixte	POSIX	Libre, MIT

Table 7: Parsers de grammaires sans contexte, conjonctives ou booléennes

Il existe beaucoup de générateurs de *parser* pour le langage C. Ils se différencient essentiellement par le type de langage reconnu : régulier, algébrique, avec ou sans contexte… L'analyse syntaxique fait l'objet de nombreuses recherches depuis les années 1960 et elle continue encore aujourd'hui de sorte qu'il existe de nombreux outils de *parsing* académiques qui vont mettre l'accent sur telle ou telle nouvelle approche ou optimisation.

Nous en avons listé un certain nombre pour information mais il est probable que qu'utiliser Flex/Bison ou ANTLR soit plus que suffisant dans la plupart des cas.

Notons que Tree-sitter est un parseur plutôt dédié aux éditeurs de textes mais cela peut être intéressant dans le cadre d'une définition de DSL ou de protocole pour l'embarqué afin de faciliter la création d'outils spécifiques.

Autrement, la différence entre les *parsers* de la liste réside essentiellement dans l'algorithme utilisé (LL, LR, LALR, …) et dans les fonctionnalités supplémentaires qu'ils offrent.

2.2.4. Meta programmation

Il n'y a pas, a proprement parler, de support pour la métaprogrammation en C. Dans la pratique cependant, le preprocesseur du C (CPP[8]) est souvent utilisé pour introduire une forme archaïque de métaprogrammation syntaxique.

CPP utilise un système de macros et d'expansions permettant de générer du code C par substitution de texte. Un exemple courant est l'utilisation de macros pour rendre des parties de code conditionnelles :

```c
#include <stdio.h>

#ifdef DEBUG
# define TRACE(...) printf(__VA_ARGS__)
#else
# define TRACE(...)
#endif

int main()
{
  TRACE("Hello, world!\n");
  return 0;
}
```

Dans cet exemple, la macro `TRACE` est définie en fonction de la macro `DEBUG`. Si `DEBUG` est définie, la macro `TRACE` est expansée en un appel à `printf` avec les arguments passés. Dans ce cas, la fonction `main` affichera `Hello, world!`. Si `DEBUG` n'est

[8] C PreProcessor

pas défini, la macro `TRACE` est expansée en un contenu vide et la fonction `main` n'affichera rien.

Le langage de macro permet de précalculer des expressions constantes comme avec l'exemple suivant qui permet de calculer la somme des entiers de 1 à N :

```c
#include <stdio.h>

#define SUM(N) (((N) * ((N) + 1)) / 2)

int main()
{
  printf("%d\n", SUM(10));
  return 0;
}
```

Comme elle sera expansée dans le code, le compilateur y vera une expression constante et la calculera à la compilation, ce qui donnera le résultat 55 immédiatement à l'exécution.

Bien que la récursion ne soit pas permise, il est possible d'en obtenir une version finie en exploitant de manière astucieuse les règles d'expansion des macros mais cela reste une pratique très avancée, peu lisible, peu maintenable et surtout généralement inutile puisque les fonction déclarées `inline` (et même parfois celles qui ne le sont pas) sont optimisées de manière similaire par le compilateur. Ainsi, le resultat constant précédent pourrait être obtenu plus simplement avec une fonction `inline` :

```c
#include <stdio.h>

inline int sum(int n)
{
  return (n * (n + 1)) / 2;
}

int main()
{
  printf("%d\n", sum(10));
  return 0;
}
```

2.2.5. Dérivation

Il n'y a pas réellement de support pour la dérivation en C. Comme pour la métaprogrammation, le préprocesseur C peut être utilisé pour dériver du code mais cela reste une pratique peu lisible.

Une technique bien connue est celle des X macros qui permet de générer du code à partir de la définition déclarative d'une relation. Par exemple, on peut mettre en relation un identifiant de couleur, un entier qui le représente et une chaîne de caractères qui la décrit:

```c
#define COULEURS \
  X(ROUGE, 0xFF0000, "rouge") \
  X(VERT, 0x00FF00, "vert") \
  X(BLEU, 0x0000FF, "bleu")
```

La macro `COULEURS` définit la relation en utilisant une macro `X` qui n'a pas encore de définition. C'est au moment où on utilise cette relation qu'on donne une définition à la macro `X`. On peut ainsi définir le type couleur :

```
1 #define X(label, value, str) label,
2 typedef enum {
3   COULEURS
4 } couleurs_t;
5 #undef X
```

ou les convertisseurs adéquats de manière à peu près automatisée :

```
1 char* couleur_to_string(couleurs_t couleur)
2 {
3   char* resultat = nullptr;
4 #define X(label, _, str) case label: { resultat = str; break; }
5   switch (couleur) {
6     COULEURS
7     default: { break; }
8   }
9 #undef X
10   return resultat;
11 }
12
13 int main()
14 {
15   printf("%s\n", couleur_to_string(ROUGE));
16   return 0;
17 }
```

Il est possible de faire des dérivations plus complexes en jouant sur la sémantique d'expansion du préprocesseur mais encore une fois, cela reste une pratique déconseillée dans les développements industriels où la maintenabilité est une priorité.

2.3. Analyse & fiabilité

2.3.1. Analyse statique

Nous avons comparés plusieurs analyseurs statiques permettant de détecter des erreurs au *runtime*. Parmi ceux-ci, nous avons uniquement considéré les cinq qui ont la propriété d'être *corrects* et de garantir l'absence de certaines catégories de *bug* : Astree, ECLAIR, Frama-C, Polyspace et TrustInSoft Analyzer.

Comme indiqué dans l'Annexe B., nous avons indiqué les erreurs détectées d'après les documents publiques. Toutefois, ceux-ci ne sont pas forcément complets et il est possible que les outils détectent d'autres erreurs non mentionnées ici. Les cases cochées indiquent que l'outil détecte *au moins* le type d'erreur correspondant.

Erreur	Astrée	ECLAIR	Frama-C	Polyspace	TIS Analyser
Division par 0	✓	✓	✓	✓	✓
Débordement de tampon	✓	✓	✓	✓	✓
Déréférencement de NULL	✓	✓		✓	
Dangling pointer	✓	✓	✓		✓
Data race	✓	✓			
Interblocage	✓	✓	✓		
Vulnérabilités de sécurité	✓	✓			
Arithmétique entière	✓		✓		✓
Arithmétique flottante	✓		✓		
Code mort	✓	✓	✓	✓	
Initialisation	✓	✓	✓	✓	✓
Flot de données	✓	✓	✓		
Contrôle de flôt	✓		✓	✓	
Flôt de signaux	✓				
Non-interférence	✓				
Fuites mémoire		✓			
Double `free`		✓			
Coercions avec perte		✓	✓		✓
Mémoire superflue		✓			
Arguments variadiques		✓	✓		
Chaînes de caractères		✓			
Contrôle d'API		✓		✓	

Table 8: Comparaison des analyseurs statiques pour le langage C

Toutes les erreurs indiquées dans la Table 8 ne sont pas forcément des erreurs à *runtime* mais cela permet de se faire une idée des possibilités de chaque outil.

2.3.2. Meta formalisation

Le langage C, ou du moins un sous-ensemble, a été formalisé à travers le projet CompCert et cette formalisation est utilisable via l'outil VerifiedC[9] pour prouver des propriétés sur la version Coq du programme donné à l'outil. C'est une démarche qui permet de formaliser le C sans y toucher directement mais qui peut être compliquée à utiliser en pratique car le Coq engendré peut radpiement être volumineux et, même si VerifiedC nous aide un peu en fournissant des tactiques, cela reste réservé à des spécialistes de la preuve formelle et de Coq en particulier.

Les autres outils permettant de formaliser un programme C fonctionnent tous sur le même principe d'annotations du code source avec une contractualisation utilisant une logique de séparation. Parmi ceux-ci, nous avons comparé Frama-C, RefinedC et VerCors.

[9] https://vst.cs.princeton.edu/veric/

La différence entre ces outils réside principalement dans la syntaxe des annotations et la manière dont les spécifications sont vérifiées. Frama-C utilise une syntaxe E-ACSL qui est un sous-ensemble du langage *ANSI/ISO C Specificiation Language* (ACSL). Les spécifications et un modèle sémantique du C sont ensuite traduites en un problème SMT[10] soumis à plusieurs solveurs (Z3, CVC5, Alt-Ergo) qui déterminent, ou pas, si les contraintes sont satisfaites. Dans certains cas, il est également possible de générer des contre-exemples. En cas d'échec dans la preuve, il est possible de raffiner la spécification pour découper les preuves en sous-problèmes plus simples.

RefinedC utilise un langage de spécification déclaratif qui a peu ou prou la même expressité que E-ACSL. En revanche, la vérification du programme se fait de manière déductive en utilisant Coq et un modèle sémantique du C écrit en Coq avec le cadre théorique Iris. L'expressivité du modèle permet d'exprimer des problématiques arbitrairement complexes (comme par exemple l'*ownership*) tant que cela reste prouvable. En cas d'échec de la preuve, il est possible d'écrire directement les preuves en Coq.

VerCors utilise un langage de spécification nommé PVL[11] qui ressemble un peu à celui de Frama-C. L'outil est en fait basé sur un autre cadre logiciel appelé Viper qui permet de vérifier des programmes en utilisant la logique de séparation mais également la logique de permission, ce qui permet d'exprimer des propriétés d'*ownership*. Viper utilise le solveur SMT Z3 pour décharger les preuves.

2.3.3. WCET

La complexité du calcul statique du WCET fait qu'il y a peu d'outil disponibles. Nous en avons comparés six: Chronos, Bound-T, aiT, SWEET, Otawa et RapiTime. Chronos, SWEET et Otawa sont des outils académiques tandis que aiT et RapiTime sont des outils commerciaux. Bound-T est à la base un outil commercial mais qui n'est plus maintenu et qui a été rendu *open source*.

[10] *Satisfiability Modulo Theories*
[11] *Prototypal Verification Language*

Outil	WCET statique	WCET dynamique	WCET hybride	Architecture cible
Chronos	✓	✓		
Bound-T	✓			• Analog Devices ADSP21020 • ARMv7 TDMI • Atmel AVR (8-bit) • Intel 8051 (MCS-51) series • Renesas H8/300 • SPARC v7 / ERC32
aiT	✓			• Am486, IntelDX4 • ARM • C16x/ST10 • C28x • C33 • ERC32 • HCS12 • i386DX • LEON2, LEON3 • M68020, ColdFire MCF5307 • PowerPC • TriCore • V850E
SWEET	✓			
Otawa	✓			• ARM v5, • ARM v7, • PowerPC (including VLE mode), • Sparc, • TriCore, • Risc-V
RapidTime			✓	• ARM (7, 9, 11, Cortex-A, Cortex-R, Cortex-M) • Analog Devices • Atmel • Cobham Gaisler • ESA • Freescale (NXP) • IBM • Infineon • Texas Instruments

Notons que Chronos et SWEET ne ciblent pas d'architecture particulière mais se basent sur une représentation intermédiaire, respectivement un sur-ensemble MIPS et ALF, pour effectuer leur analyse. L'avantage est qu'il est techniquement possible de cibler n'importe quelle architecture dès lors qu'il y a un traducteur du langage source vers la représentation intermédiaire. Comme celle-ci ne tient pas compte de toutes les spécificités de l'architecture ciblée, le WCET calculé est *a priori* moins précis.

Otawa fonctionne avec des fichiers décrivant l'architecture cible; ce qui le rend a priori compatible avec toutes les architectures modulo le temps d'écrire ces descriptions si elles n'existent pas déjà. Les architectures indiquées sont celles dont les descriptions sont déjà disponibles.

2.3.4. Pile

Il existe plusieurs outils pour analyser statiquement la pile utilisée par un programme. Parmi ceux-ci, nous avons comparé GCC, StackAnalyzer, T1.stack et Arm Linker.

GCC propose une option `-fstack-usage` qui permet de générer un fichier décrivant l'utilisation de la pile par le programme. Le fichier généré contient la taille de pile utilisée par fonction et peut être analysé par un outil tiers pour en extraire des informations sur la taille maximale de la pile utilisée.

StackAnalyzer est un outil commercial développé par AbsInt qui semble offrir une vue plus précise (et graphique) de l'utilisation de la pile par le programme et propose des rapports orientés vers la qualification logicielle. L'outil supporte une gamme précise de couple compilateur-architecture qui, pour des raisons de lisibilité, n'est pas indiquée explicitement ici mais un lien vers page idoine de l'outil est donné. Les architectures communes (Intel, ARM, PowerPC, RISC-V, ...) sont supportées.

T1.stack est un outil commercial développé par Gliwa qui propose une analyse statique de la pile pour des architectures spécifiques. L'outil peut utiliser des annotations pour résoudre les appels utilisant des pointeurs de fonctions à l'exécution. Ces annotations peuvent être manuelles ou générées automatiquement par des mesures dynamiques.

Le *linker* Arm Linker d'ARM propose une option `--callgraph` qui engendre un ficher HTML contenant l'arbre d'appels du programme et l'utilisation de la pile. Comme pour T1.stack, il est possible d'ajouter une analyse dynamique pour obtenir des informations plus précises en cas d'utilisation de pointeurs de fonction.

Outil	Annotations	Architectures
GCC		Toutes celles supportées par GCC
StackAnalyzer		Liste exhaustive sur le site de l'outil : https://www.absint.com/stackanalyzer/targets.htm
T1.stack	✓	• Infineon TC1.6.X, TC1.8 • NXP/STM e200z0-z4, z6, z7 • ARM (v7-M, v7-R, V8-R) • Renesas RH850, G3K/G3KH/G3M/G3MH • Intel x86-64
Arm Linker		ARM

2.3.5. Qualité numérique

La qualité numérique peut être analysée de deux manières: statiquement et dynamiquement. Les outils Fluctuat, Astree et Polyspace font partie des outils d'analyse statique. Polyspace détecte essentiellement des erreurs à *runtime* comme la division par 0, les dépassements de capacité et les débordements de buffer. Astree détecte également les erreurs de runtime mais réalise également un calcul d'intervalles permettant d'évaluer les erreurs d'arrondis. Fluctuat est un outil académique qui est spécifiquement dédié à l'analyse numérique flottante par interprétation abstraite en utilisant un domaine basé sur l'arithmétique affine.

L'outil Gappa fonctionne également par analyse statique mais en utilisant un programme annoté (via Frama-C) par les propriétés à vérifier sur les calculs flottants.

Les analyses dynamiques fonctionnent en instrumentant le code avec des bibliothèques logicielles dédiées. Parmi celles-ci, on trouve CADNA qui utilise une approche par estimation stochastique des arrondis de calculs.

Une autre approche consiste à utiliser directement des bibliothèques proposant des calculs flottants plus précis comme la bibliothèque MPFR qui réalise un calcul par intervalles ou la bibliothèque GMP qui permet de réaliser des calculs avec une précision arbitraire.

2.3.6. Assurances

Aujourd'hui, le niveau d'assurance proposé par le C est très inégal. Le langage en lui-même est l'un de ceux qui proposent le moins de mécanisme de protection ou de vérification mais ce manque a engendré au fil du temps une offre de fiabilisation très large à travers des outils d'analyse (statique ou dynamique), des outils de tests, des référentiels de programmation, *etc.*. Leur utilisation va, paradoxalement, permettre d'apporter un niveau d'assurance de qualité élévé pour un langage qui n'en dispose que très peu.

Toutefois, l'utilisation d'outils tiers à un coût en licences, en formation et en temps d'utilisation dans un projet. Ainsi, la fiabilité d'un programme C va essentiellement dépendre des moyens mis en oeuvre pour l'assurer et peu du langage lui même.

2.3.7. Utilisation dans le critique

Le C est notoirement utilisé dans tous les domaines critiques soit directement pour exploiter des systèmes embarqués soit indirectement comme langage cible pour d'autres langages (Ada, Scade, ...).

3. C++

3.1. Description

Le C++ est une extension du langage C créée par Bjarne Stroustrup en 1979. Il ajoute au C les concepts de la programmation orientée objet et la généricité (les *templates*). Pleinement compatible avec le C, le C++ peut s'utiliser dans les mêmes contextes que le C mais il est plus souvent utilisé pour écrire des applications complexes nécessitant une certaine abstraction.

Il est standardisé pour la première fois en 1998 [6] puis régulièrement mis-à-jour jusqu'à la dernière version en 2020 [7]. Comme pour le C, il existe des référentiels de programmation pour garantir une certaine qualité de code. Le plus connu est le référentiel MISRA-C++ [8].

3.1.1. Paradigme

Le C++ est un langage de programmation objet et impératif. Toutefois, l'un des éléments ayant particulièrement contribué au succès du C++ est l'introduction des *templates* qui offrent une forme de (meta)programmation générique.

3.1.2. Mécanismes intrinsèques de protection

Le C++ offre plus de garanties intrinsèques que le C à travers trois mécanismes :

- la programmation objet;
- les exceptions;
- les *templates*.

