Aide à l'analyse de traces d'exécution dans le contexte des microcontrôleurs 32 bits

Azzeddine Amiar

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : Arrêté 7 août 2006

Présentée par

Azzeddine AMIAR

Thèse dirigée par **Mme Lydie du Bousquet**
et codirigée par **M Yliès Falcone**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Aide à l'Analyse de Traces d'Exécution dans le Contexte des Microcontrôleurs

Thèse soutenue publiquement le **novembre 2013**,
devant le jury composé de :

Mme Mireille Ducasse
Professeur, IRISA, Rapporteur
Mme Hélène Waeselynck
Chargée de Recherche, LAAS, Rapporteur
Mme Laurence Pierre
Professeur, TIMA, Examinatrice
Mr Stéphane Frénot
Professeur, CITI, Examinateur
Mme Lydie du Bousquet
Professeur, LIG, Directrice de thèse
M Yliès Falcone
Maître de conférences, LIG, Co-Directeur de thèse

Remerciements

Cette thèse doit beaucoup aux nombreuses personnes qui m'ont encouragé et soutenu au long de toutes ces années. Qu'elles trouvent dans ce travail l'expression de mes plus sincères remerciements.

Je souhaite renouveler mes remerciements à Lydie du Bousquet pour avoir dirigé et encadré cette thèse, aussi pour ses précieux conseils. Je lui suis extrêmement reconnaissant de la confiance qu'elle m'a toujours accordée et de son enthousiasme permanent. J'aimerai aussi remercier mon co-directeur de thèse, Yliès Falcone pour sa disponibilité et son encouragement. Je remercie également Mickaël Delahaye pour sa collaboration et ses conseils.

Je remercie nos partenaires industriels, particulièrement les gens d'AIM, STMicroElectronics et EASII-IC pour leur aide. Aussi pour leurs conseils et remarques, toujours pertinents, qui m'ont permis de confronter et ajuster mes travaux à des problématiques pratiques et réelles.

Je tiens à remercier les membres de mon comité de thèse pour l'intérêt qu'ils ont porté à ce travail et pour les remarques constructives qu'ils ont faites sur mon projet de thèse. J'adresse mes remerciements aux membres du jury pour avoir accepté de juger ce travail.

Je remercie les membres de l'équipe VASCO de m'avoir permis d'évoluer au sein de cette famille pendant cinq ans.

Un grand merci à tous mes amis de Grenoble, avec qui j'ai passé d'agréables moments. Merci aussi à mes deux amis en Algérie, Houssem Abderrahmane Sahraoui et Salah Benlahrech.

Mes remerciements s'adressent enfin à mon père et ma mère, ma grand-mère et mes sœurs qui m'ont toujours épaulé et encouragé. C'est grâce à leur amour et soutien que j'ai atteint mes objectifs.

Un merci particulier à Ziri, pour son support, sa présence et sa disponibilité.

Résumé

Un microcontrôleur est un petit ordinateur qu'on peut trouver dans de nombreux appareils, comme les téléphones mobiles et les machines à laver. En cas de dysfonctionnement, déboguer un microcontrôleur reste une tâche difficile à faire. Cette difficulté est liée principalement à l'absence d'un historique du fonctionnement du microcontrôleur.

Afin de répondre au besoin de pouvoir faire du débogage, les microcontrôleurs récents permettent d'avoir des traces d'exécution. Un grand nombre de programmes embarqués sur les microcontrôleurs s'appuient sur une boucle principale qui est parcourue indéfiniment, pour par exemple lire les valeurs des capteurs. Par conséquent, à cause de cet aspect cyclique des programmes embarqués, les traces contiennent une quantité importante d'information. De plus, l'absence de données d'entrées et de sorties, ainsi que les optimisations du compilateur rendent l'analyse de ces traces encore plus difficile.

Nos partenaires industriels nous ont demandé de proposer des solutions d'analyse et d'aide au débogage (localisation de faute(s)), qui utilisent une seule trace d'exécution générée à partir d'un programme cyclique. La raison est que d'un microcontrôleur est le plus souvent influencé par l'environnement dans lequel il est installé. Par conséquent, il est très difficile de générer d'autres traces du programme embarqué.

Étant donné l'aspect cyclique des programmes de notre travail, une étape importante est l'identification de cycles dans la trace d'exécution. La détection de cycles repose sur l'identification de l'instruction représentant la boucle principale dans le code source. Cette instruction est appelée *loop-header*. Une fois le *loop-header* identifié, la trace est divisée en cycles. Cette contribution est présentée dans le chapitre 4.

Dans le but d'aider à la compréhension et à l'analyse, notre première approche [15] offre à l'ingénieur une description globale et représentative de la trace d'exécution. Cette description est produite en utilisant la compression basée sur la génération d'une grammaire. La compression est basée sur la détection de répétitions dans la trace, tout en préservant son aspect cyclique. Notre approche ainsi que notre algorithme *Cyclitur* sont présentés dans le chapitre 4. Les évaluations concernant cette contribution sont présentées dans le chapitre 5.

L'interprétation de la compression fournie par notre approche est simplifiée grâce l'interface graphique et aux visualisations proposées par notre outil, CoMET, présenté dans le chapitre 16.

Souvent, l'observation d'une défaillance est liée à l'exécution de l'instruction fautive. Ainsi, notre seconde contribution concerne une étape importante du processus de débogage qui est la *localisation de faute(s)*. Notre approche

est basée sur l'analogie entre les exécutions du programme et les cycles. Elle utilise une seule trace d'exécution et exploite les chemins d'exécution (cycles).

Le plus souvent, l'enregistrement de la trace est arrêté quand un événement empêchant la poursuite de l'exécution apparaît. Ainsi, ce mécanisme assure la présence d'un comportement anormal au moins dans le dernier cycle, appelé *cycle fautif*.

Une première contribution dans la localisation de faute(s) concerne l'utilisation de la méthode du *plus proche voisin* [16], qui permet de mettre en évidence un cycle particulier, appelé « plus proche voisin ». Ce « plus proche voisin » est le cycle d'exécution en succès qui partage le plus d'instructions avec le cycle fautif. Après l'identification du « plus proche voisin », notre adaptation de cette méthode cherche à identifier ses différences avec le cycle fautif.

Une seconde contribution concerne l'utilisation de différents cœfficients de similarité dans le but de classer les instructions pouvant contenir la faute [16]. Cette contribution est présentée dans le chapitre 9. Les cœfficients utilisés par notre approche sont : *Ochiai*, *Tarantula* [41], *Ample* [28], *Jaccard* [24] et le cœfficient O^p [58]. L'évaluation expérimentale de cette contribution est présentée dans le chapitre 10. Cette évaluation a permis également de mettre en avant l'efficacité de notre *filtrage*, qui a pour objectif de réduire l'effort d'analyse. Ce processus, décrit dans le chapitre 9, permet de réduire le nombre de cycles à prendre en considération pour la localisation de faute(s).

Afin de lever l'hypothèse utilisée dans la deuxième partie de cette thèse, notre dernière contribution prend en considération le fait que les multiples cycles d'une même exécution peuvent interagir entre eux de plusieurs façons. Ainsi, pour faire de la localisation de faute(s) pour ce type de cas, nous nous intéressons à la *fouille de données* et spécialement la *recherche de règles d'association*, présentée dans le chapitre 13. La principale motivation de l'utilisation de la fouille de données est de comparer les comportements *suspects* et les comportements *corrects*.

Ainsi, avant de rechercher les règles d'association, notre approche, en utilisant la distance de Hamming, groupe les cycles de la trace en deux ensembles : les cycles *suspects* et les cycles *corrects*. Ensuite, l'application d'un algorithme de recherche de règles d'association, appelé LCM [72], sur ces deux ensembles de transactions permettra de définir les comportements jugés corrects et ceux jugés comme suspects. Ainsi, pour la localisation de faute(s), nous proposons à l'ingénieur un diagnostic basé sur l'analyse des règles d'association selon leurs degrés de suspicion. Notre approche est présentée dans le chapitre 14. Son évaluation, présentée dans le chapitre 15, permet de mesurer son efficacité en calculant l'effort nécessaire à la localisation de faute(s).

Abstract

A microcontroller is an integrated circuit that incorporates onto the same microchip the essential computer elements [62]. Microcontrollers are embedded in various kinds of equipment such as mobile phones and washing machines. Surprisingly, even though microcontrollers are now quite affordable, the development of embedded software still weights heavily both on the final cost of the product and the time to market.

According to our industrial partners, the main costs are due to the analysis and validation steps. In fact, in case of malfunction, debugging a microcontroller remains a difficult task. This difficulty is due to the limited observability of the execution of the embedded software. Indeed, engineers still use oscilloscopes to analyze embedded applications by interpreting electric signals. Consequently, validation and fault diagnosis are usually carried out manually, and are thus tedious and time-consuming tasks [67].

Last generation of microcontrollers include parts dedicated to trace collection. For example, ARM Cortex-M includes a section dedicated to trace collection, called *Embedded Trace Macrocell* (ETM) [7]. Using specialized probes, such as Keil *UlinkPro* [2] probe, it is possible to collect basic execution traces without input/output data. However, the execution trace contains a huge volume of low-level data. In addition, because of memory constraints or performance reasons, software code is heavily optimized before being loaded in microcontrollers, which complicates the analysis.

Many embedded programs can be categorized as *cyclic programs*, as they rely on a main loop that iterates indefinitely. In the following, the instruction that defines this main loop is called the *loop header*. Usually, at each iteration of the main loop, the sensors are monitored and actions are taken in response to changes. Due to the cyclic nature of embedded programs, collected execution traces consist in long sequences of multiple repetitions of instructions.

In this context, our industrial partners would like to analyze the behavior and localize a fault in a program given a *single* trace that ends at the failure. The reason is that, most often, a microcontroller is influenced by the environment in which it is installed. Therefore, it is difficult to generate other traces.

Given the cyclic aspect of embedded programs, an important step in our work consists in identifying cycles in the execution trace. Our cycle detection process is based on the identification of the statement representing the main loop-header. Once the loop-header is identified, it is used to divide the trace into cycles. This contribution is presented in chapter 4.

Our first contribution[15], presented in chapter 4, aims to help automated or manual analysis of microcontroller traces by facilating the localization of

repetitions and by keeping the amount of data manageable. We propose a compression approach based on a grammar generation. Our algorithm, *Cyclitur*, is based on our extension of the Sequitur algorithm [61]. Sequitur produces a grammar by leveraging *regularities* found in an input trace. The output grammar is an accurate but compact representation of the input trace. The experimental evaluations regarding this contribution are presented in chapter 5.

Our tool, named *CoMET* and presented in chapter 16, implements our compression approach, and provides graphic visualizations of the generated compression. The engineer, by visualizing the compression generated by our approach, will be guided throughout the trace analysis, for instance, by identifying cycles that appear most often in the trace or by comparing cycles.

Our second contribution [16] concerns an important step in the debugging process, which is fault localization. This approach, presented in chapter 9, is based on Automatic Fault Localization (AFL) methods [65, 42] and takes advantage of the cyclic nature of traces. Our approach is based on the analogy between executions of the program and the cycles. It first automatically detects the cycles in the trace before using adapted AFL method to find the most suspicious statements. The effectiveness of our method is demonstrated by an experimental evaluation, made possible thanks to our tool CoMET [15].

The evaluation, presented in chapter 10, shows encouraging results. In fact, the proposed method allows to find the bug in several faulty programs by inspecting in most cases less than 5% of the program code, with the best suspiciousness ranking, namely Ochiai [9]. This evaluation has also enabled us highlight the effectiveness of our filtering process, which aims to reduce the cost of analysis. This process, described in chapter 9, allows to reduce the number of cycles used by the fault localization process.

Our last contribution, presented chapter 14, takes into account the fact that the multiple cycles of a same execution can interact between them in several ways. Then, to localize the fault(s) for this type of cases, we are interested in data mining and specially association rules (chapter 13). To localize the fault(s), we use data mining to compare the suspicious behaviors and the correct behaviors.

Before searching association rules, our approach uses the Hamming distance to group the cycles of the trace into two sets: the *suspect* cycles and the *correct* cycles. Then, searching association rules using these two sets of cycles will help to define the correct behaviors and the suspicious ones. To localize the fault(s), we propose to the engineer a diagnosis based on the analysis of the association rules according to their suspicion levels. The evaluation of this approach, presented in chapter 15, allowed to measure its effectiveness by calculating the expense required for fault localization.

Table des matières

Introduction

Un microcontrôleur est une sorte de petit ordinateur qu'on peut trouver dans de nombreux appareils, comme les téléphones mobiles et les machines à laver. Les microcontrôleurs sont apparus à la fin des années 70 [4], quand les applications domestiques ou industrielles avaient besoin de systèmes « intelligents » ou tout au moins programmables. Si votre radio réveil se déclenche le matin, et vous cliquez sur le bouton « répéter », la première chose que vous faites dans votre journée est d'interagir avec un microcontrôleur. Réchauffer de la nourriture dans le four micro-ondes ou utiliser votre téléphone mobile, impliquent aussi l'exploitation d'un microcontrôleur. Mais que se passerait-il si votre machine à laver, qui doit commander différents éléments selon un programme bien choisi, exécute un programme fautif ou si le microcontrôleur était défaillant ? Faudrait-il changer la machine à laver ou juste réparer son microcontrôleur ?

D'un point de vue pratique et économique, il serait plus intéressant de cibler le microcontrôleur. Pour pouvoir réparer le microcontrôleur, il est primordial de comprendre son fonctionnement pendant l'exécution de la tâche (i.e. laver le linge avec un programme bien choisi), et de déterminer la cause du dysfonctionnement : logicielle, matérielle, ou bien les deux. Cependant, il est très difficile d'observer le dispositif et d'analyser son comportement pendant l'exécution de la tâche.

Pour répondre à ces besoins d'aide à la compréhension et à la localisation d'anomalies, nous avons travaillé sur la réalisation d'une solution logicielle. Notre solution combine la compression basée sur la génération d'une grammaire, des techniques de localisation de faute, et des techniques de fouilles de données.

Développement dans le contexte des microcontrôleurs

Le contexte global dans lequel s'inscrit cette thèse est celui du développement de programmes destinés à être exécutés sur microcontrôleurs. Un microcontrôleur est un circuit intégré réunissant tous les éléments d'une structure à base de microprocesseur [62]. Il se compose d'un CPU, d'une mémoire de données (RAM et EEPROM), d'une mémoire programme (ROM, OTPROM, UVPROM ou EEPROM), d'entrées sorties, etc (Figure 2). Les microcontrôleurs font partie de notre quotidien et sont présents dans presque tout ce qui nous entoure, que ce soit dans le domaine du transport (voiture, avion, train...), de la communication (téléphone, satellite...), ou bien celui de l'électroménager (téléviseur, machine à laver...), pour ne citer que ceux-là.

Les microcontrôleurs imposent souvent de lourdes contraintes de program-

FIGURE 1 – Microcontrôleur ST3210C-EVAL

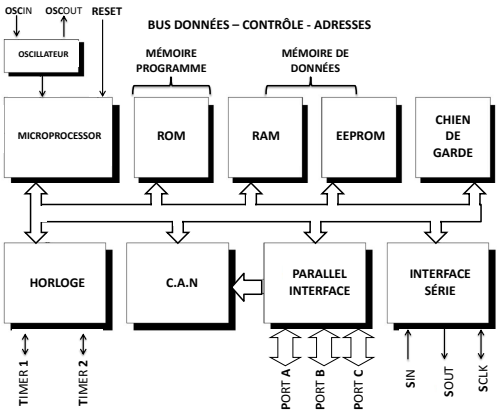

FIGURE 2 – Architecture d'un Microcontrôleur

mation, qui sont : l'utilisation de ressources matérielles, le temps de calcul, la sûreté/sécurité, la consommation d'énergie. Nous discutons ces contraintes ci-après.

L'utilisation de ressources matérielles : afin de réduire les coûts unitaires, un programme embarqué fonctionne sur du matériel conçu spécialement pour une utilisation embarquée et avec des ressources informatiques limitées, comme un microcontrôleur. Par conséquent, de nombreux mécanismes que les développeurs de programmes non-embarqués ont l'habitude d'utiliser sont absents, ou sont disponibles uniquement sous des formes rudimentaires. Dans un même temps, les systèmes embarqués sont très concurrents, et fonctionnent à un faible niveau d'abstraction du matériel. Par conséquent, leur conception et leur mise en œuvre sont compliquées par des facteurs qui peuvent être largement abstraits dans d'autres contextes. Des exemples de ces facteurs sont : la prévention des inter-blocages et les contraintes de temps.

Le temps de calcul : Souvent, dans le contexte du microcontrôleur, le temps est un paramètre essentiel. Les délais d'exécution doivent alors être connus et/ou bornés.

La sûreté/sécurité : Selon le contexte dans lequel un microcontrôleur est utilisé, son bon fonctionnement peut devenir primordial tant d'un point de vue humain (appareillage médical) que d'un point de vue économique (distributeur d'argent).

La consommation d'énergie : Il est possible qu'un microcontrôleur ne soit pas toujours relié à une source infinie d'énergie. Par conséquent, le programme embarqué doit tenir compte de cette contrainte et gérer la quantité d'énergie fournie. Comme par exemple pour les réseaux de

capteurs.

Tout développement dans le contexte des microcontrôleurs nécessite la prise en compte des contraintes exposées ci-dessus. Afin de faciliter le développement d'applications embarquées sur microcontrôleur, différentes techniques et outils ont été proposés, comme Qt Extended, qui est une plate-forme libre basée sur la bibliothèque Qt [1], ou bien LabVIEW [3] qui est un logiciel de développement basé sur un langage de programmation graphique. AGILIA est un autre exemple d'outil de développement, proposé par AIM, l'un de nos partenaires industriels. Cet outil permet de faire du développement basé sur composants. AGILA fournit un ensemble de composants prédéfinis, destinés à réaliser certaines taches, cependant l'utilisateur peut enrichir la bibliothèque en définissant ses propres composants.

L'existence de nombreux outils et techniques simplifie le développement d'applications pour microcontrôleur et le rend moins couteux en temps. Cependant il n'est pas rare que les systèmes embarqués contiennent quelques fautes logicielles. La combinaison de l'aspect embarqué des programmes et le manque d'observabilité des dispositifs, ainsi que le coté bas niveau et la caractéristique temps réel de ce domaine, créent une situation unique. Par conséquent, le débogage reste très coûteux et varie entre 50% et 75% du coût total du développement [35].

Débogage dans le contexte des microcontrôleurs

Les utilisateurs d'application se trouvent souvent confrontés à ce qu'on appelle un « bogue ». Un bogue (en anglais *bug*) est une anomalie de fonctionnement d'un programme. Cette dernière est causée par une ou plusieurs fautes, où une faute dans le programme est le plus souvent introduite par un humain [8, 50]. Comme illustré dans la Figure 3, lors de l'exécution, l'instruction fautive peut introduire une erreur dans l'exécution, qui, si elle se propage, peut aboutir à une défaillance, que l'on appelle bug.

Le débogage, ou déverminage (en anglais *debugging*), est le processus qui consiste à trouver et à corriger les fautes dans un programme à exécution défaillante, comme il est illustré dans la Figure 4. Un débogueur est un logiciel qui aide le développeur à analyser les bogues d'un programme.

Quelques différences existent entre le débogage d'un programme sur ordinateur et le débogage d'un programme embarqué sur un microcontrôleur. Pendant la phase de développement d'applications, le matériel est supposé être exempt d'erreurs, ce qui signifie que chaque sous-système (CPU, mémoire,

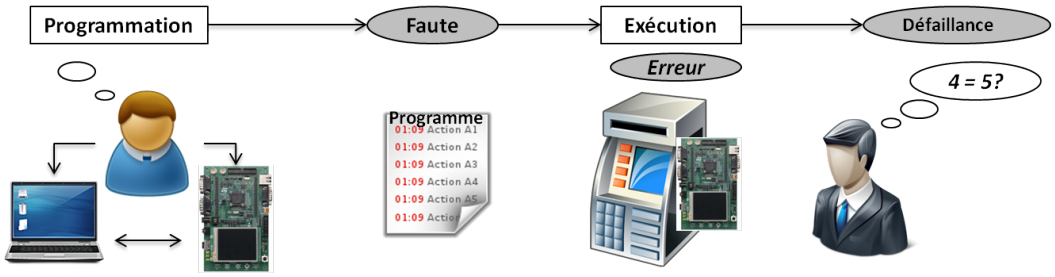

FIGURE 3 – Faute, erreur, défaillance

stockage, I/O) fonctionne parfaitement. Cependant, sur un microcontrôleur, le matériel lui-même peut être défaillant, comme une mémoire instable par exemple. Par ailleurs, comme expliqué dans la section 1.1, un microcontrôleur possède des ressources limitées, comme la puissance de calcul, la mémoire, ou bien le nombre d'entrées sorties. Cette limitation de ressource rend peu pratique, voir impossible, l'exécution d'un déboguer et du programme à déboguer sur le microcontrôleur lui-même.

Afin de répondre à cette problématique liée à ces restrictions, de nouvelles technologies ont été développées, permettant de déboguer les applications embarquées sur les microcontrôleurs en utilisant du débogage à distance (remote-debugging) : le débogueur est exécuté sur un ordinateur hôte, et commande le microcontrôleur par le biais de matériel (une sonde JTAG, par exemple) et d'un petit logiciel s'exécutant sur le microcontrôleur. Il est alors possible pour le développeur de faire usage de tout le confort de son poste de travail, alors que le microcontrôleur n'a pas besoin d'exécuter un débogueur.

Toutefois, cette solution n'est pas envisageable quand le microcontrôleur est embarqué dans un système plus grand. En cas de dysfonctionnement, déboguer un microcontrôleur, une fois qu'il est installé sur la cible (appelé débogage post-mortem), reste une tache difficile à faire. D'une part, parce qu'il n'y a aucun historique du fonctionnement du microcontrôleur, d'une autre part, parce qu'il est très difficile de simuler ou de reproduire les conditions

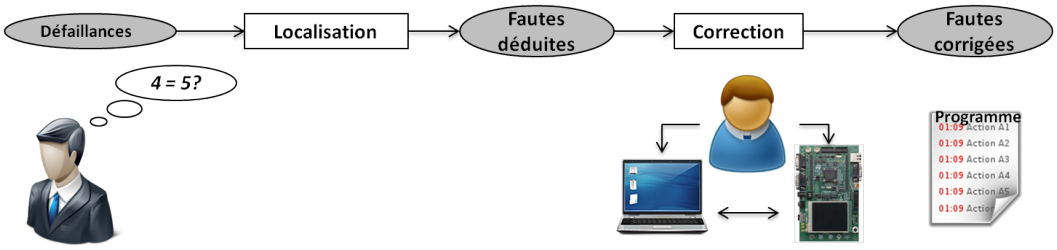

FIGURE 4 – Processus de débogage

d'environnement dans lequel se trouve le microcontrôleur.

Afin de répondre au besoin de pouvoir faire du débogage, à distance et/ou post-mortem, des microcontrôleurs avec plus de ressources sont fabriqués. Cette dernière génération de microcontrôleur utilise une nouvelle technologie permettant d'avoir une partie de l'historique d'exécution, appelée trace d'exécution, pour faire du débogage. Par exemple, le STM3210C (Figure 1), dispose d'un microprocesseur *ARM Cortex-M* qui inclut une section dédiée à la collecte et à l'enregistrement de la trace, appelée ETM (*Embedded Trace Macrocell*) [7].

Ainsi, en utilisant des sondes spécialisées, telles que Keil-UlinkPro [2] (Figure 5) et ST-Link [6], il est possible de collecter des traces d'exécution sans données (entrés/sorties).

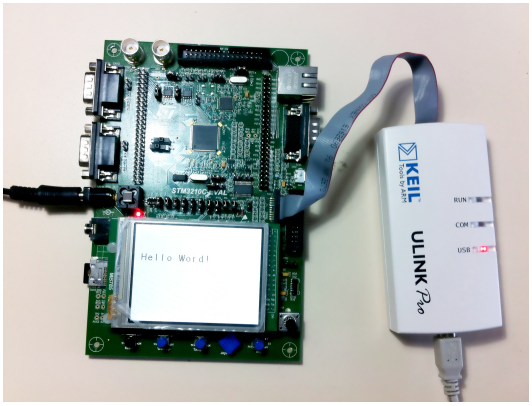

FIGURE 5 – Microcontrôleur STM32 et sa sonde UlinkPro

Les traces d'exécution comportent des informations sur le déroulement des exécutions. Les informations contenues dans les traces d'exécution sont donc des indices pour comprendre les erreurs. Dans notre contexte de travail, ces informations concernent les chemins d'exécution. Mais l'analyse des traces est difficile.

Les traces peuvent êtres très grandes, car les chemins d'exécutions peuvent être très longs. De plus, à cause des limitations matérielle, ces traces ne contiennent pas les valeurs d'entrées/sorties, ce qui rend la reconstitution du comportement complexe. Par ailleurs, les optimisations du compilateur sont à l'origine de transformations entre le programme original et ce qui est réellement exécuté. En effet, en raison des restrictions présentent dans la programmation sur microcontrôleurs, le compilateur procède à des optimisations permettant d'obtenir des gains d'utilisation mémoire et de temps de calcul.

Ainsi, le travail présenté dans cette thèse s'inscrit dans la problématique de l'analyse de trace, pour l'aide à la localisation de faute, dans le cadre du projet

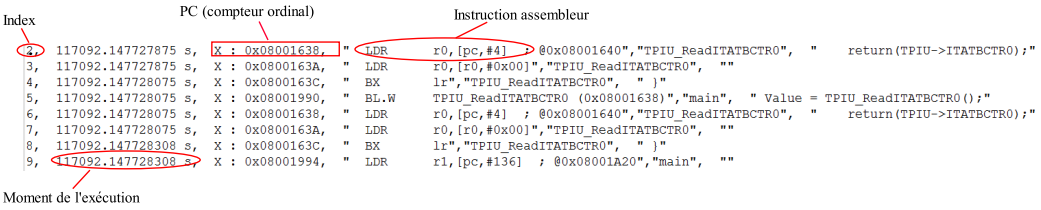

FIGURE 6 – Quelques lignes d'une trace d'exécution extraite d'un microcontrôleur

FUI-IO32 [1]. Le contexte précis de notre travail correspond à un problème de nos partenaires industriels : « que peut-on faire pour aider à la localisation de faute lorsque l'on dispose d'une seule trace d'exécution qui se termine sur une défaillance ? ». Ainsi, pour l'aide à l'analyse, nos partenaires industriels nous ont demandé de n'utiliser dans nos approches qu'une seule trace d'exécution générée à partir d'un programme cyclique. La raison est qu'un microcontrôleur est le plus souvent branché à d'autres dispositifs, tels que les capteurs de température par exemple. Par conséquent, le comportement d'un microcontrôleur est le plus souvent influencé par l'environnement dans lequel il est installé. Ainsi, lorsqu'on essaye de générer d'autres traces du programme embarqué, il est difficile de reproduire l'environnement et les conditions dans lesquelles le microcontrôleur était installé, lorsque la trace en question a été enregistrée. La principale caractéristique de nos approches est donc l'utilisation d'une seule trace d'exécution pour l'analyse et la localisation de faute. La section suivante résume les approches suivies.

Contributions

Un grand nombre de programmes embarqués sur les microcontrôleurs sont classés comme des programmes *cycliques*. Un programme cyclique est un programme qui s'appuie sur une boucle principale qui est parcourue indéfiniment. Nous appelons *tête de boucle* (en anglais *loop-header*) l'instruction qui définit la boucle principale dans le code source. La nature cyclique des programmes s'explique par le fait qu'en l'absence de système d'exploitation, le programme est en charge de scruter les entrées de façon active. À chaque itération de la boucle principale, en fonction des données d'entrée reçues de l'environnement du microcontrôleur, des actions spécifiques sont exécutées. Une trace d'exécution générée par l'exécution d'un programme cyclique est dite *trace cyclique*. C'est le type de trace que nous étudierons dans la suite.

Ainsi, à partir d'une unique trace cyclique se terminant par la mise en

1. http ://io32.forge.imag.fr/

évidence d'une défaillance, nous proposons différentes approches pour aider, premièrement à la compréhension du comportement du programme embarqué sur le microcontrôleur, deuxièment à la localisation de la faute à l'origine de la défaillance. Cette thèse comporte donc trois parties dédiées d'une part à la compréhension de la trace (partie 1) et d'autre part à la localisation des fautes dans le code source (partie 2 et 3).

Une étape préliminaire importante de nos approches est l'identification de cycles dans la trace d'exécution. La détection de cycles repose sur la localisation de l'événement, dans la trace d'exécution, qui représente le *loop-header* dans le code source. Une fois le *loop-header* identifié, il est possible de diviser la trace en blocs, où chaque bloc représente un cycle spécifique, c'est-à-dire une exécution de la boucle principale. Cette contribution est présentée dans le chapitre 4.

Une fois le loop-header identifié, il est possible d'offrir à l'ingénieur une description globale et représentative de la trace d'exécution à analyser [15]. Cette représentation permet à l'ingénieur d'avoir une idée des cycles à analyser en détails. Elle est représentée sous forme d'une grammaire générée en utilisant la compression basée sur la génération d'une grammaire. Le principe de cette compression est basé sur la détection de répétitions dans la trace, tout en préservant son aspect cyclique. L'interprétation de la compression fournie par notre approche est simplifiée grâce à l'interface graphique et aux visualisations proposées par notre outil, CoMET, présenté dans le chapitre 16. Les évaluations concernant nos approches de compression et de détection de cycles dans la trace sont présentées dans le chapitre 5.

La seconde partie de la thèse porte sur la *localisation de faute(s). La localisation de faute(s)* peut être définie comme étant le processus de repérage de faute(s) conduisant à des défaillances. C'est un processus qui peut être long et fastidieux. C'est pourquoi un enjeu consiste à automatiser la localisation de faute(s). Souvent, l'observation d'une défaillance est liée à l'exécution de l'instruction fautive. Le repérage précis de fautes étant très difficile, nous essayons plutôt d'identifier les instructions du code source susceptibles de contenir la faute.

Pour localiser une faute en utilisant *une seule* trace d'exécution, nous avons adapté des stratégies basées sur les *Spectres d'exécution* [66, 37].

Une première contribution [16] concerne l'utilisation de la méthode du *plus proche voisin* [65]. L'objectif derrière l'utilisation de cette méthode est de mettre en évidence un cycle particulier, appelé « plus proche voisin ». Ce « plus proche voisin » est le cycle d'exécution en succès qui partage le plus d'instructions avec le cycle de l'exécution en échec.

Pour trouver le « plus proche voisin », la méthode calcule une distance entre le spectre de l'exécution en échec et chaque spectre d'exécution en succès.

Pour cela, nous avons adapté la distance de Hamming [36] à notre contexte. Ce travail est présenté dans le chapitre 9.

Une seconde contribution consiste à adapter des techniques utilisant des *cœfficients de similarité* pour classer les instructions pouvant contenir la faute [16]. Cette contribution est présentée dans le chapitre 9. Notre approche utilise par défaut le cœfficient *Ochiai* [9, 17]. Cependant, notre outil CoMET permet de faire de la localisation de faute(s) en utilisant également les cœfficients : *Tarantula* [41], *Ample* [28], *Jaccard* [24] et le cœfficient O^p [58], en plus de la méthode du *plus proche voisin*. L'évaluation expérimentale de cette contribution est présentée dans le chapitre 10. Cette évaluation a permis également de mettre en avant l'efficacité de notre *Filtrage*, qui a pour objectif de réduire l'effort d'analyse. Ce processus, décrit dans le chapitre 10, permet de réduire le nombre de cycles à prendre en considération pour la localisation de faute.

Notre dernière contribution prend en considération le fait que les multiples cycles d'une même exécution peuvent interagir entre eux de plusieurs façons.

Pour faire de la localisation de fautes pour ce type de cas, nous nous intéressons à la *fouille de données* et spécialement la *recherche de règles d'association*. Notre travail concernant cette partie est inspiré des travaux de Cellier et al. [21] concernant la localisation de faute en utilisant la fouille de données. La localisation de faute proposée par Cellier et al. [21] est basée sur la recherche de *règles d'association* [13] mais également l'*analyse formelle de concepts* [64, 20]. Le résultat de leur approche est un treillis des défaillances que l'ingénieur doit parcourir et explorer pour localiser la faute.

Dans notre contexte, la principale motivation de l'utilisation de la fouille de données est d'extraire des informations de comportements considérés comme bons et des comportements considérés comme fautifs, afin de les comparer et localiser la faute. Notre approche est présentée dans le chapitre 14.

-Partie 1-

La compression de traces basée sur la génération d'une grammaire

Résumé

Afin de pouvoir faciliter le débogage, les microcontrôleurs récents permettent de récupérer des traces d'exécution. Cependant, souvent, ces traces sont très longues. De plus, l'absence des données d'entrées et de sorties rendent l'analyse de ces traces difficile.

L'objectif des approches présentées dans cette partie de la thèse est de faciliter l'analyse à l'ingénieur, en lui fournissant une description globale et représentative de la trace d'exécution à analyser [15]. Cette description est représentée sous forme d'une grammaire générée en utilisant un processus de compression.

Ainsi, le chapitre 1 de cette partie sert à introduire le contexte et les motivations de notre travail tout en présentant la compression basée sur la génération d'une grammaire.

Le principe de notre approche de compression est basé sur la détection de répétitions dans la trace, tout en préservant son aspect cyclique. Pour faire cela, nous nous sommes inspirés de l'algorithme de compression *Sequitur* [60], présenté dans le chapitre 2, en proposant une extension dans le chapitre 3.

Étant donné l'aspect cyclique des programmes embarqués, une étape importante de notre travail étant l'identification de cycles dans la trace d'exécution. Cette contribution ainsi que l'exploitation des cycles sont présentées dans le chapitre 4.

Les évaluations expérimentales concernant nos approches de compression et de détection de cycles dans la trace sont présentées dans le chapitre 5.

Chapitre 1

Introduction et Motivation

Sommaire

Afin de répondre au besoin d'analyse du comportement des microcontrôleurs, de nouveaux microcontrôleurs avec plus de ressources sont fabriqués. Les dernières générations de microcontrôleur utilisent une nouvelle technologie permettant d'avoir une partie de l'historique d'exécution, appelée trace d'exécution. Par exemple, le STM3210C, dispose d'un microprocesseur *ARM Cortex-M* qui inclut une section dédiée à la collecte et à l'enregistrement de la trace, appelée ETM (*Embedded Trace Macrocell*) [7].

Les traces d'exécution produites par l'ETM comportent des informations concernant les chemins d'exécution à l'exclusion des données (entrées/sorties et variables). Ces traces pevent êtres très longues. Par conséquent, étudier le comportement d'un microcontrôleurs à partir de sa seule trace d'exécution constitue un défi couteux en temps et en effort.

L'idée ayant donné naissance au travail présenté dans ce chapitre est simple. Un programme embarqué sur un microcontrôleur est une suite d'instructions s'exécutant de façon répétitive. Par conséquent, une trace d'exécution de ce programme consiste en des répétitions d'instructions. Pourquoi ne pas identifier ces répétitions (Figure 1.1) et les synthétiser pour offrir une description représentative du comportement du programme ? Ceci permet de faciliter la compréhension de la trace d'exécution.

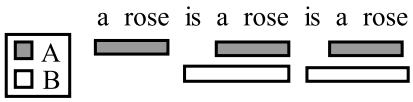

Figure 1.1 – Détection des répétitions dans la phrase « a rose is a rose is a rose »

Pour cela, nous nous inspirons d'un algorithme de compression de données, appelé Sequitur, dont le principe de compression est basé sur la génération d'une grammaire (en anglais *Based-Grammar Compression*), notée BGC [53, 26]. La compression de données consiste à réduire la taille de l'information pour son stockage et son transport. L'utilisation de la compression basée sur la génération d'une grammaire répond à ce besoin de réduction de taille. Le principe consiste à stocker une petite grammaire qui génère une séquence donnée au lieu de stocker la séquence elle-même. En cas de besoin, la séquence d'origine peut être facilement reconstruite à partir de la grammaire.