La programmation objet permet de définir des classes et des objets qui encapsulent des données et des méthodes. En C++, cette encapsulation est finement contrôlable avec des spécificateurs d'accès (`public`, `protected`, `private`) qui permettent de limiter la visibilité des membres d'une classe. Ce mécanisme, cumulé avec le polymorphisme introduit par l'héritage permet de contractualiser les interfaces. Dans l'exemple suivant :

```
1  class Animal
2  {
3  private:
4      static uint32_t _count;
5      const uint32_t _id;
6  protected:
7      std::string _name;
8      static uint32_t fresh(void) { return ++_count; }
9      virtual std::string noise(void) const = 0;
10     virtual uint32_t id(void) const { return _id; }
11 public:
12     Animal(std::string name) : _id(fresh()), _name(name) {}
13
14     virtual void show(void) const {
15         std::cout << _name << std::endl;
16     }
17
18     virtual void shout(void) const {
19         std::cout << this->noise() << "!" << std::endl;
20     }
21 };
22
23 uint32_t Animal::_count = 0;
24
25 class Dog : public Animal
26 {
27 public:
28     Dog(std::string name) : Animal(name) {}
29 protected:
30     std::string noise(void) const override { return "Ouaf"; }
31 };
32
33 class Cat : public Animal
34 {
35 public:
36     Cat(std::string name) : Animal(name) {}
37     void show(void) const override {
38         std::cout << _name << "(" << this->id() << ")" << std::endl;
39     }
40 protected:
41     std::string noise(void) const override { return "Miaou"; }
42 };
```

on définit une classe abstraite `Animal` et deux sous classes `Dog` et `Cat`. À chaque animal est associé un identifiant (le membre `_id`), une identification (la méthode `show`) et un son (la méthode `noise`). À l'instanciation d'`Animal`, on peut attribuer un identifiant à l'objet via la méthode `fresh`. `fresh` est déclarée `private` et ne peut donc être utilisée que par l'interface `Animal`. La méthode `noise` est définie par les sous-classes mais elle reste ici `protected` de sorte qu'elle ne peut pas être appelée de manière externe à l'objet ou les classes dérivées. La classe `Cat` redéfinit la methode `show` qui permet d'identifier l'animal en y ajoutant son identifiant auquel il a le droit d'accéder que grâce à la méthode `id` déclarée `protected` (le champs `_id` est privé).

Cet exemple montre bien qu'à travers un ensemble de mot clés comme les spécificateurs d'accès ou les `virtual` et `override`, on peut établir des contrats d'utili-

sation entre les classes et vis-à-vis du code appelant. Utilisé correctement, il est facile d'établir des invariants par construction même si ceux-ci ne sont pas explicitement formalisés.

Les exceptions offrent un moyen de gérer les erreurs qui, en toute théorie, est plus robuste que les codes de retour et la programmation défensive utilisée en C. Dans cette dernière, il est facile de rater ou de mal interpréter un code de retour et obtenir un programme qui arrive dans état incohérent de manière silencieuse de sorte qu'il est difficile de déméler la situation. Avec le mécanisme d'exceptions, il est commun de lancer une exception en cas d'erreur. Celle-ci peut être rattrappée à n'importe quel moment dans le flôt de contrôle appellant et, en cas de doute, il est possible d'attrapper toutes les exceptions de sorte qu'aucune erreur levée ne peut s'échapper et conduire à une erreur à l'exécution.

Enfin, le mécanisme des *templates* et de spécialisation permet d'expliciter les contraintes de typage de sorte qu'il est possible de renforcer arbitrairement le typage du langage pour obtenir un programme fortement typé de bout en bout. Il est même possible d'encoder des propriétés plus fortes puisque le langage de *template* est Turing complet et que les *templates* sont évalués à la compilation.

Tous ces mécanismes permettent, en théorie, de renforcer la qualité et la fiabilité d'un programme C++. En pratique cependant, la multiplicité des concepts et la sémantique complexe du langage font que tous ces mécanismes sont difficiles à bien maîtriser. Or, mal utilisés, ces mécanismes ont un effet inverse en fragilisant la fiabilité du code produit. Par ailleurs, même bien utilisés, la multiplication des abstractions prend les programmes C++ peu lisibles et posent souvent des problèmes de maintenabilité.

En conséquence, les normes C++ en matière de logiciel critique (comme le MISRAC++[8]) sont très strictes et demandent de bien vérifier que l'usage des fonctionnalités du C++ sont utilisées de manière à laisser le moins de doutes possibles. En conséquence et comme pour le C, la fiabilité des programmes C++ va en partie dépendre de l'utilisation d'outils tiers.

3.1.3. Compilateurs

Compilateur	Architectures	Licence
AOCC	AMD x86 (32 et 64 bits)	Propriétaire, Gratuit
C++ Builder	x86 (32 et 64 bits)	Commercial
Clang	AArch64, ARMv7, IA-32, x86-64, ppc64le	Apache 2.0
GCC	IA-32, x86_64, ARM, PowerPC, SPARC, ...	GPLv3+
ICX (Intel C/C++ Compiler)	IA32, x86-64	Propriétaire, Gratuit
MSVC	IA-32, x86_64, ARM	Propriétaire
NVIDIA HPC SDK	x86-64, ARM	Propriétaire
Oracle C++ Compiler	SPARC, x86 (32 et 64 bits)	Propriétaire, Gratuit

Le tableau ci-dessus présente les principaux compilateurs C++ de bout en bout disponibles. Nous précisons de bout en bout car il existe des *front ends* qui ne s'occupent que de traduire le C++ en C (comme EDG C++ Front End). Tous les

compilateurs ne sont pas listés ici car beaucoup d'entre eux ne sont plus maintenus ou sont *a priori* moins pertinents pour des projets critiques.

3.1.4. Interfaçage

Le C++ peut utiliser du C nativement. Pour utiliser du C++ en C, il suffit d'indiquer que les fonctions à exporter sont `extern "C"` de sorte que le compilateur en donne une version compatible avec le C.

Dès lors que l'interopérabilité avec le C est complète, celle-ci, par transitivité, l'est aussi avec tout langage s'interfaçant avec le C, c'est-à-dire la plupart des langages.

3.1.5. Adhérence au système

Comme pour le C, le C++ peut fonctionner sur un système nu et sans bibliothèque standard.

3.1.6. Gestionnaire de paquets

En plus des gestionnaires de paquets Conan et vcpkg qui supportent les paquets écrits en C++, il y a également Buckaroo qui est dédié au C++.

Gestionnaire	Buckaroo	Conan	vcpkg
Plateformes	Linux, Windows, MacOS	Toutes	Linux, Windows, MacOS
Format	ToML	Python	JSON
Résolution des dépendances	✓	✓	✓
Cache binaire	✓	✓	✓
Nombre de paquets	350	1750	2500

3.1.7. Communauté

Le C++ est un des langages les plus populaires et les plus utilisés depuis plus de 20 ans et ce, dans tous les domaines de l'informatique. Toutefois, le langage est maintenant en concurrence directe avec Rust qui intègre la plupart de ses qualités intrinsèques ou méthodologiques. Il est donc probable qu'une partie de la communauté C++ se déplace vers la communauté Rust avec le temps.

3.2. Outillage

3.2.1. Débugueurs

Les mêmes débuggeurs que pour le C sont utilisables pour le C++.

3.2.2. Tests

Certains des outils cités pour le C fonctionnent également pour le C++. C'est le cas pour Cantata ou Parasoft par exemple. D'autres outils ou bibliothèques ont été conçus spécifiquement pour le C++. La plus connue des bibliothèques C++, Boost, propose également une bibliothèque facilitant l'écriture de tests. De base, elle définit sensiblement les mêmes macros que beaucoup de bibilothèques similaires mais avec quelques options de générations intéressantes sur les rapports de test ou les possibilités de *fuzzing*. Le *mocking* est également disponible via une extension. Catch2 propose sensiblement la même chose que Boost Test Library.

Google Test est un cadre logiciel proposé par Google pour le test unitaire. Il est très complet et utilisé pour de gros projets. Il permet de découvrir les tests au-

tomatiquement, de les exécuter et de générer des rapports de tests détaillés dans des formats personnalisables. Il est aussi extensible au fil des besoins de l'utilisateur. Safetynet permet également de générer les tests automatiquement à partir de classes décrivant les tests et à l'aide d'un script Ruby. L'outil semble cependant moins mature et complet que Google Test.

Mockpp est une bibliothèque de *mocking* pour le C++. Elle s'utilise plutôt conjointement avec d'autres outils de tests unitaires pour simuler des comportements de fonctions ou de classes à la manière de Opmock.

Testwell CTC++ est un outil commercial qui, comme Cantata, Parasoft ou VectorCAST/C++ que nous avons déjà présenté dans la partie C, couvre différents types de tests (unitaires, couverture) et est adapté aux usages de l'embarqué critique.

Outil	Tests	Generation	Gestion	*mocking*
Boost Test Library	U	+	✓	
Catch2	U	+	✓	
Cantata	UIRC	+++	✓	
Criterion	U	+		
LibCester	U	+	✓	✓
Google Test	U	++	✓	✓
Mockpp	U	+		✓
Opmock	U	++		✓
Parasoft	UC	++	✓	
Safetynet	U	++	✓	
Testwell CTC++	UIC	++	✓	
TPT	UINC	+++	✓	
VectorCAST/C++	UC	++	✓	✓

Table 13: Outils de tests pour le C++

3.2.3. Parsing

Les outils de *parsing* cités pour le C fonctionnent également pour le C++ mais il en existe d'autres ciblant ce dernier. Si l'on passe les outils non maintenus, insuffisamment documentés ou trop jeunes pour être utilisés en production, on peut citer au titre des *lexers* les outils de la Table 14.

Nom	Code	Plateforme	License
Astir	Astir	Toutes	Libre, MIT
RE/flex	Flex	Lexer	Libre, BSD

Table 14: Lexers pour le C++

Au niveaux des *parsers* plus généraux, beaucoup sont dynamiques : ils se présentent sous la forme de bibliothèques dont l'API permet de décrire un langage (et la manière de le parser) à l'exécution. Cette dynamicité peut être un atout pour définir des grammaires évolutives mais dans la pratiques, les *parsers* à émission de code sont plus adaptés. Il y a toutefois des exceptions avec les bibliothèques

utilisant la méta-programmation (comme Boost Spirit ou PEGTL) pour engendrer le gros du *parsing* à la compilation.

Les *parsers* cités pour le C fonctionnent pour le C++ et les classiques Bison et ANTLR sont bien suffisants dans la plupart des cas.

3.2.4. Meta programmation

La méta programmation est un des aspects mis en avant dans le C++ à travers l'utilisation des *templates* et de la spécialisation qui peuvent être vus comme un langage de macro de haut niveau. Celui-ci est suffisamment expressif pour être Turing-complet et permet de calculer virtuellement n'importe quoi qui ne dépende pas d'une entrée dynamique. Cette façon de programmer est souvent utilisée pour précalculer des données numériques (comme des tables trigonométriques) qui pourront être utilisées instantanément à l'exécution. Le cas d'école est celui de la factorielle :

```
 1 template<unsigned int N>
 2 struct Fact
 3 {
 4     enum {Value = N * Fact<N - 1>::Value};
 5 };
 6
 7 template<>
 8 struct Fact<0>
 9 {
10     enum {Value = 1};
11 };
12
13 unsigned int x = Fact<4>::Value;
```

Dans cet exemple, on définit un *template* de structure `Fact` paramétré par un entier `N`. Dans cette structure, on définit un type énuméré n'ayant qu'une seule valeur `Value` à laquelle on attribue la valeur correspondant à l'expression `N * Fact<N - 1>::Value`. Le compilateur en prend note mais ne calcule rien car les *templates* sont instanciés à l'usage. On définit également une spécialisation *template* `Fact` pour le cas où `N` vaut 0. Dans ce cas, la valeur de `Value` est 1. Lorsque vient le moment d'instancier ce *template* avec la définition de la valeur `x` associée à `Fact<4>::Value`, le compileur va dérouler la définition en utilisant le *template* paramétré par `N` avec `N = 4`, c'est à dire `4 * Fact<3>::Value` puis il va continuer à dérouler `4 * (3 * Fact<2>::Value)` puis `4 * (3 * (2 * Fact<1>::Value))` puis `4 * (3 * (2 * (1 * Fact<0>::Value)))`. À ce moment là, comme `N = 0`, c'est la spécialisation `Fact<0>` qui est utilisée et donc `Fact<0>::Value = 1`. Cela donne au final l'expression `4 * (3 * (2 * 1))`. Le compilateur sait simplifier ce type d'expression statiquement et va la remplacer de lui même par 24 à la compilation de sorte qu'à l'exécution aucun calcul ne sera nécessaire pour calculer la valeur de `x`.

Cette technique est un peu utilisée dans la bibliothèque standard mais est très utilisée dans la bibliothèque Boost. Bien que la technique soit très séduisante pour gagner un maximum de performance à l'éxécution, elle présente plusieurs inconvénients :

- plus on l'utilise plus la compilation est longue puisqu'on demande au compilateur de calculer des choses qu'on aurait normalement calculé à l'exécution. Précalculer des tables trigonométriques peut prendre des heures, voire des jours…

- La moindre erreur dans le flôt d'instanciation déroule tout le flôt d'instanciation et les messages d'erreurs sont souvent incompréhensibles pour la plupart des développeurs.

La métaprogrammation en C++ s'accompagne donc un surcoût en temps ou en puissance de calcul avec un risque significatif d'augmentation de la dette technique.

3.2.5. Dérivation

La dérivation de code en C++ repose sur la méta-programmation autorisée par les *templates* et la spécialisation. Cependant, comme le langage n'est pas réflexif, on ne peut pas dériver directement du code à partir des définitions. Toutefois, le système des *traits* (des *templates* dirigés sur les propriétés sur les types) permet de construire des dérivations par spécialisation.

3.3. Analyse & fiabilité

3.3.1. Analyse statique

Les analyseurs statiques cités pour le C sont indiqués comme fonctionnant également pour le C++ mais avec un niveau de support obscur. Astree supporte le standard C++17 tandis que Frama-C indique clairement un support balbutiant (via un *plugin* utilisant Clang pour traduire la partie C++ en une représentation plus simple).

Les autres outils ne donnent pas explicitement leur niveau de support du C++.

3.3.2. Meta formalisation

La seule meta formalisation du C++ connue est BRiCk qui traduit essentiellement le langage C++ vers Coq pour ensuite bénéficier du cadre logiciel Iris. Le but est de pouvoir démonter des propriétés sur des programmes C++ concurrents.

3.3.3. WCET

Les outils de calcul du WCET cités pour le C fonctionnent également pour le C++. Otawa fournit notamment un cadre logiciel en C++ qui, intégré au développement, améliore l'analyse avec des données dynamiques.

3.3.4. Pile

Les outils d'analyse statique de pile cités pour le C fonctionnent également pour le C++ (tant qu'il existe un compilateur C++ pour l'architecture cible) puisqu'ils se basent sur une analyse du fichier binaire et non du code source.

3.3.5. Qualité numérique

Comme pour le C, Astree et Polyspace détectent statiquement les erreurs à *runtime*. CADNA est aussi utilisable pour des une analyse dynamique mais Fluctuat et Gappa ne sont pas utilisable directement sur du C++.

XSC (eXtended Scientific Computing) est une bibliothèque de calcul numérique qui réimplémente les calculs sur les flottants avec une précision arbitraire. Cette implémentation donne des résultats arbitrairement précis (en fonction de la précision choisie) mais est assez lente.

MPFR a plusieurs implémentations en C++. Les plus maintenues et à jour sont MPFR++ et Boost. MPFR++ utilise la précision maximale qu'il trouve dans une expression et arrondi le résultat à précision de la variable cible. Boost utilise une

précision explicite avec différentes implémentations suivant les classes choisies (pur C++, basée sur GMP, basée sur GMP et MPFR).

3.3.6. Assurances

Comme pour le C, le cas du C++ est paradoxal. En effet, le langage porte en lui-même des éléments permettant d'assurer des contraintes très fortes de typage ou certains invariants à l'aide des *templates* et les spécificateurs d'accès. Toutefois, l'ensemble est si complexe à maîtriser qu'il peut induire des surcoûts dans toutes les étapes du cycle de vie du logiciel :

- au développement : une équipe hétérogène d'ingénieur peut mettre plus de temps à concevoir et débugger le logiciel tandis qu'une équipe de spécialistes diminuera le temps de développement mais coutera plus cher.
- à la vérification : la vérification du C++ est plus complexe que celle du C et toutes les constructions C++ ne sont pas supportées par les outils de vérification.
- à la maintenance : le code étant plus complexe, il est plus difficile à maintenir et à faire évoluer.

En conséquence, lorsque le C++ est utilisé dans des domaines ou l'un de ces critèques est déterminant, il est nécessaire d'utiliser conjointement un outil de vérification de règles de codage pour s'assurer des bonnes pratiques (en plus d'une batterie de test importante).

3.3.7. Utilisation dans le critique

Comme le C, le C++ est utilisé dans tous les domaines critiques.

4. Ada

4.1. Description

Le langage Ada a été créé à la fin des années 70 au sein de l'équipe CII-Honeywell Bull dirigé par Jean Ichbiah en réponse à un cahier des charges du département de la Défense des États-Unis. Le principe fût de créer un langage spécifiquement dédié aux systèmes temps réels ou embarqués requérant un haut niveau de sûreté.

Le langage est standardisé pour la première fois en 1983 [9] sous le nom d'*Ada83*. La dernière norme ISO *Ada 2022* a été publiée en 2023 [10]. Notons que la norme *Ada 2012* est librement téléchargeable [11].

4.1.1. Paradigme

Ada est un langage de programmation objet et impératif. Depuis la norme *Ada 2012*, le paradigme contrat a été ajouté au langage.

4.1.2. Mécanismes intrinsèques de protection

Typage

Le langage Ada dispose d'un système de typage riche et statiquement vérifié par le compilateur. Contrairement aux langages comme C ou C++, Ada est *fortement typé* au sens qu'il ne fait pas de conversions implicites entre des types différents. Par exemple l'expression `2 * 2.0` produira une erreur de compilation car le type de `2` est `Integer` et le type de `2.0` est `Float`.

En plus des habituels types scalaires (entiers, flottants, ...), des enregistrements et des énumérations, le langage dispose de l'abstraction de type et d'un système de sous-typage.

Il est possible de construire un nouveau type à partir d'un autre type via la syntaxe `type New_type is new Old_type`. Par exemple, le programme suivant:

```
1 procedure Foo is
2   type Kilos is new Float;
3   type Joules  is new Float;
4
5   X : Kilos;
6   Y : constant Joules := 10.0;
7 begin
8   X := Y;
9 end Foo;
```

ne compilera pas car bien que `X` et `Y` aient la même représentation mémoire (des flottants), ils ne sont pas du même type.

Sous-typage

Ada dispose également d'une particularité qui lui permet de définir des sous-types arbitraires. Par exemple, si l'on veut un compteur dont on sait que les valeurs vont de 1 à 10, on peut définir le type correspondant :

```
1 subtype Counter is Integer range 1 .. 10;
```

Contrats

Depuis la norme Ada 2012, il est possible d'ajouter explicitement des *contrats* sous forme de préconditions, postconditions et d'invariants. Par exemple la fonction suivante implémente la racine carrée entière qui n'est définie que pour les entiers positifs.

```
1 function Isqrt (X : Integer) return Integer
2 with
3   Pre => X >= 0,
4   Post => X = Isqrt'Result * Isqrt'Result
5 is
6   Z, Y : Integer := 0;
7 begin
8   if X = 0 or X = 1 then
9     return X;
10  else
11    Z := X / 2;
12    Y := (Z + X / Z) / 2;
13    while Y < Z loop
14      Z := Y;
15      Y := (Z + X / Z) / 2;
16    end loop;
17    return Z;
18  end if;
19 end Isqrt;
```

Compilé avec l'option `-gnata` du compilateur GNAT, on obtiendra une erreur à l'exécution si on appelle cette fonction avec une valeur négative.

On peut également spécifier des invariants pour des types dans les interfaces des *packages*. Par exemple, pour une implémentation des intervalles fermés, on peut garantir que l'unique représentant de l'intervalle vide est donné par le couple d'entiers `(0, -1)`.

```
package Intervals is
  type Interval is private
    with Type_Invariant => Check (Interval);

  function Make (L : Integer; U : Integer) return Interval;

  function Inter (I : Interval; J : Interval) return Interval;

  function Check (I : Interval) return Boolean;

  private
    type Interval is record
      Lower : Integer;
      Upper : Integer;
    end record;
end Intervals;

package body Intervals is
  function Make (L : Integer; U : Integer) return Interval is
    (if U < L then (0, -1) else (L, U));

  function Min (X : Integer; Y : Integer) return Integer is
    (if X <= Y then X else Y);

  function Max (X : Integer; Y : Integer) return Integer is
    (if X <= Y then Y else X);

  function Inter (I : Interval; J : Interval) return Interval is
    Make (Max (I.Lower, J.Lower), Min (I.Upper, J.Upper));

  function Check (I : Interval) return Boolean is
    (I.Lower <= I.Upper or (I.Lower = 0 and I.Upper = -1));
end Intervals;
```

La fonction `Check`, dont l'implémentation n'est pas exposée dans l'interface, s'assure que le seul intervalle vide est l'intervalle `[0, -1]`.

Concurrence

Le langage Ada intègre dans sa norme des bibliothèques pour la programmation concurrentielle. Le concept de *tâche* permet d'exécuter des applications en parallèle en faisant abstraction de leur implémentation. Une tâche peut ainsi être exécutée via un thread système ou un noyau dédié. Il est également possible de donner des propriétés aux tâches et de les synchroniser comme avec l'exemple du Listing 3. Dans cet exemple, la procédure `Hello_World` crée deux tâches `T1` et `T2` qui décrémentent un compteur partagé `Counter` jusqu'à ce qu'il atteigne 1.

```
with Ada.Text_IO; Use Ada.Text_IO;

procedure Hello_World is
  protected Counter is
    procedure Decr(X : out Integer);
    function Get return Integer;
  private
    Local : Integer := 20;
  end Counter;

  protected body Counter is
    procedure Decr(X : out Integer) is begin
      X := Local;
      Local := Local - 1;
    end Decr;

    function Get return Integer is begin
      return Local;
    end Get;
  end Counter;

  task T1;
  task T2;

  task body T1 is
    X : Integer;
  begin
    loop
      Counter.Decr(X);
      Put_Line ("Task 1: " & Integer'Image (X));
      exit when Counter.Get <= 1;
    end loop;
  end T1;

  task body T2 is
    X : Integer;
  begin
    loop
      Counter.Decr(X);
      Put_Line ("Task 2: " & Integer'Image (X));
      exit when Counter.Get <= 1;
    end loop;
  end T2;
begin
  null;
end Hello_World;
```

Listing 3: Exemple de programmation concurrentielle en Ada

Temps réel

Le profile *Ravenscar* est un sous-ensemble du langage Ada conçu pour les systèmes temps réel. Il a fait l'objet d'une standardisation dans *Ada 2005*. En réduisant les fonctionnalités liées aux multi-tâches, ce profile facilite en outre la vérification automatique des programmes.

4.1.3. Compilateurs

Parmi tous les compilateurs Ada, nous listons uniquement ceux qui semblent maintenus et de qualité industrielle.

Compilateur	Plateformes	Licence	Normes[12]
PTC ObjectAda	Toutes	Propriétaire	95, 2005, 2012
GCC GNAT	Toutes	GPLv3+	95, 2005, 2012
GNAT Pro	Toutes	Propriétaire	95, 2005, 2012
GNAT LLVM	Toutes	GPLv3+	?
GreenHills Ada Optimizing Compiler	Windows, Linux	Proprietaire	
PTC ApexAda	Linux	Propriétaire	
Janus/Ada	Windows	Propriétaire	95, 2005, 2012

Les compilateurs GNAT, GNAT Pro et Janus/Ada proposent également un mode Ada83 mais ne donnent pas de garantie quant au respect de ce standard.

Le langage Ada a une longue tradition de validation des compilateurs. Ce processus de validation a fait l'objet en 1999 d'une norme ISO[12]. L'*Ada Conformity Assessment Authority* (abrégée *ACAA*) est actuellement l'autorité en charge de produire un jeu de tests (*Ada Conformity Assessment Test Suite*) validant la conformité d'un compilateur avec les normes Ada. Elle propose la validation pour les normes Ada83 et Ada95 à travers des laboratoires tiers indépendants.

En plus de cette validation, certains compilateurs ont fait l'objet de certifications pour la sûreté ou la sécurité. Ansi GNAT Pro dispose des certifications de sûreté :
- DO-178C ;
- EN-50128 ;
- ECSS-E-ST-40C ;
- ECSS-Q-ST-80C ;
- *ISO 26262*

ainsi que des certifications de sécurité:
- DO-326A/ED-202A ;
- DO-365A/ED-203A.

4.1.4. Interfaçage

Ada peut être interfacé avec de nombreux langages. Les bibliothèques standards contiennent des interfaces pour les langages C, C++, COBOL et FORTRAN. L'exemple ci-dessous est issu du standard d'Ada 2012:

[12] Nous ne considérons ici que les trois normes ISO Ada95, Ada 2005 et Ada 2012.

```
--Calling the C Library Function strcpy
with Interfaces.C;
procedure Test is
  package C renames Interfaces.C;
  use type C.char_array;
  procedure Strcpy (Target : out C.char_array;
                    Source : in  C.char_array)
      with Import => True, Convention => C, External_Name => "strcpy";
  Chars1 :  C.char_array(1..20);
  Chars2 :  C.char_array(1..20);
begin
  Chars2(1..6) := "qwert" & C.nul;
  Strcpy(Chars1, Chars2);
-- Now Chars1(1..6) = "qwert" & C.Nul
end Test;
```

Certains compilateurs proposent également d'écrire directement de l'assembleur dans du code Ada. Pour ce faire, il faut inclure la bibliothèque `System.Machine_Code` dont le contenu n'est pas normalisé par le standard. Par exemple, le compilateur GNAT propose une interface similaire à celle proposée par GCC en langage C:

```
with System.Machine_Code; use System.Machine_Code;

procedure Foo is
begin
  Asm();
end Foo;
```

4.1.5. Adhérence au système

Le langage Ada peut être utilisé dans un contexte de programmation *bare metal*.

- *GNAT FSF* permet l'utilisation de *runtime* personnalisé,
- *GNAT Pro* est livré avec plusieurs *runtime*:
  - *Standard Run-Time* pour les OS classiques (Linux, Windows, VxWorks et RTEMS),
  - *Embedded Run-Time* pour les systèmes *bare metal* avec le support des tâches,
  - *Light Run-Time* pour développer des applications certifiables sur des machines ayant peu de ressources;
- *GreenHills Ada Optimizing Compiler* fournit plusieurs implémentations de *runtime* pour des cîbles différentes [13],
- *PTC* distribue un *runtime* pour *PTC ObjectAda* pour VxWorks et LynxOS sur PowerPC.
- *PTC ApexAda* propose également un *runtime* dans un contexte *bare metal* pour l'architecture Intel X86-64 [14].

Notons enfin qu'une des forces du langage est qu'en proposant dans sa norme une API pour la programmation concurrentielle et temps-réel, il permet de cibler plusieurs plateformes ou runtimes différents sans avoir à modifier le code source.

4.1.6. Gestionnaire de paquets

Alire (*Ada LIbrary REpository*) est un dépôt de bibliothèques Ada près à l'emploi. L'installation se fait via l'outil `alr` à la manière de `cargo` pour Rust ou `opam` pour OCaml. L'outil dispose d'environ 460 *crates*.

Techniquement, les gestionnaires de paquets agnostiques peuvent également gérer les bibliothèques Ada mais aucun support spéficique n'est recensé au delà des paquets `alire` et `gnat` pour Nix.

4.1.7. Communauté

La communauté Ada est structurée autour d'organismes qui promeuvent l'utilisation du langage dans la recherche et l'industrie.

- **Ada Europe** est une organisation internationale promouvant l'utilisation d'Ada dans la recherche [15].
- **Ada Resource Association** (ARA) est une association qui promeut l'utilisation d'Ada dans l'industrie [16].
- **Ada - France** est une association loi 1901 regroupant des utilisateurs francophones d'Ada [17].
- **The AdaCore blog** est un blog d'actualité autour du langage Ada maintenu par l'entreprise AdaCore[18].

4.2. Outillage

4.2.1. Débugueurs

Tous les débugueurs basés sur l'analyse binaire (GDB, LLDB, ...) sont compatibles avec Ada.

4.2.2. Tests

Les différents outils de tests recensés pour Ada sont inddiqués dans la Table 16.

Outil	Tests	Generation	Gestion	*mocking*
AUnit	U	+	✓	
Ada Test 95	UI	++	✓	
Avhen	U	+	✓	
LDRA	UIC	+	✓	
VectorCAST/Ada	UC	++	✓	✓
Rational Test RealTime	UIC			

Table 16: Comparaison des outils de tests pour le langage Ada

AUnit et Avhen sont des implémentations xUnit pour Ada. Les fonctionnalités sont toujours les mêmes et permettent de décrire des suites de tests unitaires avec un systèmes d'assertion et du *tooling* simplifiant la tâche. Comme pour les autres implémentations xUnit, un rapport au format JUnit est généré.

Ada Test 95 et LDRA (TBrun) sont des outils commerciaux de génération de tests unitaire et d'intégration. Ils offrent un support pour l'automatisation et sont conformes aux exigences de test dans les standards de sûreté.

VectorCAST/Ada est la version adaptée à Ada de l'outil de génération de tests VectorCAST/C++. Il permet de générer des tests unitaires, d'intégration et propose un support pour le *mocking*. Contrairement à Ada Test 95 et LDRA qui ne supportent que Ada95, VectorCAST/Ada supporte également les normes Ada 2005 et Ada 2012. L'outil semble spécialisé dans l'avionique et est conforme aux exigences de la DO-178B.

Rational Test RealTime est un outil commercial de test unitaire et d'intégration pour les systèmes temps réel. Il procède par instrumentation du code source et

génère des rapports de couverture de code. Il permet également de faire du profilage mais les informations sur la génération, la gestion et le *mocking* ne sont pas clairement documentées.

4.2.3. Parsing

Ada est rarement employé pour l'écriture de compilateurs et il y a peu de support dans ce domaine. Il y a un équivalent de Lex/Yacc avec AFlex/AYacc fournis par l'association Ada-France. Coco/R permet également de générer des analyseurs syntaxiques pour Ada.

4.2.4. Meta programmation

Il n'y a pas de support connu pour la méta-programmation en Ada.

4.2.5. Dérivation

Il n'y a pas de support connu pour la dérivation en Ada.

4.3. Analyse & fiabilité

4.3.1. Analyse statique

Les outils d'analyse statique (corrects) pour Ada sont:

- CodePeer (ou GNAT SAS) ;
- Polyspace ;
- SPARK Toolset.

Un comparatif (non exhaustif) des trois outils est donné dans la Table 17.

Erreur	CodePeer	Polyspace	SPARK Toolset
Division par 0	✓	✓	✓
Débordement de tampon	✓	✓	✓
Déréférencement de NULL	✓	?	
Dangling pointer			
Data race	✓		
Interblocage			
Vulnérabilités de sécurité			
Dépassement d'entier	✓	✓	✓
Arithmétique flottante			
Code mort	✓	✓	
Initialisation	✓	✓	
Flot de données			
Contrôle de flôt			
Flôt de signaux			
Non-interférence			
Fuites mémoire			
Double `free`		✓	
Coercions avec perte			
Mémoire superflue			
Arguments variadiques			
Chaînes de caractères			
Contrôle d'API			

Table 17: Comparatif des outils d'analyse statique pour le langage Ada

4.3.2. Meta formalisation

SPARK est la partie formelle du langage Ada qui lui ajoute la possibilité d'ajouter des propriétés sur le code (à la manière des contrats d'Ada 2012) et de les vérifier statiquement en les déchargeant sur des démonstrateurs SMT (Z3, CVC4 et Alt-Ergo).

4.3.3. WCET

Les outils RapiTime et aiT cités dans la partie C supportent explicitement Ada.

4.3.4. Pile

Les outils analysant l'exécutable peuvent le faire tout autant pour les programmes Ada. Pour les outils ciblant spécifiquement Ada, il y GNATstack et GNAT. Pour le second, les options suivantes permettent de contrôler l'usage de la pile:

- `-fstack-usage` qui produit une estimation de la taille maximale de la pile par fonction.
- `-Wstack-usage=BYTES` qui produit un message d'avertissement pour les fonctions qui pourraient nécessiter plus de `BYTES` octets sur la pile.

4.3.5. Qualité numérique

La langage Ada a la particularité d'être très rigoureux sur l'utilisation des types arithmétiques. Il n'y a pas de conversion implicite et le compilateur va adapter la représentation des valeurs en fonction de la précision demandée:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Custom_Floating_Types is
  type T3  is digits 3;
  type T15 is digits 15;
  type T18 is digits 18;
begin
  Put_Line ("T3  requires "
            & Integer'Image (T3'Size)
            & " bits");
  Put_Line ("T15 requires "
            & Integer'Image (T15'Size)
            & " bits");
  Put_Line ("T18 requires "
            & Integer'Image (T18'Size)
            & " bits");
end Custom_Floating_Types;
```

Ici, le programe affichera:

```
T3  requires  32 bits
T15 requires  64 bits
T18 requires  128 bits
```

Par ailleurs, le compilateur va automatiquement vérifier les potentiels débordements. Dans l'exemple suivant, on ajoute 5 à la valeur maximale d'un entier:

```
procedure Main is
  A : Integer := Integer'Last;
  B : Integer;
begin
  B := A + 5;
  --  This operation will overflow, eg. it
  --  will raise an exception at run time.
end Main;
```

et la compilation indiquera le débordement:

```
main.adb:5:11: warning: value not in range of type "Standard.Integer" [enabled
by default]
main.adb:5:11: warning: Constraint_Error will be raised at run time [enabled
by default]
```

Ces vérifications sont également faites sur les sous-types définis par l'utilisateur.

Cela ne signifie pas qu'il ne peut pas y avoir de débordement car les contrôles sont faits à des points particuliers du programme et un débordement peut se produire dans un calcul intermédiaire entre deux points de contrôle. Toutefois, l'hygiène du langage permet de réduire significativement les erreurs de calculs.

Pour un contrôle plus poussé des erreurs de calculs, il est nécessaire de passer par SPARK qui permet de garantir statiquement des propriétés sur les calculs effectués.

Enfin, il existe aussi des implémentations Ada pour MPFR et GMP afin de réaliser des calculs dynamiques.

4.3.6. Assurances

Par sa conception même, le langage Ada est conçu pour offir un haut niveau de garanties et une grande part des erreurs de programmation peuvent être évitées en utilisant les mécanismes intrinsèques du langage.

Naturellement, il est toujours possible d'avoir des erreurs mais l'espace dans lequel celles-ci peuvent vivre est plus étroit que le fossé permis par le C. Par ailleurs, cet espace est adressé par les analyses statiques disponibles.

4.3.7. Utilisation dans le critique

Ada ayant été conçu pour les systèmes critique, il est naturellement utilisé dans ce domaine. D'abord présent dans un cadre militaire, Ada est utilisé, autre autres[19], dans les domaines du spatial, de l'aéronautique et du ferroviaire :

- dans les lanceurs Ariane 4, 5 et 6;
- le système de transmission voie-machine (TVM) développé par le groupe CSEE et utilisé sur les lignes ferroviaires TGV, le tunnel sous la manche, la High Speed 1 au Royaume-Uni ou encore la LGV 1 en Belgique.
- Le pilote automatique pour la ligne 14 du métro parisien dans le cadre du projet METEOR (Métro Est Ouest Rapide) [20].
- La majorité des logiciels embarqués du Boeing 777 sont écrits en Ada.

5. SCADE

5.1. Description

SCADE (*Safety Critical Application Development Environment*) est un langage de programmation et un environnement de développement dédiés à l'embarqué critique. Le langage est né au milieu des années 90 d'une collaboration entre le laboratoire de recherche VERIMAG à Grenoble et l'éditeur de logiciels VERILOG. Depuis 2000, le langage est développé par l'entreprise ANSYS/Esterel-Technologies.

À l'origine le langage SCADE était essentiellement une extension du langage Lustre v3 avant d'en diverger à partir de la version 6.

Grâce à l'expressivité réduite du langage, le compilateur KCG de SCADE est capable de vérifier des propriétés plus fortes que le compilateur d'un langage généraliste.

5.1.1. Paradigme

SCADE est un langage *data flow* déclaratif et synchrone.

Contrairement à la plupart des langages généralistes dont la brique élémentaire de donnée est l'entier, SCADE manipule des *séquences* (ou signaux) potentiellement infinies indexées par un temps discret. Ces séquences sont une modélisation des entrées analogiques.

Un programme SCADE est constitué de noeuds (`node` dans le langage) et chaque noeud définit un système d'équations entre les entrées et les sorties du noeud. Par exemple, considérons le programme suivant en *Lustre*:

```
1 node mod_count (m : int) returns (out : int);
2 let
3   out = (0 -> (pre out + 1)) mod m;
4 tel
```

où `0 -> (pre out + 1)` indique qu'à l'instant 0, le signal vaut 0 mais que pour les instants suivants, le signal vaudra `pre out + 1`. `pre` est un opérateur qui récupère la valeur du paramètre à l'instant précédent.

Par exemple si m est la séquence constante 4, on obtient :

Temps	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
m	4	4	4	4	4	4	4	4
`pre out`		0	1	2	3	0	1	2
`pre out + 1`		1	2	3	4	1	2	3
`0 -> (pre out + 1)`	0	1	2	3	4	1	2	3
`(0 -> (pre out + 1)) mod m`	0	1	2	3	0	1	2	3
`out`	0	1	2	3	0	1	2	3

5.1.2. Mécanismes intrinsèques de protection

SCADE est statiquement et fortement typé; ce qui assure déjà une certaine fiabilité. En plus du typage, la suite réalise d'autres analyses qui vérifient :

- que le programme ne contient pas d'accès à un tableau en dehors de ses bornes;
- qu'il n'y a pas de boucle causale : l'équation `a = pre a` n'est pas possible, il faut introduire un délai (via `->`) ;
- le programme peut être exécuté avec une quantité de mémoire bornée et connue,
- le programme est déterministe au sens où sa sortie ne dépend pas d'un ordonnancement des tâches du système hôte (*i.e.* il n'y a pas de *threads*).

5.1.3. Compilateurs

SCADE utilise le compilateur KCG qui est techniquement un transpileur, c'est-à-dire un traducteur d'un langage de programmation (ici SCADE) vers un langage de même niveau (ici le C ou Ada).

À notre connaissance, KCG est l'unique compilateur disponible pour le langage SCADE. Il est disponible sur Windows en 32 et 64 bits. Il est par ailleurs certifié pour les normes suivantes:

- DO-178B DAL A,
- DO-178C DAL A,
- DO-330,
- IEC 61508 jusqu'à SIL 3,
- EN 50128 jusqu'à SIL 3/4,
- ISO 26262 jusqu'à ASIL D.

5.1.4. Interfaçage

SCADE permet d'importer du code (C ou Ada) via le mot clé `imported` comme par exemple:

```
1 function imported f (i1: int32; i2: bool) returns (o1: bool; o2: int32);
```

mais il faut faire attention à conserver les invariants requis. Dans l'exemple précécent, il faut s'assurer que les sorties ne dépendent que des entrées sous peine de probablement casser les propriétés de sûreté du programme.

Le code engendré par SCADE peut être utilisé dans un programme C ou Ada.

5.1.5. Adhérence au système

Un programme SCADE se compile vers du C (ou Ada) et peut donc fonctionner du matériel nu.

5.1.6. Gestionnaire de paquets

Aucun gestionnaire de paquet n'est indiqué pour SCADE.

5.1.7. Communauté

La communauté SCADE semble restreinte aux utilisateurs de la suite, notamment les grands comptes de l'avionique, et à la communauté de chercheurs en lien avec le langage Lustre.

5.2. Outillage

5.2.1. Débugueurs

Le débug d'un programme synchrone est un peu particulier car on s'intéresse pas ou très peu au contrôle de flôt étant donné que celui-ci est très limité dans un

langage *data flow*. En revanche, ces flux eux-mêmes sont beaucoup plus intéressants et déboguer un programme *data flow* consiste simplement à :

- afficher les flux (leur valeur aux cours du temps);
- pouvoir jouer avec les entrées pour voir comment les sorties réagissent;
- éventuellement ajouter des noeuds de contrôle (des *watch dogs*) permettant de contrôler des propriétés au cours du temps.

La suite SCADE permet tout cela en plus de visualiser graphiquement les noeuds et leur dépendances.

5.2.2. Tests

Les langages *data flow* se prettent facilement aux tests puisque qu'il est facile de créer des noeuds qui testent d'autres noeuds ou de faire varier les entrées pour vérifier les sorties. Ce travail est tout aussi facilement automatisable à l'aide des indications de type. Aussi, la suite SCADE supporte la génération automatique de tests sur plusieurs aspects (unitaires, intégration, couverture) et propose également un cadre de gestion des tests et la génération de rapport à l'intention des processus de certification/qualification.

5.2.3. Parsing

Le langage n'est pas adapté à l'analyse syntaxique car celle-ci suppose un traitement dynamique avec une consommation mémoire arbitraire; ce qui va à l'encontre des limites imposées par le langage. On peut tout à fait écrire des parser à mémoire bornée en SCADE pour analyser des flux syntaxiques particuliers (comme des flux protocolaires) mais, à notre connaissance, il n'existe pas de générateur de parseur pour SCADE.

5.2.4. Meta programmation

Le langage n'offre pas de support pour la métaprogrammation.

5.2.5. Dérivation

Le langage n'offre pas de support pour la dérivation de programme.

5.3. Analyse & fiabilité

5.3.1. Analyse statique

Il ne semble pas y avoir d'analyseur statique autre que ceux présents dans la suite SCADE. Celle-ci fait déjà un certain nombre d'analyse permettant d'éviter les débordements de tampons ou les boucles causales. Par ailleurs, il n'y a pas de manipulation de pointeurs donc les erreurs dessus ne sont pas non plus possibles.

Toutefois, le code généré peut être passé aux analyseurs C & Ada pour éventuellement trouver les erreurs qui seraient passées au travers de l'analyse de la suite logicielle.

5.3.2. Meta formalisation

Le langage Lustre a fait l'objet d'une formalisation en Coq[21] mais, à notre connaissance, il n'existe pas d'outil de formalisation pour SCADE. Cependant, il est possible de formaliser des propriétés sur les programmes SCADE en utilisant des noeuds de vérification pour faire du *model checking*. Si ces noeuds sont toujours vrais pour toutes les entrées possibles, alors le programme est formellement valide *a posteriori*.

5.3.3. WCET

L'estimation du WCET d'un programme SCADE est cruciale pour s'assurer qu'il pourra s'exécuter en temps réel: on doit garantir que chaque pas d'exécution avant le *quantum* de temps autorisé.

La suite SCADE intègre l'outil aiT[22]. Cet outil permet l'estimation et l'optimisation du WCET lors de la modélisation[23].

5.3.4. Pile

La suite SCADE intègre l'outil StackAnalyzer[24], [23]. Il est aussi possible d'utiliser les outils travaillant directement sur le binaire produit par la compilation du code C ou Ada engendré par SCADE.

5.3.5. Qualité numérique

Il ne semble pas y avoir d'analyse statique pour les erreurs numériques.

5.3.6. Assurances

La suite SCADE offre un très bon niveau de fiabilité du à la simplicité du langage *data flow* sous-jacent et des analyses effectuées par la suite. Par ailleurs, le compilateur KCG est certifié pour les normes du domaine critique.

5.3.7. Utilisation dans le critique

Du fait des certifications proposées par le compilateur *KCG*, le langage SCADE est beaucoup utilisé dans l'aviation civile et militaire. On peut citer notamment:

- les commandes de vol des Airbus,
- le projet openETCS visant à unifier des systèmes de signalisation ferroviaires en Europe,
- dans l'automobile.

6. OCaml

6.1. Description

Le langage OCaml est issu de la famille des langage ML via différentes implémentations (Caml, Caml light) et première version officielle en 1996. Le langage connait une popularité croissante dans les milieux académiques car il est particulièrement adapté à l'enseignement de l'algorithmique et de la compilation.

Le langage se prette tellement bien à l'exercice qu'il est souvent utilisé, y compris dans les sociétés privées, pour développer des outils d'analyse ou des compilateurs pour d'autres langages. Pour des applications en production, le langage reste toutefois peu utilisé et son utilisation la plus notoire à l'heure actuelle est celle de Jane Street pour réaliser du *trading* haute fréquence.

Comme les garanties du langage suffisent généralement à son usage, il n'y a pas eu de développements majeurs sur des outils d'analyse statique dessus : le langage étant principalement utilisé pour écrire ces analyseurs pour d'autres langages.

6.1.1. Paradigme

OCaml est un langage de programmation multiparadigme. Il est conçu comme étant avant tout un langage fonctionnel mais il est dit *impur* dans le sens où la mutabilité est autorisée. Cette dernière et les boucles (`while`, `for`) en font tout aussi bien un langage impératif dans lequel on peut très bien programmer de manière procédurale.

Le O de OCaml signifie *Objective* et fait référence au fait que le langage est également objet. Ce trait est toutefois relativement peu usité en pratique.

6.1.2. Mécanismes intrinsèques de protection

Les deux principaux atouts d'OCaml est son système de type riche avec inférence et son ramasse-miettes. Les deux permettent de se prémunir contre un large éventail d'erreurs de programmation sans perdre en expressivité.

Son système de type permet notamment de définir des types algébriques (ou variants) qui permettent de définir un type par une somme de valeurs possibles, elles-mêmes pouvant être valuées. Cela permet d'encoder le *pattern* des unions à discriminant du C de manière synthétique mais également avec des vérifications beaucoup plus poussées par le compilateur.

Typiquement, l'exemple suivant:

```
1 type state = A | B of int | C of string list
2
3 let f s =
4   match s with
5   | A -> 0
6   | B i -> i
7   | C (h :: t) -> String.length h + List.length t
```

ne sera pas accepté par le compilateur qui signalera une erreur de *pattern matching* incomplet. En effet, le cas où la liste donnée en argument au constructeur `C` est vide n'est pas traité.

Cette programmation par cas est extrèmement courante dans la programmation en général et pouvoir identifier les cas non traités facilement à la compilation est une des forces du langage. Ce mécanisme est hérité de la famille des langages ML et a été reprit dans les langages plus récents (Swift, Rust, ...) pour les mêmes raisons.

Le langage n'expose à l'utilisateur qu'une notion de *valeur* qui n'autorise pas la manipulation manuelle des pointeurs. Ceux-ci sont gérés automatiquement à la compilation et par le ramasse miettes. Cela permet de se prémunir contre toutes les erreurs de gestion de la mémoire.

Les valeurs sont alignées systématiquement sur la taille d'un mot mémoire et les valeurs numériques sont encodées sur 31 ou 63 bits suivant les architectures (respectivement 32 ou 64 bits). Le bit manquant sert au ramasse miettes. Cette représentation uniforme diminue les occurences de dépassement de capacité puisqu'on ne peut pas manipuler des entiers plus petits qui seraient mal dimensionnés mais l'erreur est toujours possible.

Le ramasse-miettes, quant à lui, permet de se prémunir contre les fuites mémoires et les erreurs de gestion de la mémoire. Il est particulièrement efficace pour les programmes OCaml car il utilise un système a double tas : un mineur et un majeur. Les valeurs sont allouées dans le tas mineur et sont déplacées dans le tas majeur si leur durée de vie dépasse un certain seuil. Comme la plupart des valeurs allouées ont une durée de vie très faible, elles ne vont généralement pas plus loin et comme les allocations dans le tas mineur sont très efficaces (cela revient à globalement incrémenter un compteur), celui-ci se comporte à peu près comme la pile et bénéficie des effets de cache.

Notons que le langage a fait l'objet d'études de sécurité[25] qui sont des guides de bonnes pratiques pour écrire du code sûr en OCaml. Ces guides sont à l'origine des guides orientés vers la sécurité mais ils sont valides pour la sûreté.

6.1.3. Compilateurs

Le compilateur fourni par l'INRIA est le seul compilateur officiel du langage OCaml.

Il existe des *forks* maintenus en interne par certains industriels mais ils ne font pas référence.

6.1.4. Interfaçage

OCaml a plusieurs moyens de s'interfacer avec le C (et donc avec tous les langages compatibles avec le C):

- la FFI pour le *link* statique;
- `Ctypes` pour le *link* dynamique.

La FFI d'OCaml est bien documentée et, modulo quelques subtilités sur la bonne entente avec le ramasse-miettes et les *threads*, s'utilise raisonnablement facilement pour intégrer du code C dans OCaml et l'export d'une fonction OCaml vers C se fait sur une simple déclaration. Il existe par ailleurs un outil camlidl pour engendrer le code de conversion automatiquement à partir de fichiers `.h` annotés.

`Ctypes` est une biblothèque OCaml qui permet d'encoder les types C dans OCaml et d'engendrer automatiquement les bons appels à partir des types déclarés. Bien

qu'utiliser `Ctypes` ne soit pas compliqué, il existe des outils pour dériver automatiquement les déclarations soit par PPX soit intégré dans le système de *build* `dune`.

6.1.5. Adhérence au système

OCaml a besoin d'un système POSIX pour fonctionner en natif mais en *bytecode*, il suffit de porter la machine virtuelle OCaml (écrite en C) pour qu'il fonctionne sur n'importe quelle plateforme.

6.1.6. Gestionnaire de paquets

Plusieurs tentatives de gestionnaires de paquet ont vu le jour mais celle qui s'est imposé de fait et qui est devenu la référence non seulement pour OCaml mais aussi pour Coq est opam. Celui-ci est stable et contient environ 4800 paquets logiciels. Il permet de créer des environnements de *build* propres (des *switchs*) et de contrôler (ou pas) les versions de manière assez fine pour de la production industrielle.

6.1.7. Communauté

La taille de la communauté OCaml n'est pas connue. Elle est manifestement plus petite que la plupart des langages *mainstream* mais la diffusion académique semble avoir engendré une diaspora solide de pratiquants et un noyau dur d'entreprises utilisant OCaml. Parmi celles-ci on distingue les entreprises dont OCaml est le coeur technologique (Jane Street, OCamlPro, Tarides, Lexifi, ...) et celles qui l'utilisent dans leur équipes de R&D pour construire de l'outillage (Microsoft, Meta, Intel, ...).

6.2. Outillage

6.2.1. Débugueurs

OCaml est fourni avec un debuggeur par défaut : `ocamldebug` qui se comporte peu ou prou comme GDB pour un programme OCaml mais seulement avec le programme compilé en *bytecode*.

Techniquement, GDB et les autres débugueurs pour le C peuvent être utilisé sur les programmes OCaml compilés en natif mais ce n'est pas forcément pratique car il faut composer avec le nommage d'OCaml dans le code engendré qui suit le schéma `caml<MODULE>.<FONCTION>_<NNN>` où `NNN` est un entier calculé au hasard à la compilation. Par ailleurs, l'examen des valeurs OCaml est également pénible car les débugueurs classiques ne savent pas comment les interpréter donc il y a là aussi une petite gymnastique à effectuer.

6.2.2. Tests

Il existe globalement trois façons de tester un programme OCaml : par des cadres logiciels de test à la manière de ce qui existe en C/C++, par des préprocesseurs et par l'outil `dune`.

Les cadres logiciels indiqués dans la Table 18 se présentent comme des bibliothèques à utiliser pour définir un programme de test. Alcotest fut l'un des premiers à être populaire; il offre une sortie lisible mais non standard et convient pour des petits projets. OUnit est une adapation pour OCaml de JUnit initialement développé pour Java et qui a été adapté pour plusieurs autres langages. L'avantage de cette bibliothèque est qu'elle engendre des sorties au format JUnit permettant à des outils tiers de travailler sur les résultats de test.

QCheck donne également des primitives de test à la manière des bibliothèques précédentes mais incorpore également un générateur de valeur pour les types de données utilisés, ce qui permet de faire du test automatisé par *fuzzing*. Par ailleurs, QCheck est compatible avec OUnit pour éventuellement obtenir des rapports au format JUnit.

Crowbar utilise également le *fuzzing* via l'outil `afl-fuzz` mais il nécessite d'utiliser un compilateur OCaml compilé avec le support de l'intrumentation (les variantes `+afl` dans `opam`).

Outil	Tests	Generation	Gestion	*mocking*
Alcotest	U	+		
OUnit	U	+	+	
QCheck	U	+++		
Crowbar	U	+++		

Table 18: Cadres de test OCaml

OCaml fait partie des langages ayant une phase de *preprocessing* éditable via des PPX. Ces PPX permettent de transformer l'AST du programme source pour y ajouter automatiquement des bouts de programmes supplémentaires *a priori* déduits de l'AST précédent. Il existe notamment deux PPX qui sont souvent utilisés : ppx_inline_test et ppx_expect. Le premier permet de créer des tests à la volée et le second permet de comparer des sorties :