De nombreuses procédures de compression de données utilisent ce principe et plusieurs résultats empiriques indiquent que souvent, l'utilisation d'une stratégie basée sur la grammaire fournit une meilleure compression comparée à l'utilisation d'autres techniques (p.ex. Unix Compress ou les algorithmes de Gzip) [77, 51]. Cependant, la plupart des outils de compression utilisant des grammaires peuvent fournir une mauvaise compression dans les cas où pour une séquence de symboles donnée, une petite grammaire pouvant la générer existe, alors que ces outils produisent de grandes grammaires [23]. Par conséquent, de nombreuses recherches ont été menées dans le but de répondre au besoin de génération de la plus petite grammaire [45, 77, 44, 23].

Le principe de la compression basée sur la génération d'une grammaire peut être exprimé en utilisant un exemple intuitif. Si un mot σ contient de nombreuses occurrences d'une sous-séquence ω, ces occurrences seront remplacées par une seule variable A qui est associée à ω, noté comme suit : $A \rightarrow \omega$. La séquence de symboles σ est alors compressée selon la fréquence d'apparition de la sous-séquence ω.

Les traces extraites de microcontrôleurs contiennent des sous-séquences qui se répètent à des fréquences plus ou moins importantes. En considérant une trace d'exécution comme une séquence de symboles, et les séquences qui se répètent comme des sous-séquences, il devient alors possible d'appliquer une approche de compression basée sur la génération d'une grammaire sur les traces d'exécution.

La principale motivation de l'utilisation d'une telle approche sur une trace d'exécution extraite d'un microcontrôleur est d'aider à la compréhension du comportement du dispositif, grâce à la détection de séquences répétitives dans la trace. Dans le cas des programmes hautement répétitifs, cette approche permet aussi de détecter des anomalies dans le comportement du dispositif, s'il en existe, par exemple, en repérant les répétitions les moins fréquentes dans une trace.

Le travail présenté dans cette partie nous a valu la publication de l'ar-

ticle [15] dans *IEES 2013*[1].

Ce chapitre définit les notions permettant de mieux comprendre le fonctionnement de la compression basée sur la génération d'une grammaire, ainsi que les contextes de son utilisation. Ensuite, nous détaillons l'algorithme Sequitur et les propriétés des grammaires qu'il construit (Chapitre 2). Puis, nous verrons comment nous avons adapter l'algorithme pour répondre au mieux à notre problématique.

1.1 Préliminaires

Le but de cette section est d'introduire les définitions qui serviront de bases au reste de ce document. Ces définitions concernent *les séquences de symboles* et *les grammaires*, qui sont utilisées dans la compression basée sur la génération d'une grammaire.

1.1.1 Notions de bases

Un *alphabet* Σ est un ensemble fini de symboles. Étant donné un alphabet Σ, une *séquence de symboles* est une suite d'éléments de Σ. L'ensemble des séquences finies dans Σ est noté Σ^*. La longueur d'une séquence de symboles $\sigma \in \Sigma^*$, notée $|\sigma|$, représente le nombre de symboles dans σ. Le i-ème élément de σ est noté σ_i. La séquence vide, c'est-à-dire la séquence de longueur 0, est notée ϵ_Σ ou bien simplement ϵ. Un *digramme* est défini comme une séquence de longueur deux.

1.1.2 Grammaire

Une grammaire hors-contexte (appelée grammaire dans la suite) est un quadruplet $\langle \Sigma, \Gamma, S, \Delta \rangle$ tel que :

- Σ est un alphabet fini de symboles *terminaux*,
- Γ est un alphabet fini de symboles *non-terminaux*, disjoint de Σ,
- $S \in \Gamma$ est le *symbole de départ*, c.-à-d., un non-terminal particulier,
- $\Delta \subseteq \Gamma \times (\Sigma \cup \Gamma)^*$ est un ensemble fini de *règles de production* (ou plus simplement *règles*),

L'ensemble Σ représente l'ensemble des symboles de la séquence pour laquelle une grammaire est construite. L'ensemble Γ représente l'ensemble des nouveaux symboles utilisés pour la construction de la grammaire. Une règle

1. http ://www.iess.org/

de production $\langle A, \alpha \rangle \in \Delta$ est notée $A \to \alpha$, où le non-terminal A (respectivement la séquence de symboles α) est appelé *entête* (respectivement *corps*) de la règle. La *règle de départ* est l'unique règle contenant le non-terminal S comme entête. Par extension, le *corps de la grammaire* est l'ensemble des corps des règles, c'est-à-dire que le corps d'une grammaire $G = \langle \Sigma, \Gamma, S, \Delta \rangle$ est défini comme suit :

$$Corps(G) = \{\alpha \mid \exists A : \langle A, \alpha \rangle \in \Delta\}. \tag{1.1}$$

Exemple

Dans cet exemple, une grammaire est générée pour le proverbe anglais : « a rose is a rose is a rose ». Ce proverbe signifie que « les choses sont ce qu'elles sont ».

$$S \to BBA$$
$$A \to \underline{a\ rose}$$
$$B \to A\ \underline{is}$$

Dans cet exemple, l'ensemble des terminaux utilisés est $\Sigma = \{a, e, i, o, r, s\}$. Pour la construction de la grammaire, trois nouveaux symboles sont utilisés : A, B, et le symbole de départ S. L'ensemble de non-terminaux est donc $\Gamma = \{A, B, S\}$. L'ensemble des règles de production est $\Delta = \{S \to BBA,\ A \to \underline{a\ rose},\ B \to A\ \underline{is}\}$. Par conséquent, le corps de la grammaire générée est $Corps(G) = \{BBA,\ \underline{a\ rose},\ A\ \underline{is}\}$.

En comptant les espaces, le texte initial a 26 caractères. La compression générée contient 10 symboles de moins que la chaîne initiale. Ainsi, dans cet exemple, la compression a un gain supérieur à 38%.

1.2 Contexte d'utilisation de la BGC

A présent que les notions de bases de la compression basée sur la génération d'une grammaire ont été définies, dans cette section nous discutons son utilisation. Le contexte présenté dans cette section donne un aperçu du contexte qui nous a inspiré dans notre travail.

La compression basée sur la génération d'une grammaire (BGC) est une technique largement étudiée [19, 57]. Ce type de compression permet de capturer des répétitions, même si elles sont très éloignées dans le texte. De ce fait, cette approche peut être très efficace en ce qui concerne la compression des textes « hautement-répétitif » [68, 27, 46].

L'utilisation de ce type de compression permet d'accélérer le traitement sur les séquences, par exemple, le « combinatorial pattern matching » [19, 54,

69, 76], « edit-distance computation » [29], et la recherche de patrons particuliers [38, 57]. Nous nous intéresserons plus particulièrement à cette dernière utilisation afin de proposer une représentation synthétique et pertinente de la trace d'exécution.

La détection de patrons se traduit par la localisation des non-terminaux leur correspondant dans la grammaire générée. Un exemple fondamental de l'utilisation de la reconnaissance de patrons dans une séquence, consiste à identifier les régularités dans des séquences d'ADN [59, 49], où un algorithme basé sur la génération d'une grammaire, appelé *Sequitur*, a été utilisé. Des algorithmes utilisant cette approche ont servi également à mettre en évidence des patrons dans des partitions musicales [61, 59], et à découvrir des propriétés de la langue à partir d'un ensemble de textes [30]. L'utilisation de ce type de compression dans les exemples cités ci-dessus est possible parce qu'une grammaire représentant une séquence de symboles reste relativement compréhensible. La génération de compressions compréhensibles est un atout majeur de ce type de compression comparée à d'autres schémas de compression.

Dans notre contexte, l'idée de compresser les traces d'exécution en utilisant la génération d'une grammaire, afin d'aider dans leur analyse et compréhension, s'inspire principalement du contexte présenté ci-dessus. Ainsi, il est possible de proposer une vision globale et représentative de la trace.

Sequitur est un algorithme de compression dont le principe est basé sur la détection de patrons répétitifs. Plusieurs améliorations [48, 77], ou adaptations aux contextes [52, 47], ont été apportées à *Sequitur*. Cependant il n'existe pas, à notre connaissance, une optimisation/adaptation répondant aux besoins de notre problématique d'aide à l'analyse de la trace. Afin de mieux comprendre les modifications apportées à *Sequitur*, l'algorithme, ainsi que les propriétés de la grammaire qu'il génère sont discutées dans le chapitre suivante.

Chapitre 2

Sequitur

Sommaire

2.1 Introduction

Créé par Nevill-Manning et Witten en 1997, Sequitur est un algorithme de compression en temps linéaire, utilisant la génération de grammaires [61, 60]. Il est utilisé dans une variété de domaines, comme la bioinformatique [26], et le traitement du langage naturel [74]. Sequitur prend une séquence de symboles en entrée et génère en sortie une grammaire représentant la séquence donnée. De ce fait, la compression de Sequitur est une compression sans perte d'information, basée sur la détection des répétitions de sous-séquences dans une séquence donnée.

Le principe de Sequitur est très simple : chaque fois qu'un motif de deux symboles successifs apparaît au moins deux fois, Sequitur remplace ses occurrences par un nouveau non-terminal, dont l'unique production est le motif en question. Ce motif peut contenir des non-terminaux précédemment générés. Donc, chaque répétition de motif donne lieu à une règle dans la grammaire. Dans le cas où la création d'un nouveau non-terminal implique qu'un ancien non-terminal n'est utilisé qu'une seule fois, ce dernier est supprimé et remplacé par le motif qu'il engendre.

Sequitur exécute le processus de compression de manière itérative, c'est-à-dire qu'il construit une grammaire au fur et à mesure qu'il parcourt la séquence de symboles. En considérant que le stockage et la récupération de l'information dans une table de hachage se réalisent en temps constant, l'algorithme a alors une complexité temporelle linéaire par rapport à la taille de la séquence.

Sequitur est décrit de façon informelle dans l'Algorithme 1, page 24. Il commence par créer une règle, dont l'entête est le symbole de départ S et le

corps est la séquence vide ϵ, après il poursuit son traitement en itérant sur les symboles de la séquence d'entrée. Chaque symbole subit un traitement en deux phases. Premièrement, Sequitur ajoute le nouveau symbole comme un symbole terminal à l'extrémité droite du corps de la règle de départ. Deuxièment, s'il détecte une répétition d'un motif, il applique une ou plusieurs étapes de réduction afin d'assurer deux propriétés sur la grammaire : l'*unicité du digramme* et l'*utilité des règles*. Ces deux propriétés sont expliquées dans la suite. Il est important de préciser que ces propriétés ont été identifiées et prouvées par Nevill-Manning et Witten dans [61, 60].

2.2 Propriétés des grammaires générées par Sequitur

Les lignes citées dans les explications des propriétés font références aux lignes de l'Algorithme 1, donné page 24. Cet algorithme prend comme entrée une séquence σ. En respectant deux propriétés : l'*Unicité du digramme* (lignes 6-15) et l'*Utilité des règles* (lignes 16-21), il produit comme sortie une grammaire G permettant de générer cette séquence σ. Ces deux propriétés sont discutées ci-dessous.

Unicité du digramme : Cette propriété spécifie que le corps d'une grammaire ne doit pas contenir deux occurrences, qui ne s'entremêlent pas, du même digramme (première ligne de la Table 2.1).

Chaînes	Digrammes	Description
abab	*ab ab*	occurrences non entremêlés
aaa	*aa aa*	occurrences entremêlés

TABLE 2.1 – Exemple d'occurrences de digramme

Propriété 1. *La propriété « unicité du digramme », notée Uniqueness*(G)*, est maintenue pour une grammaire* G*, si pour tous non-terminaux* $A, B \in \Gamma$*, symboles* $a, b, c, d \in \Sigma \cup \Gamma$*, et séquences de symboles* $\alpha, \beta, \gamma, \delta \in (\Sigma \cup \Gamma)^*$*, les deux assertions suivantes sont vérifiées :*

$$(A \neq B \wedge A \to \alpha\ ab\ \beta \wedge B \to \gamma\ cd\ \delta) \Rightarrow ab \neq cd \tag{2.1}$$

$$(A \to \alpha\ ab\ \beta\ cd\ \gamma) \Rightarrow ab \neq cd \tag{2.2}$$

L'assertion (2.1) exprime que si une grammaire G contient deux règles $A \to \alpha\, ab\, \beta$ et $B \to \gamma\, cd\, \delta$, avec $A \neq B$, alors ab et cd sont deux motifs

(digrammes) différents. L'assertion (2.2) exprime que, si la grammaire G contient une règle $A \to \alpha\, ab\, \beta\, cd\, \gamma$, alors ab et cd sont deux motifs (digrammes) différents.

Sequitur assure l'*unicité du digramme* comme suit : si l'ajout d'un symbole (ligne 4) produit une répétition d'un digramme (motif) dans le corps de la grammaire, où les occurrences de la répétition ne s'entremêlent pas, la propriété n'est plus maintenue (ligne 6). Sequitur restaure la propriété, soit en créant une nouvelle règle, soit en réutilisant une règle existante (lignes 7-15). Dans le cas où le digramme répété correspond déjà au corps d'une règle existante, Sequitur remplace ce digramme avec le non-terminal correspondant (ligne 9), c'est-à-dire l'entête de la règle concernée. Dans le cas contraire, il crée une nouvelle règle avec le digramme qui se répète comme corps et un nouveau non-terminal comme entête, pour ensuite remplacer les deux occurrences du digramme avec ce nouveau non-terminal (lignes 11-14).

Utilité des règles : cette propriété assure que chaque règle, plus précisément le non-terminal représentant son entête, à l'exception du symbole de départ S, soit utilisé au moins deux fois dans le corps de la grammaire.

Propriété 2. *La propriété « utilité des règles », notée Utility*(G)*, est vérifiée pour une grammaire* G *si :*

$$\forall A \in \Gamma \setminus \{S\} : \left(\sum_{\langle B,\beta\rangle \in \Delta} \left|\{i \in [1..|\beta|] \mid \beta_i = A\}\right| \right) \geq 2\,. \tag{2.3}$$

Sequitur assure l'*utilité des règles* comme suit : si l'entête d'une règle n'est utilisée qu'une seule fois (ligne 16) dans le corps de la grammaire, Sequitur supprime la règle et remplace l'apparition de son entête (non-terminal) avec son corps (lignes 17-20). Ce mécanisme permet la formation de règles dont le corps est constitué de plus de deux symboles.

Afin d'aider à mieux comprendre les propriétés de la grammaire générée par Sequitur, un exemple détaillé est présenté ci-dessous.

2.3 Exemple

Dans cet exemple, une compression grammaticale en utilisant Sequitur est générée pour la séquence de symboles *cabcab*. Les étapes de construction de la grammaire sont détaillées dans la Table 2.2.

La première étape de l'algorithme consiste à créer une règle de départ, avec le non-terminal S comme entête et le premier symbole de *cabcab* comme corps.

Algorithme 1: Sequitur$(\Sigma, \sigma, \Gamma_0, \Delta_0)$

let S be a fresh nonterminal ($S \notin \Sigma \cup \Gamma_0$);
$G \leftarrow \langle \Sigma, \Gamma_0 \cup \{S\}, S, \Delta_0 \cup \{\langle S, \epsilon \rangle\} \rangle$;
for $i = 1$ **to** $|\sigma|$ **do**
    append σ_i to body of rule of S ;
    while $\neg$ *Uniqueness*(G) $\vee$ $\neg$ *Utility*(G) **do**
        if $\neg$ *Uniqueness*(G) **then**
            let δ be a repeated digram in G;
            if $\exists \langle A, \alpha \rangle \in \Delta : \alpha = \delta$ **then**
                replace the other occurrence of δ in G with A;
            else
                form new rule $\langle D, \delta \rangle$ where $D \notin (\Sigma \cup \Gamma)$;
                replace both occurrences of δ in G with D;
                $\Delta \leftarrow \Delta \cup \{\langle D, \delta \rangle\}$;
                $\Gamma \leftarrow \Gamma \cup \{D\}$;
            end
        else if $\neg$ *Utility*(G) **then**
            let $\langle A, \alpha \rangle \in \Delta$ be a rule used once;
            replace the occurrence of A with α in G;
            $\Delta \leftarrow \Delta \setminus \{\langle A, \alpha \rangle\}$;
            $\Gamma \leftarrow \Gamma \setminus \{A\}$;
        end
    end
end
return G;

Étape	i	Grammaire	Action	Ligne(s)	Unic.	Util.
1	1	$S \to c$	concaténation	4	true	true
2	2	$S \to ca$	concaténation	4	true	true
3	3	$S \to cab$	concaténation	4	true	true
4	4	$S \to cabc$	concaténation	4	true	true
5	5	$S \to cabca$	concaténation	4	**false**	true
6	–	$S \to BbB$ $B \to ca$	ajout règle	11–14	true	true
7	6	$S \to BbBb$ $B \to ca$	ajout	4	**false**	true
8	–	$S \to CC$ $B \to ca$ $C \to Bb$	ajout règle	11–14	true	**false**
9	–	$S \to CC$ $C \to cab$	suppression règle	17–20	true	true

TABLE 2.2 – Construction d'un grammaire pour *cabcab*

De l'étape 1 à l'étape 4 de la Table 2.2, l'algorithme fait de simples concaténations de symboles au corps de la règle de départ sans autre traitement spécifique. Compte tenu du fait que les deux propriétés *unicité du digramme* et *utilité des règles* sont respectées par la grammaire générée. Après la concaténation du symbole a dans le corps de la règle de départ $S \to cabc$, à l'étape 5 de la Table 2.2, la propriété *unicité du digramme* n'est plus maintenue, car le digramme ca apparaît deux fois. Sequitur crée une nouvelle règle $B \to ca$ avec un nouveau non-terminal B comme entête et le digramme ca comme corps, afin de remplacer les deux occurrences du digramme ca dans la grammaire avec B, comme il est illustré à l'étape 6 de la Table 2.2.

Après la concaténation du symbole b au corps de la règle de départ à l'étape 7 du tableau 2.2, la propriété *unicité du digramme* n'est plus maintenue par la grammaire. À l'étape 8, Sequitur introduit une nouvelle règle $C \to Bb$ dans la grammaire afin d'assurer la propriété *unicité du digramme*. Dans la grammaire qui en résulte, le non-terminal B n'apparaît qu'une seule fois dans le corps de la grammaire, uniquement dans le corps de la règle $C \to Bb$. Par conséquent, la propriété *utilité des règles* n'est plus maintenue par la grammaire. Dans ce cas, à l'étape 9, afin d'assurer la propriété *utilité des règles*, Sequitur élimine la règle $B \to ca$ et remplace l'apparition du non-terminal B dans la règle $C \to Bb$ par le digramme ca. Ainsi la grammaire générée comme sortie est bien valide parce qu'elle vérifie bien les deux propriétés *unicité du digramme* et *utilité des règles*.

2.4 Inconvénients des grammaires générées par Sequitur

Les grammaires générées en utilisant Sequitur ne sont pas particulièrement petites. De plus, pour générer une compression, Sequitur accorde la même importance à toutes les répétitions détectées. Or, dans notre contexte de travail, en raison de la nature cyclique de la plupart des programmes embarqués, les traces sont constituées de répétitions de séquences d'instructions. Il est donc important de distinguer deux types de répétitions : les *cycles* et les *non-cycles*, afin d'aider à l'analyse de trace, .

Les *cycles*, sont plus importants que les autres répétitions et méritent un traitement spécifique. Cela, parce que chaque cycle représente un comportement spécifique du microcontrôleur. Étant donné que *Sequitur* ne prend pas en considération l'importance de ces patrons, souvent, dans la grammaire qu'il génère, les cycles sont entremêlés, ce qui rend donc la détection d'un cycle en particulier coûteuse en temps.

Dans le but de fournir une compression plus pertinente pour l'analyse du comportement, nous proposons dans le chapitre suivant une extension de *Sequitur*. Dans la chapitre 4 nous proposons un algorithme inspiré de qui répond aux limitation de *Sequitur*, dont la détection de *cycles*.

Chapitre 3

Compression de traces d'exécution

Sommaire

3.1 Introduction

Un grand nombre des programmes embarqués sur les microcontrôleurs peuvent être classés comme des programmes *cycliques*. La nature cyclique des programmes s'explique par le fait qu'en l'absence de système d'exploitation, le programme est en charge de scruter les entrées de façon active. Ainsi, dans l'exemple illustré dans la Figure 3.1, le programme réagit en fonction de l'action effectuée par l'utilisateur (mouvement du JoyStick). Dans ce qui suit, nous appelons *tête de boucle* (en anglais loop-header) l'instruction qui définit la boucle principale. À chaque itération de la boucle principale, en fonction des données d'entrée reçues de l'environnement du microcontrôleur, des actions spécifiques sont exécutées. Une trace d'exécution générée par l'exécution d'un programme cyclique est dite *trace cyclique*.

Dans cette section, une méthode est proposée pour améliorer le processus d'analyse des traces cycliques. Pour cela, nous proposons une méthode de compression inspirée de *Sequitur*.

```
1  int main(void){
2    while(1){
3      static JOY_State_TypeDef JoyState = JOY_NONE;
4      static TS_STATE* TS_State;
5      JoyState = IOE_JoyStickGetState();
6      switch (JoyState){
7        case JOY_NONE:
8          LCD_DisplayStringLine(Line5, "JOY: ----");
9          break;
10       case JOY_UP:
11         LCD_DisplayStringLine(Line5, "JOY: UP");
12         break;
13       case JOY_DOWN:
14         LCD_DisplayStringLine(Line5, "JOY: DOWN");
15         break;
16       case JOY_LEFT:
17         LCD_DisplayStringLine(Line5, "JOY: LEFT");
18         break;
19       case JOY_RIGHT:
20         LCD_DisplayStringLine(Line5, "JOY: RIGHT");
21         break;
22       case JOY_CENTER:
23         LCD_DisplayStringLine(Line5, "JOY: CENTER");
24         break;
25       default:
26         LCD_DisplayStringLine(Line5, "JOY: ERROR");
27         break;
28     }
29     TS_State = IOE_TS_GetState();
30     Delay(1);
31     if (STM_EVAL_PBGetState(Button_KEY) == 0){
32       STM_EVAL_LEDToggle(LED1);
33       LCD_DisplayStringLine(Line4, "Pol: KEY Pressed");
34     }
35     if (STM_EVAL_PBGetState(Button_TAMPER) == 0){
36       STM_EVAL_LEDToggle(LED2);
37       LCD_DisplayStringLine(Line4, "Pol: TAMPER Pressed");
38     }
39     if (STM_EVAL_PBGetState(Button_WAKEUP) != 0){
40       STM_EVAL_LEDToggle(LED3);
41       LCD_DisplayStringLine(Line4, "Pol: WAKEUP Pressed");
42     }
43   }
44 }
```

FIGURE 3.1 – Exemple de code C pour une application embarquée

3.2 Préliminaires

Dans cette partie du document, nous proposons une extension de Sequitur, appelée *Cyclitur*, afin de compresser les traces d'exécution cycliques. *Cyclitur* compresse les répétitions consécutives et tire parti de la nature cyclique de la trace extraite d'un microcontrôleur. Par exemple, pour la séquence

cabcabcabcabcad, Sequitur génère la grammaire suivante :

$$S \to AABd$$
$$A \to CC$$
$$B \to ca$$
$$C \to Bb$$

alors que pour la même séquence, si on suppose que la tête de boucle est « *a* », *Cyclitur* génère la grammaire suivante :

$$S \to cA^4B$$
$$A \to abc$$
$$B \to ad$$

La séquence originale contient 15 symboles. *Sequitur* génère une grammaire qui contient 14 symboles, cela grâce au processus de compression qui a permis de capturer explicitement les répétitions de la séquence *cba*. Cependant, la structure cyclique de la trace est complètement invisible, c'est-à-dire qu'il n'est pas possible de dire si A ou B sont des cycles sans parcourir les règles de la grammaire. La grammaire générée par *Cyclitur* ne contient que 11 symboles, où chaque cycle de la séquence est représenté par un seul symbole (c, abc représenté par A, et ad représenté par B). Ceci permet de sauvegarder la structure cyclique de la trace.

Pour mieux comprendre la méthode de compression que nous proposons, nous commençons par définir les notions utilisées. Ensuite nous détaillerons l'algorithme *R-Sequitur*, notre amélioration de Sequitur utilisée par *Cyclitur*, ainsi que les propriétés des grammaires qu'il construit. Puis, dans la section 3.5 nous verrons comment exploiter l'aspect cyclique des traces d'exécution pour offrir une compression plus pertinente, ceci en utilisant notre algorithme *Cyclitur*.

3.2.1 Notions de base

Soit Σ un alphabet donné, une *r-chaîne* α est une séquence de paires (symbole, nombre de répétitions consécutives), c'est-à-dire, une séquence de paires $\langle symbole, entier\ strictement\ positif \rangle$. Par exemple $\langle A, 4\rangle\langle B, 1\rangle$. L'ensemble des *r-chaînes* sur Σ est $\Sigma_r^* = (\Sigma \times \mathbb{N} \setminus \{0\})^*$.

Soit α une r-chaîne :

- $\alpha_{i,1}$ représente le i-ème symbole de la séquence α.
- $\alpha_{i,2}$ représente le nombre de répétitions consécutives du i-ème symbole de la séquence α.

La r-chaîne correspondant à la séquence de symboles « $\sigma = AAAAB$ » est : « $\alpha = A^4B$ », où :

- $\alpha_{1,1} = A$
- $\alpha_{1,2} = 4$
- $\alpha_{2,1} = B$
- $\alpha_{2,2} = 1$

Le nombre d'éléments dans une *r-chaîne* α est noté $|\alpha|$. Pour alléger les notations, le numéro de répétition est placé comme exposant après le symbole et il est omis lorsqu'il est égal à un. Par exemple, A^4B est la représentation de la *r-chaîne* $\langle A, 4\rangle\langle B, 1\rangle$. Un *r-digramme* est une *r-chaîne* dont la taille est égale à deux.

3.2.2 R-Grammaire

Une r-grammaire est une grammaire permettant de définir une syntaxe pour une chaîne donnée en prenant en compte le nombre de répétition dans cette chaîne.

Formellement, une r-grammaire est un quadruplet $\langle \Sigma, \Gamma, S, \Delta\rangle$ tel que :

- Σ est un alphabet fini de symboles *terminaux*,
- Γ est un alphabet disjoint de Σ, fini de symboles *non-terminaux*,
- $S \in \Gamma$ est le *symbole de départ*, c.-à-d., un non-terminal particulier,
- $\Delta \subseteq \Gamma \times (\Sigma \cup \Gamma)_r^*$ est un ensemble fini de *r-règles de production* (ou plus simplement *r-règles*),

Une r-règle $\langle A, \alpha\rangle \in \Delta$ associe le non-terminal A à la r-chaîne α, respectivement appelés *entête* et *corps* de la r-règle. Le *corps de la r-grammaire* représente l'ensemble des corps des règles de la r-grammaire.

Afin d'illustrer les définitions introduites dans cette section, un exemple est présenté ci-dessous.

3.2.3 Exemple

Nous utiliserons comme exemple la grammaire présentée ci-dessous, générée par note approche pour la trace *abcabcabcabcad* :

$$S \to A^4B$$
$$A \to abc$$
$$B \to ad$$

L'ensemble de terminaux utilisés est $\Sigma = \{a, b, c, d\}$. Pour la construction de la grammaire trois nouveaux symboles sont utilisés, A, B, et le symbole de

départ S. L'ensemble de non-terminaux est donc $\Gamma = \{A, B, S\}$. L'ensemble des règles de production est $\Delta = \{S \rightarrow A^4B,\ A \rightarrow abc,\ B \rightarrow ad\}$. Par conséquent, le corps de la grammaire générée dans cet exemple est $Corps(G) = \{A^4B,\ abc,\ ad\}$.

3.3 R-Sequitur

A présent que les notions de bases nécessaires à la compréhension de notre extension de *Sequitur* ont été définies, dans cette section nous présentons notre algorithme *R-Sequitur*, ainsi que les propriétés de la r-grammaire qu'il génère.

Notre algorithme *R-Sequitur* est inspiré de Sequitur [61]. Il assure la vérification des propriétés *unicité du digramme*, *utilité des règles* et *absence de répétitions consécutives* sur la r-grammaire générée. Notre extension permet d'avoir une compression plus synthétique comparée à celle générée par Sequitur, comme nous l'expliquons par la suite.

Unicité du r-digramme : la propriété « unicité du digramme » veille à ce que le corps d'une r-grammaire ne continent pas deux occurrences du même r-digramme. Il important de rappeler que les deux occurrences du même r-digramme ne doivent pas s'entremêler (exemple de la Table 2.1).

Propriété 3. *La propriété « unicité du digramme », notée RUniqueness(G'), est vérifiée pour une r-grammaire G', si pour tout non-terminaux $A, B \in \Gamma$, symboles $a, b, c, d \in \Sigma \cup \Gamma$, entiers strictement positifs $n, m, p, q \in \mathbb{N} \setminus \{0\}$, et séquences de symboles $\alpha, \beta, \gamma, \delta \in (\Sigma \cup \Gamma)^*_r$, les deux assertions suivantes sont vérifiées :*

$$(A \neq B \wedge \{\langle A, \alpha\, a^n b^m\, \beta\rangle, \langle B, \gamma\, c^p d^q\, \delta\rangle\} \subset \Delta) \Rightarrow a^n b^m \neq c^p d^q \quad (3.1)$$

$$(\langle A, \alpha\, a^n b^m\, \beta\, c^p d^q\, \gamma\rangle \in \Delta) \implies a^n b^m \neq c^p d^q \quad (3.2)$$

L'assertion (3.1) exprime que si une r-grammaire G' contient deux r-règles $A \rightarrow \alpha\, a^n b^m\, \beta$ et $B \rightarrow \gamma\, c^p d^q\, \delta$, avec $A \neq B$, alors $a^n b^m$ et $c^p d^q$ sont deux motifs (r-digrammes) différents, c'est-à-dire que, $a \neq c \vee b \neq d \vee n \neq p \vee m \neq q$. L'assertion (3.2) exprime que si une r-grammaire G' contient une r-règle $A \rightarrow \alpha\, a^n b^m\, \beta\, c^p d^q\, \gamma$, alors $a^n b^m$ et $c^p d^q$ sont deux motifs (r-digrammes) différents.

Par exemple, si le symbole a^3 apparaît à la fin de la séquence $c^2a^3bc^2$, la propriété « unicité du digramme » n'est plus maintenue par la grammaire, car le r-digramme c^2a^3 apparaît deux fois dans le corps de la

grammaire. Une nouvelle règle est alors construite pour avoir la grammaire suivante :

$$S \rightarrow AbA$$
$$A \rightarrow c^2a^3$$

Utilité des règles : La propriété « utilité des règles » assure que chaque règle et plus précisément le non-terminal représentant son entête, à l'exception du symbole de départ S, soit utilisée au moins deux fois dans le corps de la r-grammaire. Cette propriété est définie formellement comme suit :

Propriété 4. *La propriété « utilité des règles », notée RUtility(G), est vérifiée pour une r-grammaire G' si et seulement si :*

$$\forall A \in \Gamma \setminus \{S\} : \left| \sum_{\langle B,\beta\rangle \in \Delta} \sum_{i \in [1..|\beta|]} \begin{cases} \beta_{i,2} & si\ \beta_{i,1} = A \\ 0 & si\ \beta_{i,1} \neq A \end{cases} \right| \geq 2 \tag{3.3}$$

Si le symbole b apparaît à la fin de la séquence $c^2a^3bc^2a^3$, la propriété « unicité du digramme » n'est plus maintenue par la grammaire, car le r-digramme Ab, où $A \rightarrow c^2a^3$, apparaît deux fois dans le corps de la grammaire($S \rightarrow AbAb$). Une nouvelle règle est alors construite pour avoir la grammaire suivante :

$$S \rightarrow BB$$
$$A \rightarrow c^2a^3$$
$$B \rightarrow Ab$$

Dans la grammaire générée, le non-terminal A n'est utilisé qu'une fois dans le corps de la grammaire, et la propriété « utilité des règles » n'est donc plus maintenue par la grammaire. Par conséquent, la règle dont l'entête est le non terminal A est supprimée, et l'occurrence du non-terminal A est alors remplacée par le corps de la règles supprimée pour avoir la grammaire suivante :

$$S \rightarrow BB$$
$$B \rightarrow c^2a^3b$$

Pas de répétitions consécutives : Outre les propriétés « unicité du digramme » et « utilité des règles », l'algorithme *R-Sequitur* assure une autre propriété qui vérifie que tout r-digramme dans le corps de la r-grammaire se compose de différents symboles. Cette propriété est appelée « *pas de répétitions consécutives* ».

Propriété 5. *Une r-grammaire G' vérifie la propriété « pas de répétitions consécutives », notée RConsecutive(G'), si et seulement si :*

$$\begin{gathered}\forall a, b \in \Sigma \cup \Gamma, \forall n, m \in \mathbb{N} \setminus \{0\}, \forall \alpha, \beta \in (\Sigma \cup \Gamma)_r^*, \forall C \in \Gamma : \\ \langle C, \alpha\, a^n b^m \beta \rangle \in \Delta \Rightarrow a \neq b\end{gathered} \tag{3.4}$$

Pour assurer cette propriété, R-Sequitur fusionne tous les digrammes de la forme $a^n a^m$ en un symbole unique de la forme a^{n+m}.

Dans la grammaire générée précédemment, nous avons $S \leftarrow BB$, donc la propriété « pas de répétitions consécutives » n'est pas maintenue. Par conséquent la grammaire suivante est générée :

$$S \rightarrow B^2$$
$$B \rightarrow c^2 a^3 b$$

L'algorithme étendant de *Sequitur*, appelé *R-Sequitur*, est introduit dans l'Algorithme 2. Cette extension consiste à améliorer la grammaire générée comme compression pour qu'elle soit plus facile et plus intuitive à comprendre. Cela en différenciant les répétitions des digrams de même symbole (*aaaa*), des digrams de symboles différents (*abacab*).

3.4 Exemple

Dans cet exemple, une compression grammaticale en utilisant R-Sequitur est générée pour la séquence de symboles *cabcab*. Les étapes de construction de la grammaire sont détaillées dans la Table 3.1. Les lignes citées dans cette exemple font référence à l'Algorithme 2.

La première étape de l'algorithme consiste à créer une règle de départ, avec le non-terminal S comme entête et le premier symbole de *cabcab* comme corps. De l'étape 1 à l'étape 4 de la Table 3.1, l'algorithme fait de simples ajouts de symboles au corps de la règle de départ sans aucun traitement spécifique. Compte tenu du fait que les trois propriétés *unicité du digramme*, *utilité des règles* et *pas de répétitions consécutives* sont respectées par la grammaire générée.

Après l'ajout du symbole a dans le corps de la règle de départ $S \rightarrow cabc$, à l'étape 5 de la Table 3.1, la propriété *unicité du digramme* n'est plus maintenue, car le digramme *ca* apparaît deux fois. R-Sequitur crée une nouvelle règle $B \rightarrow ca$ avec un nouveau non-terminal B comme entête et le digramme *ca* comme corps, afin de remplacer les deux occurrences du digramme *ca* dans la grammaire avec B, comme il est illustré à l'étape 6 de la Table 3.1.

Algorithme 2: Function $\mathsf{R} - \mathsf{Sequitur}(\Sigma, \sigma, \Gamma_0, \Delta_0)$

```
let S be a fresh nonterminal representing a rule (S ∉ Σ ∪ Γ0);
G ← ⟨Σ, Γ0 ∪ {S}, S, Δ0 ∪ {⟨S, ϵ⟩}⟩;
for i ← 1 to |σ| do
    append (σ_i)^1 to body of rule S ;
    while ¬RUniqueness(G) ∨ ¬RUtility(G) ∨ ¬RConsecutive(G) do
        if ¬RConsecutive(G) then
            let a, n, m be s.t. a^n a^m is a r-digram in G;
            replace every occurrence of a^n a^m in G with a^{n+m};
        else if ¬RUniqueness(G) then
            let δ be a repeated r-digram in G;
            if ∃⟨A, α⟩ ∈ Δ : α = δ then
                replace the other occurrence of δ in G with A;
            else
                form new rule ⟨D, δ⟩ where D ∉ (Σ ∪ Γ);
                replace both occurrences of δ in G with D;
                Δ ← Δ ∪ {⟨D, δ⟩};
                Γ ← Γ ∪ {D};
            end
        else if ¬RUtility(G) then
            let ⟨A, α⟩ ∈ Δ be a rule used once;
            replace the occurrence of A with α in G;
            Δ ← Δ ∖ {⟨A, α⟩};
            Γ ← Γ ∖ {A};
        end
    end
end
return G;
```

Après l'ajout du symbole b au corps de la règle de départ à l'étape 7 du tableau 3.1, la propriété *unicité du digramme* n'est plus maintenue par la grammaire. À l'étape 8, Sequitur introduit une nouvelle règle $C \rightarrow Bb$ dans la grammaire afin d'assurer la propriété *unicité du digramme*.