```
1 let#test _ = 3 <> 4
2
3 let#expect "addition" =
4   Printf.printf "%d" (1 + 2);
5   [%expect {| 3 |}]
```

Dans le premier cas, le test se fait directement alors que dans le second, le test effectue un calcul et l'affiche sur la sortie standard qui est capturée et comparée avec le contenu de l'extension `[%expect ...]`.

Enfin, les Cram tests de Python sont également disponibles dans `OCaml` via l'outil `dune` qui peut interpréter directement un fichier au format Cram pour effectuer les tests:

```
1 Ceci est un test
2   $ echo "Hello world!"
3   Hello world!
```

Ce format est plus adaptés aux tests de plus haut niveau (fonctionnels, validation) où l'on va tester le comportement du binaire produit par rapport à une sortie attendue (les lignes indentées sans `$`).

6.2.3. Parsing

OCaml étant historiquement utilisé pour écrire des compilateurs, il dispose d'outils d'analyse lexicale et syntaxique complets. Un générateur de *lexer* (`ocamllex`) et un générateur de *parser* (`ocamlyacc`) sont même fournis par défaut avec OCaml. Ceux-ci fonctionnent très bien mais sont un peu limités et d'autres outils plus complets sont venus compléter l'offre.

Au niveau des *lexers*, on citera sedlex et ulex qui permettent l'analyse de l'unicode. Par ailleurs, sedlex s'intègre facilement avec les *parsers* existant.

Nom	Code	Plateforme	License
ocamllex	Lex	POSIX	Libre, LGPL
Dypgen	EBNF	POSIX	Libre, CeCILL-B
sedlex	OCaml	POSIX	Libre, MIT
ulex	OCaml	POSIX	Libre, MIT

Table 19: *Lexers* OCaml

Au niveau des *parsers*, en plus d'`ocamlyacc`, il y a des bibliothèques de combinateurs de *parser* avec angstrom et mparser qui sont utiles dans le *parsing* dynamique de petites grammaires. Pour les grammaires plus conséquentes, Dypgen ou Menhir sont des outils plus adaptés. Menhir est probablement le générateur de *parser* le plus abouti et est la référence actuelle dans le monde OCaml. Celui-ci permet en plus d'engendrer la preuve de correction (en Coq) du *parser* engendré.

Nom	Code	Plateforme	License
ocamlyacc	Yacc	POSIX	Libre, LGPL
angstrom	OCaml	POSIX	Libre, BSD
Dypgen	EBNF	POSIX	Libre, CeCILL-B
Menhir	Yacc	POSIX	Libre, GPL
mparser	OCaml	POSIX	Libre, LGPL

Table 20: *Parsers* OCaml

Il existe également un autre outil historiquement utilisé pour préprocesser OCaml mais qui peut servir à définir des langages à part entière : Camlp5.

6.2.4. Meta programmation

La métaprogrammation en OCaml passe essentiellement par le système des PPX qui permettent d'engendrer du code spécifique à l'aide des extensions (`[%... ...]`). Les PPX peuvent offrir des fonctionnalités très utiles en production comme de outils d'inspection pour la plupart des types de données, la possibilité d'écrire du SQL (typé) dans le code OCaml ou encore d'extraire des métadonnées pour la documentation.

Le typage OCaml est également suffisamment expressif pour décrire des propriétés statiques sur le code permettant une forme limitée de programmation par spécialisation. Ainsi, il est possible d'encoder les entiers naturels dans le système de type et donc de faire du code spécificique pour des listes de longueur 4.

6.2.5. Dérivation

Comme pour la métaprogrammation, la dérivation se fait en utilisant les PPX et les annotations. L'usage classique (mais non limité à cela) consiste à annoter une définition de type pour indiquer au PPX une dérivation à effectuer. Typiquement :

```
1 type foo = {
2   bar : int;
3   baz : string list;
4 }[@@deriving yojson]
```

engendrera automatiquement les fonctions de conversion `foo_to_yojson` et `foo_of_yojson` pour traduire une valeur de type `foo` en une valeur JSON et reciproquement.

6.3. Analyse & fiabilité

6.3.1. Analyse statique

Il n'y a pas d'outils tiers pour analyser statiquement le code OCaml et se prémunir contre les erreurs à *runtime*. Cela s'explique par le fait que le langage est justement conçu pour les éviter dans une large mesure au travers un système de typage très puissant et un ramasse-miettes à l'état de l'art.

Il y a eu des initiatives pour ajouter plus d'analyse statique, notamment sur les exceptions, mais aucune n'a donné d'outil mature indépendant ou intégré au compilateur.

6.3.2. Meta formalisation

Le moyen le plus simple de faire de la programmation formelle avec OCaml est d'utiliser Coq. Ce dernier, implémenté en OCaml, permet d'implémenter et prouver des programmes qui peuvent être ensuite extraits vers OCaml. C'est la stratégie utilisée pour implémenter le compilateur CompCert.

Notons qu'il existe une initiative pour permettre de prouver des programmes OCaml dans le texte en utilisant la même stratégie que Frama-C: Cameleer. Cependant, le projet est encore en cours de développement et n'a pas encore atteint un niveau de maturité suffisant pour être utilisé en production.

6.3.3. WCET

Il n'y a pas d'analyseur statique de WCET ciblant spécifiquement OCaml. Les outils utilisés pour le C peuvent probablement être utilisés sur les programmes OCaml compilé nativement mais il n'est pas garanti que les résultats soient pertinents.

Il est aussi possible de ne pas utiliser le code natif et de faire l'analyse sur le *byte code* OCaml[26] en simplifiant le langage pour le ramener à du Lustre à la manière de SCADE.

6.3.4. Pile

Il n'y a pas d'analyseur statique de pile ciblant spécifiquement OCaml. Les outils utilisés pour le C peuvent cependant être utilisés sur les programmes OCaml compilés nativement.

6.3.5. Qualité numérique

Il n'y a pas d'analyseur statique pour les calculs numériques des programmes OCaml. Toutefois, les bilbiothèques MPFR et GMP ont été portées en OCaml.

6.3.6. Assurances

L'utilisation d'OCaml permet de se prémunir naturellement contre un certains nombre d'erreurs à *runtime* : tous ce qui découle des erreurs de typage et de la gestion des pointeurs. Dans un développement standard, ces erreurs étant les plus coûteuses, un développement OCaml assure un gain en fiabilité et en coût par son usage même.

La possibilité d'utiliser Coq au dessus OCaml permet de gagner encore plus en fiabilité et d'adresser des normes très exigentes comme les Critères Communs aux niveaux d'assurance les plus élevés (EAL6+).

Pour des niveaux d'assurances élevés en sûreté, il manque quelques analyses:
- le WCET statique, quitte à restreindre le langage;
- l'échappement des exceptions;
- les allocations bornées pour majorer le temps utilisé par le ramasse miettes;
- récursion bornée.

Ces analyses, couplées avec un outil comme Cameleer permettraient de circonvenir aux besoins d'une utilisation dans le domaine critique.

6.3.7. Utilisation dans le critique

L'utilisation de MirageOS, un unikernel écrit en OCaml est pressenti comme étant une technologie utilisable dans le spatial mais aucune utilisation avérée n'a été recensée jusqu'à présent.

Toutefois, si le langage n'est pas utilisé directement, il est utilisé au travers des outils développés pour la sûreté : KCG[27], Astree et CompCert.

7. Rust

7.1. Description

Rust est un langage de programmation créé par Graydon Hoare en 2006, soutenu par la fondation Mozilla à partir de 2009, et dont la première version stable fût publiée en 2015. Le projet est devenu indépendant de Mozilla et est soutenu par sa propre entité légale: la fondation Rust. Conçu comme un successeur aux langages C et C++, Rust cherche à concillier sûreté, performance et correction du code concurent. Il est souvent utilisé pour des applications systèmes, des logiciels embarqués, des navigateurs et des applications web.

Il n'existe pas à l'heure actuelle de standard du langage, mais deux projets dans ce sens existent :

1. Les entreprises AdaCore et Ferrous Systems ont développé une version qualifiée du compilateur standard Rust, nommée Ferrocene, certifiée ISO26262 (ASIL D) et IEC 61508 (SIL 4) pour les plateformes x86 et ARM. Celle-ci est mise à jour tous les trois mois, soit toutes les deux à trois versions de Rust.
2. La fondation Rust a lancé un projet de rédaction d'une spécification. Ce projet semble sérieux et devrait aboutir a une spécification qui serait mise à jour à chaque parution d'une nouvelle version du compilateur (toutes les six semaines). Contrairement à la specification de Ferrocene, qui n'est pas un standard du langage mais se borne à décrire le fonctionnement d'un compilateur donné, la specification proposée par la fondation serait officielle et pourrait à terme aboutir à un standard.

Il existe de plus un guide [28] édité par l'ANSSI pour le développement d'applications sécurisées en Rust.

7.1.1. Paradigme

Rust est un langage multi-paradigme qui permet la programmation impérative, fonctionnelle, et (de manère plus limitée) orientée objet.

Pour illuster l'aspect multi paradigmes, considérons l'exemple d'un code qui prend en entrée une liste de chaînes de caractères représentant des entiers et compte combien sont plus petits que 10.

Dans un style fonctionnel:

```rust
fn count_smaller_than_10(l: &[&str]) -> usize {
    l.iter()
        .filter_map(|s| s.parse::<i32>().ok())
        .filter(|&x| x < 10)
        .count()
}
```

et l'équivalent dans un style impératif:

```rust
fn count_smaller_than_10(l: &[&str]) -> usize {
    let mut count = 0;
    for s in l {
        if let Ok(n) = s.parse::<i32>() {
            if n < 10 {
                count += 1;
            }
        }
    }
    count
}
```

7.1.2. Mécanismes intrinsèques de protection

Erreurs

En Rust, les erreurs sont rendues explicites dans les types de retour, et le compilateur signale lorsque l'on a omis de vérifier une potentielle valeur d'erreur. Par exemple, pour cette fonction qui vérifie si un fichier fait moins de 100 kilo-octets, le type de retour `Result<bool, std::io::Error>` indique qu'il s'agit d'un booléen dans le cas normal ou potentiellement d'une erreur d'IO.

```rust
fn file_is_less_than_100kb(path: impl AsRef<Path>) -> Result<bool,
std::io::Error> {
    let path = path.as_ref();
    let res = path.exists() &&
              path.is_file() &&
              std::fs::metadata(path)?.len() < 1024 * 100;
    Ok(res)
}
```

On ne peut pas accéder à la valeur booléenne sans vérifier l'erreur: le code ci-dessous donne une erreur de compilation.

```rust
fn main() {
  if file_is_less_than_100kb("user.txt") {
    println!("user.txt is missing")
  }
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:12:6
  |
12 |   if file_is_less_than_100kb("user.txt") {
  |      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected `bool`, found
`Result<bool, Error>`
  |
  = note: expected type `bool`
              found enum `Result<bool, std::io::Error>`
help: consider using `Result::expect` to unwrap the `Result<bool,
std::io::Error>` value, panicking if the value is a `Result::Err`
  |
12 |   if file_is_less_than_100kb("user.txt").expect("REASON") {
  |                                          ++++++++++++++++++

For more information about this error, try `rustc --explain E0308`.
```

Il en est de même si on oublie de vérifier qu'une fonction à effet de bord n'a pas retourné d'erreur:

```
 1 fn write(path: impl AsRef<Path>, content: &str) -> Result<(), std::io::Error>
{
 2     std::fs::write(path, content)
 3 }
 4
 5 fn main() {
 6     write("user.txt", "John Smith");
 7 }
```

on obtient alors le warning:

```
 1 warning: unused `Result` that must be used
 2 --> src/main.rs:8:5
 3   |
 4 8 |     write("user.txt", "John Smith");
 5   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 6   |
 7   = note: this `Result` may be an `Err` variant, which should be handled
 8   = note: `#[warn(unused_must_use)]` on by default
 9 help: use `let _ = ...` to ignore the resulting value
10   |
11 8 |     let _ = write("user.txt", "John Smith");
12   |     +++++++
```

Régions mémoire

Rust dispose d'un mécanisme innovant de gestion de la mémoire par régions qui permet de garantir la sûreté mémoire au moment de la compilation sans avoir recours à un rammasse-miettes au *runtime*. Celui-ci, lié au système d'*ownership* permet d'éviter les erreurs suivantes:

- *use after free*
- *data races*
- *dangling pointers*
- erreurs liés à l'*aliasing* de pointeurs

7.1.3. Compilateurs

Hormis le compilateur Ferrocene mentionné précedemment qui n'est autre qu'une version qualifiée du compilateur standard `rustc`, il n'existe aujourd'hui pas de compilateur alternatif utilisable.

On peut néanmoins mentionner les projets suivants:

- **gccrs** vise à développer un *frontend* Rust pour GCC. L'ensemble formerait un compilateur entièrement différent de `rustc`.
- **rust codegen gcc** qui consiste à brancher le *backend* de GCC au *frontend* de `rustc`. Cela permettrait de supporter les plateformes déjà supportées par GCC mais ne constituerait pas une implémentation différente de `rustc`.
- **rust codegen cranelift** qui consiste à brancher *cranelift* sur le *frontend* de `rustc` afin de permettre des temps de compilation plus rapides en mode dit *debug* (c'est à dire sans les optimisations). Là encore, il s'agirait d'utiliser une partie de `rustc`.

7.1.4. Interfaçage

Rust a une FFI lui permettant d'interagir avec le C soit en important du C :

```rust
#[link(name = "my_c_library")]
extern "C" {
    fn my_c_function(x: i32) -> bool;
}
```

soit en exportant du Rust vers du C:

```rust
#[no_mangle]
pub extern "C" fn callable_from_c(x: i32) -> bool {
    x % 3 == 0
}
```

Par transitivité, Rust est compatible avec tous les langages compatibles avec le C.

7.1.5. Adhérence au système

Comme le C, Rust peut être utilisé sur un système nu en spécifiant `no_std` :

```rust
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

7.1.6. Gestionnaire de paquets

Le gestionnaire de paquet officiel de Rust est `cargo`. C'est également l'outil de *build* du langage. Il permer de télécharger, compiler, distribuer et de téléverser des paquets (nommés *crates*) dans des registres partagés ou de dépôts Git. Les registres peuvent être publics ou privés. Le registre public par défaut est `crates.io` mais il y a également `lib.rs` qui est également très utilisé.

Ce gestionnaire de paquets est l'un des points forts de Rust car il facile l'installation d'un environnement de développement propre et le *build* d'un projet Rust.

7.1.7. Communauté

La communauté Rust comprend un tissu associatif (fondation Rust, Rust France, ...) et des entreprises qui organisent régulièrement des événements (meetups, conférences, ...) au travers le monde.

Le langage attire beaucoup de jeunes développeurs séduits par la fiabilité et les performances mis en avant par le langage. Il suffit de voir le nombre de paquets logiciels crées pour Rust qui atteint plus de 130 0000 sur `crates.io` en moins de 10 ans[13].

7.2. Outillage

7.2.1. Débugueurs

Le langage Rust supporte officiellement GDB, LLDB et WinDbg/CDB.

[13] https://lib.rs/stats

7.2.2. Tests

Rust inclut un cadre de test standard qui permet de gérer les tests unitaires via des annotations et des macros:

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Avec ces annotations, le compilateur Rust génère un exécutable de test qui peut être lancé avec la commande `cargo test`.

Toutefois, les fonctionnalités fournies par ce cadre sont minimalistes et des paquets Rust complètent le service en y ajoutant du *fuzzing* (`quickcheck`, `proptest`) ou du *mocking* (`mockall`).

Il ne semble pas exister d'outil permettant d'engendrer des rapports de tests standardisés ou conforme à une norme particulière. `cargo test` permet toutefois d'engendrer un rapport JSON qui peut être traité par des outils tiers.

7.2.3. Parsing

La jeunesse du langage fait qu'il n'est pas encore très utilisé pour l'écriture de compilateurs ou d'interpréteurs. De fait, il n'y a pas encore beaucoup d'outils pour l'écriture de parseurs spécifiquement pour Rust. Quelques outils ciblant plusieurs langages ont simplement développé un *backend* supplémentaire pour Rust.

Au niveau des *lexers*, il n'y a que Re2c supporte Rust. LALRPOP a également un générateur de *lexer* intégré. Pour les *parsers*, il y en a essentiellement trois: LALRPOP, Hime et Syntax. Notons que LALRPOP embarque le *parsing* dans Rust via des macros procédurales.

Nom	Algorithme	Grammaire	Code	Plateforme
LALRPOP	LR(1), LALR(1)	Rust	Rust	Toutes
Hime	LR(1), GLR, LALR(1)	EBNF	Séparé	.NET, JVM
Syntax	LR, LL	JSON, Yacc	Mixte	Toutes

7.2.4. Meta programmation

Le système de macro de Rust est expressement fait pour permettre la métaprogrammation de manière hygiénique. Par exemple, le code suivant permet de créer un vecteur de trois entiers:

```rust
let v: Vec<i32> = vec![1, 2, 3];
```

où le symbole `!` indique l'appel à une macro (ici `vec`). Cette macro pourrait être définie comme suit:

```
macro_rules! vec {
    ($($x:expr),*) => {{
        let mut temp_vec = Vec::new();
        $(temp_vec.push($x);)*
        temp_vec
    }};
}
```

où, sans entrer dans les détails, on dit au compilateur que si la macro est appelée avec une liste d'expressions séparées par des virgules (`$($x:expr),*`), alors il crée un nouveau vecteur (`let mut temp_vec = Vec::new();`), y ajoute les éléments de la liste (`$(temp_vec.push($x);)*`) et retourne le vecteur.

Ce système de macro est suffisamment riche pour permettre l'écriture de mini-DSL, des générateurs de code ou des dérivations à partir des types.

7.2.5. Dérivation

La métaprogrammation autorisée par les macros de Rust permet également de dériver du code à partir des types. Le code suivant définit ce qu'on appelle une macro procédurale qui va générer du code Rust à partir de code Rust.

```
use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
  // Construct a representation of Rust code as a syntax tree
  // that we can manipulate
  let ast = syn::parse(input).unwrap();

  // Build the trait implementation
  let name = &ast.ident;
  let gen = quote! {
      impl HelloMacro for #name {
          fn hello_macro() {
              println!("Hello, Macro! My name is {}!", stringify!(#name));
          }
      }
  };
  gen.into()
}
```

Dans cet exemple, on définit une bibliothèque de dérivation qui prend un code Rust (`input: TokenStream`), le parse (`let ast = syn::parse(input).unwrap();`) en un bout d'AST puis contruit un nouveau code Rust (`let gen = quote! { ... };`) avec les informations contenues dans l'AST. En définissant le trait `HelloMacro` par ailleurs:

```
pub trait HelloMacro {
    fn hello_macro();
}
```

on peut alors utiliser la macro `hello_macro` pour dériver le code idoine:

```rust
#[derive(HelloMacro)]
struct Foo;


fn main() {
    Pancakes::hello_macro();
}
```

qui affichera `Hello, Macro! My name is Foo!.`

7.3. Analyse & fiabilité

7.3.1. Analyse statique

La jeunesse du langage fait que la plupart des outils d'analyse ou de vérification sont encore des projets de recherche. En terme d'analyse statique, il existe Mirai qui fait de l'analyse statique sur le langage intermédiaire du compilateur Rust. Il peut identifier certaines classes d'erreurs en s'aidant éventuellement d'annotations.

7.3.2. Meta formalisation

La vérification formelle de code Rust suscite un certain intérêt puisque malgré la jeunesse du langage, il y a déjà plusieurs initiatives sur le sujet. Toutefois, il s'agit de projets académiques et aucun ne bénéficie d'un support commercial pour le moment.

Creusot

Creusot[14] est un outil de vérification déductive de code Rust. Il a son propre langage de specification nommé *pearlite*. Il traduit le code Rust et sa specification *pearlite* en code WhyML, utilisé ensuite par l'outil Why3. A ce jour, il a été utilisé pour vérifier formellement un SAT solver, CreuSAT. Voici un exemple d'utilisation de spécification Creusot:

```rust
use creusot_contracts::*;

#[requires(x < i32::MAX)]
#[ensures(result@ == x@ + 1)]
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

et la commande

Flux

Flux[15] augmente Rust en lui rajoutant des types par rafinement. Il a également son propre langage de spécification qui ne porte pas de nom. Il transforme ces types en Clauses de Horn Contraintes (CHC) et utilise directement le solver Z3 pour les résoudre. Par exemple :

[14] https://github.com/creusot-rs/creusot
[15] "https://flux-rs.github.io/flux/

```
#[flux::sig(fn(x: i32) -> i32{v: x < v})]
pub fn inc(x: i32) -> i32 {
  x - 1
}
```

donnera une erreur lors de la compilation:

```
error[FLUX]: postcondition might not hold
--> test0.rs:3:5
  |
3 |     x - 1
  |     ^^^^^
```

Aenas

Aenas et Charon sont deux outils, développés dans le cadre du même projet. Ils permettent de traduire du code Rust vers différents langages de spécification formelle afin de pouvoir écrire des preuves dessus. Les langages de preuve supportés sont F*, Coq, HOL et Lean.

Verus

Verus permet également d'ajouter des spécifications à du code Rust, et de vérifier qu'elles sont satisfaites pour toute execution possible du programme. Il s'appuie directement sur le solveur SMT Z3. Voici un exemple d'utilisation:

```
fn octuple(x1: i8) -> i8
    requires
        -64 <= x1,
        x1 < 64,
{
    let x2 = x1 + x1;
    let x4 = x2 + x2;
    x4 + x4
}
```

7.3.3. WCET

Il n'existe à ce jour pas d'outil sur étagère permettant de faire de l'analyse WCET de programmes spécifiquement Rust. Néanmoins, les outils fonctionnant sur du code machine évoqués dans la partie C peuvent être utilisés.

7.3.4. Pile

Il n'y a pas d'outil recensé faisant de l'analyse statique de pile sur un programme Rust. Toutefois, les outils fonctionnant sur du code machine évoqués dans la partie C peuvent être utilisés.

7.3.5. Qualité numérique

Il n'y a pas d'outils recensé pour faire de l'analyse numérique statique Rust.

Il existe néanmoins plusieurs bibliothèques permettant de travailler avec plus de précision qu'avec des flottants 32 ou 64 bits. Certaines permettent

- du calcul à virgule fixe (bigdecimal, rust_decimal);
- des entiers de taille arbitraire(num, ruint rug);
- des fractions (num, rug);

- des flottants de précision arbitraire (rug ou gmp-mpfr-sys basées sur MPFR ou GMP).

7.3.6. Assurances

Le langage Rust intègre nativement des mécanismes de sécurité et de fiabilité :

- un typage statique et fort;
- un système de gestion de la mémoire et son modèle d'*ownership*.

Ces mécanismes, et tout une panoplie de contrôles à la compilation, permettent de limiter le besoin d'outils externes pour fiabiliser la production.

Cette rigueur peut amener le compilateur à rejeter des programmes qui pourraient être valides et cela peut destabiliser les nouveaux programmeurs habitués au C/C++. De fait, le langage est reconnu pour avoir un ticket d'entrée plutôt élévé et il n'est pas clair aujourd'hui si cela est un avantage ou un inconvénient d'un point de vue industriel. D'un côté, cela permet de réduire les erreurs de programmation et de limiter les failles de sécurité. D'un autre côté, cela peut ralentir le développement et engendrer des circonvonlutions pour faire accepter le programme au compilateur, engendrant ainsi une dette technique similaire à celle du C++ utilisé par des spécialistes.

Toutefois, le langage jouit d'un réel engouement et il est même conseillé par l'ANSSI dans les développements sécurisé au titre de la démarche de sécurité par conception.

7.3.7. Utilisation dans le critique

Il n'existe à ce jour pas de communication officielle sur un logiciel embarqué critique qui serait en Rust. Toutefois les demarches de Ferocene et la fondation Rust indiquent clairement une volonté de pénétrer le marché du logiciel critique et certains constructeurs automobiles ont déjà manifesté leur intérêt pour le langage.

8. Conclusion

Ce rapport fourni un aperçu des langages de programmation C, C++, Ada, SCADE, OCaml, Rust et de leur écosystème du point de vue de la sûreté logicielle.

Comme à chaque fois qu'on étudie plusieurs langages, il est difficile d'échapper à la question toute naturelle : *in fine*, quel est le langage le plus sûr ? La réponse sous-tend qu'on compare les langages et leur écosystème entre eux et, comme nous l'avons vu, tous ces langages ne sont pas vraiment comparables car il vont toucher des usages particuliers avec des techniques particulières. Comme un bon bricoleur choisi bien son outil en fonction de la tâche à réaliser, un bon développeur doit choisir le langage le plus adapté à son besoin.

Le besoin en sureté est celui de la confiance dans le logiciel. Celle-ci repose sur deux piliers:
- les vérifications statiques, que ce soit par le compilateur lui-même ou des outils tiers;
- les tests.

Ni l'un ni l'autre ne suffisent, seuls, à garantir la sûreté d'un logiciel : les analyses peuvent être mal implémentées ou reposer sur des hypothèses fausses et les tests peuvent ne pas couvrir toutes les combinaisons possibles pour des raisons combinatoires ou techniques.

Or, nous l'avons vu, tous les langages présentés disposent d'environnements de test plus ou moins automatisés mais suffisants pour réaliser des bancs de tests complets. En revanche, OCaml et Rust, malgré leur très bon degré de vérification intrinsèque, manquent d'analyses complémentaires qui ne sont, pour l'heure, qu'à l'état de projets de recherche.

Aussi, pour des projets très critiques, les langages Ada et SCADE semblent être les plus adaptés car ils offrent des garanties de vérification à la fois intrinsèques et externes très fortes.

Pour les projets un peu moins critiques, OCaml et Rust ont probablement une place à prendre en fonction des objectifs recherchés :
- Rust pour les mêmes raisons que le C/C++ mais avec des garanties supplémentaires;
- OCaml pour son efficacité productive.

Lorsque plusieurs langages sont utilisables, la gestion du *coût* intervient afin d'être compétitif. Ce coût provient :
- du temps de développement;
- du temps de débogue;
- des licences d'outils tiers;
- de la formation du personnel (pour le langage et les outils);
- de la dette technique.

La dette technique est un point important car elle est souvent sous-estimée et engendre un coût caché de production qui augmente, parfois de manière dramatique, avec le temps. La dette technique est évaluée par rapport à :
1. la complexité et maintenabilité de la base de code au fil du temps. Par exemple, les programmes C++ ont une tendance naturelle à devenir difficile à suivre;

2. la disponibilité des ressources : les ressources qui savent faire de l'Ada ou du SCADE sont plus rares que celles qui savent faire du C et coûtent naturellement plus cher sur le marché du travail.

Comme il n'existe pas à notre connaissance de métriques sur des développements comparables, il est difficile de donner une vue complètement objective de la comparaison entre les langages selon les critères ci-dessus. A titre informatif, nous pouvons cependant donner le résultat d'une étude informelle réalisée dans le cadre du développements de systèmes embarqués (mais lourds) a haut niveau de confiance avec du temps réel avec des garanties de sécurité[16]. Ces systèmes n'étaient pas soumis au normes de sûreté mais l'expérience est toutefois pertinente puisque le hasard a voulu que les langages étudiés dans ce rapport y soient utilisés (ou dans une forme équivalente). Ces développements ont été fait avec une équipe d'ingénieurs représentative de l'ingénieur moyen et le résultat est résumé dans la Table 22. Les coûts sont donnés en pire cas (avec des € croissants) : par exemple, si l'équipe d'ingénieur est déjà versée dans l'art de faire du bon C++, son coût de formation sera moindre ou nul.

	C	C++	Ada	SCADE	OCaml	Rust
Temps Réel dur	Oui	Oui	Oui	Oui	Non	Oui
Temps Réel mou	Oui	Oui	Oui	Oui	Oui	Oui
Environnement contraint	Oui	Oui	Oui	Oui	Pas directement	Oui
Temps de développement	Long	Moyen	Moyen	Long	Court	Moyen
Temps de débogage	Long	Long	Court	Court	Court	Court
Licences	€€€	€€€	€€€	€€€	?	?
Formation	€	€€€	€€€	€€	€	€€€
Dette technique	€	€€	€	?	€	€€

Table 22: Bilan comparatif des langages

Nous n'avons pas assez de recul pour évaluer la dette technique liée à SCADE ni le prix des licences sur l'éventuelle commercialisation des outils actuellement à l'état de projet de recherche pour OCaml et Rust. Toutefois, nous pensons que ce bilan et le rapport dans sa globalité donneront une idée générale assez fidèle à l'état actuel des langages étudiés.

[16] Le système d'exploitation PolyXene (https://cyber.gouv.fr/produits-certifies/polyxene-version-11) et ses variantes.

Références

[1] Danko ILIK, “Clauses Techniques - Action « COTS de qualité : logiciels critiques et temps réel »,” Feb. 2024.

[2] “ANSI X3.159-1989 "Programming Language C",” ANSI.

[3] “ISO/CEI 9899:1990 "Programming Language C",” ISO/CEI.

[4] “ISO/CEI 9899:2018 "Programming Language C",” ISO/CEI.

[5] “MISRA C:2023 Third Edition, Second Revision,” MISRA.

[6] “ISO/CEI 14882:1998 "Programming Language C++",” ISO/CEI.

[7] “ISO/CEI 14882:2020 "Programming Language C++",” ISO/CEI.

[8] “MISRA C++:2023,” MISRA.

[9] “ANSI/MIL-STD-1815A-1983 "reference manual for the Ada programming language",” ANSI.

[10] “ISO/CEI 8652:2023 "Information technology - Programming languages - Ada",” ISO/CEI.

[11] “Ada Reference Manuel ISO/IEC 8652:2012(E).” [Online]. Available: https://www.ada-europe.org/manuals/LRM-2012.pdf

[12] “ISO/CEI 18009:1999 "Information technology - Programming languages - Ada Conformity assessment of a language processor",” ISO/CEI.

[13] “Green Hills Optimizing Ada Compilers.” [Online]. Available: https://www.ghs.com/products/ada_optimizing_compilers.html

[14] “Ada for Embedded Systems.” [Online]. Available: https://www.ptc.com/en/products/developer-tools/apexada

[15] “Ada Europe.” [Online]. Available: https://www.ada-europe.org/

[16] “Ada Resource Association.” [Online]. Available: https://www.adaic.org/

[17] “Ada - France.” [Online]. Available: https://www.ada-france.org/

[18] “The AdaCore Blog.” [Online]. Available: https://blog.adacore.com/

[19] “Who's Using Ada?.” [Online]. Available: https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html#Banking_and_Financial_Systems

[20] “Ada and the Paris Subway.” [Online]. Available: http://archive.adaic.com/projects/atwork/paris.html

[21] “Vérification formelle d'un compilateur Lustre.” [Online]. Available: https://www.college-de-france.fr/fr/agenda/seminar/esterel-from-to-z/la-verification-formelle-un-compilateur-lustre

[22] “Optimize your timing before writing a single line of code.” [Online]. Available: https://www.absint.com/ait/scade.htm

[23] “Combining a High-Level Design Tool for Safety-Critical Systems with a Tool for WCET Analysis on Executables.” [Online]. Available: https://insu.hal.science/insu-02270107

[24] "aiT, StackAnalyzer integrated into SCADE Suite." [Online]. Available: https://www.absint.com/releases/080922.htm

[25] "LaFoSec : Sécurité et langages fonctionnels ." [Online]. Available: https://cyber.gouv.fr/publications/lafosec-securite-et-langages-fonctionnels

[26] Steven Varoumas and Tristan Crolard, "WCET of OCaml Bytecode on Microcontrollers: An automated Method and Its Formalisation ."

[27] Bruno Pagano *et al.*, "Experience Report: Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework ."

[28] "Règles de programmation pour le développement d'applications sécurisées en Rust," ANSSI. [Online]. Available: https://cyber.gouv.fr/publications/regles-de-programmation-pour-le-developpement-dapplications-securisees-en-rust

[29] "Frama-C site." [Online]. Available: https://frama-c.com/

[30] "IEEE Standard for Floating-Point Arithmetic," IEEE.

Annexe A. Paradigmes

À chaque langage de programmation vient sa manière de decrire les solutions algorithmiques au problèmes posés. Cette manière d'aborder les problèmes est ce qu'on appelle le *paradigme* du langage.

Les paradigmes les plus courants des langages informatiques sont la plupart du temps les suivants:
- impératif
- fonctionnel
- objet.

Il existe d'autres paradigmes moins répandus mais qui concernent les langages de programmation du rapport:
- déclaratif
- contrat
- synchrone.

Notons que les paradigmes ne sont pas exclusifs : un langage peut être composer plusieurs paradigmes et c'est même le cas général dans les langages généralistes. Toutefois, même en composant plusieurs paradigmes, il y en a souvent un qui se dégage plus que les autres à l'usage.

A.1 Impératif

Le paradigme impératif consiste à exprimer les calculs sous une forme structurée d'instructions modifiant un état mémoire. Les instructions emblématiques de ce paradigme sont
- l'affectation;
- le branchement conditionnel (si .. alors .. sinon ..);
- les sauts.

Les sauts ne sont pas toujours disponibles explicitement mais ils sont utilisés implicitement dans les boucles (`for` ou `while`) que l'on trouve dans tous les langages impératifs. Par exemple, en C, on peut calculer la longueur d'une liste chainée de la manière indiquée dans le 6 :

```c
struct list {
  int value;
  struct list *next;
};

int length(struct list *l)
{
  int len = 0;
  while (nullptr != l) {
    len++;
    l = l->next;
  }
  return len;
}
```

Calcul de la longueur d'une liste chainée en C

En considérant que `nullptr` représente la liste vide, la fonction `length` parcourt la liste en incrémentant un compteur `len` à chaque élément de la liste. Lorsque la liste est vide, la fonction retourne la longueur obtenue.

Le style impératif est le plus répandu dans les langages informatiques. Il est relativement proche de la machine mais propice aux erreurs de programmation car il n'est pas toujours évident de suivre l'état d'un système lorsque plusieurs morceaux de programme le modifient.

A.2 Fonctionnel

Le paradigme fonctionnel consiste à exprimer les calculs sous la forme d'une composition de fonctions mathématiques. Dans sa forme dite *pure*, les changements d'état du système sont interdits. En conséquence, il n'y a pas d'affectations ni de sauts.

Les branchements conditionnels existent mais les boucles sont remplacées par la récursivité, c'est-à-dire la possibilité pour une fonction de s'appeler elle-même. Par exemple, pour calculer la longueur d'une liste en OCaml, on procède récursivement :