Dans la grammaire qui en résulte, le non-terminal B n'apparaît qu'une

Étape	i	Grammaire	Action	Ligne(s)	Unic.	Util.	Conse.
1	1	$S \to c$	concaténation	4	true	true	true
2	2	$S \to ca$	concaténation	4	true	true	true
3	3	$S \to cab$	concaténation	4	true	true	true
4	4	$S \to cabc$	concaténation	4	true	true	true
5	5	$S \to cabca$	concaténation	4	**false**	true	true
6	–	$S \to BbB$ $B \to ca$	ajout règle	9–12	true	true	true
7	6	$S \to BbBb$ $B \to ca$	concaténation	4	**false**	true	true
8	–	$S \to CC$ $B \to ca$ $C \to Bb$	ajout règle	9–12	true	**false**	true
9	–	$S \to CC$ $C \to cab$	suppression règle	19–23	true	true	**false**
10	–	$S \to C^2$ $C \to cab$	suppression répétition	6–8	true	true	true

TABLE 3.1 – Construction de grammaire pour *cabcab* en utilisant R-Sequitur

seule fois dans le corps de la grammaire, uniquement dans le corps de la règle $C \to Bb$. Par conséquent, la propriété *utilité des règles* n'est plus maintenue par la grammaire. Dans ce cas, à l'étape 9, afin d'assurer la propriété *utilité des règles*, R-Sequitur élimine la règle $B \to ca$ et remplace l'apparition du non-terminal B dans la règle $C \to Bb$ par le digramme ca.

À l'étape 10, afin d'assurer la propriété *pas de répétitions consécutives* le digrame CC est remplacé par le nombre de répétition et le non terminal C, c'est-à-dire C^2. Ainsi la grammaire générée comme sortie est bien valide parce qu'elle vérifie bien les trois propriétés *unicité du digramme*, *utilité des règles* et *pas de répétitions consécutives*.

3.5 Compression des traces cyclique avec *Cyclitur*

Étant donné une trace cyclique σ, et une tête de boucle lh, l'ensemble des cycles $C(\sigma, lh)$ est un ensemble de paires d'indices sur la trace d'exécution σ.

Algorithme 3: Function $\mathsf{Cyclitur}(\Sigma, \sigma, \Gamma_0 = \varnothing, \Delta_0 = \varnothing, lh)$

```
ω' ← ϵ;
foreach ⟨i, j⟩ ∈ C(σ, lh) do
    ⟨Σ', Γ', S', Δ'⟩ ← ReSequitur(Σ, σ_{i..j}, Γ_0, Δ_0);
    Γ_0 ← Γ';
    Δ_0 ← Δ';
    σ' ← σ' · S';
end
⟨Σ'', Γ'', S'', Δ''⟩ ← ReSequitur(Σ, ω', Γ_0, Δ_0)
return ⟨Σ'', Γ'', S'', Δ''⟩;
```

Cet ensemble de cycles est défini formellement comme suit :

$$\begin{aligned} C(\sigma, lh) = \{\langle i, j\rangle \in [1..|\sigma|]^2 \mid i \leq j \wedge (i = 0 \vee \sigma_i = lh) \\ \wedge (j = |\sigma| \vee \sigma_{j+1} = lh) \\ \wedge \forall k \in [i+1..j] : \sigma_k \neq lh\} \end{aligned} \tag{3.5}$$

Pour compresser une trace dont les cycles sont identifiés, l'idée est d'appliqué R-Sequitur sur chaque cycle afin de détecter les répétitions à l'intérieur des cycles. Puis l'application de R-Sequitur sur la compression générée par l'étape précédente permet la détection des séquences de cycles similaires dans la trace. C'est le principe de notre algorithme Cyclitur, présenté dans Algorithme 3, et implémenté dans notre outil *CoMET*.

Les résultat de l'évaluation de notre approche, page 54, nous ont permis d'observer que Cyclitur fournit de meilleurs résultats de compression que Sequitur sur les traces cycliques, mais permet également une meilleure compréhension de la trace. Ce résultat de compréhension s'explique par le fait que la compression générée par Sequitur ne différencie pas les cycles. De ce fait, souvent, dans la grammaire générée, les cycles sont entremêlés, comme dans l'exemple de la Figure 3.2. Ceci rend donc la détection d'un cycle particulier coûteuse en nombre de branches à parcourir. Avec Cyclitur, chaque cycle est représenté par un symbole unique, par conséquent, le nombre de branches parcourues est toujours égale à 1, comme illustré dans la Figure 3.3.

Pour l'exemple de la Figure 3.2, Sequitur génère pour la séquence *cabcabc-*

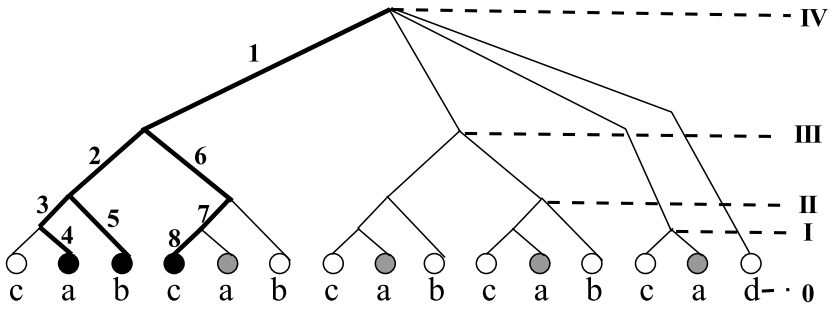

FIGURE 3.2 – Arbre de construction de grammaire pour *cabcabcabcabcad* avec Sequitur

Étape	Branche	Cycle	Niveau
1	$S \rightarrow \underline{A}ABd$	ϵ	de IV à III
2	$A \rightarrow \underline{C}C$	ϵ	de III à II
3	$C \rightarrow \underline{B}b$	ϵ	de II à I
4	$B \rightarrow c\underline{a}$	a	de I à 0
5	$C \rightarrow B\underline{b}$	ab	de II à 0
6	$A \rightarrow C\underline{C}$	ab	de III à II
7	$C \rightarrow \underline{B}b$	ab	de II à I
8	$B \rightarrow \underline{c}a$	abc	de I à 0

TABLE 3.2 – Le chemin parcouru pour détecter le premier cycle *abc* -Sequitur-

abcabcad la grammaire suivante :

$$S \rightarrow AABd$$
$$A \rightarrow CC$$
$$B \rightarrow ca$$
$$C \rightarrow Bb$$

Soit le symbole a défini comme loop-header. Pour détecter le premier cycle *abc* de la séquence *cabcabcabcabcad*, à partir de la grammaire ; et en utilisant l'arbre de construction illustré dans la Figure 3.2, il faut parcourir huit branches. Les étapes parcourues sont listées dans la Table 3.2. En commençant avec un nombre de branches égale à zéro, la première étape consiste à analyser le premier symbole de la règle $S \rightarrow \underline{A}ABd$. Cela se traduit dans l'arbre de construction de grammaire de la Figure 3.2, par le passage du niveau IV au niveau III, et donc le nombre de branches est incrémenté de un. La seconde étape effectue un passage du niveau III au niveau II dans le but d'analyser le premier symbole de la règle du non-terminal A ($A \rightarrow \underline{C}C$). Par conséquent, le nombre de branches est encore incrémenté de un et vaut à présent deux.

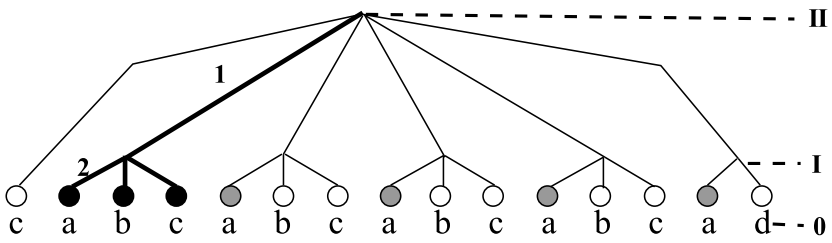

FIGURE 3.3 – Arbre de construction de grammaire pour *cabcabcabcabcad* avec Cyclitur

Ainsi, à chaque passage d'un niveau supérieur à un niveau inférieur le nombre de branches est incrémentée de un, jusqu'à la détection complète du premier cycle *abc*, où elle sera égale à huit.

Il est important de préciser qu'afin d'aider à la compréhension de la trace cyclique, une bonne compression avec une bonne détection de cycles est nécessaire. Cela permet d'avoir une description pertinente de l'ensemble de la trace et aide ainsi à l'identification des cycles (événements) importants. La pertinence de la compression dépend donc du bon découpage de la trace en cycles, ceci se traduit par une bonne détection du loop-header (tête de la boucle principale). La méthode utilisée pour la détection de la tête de boucle est présentée ci-dessous.

Chapitre 4

Détection de cycles dans les traces d'exécution

Sommaire

4.1 Introduction

L'aspect cyclique est la principale caractéristique des programmes à l'origine des traces traitées dans notre travail. Cette caractéristique est souvent exprimée à travers une boucle principale active, en utilisant dans la plupart des cas l'instruction *while(**1**)*, ou *while(**true**)*.

Comme nous l'avons dis plutôt dans cette thèse, la division de la trace en cycles repose sur la localisation du loop-header, noté lh. Cependant, la valeur du loop-header n'est pas toujours connue car, il reste très difficile de faire le lien entre le code source et la trace d'exécution à cause des optimisations apportées lors de la compilation.

Ce chapitre porte donc sur la mise en œuvre de stratégies permettant d'identifier au mieux le loop-header dans la trace. Pour cela, nous proposons une approche qui consiste à calculer un score de vraisemblance pour chaque instruction. Plus une instruction aura un score élevé, plus grande sera la probabilité que celle-ci sera le loop-header.

Différentes stratégies ont été explorées. Pour chaque stratégie, une métrique de calcul de score est définie. Ces stratégies permettent d'avoir une liste de candidats classés par ordre décroissant, selon leurs scores. Le candidat au premier rang est ainsi sélectionné comme étant le loop-header. Les trois stratégies explorées, présentées ci-dessous, utilisent respectivement pour le calcul du score :

- Le nombre d'occurrences,
- Le nombre d'occurrences et la distance entre les occurrences,
- Le nombre d'occurrences, la distance entre les occurrences et l'indice de la première occurrence.

Il est important de préciser que notre approche de détection de cycles est définie pour les traces d'exécution dites cycliques, c'est-à-dire qu'elles sont générées à partir de programmes conforme au modèle **while**. La figure 4.1 illustre le code du **while** modèle.

```
s1();
while(true){
  s2();
}
s3();
```

FIGURE 4.1 – Modèle de programme **while**

Nous utilisons s1() pour modéliser le fait que des instructions peuvent précéder la boucle **while**(true), comme l'initialisation des variables par exemple. Nous utilisons s3() pour modéliser le fait que la boucle **while**(true) peut être suivie d'instructions, pour un traitement d'exception par exemple. Nous utilisons s2() pour représenter les instructions qui se trouvent à l'intérieur de la boucle **while**(true). Il est important de noter que (s1),(s2) et (s3) peuvent contenir une ou plusieurs instructions conditionnelles (**if**, **switch**) et/ou boucles (**for**, **while**). Toutefois, si ces boucles sont présentes dans s1, s2, ou s3, elles ne sont pas infinies et sont supposées terminer.

4.2 Nombre d'occurrences

Une première idée pour identifier le loop-header consiste à calculer la fréquence d'occurrence de chaque symbole et à considérer que le loop-header est le plus fréquent, car l'instruction va être exécutée à chaque cycle. Cette sous-section formalise et évalue cette hypothèse.

Soit un symbole k dans la trace σ. Le nombre d'occurrence de k dans la trace σ, noté $Na(k, \sigma)$, est défini comme suit :

$$Na(k,\sigma) = \sum_{i=1}^{|\sigma|} x_i \text{ où } \begin{cases} x_i = 1 & \text{si } \sigma_i = k, \\ x_i = 0 & \text{si } \sigma_i \neq k. \end{cases} \tag{4.1}$$

Score et classement : Cette métrique est un calcul de score, où le symbole avec la plus grande valeur est plus susceptible à être le loop-header. Soit la fonction de calcul du score $\mathsf{score}_1(k, \sigma)$ définie comme suit :

$$\mathsf{score}_1(k, \sigma) = Na(k, \sigma)\,.$$

En utilisant le résultat du calcul de score, il est possible de classer les symboles en fonction de leurs probabilités d'être le loop-header. Ainsi, pour toute fonction de score s estimant la probabilité d'un symbole à être le loop-header, il est possible de définir le *classement* d'un symbole k, noté $\mathsf{rank}(s, k, \sigma)$, comme suit :

$$\mathsf{rank}(s, k, \sigma) = \left|\{l \in \Sigma \mid s(l, \sigma) \geq s(k, \sigma)\}\right|. \tag{4.2}$$

Le classement du symbole k est donc le nombre de symboles qui ont un score plus grand ou égal au score de k. Soit $\mathsf{rank}_1(k, \sigma)$ définit comme synonyme à $\mathsf{rank}(\mathsf{score}_1, k, \sigma)$.

Limites et contre-exemple : En se basant sur le modèle de programme **while** (Figure 4.1), cette métrique semble logique. La condition **while**(true) est l'instruction la plus exécutée dans le programme. Cependant, dans certain cas, le symbole (événement) sélectionné n'est pas le bon loop-header. En effet, certains événements dans la boucle principale peuvent être exécutés plusieurs fois à chaque exécution de la boucle.

Prenons l'exemple de la Figure 4.2. En utilisant la métrique actuelle, l'événement sélectionné comme loop-header représente la boucle **for** dans le programme, plutôt que de l'en-tête de la boucle **while**(1). En effet, si la boucle **while** est exécutée n fois, son symbole correspondant apparaîtra n fois dans la trace. Cependant, la boucle **while** contient une boucle **for**, qui est exécutée cinq fois à chaque itération de la boucle **while**. Ainsi, le symbole correspondant à la boucle **for** apparaîtra $n \times 5$ fois dans la trace. Par conséquent, le fait de prendre en compte uniquement le nombre d'occurrences d'un symbole, dans une trace d'exécution, pour la localisation du loop-header est insuffisant.

```
int i, j;
while (1) {
  j = 1;
  for (i = 0; i < 5; i++)
     j = j*2;
}
```

FIGURE 4.2 – Programme 1

4.3 Distance entre occurrences

On constate sur le programme 1 que la boucle principale du programme cyclique a tendance à avoir des itérations plus longues que celles des autres boucles. Cela correspond au fait que la boucle **while** *englobe* le code.

Pour détecter le loop-header, une idée consiste donc à utiliser le nombre d'occurrences et les distances entre ces occurrences. Pour prendre en compte les distances entre les occurrences d'un symbole dans une même trace, nous utilisons la moyenne de la distance entre les occurrences consécutives d'un symbole particulier.

Soit $x_1, x_2, x_3, \ldots, x_n$ les index d'occurrences d'un symbole k dans la trace d'exécution σ, où $\forall i, j \in [1..|\sigma|] : i \leq j \Rightarrow x_i \leq x_j$. La moyenne des distances entre ses occurrences *consécutives*, notée $Da(k, \alpha)$, est définie comme suit :

$$\begin{aligned} Da(k,\sigma) &= \frac{(\sigma_{x_2} - \sigma_{x_1}) + (\sigma_{x_3} - \sigma_{x_2}) + \cdots + (\sigma_{x_n} - \sigma_{x_{n-1}})}{n-1} \\ &= \frac{\sum_{i=1}^{Na(k)-1} \sigma_{x_{i+1}} - \sigma_{x_i}}{Na(k,\sigma) - 1} \end{aligned} \tag{4.3}$$

Score et classement : La moyenne des distances entre les occurrences, notée Da, est combinée avec le nombre d'occurrences d'un symbole dans la trace, noté Na. L'heuristique de localisation du loop-header detection consiste en la considération du symbole avec la plus grande valeur de $(Na \times Da)$.

De la même manière que score_1 et rank_1, le score et le classement utilisant cette métrique, notés respectivement score_2 et rank_2, sont définis comme suit :

$$\mathsf{score}_2(k,\sigma) = Na(k,\sigma) \times Da(k,\sigma) \tag{4.4}$$

$$\mathsf{rank}_2(k,\sigma) = \mathsf{rank}(\mathsf{score}_2, k, \sigma) \tag{4.5}$$

Limite et contre-exemple : D'après les résultats de notre évaluation, page 51, L'utilisation de la seconde métrique pour la localisation du loop-header dans la trace d'exécution fournit de meilleurs résultats que la première métrique, qui utilise uniquement le nombre d'occurrences pour l'identification du loop-header. Sur les différentes traces analysées dans cette évaluation, cette métrique a permis l'identification du bon loop-header dans plus de 57% des cas. Néanmoins, dans le cas où plusieurs symboles ont le même nombre d'occurrences Na et la même moyenne de la distance entre les occurrences Da, le loop-header sélectionné peut ne pas être le bon.

Une trace d'exécution générée par le programme 2 (Figure 4.3) contient le même nombre d'occurrences n de symboles représentants :

```
int i = 1, j = 2;
while (1) {
  j = i*2;
  i = j*3;
}
print(j);
```

FIGURE 4.3 – Programme 2

- `while(1)`
- `j = i*2`
- `i =j*3`

Pour simplifier le calcul, la distance entre les occurrences est mesurée directement à partir du programme de la Figure 4.3. Ainsi, la distance entre deux occurrences de `while(1)` est égale à 2 (`j = i*2` et `i = j*3`). Il est important de noter que la distance entre deux occurrences de `j = i*2` vaut également 2 (`i = j*3` et `while (true)`), pareillement pour la distance entre deux occurrences de `i = j*2`. Par conséquent, le score_2 vaut :

- $(n \times 2)$ pour `while(1)`
- $(n \times 2)$ pour `j = i*2`
- $(n \times 2)$ pour `i = j*3`

Étant donné que certains symboles peuvent avoir le même score, l'utilisation de la deuxième métrique de localisation du loop-header dans la trace ne permet pas de lever les ambiguïtés. C'est pourquoi nous proposons une troisième métrique.

4.4 Première occurrence

Il est important de noter qu'en utilisant le modèle de programme *While*, illustré dans la Figure 4.1, la première occurrence de l'instruction $while(true)$ apparaît avant la première occurrence de $s2()$ dans la trace d'exécution. Par conséquent, afin d'avoir une meilleure localisation du loop-header dans la trace, la métrique finale que nous proposons, prend en compte l'index de la première occurrence d'un symbole.

L'index de la première occurrence d'un symbole k dans la trace σ, notée

$\mathsf{first}(k, \sigma)$ est définie comme suit :

$$\mathsf{first}(k, \sigma) = \min\{i \in [0..|\sigma|] \mid \sigma_i = k\}\,.$$

Score et classement : La métrique finale que nous proposons calcule le score en combinant le nombre d'occurrences, la moyenne des distances entre les occurrences et l'index de la première occurrence. Cette métrique est notée score_3, est définie comme suit :

$$\mathsf{score}_3(k, \sigma) = \frac{Na(k, \sigma) \times Da(k, \sigma)}{\mathsf{first}(k, \sigma)}$$

Le classement de cette métrique est noté rank_3 est se définit pour un symbole k comme suit :

$$\mathsf{rank}_3(k, \sigma) = \mathsf{rank}(\mathsf{score}_3, k, \sigma)$$

Pour calculer les candidats et identifier le loop-header dans la trace nous utilisons l'Algorithme 4. Cet algorithme a été implanté dans notre outil CoMET. En utilisant cet algorithme, notre outil propose une liste ordonnée de candidats, ce qui permet à l'utilisateur de choisir un loop-header différent de celui désigné comme étant le plus probable par $score_3$.

Algorithme 4: Function Detection_lh(σ)

```
1  seen ← ∅;
2  candidates ← ∅;
3  for i ← 1 to |σ| do
4  |  if σ_i ∉ seen then
5  |  |  candidates ← candidates ∪ {(σ_i, score_3(σ_i, σ)};
6  |  |  seen ← seen ∪ {σ_i};
7  |  end
8  end
9  candidates ← Sort_Desc(candidates);
10 return candidates;
```

Algorithme : Tel qu'illustré dans l'Algorithme 4, notre algorithme analyse les symboles d'une séquence σ, de gauche à droite, en calculant le degré de correspondance au $loop-header$ pour chaque symbole. L'ensemble $seen$ contiens les symboles pour lesquels les degrés de correspondance ont déjà été calculés. Ainsi, même si un symbole σ_i se répète plusieurs fois dans la trace d'exécution, son degré de correspondence $\mathsf{score}_3(\sigma_i, \sigma)$ ne sera calculé qu'une seule

fois. L'algorithme retourne comme résultat un ensemble de candidats, classé dans ordre décroissant selon leur valeur de correspondance au loop-header.

L'utilisation de l'Algorithme 4 basé sur le $score_3$, nous a permis d'obtenir une identification correcte pour 100% des cas, sur les 27 traces étudiées. Comme on peut le voir chaiptre 5, section 5.1.

Optimisation : Afin d'accélérer le processus de détection du loop-header, des bornes inférieures peuvent être fixées pour les deux métriques, Na et Da. Dans l'évaluation présentée chapitre 5, les bornes inférieures suivantes : $Na > 10$ et $Da > 20$, nous ont permis de réduire le nombre d'événements candidats à être le loop-header, en ne considérant que les événements qui apparaissent au moins 11 fois dans la trace, avec une distance moyenne supérieure à 20 symboles (événements).

Chapitre 5

Évaluations

Sommaire

Dans le chapitre précédent nous avons défini trois métriques pour identifier le loop-header. Ce loop-header servira à la détection de cycles dans la trace. Dans le but d'évaluer la pertinence des stratégies utilisant ces trois métriques à identifier le loop-header, une évaluation expérimentale a été menée, section 5.1.

La deuxième évaluation, présentée dans la section 5.2, nous a permis de mesurer la capacité de notre approche à offrir une compression représentative de la trace tout en conservant son aspect cyclique.

5.1 Détection de la tête de boucle

Une évaluation expérimentale a été menée dans le but de vérifier la pertinence de la métrique définie pour la détection automatique du $loop - header$ dans une trace d'exécution.

Une trace d'exécution d'un microcontrôleur est constituée principalement de compteurs ordinaux (Figure 6, page 7). Par conséquent, cette évaluation consiste à comparer le compteur ordinal choisi par notre approche comme loop-header avec le compteur ordinal identifié par l'oracle. L'oracle dans notre évaluation expérimentale est un ingénieur. L'oracle analyse la trace d'exécution et le code source pour chaque programme utilisé dans l'évaluation expérimentale afin d'identifier le loop-header.

5.1.1 Métriques de l'évaluation expérimentale

Pour chaque programme utilisé dans l'évaluation expérimentale, le loop-header (compteur ordinal) proposé par notre approche est comparé avec le loop-header réel. Si c'est le même, la localisation du loop-header est alors un succès. Sinon, s'ils sont différents, nous analysons le classement proposé par notre approche afin de déterminer le rang du loop-header dans la liste des candidats.

Notre objectif est d'aider dans l'analyse de la trace. Ainsi, dans le but de minimiser l'effort fourni par l'ingénieur concernant l'analyse des candidats proposés par notre approche comme loop-header, nous avons limité l'analyse aux trois premiers candidats. Cette décision se justifie par les résultats de nos expérimentations, où nous avons observé que si le loop-header n'est pas classé parmi les premiers, il se trouve souvent loin dans le classement. Par conséquent, la stratégie utilisée perd en pertinence parce que l'effort fournit pour l'identification du loop-header grandit. Ainsi, si le rang du loop-header est inférieure ou égale à 3, la localisation du loop-header est considérée comme un succès (Tables 5.1, 5.2, 5.3). Dans le cas où le rang du loop-header est supérieur à 3, la localisation du loop-header est alors un échec (Table 5.4).

Rank	PC	Oracle	Verdict
1	0x08000CA0	•	Succès
2	0x08000D74		
3	0x08000DC8		
4	0x08000DCC		

TABLE 5.1 – Localisation du loop-header avec un effort minimal

Rank	PC	Oracle	Verdict
1	0x08000CA0		
2	0x08000D74	•	Succès
3	0x08000DC8		
4	0x08000DCC		

TABLE 5.2 – Localisation du loop-header avec un effort moyen

5.1.2 Programmes et traces d'exécution

Pour étudier la pertinence et la précision de notre approche de détection de loop-header, nous avons analysé 14 traces d'exécution provenant de 14

Rank	PC	Oracle	Verdict
1	0x08000CA0		
2	0x08000D74		
3	0x08000DC8	•	Succès
4	0x08000DCC		

TABLE 5.3 – Localisation du loop-header avec un effort important

Rank	PC	Oracle	Verdict
1	0x08000CA0		
2	0x08000D74		
3	0x08000DC8		
4	0x08000DCC	•	Echec

TABLE 5.4 – Échec de la localisation du loop-header

programmes java, notés C_i, où i indique le numéro du programme. Nous avons également évalué notre approche sur 13 traces provenant de 13 programmes embarqués sur microcontrôleur. Chacun des programmes étudiés contient une boucle principale **while**(true) exécutée tout au long de l'exécution.

La Table 5.5 contient les informations concernant les programmes java et leurs traces d'exécution. Les colonnes *Code* et *Trace* représentent respectivement le nombre de lignes du code source et de la trace d'exécution. Il est important de rappeler que le critère de sélection de ces programmes était l'aspect cyclique.

La Table 5.6 contient les informations concernant les programmes embarqués ainsi que leurs traces d'exécution. Ces programmes embarqués sont décrits en détail chapitre 10, section 10.1, page 101. Les colonnes *Taille* et *Trace* représentent respectivement la taille du programme et le nombre de lignes de la trace d'exécution. Ces traces ont des résidus avant le premier cycle et sont terminées par des crashs.

Il est important de préciser que pour chaque programme de notre évaluation, une seule trace d'exécution a été utilisée. La raison est que, pour un programme donné, le compteur ordinal représentant le loop-header dans ses traces d'exécution reste le même.

Avant d'évaluer notre approche, l'oracle analyse chaque programme ainsi que sa trace d'exécution pour identifier le loop-header. Ensuite, notre approche de détection du loop-header est exécutée sur chaque trace. Afin d'évaluer notre approche, le compteur ordinal (PC) sélectionné est comparé avec le loop-header défini par l'oracle, permettant ainsi de déterminer si l'identification est correcte ou pas. Il est important de rappeler que l'oracle dans notre évaluation

Prog.	Code	Fonction	Trace
C_1	154	Afficher le fuseau horaire d'une ville	24747
C_2	162	Comparer des fichiers texte	614391
C_3	358	Ordonnancer des processus	11593
C_4	254	Simuler un conte bancaire	5110
C_5	222	Remplir une « progress bar »	5789
C_6	61	Créer et supprimer des fichiers	2681
C_7	126	Simuler une horloge	2000
C_8	108	Mélanger et distribuer des cartes	11797
C_9	87	Compter le nombre de maj., min., num.	6701
C_{10}	146	Afficher la liste des utilisateurs d'un réseaux	2740
C_{11}	82	Parser un nombre	924
C_{12}	61	Trier une liste d'entier	5471
C_{13}	31	Manipuler une pile	1702
C_{14}	74	Mettre en attente une application	802

TABLE 5.5 – Informations concernant les programmes Java et leurs traces d'exécution

Prog.	Taille	Fonction	Trace
P_1	14.4 Mo	Manipuler la pile des appels	1048579
P_2	14.4 Mo	Manipuler la pile des appels	1045869
P_3	143 Mo	Manipuler les fonctions de comparaison	280049
P_4	218 Mo	Manipuler les adresses mémoire	237062
P_5	143 Mo	Manipuler les LEDs	207914
P_6	140 Mo	Manipuler les fichiers textes	240829
P_7	207 Mo	Manipuler les LEDs	1048577
P_8	139 Mo	Manipuler des casts	235788
P_9	218 Mo	Manipuler des fonctions mathématiques	241404
P_{10}	143 Mo	Manipuler des boucles	280298
P_{11}	207 Mo	Manipuler les LEDs et l'écran LCD	1048576
P_{12}	14.4 Mo	Manipuler la pile des appels	1048573
P_{13}	50.1 Mo	Manipuler le chien de garde	1047568

TABLE 5.6 – Informations concernant les programmes embarqués et leurs traces d'exécution

expérimentale est un ingénieur.

P	score_1		score_2		score_3	
	$\mathsf{rank}_1(lh,\sigma)$	$\mathsf{Verdict}_1$	$\mathsf{rank}_2(lh,\sigma)$	$\mathsf{Verdict}_2$	$\mathsf{rank}_3(lh,\sigma)$	$\mathsf{Verdict}_3$
C_1	62	–	22	–	1	•
C_2	27	–	12	–	1	•
C_3	76	–	2	•	1	•
C_4	1	•	1	•	1	•
C_5	19	–	16	–	1	•
C_6	4	–	1	•	1	•
C_7	1	•	1	•	1	•
C_8	7	–	7	–	1	•
C_9	4	–	1	•	1	•
C_{10}	3	•	1	•	1	•
C_{11}	1	•	5	–	1	•
C_{12}	13	–	5	–	1	•
C_{13}	3	•	1	•	1	•
C_{14}	1	•	1	•	1	•
S	6		8		14	
E	8		6		0	

TABLE 5.7 – Résultats de l'évaluation des traces Java

5.1.3 Résultats

La Table 5.7 contient les résultats de l'évaluation expérimentale, où chaque ligne i dans la table représente la trace du programme C_i. L'évaluation menée concerne trois scores, représentés dans la table par : score_1, score_2 et score_3. Pour chaque score_j, où $j = \{1, 2, 3\}$, la table contient deux informations :

- $\mathsf{rank}_j(lh, \sigma)$: le rang du compteur ordinal lh, de la trace σ en utilisant le score_j.
- $\mathsf{Verdict}_j$: le verdict de la localisation du loop-header en utilisant score_j, où le point noir « • » représente le succès de la localisation et le point blanc « – » représente son échec.

Les résultats de la Table 5.7 montrent que dans 57% des cas, l'utilisation de la métrique score_1 pour la localisation du loop-header, dans les traces Java, ne permet pas une bonne identification du bon compteur ordinal. Dans 33% des succès, le loop-header n'est pas classé en première position.

Les résultats de la Table 5.8 montrent que dans 46% des cas, l'utilisation de la métrique score_1 pour la localisation du loop-header, dans les traces des programmes embarqués, ne permet pas une bonne identification du bon comp-

P	score_1		score_2		score_3	
	$\mathsf{rank}_1(lh,\sigma)$	$\mathsf{Verdict}_1$	$\mathsf{rank}_2(lh,\sigma)$	$\mathsf{Verdict}_2$	$\mathsf{rank}_3(lh,\sigma)$	$\mathsf{Verdict}_3$
P_1	11	–	3	•	1	•
P_2	2	•	1	•	1	•
P_3	2	•	1	•	1	•
P_4	1	•	1	•	1	•
P_5	7	–	4	–	1	•
P_6	3	•	1	•	1	•
P_7	5	–	4	–	1	•
P_8	13	–	5	–	1	•
P_9	3	•	1	•	1	•
P_{10}	12	–	5	–	1	•
P_{11}	4	–	1	•	1	•
P_{12}	3	•	3	•	1	•
P_{13}	1	•	1	•	1	•
S	7		9		13	
E	6		4		0	

TABLE 5.8 – Résultats de l'évaluation en utilisant les programmes embarqués

teur ordinal. Dans 71% des succès, le loop-header n'est pas classé en première position.

D'après les résultats de l'évaluation et l'analyse du code source des programmes nous avons observé que l'utilisation de cette métrique peut fournir une bonne identification dans le cas où la boucle principale **while** ne contient pas d'autres itérations, sinon, il est fort probable d'avoir une mauvaise identification.

La seconde métrique utilisée pour la localisation du loop-header, notée score_2, permet une meilleure identification comparée à la première métrique score_1.

Pour les traces Java, elle permet l'identification du bon compteur ordinal dans 57% des cas, c'est-à-dire 14% mieux que score_1. Concernant le classement du loop-header, dans 12.5% des succès, le bon compteur ordinal est classé premier.

Pour les traces des programmes embarqué, cette seconde métrique permet l'identification du bon compteur ordinal dans 69% des cas. Concernant le classement du loop-header, dans 54% des succès, le bon compteur ordinal est classé premier.

L'utilisation de cette métrique fournit une bonne identification dans les cas ou le résultat de la multiplication $Na \times Da$ de la boucle **while** est le plus

grand.

Les résultats de la Table 5.7, ainsi que ceux de la Table 5.8, montrent que l'utilisation de la dernière métrique score_3, fournit dans 100% des cas une bonne identification du loop-header. Il est important de préciser qu'en utilisant cette métrique le loop-header est toujours classé en première position. L'utilisation de cette dernière métrique permet donc une bonne identification du loop-header dans les traces d'exécution de programmes cycliques.

5.2 Compression de la trace

Dans le chapitre 3, nous avons proposé une approche de compression de traces cycliques. Dans cette section nous cherchons à déterminer la pertinence de l'approche proposée par rapport à Sequitur. L'idée consiste donc à comparer les grammaires obtenues en appliquant Sequitur et Cyclitur sur les différentes traces d'exécution. Notre évaluation concerne deux types de traces : des traces de microcontrôleurs et des traces réseaux.

La première partie de cette section vise à expliquer les critères de comparaison sélectionnés. Dans ce qui suit, nous utilisons une séquence de symboles (trace) σ, et la grammaire de sortie (respectivement r-grammaire) générée avec Sequitur (respectivement Cyclitur), notée $G = \langle \Sigma, \Gamma, S, \Delta \rangle$.

5.2.1 Métriques

Dans notre évaluation expérimentale, deux métriques sont utilisées : la taille de la grammaire et le taux de compression. L'objectif de notre travail n'est pas de générer la meilleure compression possible, mais une compression concise et représentative pour l'analyse des traces d'exécution de microcontrôleurs. De plus, nous ne visons pas à définir un algorithme en se focalisant sur la vitesse de compression. Par conséquent, le temps de compression n'est pas pris en compte dans notre évaluation. Cependant, étant donné que notre approche est inspirée de Sequitur, nous avons observé que leurs temps de compressions sont très proches.

La **taille** de la grammaire G est la somme du nombre d'occurrences de symboles (terminaux et non-terminaux) dans son corps et le nombre de ses règles.

$$Size(G) = \sum_{(\beta,\sigma)\in\Delta} |\sigma| + 1$$

Le taux de compression, noté $Comp(G)$, est utilisé pour comparer les degrés de compression des grammaires générées par Sequitur et Cyclitur.

$$Comp(G) = \frac{Size(G)}{|\omega|}$$

5.2.2 Traces d'exécution de programmes sur microcontrôleurs

Les traces utilisées pour évaluer notre approche proviennent de cinq programmes embarqués sur microcontrôleurs, ces programmes sont fournis par nos partenaires industriels, STMicroelectronics et EASII-IC. Pour des raisons de confidentialité, les programmes ne sont pas décrits. Dans ce qui suit, nous noterons P_i le programme numéro i.

Comme pour les programmes embarqués utilisés dans l'évaluation de l'identification du loop-header (section 5.1), chaque programme de cette évaluation est chargé et exécuté sur une carte microcontrôleur STM32F107-EVAL-C. La trace d'exécution est récupérée à l'aide d'une sonde UlinkPro Keil [2], elle est enregistrée sous format CSV. Pour chaque programme, cinq traces, représentant des exécutions différentes, sont extraites. Pour chaque instruction, les informations suivantes sont enregistrées dans la trace d'exécution :

- son index : qui est un identifiant unique dans la trace,
- le moment de son exécution,
- l'instruction assembleur qui lui correspond,
- le compteur ordinal (en anglais program counter).

Pour notre approche de compression, nous nous intéressons aux compteurs ordinaux (PCs).

Résultats

La Table 5.9 contient les résultats de l'évaluation expérimentale. Chaque ligne du tableau représente une trace d'un programme. Les deux colonnes (*# Symbols*) et (*# Cycles*) représentent respectivement le nombre de symboles et le nombre de cycles identifiés dans une trace.

Pour chaque grammaire G générée par Sequitur, la Table 5.9 contient sa taille ($Size(G)$) et son taux de compression. Pour chaque grammaire G' générée par Cyclitur, la Table 5.9 contient sa taille ($Size(G')$), et son taux de compression (($Comp(G')$).

Dans la Figure 5.1, pour chaque programme, les moyennes des compressions sont calculées à l'aide des grammaires générées par Sequitur et les gram-

P.	T.			Sequitur		Cyclitur	
		# Symbols	# Cycles	$Size(G)$	$Comp(G)$	$Size(G')$	$Comp(G')$
P1	T 1	1048575	53821	2631	0.002509120	2044	0.001949312
	T 2	1048576	51574	1884	0.001796722	1455	0.001387596
	T 3	1048576	50482	1814	0.001729965	1798	0.001714706
	T 4	1048571	53819	1880	0.001792916	1510	0.001440055
	T 5	1048574	31562	1181	0.001126292	1139	0.001086237
P2	T 1	1048575	27542	22012	0.020992299	17945	0.017113702
	T 2	1048575	10621	20317	0.019375820	17728	0.016906754
	T 3	1048575	26043	19662	0.018751162	15190	0.014486327
	T 4	1048576	1038	20674	0.019716263	17002	0.016214371
	T 5	1048574	32515	20116	0.019184149	15888	0.015152006
P3	T 1	1048572	49208	1918	0.001829154	1331	0.001269345
	T 2	1048571	49207	1830	0.001745232	1397	0.001332289
	T 3	1048573	49205	1813	0.001729016	1463	0.001395230
	T 4	1048576	49207	1961	0.001870155	1409	0.001343727
	T 5	1048576	49209	1842	0.001756668	1741	0.001660347
P4	T 1	1048567	47029	1846	0.001760498	1498	0.001428616
	T 2	1048571	53854	2250	0.002145777	1630	0.001554497
	T 3	1048574	47031	2032	0.001937870	1458	0.001390460
	T 4	1048575	53860	1808	0.001724245	1494	0.001424791
	T 5	1048575	53866	1947	0.001856806	1716	0.001636507
P5	T 1	1048573	64527	309	0.000294686	139	0.000132561
	T 2	1048576	64527	304	0.000289917	142	0.000135422
	T 3	1048570	52782	1641	0.001564989	1334	0.001272209
	T 4	1048571	64526	317	0.000302316	144	0.000137330
	T 5	1048576	64528	298	0.000284195	146	0.000139236

TABLE 5.9 – Résultats de l'évaluation de la compression de traces

maires générées par Cyclitur. Les valeurs des taux de compression varient entre 0 et 1. La valeur 0 représente la meilleure compression et la valeur 1, la pire.

Nous observons d'après les résultats de l'évaluation, illustrés dans la Figure 5.1, que pour les cinq programmes, notre approche de compression basée sur l'identification des cycles permet d'obtenir une meilleure compression que celle générée par Sequitur. De plus, Cyclitur génère une grammaire qui contient plus de règles que la grammaire générée par Sequitur, cependant la compression fournit par Cyclitur est plus facile à comprendre et à analyser, car elle facilite la détection de cycles.

Par exemple, pour le programme $P1$, les tailles des grammaires générées par Sequitur pour les traces d'exécution varient de 1,181 à 2,631 symboles. Elles contiennent entre 214 et 387 terminaux, 728-1,689 non-terminaux et 239-1,689 règles. Les tailles des traces d'origine varient entre 1,048,571 et 1,048,576, avec 31,562-53,821 cycles. Par conséquent, les ratios de compression varient entre 0.0011 et 0.0025. Alors que Cyclitur génère des grammaires dont les tailles varient de 1,139 à 2,044 symboles avec 213-385 terminaux, 687-1,237 non-terminaux et 239-423 règles. Les taux de compression varient entre 0.0010 et 0.0019.

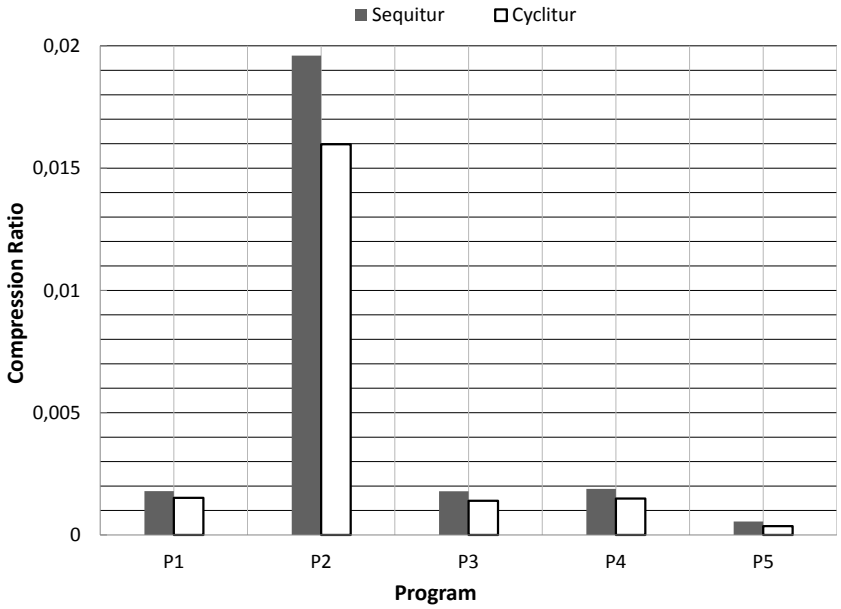

FIGURE 5.1 – Comparaison des moyennes de taux de compression pour chaque programme

D'après la Figure 5.1, nous notons que, pour tous les programmes utilisés dans l'évaluation expérimentale, l'utilisation de Cyclitur génère une meilleure compression que Sequitur. Les ratios de compression sont meilleurs de 15% à 30%.

5.2.3 Compression des traces réseaux

Nous avons évalué notre approche de compression sur quatre traces supplémentaires obtenues à partir de simulations de réseaux. Ces traces sont cycliques. Elles nous ont été fournies par une doctorante qui travaillait sur le réseau considéré « Multi-Channel Multi-Interface Wireless Mesh Network (WMN) » avec des routeurs basés sur la norme IEEE technologie 802.11 [31].

La tête de la boucle *lh* utilisée pour détecter les cycles et diviser les traces générées en blocs est un événement spécifique qui se réfère à l'émission d'une demande du client au serveur.

La première trace se compose de 6,011,850 événements répartis sur 9,574 cycles. Le taux de compression en utilisant Sequitur est de 0.0027% pour une grammaire générée de taille égale à 16,531, elle contient 744 terminaux, 12,375 non-terminaux et 3,412 règles. La taille de la grammaire générée par Cyclitur est égale à 16,057, elle contient 373 terminaux, 13,884 non-terminaux et 4182 règles. Le taux de compression pour la grammaire générée par Cyclitur est de

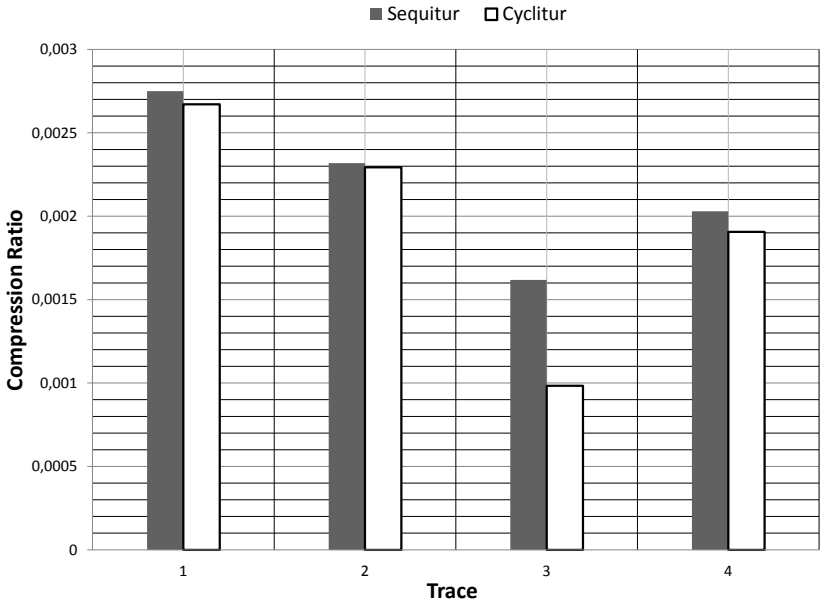

FIGURE 5.2 – Comparaison des ratios de compression pour chaque trace WMN

0.0026%.

La seconde trace se compose de 8,040,942 événements répartis sur 9.574 cycles. Sequitur génère une grammaire dont la taille est égale à 18,639, elle contient 716 terminaux, 14,042 non-terminaux et 3,881 règles. La taille de la grammaire générée à l'aide de Cyclitur est égale à 18,439, elle contient 373 terminaux, 13,884 non-terminaux et 4182 règles. L'utilisation de Sequitur et Cyclitur sur la seconde trace donne comme taux de compression, respectivement, 0.00023% et 0.0022%.

La troisième trace WMN contient 10,312,955 événements répartis sur 33,834 cycles. Sequitur génère une grammaire qui contient 605 terminaux, 12,584 non-terminaux et 3501 règles. Le taux de compression pour la grammaire générée par Sequitur est de 0.0016%. Cyclitur génère une grammaire qui contient 272 terminaux, 7783 non-terminaux et 2090 règles. Le taux de compression pour la grammaire générée par Cyclitur est de 0.0009%.

La dernière trace WMN contient 13,883,977 événements répartis sur 2,797 cycles. Le taux de compression en utilisant Sequitur est de 0.0020% pour une grammaire générée de taille égale à 28,181, elle contient 661 terminaux, 21,575 non-terminaux et 5945 règles. La taille de la grammaire générée par Cyclitur est égale à 26,468, elle contient 349 terminaux, 20,108 non-terminaux et 6011 règles. Le taux de compression pour la grammaire générée par Cyclitur est de 0.0019%.

Les résultats de la Figure 5.2 montrent que Sequitur et Cyclitur peuvent

être aussi utilisés pour la compression des traces réseaux. Pour toutes les traces réseaux utilisées dans l'évaluation expérimentale, la Figure 5.2 montre que l'utilisation de Cyclitur génère dans ce contexte-là aussi une meilleure compression que Sequitur.

Chapitre 6
Conclusion

La plupart des programmes embarqués sur microcontrôleurs sont des programmes *cycliques*. Par conséquent, leurs traces d'exécution contiennent souvent des séquences qui se répètent à des fréquences plus ou moins importantes.

En considérant une trace d'exécution comme une séquence de symboles, nous visons à aider à son analyse et compréhension. Cela en utilisant la *compression basée sur la génération d'une grammaire*. La compression générée par notre approche est donc une description synthétique et représentative de la trace.

L'algorithme de compression de données *Sequitur* (chapitre 2) possède plusieurs qualités pour être un prétendant pour ce type de détection de séquences répétitives. Dans notre contexte de travail, les *cycles* sont plus importants que les autres répétitions dans la trace et nécessitent un traitement spécifique. Étant donné que *Sequitur* ne différencie pas les cycles des autres répétition, souvent, dans la grammaire qu'il génère, les cycles sont entremêlés, ce qui rend la détection d'un cycle en particulier coûteuse en temps.

Pour cette raison, nous avons proposé une extension de Sequitur, appelée *Cyclitur*. Notre algorithme permet, grâce à la détection de cycles, d'avoir une compression plus représentative de la trace. Cette contribution nous a valu la publication d'un article [15] dans *IESS 2013*[1].

Notre approche se divise donc en deux étapes. La première étape est la *détection de cycles*, qui repose sur la localisation du loop-header dans la trace. Souvent, due aux optimisations apportées par la phase de compilation, l'identification manuelle du loop-header devient difficile. Par conséquent, en se basant sur le modèle des programmes cycliques, nous avons défini trois métriques pour identifier automatiquement le loop-header (chapitre 4).

La seconde étape consiste en l'application d'une version améliorée de Sequitur, appelée *R-Sequitur*, sur la trace découpée en cycles. *R-Sequitur* est appliqué en premier lieu sur chaque cycle afin de détecter les répétitions à l'intérieur des cycles. Puis l'application de R-Sequitur sur la compression générée par l'étape précédente permet la détection des séquences de cycles similaires dans la trace.

1. http ://www.iess.org/

Afin d'évaluer la pertinence des stratégies proposées pour identifier le loop-header, une évaluation expérimentale a été menée, section 5.1. Les résultats de cette évaluation montrent qu'en utilisant des traces cycliques, dans la plupart des cas, notre approche permet une bonne identification du loop-header et ainsi une bonne détection des cycles dans la trace. Nous pensons que notre approche peut échouer dans l'identification du loop-header dans le cas où l'exécution s'arrête avant d'itérer sur la boucle principale, par exemple lors de la phase d'initialisation de variables.

Une évaluation de notre approche de compression, présentée dans la section 5.2, a été menée. L'objectif de cette évaluation était de mesurer la capacité de notre approche à offrir une compression concise et représentative de la trace tout en conservant son aspect cyclique. Cela en comparant les grammaires obtenues en appliquant Sequitur et Cyclitur sur les différentes traces d'exécution. Cette évaluation concernait des traces d'exécution de microcontrôleur, mais également des traces d'exécution de réseaux. D'après les résultats de cette expérimentation, notre approche, tout en permettant une bonne détection de cycles, offre dans tous les cas une meilleure compression que Sequitur.

Cependant, la grammaire générée par notre approche possède deux limites. Ces limites sont liées principalement à la compression basée sur la génération d'une grammaire et sont présentées en détail ci-après. La première limite concerne la génération d'une grammaire irréductible, mais pas unique. La conséquence de cette limite est la possibilité d'avoir plusieurs représentations pour une même répétition. La seconde limite concerne la représentation des répétitions, où deux répétitions qui ne diffèrent que par un symbole, se verront attribuer deux représentations différentes. Cela a pour conséquence la génération d'un grand nombre de règles. Par conséquent, l'effort d'analyse de la compression grandit.

Grammaire irréductible mais pas unique. Pour une trace d'exécution donnée, l'utilisation de la compression sur cette trace ne garantit pas la génération d'une grammaire unique. Ainsi, pour un même cycle, il est possible d'avoir deux représentations différentes, comme le montre la trace et sa compression dans l'exemple suivant : trace = [1 2 3 4 5 6][1 5 6][1 2 3 4 5][1 2 3 4 5 6].

$$S \rightarrow C1\ C2\ C3\ C4$$
$$R1 \rightarrow 1\ 2\ 3\ 4$$
$$R2 \rightarrow 5\ 6$$
$$C2 \rightarrow 1\ R2$$
$$C3 \rightarrow R1\ 5$$
$$C1 \rightarrow R1\ R2$$
$$C4 \rightarrow C3\ 6$$

Il est important de remarquer que dans la compression générée, le même cycle [1 2 3 4 5 6] a été représenté de deux façons différentes, $C1$ et $C4$. Cette situation pourrait induire l'ingénieur en erreur pendant l'analyse du comportement en pensant que les cycles $C1$ et $C4$ sont deux cycles différents.

Une solution possible pour cette limite serait d'analyser la grammaire une fois la compression terminée, cependant cela augmenterait la complexité du processus de compression.

Cette limite ne représente pas un handicap pour notre travail, car notre objectif n'est pas de générer la plus petite grammaire possible, mais d'offrir à l'ingénieur une description synthétique et représentative afin d'aider à l'analyse de la trace.

Représentation de répétitions. Un autre inconvénient de la compression basée sur la génération d'une grammaire, est que deux sous-séquences ω_1 et ω_2, qui ne diffèrent que par un symbole, se verront attribuer deux représentations différentes, comme illustré dans l'exemple de la trace suivante : [1 2 3 4 5][1 2 3][1 2 4 5][1 2 4], où $\omega_1 = C1 = [1\ 2\ 3\ 4\ 5]$ et $\omega_2 = C3 = [1\ 2\ 4\ 5]$.

$$S \rightarrow C1\ C2\ C3\ C4$$
$$C2 \rightarrow R1\ 3$$
$$C4 \rightarrow R1\ 4$$
$$R1 \rightarrow 1\ 2$$
$$C1 \rightarrow C2\ 4\ 5$$
$$C3 \rightarrow C4\ 5$$

Une solution possible pour cette limite serait de représenter les répétitions qui se ressemblent par une même lettre alphabétique et ne changer que l'index numérique. Ainsi, dans notre exemple, nous aurons la compression suivante $S \rightarrow A1\ C1\ A2\ C2$. L'inconvénient de cette solution est une augmentation de la complexité liée à la comparaison des cycles entre eux.

Ainsi, l'étude de cette limite pour proposer une meilleure représentation des répétitions constitue une perspective de notre travail.

-Partie 2-

Localisation de faute(s) dans les programmes

Résumé

Une étape importante du processus de débogage consiste à localiser des fautes. Le repérage précis de fautes étant très difficile, on essaye plutôt d'identifier les instructions du code source susceptibles de contenir des fautes.

Dans cette seconde partie de la thèse nous proposons de faire de la *localisation de faute* en utilisant une seule trace d'exécution [16]. Pour cela notre approche exploite les cycles en les considérant comme des *spectres d'exécution*. Par conséquent, notre approche fait de la *localisation de faute(s) basée sur les spectres (SBFL)*, qui est discutée dans le chapitre 8. Plusieurs techniques de localisation de faute(s) sont présentées dans ce chapitre, comme par exemple : la méthode du *plus proche voisin* [65], et *Tarantula* [42].

Le chapitre 9 présente notre contribution concernant l'adaptation des méthodes présentées dans le chapitre 8, cela en utilisant une seule trace d'exécution et en considérant ses cycles comme des *spectres d'exécution*.

L'évaluation expérimentale de cette contribution est présentée dans le chapitre 10. Cette évaluation a permis également de mettre en avant l'efficacité de notre *fitrage*, qui a pour objectif de réduire l'effort d'analyse. Ce processus, décrit dans le chapitre 10, permet de réduire le nombre de cycles à prendre en considération pour la localisation de faute.

Chapitre 7
Introduction

Au cours du développement d'une application, des défaillances peuvent apparaître. Elles sont la conséquence de fautes introduites dans le code. L'exécution des instructions fautives peut entrainer des erreurs. Des défaillances pourront (ou non) être observées par les développeurs, le client ou les utilisateurs pendant l'exploitation.

Pour mettre en évidence des défaillances, des méthodes de validation et de vérification peuvent être utilisées. La méthodes la plus classiquement utilisée dans l'industrie étant le test [35]. Une fois une défaillance identifiée, il est nécessaire de la corriger : c'est l'étape de débogage. Cela consiste en premier lieu à rechercher la cause de la défaillance, c'est-à-dire l'instruction erronée. On parle alors de localisation de faute.

La localisation de faute(s) peut être définie comme étant le processus de repérage de faute(s) conduisant à des défaillances. C'est un processus qui peut être long et fastidieux. C'est pourquoi un enjeu consiste à automatiser la localisation de faute(s). Le repérage précis de fautes étant très difficile, la plupart des méthodes de localisation de faute(s) essayent plutôt d'identifier les partitions du code source susceptibles de contenir la faute. Ainsi, basées sur un ensemble d'observations (exécutions), les approches automatiques de localisation de faute(s) fournissent au développeur chargé du débogage une liste d'emplacements susceptibles de contenir la faute.

Dans notre contexte de travail, il est devenu possible d'enregistré l'exécution du programme embarqué sur le microcontrôleur. Par conséquent, pour la localisation de faute, nous nous intéressons au suivi et au résultat de l'exécution du programme embarqué en manipulant les informations contenues dans la trace d'exécution.

Dans ce qui suit, nous présentons quelques approches permettant l'automatisation de la localisation de faute(s). Ces approches de localisation de faute(s) sont classées dans la famille des approches dynamiques.

Il existe des « approches statiques » qui visent à vérifier les programmes en utilisant des éléments statiques, comme la syntaxe. Par conséquent, pour cette famille de méthodes, l'information calculée est vraie pour l'ensemble des exécutions du programme.

A l'inverse des approches statiques, les méthodes dynamiques utilisent des données issues de l'exécution du programme (comme les instructions exécu-

tées). Ainsi, l'information calculée n'est vraie que dans le contexte d'un sous-ensemble d'exécutions. Ce sous-ensemble ne contient que les exécutions qui ont généré les données à partir desquelles les calculs ont été faits. L'analyse de traces d'exécution étant une approche dynamique de localisation de faute(s), dans cette partie nous nous intéressons aux approches de cette famille.

Approches dynamiques : Les approches de cette famille d'analyse s'intéressent au suivi et au résultat de l'exécution du programme. Elles manipulent l'information issue de l'exécution des programmes, comme les instructions exécutées, les valeurs des variables ou encore le temps d'exécution. Afin de permettre la manipulation des informations issues de l'exécution d'un programme, ces informations sont stockées dans une *trace d'exécution*. Il est important de noter que selon l'approche utilisée, certaines informations issues de l'exécution sont plus pertinentes que d'autres. Les approches de cette famille utilisent donc une abstraction de la trace d'exécution, appelée *spectre d'exécution*.

Le plus souvent, un spectre contient le verdict de l'exécution : succès (pass) ou échec (fail). De nombreuses formes de spectres d'exécution existent [37], comme par exemple, le spectre *component-hit* qui indique si un composant a été impliqué dans l'exécution du programme ou non.

Les méthodes dynamiques qui utilisent les spectres d'exécution pour la localisation de faute(s) sont appelées « méthodes par différence de traces (spectres) ». L'idée principale de ces méthodes est de comparer les comportements des exécutions en échec et les comportements des exécutions en succès, afin de renvoyer les différences entre ces comportements comme information pour le débogage. Dans le but de mieux cibler la faute, des méthodes statistiques sont utilisées pour calculer un ordre total sur des prédicats, ou sur les lignes d'un programme à inspecter pour trouver la faute.

À cause de sa simplicité et de son efficacité, la localisation de faute(s) basée sur les spectres, notée SBFL, a reçu beaucoup d'attention dans le domaine de la recherche. Ainsi, afin d'améliorer la précision du diagnostic, certaines études récentes portées sur la SBFL proposent les approches, comme Tarantula [41] et Ochiai [17]. Ces approches se distinguent principalement par la sélection du spectre, mais aussi par le choix de la formule utilisée pour l'évaluation du risque pour chaque entité du programme.

Des exemples bien connus de techniques de localisation de faute(s) basées sur les spectres, sont l'outil Tarantula proposé par Jones, Harrold et Stasko [42], la technique du plus proche voisin proposée par Renieris et Reiss [65], l'outil Sober proposé par Lui, Yan , Fei, Han, et Midkiff [55], et l'outil Ample proposé dans [28] par Dallmeier, Lindig, et Zeller.

Bien que ces approches soient différentes dans la façon dont elles calculent

l'ordre total des prédicats/lignes pouvant contenir la faute, elles sont toutes basées sur l'utilisation des spectres de programme. Ces approches peuvent être interprétées comme étant des procédures de fouille de données, comme il est, par exemple, expliqué par Denmat et al. [32] concernant la méthode Tarantula[42].

Dans notre contexte de travail, nous disposons d'une trace d'exécution fautive. Ainsi, il serait intéressant d'explorer ces approches pour la localisation de faute(s) dans le contexte des programmes pour microcontrôleurs. Les approches étudiées sont analysées dans le chapitre suivant.

Il est important de préciser que dans cette thèse nous nous intéressons à la localisation d'une seule faute à la fois. La localisation simultanée de plusieurs fautes représente une perspective de notre travail. Ainsi, nous parlons bien de localisation de *faute* et non de localisation de *fautes*. Le travail présenté dans cette partie nous a valu la publication de l'article [16] dans *ISSRE 2013*[1].

1. http ://2013.issre.net/

Chapitre 8

Localisation de faute basée sur les spectres

Sommaire

Comme nous l'avons dit plus tôt, une partie importante du diagnostic consiste à localiser les fautes. Plusieurs outils de débogage automatisé et de diagnostic de systèmes utilisent la *localisation de faute basée sur les spectres (SBFL)*. La SBFL est une approche de diagnostic basée sur l'analyse des différences dans les spectres d'un programme [37, 66]. Cette analyse de différence concerne des exécutions qui sont des succès et des exécutions qui sont des échecs. Si le résultat de l'exécution d'un programme est celui attendu alors l'exécution est un succès, sinon on dit que cette exécution est un échec. Une exécution en échec est une exécution dans laquelle une défaillance a été détectée.

Un spectre d'exécution est une abstraction de la trace d'exécution, il indique quelles parties du programme sont actives, pendant une exécution spécifique. Ainsi, la localisation de faute basée sur les spectres identifie la partie du programme dont l'activité est la plus corrélée avec la détection d'erreurs.

Des exemples d'outils qui mettent en œuvre cette approche sont Tarantula [42], pour l'analyse des programmes en C, ou Ample [28], qui met l'accent sur les programmes orientés objet.

Dans le but d'améliorer le diagnostic en utilisant la SBFL, la plupart des techniques utilisent *un cœfficient de similarité* pour classer les instructions pouvant contenir la faute. Par conséquent, la précision du diagnostic est influencée par le cœfficient de similarité utilisé, mais également par la qualité et la quantité des observations utilisées dans l'analyse. Ces observations sont généralement résumées dans les spectres d'exécution, qui sont discutés dans la section suivante.

8.1 Spectre d'exécution

Un spectre d'exécution est un ensemble de données qui fournit une vue spécifique sur le comportement dynamique du programme [66]. La vue est dite spécifique parce qu'elle dépend du type de traitement qui sera effectué sur le spectre d'exécution.

Par exemple l'ensemble de données fourni par un spectre d'exécution pourrait être des variables et leurs valeurs, ou bien l'ensemble de lignes exécutées. Par conséquent, différentes formes de spectres d'exécutions existent [37].

Le tableau 8.1 énumère quelques exemples de spectre et les types de données auxquels ils s'intéressent.

Comme pour une trace d'exécution, les données fournies par un spectre d'exécution sont collectées pendant l'exécution du programme. Ils sont principalement utilisés pour localiser les fautes, comme expliqué dans les sections suivantes.

8.2 Méthodes par différence de spectres

Souvent, la localisation de faute utilisant des méthodes par différence de spectres met en œuvre des spectres de type « statement-hit ». Ce type de spectre d'exécution s'intéresse aux instructions parcourues pendant l'exécution du programme (Figure 8.1). La principale idée utilisée par les méthodes par différence de spectres est la comparaison de spectres lors d'exécutions en échec et en succès. Ces méthodes sont basées sur deux intuitions. La première suppose que les instructions exécutées *uniquement* pendant les exécutions en succès ne sont pas liées aux défaillances observées, alors que les lignes qui sont exécutées pendant les exécutions en échec peuvent être à l'origine des défaillances. La deuxième intuition suppose que l'absence des instructions exécutées pendant les exécutions en succès, dans les exécutions en échecs, est liée

Forme du spectre	Description de l'ensemble de données produit
Statement-hit	instructions exécutées
Statement-count	instructions exécutées et nombre de fois qu'une instruction a été exécutée
Block-hit	branchements conditionnels exécutés
Block-count	branchements conditionnels exécutés et nombre de fois qu'un branchement conditionnel a été exécuté
Path-hit	chemins (complets ou incomplets) exécutés
Path-count	chemins (complets ou incomplets) exécutés et nombre de fois qu'un chemin a été exécuté
Complete-path	chemins complets exécuté
Data-dependence-hit	paires définition-utilisation (def-use) exécutées
Data-dependence-count	paires définition-utilisation exécutées et nombre de fois qu'une paire définition-utilisation a été exécuté
Output	sorties (résultats) produites
Execution trace	trace d'exécution produite
Time Spectra	temps d'exécution, i.e. temps d'exécution de fonctions

TABLE 8.1 – Énumération de spectres d'exécution et leurs ensembles de données

aux défaillances observées.

Dans cette section, trois méthodes proposées par Renieris et Reiss sont présentées [65] : *union model*, *intersection model*, et *nearest neighbor*. Ces trois méthodes exposent les différences entre deux ensembles de spectres d'exécutions. Le premier ensemble contient un ou plusieurs spectres des exécutions en succès, alors que le second ensemble ne contient que le spectre de l'exécution en échec. Le résultat produit par chacune de ces trois méthodes est un ensemble d'instructions suspectes, qu'il faut inspecter afin de localiser la faute.

8.2.1 Union Model

Cette méthode vise à repérer les instructions qui apparaissent uniquement dans le spectre de l'exécution en échec mais dans aucun spectre des exécutions en succès. Intuitivement, cela correspond à l'idée qu'une défaillance apparaît (systématiquement) lorsqu'une instruction fautive est exécutée. Si elle n'est pas exécutée, alors aucune défaillance n'est observée. Cela se traduit mathématiquement par : $spectre_{échec} - (\bigcup spectre_{succès})$.

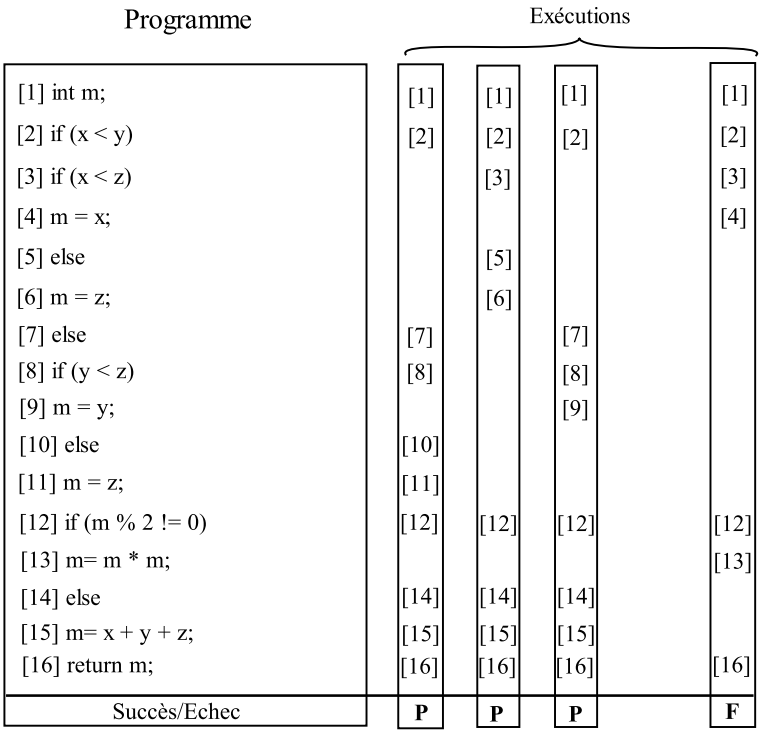

FIGURE 8.1 – Exemple de spectre *statement-hit*

La Figure 8.1 illustre un programme et quatre spectres de son exécution : trois spectres d'exécutions en succès (verdict = **P**) et un spectre d'une exécution en échec (verdict = **F**). Chaque spectre contient la liste des lignes (instructions) parcourues pendant l'exécution du programme, par exemple le spectre de l'exécution en échec contient les lignes : 1, 2, 3, 4, 12, 13, 16.

Sur l'exemple de la Figure 8.1, la méthode fournit un ensemble composé de deux suspects : la ligne 4 et la ligne 13. Ces deux lignes n'apparaissent que dans le spectre de l'exécution en échec, c'est-à-dire $\{4,\ 13\} = spectre_{échec} - (\bigcup spectre_{succès})$ où $spectre_{échec} = \{1, 2, 3, 4, 12, 13, 16\}$ et $\bigcup spectre_{succès} = \{1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16\}$. Le résultat de l'application de la méthode *union model* est illustré dans la Figure 8.2.

Ainsi, la méthode *union model* est basée sur le calcul de l'ensemble de lignes caractérisant une exécution en échec. Cependant, l'exécution d'une instruction fautive n'entraîne pas toujours un échec. Par conséquent, une exécution en échec et une exécution en succès peuvent fournir des spectres identiques, dans ce cas, cette méthode perd en efficacité et son application renvoie un ensemble vide.

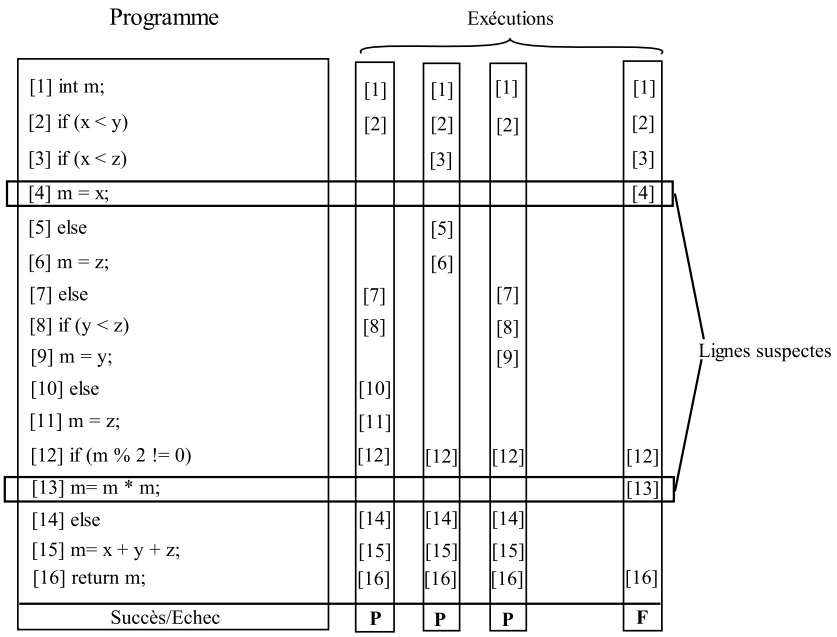

FIGURE 8.2 – Résultat de la méthode *union model*

8.2.2 Intersection Model

À l'inverse de la méthode *union model*, l'ensemble produit par la méthode *intersection model* ne contient pas les instructions (lignes) qui ne sont présentent que dans l'exécution en échec. L'ensemble généré contient les instructions (lignes) qui sont présentes dans tous les spectres des exécutions en succès, mais qui sont absentes dans le spectre de l'exécution en échec. Intuitivement, cela correspond au fait que l'absence d'une instruction peut conduire à une défaillance. Cela se traduit mathématiquement par : $(\bigcap spectre_{succès}) - spectre_{échec}$.

L'application de l'*intersection model* sur l'exemple de la Figure 8.1 fournit un ensemble composé de deux suspects : la ligne 14 et la ligne 15. Ces deux lignes apparaissent dans tous les spectres des exécutions en succès mais n'apparaissent pas dans le spectre de l'exécution en échec, c'est-à-dire $\{14,\ 15\} = (\bigcap spectre_{succès}) - spectre_{échec}$, où $\bigcap spectre_{succès} = \{1, 2, 12, 14, 15, 16\}$ et $spectre_{échec} = \{1, 2, 3, 4, 12, 13, 16\}$. Le résultat de l'application de la méthode *intersection model* est illustrée dans la Figure 8.3.

L'exécution d'une instruction fautive n'entraîne pas toujours un échec. Par conséquent, cette méthode possède la même faiblesse que le méthode *union model*, c'est-à-dire qu'une exécution en échec et une exécution en succès peuvent fournir des spectres identiques, ainsi l'application de la méthode *intersection model* dans ce cas ne renvoie aucun résultat.

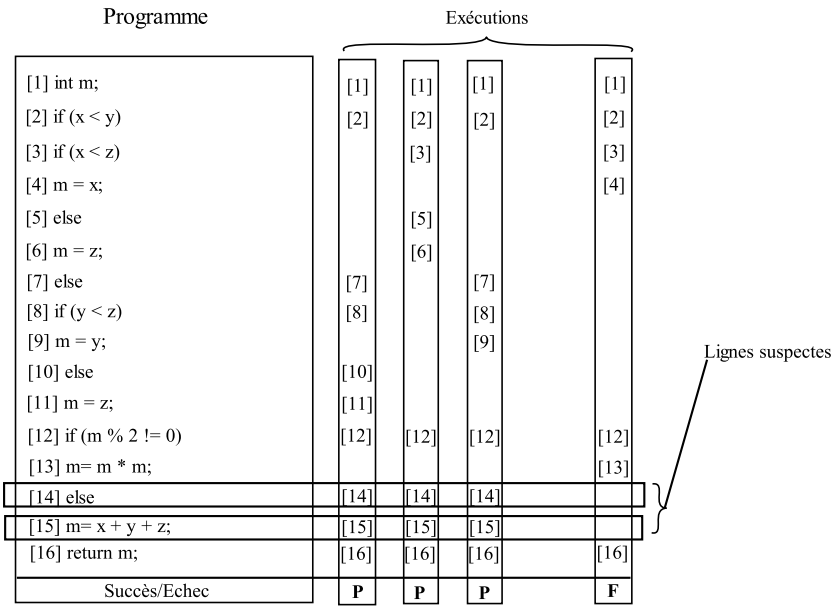

FIGURE 8.3 – Résultat de la méthode *intersection model*

Il est important de noter que les deux méthodes *union model* et *intersection model* sont complémentaires. La méthode *union model* cherche à localiser la faute en identifiant ce qui apparait en plus dans une exécution qui échoue par rapport à des exécutions en succès, alors que la méthode *intersection model* cherche à localiser la faute en identifiant ce qui manque à l'exécution en échec pour être en succès. Ainsi, un raisonnement est né à partir du fait que ces deux méthodes soient complémentaires et a permis de définir une nouvelle méthode appelée *voisin le plus proche*.