```
1 let rec length (l : 'a list) : int = match l with
2   | [] -> 0
3   | _ :: tl -> 1 + length tl
```

Dans cet exemple, la fonction `length` prend en paramètre une liste et on fait une analyse de cas sur la structure de la liste. Si la liste est vide (`[]`), la longueur est 0. Sinon on analyse la tête et la queue (`tl`) de la liste et on ajoute 1 à la longueur de `tl`.

Le style fonctionnel est considéré comme plus sûr que le style impératif. Chaque fonction est une boite noire qui ne produit pas d'effets de bord, ce qui la rend plus facile à tester et à paralléliser. En contre-partie, le style fonctionnel nécessite beaucoup d'allocations dynamiques et donc un ramasse-miettes pour les gérer automatiquement. Cela peut avoir un impact négatif sur les performances et les capacités temps réel.

Un langage fonctionnel qui accepte la mutabilité est dit *impur*. Les langages fonctionnels impurs partagent les avantages et les inconvénients des deux mondes : ils permettent un gain en performances en évitant potentiellement beaucoup de copies de données mais héritent également de la difficulté de gérer un état mutable de manière sûre.

A.3 Objet

Le paradigme objet (ou POO[17]) consiste à traiter les données comme des entités, les *objets*, ayant leur propre état et des méthodes pour leur passer des messages. Chaque problème informatique est vu comme l'interaction d'un objet avec un ou plusieurs autres objets via les appels de méthode.

En POO, on distingue les classes, qui sont des modèles d'objets, et les objets eux-mêmes, qui sont des *instances* de classes. Par exemple, on peut représenter un point d'un espace bidimensionnel en C++ de la manière suivante :

[17] Programmation Orientée Objet

```
class Point2D
{
protected:
    uint32_t pos_x = 0;
    uint32_t pos_y = 0;
public:
    Point2D(uint32_t x, uint32_t y) : pos_x(x), pos_y(y) {}

    uint32_t get_x(void) const { return pos_x; }

    uint32_t get_y(void) const { return pos_y; }

    virtual void print(void) const {
        std::cout << "Point2D(" << pos_x << ", " << pos_y << ")" << std::endl;
    }
};
```

Le mot clé `class`, présent dans pratiquement tous les langages orientés objet, permet de déclarer un « patron » d'objet. Ici, le patron `Point2D` prend en paramètre de construction deux entiers `x` et `y` qui représentent les coordonnées du point. Ces coordonnées sont enregistrées dans les attributs `pos_x` et `pos_y` qui représentent l'état interne d'un objet `Point2D`. Les méthodes `get_x` et `get_y` permettent de récupérer les coordonnées du point et la méthode `print` permet de les afficher.

On peut alors créer une instance de `Point2D` de la manière suivante :

```
Point2D p(1, 2);
```

et afficher les coordonnées du point avec l'appel de la méthode `print` de l'objet `p` :

```
p.print()
// affiche "Point2D(1, 2)"
```

La POO introduit également la notion d'héritage qui permet de partager du code de manière concise. Par exemple, on peut étendre la classe `Point2D` pour les points d'un espace tridimensionnel :

```cpp
class Point3D : protected Point2D
{
protected:
    uint32_t pos_z = 0;
public:
    Point3D(uint32_t x, uint32_t y, uint32_t z) : Point2D(x, y), pos_z(z) {}

    uint32_t get_z(void) const { return pos_z; }

    void print(void) const override {
        std::cout <<
        "Point3D(" << pos_x << ", " << pos_y << ", " << pos_z << ")" <<
        std::endl;
    }
};

int main(void)
{
    Point3D p(1, 2, 3);
    p.print(); // affiche "Point3D(1, 2, 3)"
    return 0;
}
```

Ici, la classe `Point3D` hérite de `Point2D` en en récupérant toutes les méthodes et ajoute un attribut `pos_z` pour la coordonnée de profondeur. La méthode `print` est redéfinie pour afficher les trois coordonnées du point.

Le succès des langages objets (C++, Java, Javascript, ...) indique que le paradigme est très populaire dans les domaines applicatifs et le Web où la réutilisabilité est un critère de production déterminant.

Généralement, les objets ont un état interne mutable qui peut être modifié par les méthodes de la classe. De fait, le paradigme objet va intrinsèquement souffrir des mêmes défauts que le paradigme impératif sur la difficulté de suivre l'état réel d'un programme.

Par ailleurs, la hiérarchie des classes forme une structure arborescente qu'il est rapidement difficile de se représenter mentalement. Au delà d'une certaine profondeur, comprendre le flux de contrôle d'un programme sans outillage peut être une gageure.

A.4 Déclaratif

Le paradigme déclaratif consiste à décrire le problème à résoudre sans préciser comment le résoudre. Il s'agit ici de décrire le *quoi* et non le *comment*.

Les langages déclaratifs sont généralement orientés vers la description de données (XML, LaTeX, ...) mais il existe également des langages de programmation déclaratifs. Par exemple, Prolog est un langage de programmation qui permet de décrire un problème de manière logique et de laisser le moteur d'inférence résoudre le problème. Par exemple, on peut décrire un ensemble de faits:

```
animal(chat).
animal(chien).
```

et demander à Prolog quels sont les animaux possibles :

```
1 ?- animal(X).
2 X = chat ;
3 X = chien.
```

Cette abstraction permet généralement une programmation plus concise mais cela va induire :

- soit une perte de performances : le *comment* doit être retrouvé dynamiquement par le programme (Prolog) ;
- soit un modèle de compilation moins optimal (comme pour les langages fonctionnels purs comme Haskell) ;
- soit par une perte d'expressivité en restreignant le langage à un sous-ensemble plus facilement optimisable (Lustre).

A.5 Contrat

La programmation par contrat est un paradigme secondaire qui complète un les autres paradigmes. Cela consiste à ajouter dans le programme des propriétés qui peuvent être vérifiées statiquement par le compilateur (ou un outil tiers) ou dynamiquement à l'exécution et ajoutant des gardes automatiquement dans le code.

Ces propriétés prennent la forme d'assertions insérées à des endroits spécifiques. L'ensemble des assertions associé à un morceau de code est alors q forme un *contrat*. Les contrats permettent de clarifier la sémantique du programme et de réduire le nombre de *bugs*. Ils jouent ainsi un rôle complémentaire aux types et aux commentaires.

On distingue quatre types d'assertions:

- Les *préconditions* qui doivent être vérifiées à l'entrée d'une fonction ou d'une procédure ;
- Les *postconditions* qui doivent être vérifiées à la sortie d'une fonction ou d'une procédure ;
- Les *invariants* qui doivent toujours être vrais pour des valeurs d'un certain type ou les instances d'une classe ;
- Les *variants* qui sont des quantités attachées aux boucles ou aux fonctions récursives. Elle sont censées croitre ou décroitre de manière bornée à chaque itération ou appel récursif. Elles permettent de vérifier la terminaison d'un algorithme.

L'exemple ci-dessous est une implémentation de l'algorithme d'Euclide en Eiffel avec un contrat qui assure que l'on fournit une entrée valide et que l'algorithme termine:

```
gcd (value_1, value_2: INTEGER): INTEGER
   require
      value_1 > 0
      value_2 > 0
   local
      value: INTEGER
   do
      from
         Result := value_1
         value := value_2
      invariant
         Result > 0
         value > 0
         gcd(Result, value) = gcd(value_1, value_2)
      variant
         Result.max(value)
      until
         Result = value
      loop
         if Result > value then
            Result := Result - value
         else
            value := value - Result
         end
      end
   ensure
      Result = gcd(value_2, value_1)
   end
```

Dans cet exemple, les assertions `value_1 > 0` et `value_2 > 0` indiquées dans la section `require` sont des *précondistions*. L'assertion `Result = gcd(value_2, value_1)` dans la section `ensure` est une *postcondition*. Les invariants et les variants sont indiqués dans les sections idoines.

A.6 Synchrone

La programmation *synchrone* est utilisée dans le cadre des *systèmes réactifs*, c'est-à-dire des systèmes qui maintiennent en permanence une interaction avec un environnement et qui doivent être en mesure d'y réagir de façon synchrone, sûre et déterministe.

La famille des langages synchrones est elle-même divisée en deux familles:
- les langages *dataflow* (à *flots de données*);
- les langages *orientés contrôle*.

Les langage de programmation *dataflow* (Lustre, Scade) sont basés sur l'idée que les données sont variables au cours du temps et forment des *flôts* et que les opérations des combinateurs de flôts. Par exemple, le flôt `x` de type entier est vu d'un point de vue logique comme une suite infinie de valeurs entières. Si elle est constante (par exemple 1), on peut la représenter par de la manière suivante:

Temps	t_0	t_1	t_2	t_3	t_4	t_5	...
x	1	1	1	1	1	1	...

Si l'on combine ce flôt avec un autre flôt `y` variable et l'addition, on obtient un autre flôt `z` qui varie lui aussi au cours du temps en suivant les valeurs de `x` et `y`:

Temps	t_0	t_1	t_2	t_3	t_4	t_5	...
x	1	1	1	1	1	1	...
y	2	–1	4	3	–3	1	...
$z = x + y$	3	0	5	4	–2	2	...

Un programme revient alors à une description equationnelle des sorties en fonction des entrées.

Dans les programmes *orientés contrôle* (Esterel, ReactiveML), la temporalité s'exprime par les structures de contrôle du langage qui opèrent sur des signaux équivalent aux flôts précédents mais dont la valeur peu être présente ou absente. On distingue alors les instructions qui *prennent du temps* de celles qui sont considérées comme logiquement instantannées. Voici par exemple un programme ReactiveML (un surchouche réactive au dessus d'OCaml):

```
let process produce nat =
   let n = ref 0 in
   loop
      n := !n + 1;
      emit nat !n;
      pause
   end

let process print nat =
   loop
      await nat(n) in
      print_int n;
   end

let process main =
   signal nat in
   run (produce nat) || run (print nat)

let () = run main
```

Dans ce programme, on crée un processus `produce` qui prend un signal `nat` et boucle indéfiniment en incrémentant le compteur `n` et en émétant la valeur de `n` dans le signal `nat`. Comme toutes ces oprations sont instantannées, on insère une instruction `pause` qui prend du temps (logique) et évite à la boucle `loop` de tourner indéfiniment dans le même pas de temps. Le processus `print` boucle indéfiniment en attendant la valeur de `nat` et l'affiche. Comme attendre un signal prend du temps, la boucle `loop` n'est pas instantannée et n'a pas besoin de pause. Le processus `main` crée un signal `nat` et lance les processus `produce` et `print` en parallèle. Lorsqu'on exécute le programme, on obtient l'affichage 1, 2, 3, 4, 5, ... qui montre bien que les deux processus tournent en parallèle de manière synchrone.

Ce genre de programmation est particulièrement adapté aux systèmes interactifs car il permet de décrire de manière intuitive et concise les interactions en obtenant un résultat déterministe. Obtenir le même résultat avec un langage utilisant des *threads* est généralement plus complexe et moins sûr.

Dans les deux cas, les langages synchrones utilisent un modèle temporel logique pour ordonnancer les évènements et les réactions à ces évènements. Ce modèle de temps logique peut être callé sur le temps réel en choissisant une borne max-

imale entre deux instants logiques du système. Dans les langages comme Lustre ou l'ordonancement est statique et que le code engendré pour un pas de temps est déterministe, on peut calculer le WCET maximal d'un pas de temps et donc garantir l'adéquation avec le temps réel.

Annexe B. Analyseurs statiques

B.1 Généralités

Pour chaque langage étudié dans le rapport, nous présentons des outils d'analyse statique permettant de déduire de l'information par une interprétation abstraite du programme.

Les analyseurs statiques peuvent établir des métriques sur le code ou faire de la rétro-ingénierie pour en déduire des propriétés (et en vérifier). Ils peuvent également détecter des erreurs de programmation par heuristique ou vérifier le respect de normes de codage.

Étant donné le volume d'analyseurs sur le marché, le rapport ne prend en compte que les analyseurs dits *corrects*. Un analyseur statique est dit *correct* s'il ne donne pas de faux-négatif pour un programme sans *bug*. *A contrario*, il indiquera au moins un *bug* pour un programme qui en comporte. Cette garantie n'est donnée que par rapport à la modélisation mathématique qui en est faite et il est donc important que ce modèle soit raisonnablement proche de la réalité.

Chaque analyseur est étudié par rapport aux ressources publiques disponibles. Cela comprend le manuel utilisateur, les publications scientifiques, les brochures commerciales ou le site internet dédié.

Les types d'erreurs détectées dépendent des analyses effectuées par chaque outils et tous n'utilisent pas toujours la même terminologie pour désigner un type d'erreur. En conséquence, l'absence d'indication sur la présence d'une analyse particulière ne signifie pas que l'outil ne la fasse pas mais seulement que nous n'avons pas trouvé d'information à ce sujet.

B.2 Notes sur les analyseurs statiques

B.2.1 Frama-C

Parmi les analyseurs statiques étudiés, Frama-C[29] a la particularité de fonctionner avec des *plugins* dédiés à un type d'analyse particulier. La complétion de l'analyse globale considérée dans les comparatifs du document suppose que les *plugins* usuels soient utilisés, notamment :

- Eva ;
- Wp ;
- Mthread.

Certaines analyses complémentaires peuvent être proposées par l'outil via des plugins payants ou réalisés *ad hoc*.

B.2.2 Polyspace

Polyspace désigne une gamme de produit de la société MathWorks[18] qui propose différentes solutions autour de l'analyse statique.

Dans le cadre de cette étude, nous désignons par Polyspace la solution *Polyspace Code Prover*.

[18] https://fr.mathworks.com

B.2.3 TIS Analyser

TrustInSoft Analyser est un outil d'analyse statique développé par la société *TrustInSoft*[19]. C'est un *fork* de Frama-C qui utilise un ancien plugin *value* basé sur le calcul d'intervalles. *TrustInSoft* a amélioré le plugin, la traçabilité des erreurs et l'expérience utilisateur.

[19] https://trust-in-soft.com

Annexe C. Analyse numérique

C.1 Entière

Les erreurs d'arithmétique entière sont des erreurs qui surviennent lors de calculs sur des entiers machine. Il s'agit là des entiers que l'on trouve dans la plupart des langages de programmation et souvent désignés par le mot clé `int` ou dérivés.

La taille des entiers manipulables dépend de l'architecture du processeur utilisé. Sur la plupart des architectures, elle correspond aux puissances de 2 entre 8 et 64. Par exemple, un entier de 8 bits non signé (uniquement positif) peut représenter un entier mathématique de 0 à 255. Les formes signées utilisent un bit pour indiquer si l'entier est positif ou négatif; ce qui permet de représenter les entiers mathématiques de –128 à +127.

Dès lors que les entiers machines ont une taille finie, il est possible de réaliser des opérations qui dépassent cette taille. Par exemple, calculer $200 + 100$ sur un entier non signé de 8 bits donnera 44 (300 modulo 256). Ce résultat est correct dans une arithmétique modulo 256 mais il est rare que le programmeur pense dans cette arithmétique et il est plus probable qu'il s'agisse d'une erreur involontaire. Ce type d'erreur est appelé *overflow* (ou dépassement de capacité).

Ce genre d'erreur peut être assez subtil car bien qu'il soit détecté par le processeur qui positionne un drapeau signalant le dépassement, il nécessite que le programmeur vérifie ce drapeau explicitement pour en tenir compte; ce qu'il ne fait généralement pas.

Il y a aussi des opérations qui peuvent conduire à des comportements indéfinis par le langage. Les langages dont la sémantique n'est pas complètement standardisée donnent une liberté aux compilateurs pour interpréter certaines constructions. C'est ce qu'on appelle les *undefined behavior*. Cela est problématique lorsque le comportement du compilateur n'est pas celui imaginé par le développeur ou quand le programme est compilé avec un autre compilateur. Ce genre d'erreur doit également être relevé car il est susceptible de conduire à des erreurs d'exécution.

Une autre erreur couverte par l'analyse de l'arithmétique entière est la division par 0. Celle-ci n'est pas plus autorisée en informatique que dans les mathématiques générales et provoque une erreur fatale à l'exécution.

C.1.1 Flottante

L'arithmétique flottante est basée sur une représentation binaire des nombres réels. Les nombres flottants sont représentés par trois valeurs : le *signe*, la *mantisse* et l'*exposant*. La mantisse est la partie significative du nombre, l'exposant est la puissance de 2 ou de 10 à laquelle il faut multiplier la mantisse et le signe indique si le nombre est positif ou négatif.

Comme pour les nombres entiers, il existe plusieurs tailles de flottants et la norme [30] définit plusieurs formats en fonction de la base (2 ou 10) et le nombre de bits utilisés (en puissance de 2 de 16 à 128).

Bien que les nombres flottants permettent de représenter des nombres plus grands que leur équivalent entier, ils n'en sont pas moins sujet à des dépassement

de capacité. Cela étant, le calcul flottant est complet et en cas de dépassement ou de calcul impossible (comme la division par 0), le résultat est donné par des nombres spéciaux comme $+\infty$ ou $-\infty$ ou *NaN*[20].

Le fait que l'arithmétique flottante soit standardisée et complète en font parfois un choix de prédilection pour les calculs en général. Certains langages de programmation (comme Lua) ont un seul type numérique qui est, par défaut, représenté par un flottant.

D'un point de vue mathématique, l'arithmétique flottante a cependant un soucis d'arrondi inhérent au fait que la décomposition binaire d'un nombre peut être supérieure à sa capacité. Elle est même d'ailleurs infinie si le dénominateur du nombre n'est pas une puissance de 2. Par exemple, le nombre décimal 0.5 a une représentation flottante exacte car c'est $2^{-1} = \frac{1}{2}$. En revanche 0.1 ($\frac{1}{10}$) n'a pas de représentation exacte car sa décomposition binaire donne $0,0\overline{0011}$ (où 0011 se répète à l'infini). Pour pouvoir représenter ce nombre, il faut l'arrondir à une certaine précision et généralement 0.1 vaut en 0.10000000000000000555 en flottant.

Ces arrondis donnent lieu à des approximations qui peuvent s'accumuler et devenir problématiques lorsqu'on enchaîne les opérations. Elles peuvent également donner lieu à des résultats contre intuitifs. Par exemple, en calcul flottant, l'égalité $0.1 + 0.2 = 0.3$ est fausse ! Avec les arrondis, $0.1 + 0.2$ vaut 0.30000000000000004441 tandis que 0.3 vaut 0.29999999999999998890. Dans ce contexte, écrire des algorithmes numériques fiables peut être très délicat.

Les analyses statiques sur l'arithmétique flottante peuvent détecter les opérations ambiguës comme la comparaison ci-dessus et vérifier la marge d'erreur en effectuant un calcul abstrait sur les intervalles possibles ou en évaluant le delta de précision qui est, de manière qui peut paraître paradoxale, calculable de manière exacte. Il existe également des bibliothèques de calcul pour réaliser à *runtime* les opérations en utilisant ces deux techniques.

[20] *Not A Number* (pas un nombre), utilisé pour les opérations invalides

Annexe D. Pointeurs & Mémoire

D.1 Gestion des pointeurs

Les pointeurs sont des valeurs représentant l'adresse en mémoire d'une autre valeur. Ils sont couramment utilisés pour passer des valeurs par référence, et par conséquent, éviter de copier ces valeurs à chaque fois qu'on les utilise dans le flôt de contrôle.

La manipulation des pointeurs est plus ou moins explicite suivant les langages. En C, les pointeurs sont des valeurs de première classe et peuvent être manipulés directement. À l'inverse, en OCaml, les pointeurs n'apparaissent jamais directement dans le code source mais sont utilisés implicitement par le compilateur.

Lorsque les pointeurs sont explicites, il y a trois type d'erreurs courantes qui peuvent survenir :

- le débordement;
- le déréférencement de `NULL`;
- le *dangling pointer*

Le débordement survient lorsqu'on écrit ou lit en mémoire à une adresse qui n'est pas dans une plage de valeurs attendues. Typiquement, cela arrive lorsqu'on adresse un tableau en dehors de ses limites. Si on a un tableau de 10 éléments et qu'on accède au 11ème élément, on accède à une zone pouvant contenir n'importe quelle donnée et très probablement pas celle qui était imaginée par le programmeur.

Le déréférencement de `NULL`, ou plus généralement le déréférencement invalide, consiste à utiliser une adresse invalide et à tenter de lire ou écrire à cette adresse. Quand un programme est lancé par un système d'exploitation, ce dernier lui octroit une zone mémoire pour son usage exclusif. Si le programme tente d'accéder à une zone mémoire qui ne lui appartient pas, une erreur d'accès mémoire est levée et le programme est arrêté. Le cas le plus courant est celui du déréférencement de `NULL`. `NULL` est une valeur spéciale utilisée pour dénoter des valeurs particulières, en parucilier l'absence de valeur. Par exemple, une liste chaînée se termine souvent par un pointeur `NULL` pour indiquer la fin de la liste. Dans la plupart des systèmes, `NULL` vaut techniquement 0 et ce n'est pas une adresse valide.

Le *dangling pointer* survient lorsqu'on utilise un pointeur sur une valeur qui n'a jamais été allouée ou été libérée. Dans les deux cas, la valeur pointée est indéterminée et l'utilisation de cette valeur peut conduire à des comportements indéfinis. Par exemple, en C le code suivant pose problème :