8.2.3 Le voisin le plus proche

Le voisin le plus proche (en anglais *Nearest neighbor*) est une méthode proposée également par Renieris et Reiss [65].

Cette méthode met en évidence un spectre d'exécution particulier, appelé « plus proche voisin ». Ce « plus proche voisin » est le spectre d'exécution en succès qui partage le plus d'instructions avec le spectre de l'exécution en échec. Pour trouver le « plus proche voisin », la méthode *Nearest neighbor* calcule une distance entre le spectre de l'exécution en échec et chaque spectre d'exécution en succès.

Ensuite, la méthode cherche à identifier les différences entre le spectre de l'exécution en échec et son « plus proche voisin ». Cela se traduit mathématiquement par : $spectre_{échec} - spectre_{ppv}$.

L'application de cette méthode sur l'exemple de la Figure 8.1 est illus-

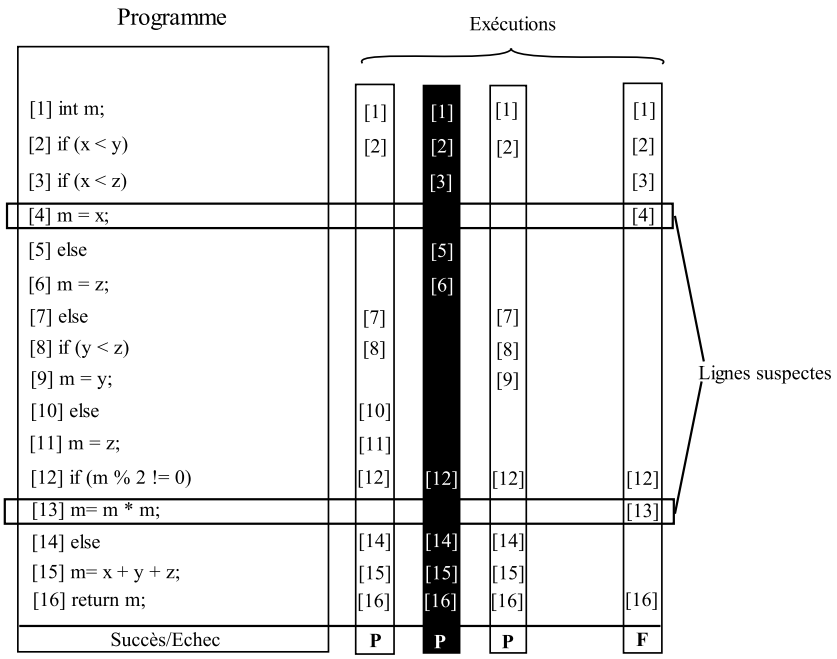

FIGURE 8.4 – Résultat de la méthode *nearest neighbor*

trée dans la Figure 8.4. La méthode fournit un ensemble composé de deux suspects : la ligne 4 et la ligne 13. Ces deux lignes apparaissent dans le spectre de l'exécution en échec, mais n'apparaissent pas dans le spectre d'exécution en succès sélectionné comme étant le « plus proche voisin », c'est-à-dire $\{4,\ 13\} = spectre_{échec} - spectre_{ppv}$ où $spectre_{échec} = \{1, 2, 3, 4, 12, 13, 16\}$ et $spectre_{ppv} = \{1, 2, 3, 5, 6, 12, 14, 15, 16\}$.

Comme pour les deux méthodes précédentes, *union model* et *intersection model*, la méthode du *plus proche voisin* ne donne pas de résultat dans le cas où une exécution en échec et une exécution en succès fournissent des spectres identiques.

8.3 Méthodes utilisant les scores de suspicion

Jones, Harold et Stasko ont proposé un autre type de localisation de faute basée sur l'utilisation de spectres [42]. Ce type de localisation de faute vise à identifier les éléments (lignes) du spectre, dont la présence ou l'absence est *fortement corrélée* avec l'échec de l'exécution. Ces éléments (lignes) sont considérés comme étant probablement à l'origine de la défaillance constatée. Ainsi, pour avoir cet ensemble d'élément suspects, pour chaque élément (ligne) du spectre d'exécution, un *score de suspicion* (appelé aussi *degré de similarité*) est calculé. Le score de suspicion est défini comme étant une mesure de cor-

rélation entre l'exécution de l'élément (ligne) et l'échec de l'exécution. Ainsi, l'élément ayant le degré de suspicion le plus élevé est interprété comme étant probablement l'élément défectueux, et doit être examiné avec une priorité plus élevée.

Pour calculer les degrés de similarité et identifier les éléments suspects, ce type de localisation de faute utilise un ensemble de spectres de type *statement-hit*, mais également les verdicts des exécutions (succès/échec). Ce type de localisation nécessite la présence d'au moins un spectre d'exécution en échec. Les calculs nécessaires à l'identification des éléments suspects et cinq méthodes utilisant les scores de suspicion sont présentés dans les sections suivantes.

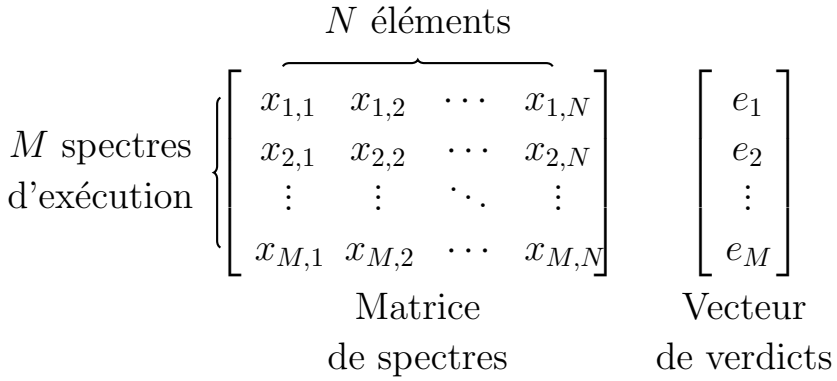

FIGURE 8.5 – La matrice de spectres et le vecteur de verdicts

8.3.1 Préliminaires

L'identification des éléments suspects se déroule en deux étapes. La première étape consiste à construire la matrice de spectres et le vecteur de verdicts correspondants aux exécutions. La seconde étape est le calcul des scores de suspicion.

Matrice de spectres et vecteur de verdicts : Afin de calculer le score de suspicion pour chaque instruction du programme, une matrice particulière, appelée « matrice de spectres », est construite. Comme illustré dans la Figure 8.5, pour M spectres d'exécutions et N éléments (lignes) du programme, une matrice de spectres de $M \times N$ éléments est construite, telle que :

- Chaque ligne i dans la matrice correspond à un spectre d'une exécution particulière ;
- Chaque colonne j correspond à un élément particulier du programme exécuté (i.e. instruction (ligne), bloc d'instructions) ;

– Chaque valeur $x_{i,j}$ de la matrice est une valeur booléenne indiquant si durant une exécution i l'élément j a été exécuté ($x_{i,j} = 1$) ou non ($x_{i,j} = 0$).

Le vecteur de verdicts, appelé aussi vecteur d'erreur, représente les verdicts des exécutions, tel que :

– Chaque valeur e_i du vecteur correspond au verdict de l'exécution associée au $i$ème spectre de la matrice de spectres ;
– Chaque valeur e_i du vecteur est une valeur booléenne indiquant si l'exécution i était un succès ($e_i = 0$) ou un échec ($e_i = 1$).

La matrice de spectres et le vecteur de verdicts de l'exemple de la Figure 8.1 sont illustrés dans la Figure 8.6.

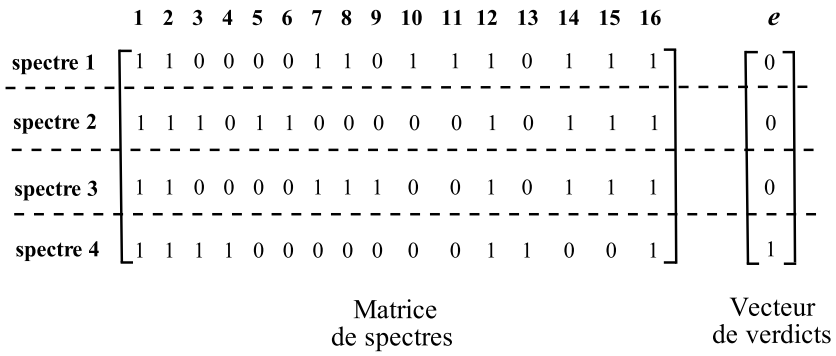

FIGURE 8.6 – Matrice de spectres et vecteur de verdicts de l'exemple de la Figure 8.1

Degré de suspicion : Après la construction de la matrice de spectres et du vecteur de verdicts, chaque élément du programme se voit attribuer un degré de suspicion. Les degrés de suspicion sont calculés en utilisant des cœfficients de similarité entre chaque colonne de la matrice et le vecteur de verdicts. L'ordre sur les degrés de suspicion est très important, car l'élément dont la colonne a le score de suspicion le plus élevé est considéré comme le plus probable à contenir la faute.

Un degré de suspicion est une fonction utilisant quatre compteurs : $a_{te}(j)$, où t vaut 1 si l'élément j est présent dans le spectre d'une exécution, sinon il vaut 0. La valeur de e vaut 1 si l'exécution est en échec, sinon si l'exécution est en succès, e vaut 0. Le calcul des quatre compteurs est illustré dans l'Algorithme 5 et ils sont :

- $a_{11}(j)$: compte le nombre de fois où l'élément j a été exécuté pendant une exécution en échec.
- $a_{10}(j)$: compte le nombre de fois où l'élément j a été exécuté pendant une exécution en succès.
- $a_{01}(j)$: compte le nombre de fois où l'élément j n'a pas été exécuté pendant une exécution en échec.
- $a_{00}(j)$: compte le nombre de fois où l'élément j n'a pas été exécuté pendant une exécution en succès.

Il existe de nombreux cœfficients de similarité. Connus pour être parmi les meilleurs pour la SBFL [10, 9], nous considérons les cinq cœfficients de similarité : Tarantula [42], Jaccard [24], Ample [28], Ochiai [9] et le cœfficients O^p [58]. Nous les présentons dans ce qui suit.

8.3.2 Tarantula

Tarantula [42, 41] est un outil de localisation de faute, développé pour les programmes écrits en langage C. Cet outil utilise la SBFL en se basant sur plusieurs exécutions en échecs et en succès, où pour chaque exécution, le spectre *statement-hit*, contenant les lignes exécutées, ainsi que le verdict de l'exécution sont générés.

À la différence de l'approche *union model*, proposée par Renieris et Reiss, qui calcule les instructions qui apparaissent *toujours* dans les spectres des exécutions en échec, mais n'apparaissent *jamais* dans les spectres des exécutions en succès, Tarantula calcule les instructions qui apparaissent *très souvent* dans les spectres des exécutions en échec et *rarement* dans les spectres d'exécutions en succès.

Le calcul du degré de suspicion pour une ligne j avec Tarantula, en utilisant les quatre compteurs $(a_{11}, a_{10}, a_{01}, a_{00})$ se définit comme suit :

$$Tarantula(j) = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}}$$

Le résultat de la localisation de faute en utilisant le cœfficient Tarantula sur l'exemple de la Figure 8.1 est illustré dans la Figure 8.7.

Afin de faciliter le diagnostic, Tarantula implémente une interface graphique grâce à laquelle les instructions sont colorées en fonction de leurs degrés de suspicion. Pour cela, l'outil associe à chaque ligne du code source deux indices : un indice de « *couleur* » et un indice de « *brillance* », décrient ci-dessous :

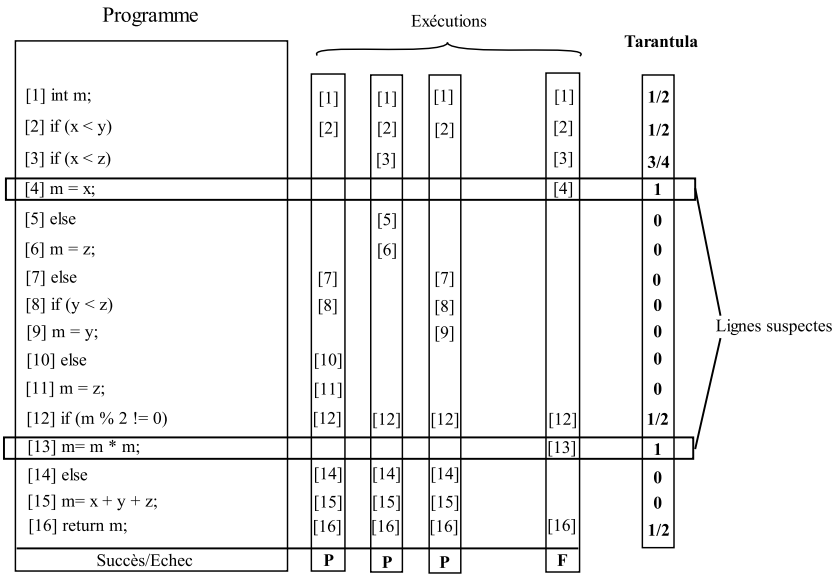

FIGURE 8.7 – Résultats de l'application de Tarantula sur l'exemple de la figure 8.1

L'indice de couleur (color) : s'étend de la couleur verte à la couleur rouge, où une instruction colorée en vert est supposée être correcte, alors qu'une instruction colorée en rouge est suspecte. Le calcul d'une couleur pour une instruction se définit par le calcul de son degré de suspicion.

L'indice de brillance (brightness) : est utilisé pour donner de l'importance au nombre d'apparitions d'une instruction dans des spectres du même verdict. Par exemple, si une instruction apparaît beaucoup dans des spectres d'exécutions en succès, elle sera alors colorée en vert très brillant. Une instruction qui apparaît beaucoup dans des spectres d'exécutions en échec sera colorée en rouge très brillant.

8.3.3 Jaccard

Jaccard est le nom du cœfficient utilisé par l'outil Pinpoint [24]. Ce dernier est un détecteur de panne destiné aux services internet, grands et dynamiques, tels que les services de messagerie et les moteurs de recherche.

Comme il est illustré dans la Figure 8.8, le principe de l'outil repose sur une analyse comportementale du système [24]. La détection des pannes se fait par l'observation d'une variation anormale du comportement.

Pinpoint permet de détecter une défaillance, mais aussi de localiser son origine.

Pour la localisation de l'origine de la défaillance, l'outil combine l'utilisa-

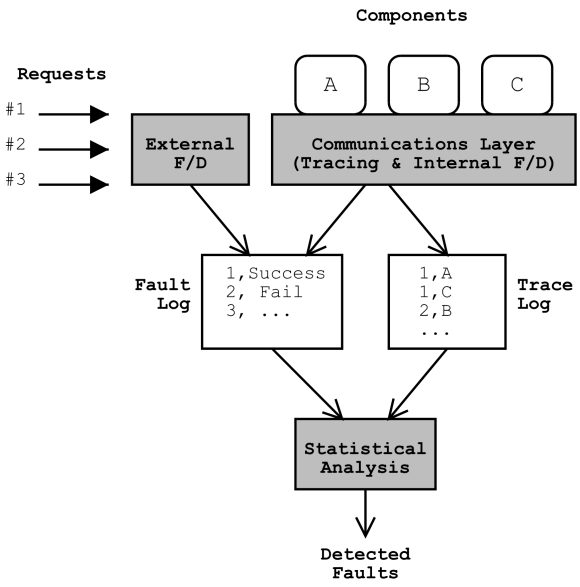

FIGURE 8.8 – Le framework Pinpoint

tion de la SBFL avec une forme spécifique de détection d'erreur. La détection d'erreur est réalisée en utilisant les informations provenant du framework J2EE, telles que les exceptions interceptées. Ainsi, cette détection d'erreur sert à classifier les spectres en spectres d'exécutions en échec ou en succès. Après la classification des spectres, Pinpoint calcule les degrés de suspicion des composants du système, cela en utilisant le cœfficient de Jaccard.

Le calcul du degré de suspicion en utilisant Jaccard est défini comme suit :

$$Jaccard(j) = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)}$$

Le résultat de la localisation de faute en utilisant le cœfficient Jaccard sur l'exemple de la Figure 8.1 est illustré dans la Figure 8.9.

Le cœfficient de Jaccard est bien connu dans le domaine du « data-clustering » (voir par exemple [39]).

8.3.4 Ample

Ample (Analyzing Method Patterns to Locate Errors) [28] est un système destiné aux programmes orientés objets. Il permet d'identifier les classes défaillantes. Cet outil manipule des spectres de type *statement-hit* restreints aux séquences d'appels de méthodes des objets d'une classe [10].

Ample attribut un poids à chaque séquence d'appels. Ce poids représente la corrélation entre la présence ou l'absence d'une séquence d'appels de méthodes

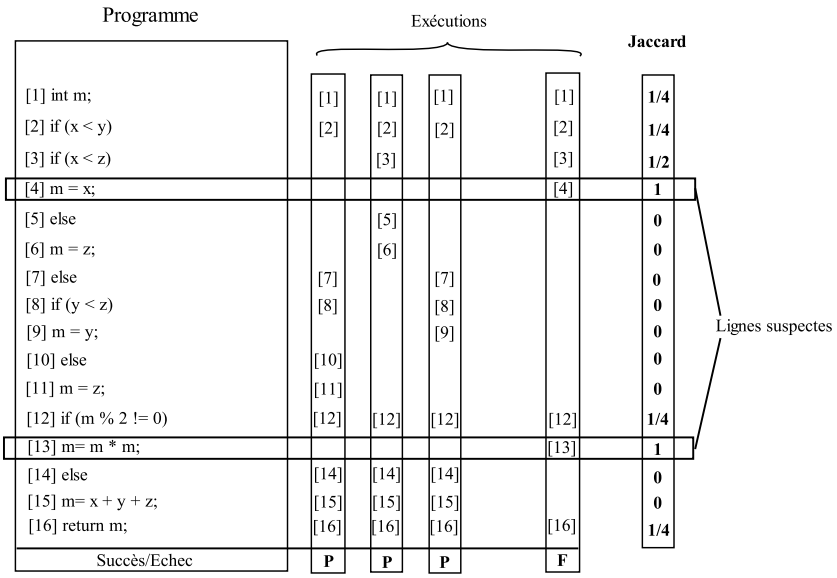

FIGURE 8.9 – Résultats de l'application de Jaccard sur l'exemple de la figure 8.1

et la détection d'une erreur. Ces poids sont ensuite utilisés pour calculer une moyenne sur l'ensemble des séquences d'appels des méthodes d'une classe, pour obtenir ainsi un poids pour chaque classe. Les classes avec des poids élevés sont plus susceptibles de contenir la faute qui provoque la défaillance détectée.

Bien que le calcul des poids de séquence d'appels dans Ample soit basé sur le calcul de leurs degrés de suspicion, le diagnostic se fait au niveau des classes. Donc les degrés de suspicion calculés ne servent pas à identifier les séquences d'appels de méthodes suspectes, mais sont uniquement utilisés pour recueillir des indications sur les classes.

Ample calcule le degré de suspicion (le poids) pour une séquence d'appels de méthode j comme suit :

$$Ample(j) = \left| \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j)} - \frac{a_{10}(j)}{a_{10}(j) + a_{00}(j)} \right|$$

Le résultat de la localisation de faute en utilisant le cœfficient Ample sur l'exemple de la Figure 8.1 est illustré dans la Figure 8.10.

8.3.5 Ochiai

Le cœfficient Ochiai est utilisé pour le calcul de similarités génétiques en biologie moléculaire. Dans [9, 17], les auteurs ont démontré que l'utilisation du cœfficient de similarité Ochiai fournissait une meilleure mise en évidence

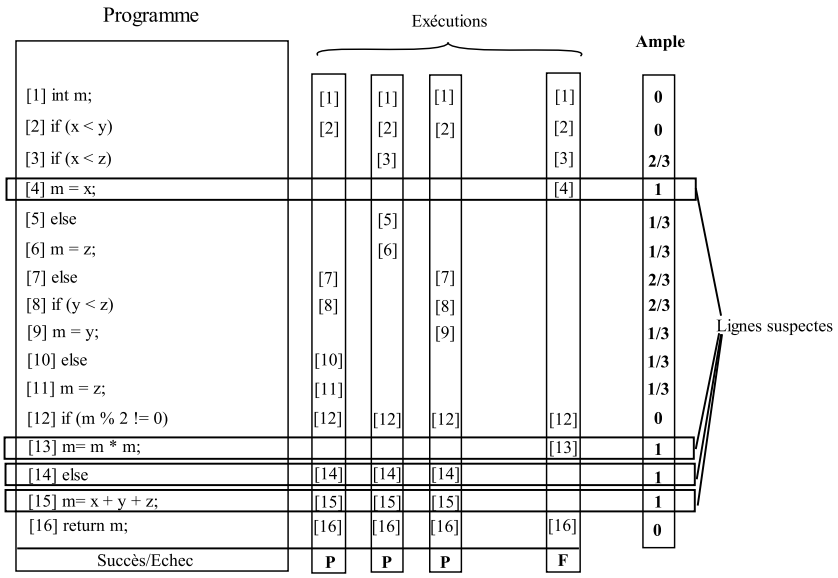

FIGURE 8.10 – Résultats de l'application d'Ample sur l'exemple de la figure 8.1

des suspects comparé aux trois autres cœfficients mentionnés précédemment, permettant ainsi un meilleur diagnostic pour la localisation de faute. Les résultats de [9] démontrent notamment que, contrairement aux autres cœfficients, la supériorité des performances du cœfficient Ochiai à identifier les suspects, ne dépend pas de la qualité ou de la quantité de spectres.

La formule mathématique permettant de calculer le degré de suspicion avec Ochiai est défini comme suit :

$$Ochiai(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) \times (a_{11}(j) + a_{10}(j))}}$$

Le résultat de la localisation de faute en utilisant le cœfficient Ochiai sur l'exemple de la Figure 8.1 est illustré dans la Figure 8.11.

8.3.6 Le cœfficient O^p

Le cœfficient O^p a été proposé par Naish et al. [58]. La localisation de faute en utilisant ce cœfficient est considérée comme optimal sur les programmes respectant le modèle ITE2, c'est-à-dire, constitué d'une séquence de deux constructions de type if-then-else comme illustré dans la Figure 8.12.

La formule mathématique permettant de calculer le degré de suspicion avec O^p est définit comme suit :

$$O^p(j) = a_{11} - \frac{a_{10}}{a_{10} + a_{00} + 1}$$

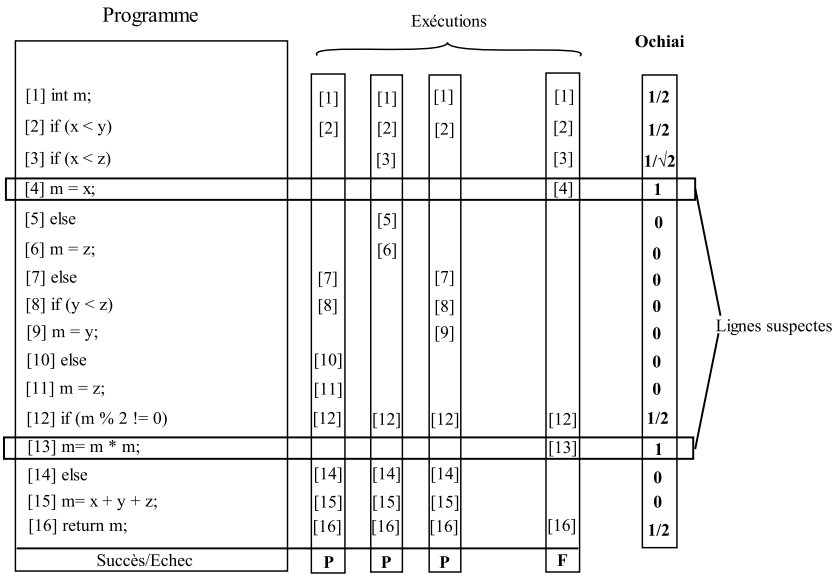

FIGURE 8.11 – Résultats de l'application d'Ochiai sur l'exemple de la figure 8.1

```
if(t1())
   s1();
else
   s2();

if(t2())
   s3();
else
   s4();
```

FIGURE 8.12 – Program segment If-Then-Else-2 (ITE2)

Le résultat de la localisation de faute en utilisant le cœfficient O^p sur l'exemple de la Figure 8.1 est illustré dans la Figure 8.13.

8.3.7 Algorithme général de la SBFL

Dans ce chapitre nous avons présenté cinq cœfficients de similarité. Ces cœfficients sont utilisés pour localiser les fautes à partir de spectres d'exécution qui mènent ou non à une défaillance. La façon d'utiliser ces cœfficients pour générer un classement des instructions est globalement la même selon les approches, et correspond à l'Algorithme 5. Cet algorithme prend comme

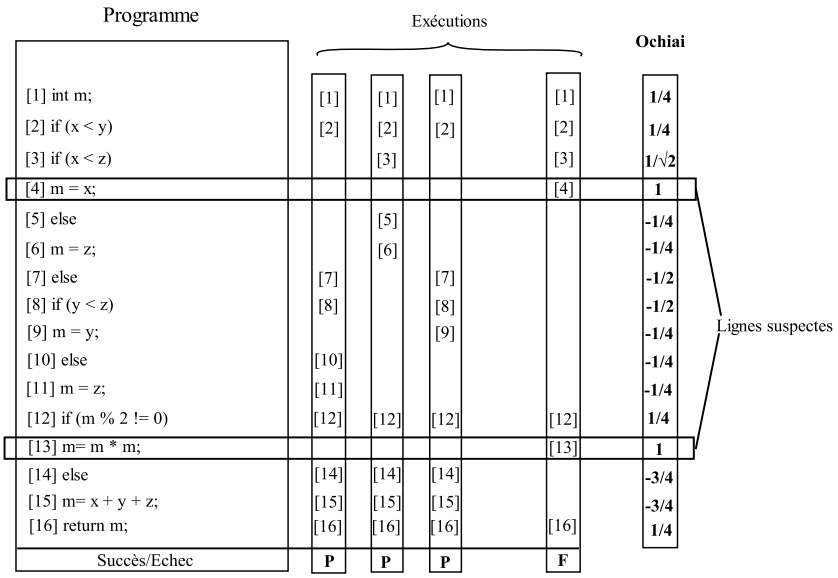

FIGURE 8.13 – Résultats de l'application du cœfficient O^p sur l'exemple de la figure 8.1

entrée : un programme P, une suite de tests T, et un cœfficient de calcul de suspicion s. Il génère comme sortir un rapport de diagnostic, où les éléments du programme P sont classés par ordre décroisent de suspicions (ligne 26). Afin de calculer le degré de suspicion de chaque élément de P, la matrice de spectres MS et le vecteur de verdicts e sont construits, cela en exécutant le programme P avec les différents cas de tests de T et en analysant la conformité des résultats fournis (ligne 10). Pour chaque élément de P le degré de suspicion est calculé en utilisant le cœfficient s donné comme entrée (de la ligne 11 à 26).

8.4 Inconvénients de la SBFL

La localisation de faute basée sur les spectres présente deux inconvénients : elle nécessite la présence d'un oracle et ne traite qu'une défaillance à la fois. Ces deux points sont discutés ci-dessous.

La présence d'un oracle : pour la plupart des techniques de type SBFL, afin d'évaluer les instructions du programme, les verdicts en terme de succès ou d'échec pour les exécutions sont nécessaires. En d'autres termes, les techniques de type SBFL nécessitent la présence d'un oracle de tests, qui vérifie l'exactitude des résultats du programme. Cependant, dans de nombreuses applications, il est impossible, ou trop coûteux, de vérifier l'exactitude des résultats calculés. Pour de telles applications, un spectre

Algorithme 5: SBFL Algorithm(P, T, s)

```
1  C ← Get_NumOfComponents(P);
2  for j = 1 to C do
3  |  a11(j) ← 0;
4  |  a10(j) ← 0;
5  |  a01(j) ← 0;
6  |  a00(j) ← 0;
7  |  Susp[j] ← 0;
8  end
9  (MS, e) ← RunProgram(P, T);
10 for i = 1 to |T| do
11 |  for j = 1 to C do
12 |  |  if MS[i, j] = 1 ∧ e[i] = 1 then
13 |  |  |  a11(j) ← a11(j) + 1;
14 |  |  else if MS[i, j] = 0 ∧ e[i] = 1 then
15 |  |  |  a01(j) ← a01(j) + 1;
16 |  |  else if MS[i, j] = 1 ∧ e[i] = 0 then
17 |  |  |  a10(j) ← a10(j) + 1;
18 |  |  else if MS[i, j] = 0 ∧ e[i] = 0 then
19 |  |  |  a00(j) ← a00(j) + 1;
20 |  |  end
21 |  end
22 end
23 for j = 1 to C do
24 |  Susp[j] ← s(a11(j), a10(j), a01(j), a00(j));
25 end
26 returnDESC_Sort(Susp);
```

d'exécution n'est pas associé à un verdict d'exécution. Par conséquent, l'information est incomplète pour l'évaluation des risques, et de ce fait, la SBFL devient inapplicable pour de telles applications.

Le contexte de notre travail et un exemple de situation où les applications ne disposent pas d'oracle, c'est-à-dire qu'il n'est pas possible de vérifier l'exactitude des résultats de l'exécution du programme embarqué, spécialement lorsqu'il s'exécute sur un microcontrôleur. Par conséquent, il est nécessaire de définir des hypothèses sur l'exactitude des exécutions et de procéder à la localisation de faute à partir de ces hypothèses.

Le nombre de défaillances : la majorité des techniques de type SBFL fournissent un bon résultat dans le cas où il n'y a qu'une seule défaillance.

Cependant, elles ne traitent pas le cas de multiples défaillances, qui est généralement beaucoup plus compliqué. Dans ce genre de situation certaines défaillances peuvent masquer ou en camoufler d'autres. Ainsi un échec observé peut être dû à l'interaction entre une série de fautes. Récemment, plusieurs techniques ont été proposées pour remédier à ce problème [78, 40, 12, 11]. Cependant, dans cette thèse nous nous intéressons à la localisation d'une faute à la fois.

Pour aider à la localisation de faute à partir d'une trace d'exécution, notre idée est d'adapter les approches présentées dans ce chapitre. Ce travail est présenté dans le chapitre suivant.

Chapitre 9

Localisation de faute en utilisant une seule trace d'exécution

Sommaire

9.1 Introduction

En raison de la nature cyclique de la plupart des programmes embarqués, les traces extraites sont constituées de répétitions de séquences d'instructions. Dans notre contexte industriel, le plus souvent, l'enregistrement de la trace est arrêté quand un événement empêchant la poursuite de l'exécution apparaît (par exemple, une exception est levée). Ainsi, ce mécanisme assure la présence d'un comportement anormal au moins dans les derniers événements enregistrés. Le problème que nous ont posé nos partenaires industriels est de proposer des approches permettant d'aider à la localisation d'une faute à partir d'une seule trace d'exécution se terminant sur une exception.

La première idée qui vient naturellement à l'esprit est d'appliquer les approches de localisation de faute basées les traces, que nous avons présenté dans le chapitre 8. Il existe de nombreuses techniques qui utilisent les traces d'exécutions pour la localisation de faute, par exemple Tarantula [42], ou Ochiai [9]. Cependant, ces techniques exploitent un *ensemble* de spectres d'exécution. Or nous n'avons qu'une seule trace à notre disposition.

En effet, un microcontrôleur est le plus souvent branché à d'autres dispositifs, tel que des capteurs de température par exemple. Par conséquent, le comportement d'un microcontrôleur est le plus souvent influencé par l'environnement dans lequel il est installé. Ainsi, lorsque nous essayons de générer d'autres traces du programme embarqué, il est difficile de reproduire l'environnement dans lequel le microcontrôleur était installé, lorsque la trace en question a été enregistrée.

Dans cette partie, nous proposons une adaptation des méthodes de localisation de faute pour tenir compte de notre contexte, où une seule trace d'exécution cyclique est disponible [16].

Notre approche utilise principalement le cœfficient Ochiai. Cependant, notre outil CoMET permet l'utilisation des autres cœfficients présentés dans cette partie de la thèse. Cela nous a permis d'évaluer l'efficacité des différents cœfficients en utilisant notre outil. Le choix d'utilisation du cœfficient Ochiai est appuyé par les résultats de notre évaluation expérimentale, présentée dans le chapitre 10, et qui montrent l'efficacité de ce cœfficient par rapport aux autres cœfficients discutés dans cette partie. Notre outil est présenté chapitre 16

9.2 Exploitation de cycles pour la localisation de faute

L'adaptation des approches de localisation de faute présentées précédemment, en n'utilisant qu'une seule trace d'exécution, est basée sur l'exploitation de l'aspect cyclique du programme à l'origine de la trace traitée. Par conséquent, il est nécessaire de diviser la trace en cycles, où chaque cycle définit une exécution de la boucle principale du programme embarqué. Il est important de rappeler que cette division repose sur la localisation de l'événement représentant la tête de la boucle principale dans la trace. Cet événement particulier est appelé *loopheader*, noté lh. Dans l'exemple illustré dans la Figure 9.1 le symbole a représente le *loopheader* dans la trace, ainsi le découpage de la trace en cycles fourni comme résultat : $[abcd][abcef][abcd][abccf][abcg]$.

Afin de pouvoir utiliser les cycles d'une trace d'exécution pour la localisation de faute, notre approche est basée sur l'analogie entre les exécutions du programme et les cycles. Ainsi, dans cette partie, nous supposons que les cycles du programme sont analogues aux exécutions du programme. Nous avons constaté dans nos expérimentations que cette hypothèse est valable (spectre $S2$ de la Figure 9.2), car de nombreux programmes embarqués sont des programmes (quasiment) sans mémoire ou bien avec des variables locales au corps de la boucle principale. Dans le cas contraire, notre hypothèse reste

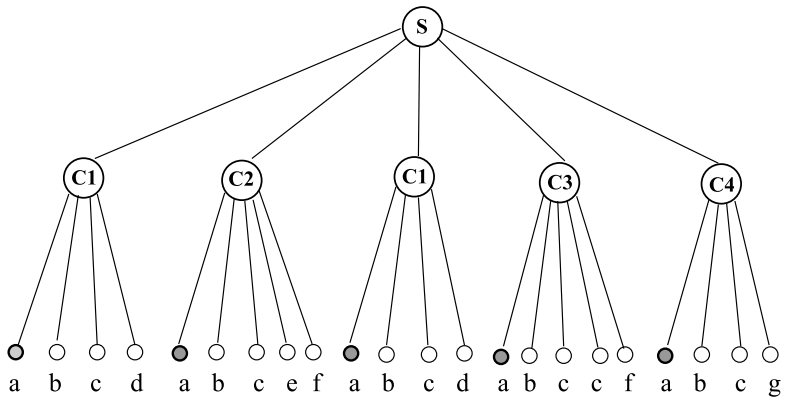

FIGURE 9.1 – Trace divisée en cycles

correcte, car il suffit de considérer le(s) résultat(s) d'un cycle précédent (ou plusieurs), comme une entrée pour la nouvelle exécution.

Un exemple assez simple d'une situation sans mémoire globale concerne un programme (fournit avec le kit STM-32) permettant d'allumer une LED si l'utilisateur appuie sur un bouton spécifique. Si l'utilisateur appuie sur le bouton B_1 le programme allume la LED L_1, s'il appuie sur le bouton B_2 le programme allume la LED L_2. Ainsi, dans cet exemple, l'exécution d'un cycle est indépendante du résultat d'exécution du cycle précédent.

En outre, parce que le comportement anormal apparaît nécessairement à la fin de la trace, impliquant ainsi l'arrêt de l'enregistrement de la trace, nous supposons que le dernier cycle de la trace correspond à une exécution en échec (cycle $C4$ de la Figure 9.1), les autres cycles sont considérés comme des exécutions en succès (cycle $C1$, $C2$ et $C3$ de la Figure 9.1). Par conséquent, la défaillance et l'échec de l'exécution se produisent dans un même cycle : le dernier.

Les exécutions d'un programme sont généralement indépendantes, surtout si elles sont obtenues par l'exécution d'une suite de tests, qui se compose habituellement de cas de tests indépendants. Contrairement aux exécutions indépendantes, il existe des cas où les multiples cycles d'une même exécution peuvent interagir entre eux. Dans cette partie nous considérons que ces cycles sont indépendants. Nous sommes donc conscients que l'analogie n'est pas parfaite et que l'indépendance entre les cycles ne peut pas être garantie. Cependant, comme nous l'avons dit plus tôt, nos expérimentations réalisées à postériori, montrent que cette hypothèse est pertinente. Dans la partie 3, nous proposons une autre approche permettant de relâcher cette hypothèse.

En se basant sur l'hypothèse d'indépendance des cycles et après avoir effectué le découpage de la trace en cycles, la localisation de faute est effectuée

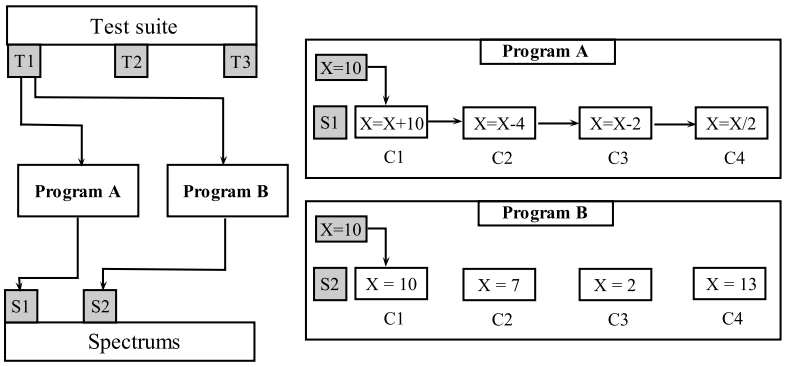

FIGURE 9.2 – Deux spectres avec cycles dépendants et indépendants

en utilisant la méthode du *plus proche voisin* ou l'une des méthodes utilisant un *score de suspicion*.

9.3 Le voisin le plus proche dans une trace

L'approche du plus proche voisin, décrite dans le section 8.2.3, repose sur la comparaison entre le spectre de l'exécution en échec et le spectre de l'exécution en succès qui lui ressemble le plus. Dans cette section, nous proposons une adaptation de la méthode dans le cadre de l'utilisation d'une seule trace. Comme nous l'avons dit plus tôt, cette adaptation repose sur le découpage de la trace en cycles (Figure 9.1), où chaque cycle de la trace représente une exécution indépendant de la boucle principale du programme embarqué. Il est important de rappeler que dans notre contexte de travail, l'enregistrement de la trace est interrompu dès l'apparition d'un comportement anormal du microcontrôleur. Par conséquent, la comparaison est faite entre le dernier cycle et le cycle qui lui ressemble le plus dans la trace. Le plus proche voisin est détecté en utilisant une mesure de distance, appelée *distance de Hamming*, décrite ci-dessous.

9.3.1 Utilisation de la distance de Hamming

La détection du cycle dont l'exécution est en succès, qui correspond le plus au cycle dont l'exécution est en échec, est basée sur l'utilisation de la distance de Hamming.

La distance de Hamming a été conçue à l'origine pour la détection et la correction d'erreurs dans le domaine de la communication numérique [36], où elle est définie comme étant le nombre de bits différents entre deux vecteurs de bits [36]. Cette mesure est notamment utilisée dans d'autres domaines pour

quantifier la différence entre deux chaînes de même dimension. Par conséquent, plus la distance de Hamming est petite, plus la similitude est grande.

Afin de trouver le voisin le plus proche du dernier cycle, un vecteur binaire est associé à chaque cycle de la trace. Les compteurs ordinaux (Pcs) sont utilisés pour diviser la trace d'exécution en cycles, par conséquent, ils sont également utilisés pour calculer la distance de Hamming.

	C_1	C_2	...	C_n
PC_1	$x_{1,1}$	$x_{1,2}$	...	$x_{1,n}$
PC_2	$x_{2,1}$	$x_{2,2}$	...	$x_{2,n}$
⋮	⋮	⋮	⋮	⋮
PC_m	$x_{m,1}$	$x_{m,2}$	...	$x_{m,n}$

TABLE 9.1 – Construction de vecteurs binaires

Soit $PCs = \{PC_1, PC_2, \ldots, PC_m\}$ l'ensemble des compteur ordinaux dans une trace d'exécution et soit $C = \{C_1, C_2, \ldots, C_n\}$ l'ensemble des cycles de cette trace. Comme illustré dans la Table 9.1, pour chaque cycle C_j ($j \in [1..n]$), un vecteur binaire de taille $|PCs|$ est construit, où pour $i \in [1..m]$:

- $x_{i,j} = 1$ si PC_i apparaît dans le cycle C_j.
- $x_{i,j} = 0$ si PC_i n'apparaît pas dans le cycle C_j.

Cela se définit formellement comme suit :

$$\forall C_{j(\in[1..n])} \in C, \forall PC_{i(\in[1..m])} \in PCs : x_{i,j} = \begin{cases} 1 & \text{si } \exists k \in [1..|C_j|] : (C_j)_k = PC_i \\ 0 & \text{sinon} \end{cases} \tag{9.1}$$

La distance de Hamming calcule la similarité entre deux vecteurs binaires de même taille. Cependant, les cycles d'une trace ne contiennent pas forcément le même nombre d'événements, comme dans l'exemple de la Figure 9.1, où le cycle $C1$ contient quatre symboles et le cycle $C2$ contient cinq symboles. Ainsi, si les vecteurs binaires sont construits en se basant sur le nombre d'événement que contiennent leurs cycles respectifs, il sera impossible de calculer la distance de Hamming. Par conséquent, pour résoudre cette problématique, la construction d'un vecteur binaire pour un cycle est réalisée en utilisant l'ensemble d'événements de la trace et non seulement l'ensemble d'événements du cycle. La construction des vecteurs binaires des cycles de l'exemple de la Figure 9.1 est illustrée dans la Table 9.2, où la ligne $Ham(C_4)$ représente le calcul de la distance de Hamming avec le cycle C_4.

	C_1	C_2	C_3	C_4
a	1	1	1	1
b	1	1	1	1
c	1	1	1	1
d	1	0	0	0
e	0	1	0	0
f	0	1	1	0
g	0	0	0	1
Ham(C_4)	2	3	2	/

TABLE 9.2 – Construction des vecteurs binaires pour l'exemple de la Figure 9.1

Le calcul de la distance de Hamming entre deux cycles C_j et C'_j se définit donc formellement comme suit :

$$Ham(C_j, C_{j'}) = \sum_{i=1}^{m} a_i \text{ où } a_i = \begin{cases} 1 & \text{si } x_{i,j} \neq x_{i,j'}, \\ 0 & \text{sinon.} \end{cases} \tag{9.2}$$

Il est important de préciser que dans notre contexte, souvent, les cycles ne contiennent pas le même nombre d'événements. Par conséquent, notre calcul de la distance de Hamming ne prend pas en compte le nombre de répétitions d'un événement dans un cycle, mais cherche à déterminer si un événement (compteur ordinal) apparaît ou non dans la trace d'exécution. Cela permet de fixer une même taille pour tous les vecteurs binaires représentant les cycles, afin de calculer les différences. Par exemple, dans la Table 9.2, la construction du vecteur binaire du cycle $C3$ ignore le fait que l'événement c se répète deux fois. L'ordre dans lequel les événements apparaissent dans un cycle n'est pas pris en compte pour le calcul de la distance.

9.3.2 Application du *plus proche voisin*

Une fois que les calculs des valeurs de similarité entre le dernier cycle, noté C_n, et les autres cycles de la trace sont effectués, il est possible d'obtenir deux résultats. Le premier résultat consiste à n'avoir qu'un seul cycle comme voisin le plus proche, c'est-à-dire que le calcul des valeurs de similarité a mis en évidence un seul cycle, dont la distance de Hamming est la plus petite. Cela se traduit formellement comme suit :

$$PPV(C_n) = \min\{Ham(C_j, C_n) \mid (j \in [1..n-1])\} \tag{9.3}$$

Un exemple de ce cas est illustré dans la Figure 9.3. Dans l'exemple de cette figure, l'exécution en échec est représentée par le cycle C_8, et les exécu-

tions en succès sont les cycles de l'ensemble $\{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$. Dans cet exemple, le cycle C_7 représente l'exécution en succès qui a la plus grande similarité (la plus petite distance de Hamming) avec C_8. Par conséquent, l'approche du *plus proche voisin* est appliquée sur ces deux cycles pour mettre en évidence leurs différences.

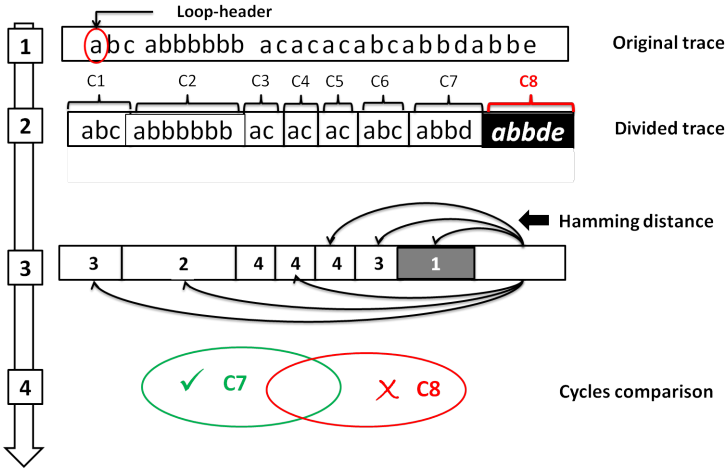

FIGURE 9.3 – Processus de localisation de faute en utilisant une seule trace

Le second résultat du calcul des valeurs de similarité consiste à avoir un ensemble de voisins, où pour un cycle avec une exécution en échec, deux cycles ou plus peuvent être considérés comme voisins plus proches. Dans l'exemple de la Table 9.2, les deux cycles C_1 et C_3 ont la même plus petite valeur 2. Par conséquent, il n y a pas qu'un seul voisin plus proche, mais deux. Dans ce cas il existe deux manières de procéder à la localisation de faute :

Sélection d'un seul voisin : ce cas consiste à définir un seul cycle parmi l'ensemble des voisins les plus proches comme étant le voisin le plus proche du dernier cycle. L'application des deux approches union et intersection model sur le cycle choisi et le cycle avec l'exécution en échec permet de mettre en évidence leurs différences. Cette étape est donc une simplification du traitement en choisissant un cycle parmi l'ensemble des cycles considérés comme les plus proches voisins. Dans l'exemple de la Table 9.2 ceci reviendrait à choisir le cycle C_2 (ou le cycle C_3) pour procéder à la localisation de faute.

Utilisation de l'ensemble des voisins les plus proches : ce cas consiste à utiliser l'ensemble des cycles considérés comme voisins les plus proches du dernier cycle, pour la localisation de faute. Dans ce cas, des méthodes de localisation de faute utilisant les scores de suspicion définiront l'ensemble d'événements suspects, ceci en utilisant tous les cycles considérés

comme voisins plus proche. Cette approche est détaillée dans la sous-section suivante.

9.4 Calcul du score de suspicion en utilisant une seule trace

L'adaptation présentée dans cette section prend en compte l'importante taille de la trace d'exécution et le grand nombre de cycles qu'elle peut contenir. Il est important de préciser qu'une grande partie des traces des logiciels embarqués contiennent des cycles très distincts, qui correspondent à des comportements très différents. Ceci est dû au fait que la programmation cyclique tend à répartir les différentes tâches sur des cycles différents.

Afin de réduire le coût du calcul des scores de suspicion des événements dans une trace, il est nécessaire de limiter le nombre de cycles à analyser. Cependant, il est important de préciser que le choix des cycles à analyser est important. Par conséquent, il est nécessaire d'effectuer un filtrage sur l'ensemble des cycles de la trace. Après la sélection des cycles à analyser, les scores de suspicion pour les composants de la trace sont calculés, ceci en utilisant les techniques de localisation de faute, comme le montre la Figure 9.4.

9.4.1 Filtrage

Dans une trace d'exécution, certains cycles présentent des comportements très différents des autres. Une telle disparité peut nuire à la localisation de faute en utilisant les scores de suspicion. L'exemple de la Table 9.3 illustre ce type de nuisance à la localisation de faute, où un grand nombre d'événements du dernier cycle (le cycle avec l'exécution en échec), noté C_5, sont considérés comme suspects. La cause de la sélection de la plupart des événements du dernier cycle comme des événements suspects est dû au fait que ces événements n'apparaissent pas souvent dans les cycles considérés comme des exécutions en succès. Ainsi une sélection non pertinente des suspects influe sur l'effort fourni pour la localisation de faute, c'est-à-dire plus il y a de suspects, plus l'effort est important.

Pour atténuer ce phénomène lié à la disparité de certains cycles, une première étape, nommée « processus de filtrage », permet de déterminer à l'aide de la distance de Hamming les cycles les plus ressemblants au cycle erroné. Cette étape a pour but de réduire l'effort fourni pour la localisation de faute, ceci en offrant une meilleure évaluation des valeurs de suspicion. La Table 9.4 illustre les valeurs de calcul de la distance de Hamming entre le cycle C_5 et les autres cycles.

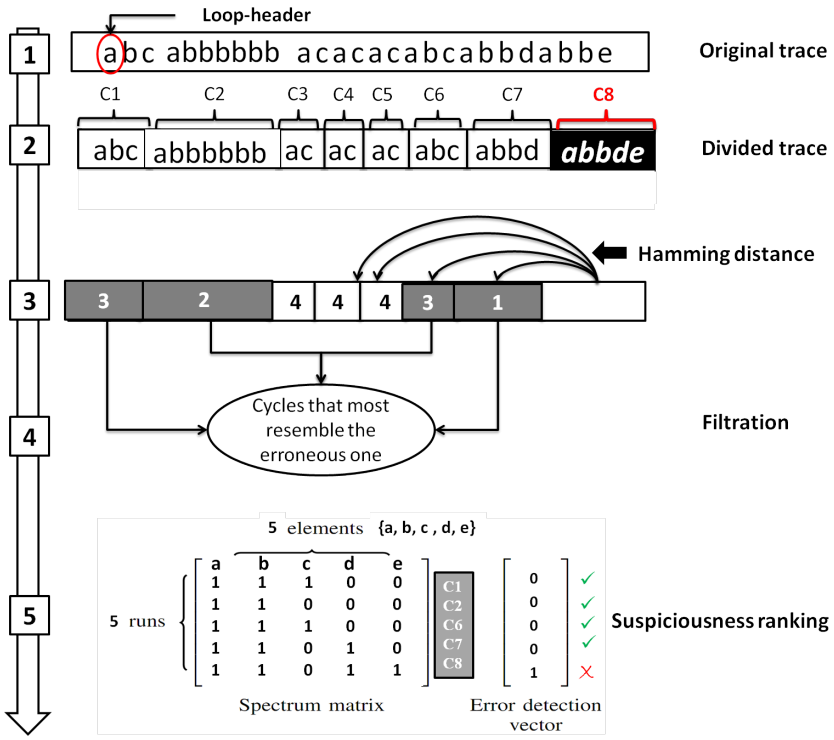

FIGURE 9.4 – Calcul de suspicion en utilisant une trace

	C_1	C_2	C_3	C_4	C_5	Tarantula
a	1	1	1	1	1	0.5
b	1	1	1	1	0	0
c	1	1	1	0	0	0
d	1	1	1	1	0	0
e	0	0	0	1	1	0.8
f	0	0	0	0	1	1
g	0	0	0	1	1	0.8
h	0	0	0	0	1	1
verdict	P	P	P	P	F	/

TABLE 9.3 – Localisation de faute pour des cycles à comportement différent

	C_1	C_2	C_3	C_4	C_5
Hamming	7	7	7	4	/

TABLE 9.4 – Calcul de la distance de Hamming pour l'exemple de la Table 9.3

Afin de filtrer les cycles, après le calcul des valeurs de distance, une borne

	C_4	C_5	Tarantula
a	1	1	0.5
b	1	0	0
c	0	0	0
d	1	0	0
e	1	1	0.5
f	0	1	1
g	1	1	0.5
h	0	1	1
verdict	P	F	/

TABLE 9.5 – Localisation de faute après filtrage de cycles

supérieure peut être définie par l'ingénieur. Notre approche de localisation de faute, en s'appuyant sur les résultats de l'évaluation présentée dans la section 10.4, utilise par défaut 30% des cycles de la trace d'exécution. Ces cycles utilisés sont les voisins les plus proches du cycle fautif. Donc, en se basant sur le nombre défini par l'utilisateur, concernant les cycles à utiliser par notre approche, une borne supérieure est calculée. Seuls les cycles dont les valeurs de distance sont strictement inférieures à cette borne sont sélectionnés pour servir dans la localisation de faute.

Pour une trace d'exécution σ, son ensemble de cycles est noté C_σ et son cycle fautif est noté C_n. L'ensemble des plus proches voisins du cycle fautif, noté neighbors, en utilisant la borne supérieure b, est défini comme suit :

$$\mathsf{neighbors}(C_n, b, \sigma) = \{C_j \in C_\sigma \mid Ham(C_j, C_n) < b\}\,.$$

Dans l'exemple de la Table 9.4, le taux de cycles à utiliser est fixé à 20%, ce qui correspond à un seul cycle. Le cycle le plus proche est alors C_4. Par conséquent, les cycles qui serviront dans la localisation de faute sont les cycles C_4 et C_5, comme illustré dans la Table 9.5.

9.4.2 Classement de suspects

Le calcul de la suspicion d'un élément du spectre repose sur la comparaison entre le vecteur d'erreur avec le vecteur binaire de l'élément. Les suspects sont classés dans un ordre décroissant, c'est-à-dire que l'élément avec la valeur de suspicion la plus grande est analysé en premier, ainsi de suite. Selon le cœfficient utilisé, la valeur de suspicion d'un élément peut changer. Cependant, dans la plupart des cas le classement des suspects reste le même.

En utilisant l'exemple de la Table 9.3, où il n'y a pas eu de filtrage de

Suspicion	Élément(s)
1	**h, f**
0.8	**g, e**
0.5	**a**
0	**b, c, d**

TABLE 9.6 – Classement des suspects sans filtrage

Suspicion	Élément(s)
1	**h, f**
0.5	**g, e, a**
0	**b, c, d**

TABLE 9.7 – Classement des suspects avec filtrage

cycles, le classement des suspects est illustré dans la Table 9.6. D'après les résultat du classement, l'analyse se fait en commençant par l'événement h, ceci en analysant l'ensemble des instructions reliées à lui. Si la faute est localisée l'analyse est terminée, sinon le prochain suspects (événement f) et l'ensemble des instructions liées à lui sont analysés, ainsi de suite.

Le classement des résultats illustrés dans la Table 9.5, après l'étape de filtrage de cycles, est illustré dans la Table 9.7. Il est important de remarquer que les événements g et e ne sont plus à un niveau plus élevé mais au même niveau de suspicion que l'événement a, ce qui est plus correcte.

Cette étape de filtrage de cycles permet donc d'augmenter la pertinence et la précision du diagnostic. Par conséquent, l'effort d'analyse fourni par l'ingénieur pour identifier la faute est réduit. Elle permet également d'augmenter la rapidité du processus de localisation de faute, parce qu'il y a moins de cycles à analyser.

L'évaluation de cette approche de filtrage ainsi que de celle de localisation de faute sont présentées dans le chapitre suivant.

Chapitre 10

Évaluations

Sommaire

Dans ce chapitre, nous présentons l'évaluation de l'approche de localisation de faute. Il est important de rappeler que l'approche à évaluer dans cette partie traite une seule trace, où la cause de la défaillance et la défaillance elle même se trouvent dans le dernier cycle. L'évaluation discutée dans cette partie consiste essentiellement à appliquer notre approche sur des programmes contenant des fautes connues, dans le but de mesurer la qualité du diagnostic fourni par notre approche de localisation de faute. Cette évaluation a été rendue possible grâce à notre outil nommé CoMET, présenté dans le chapitre 16.

10.1 Programmes et erreurs

Nous rappelons que notre travail s'insère dans le cadre d'un projet FUI IO32 [1]. Les traces utilisées pour évaluer notre approche proviennent de 13 programmes embarqués, qui ont été fournis par deux de nos partenaires dans ce projet : STMicroelectronics [2] et EASii-IC [3]. Chacun des 13 programmes permet à l'utilisateur d'interagir avec un microcontrôleur, ceci en utilisant soit un écran LCD, soit les 4 boutons du microcontrôleur. Quand l'utilisateur appuie sur un bouton, le microcontrôleur exécute un traitement spécifique. Si le traitement se termine sans erreur, le voyant LED correspondant au bouton pressé est allumé et un message est affiché sur l'écran LCD. Dans ce qui suit le programme numéro i est noté P_i. Des informations concernant les 13 programmes ainsi que leurs traces sont fournis dans la Table 10.1.

1. http ://io32.forge.imag.fr/
2. http ://www.st.com
3. http ://www.easii-ic.com/

Programme	Taille	# Fichiers	Taille de la trace	# Lines
P1	14.4 Mo	165	95.2 Mo	1048579
P2	14.4 Mo	165	86.9 Mo	1045869
P3	143 Mo	151	25.1 Mo	280049
P4	218 Mo	152	21.1 Mo	237062
P5	143 Mo	151	18.7 Mo	207914
P6	140 Mo	151	21.7 Mo	240829
P7	207 Mo	158	94.3 Mo	1048577
P8	139 Mo	151	21.8 Mo	235788
P9	218 Mo	152	22.4 Mo	241404
P10	143 Mo	151	25.2 Mo	280298
P11	207 Mo	158	95.6 Mo	1048576
P12	14.4 Mo	165	92.2 Mo	1048573
P13	50.1 Mo	164	84.7 Mo	1047568

TABLE 10.1 – Information concernant les programmes et leurs traces

Les programmes P_1, P_2 et P_{12} exploitent la pile des appels. Lorsque l'utilisateur appuie sur le bouton B_1, un élément est empilé sur la pile. Le bouton B_2 permet de dépiler un élément de la pile. Le bogue de $P1$ consiste à ne pas vérifier si la pile est vide avant de dépiler un élément. Par conséquent, si l'utilisateur appuie sur B_2 plusieurs fois de suite sans appuyer sur B_1 on observe une défaillance. Le programme P_2 ne prend pas en charge le débordement de pile. Le programme P_{12} utilise une fonction récursive pour remplir la pile, cependant lorsque cette fonction est appelée avec un argument assez grand, la pile déborde et une interruption mémoire est générée.

La faute introduite dans le programme P_3 est de type syntaxique et concerne la vérification d'une valeur entière, où le signe >= est utilisé au lieu de ==.

Le bug dans le programme P_4 consiste à utiliser une adresse mémoire comme argument d'une fonction au lieu d'une variable entière, c'est-à-dire, soit `value_check(int x)` une fonction et `v` une variable entière, la faute réside dans le fait d'utiliser `value_check(&v)` au lieu de `value_check(v)`.

Le programme P_6 contient une erreur commune dans la programmation en C, cette erreur concerne la comparaison des chaînes de caractères. Pour comparer deux chaînes `str1` et `str2`, des fonctions de comparaison de chaînes comme `strcmp(str1, str2)` doivent être utilisées. Dans le programme P_6, la comparaison entre les chaînes `str1` et `str2` se fait en utilisant l'instruction `str1 == str2`. Une telle instruction compare les adresses mémoire au lieu de comparer les valeurs des chaînes. Cette erreur conduit à une interruption à un moment de l'exécution.

Les fonctions lcd_check() et led_check() dans le programme P_7 et P_{11}, respectivement, génèrent une interruption de l'exécution après n exécutions de la boucle principale.

La fonction value_check(**float** v) dans le programme P_8 vérifie si v > 0.3. A un certain moment de l'exécution, la valeur de v est modifiée par un *cast*, ce qui implique l'arrêt de l'exécution. Par exemple, si v vaut $0,4$ le *cast* renvoie la valeur 0, par conséquent la condition v > 0.3 n'est plus vérifiée, ce qui implique l'arrêt de l'exécution.

L'oubli du **break** dans un **case** et la faute existante dans le programme P_9. Si pour un **case** 1, qui exécute v = v * 3, l'instruction **break** est oubliée, le programme exécutera aussi l'instruction v = v * 2 du **case** 2. c'est-à-dire que la valeur de v sera multipliée par 6 au lieu de 3.

La boucle **for** dans le programme P_{10} contient un "**;**" de trop, comme il est illustré dans l'exemple suivant : **for** (i = 0; i < 10; i++);. Dans ce cas, chaque instruction dans la boucle **for** ne sera exécutée qu'une seule fois au lieu de de 10 fois.

Un *chien de garde* (en anglais *watchdog*) est un mécanisme qui déclenche une ré-initialisation du système dans le cas où le programme effectue pas une vérification régulière auprès du *chien de garde*, ceci a cause d'une erreur. Dans le programme P_{13}, si l'utilisateur appuie sur le bouton B_3, la vérification régulière auprès du *chien de garde* est désactivée, ainsi une ré-initialisation du système est déclenchée et une interruption matérielle est générée. Il nous a été rapporté par l'un de nos partenaires, que ce bogue est difficile à déboguer en raison du retard entre l'exécution de l'instruction défectueuse et l'interruption matérielle.

Chacun des 13 programmes est téléchargé sur une carte microcontrôleur STM32F107 EVAL-C (Figure 5) et exécuté. La trace produite par l'exécution est récupérée à l'aide d'une sonde Keil-UlinkPro [2]. Les fautes introduites dans les 13 programmes sont significatives, car ce sont des cas que nos partenaires rencontrent souvent lors du débogage.

10.2 Méthodologie

L'évaluation expérimentale de cette partie concerne deux approches de localisation de faute. Ces deux approches n'utilisent qu'une seule trace d'exécution. La première approche consiste à établir un diagnostic en utilisant qu'un cycle, ce dernier étant le voisin le plus proche du dernier cycle. La deuxième approche à évaluer utilise un ensemble de cycles, ceci dans le cas où plusieurs voisins plus proches existent, ou bien dans le cas où la localisation de faute est basée sur l'utilisation des scores de suspicion.

Afin d'évaluer nos deux approches, les six méthodes de localisation de faute suivantes ont été adaptées :

- Le voisin le plus proche,
- le classement des suspects en utilisant le cœfficient Tarantula,
- le classement des suspects en utilisant le cœfficient Jaccard,
- le classement des suspects en utilisant le cœfficient Ample,
- le classement des suspects en utilisant le cœfficient Ochiai.
- le classement des suspects en utilisant le cœfficient O^p

Pour chaque trace, un diagnostic est généré pour chaque application de l'une de ces six méthodes. Ainsi, les six diagnostics générés pour chaque trace sont comparés entre eux pour évaluer les performances de leurs méthodes respectives à localiser les fautes.

Pour évaluer les approches de localisation de faute, souvent le rang de l'instruction fautive est utilisé comme métrique [9, 41, 75]. La localisation de faute en utilisant la méthode du « plus proche voisin » ne définit pas un ordre spécifique sur l'ensemble des suspects. Ainsi, dans le but de ne pas désavantager cette méthode par rapport aux cinq autres utilisées dans notre expérimentation, nous mesurons l'*effort*. L'*effort* fourni pour localiser une faute, noté E, dépend du nombre d'instructions inspectées I (rang de l'instruction fautive) et du nombre total d'instructions T dans le programme. Il est défini comme suit :

$$E = \frac{I}{T}$$

Pour la localisation de faute en utilisant le classement des valeurs de suspicion, en cas d'égalité dans le classement, les instructions sont inspectées dans l'ordre inverse de la trace. Selon nos partenaires, cet ordre est utilisé souvent dans le débogage manuel, ainsi nous l'utilisons également pour la localisation de faute en utilisant la méthode du « plus proche voisin ».

10.3 Résultats

La Figure 10.1 représente les résultats de l'évaluation expérimentale, où l'axe des abscisses représente les programmes de P_1 à P_{13} et l'axe des ordonnées représente l'effort fourni pour la localisation de faute. Pour chaque programme, l'effort est calculé pour : le voisin le plus proche, Tarantula, Jaccard, Ample, Ochiai et O^p.

La localisation de faute en utilisation les traces d'exécution des programmes P_1, P_5, P_6, P_8 et P_{11} nécessite un effort manuel plus significatif pour la méthode du plus proche voisin. Tandis que l'effort fourni pour la localisation de

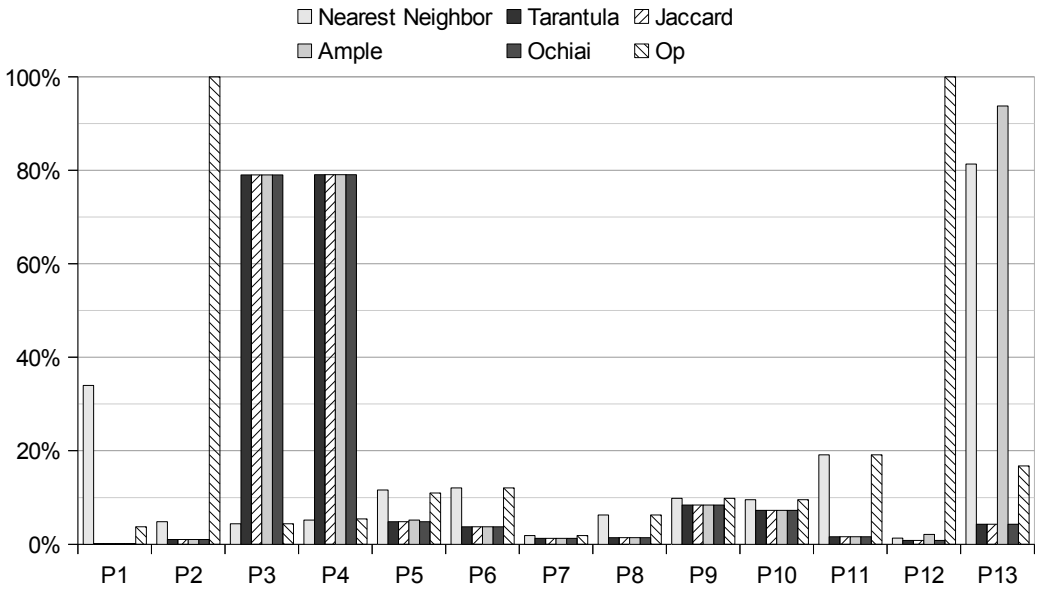

FIGURE 10.1 – Localisation de faute sur les 13 programmes et en utilisant les six methodes

faute en utilisant la méthode du plus proche voisin s'élève jusqu'à 35 %, les efforts fournis en utilisant les autres méthodes restent sous la barre des 5%. Ce décalage d'effort entre les méthodes utilisant des scores de suspicion et la méthode du plus proche voisin se justifie par le manque de classement des instructions suspectes dans cette dernière.

L'effort fourni pour la localisation de faute en utilisant la méthode du plus proche voisin pour les programmes P_7, P_9 et P_{10} est plus élevé que les autres méthodes, néanmoins les efforts sont presque équivalents pour les six méthodes. L'effort pour ces quatre programmes est inférieur à 10% .

L'utilisation des techniques basées sur le calcul du score de suspicion, pour les traces des programmes P_3 et P_4, sauf pour le cœfficient O^p, nécessite un effort d'analyse supérieur à 75%. Le coût élevé de l'effort d'analyse est relatif au nombre d'apparitions de l'instruction contenant la faute dans la trace. C'est-à-dire qu'en raison du grand nombre d'apparitions de l'instruction fautive dans les cycles considérés comme correctes, celle-ci a été considérée comme moins suspecte par rapport aux autres instructions du cycle fautif. Ainsi, selon le classement proposé par ces méthodes, il est inévitable de consulter un certain nombre d'instructions considérées comme suspectes avant d'arriver à l'instruction contenant la faute. L'effort fourni pour ces mêmes traces, en utilisant la méthode du plus proche voisin, est inférieur à 5%.

Pour les programmes P_2 et P_{12}, le cœfficient O^p fournit un classement non-pertinent des suspects, où presque toutes les instructions obtiennent des scores

de méfiance similaires. Par conséquent, afin de localiser la faute, la trace et le code source sont inspectés entièrement (100%) par l'ingénieur. L'explication de ce mauvais diagnostic est que les programmes sont trop différents du modèle « ITE2 » pour lequel le cœfficient O^p est optimal.

Pendant la localisation de faute pour le programme P_{13}, ce sont la méthode du plus proche voisin et Ample qui nécessitent une analyse détaillée de la trace, avec des efforts supérieurs à 80%. Ce résultat, ainsi que les résultats d'analyse des programmes P_5 et P_{12} indiquent un manque de précision du calcul de suspicion en utilisant les cœfficient Ample et O^p, comparé aux trois autres techniques.

D'après ces résultats, nous observons que la méthode du voisin le plus proche et le classement basé sur l'utilisation du cœfficient O^p nécessitent une implication manuelle plus importante afin de localiser la faute comparé aux autres méthodes de classements des suspects. En particulier, pour les programmes P_1, P_2, P_5, P_6, P_8 et P_{11}, où les efforts pour la localisation de faute en utilisant le cœfficient O^p et la méthode du voisin le plus proche s'élève à plus de 35%, tandis que les efforts en utilisant Tarantula, Jaccard, Ample et Ochiai restent sous la barre de 5%. Il est important de constater que l'utilisation du cœfficient Ochiai fournit dans plus de 86% des cas un meilleur classement des instructions suspectes. Aussi, dans 75% des cas, pour la localisation de faute, moins de 20% du code source est analysée. Cependant, même dans le pire des cas, le code source et la trace ne sont pas analysés à 100%.

En conclusion, dans l'outil Comet, nous offrons par défaut un classement basé sur l'utilisation du cœfficient Ochiai.

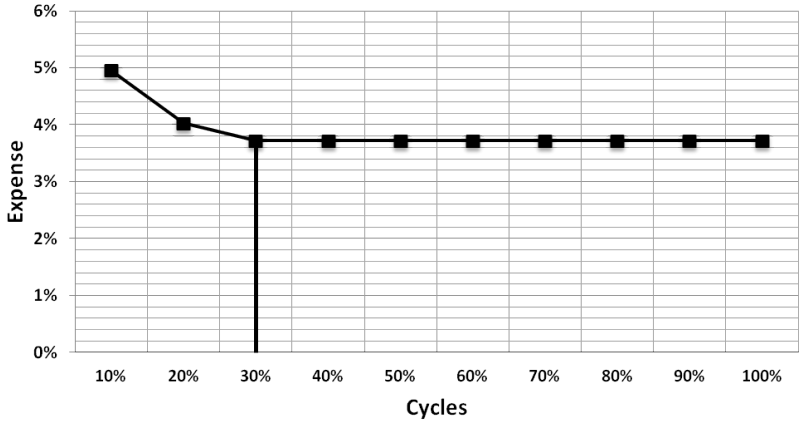

FIGURE 10.2 – Localisation de faute en utilisant le filtrage de cycles et Ochiai pour P_6

10.4 Évaluation du filtrage

Dans le but de rendre le diagnostic plus pertinent et plus rapide, nous avons proposé (section 9.4.1) une approche de filtrage. Cette approche a pour objectif d'atténuer le phénomène lié à la disparité des cycles, en ne considérant que les cycles les plus ressemblants au cycle erroné. Ce qui permet aussi d'accélérer les calculs, parce qu'il y a moins de cycles à comparer.

Cette section présente l'évaluation de cette approche. Pour les 13 traces d'exécution présentés précédemment, nous avons varié le nombre de cycles utilisés pour la localisation de faute de 10% à 100%. À chaque fois, l'effort a été calculé afin de mesurer l'influence du filtrage sur l'effort fourni. Étant donné les bons résultats du cœfficient Ochiai concernant la localisation de faute, nous l'avons utilisé pour l'évaluation de notre approche de filtrage.