```c
int *p = calloc(1, sizeof(int)); /* allocation d'un entier dans le tas */
*p = 42;                         /* écriture de 42 à l'adresse pointée */
free(p);                         /* libération de la mémoire */
printf("%d\n", *p);              /* utilisation de la valeur pointée */
```

Après le `free(p)`, l'adresse mémoire pointée par `p` peut être réutilisée par une autre partie du programme et contenir n'importe quelle valeur. Il n'est donc pas possible de prédire la valeur affichée par le `printf`.

Notons que le *dangling pointer* apparaît généralement dans les situations où l'allocation dynamique se fait manuellement (C, C++, ...) mais il peut très bien arriver sans allocations dynamiques par échappement de portée :

```c
int *p;
{
  int x = 42;
  p = &x;
}
printf("%d\n", *p);
```

ou plus généralement :

```c
int *func(void)
{
    int n = 42;
    /* ... */
    return &n;
}
```

Dans les deux cas, le pointeur (`p` ou `&n`) pointe sur une valeur qui a été allouée dans la pile mais qui n'est plus valide à la sortie du bloc représenté par les accolades.

D.2 Gestion de la mémoire

Un programme utilise deux zones de mémoire : une zone statique et une zone dynamique. La zone statique est prédéfinie à la compilation et contient les données globales (ou explicitement statiques avec le mot clé `static` du C) et ne peut être modifiée. La zone dynamique est réservée à l'éxécution du programme et contient deux sous-zones : la pile et le tas.

La *pile* est une zone de mémoire avec une taille maximale définie par le système et qui se remplit automatiquement en fonction du modèle d'execution du langage. En général, elle se remplit à chaque appel de fonction avec les données locales et l'adresse de retour des fonctions, puis se vide lorsque les fonctions terminent. Un dépassement de pile (*stack overflow*) peut survenir lorsque qu'on cumule trop d'appel de fonction. Cela peut arriver facilement avec une récursion mal maîtrisée. Un dépassement de pile est une erreur fatale au programme.

Le *tas* est une zone libre de mémoire dans laquelle on peut allouer des données dynamiquement. L'allocation dynamique est réalisée par l'appel de fonctions spécifiques qui demandent au système d'exploitation de réserver une zone de mémoire de taille donnée. Lorsque la mémoire n'est plus utilisée, il est nécessaire de la libérer pour éviter les *fuites mémoire*. Si trop de mémoire est allouée sans être libérée, le programme peut consommer toute la mémoire disponible et être arrêté par le système d'exploitation dans le meilleur des cas. Dans le pire des cas, le système entier est paralysé car privé des ressources mémoires nécessaires à son fonctionnement nominal.

Dans la bibliothèque standard du C, la fonction `malloc` et ses dérivées peuvent être utilisées pour allouer et `free` pour libérer la mémoire. Pour éviter les fuites mémoires, il faut donc un `free` pour chaque allocation réalisée. Cependant, les programmes non triviaux mettent en oeuvre des structures de données complexes dont certaines parties sont allouées dynamiquement et savoir quand faire le `free`

peut être délicat. Il peut arriver que le programmeur pêche par exces de prudence et se retrouve à libérer deux fois une même allocation (*double free*).

La double libération a un comportement indéterminé et peut conduire à une corruption de l'allocateur de mémoire entrainant des allocations incohérentes et difficiles à diagnosiquer.

Certaines analyses statiques peuvent détecter les fuites mémoires et les doubles libérations en suivant le flôt de contrôle du programme et en vérifiant que chaque allocation est bien libérée une et une seule fois.

Notons que les langages à gestion automatique de la mémoire ne sont pas exempt de fuites mémoires. La gestion automatique de la mémoire est basée sur un ramasse-miettes qui libère la mémoire automatiquement lorsqu'elle n'est plus utilisée. Toutefois, lorsque le ramasse-miettes ne peut pas déterminer si une donnée est encore utilisée ou non, il peut décider de la conserver alors qu'elle n'est plus utilisée. C'est le cas des ramasse-miettes dits «conservatifs».

Annexe E. Mesures statiques

E.1 WCET

Le calcul du WCET est important sur les systèmes temps réel pour garantir que les tâches critiques se terminent dans un temps donné. Le WCET est le temps maximal que peut prendre une tâche pour se terminer. Il est calculé en fonction des temps d'exécution des instructions, des branchements et des accès mémoire.

Il existe deux méthodes pour calculer le WCET : la méthode *dynamique* et la méthode *statique*. La méthode dynamique consiste à exécuter le programme dans l'environnement cible et à mesurer le temps d'exécution un nombre de fois jugé suffisant pour établir un WCET statistique. Cette méthode est fiable mais coûteuse en temps et en ressources.

La méthode statique consiste à analyser le programme sans l'exécuter et à déduire le WCET à partir de cette analyse. Le problème de cette méthode est qu'il est difficile de calculer un WCET précis du fait des optimisations réalisées par le compilateur et de la complexité des architectures modernes. La prédiction de branchement ou l'utilisation des caches peut faire varier le le temps d'exécution de manière importante et rendre le WCET difficile à calculer sans le majorer excessivement.

E.2 Analyse de la pile

L'analyse de la pile consiste à calculer la taille maximale de la pile utilisée par un programme. La plupart du temps, cette analyse se fait directement sur le binaire car le langage source ne contient pas forcémement les informations adéquates pour réaliser cette analyse puisqu'elles sont ajoutées durant la compilation.

Connaître la taille maximale de la pile utilisée est important pour dimensionner et optimiser les systèmes embarqués qui peuvent être très contraints en mémoire. Cela permet d'éviter les dépassements de pile (*stack overflow*) qui sont des erreurs fatales.

L'analyse de la pile peut n'est pas décidable lorsque le programme utilise des pointeurs de fonctions car il n'est pas forcément possible de déterminer la suite d'appels de fonctions à l'avance. Dans ce cas, l'analyse de la pile peut demander à annoter le code source pour avoir plus d'informations.

Annexe F. Concurrence

L'utilisation du calcul parallèle au sein d'un même programme se fait généralement par l'utilisation des *threads* fournis par le système d'exploitation sous-jacent. Les threads permettent de lancer plusieurs fonctions en parallèle en exploitant éventuellement la multiplicité des coeurs de calcul de la machine. En toute théorie, cela permet d'accélérer le calcul et de rendre le programme plus réactif.

Pour raisonner avec du calcul parallèle, il est nécessaire de mettre en place des mécanismes de partage d'information entre les *threads*. Ces mécanismes sont généralement des variables partagées ou des files de messages. Lorsque plusieurs *threads* accèdent à une même variable, il est possible que les valeurs lues ou écrites soient incohérentes si les *threads* ne sont pas synchronisés. Par exemple, si un *thread* écrit une valeur dans une variable et que l'autre *thread* lit cette valeur avant que l'écriture ne soit terminée, il est possible que le second *thread* lise une valeur intermédiaire qui n'est pas celle attendue.

Pour éviter ce genre de problème, il est nécessaire de synchroniser les *threads* entre eux. Les mécanismes de synchronisation les plus courants sont les *mutex* (verrous) et les *sémaphores*. Toutefois, ces mécanismes se mettent en place manuellement en fonction de ce que le programmeur imagine comme étant le bon ordonnancement des *threads*. Or, cet ordonnancement n'est pas toujours celui qui est effectivement réalisé par le système d'exploitation, ce qui rend les problèmes de concurrence très difficiles à reproduire et à corriger.

Certaines analyses statiques peuvent détecter des erreurs de concurrence en simulant les ordonnancements possibles et en vérifiant que les valeurs lues et écrites sont cohérentes. En fonction de l'analyse effectuée, il est possible de détecter deux types d'erreurs :

- les *data races* (courses critique) qui surviennent lorsqu'un *thread* lit ou écrit une valeur partagée sans synchronisation;
- les *deadlocks* (interblocages) qui surviennent lorsqu'un *thread* attend une ressource qui est détenue par un autre *thread* qui lui-même attend une ressource détenue par le premier.

Annexe G. Meta formalisation

Pour certains langages, il est possible d'utiliser des outils de formalisation pour spécifier des programmes et prouver que des propriétés. Il existe globalement trois stratégies pour développement formel pour les langages cibles:

- l'extraction de code;
- la génération de code;
- la vérification de code.

L'extraction de code consiste à écrire un programme dans un langage de spécification (comme Coq ou Isabelle/HOL par exemple) et à extraire un programme exécutable dans un langage cible. Ce dernier peut être soit le langage voulu soit un langage compatible avec le langage voulu. Par exemple, on peut extraire un programme OCaml de Coq et compiler ce programme en un fichier objet pouvant être lié à un programme C. C'est l'approche la plus simple pour faire des développements formels intégrables dans des systèmes existants mais elle a plusieurs inconvénients :

- elle suppose que l'extraction de programme est correcte;
- le langage cible n'est pas forcément agréé pour le développement critique.

La génération de code consiste à injecter le langage cible dans le langage formel (qui devient un langage hôte). Cette injection se fait en

1. décrivant le langage cible dans le langage hôte ;
2. programmant dans le langage hôte le programme du langage cible ;
3. imprimer le programme cible dans un fichier ou plusieurs fichiers.

Les fichiers obtenus reprensent alors un programme valide du langage cible mais qui a été formalisé dans le langage hôte. Au final, cette approche permet de décrire des programmes idiomatique au langage cible dans une syntaxe plus ou moins proche de celui-ci suivant les capacités du langage hôte et de démontrer des propriétés sur ces programmes. L'inconvénient réside essentiellement dans le travail nécessaire pour décrire une injection complète du langage cible dans le langage hôte qui peut se compter en années de travail.

La vérification de code consiste partir du programme à vérifier dans le langage cible et à décrire des propriétés sur ce programme par le biais d'annotations (la plupart du temps dans les commentaires). Ces propriétés sont ensuite vérifiées par un outil de vérification tiers. Cette approche est la plus pratique d'un point de vue opérationnel car elle ne nécessite pas de manipuler des langages formels directement. Les équipes de développement peuvent continuer à travailler dans leur langage habituel et l'ajout des annotations peut se faire par d'autres personnes. Toutefois, cette approche est limitée par les capacités de l'outil de vérification et les annotations doivent être suffisamment précises pour être utiles. Cela demande souvent une connaissance assez fine de l'outil de vérification pour être efficace. Par ailleurs, et comme pour la solution précédente, l'outil doit avoir un modèle sémantique du langage cible qui nécessite beaucoup de travail. C'est l'approche des outils de vérification de code comme Frama-C avec par exemple let code annoté suivant:

```
1 /*@
2   requires \valid(a) && \valid(b);
3   assigns *a, *b;
4   ensures *a == \old(*b);
5   ensures *b == \old(*a);
6 */
7 void swap(int* a, int* b){
8   int tmp = *a;
9   *a = *b;
10  *b = tmp;
11 }
12
13 int main(){
14  int a = 42;
15  int b = 37;
16  swap(&a, &b);
17  //@ assert a == 37 && b == 42;
18  return 0;
19 }
```

où l'assertion sera vérfiée par l'outil et les solveurs sous-jacents à l'aide des propriétés spécifiées dans le commentaire de la fonction `swap`.

Annexe H. Test

Il y a plusieurs type d'outils pour tester un programme que l'on peut diviser en trois catégories qui ne sont pas forcément exclusives:

- les outils d'écriture de tests;
- les outils de génération de tests;
- les outils de gestion des tests.

Les outils d'écriture de tests permettent de décrire les tests à réaliser. Ils sont souvent fournis sous la forme de bibliothèques logicielles qui fournissent des fonctions, des macros ou des annotations pour décrire les tests à l'intérieur du programme lui même ou d'un programme à part.

L'écriture des tests étant souvent laborieuse, il existe des générateurs de tests qui offrent la possibilité de réaliser une bonne partie des tests unitaires et parfois fonctionnels de manière automatique. Ces générateurs peuvent être basés sur des techniques de génération aléatoire (*fuzzing*) qui peut être guidée, de génération de modèle (*model checking*) ou de génération de cas de test aux limites.

Les outils de gestion des tests engloblent les outils qui vont ajouter de l'intelligence dans l'éxécution des tests et factoriser le plus possible les exécutions qui peuvent être coûteuses en temps et en ressources.

Notons que certains outils proposent également la générations de méta données utiles pour la certification ou qualification de logiciels. Certains peuvent engendrer des matrices de tracabilité ou des rapports préformatés pour les processus idoines.

Étant donné qu'il existe pléthore de cadres logiciels pour uniquement écrire des tests et que la plus-value de ces cadres dans le processus d'édition de logiciel critique est limitée, nous nous concentrerons essentiellement sur les outils qui offrent un minimum de génération ou de gestion de tests offrant le plus de valeur ajoutée pour les logiciels critiques.

Par ailleurs, il faut aussi distinguer quels types de test l'outil peut gérer. Dans le rapport, nous distinguerons le type de test avec une lettre majuscule selon le tableau suivant :

Type	Description	Identifiant
Unitaire	Teste une unité de code (fonction, module, classe, ...)	U
Intégration	Teste l'intégration de plusieurs unités de code	I
Fonctionnel	Teste une fonctionnalité du programme	F
Non régression	Teste que les modifications n'ont pas introduit de régression	N
Robustesse	Teste une unité de code ou une fonctionnalité avec des valeurs aux limites, voir hors limites	R
Couverture	Teste la couverture du code par les tests	C

Les critères d'analyse utilisés sont la capacité de

- générer des tests automatiquement
- de gérer efficacement les tests (factorisation, parallélisation, rapports, …)
- faire du *mocking*, c'est-à-dire de la simulation de dépendances.