Pour 12 programmes sur 13, l'effort d'analyse se stabilise dès l'utilisation de 10% des cycles de la trace, c'est-à-dire que l'effort fourni reste le même dès l'utilisation de 10% des cycles jusqu'à l'utilisation de 100% des cycles de la trace. Ainsi, le filtrage permet d'avoir un diagnostic tout aussi pertinent en utilisant selement 10% des cycles de la trace, ce qui permet d'accélérer les calculs et donc le diagnostic.

La Figure 10.2 illustre le résultat de l'évaluation du filtrage pour le programme P_6. Le graphique représente l'effort fourni pour localiser la faute dans le programme en utilisant le cœfficient Ochiai. Quand notre approche utilise 10% des cycles de la trace, l'ingénieur a besoin d'analyser 5% du code source pour localiser la faute. La trace d'exécution générée par le programme P_6 contient 323 cycles différents. Ainsi, en utilisant les 32 cycles les plus ressemblant au cycle fautif, l'ingénieur analyse 5% du code source avant de localiser la faute. Toutefois, en utilisant de 30% à 100% des cycles, l'effort pour la localisation de faute reste stable (moins de 4% du code source à analyser).

D'après les résultats de cette évaluation, nous observons qu'il est donc possible d'avoir un diagnostic pertinent en utilisant seulement 30% des cycles d'une trace, ce qui pemert d'acclérer la localisation de faute. Notre outil CoMET utilise par défaut 30% des cycles d'une trace.

Chapitre 11

Conclusion

La grande implication manuelle des ingénieurs dans le domaine de localisation de faute rend cette analyse très coûteuse en ressources. Par conséquent, l'automatisation de cette tâche est importante, ce qui peut considérablement accroître son efficacité et réduire ses coûts.

Les méthodes dynamiques de localisation de faute(s) permettent de répondre à ce besoin d'automatisation. Ces méthodes s'intéressent au suivi et au résultat de l'exécution du programme. Par conséquent, elles manipulent l'information issue de l'exécution des programmes. Afin de permettre la manipulation des informations issues de l'exécution d'un programme, ces informations sont stockées dans une *trace d'exécution*. Selon l'approche utilisée, certaines informations issues de l'exécution sont plus pertinentes que d'autres. Un spectre d'exécution est une abstraction de la trace d'exécution, contenant le verdict de l'exécution : succès ou échec. Si le résultat de l'exécution d'un programme est celui attendu alors l'exécution est en succès, sinon on dit que l'exécution est un échec. Comme les traces d'exécution, les spectres d'exécution sont collectés au cours de l'exécution du programme.

Les récents microcontrôleurs incluent des composants dédiés à la collecte de traces. Ainsi, en utilisant des sondes spécialisées, telles que Keil-UlinkPro [2] et ST-Link [6], il est possible de collecter des traces d'exécution sans données. Dans notre contexte industriel, le plus souvent, l'enregistrement de la trace est arrêté quand un événement empêchant la poursuite de l'exécution apparaît (par exemple, une exception est levée). Ainsi, ce mécanisme assure la présence d'un comportement anormal au moins dans les derniers événements enregistrés.

L'utilisation des méthodes dynamiques manipulant les spectres d'exécution pour la localisation de faute devient alors possible dans le contexte des microcontrôleurs. Ces méthodes sont appelées « méthodes par différence de traces (spectres) ». L'idée principale de ces méthodes est de comparer les comportements des exécutions en échec et les comportements des exécutions en succès, afin de renvoyer les différences entre ces comportements comme information pour le débogage. Dans le but de mieux cibler la faute, des méthodes statistiques sont utilisées pour calculer un ordre total sur des prédicats, ou sur les lignes d'un programme à inspecter pour trouver la faute. Ainsi, basées

sur un ensemble d'observations (exécutions), les approches automatiques de localisation de faute fournissent au développeur chargé du débogage une liste d'emplacements susceptibles de contenir la faute.

Il est important de préciser que dans notre contexte, une seule trace d'exécution est disponible. Cette situation rend l'application des méthodes dynamiques impossible telles qu'elles sont définies. Notre contribution consiste donc en une proposition d'adaptation. Cette adaptation s'appuie sur la nature cyclique des programmes embarqués et donc des traces d'exécution.

Le travail présenté dans cette partie nous a valu la publication de l'article [16] dans *ISSRE 2013*[1].

Notre approche de localisation de faute, utilisant une trace d'exécution de microcontrôleur, se divise en deux étapes. La première étape est la *détection de cycles*. Cette étape repose sur la détection et la localisation de la tête de la boucle principale (loop-header) du programme, dans la trace. Le résultat de cette première étape est une trace divisée en blocs, où chaque bloc représente un cycle spécifique (une exécution de la boucle principale). La seconde étape consiste à utiliser les « méthodes par différence de traces (spectres) », notées SBFL, sur la trace découpée pour localiser la faute. Les SBFL nécessitent la présence d'un oracle pour déterminer le bon déroulement des exécutions. Dans cette partie de la thèse, chaque cycle définit une exécution de la boucle principale du programme. Aussi le mécanisme d'enregistrement de la trace assure la présence d'un comportement anormal dans le dernier cycle. Ainsi, le dernier cycle se voit affecter le verdict d'une exécution en échec, alors que les autres cycles de la trace sont considérés comme des exécutions en succès.

Une évaluation expérimentale concernant six méthodes de localisation de faute a été menée. Ces six méthodes sont : le voisin le plus proche , Tarantula, Ample, Jaccard, Ochiai et l'utilisation du cœfficient O^p. Chaque méthode a été donc appliquée sur 13 traces de différents programmes embarqués. Pour chaque trace, un diagnostic est généré pour chaque application de l'une de ces six méthodes. Ainsi, les six diagnostics générés pour chaque trace sont comparés entre eux pour évaluer les performances de leurs méthodes respectives à localiser les fautes. La métrique utilisée pour évaluer les six méthodes est appelée *effort*. L'*effort* se définit comme étant le nombre d'instructions analysées pour localiser la faute [9, 41, 75] par rapport au nombre total d'instructions dans le code source.

D'après les résultats de l'expérimentation, dans 85% des cas, la localisation de faute en utilisant les techniques basées sur les scores de suspicion nécessitent moins d'effort que la méthode du « plus proche voisin ». Le décalage d'effort entre les méthodes utilisant des scores de suspicion et la méthode du

1. http ://2013.issre.net/

« plus proche voisin » se justifie par le manque du classement des instructions suspectes dans cette dernière. Les résultats de l'expérimentation montrent également l'efficacité du cœfficient Ochiai, qui fournit dans plus de 86% des cas un meilleur classement des instructions suspectes. Aussi, dans 75% des cas, pour la localisation de faute(s), moins de 20% du code source est analysé. Même dans le pire des cas, le code n'est jamais analysé à 100%.

Cependant, l'utilisation des techniques basées sur le calcul du score de suspicion est peu efficace dans le cas où l'instruction fautive apparait dans de nombreux cycles considérés comme correctes. Cela se traduit donc par un coût élevé d'effort d'analyse, qui est relatif au nombre d'apparitions de l'instruction contenant la faute dans la trace. C'est-à-dire qu'en raison du grand nombre d'apparitions de l'instruction fautive dans les cycles considérés comme corrects, celle-ci est considérée comme moins suspecte par rapport aux autres instructions du cycle fautif. Ainsi, selon le classement proposé par les différentes méthodes, il est inévitable de consulter un certain nombre d'instructions considérées comme suspectes avant d'arriver à l'instruction contenant la faute.

Afin de répondre à cette problématique, dans la partie suivante, nous proposons une autre approche de localisation de faute basée sur l'utilisation des règles d'association.

-Partie 3-

Fouille de Données pour la Localisation de faute(s)

Résumé

Notre approche de localisation de faute, présentée dans le chapitre 9, est basée sur l'hypothèse que les cycles d'une trace d'exécution sont indépendants. Par conséquent, elle risque d'être peu efficace dans le cas où les cycles d'une même exécution interagissent entre eux.

Pour faire de la localisation de faute pour ce type de cas, dans cette partie de la thèse, nous nous intéressons à la *fouille de données* et spécialement à la *recherche de règles d'association*, présentées dans le chapitre 13. La principale motivation de l'utilisation de la fouille de données dans notre approche est l'extraction d'informations de comportements considérés comme bons et des comportements considérés comme fautifs, afin de les comparer et ainsi localiser la faute. Ainsi basée sur l'utilisation des cycles d'une trace d'exécution, notre approche est présentée dans le chapitre 14.

Le chapitre 15 concerne l'évaluation de cette contribution et le chapitre 16 présente notre outil *CoMET*.

Chapitre 12

Introduction

Au cours du développement d'une application, des fautes peuvent être introduites dans le code. L'exécution des instructions fautives peut entrainer des erreurs et des défaillances pourront (ou non) être observées. Une fois une défaillance identifiée, il est nécessaire de la corriger : c'est l'étape de débogage. Cela consiste en premier lieu à rechercher la cause de la défaillance, c'est-à-dire l'instruction erronée. On parle alors de localisation de faute.

La localisation de faute(s) peut être définie comme étant le processus de repérage de faute(s) conduisant à des défaillances. Le repérage précis de fautes étant très difficile, la plupart des méthodes de localisation de faute(s) essayent plutôt d'identifier les parties du code source susceptibles de contenir la faute.

Dans la deuxième partie de cette thèse, nous avons présenté une approche de localisation de faute utilisant une seule trace d'exécution. Notre approche effectue en premier une *détection de cycles*. Le résultat obtenu est une trace divisée en cycles. Notre approche utilise ensuite les « méthodes par différence de traces » sur la trace découpée pour localiser l'instruction fautive. Cette approche est basée sur l'analogie entre les exécutions du programme et les cycles. Nous sommes conscients que l'analogie n'est pas parfaite et que l'indépendance entre les cycles ne peut pas être garantie. Cependant, les résultats de nos expérimentations (chapitre 10), ont montré la pertinence de cette hypothèse.

Par ailleurs, il existe également des cas où les multiples cycles d'une même exécution interagissent entre eux de plusieurs façons. Notre approche de localisation de faute risque donc d'être peu efficace pour ce type de cas.

Ainsi, dans le but de faire de la localisation de faute(s) pour des programmes où les cycles d'une même exécution interagissent entre eux, nous nous sommes intéressé à la fouille de données.

La fouille de données tire son nom du terme anglais *data mining*. C'est une méthode qui a pour objectif de trouver la pépite, non pas d'or, mais de connaissances. Elle est définie comme étant « the non trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data » [33]. La fouille de données est ainsi constitué d'un ensemble d'outils informatique servant à rechercher et à extraire de l'information (utile et inconnue) de gros volumes de données stockées dans des bases ou des entrepôts de données. La fouille de données s'effectue en respectant un ensemble de

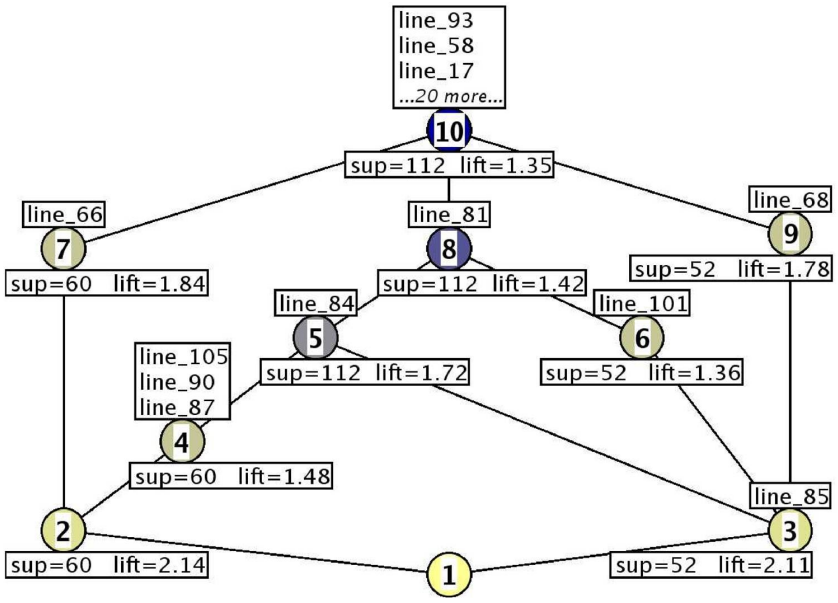

FIGURE 12.1 – Treillis des défaillance

processus d'extraction de l'information, en passant par les phases de stockage de données, d'interprétation des résultats, aussi de filtrage et nettoyage. Selon le problème traité, de nombreuses techniques de fouille de données existent, comme le regroupement, la classification ou la recherche de règles d'association.

Dans cette partie de la thèse nous nous intéressons à la recherche de règles d'association dans le but de localiser la faute dans les programmes de microcontrôleurs. Ce travail est inspiré des travaux de Cellier et al. [21] concernant la localisation de faute(s) en utilisant la fouille de données. La localisation de faute(s) proposée par Cellier et al. [21] est basée sur la recherche de règles d'association [13] mais également sur l'analyse formelle de concepts [64, 20], qui est une méthode de *classification non supervisée (clustering)*. Comme illustré dans la Figure 12.1, le résultat proposé est un treillis des défaillances que l'ingénieur doit parcourir et explorer pour localiser la faute. Chaque concept dans le treillis représente une ou plusieurs lignes du code source, avec les valeurs du *lift* et *support* correspondant au concept. L'ingénieur commence par analyser le treillis de bas en haut, c'est-à-dire en commençant par analyser le concept défaillant, puis analyser les concepts d'au-dessus et ainsi de suite jusqu'à la localisation de la faute.

Il est important de préciser que la localisation de faute, dans cette partie, concerne des programmes qui continuent leurs exécutions un certain moment après l'apparition de la défaillance. C'est précisément une situation où une méthode de localisation de faute basée sur l'analyse statistique des cycles de la

trace, peut fournir un mauvais diagnostic. Ainsi, la principale motivation pour utiliser la fouille de données est d'extraire des informations de comportements considérés comme bons et des comportements considérés comme fautifs, afin de les comparer et localiser la faute. La technique de fouille de données utilisée par notre approche est la recherche de règles d'association, discutée dans le chapitre 13. Dans le chapitre 14 nous présentons notre approche de localisation de faute basée sur l'utilisation des règles d'association.

Chapitre 13

Règles d'association

Sommaire

13.1 Introduction

Dans cette partie de la thèse nous nous intéressons à la localisation de faute en recherchant des règles d'association. La recherche de règles d'association est une méthode de fouille de données introduite par Agrawal et al. [13]. Agrawal et al. ont été amenés à travailler sur une application de produits dans les supermarchés [14]. Dans le domaine de la vente, la recherche de règles d'association est appelée « Analyse du panier de la ménagère », où une règle d'association sert à définir si l'achat d'un article a tendance à *entraîner* l'achat d'un autre article. Il s'agit donc d'obtenir des relations ou des corrélations du type « Si Condition alors Résultat ».

Le plus célèbre exemple de recherche de règles d'association est l'histoire des « beers and diapers ». Cette exemple concerne l'analyse des informations issues des cartes de fidélité d'une grande enseigne de supermarchés aux États-Unis. Le résultat de cette étude était assez étonnant car il permettait de constater que : le vendredi après-midi, les jeunes hommes américains qui achètent des couches culottes achètent *souvent* aussi de la bière. L'explication de ce constat est que les jeunes pères profitent de l'achat des couches pour leur bébé pour acheter de la bière pour leur soirée devant le programme de sport. Cet exemple permet d'avoir une idée de l'efficacité des règles d'associations à trouver des associations auxquelles on ne s'attend pas.

13.2 Définition d'une règle d'association

La recherche de règles d'association se fait sur un ensemble de *transactions*. Chaque transaction concerne un ensemble d'*items* appelé *itemset*.

Transaction	*Couches*	*Bière*	*Lait*	*Pain*	*Oeufs*	*Soda*
$Ticket_1$	X	X				
$Ticket_2$	X	X		X	X	
$Ticket_3$		X	X	X		X
$Ticket_4$	X	X	X	X		
$Ticket_5$	X	X	X			X

TABLE 13.1 – Construction de vecteurs binaires

Une règle d'association a la forme $P \rightarrow C$, où P et C sont des items. L'item P est appelé *prémisse* de la règle et l'item C est sa *conclusion*.

Il est important de préciser qu'une règle d'association est différente d'une implication logique. En effet, une règle d'association peut souffrir d'exceptions, mais pas une implication logique. Ainsi, une implication $A \Rightarrow B$ garanti que la réalisation de A entrainera la réalisation de B, alors que la règle d'association $A \rightarrow B$ signifie que la réalisation de A peut, à forte probabilité, entrainer la réalisation de B, ou non.

La Table 13.1 décrit des ventes d'un supermarché. Les transactions dans cet exemple sont les tickets (du $Ticket_1$ au $Ticket_5$). Les items sont les produits achetés (Bière, Couches, Lait, Pain, Oeufs, Soda). L'information « $Couche \rightarrow Bière$ » est une règle d'association qui signifie que l'achat des Couches entraîne souvent l'achat de Bière.

Un point important à prendre en considération lors de la recherche de règles, concerne la pertinence des règles associations, où dans certains cas quelques associations apparaissent simplement par hasard. Ainsi, il est nécessaire d'avoir des mesures qui permettent d'évaluer la pertinence d'une règle d'association, comme nous le détaillons dans la section suivante.

13.3 Évaluation des règles d'association

Il est possible d'avoir un assez grand nombre de règles d'association à partir d'un ensemble de transactions. Cependant, certaines sont plus pertinentes que d'autres. Afin d'évaluer la pertinence d'une règle d'association de nombreux indices statistiques peuvent êtres utilisés [18, 25]. Dans cette thèse nous présentons les trois indices les plus souvent utilisés [25], qui sont : le *support*, la *confiance* et le *lift*.

13.3.1 Support

Soient I l'ensemble des items et $T = \{t_1, \ldots, t_i\}$ l'ensemble des transactions. Chaque transaction contient un sous ensemble d'items, appelé *itemset*. Soient P et C deux itemsets.

Le *support* de la règle d'association $P \to C$, noté $sup(P \to C)$, est la proportion de transactions contenant les items de P et de C. Mathématiquement, il est défini par :

$$sup(P \to C) = \frac{\left|\{t_i \in T \mid P \cup C \subseteq t_i\}\right|}{\left|T\right|}. \tag{13.1}$$

Le support est donc calculé en divisant le nombre de transactions contenant les items de P mais également les items de C sur le nombre de toutes les transactions $\left|T\right|$.

Exemple : Dans les données de la Table 13.1, le support de la règle d'association « $Couche, Bière \to Lait$ » est égal à $\frac{2}{5}$, car deux tickets, $Ticket_2$ et $Ticket_4$, contiennent à la fois $Couche$, $Bière$ et $Lait$.

Dans le but d'éviter la génération d'un grand nombre de règles peu fréquentes, souvent, lors de la recherche de règles d'association, un seuil de support minimal est fixé.

13.3.2 Confiance

La *confiance* d'une règle d'association est le rapport entre le nombre de transactions vérifiant cette règle et le nombre de transactions vérifiant sa prémisse. Mathématiquement, la confiance $conf(P \to C)$ d'une règle d'association $P \to C$ est définie par :

$$conf(P \to C) = \frac{\left|\{t_i \in T \mid P \cup C \subseteq t_i\}\right|}{\left|\{t_i \in T \mid P \subseteq t_i\}\right|}. \tag{13.2}$$

La confiance est donc calculée en divisant le nombre de transactions contenant les items de P mais également les items de C sur le nombre de toutes les transactions contenant les items de P.

Exemple : dans les données de la Table 13.1, la confiance de la règle « $Lait \to Pain$ » a une confiance de $\frac{2}{3}$, car deux tickets, $Ticket_3$ et $Ticket_4$ contiennent $Pain$ et $Lait$, alors que trois tickets, $Ticket_3$, $Ticket_4$ et $Ticket_5$, contiennent $Lait$.

Dans le but d'éviter la génération d'un grand nombre de règles souffrant de « trop d'exceptions », souvent, lors de la recherche de règles d'association, un seuil de confiance minimal est fixé.

13.3.3 Lift

Le *lift* d'une règle d'association est défini comme le rapport de la confiance de cette règle et la confiance d'une règle ayant la même conclusion quelle que soit sa prémisse. Par exemple, une règle $P \rightarrow C$ ayant un lift égal à deux indique que, comparés aux autre individus, les individus ayant l'item P ont deux fois plus de chances d'avoir l'item C. Ainsi, le lift sert à mesurer l'apport réel d'une prémisse. Il est défini mathématiquement comme suit :

$$lift(P \rightarrow C) = \frac{conf(P \rightarrow C)}{sup(C)}. \tag{13.3}$$

Exemple : Dans les données de la Table 13.1, le lift de la règle d'association « $Couches \rightarrow Pain$ » est égal à 1. En effet, la confiance de la règle est égale à $\frac{1}{2}$. De plus, trois tickets sur six contiennent $Pain$ dans leur description : $Ticket_2$, $Ticket_3$ et $Ticket_4$. Donc $sup(Pain) = \frac{1}{2}$.

Le lift mesure donc l'influence de l'apparition de la prémisse dans une transaction sur l'apparition de la conclusion dans cette même transaction. Il y a trois cas à considérer :

- $lift(P \rightarrow C) > 1$: signifie que l'observation de la prémisse augmente la probabilité d'observer la conclusion. Cet effet est appelé attraction, P attire C.
- $lift(P \rightarrow C) = 1$: signifie que l'observation de la prémisse et de la conclusion sont deux faits indépendants.
- $lift(P \rightarrow C) < 1$: signifie que l'observation de la prémisse diminue la probabilité d'observer la conclusion. Ces deux items sont dits alors négativement corrélés. Cet effet est appelé répulsion, P repousse C.

13.4 L'algorithme LCM

La recherche d'items fréquents est l'un des problèmes fondamentaux dans l'exploration de données et possède de nombreuses applications telles que la règle d'association minière [14], les bases de données inductives [56], et l'expansion de requête [63].

Soit $I = \{1, \ldots, n\}$ l'ensemble des items. Un sous-ensemble X de I est appelé itemset. Soit T l'ensemble des transactions sur I, c'est-à-dire, chaque $t \in T$ est composé d'items de I. Pour un itemset X, soit $T(X) = \{t \in T \mid X \subseteq t\}$ l'ensemble des transactions concernant X. Chaque transaction de $T(X)$ est appelée *occurrence* de X. Pour une constante donnée $a \geq 0$, un itemset X est dit fréquent si $|T(X)| \geq a$. Si un itemset fréquent n'est inclus dans aucun autre

itemset fréquent, il est dit *maximal*. Pour un ensemble de transactions $S \subseteq T$, soit $L(S) = \bigcap_{T \in S} T$. Si un itemset X satisfait la condition $L(T(X)) = X$, alors X est appelé itemset fermé.

LCM [72], pour « Linear time Closed itemset Miner », est un algorithme qui sert à énumérer les itemsets fermés fréquents. Tel qu'illustré dans l'exemple de la Figure 13.1 [34], cet algorithme est basé sur la définition de la relation parent-enfant entre les itemsets fermés fréquents. Cette relation produit un arbre composé d'un ensemble d'itemsets fermés fréquents. L'algorithme parcoure ainsi l'arbre en un temps linéaire selon le nombre d'itemsets fermés fréquents. Cet algorithme est inspiré des algorithmes utilisés pour l'énumération des cliques bipartites maximales [71, 70].

Dans notre travail, LCM est utilisé par notre approche afin d'extraire les règles d'association d'un ensemble de cycles. L'algorithme est donné dans les annexes, page 156.

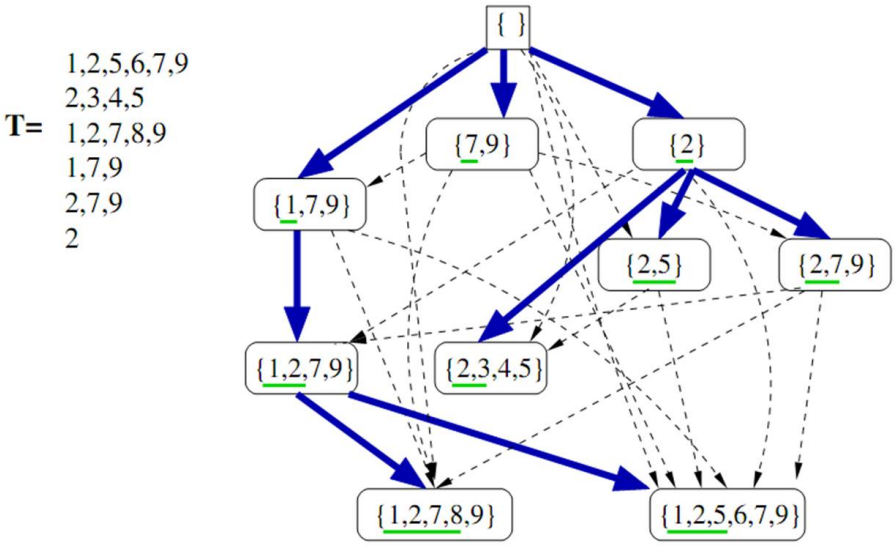

FIGURE 13.1 – Exemple d'utilisation de l'algorithme LCM pour découvrir les itemsets fréquents

Chapitre 14

Recherche de règles d'association en utilisant une seule trace d'exécution

Sommaire

14.1 Introduction

Dans la seconde partie de cette thèse, nous avons présenté une approche de localisation de faute en considérant que les cycles sont indépendants.

Afin de lever cette hypothèse, dans cette partie, nous proposons une approche de localisation de faute basée sur la recherche de règles d'association dans une trace d'exécution. L'idée intuitive derrière notre approche est l'utilisation des cycles de la trace pour générer des règles d'association *fiables* et *peu fiables*, afin de les comparer et localiser la faute.

L'arrêt de l'enregistrement de la trace est lié directement à l'apparition d'un comportement anormal du microcontrôleur, par conséquent le dernier cycle est considéré comme fautif. Ainsi, pour rechercher les règles d'association, notre approche groupe les cycles de la trace en deux ensembles : un ensemble de cycles « *loin* du cycle fautif » et un ensemble de cycles « *proches* du cycle fautif ». Ensuite, l'application de l'algorithme LCM sur ces deux ensembles de transactions permettra de définir les comportements jugés corrects et ceux jugés comme suspects. Ainsi, pour la localisation de faute, le diagnostic concernera plus les comportements suspects.

14.2 Groupement des cycles

Un prétraitement important consistant à diviser la trace d'exécution en cycles est nécessaire avant l'application de la localisation de faute basée sur la recherche des règles d'association. Cette division de la trace est effectuée par le processus de localisation du loop-header, présenté dans le chapitre 4 de cette thèse.

Générer des règles d'association nécessite une quantité minimum d'échantillons, à partir duquel les règles d'association seront déduites. Ainsi, tel qu'illustré dans la Figure 14.1, avant de procéder à la localisation de faute basée sur l'utilisation des règles d'association, en comparant les comportements correctes et suspects, il est important de séparer les cycles en deux groupes distincts :

- l'ensemble des cycles qui ressemblent le *plus* au cycle fautif,
- l'ensemble des cycles qui ressemblent le *moins* au cycle fautif.

Pour chaque cycle de la trace, son degré de ressemblance avec le cycle fautif est mesuré en utilisant la distance de Hamming [36], que nous avons présenté section 9.3.1.

Afin d'éviter un groupement non pertinent, deux bornes peuvent être définies pour la distance de Hamming :

- **Une borne de ressemblance** : pour qu'un cycle soit considéré comme ressemblant au cycle fautif, il est nécessaire que son degré de ressemblance soit supérieur à 70%.
- **Une borne de non-ressemblance** : pour qu'un cycle soit considéré comme différent du cycle fautif, il est nécessaire que son degré de ressemblance soit inférieur à 30%.

Les valeurs attribuées pour ces bornes (70% et 30%), dans cette thèse, sont définies grâce à de nombreuses expérimentations effectuées dans le but d'obtenir un bon groupement des cycles et ainsi des règles d'association plus pertinentes.

L'objectif de notre approche est de comparer les règles d'association générées à partir de deux groupes de cycles, les cycles les plus ressemblant au cycle fautif, et ceux qui lui ressemblent le moins. L'utilisation de ces bornes permet donc d'éviter d'avoir un groupement trop général, spécialement concernant les cycles dont les degrés de ressemblance avec le cycle fautif varient autour de 50%. Comme il est difficile de déterminer si ces cycles sont plus ou moins resemblant au cycle fautif, leurs utilisation risque d'engendrer des règles d'association souffrant d'exceptions. Ces bornes permettent donc d'éviter la génération d'un grand nombre de règles souffrant de « trop d'exceptions ». De

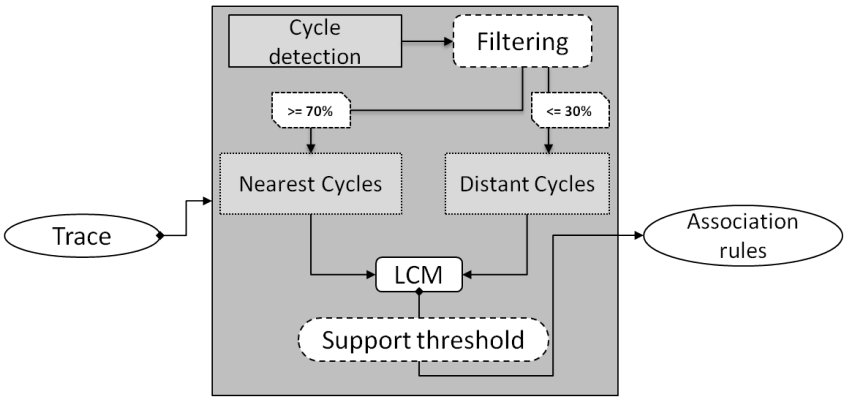

FIGURE 14.1 – Processus de génération des règles d'association

plus, afin de limiter la génération des règles peu fréquentes, un seuil de support minimal est fixé. D'après nos évaluations, un support minimal de 30% permet de réduire considérablement la génération de règles d'association non pertinentes pour la localisation de faute. Ainsi, deux filtres sont appliqués pour la génération de règles d'association pertinentes. Le premier filtrage est appliqué sur le groupement des cycles de la trace. Le second concerne le support et est fourni comme paramètre à l'algorithme LCM.

Par la suite, afin de permettre une meilleure localisation de faute(s), notre approche attribue à chaque règle d'association générée un degré de fiabilité, tel qu'expliqué dans la section suivante.

14.3 Fiabilité des règles d'association

Pour chacun des deux ensembles de cycles générés par l'étape de groupement, un certain nombre de règles d'association est produit. Avant de pouvoir utiliser ces règles d'association pour la localisation de faute(s), il est important de mesurer le degré de pertinence de leurs informations. Ainsi, un degré de fiabilité est associé à chaque règle d'association selon les critère suivants :

- **Fiable** : une règle d'association est dite fiable si elle est générée à partir de l'ensemble des cycles qui ressemblent le moins au cycle fautif.
- **Moyennement fiable** : une règle d'association est dite moyennement fiable si elle est générée à partir de l'ensemble des cycles qui ressemblent le plus au cycle fautif, mais il existe une règle d'association générée à partir de l'ensemble des cycles qui ressemblent le moins au cycle fautif avec la même prémisse (respectivement conclusion), mais une conclusion (respectivement prémisse) différente.

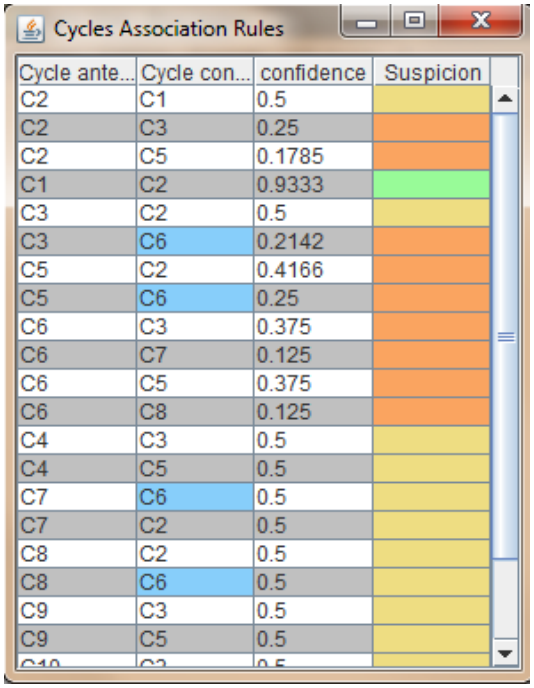

FIGURE 14.2 – Représentation graphique de la fiabilité des règles d'association des cycles

- **Peu fiable** : une règle d'association est dite peu fiable si elle est générée à partir de l'ensemble des cycles qui ressemblent le plus au cycle fautif, mais il n'existe aucune règle d'association générée à partir de l'ensemble des cycles qui ressemblent le moins au cycle fautif avec la même prémisse ou conclusion.

Si une règle d'association générée par l'ensemble des cycles qui ressemblent le plus au cycle fautif a déjà été générée par l'ensemble des cycles qui ressemblent le moins au cycle fautif, elle est ignorée, car elle ne représente pas une nouvelle source d'information. Ce mécanisme permet ainsi de réduire le nombre de règles d'association à analyser par l'ingénieur. Tel qu'illustré dans la Figure 14.2, afin de faciliter l'analyse, une couleur est associée à chaque type de règle :

- **Vert** : pour les règles fiables,
- **Jaune** : pour les règles moyennement fiables,
- **Orange** : pour les règles peu fiables.

Pour localiser la faute, l'ingénieur analyse les règles d'association générées en commençant par les peu fiables, parce qu'elles sont plus suspectées de contenir la faute. C'est-à-dire que les règles d'association coloriées en orange sont analysées en premier, après celles coloriées en jaune, puis les vertes à la

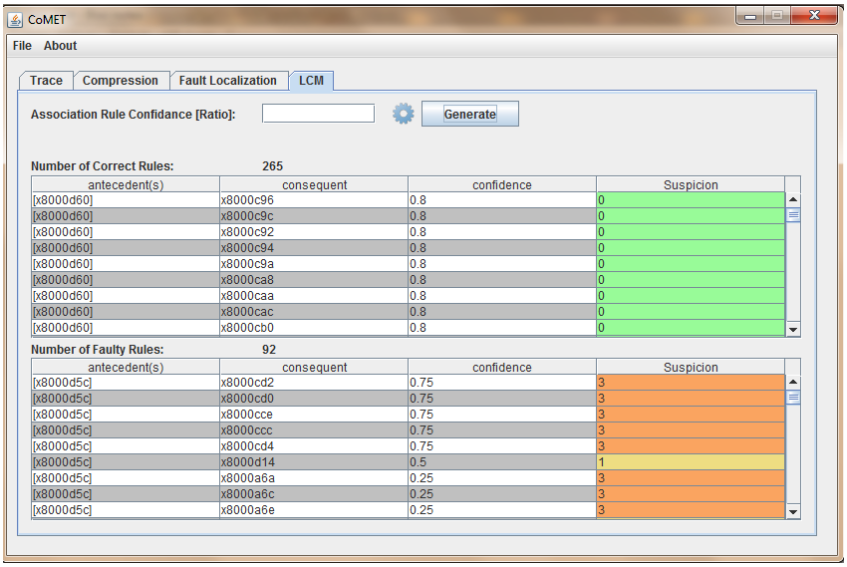

FIGURE 14.3 – Représentation graphique de la fiabilité des règles d'association des PCs

fin.

Notre approche génère deux types de règles d'association. Le premier type de règles d'association concerne les cycles, tel qu'illustré dans la Figure 14.2. Ces règles aident dans l'analyse du comportement du microcontrôleur en fournissant des informations sur les relations entre les cycles, où par exemple l'apparition du cycle $C1$ dans la trace d'exécution entraine, avec une confiance de $0,93$, l'apparition du cycle $C2$. Le deuxième type de règles illustré dans la Figure 14.3 concerne les instructions du programme embarqué et est utilisé pour la localisation de la faute, cela grâce au degré de fiabilité attribué à chaque règle analysée. Ainsi, pour localiser la faute, l'ingénieur analyse les règles d'association en commençant par les plus suspectes.

Une fois qu'une règle d'association analysée est validée par l'ingénieur, son degré de fiabilité passe de moyennement ou peu fiable à fiable. Par conséquent les degrés de fiabilité des règles d'association restantes sont réévalués, cela en vérifiant si la prémisse et la conclusion de la règle validée apparaissent dans les règles restantes à analyser. Ainsi, le degré de fiabilité de certaines règles d'association passe de faible à moyen. Ce mécanisme a pour objectif l'amélioration de la pertinence du diagnostic et ainsi la localisation de faute.

L'évaluation de cette approche de localisation de faute en utilisant les règles d'association est présentée dans le chapitre suivant.

Chapitre 15

Évaluation

Sommaire

Dans ce chapitre, nous présentons l'évaluation de l'approche de localisation de faute basée sur l'analyse de règles d'associations. Il est important de rappeler que l'approche à évaluer dans cette partie traite une seule trace, où la cause de la défaillance ne se trouve pas forcément dans le dernier cycle. L'évaluation discutée dans cette partie consiste essentiellement à appliquer notre approche sur des programmes contenant des fautes connues, dans le but de mesurer la qualité du diagnostic fourni par notre approche de localisation de faute. Cette évaluation utilise CoMET (chapitre 16).

15.1 Programmes et erreurs

Les traces utilisées pour évaluer notre approche proviennent des 13 programmes embarqués, utilisés dans l'évaluation présentée chapitre 10. Cependant, pour l'évaluation de notre approche de localisation de faute en utilisant les règles d'association, des modifications concernant les gestions des exceptions ont été apportées aux 13 programmes. L'objectif de ces modifications est de permettre aux programmes de continuer à s'exécuter après l'apparition d'une anomalie (matérielle ou logicielle). Néanmoins, si un comportement anormale est répété un certain nombre de fois, l'exécution est arrêtée. Afin de garder une certaine cohérence dans les traces d'exécution et éviter d'avoir des traces qui ne contiennent que des exécutions défaillantes, le nombre d'exécutions défaillantes maximum a été fixé à cinq. Dans le cas contraire l'exécution et l'enregistrement de la trace sont arrêtés.

Ces programmes sont fournis par nos partenaires industriels : STMicroelectronics [1] et EASii-IC [2]. Nous rappelons que chacun des 13 programmes permet à l'utilisateur d'interagir avec le microcontrôleur, ceci en utilisant soit

1. http ://www.st.com
2. http ://www.easii-ic.com/

l'écran LCD, soit les 4 boutons du microcontrôleur. Par conséquent, quand l'utilisateur appuie sur un bouton, le microcontrôleur exécute un traitement spécifique. Si le traitement se termine sans erreur, le voyant LED correspondant au bouton pressé est allumé et un message est affiché sur l'écran LCD. Dans ce qui suit le programme numéro i est noté P_i. Les informations concernant les 13 programmes ainsi que leurs traces sont fournis dans la section 10.1 et la Table 10.1.

Chacun des 13 programmes est téléchargé sur un microcontrôleur STM32-F107 EVAL-C (Figure 5) et exécuté. La trace produite par l'exécution est récupérée à l'aide d'une sonde Keil-UlinkPro [2].

15.2 Résultats

L'évaluation expérimentale de cette partie concerne la localisation de faute en utilisant les règles d'association. La génération des règles d'association est basée sur l'utilisation des cycles d'une trace.

Dans le but de rechercher les règles d'association, l'algorithme LCM est appliqué sur chaque trace d'exécution.

Programme	**# RA F**	**# RA MF**	**# RA PF**
P1	772	1085	332
P2	5	24	70
P3	234	102	54
P4	403	611	87
P5	461	696	151
P6	893	1159	39
P7	1415	1049	671
P8	323	527	103
P9	371	502	133
P10	231	12	44
P11	169	6	150
P12	7	5	1004
P13	406	66	251

TABLE 15.1 – Information concernant les règles d'association

Les informations présentées dans la Table 15.1 concernent les règles d'association générées en utilisant les traces d'exécution des différents programmes. La deuxième colonne **# RA F** représente le nombre de règles d'association générées, considérés comme fiables. La troisième colonne **# RA MF** représente le nombre de règles d'association générées, considérés comme moyennement

fiables. La dernière # **RA PF** colonne représente le nombre de règles d'association générées, considérés comme peu fiables. Ainsi, le nombre total des règles d'association, générées en utilisant LMC sur la trace d'exécution d'un programme, est obtenu en additionnant le nombre de règles des trois colonnes. Par exemple, le nombre de règles générées pour le programme $P1$ est égal à 2189 (772 + 1085 + 332).

Afin d'évaluer l'efficacité de notre approche, nous mesurons l'effort fourni par l'ingénieur pour localiser la faute. Étant donné que plusieurs règles d'association peuvent partager des instructions, nous nous intéressons au nombre de règles analysées plutôt qu'au nombre d'instruction analysées. L'*effort* fourni pour localiser une faute, noté E, dépend donc du nombre de règles analysées A et du nombre total de règles R générées par LCM. Il est défini comme suit :

$$E = \frac{A}{R}$$

L'ingénieur analyse les règles d'association considérées comme peu fiables en premier. Si une règle peu fiable ne permet pas d'identifier la faute, elle est validée. La conséquence de cette validation est le passage de cette règle de l'ensemble des règles peu fiables à l'ensemble des règles fiables. Souvent, ce passage entraine la réévaluation des règles d'association peu fiables, où le niveau de fiabilité de certaines d'entre elles devient moyen. Par conséquent, tout au long de l'analyse, le nombre de règles peu fiables diminue, alors que les nombres des règles fiables et moyennement fiables augmentent.

Quand l'ensemble des règles d'association peu fiables devient vide, l'ingénieur analyse alors l'ensemble des règles d'association moyennement fiables. À cette étape, où il n'y a plus de règles peu fiables, c'est le nombre de règles moyennement fiables qui diminue, alors que celui des règles fiables augmente. Nous souhaitons réduire l'effort d'analyse en évitant l'analyse des règles d'association considérées comme étant fiables. Par conséquent, si l'ensemble des règles d'association moyennement fiables devient vide à son tour, la localisation de faute est considérée comme un échec, car toutes les règles d'association générées sont validées comme étant fiables.

La Figure 15.1 représente les résultats de l'évaluation expérimentale, où l'axe des abscisses représente les programmes de P_1 à P_{13} et l'axe des ordonnées représente l'effort fourni en analysant les règles d'association pour la localisation de faute.

La localisation de faute en utilisant les traces d'exécution des programmes P_1, P_3, P_4, P_5, P_6 et P_7 nécessite un effort d'analyse inférieur à 20%. Par exemple, pour le programme P_3 il a fallu analyser 53 règles sur 390 afin d'identifier l'instruction contenant la faute, ce qui représente un effort de 13,6%.

Concernant les programmes P_9, P_{12} et P_{13} la localisation de faute a nécessité un effort d'analyse entre 20% et 30%.

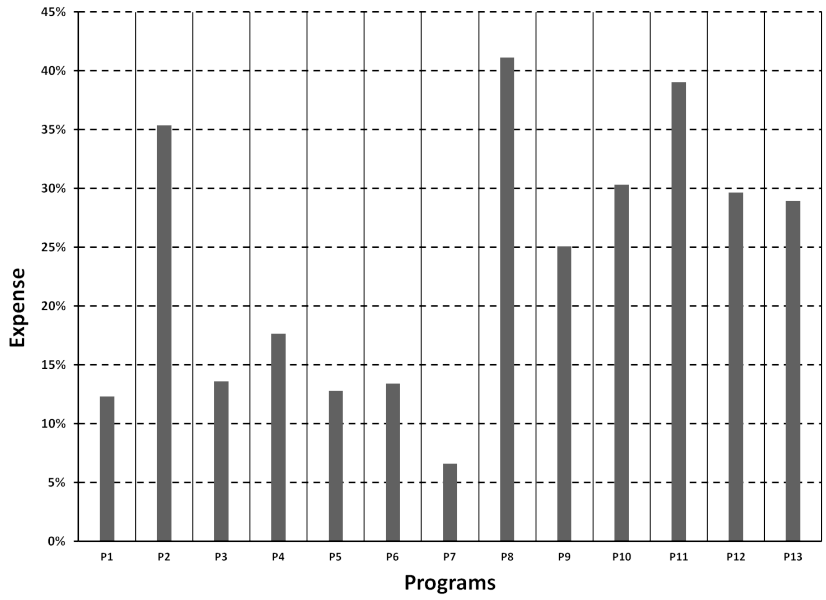

FIGURE 15.1 – Localisation de faute sur les 13 programmes en utilisant les règles d'association

La localisation de faute en utilisant les traces d'exécution des programmes P_2, P_8, P_{10} et P_{11} a nécessité un effort d'analyse supérieur à 30%. L'effort maximum concerne la localisation de faute pour le programme P_8 avec une valeur de 41,13%.

D'après ces résultats, nous observons que l'utilisation des règles d'association pour la localisation de faute dans des programmes où les multiples cycles d'une même exécution peuvent interagir entre eux, nécessite un effort d'analyse inférieur à 42%. Cependant, nous pensons qu'il est encore possible de réduire cet effort en appliquant des filtres de sélection sur les règles d'association à vérifier, comme il est expliqué dans les perspectives de notre travail.

Les résultats présentés dans cette partie dépendent principalement des programmes utilisées. Étant donne que le projet dans lequel s'inscrit notre travail (FUI IO32 [3]) est arrivé à sa fin, nous n'avons pas eu le temps de faire d'avantage d'expérimentations. Par conséquent, l'application de notre approche sur d'autres programmes avec des exécutions différentes peut nécessiter un plus (moins) grand effort d'analyse. Nous pensons qu'il faudrait changer les valeurs attribuées aux bornes de ressemblance dans le cas où il y a très peu de cycles qui ressemblent le plus au cycle fautif. Sinon, il est possible de n'avoir que des règles fiables. Cela ne représente pas forcement une limite de notre approche, mais nécessite l'analyse des règles considérées comme fiables, ce que

3. http ://io32.forge.imag.fr/

nous souhaitions éviter à l'ingénieur.

Chapitre 16

CoMET

Dans le but d'offrir une solution logicielle à nos partenaires industriels, nous avons développé l'outil *CoMET*. Afin de faciliter son utilisation, notre outil dispose d'une interface graphique, comme le montre la Figure 16.1. Une version récente de l'outil a été distribuée à nos partenaires, ce qui permettra d'évaluer l'utilisabilité de CoMET et l'efficacité des différentes approches présentées dans cette thèse. Un dossier de valorisation logiciel a été déposé auprès de l'Agence pour la Protection des Programmes. L'architecture de notre outil est décrite dans la section suivante.

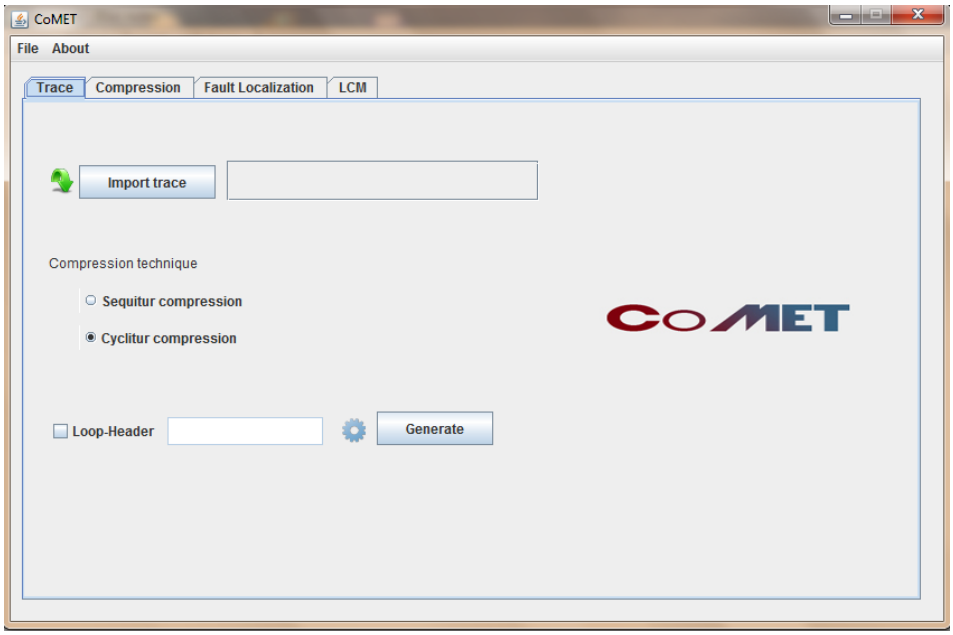

Figure 16.1 – Interface graphique de CoMET

16.1 Architecture

L'outil est écrit en Java et contient 11 packages, 100 classes et plus de 12,000 lignes de code. Nos approches sont implémentées dans CoMET, qui a servi pour nos évaluations.

Afin de permettre la manipulation de traces d'exécution de grandes tailles (plus de 3 giga-octets), nous avons choisi d'utiliser des bases de données. Par conséquent, CoMET permet la manipulation des bases de données en utilisant MySQL [5]. L'autre avantage concernant l'utilisation des bases de données par notre outil concerne le stockage des traces d'exécution d'un même programme, ainsi il est possible de les réutiliser dans le cas d'une obtention d'une nouvelle trace du même programme.

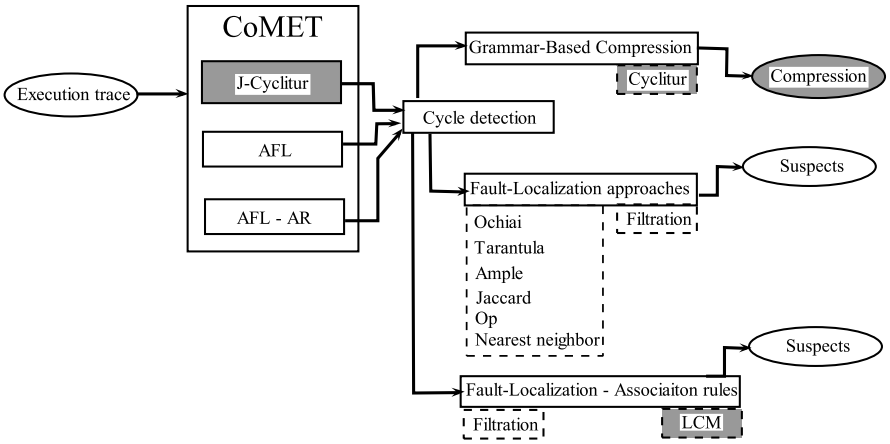

FIGURE 16.2 – Mécanisme de CoMET

CoMET prend en entrée un fichier de trace d'exécution et peut générer plusieurs sorties.

Ainsi, pour la compréhension et l'analyse de la trace, CoMET génère une grammaire comme sortie. La compression générée est soit sous forme de texte, soit sous forme d'objet Java pour une utilisation orientée programmation, comme la comparaison des cycles par exemple.

Aussi, selon le type de localisation de faute choisi, soit par analyse de spectres, soit par analyse de règles d'association, CoMET génère également comme sortie, soit une liste ordonnée d'instructions suspectes, soit un ensemble de règles d'association à analyser. Il est important de préciser que notre outil implémente toutes les approches présentées dans le chapitre 8. Pour la recherche de règles d'association, notre outil fait appel à l'algorithme LCM [72].

16.2 Fonctionnalités

Concernant la contribution d'aide à l'analyse et à la compréhension de traces, CoMET implémente les deux algorithmes de compression *Sequitur* et

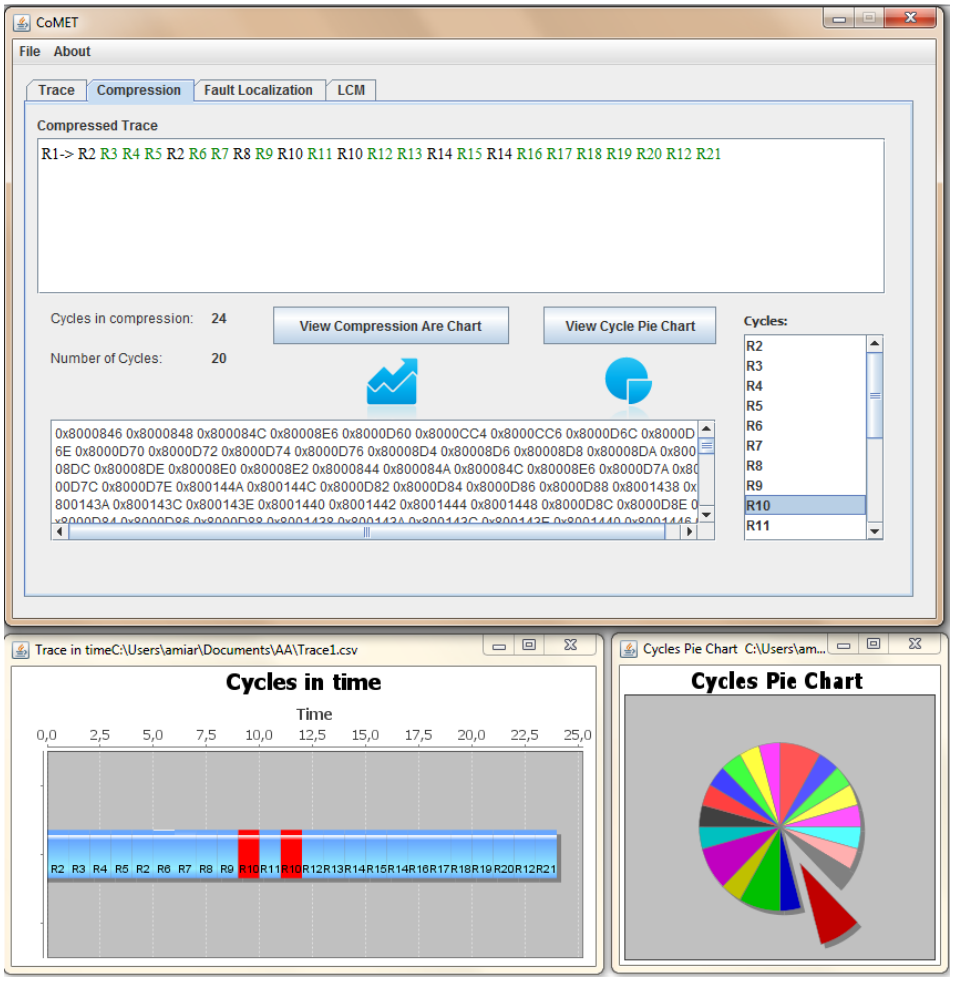

FIGURE 16.3 – Comet et ses visualisations

Cyclitur. Ainsi, l'utilisateur a la possibilité de choisir l'algorithme de compression à utiliser. Cyclitur est l'algorithme utilisé par défaut.

Notre outil supporte deux types de définition du loop-header. Une définition manuelle, qui concerne les situations où l'utilisateur a une totale maitrise du domaine, de la trace d'exécution, ainsi que du code source. La seconde définition du loop-header est basée sur une détection automatique utilisant notre approche, présentée dans le chapitre 4.

Dans le but de faciliter l'analyse de la compression générée par notre approche, comme le montre la Figure 16.3, *CoMET* fournit deux visualisations :

- *Cycle pie chart* : cette visualisation fournit la fréquence d'apparition de cycles dans la trace.
- *Cycle in time* : cette visualisation met l'accent sur toutes les occurrences

d'un cycle donné dans la trace.

Notre approche de localisation de faute est précédée par le processus de division de la trace en cycles, tel qu'illustré dans la Figure 16.2. Cette étape est effectuée automatiquement par CoMET en utilisant la bibliothèque de compression. En se basant sur les résultats de nos expérimentations (chapitre 10), notre technique de localisation de faute utilise par défaut le cœfficient Ochiai. Cependant, l'utilisateur a la possibilité de choisir une autre parmi les techniques présentées dans cette thèse (Figure 16.4). Ainsi, selon la technique de localisation de faute choisie par l'utilisateur, un diagnostic est généré comme sortie. Selon la technique choisie, deux types de diagnostic sont possibles. Pour la méthode du « plus proche voisin », le résultat obtenu se compose de deux ensembles : un ensemble d'événements manquants et un ensemble d'événements considérés comme potentiellement superflus. Pour les techniques basées sur l'utilisation des scores de suspicion, le résultat obtenu est un classement décroissant des événements considérés comme suspects. Le classement est basé sur les valeurs de suspicion. Les techniques basées sur l'utilisation des scores de suspicion implémentées sont : Tarantula, Jaccard, Ample et Ochiai. Pour ces techniques, l'utilisateur a la possibilité de choisir la taille de l'échantillon contenant les cycles utilisés pour la localisation de faute, comme le montre la Figure 16.4. Par exemple, si une trace d'exécution contient 13 cycles différents, l'utilisateur peut choisir pour la localisation de faute un ensemble de cycles allant d'un cycle jusqu'à douze cycles. Ainsi, si l'utilisateur choisi de n'utiliser que 50% des cycles de la trace, seulement les six cycles qui ressemblent le plus au dernier cycle seront utilisés pour la localisation de faute. En se basant sur les résultats de notre expérimentation présentée dans la section 10.4, la taille d'échantillon utilisée par défaut par notre outil est de 30%. CoMET implémente également notre approche présentée dans cette partie concernant la localisation de faute basée sur l'analyse des règles d'association. Comme pour la localisation de faute basée sur les statistiques, la localisation de faute basée sur l'analyse des règles d'association est précédée par le processus de division de la trace en cycles, effectuée automatiquement par CoMET en utilisant la bibliothèque de compression. Ensuite, vient l'étape de calcul des distances, en utilisant la distance de Hamming, ainsi que le regroupement des cycles selon les critère discutés dans la section 14.1. Un ensemble de règles d'association avec des degrés de fiabilité est généré comme sortie. Un classement ascendant des règles d'association selon leurs degrés de fiabilité est proposé à l'ingénieur. L'étape d'analyse est simplifiée grâce à une interface graphique et en proposant une coloration des règles d'association selon leurs degrés de suspicion, comme illustré dans les Figures 14.2 et 14.3. Comme nous l'avons dit précédemment, notre outil permet la génération de deux types de règles

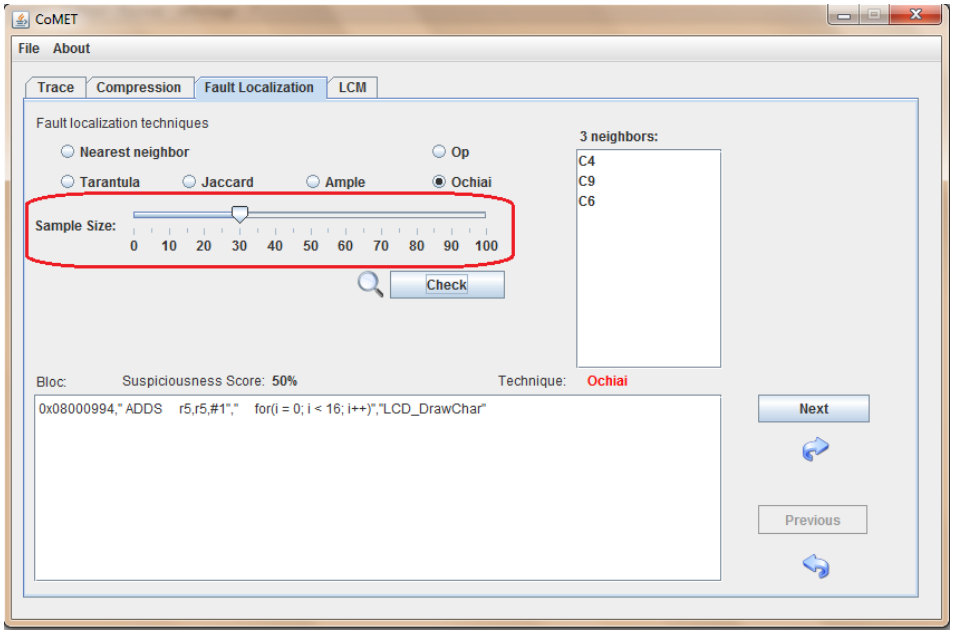

FIGURE 16.4 – Localisation de faute avec CoMET

d'association. Le premier type concerne les cycles et a pour objectif d'aider à l'analyse et à la compréhension de la trace. Le second type concerne les instructions du code et a pour objectif d'aider à la localisation de faute.

Chapitre 17
Conclusion

L'analyse dynamique s'intéresse au suivi et au résultat de l'exécution du programme. Par conséquent, elle manipule l'information issue de l'exécution des programmes. Afin de permettre la manipulation des informations issues de l'exécution d'un programme, ces informations sont stockées dans une *trace d'exécution*. Si le résultat de l'exécution d'un programme est celui attendu alors l'exécution est en succès, sinon on dit que l'exécution est un échec.

Les récents microcontrôleurs incluent des composants dédiés à la collecte de traces. Ainsi, en utilisant des sondes spécialisées, telles que Keil-UlinkPro [2] et ST-Link [6], il est possible de collecter des traces d'exécution sans données. Il est important de préciser que dans notre contexte, pour un programme embarqué, une seule trace d'exécution est disponible. Une contribution dans cette thèse, présenté dans le chapitre 4 concerne la *détection de cycles* dans une trace d'exécution. Le résultat de cette étape est une trace divisée en blocs, où chaque bloc représente un cycle spécifique (une exécution de la boucle principale). Étant donné que dans notre contexte de travail, l'enregistrement de la trace est arrêté quand un événement empêchant la poursuite de l'exécution apparaît, nous sommes sûrs de la présence d'un comportement anormal au moins dans le dernier cycle de la trace.

Notre approche de localisation de faute utilisant une trace d'exécution de microcontrôleur, présentée dans le chapitre 9, est basée sur l'analyse de différences entre les cycles de la trace. Cette approche est basée sur l'analogie entre les exécutions du programme et les cycles, car il existe sûrement des cas où les exécutions des cycles sont complètement indépendantes. Cependant, il existe également des cas où les multiples cycles d'une même exécution interagissent entre eux de plusieurs façons. Notre approche de localisation de faute est alors peu efficace pour ce type de cas.

Afin de répondre au besoin de faire de la localisation de faute en utilisant une seule trace d'exécution, pour des exécutions où les multiples cycles interagissent entre eux, nous nous sommes intéressé à la fouille de données, et plus précisément à *la recherche de règles d'association*. Dans notre contexte, les règles d'association permettent d'exprimer les liens entre les instructions, cela en utilisant les cycles d'une trace d'exécution. Dans le but de rechercher les règles d'association, les cycles d'une même trace d'exécution sont séparés en deux groupes. Le premier groupe concerne les cycles qui ressemblent le

moins au cycle fautif, ils sont considérés comme corrects. Le deuxième groupe concerne les cycles qui ressemblent le plus au cycle fautif, ils sont considérés comme fautifs. La principale motivation de l'utilisation de la fouille de données est donc d'extraire des informations de comportements considérés comme correctes et des comportements considérés comme fautifs, afin de les comparer et localiser la faute. Ce travail est inspiré des travaux de Cellier et al. [21] concernant la localisation de faute en utilisant la fouille de données et l'analyse formelle de concepts.

Une évaluation expérimentale de notre approche a été menée (chapitre 15). Notre approche a été appliquée sur 13 traces de différents programmes embarqués. Pour chaque trace, trois ensembles de règles sont générés : un ensemble de règles fiables, un ensemble de règles moyennement fiables et un ensemble de règles peu fiables. L'ingénieur commence par analyser les règles d'association considérées comme peu fiables en premier. Si cet ensemble ne lui permet pas d'identifier l'instruction fautive, il passe à l'analyse de l'ensemble de règles moyennement fiables. Si l'ensemble des règles d'association moyennement fiables ne permet pas à son tour d'identifier l'instruction fautive, la localisation de faute est alors considérée comme un échec. Nous estimons à ce stade que la localisation de faute est un échec, car notre but est de réduire l'effort d'analyse, ainsi nous souhaitons éviter à l'ingénieur l'analyse des règles d'association considérées comme étant fiables.

La métrique utilisée pour évaluer notre approche est donc l'*effort* d'analyse. L'*effort* se définit comme étant le nombre de règles analysées avant l'identification de l'instruction fautive.

D'après les résultats de l'évaluation, pour les 13 programmes, la localisation de faute en utilisant les règles d'association nécessite un effort d'analyse inférieur à 42%. Cependant, nous pensons qu'il est possible de réduire encore l'effort d'analyse en appliquant des filtres de sélection sur les règles d'association à vérifier.

Les résultats de notre évaluation dépendent des programmes utilisés. Par conséquent, la localisation de faute sur d'autres programmes, en utilisant notre approche, peut nécessiter un plus (moins) grand effort d'analyse. Les valeurs attribuées aux bornes de ressemblance ne concernent que les programmes utilisés dans notre évaluation. Par conséquent, pour d'autres programmes, avec peu de cycles qui ressemblent au cycle fautif, il faudra probablement changer ces valeurs. Sinon, il est possible de n'avoir que des règles fiables. Cela ne représente pas forcement une limite de notre approche, mais nécessite l'analyse des règles considérées comme fiables, ce que nous souhaitions éviter à l'ingénieur.

Conclusion et perspectives

Afin de permettre l'analyse du comportement des microcontrôleurs, la dernière génération de microcontrôleurs utilise une technologie permettant d'obtenir une partie de l'historique d'exécution, appelée trace d'exécution.

Souvent, les traces d'exécution comportent des informations concernant les chemins d'exécution. Cependant, ces traces peuvent contenir une importante quantité d'information, car les chemins d'exécutions peuvent être très longs. Par conséquent, étudier le comportement d'un microcontrôleurs à partir de sa trace d'exécution, afin de comprendre son comportement et l'analyser, constitue un défi couteux en temps et en effort. De plus, l'absence de données d'entrées et de sorties rend l'analyse de ces traces encore plus difficile.

Afin d'aider à l'analyse du comportement, nos partenaires industriels nous ont demandé de proposer des solution qui n'utilisent qu'une seule trace d'exécution générée à partir d'un programme embarqué. La raison est qu'un microcontrôleur est le plus souvent branché à d'autres dispositifs, tels que les capteurs de température par exemple. Par conséquent, le comportement d'un microcontrôleur est le plus souvent influencé par l'environnement dans lequel il est installé. Ainsi, lorsqu'on essaye de générer d'autres traces du programme embarqué, il est difficile de reproduire l'environnement et les conditions dans lesquelles le microcontrôleur était installé, lorsque la trace en question a été enregistrée

Dans notre travail, nous avons observé qu'un grand nombre de programmes embarqués sur les microcontrôleurs sont des programmes *cycliques*. Un programme cyclique est un programme qui s'appuie sur une boucle principale qui est parcourue indéfiniment. Nous appelons *loop-header* l'instruction qui définit cette boucle principale. La nature cyclique des programmes s'explique par le fait qu'en l'absence de système d'exploitation, le programme est en charge de scruter les entrées de façon active. À chaque itération de la boucle principale, en fonction des données d'entrée reçues de l'environnement du microcontrôleur, des actions spécifiques sont exécutées. Une trace d'exécution générée par l'exécution d'un programme cyclique est dite *trace cyclique*.

En cas de dysfonctionnement, déboguer un microcontrôleur reste une tache difficile à faire même en utilisant la trace d'exécution, qui souvent est très grande en taille.

Dans le but de faciliter l'analyse d'un microcontrôleur, dans cette thèse nous avons proposé une approche d'aide à la compréhension de la trace d'exécution ainsi que deux approches de localisation de fautes. En plus de n'utiliser qu'une seule trace d'exécution, nos approches tirent profit de l'aspect cyclique des programmes embarqués.

Notre première approche offre à l'ingénieur une description pertinente de la trace d'exécution à analyser. Cette description permet à l'ingénieur d'avoir une idée des cycles à analyser en détails. Pour générer cette description nous

utilisons la *compression basée sur la génération de grammaires*. Le principe de cette compression est basé sur la détection de répétitions dans la trace d'exécution.

L'algorithme de compression de données *Sequitur* possède plusieurs qualités pour être un prétendant pour ce type détection de séquences répétitives. Malheureusement, pour générer une compression, Sequitur ne distingue pas les cycles et considère toutes les répétitions détectées au même niveau d'importance. Par conséquent, souvent, dans la grammaire générée, les cycles sont entremêlés, ce qui rend la détection d'un cycle en particulier couteuse en temps.

Dans notre contexte de travail les *cycles* sont plus importants que les autres répétitions dans la trace et nécessitent un traitement spécifique. Pour cette raison, nous avons proposé une extension de Sequitur, appelé appelée *R-Sequitur*, et définit l'algorithme *Cyclitur*. Notre algorithme *Cyclitur* permet de faire de la détection de cycles et compresse la trace cyclique, dans le but de faciliter son analyse en offrant à l'ingénieur une description pertinente.

La *détection de cycles* repose sur l'identification du loop-header dans la trace. Souvent, due à l'absence de code source et aux optimisations apportées par la phase de compilation, la comparaison entre le code source et la trace pour l'identification des cycles devient impossible. Par conséquent, en se basant sur le modèle des programmes cycliques, nous avons défini trois métriques pour identifier le loop-header. Ce loop-header servira à la détection de cycles dans la trace.

Afin d'évaluer l'efficacité des stratégies proposées pour identifier le loop-header, une évaluation expérimentale a été menée. Les résultats de cette évaluation montrent qu'en utilisant des traces cycliques, dans la plupart des cas, notre approche permet une bonne identification du loop-header et ainsi une bonne détection des cycles dans la trace.

Dans le but de mesurer la capacité de notre approche à aider dans la compréhension et l'analyse du comportement du microcontrôleur, nous avons mené une évaluation qualitative auprès de 14 ingénieurs. Cette étude était destinée à évaluer l'impact des différents facteurs affectant la génération des compressions et plus particulièrement la détection de cycles, sur la qualité de la compression. Les résultats de l'évaluation qualitative montrent que souvent (87% des cas etudiés), notre approche permet de baisser le niveau de difficulté d'analyse du compretement du microcontrôleur.

L'évaluation quantitative de notre approche a permis de mesurer la capacité de notre approche à offrir une compression pertinente tout en conservant l'aspect cyclique de la trace. Cela en déterminant son efficacité par rapport à Sequitur. Cette évaluation concernait des traces d'exécution de microcontrôleur, mais également des traces d'exécution de réseaux. D'après les résultats de cette expérimentation, notre approche, tout en permettant une bonne détec-

tion de cycles, offre dans tous les cas une meilleure compression que Sequitur.

Au cours de l'exploitation du microcontrôleur des défaillances peuvent être observées. Pour mettre en évidence des défaillances, des méthodes de validation et de vérification peuvent être utilisées. La méthodes la plus classiquement utilisée dans l'industrie est le test [35]. Une fois une défaillance identifiée, il est nécessaire de la corriger : c'est l'étape de débogage. Cela consiste en premier lieu à rechercher la cause de la défaillance, c'est-à-dire l'instruction erronée. On parle alors de localisation de faute.

La grande implication manuelle dans la localisation de fautes la rend très coûteuse en ressources. Par conséquent, son automatisation est un enjeu important, qui peut considérablement accroître son efficacité et réduire ses coûts.

Les méthodes dynamiques permettent de répondre à ce besoin d'automatisation. Ces méthodes s'intéressent au suivi et au résultat de l'exécution du programme, par conséquent, elles manipulent l'information issue de l'exécution des programmes. Pour cela, les informations issues de l'exécution d'un programme sont stockées dans une trace d'exécution. Un spectre d'exécution est une abstraction de la trace d'exécution, contenant le verdict de l'exécution : succès ou échec. Comme les traces d'exécution, les spectres d'exécution sont collectés au cours de l'exécution du programme.

Étant donné que la collecte de trace est devenue possible sur les récents microcontrôleurs, l'utilisation des méthodes dynamiques manipulant les spectres d'exécution pour la localisation de fautes devient alors possible. Ces méthodes sont appelées « méthodes par différence de traces (spectres) », notées SBFL. L'idée principale de ces méthodes est de comparer les comportements des exécutions en échec et les comportements des exécutions en succès, afin de renvoyer les différences entre ces comportements comme information pour le débogage. Dans le but de mieux cibler la faute, des méthodes statistiques sont utilisées pour calculer un ordre total sur des prédicats, ou sur les lignes d'un programme à inspecter pour trouver la faute. Ainsi, basées sur un ensemble d'observations (exécutions), les approches automatiques de localisation de fautes fournissent au développeur chargé du débogage une liste d'emplacements susceptibles de contenir la faute.

Dans notre contexte industriel, le plus souvent, l'enregistrement de la trace est arrêté quand un événement empêchant la poursuite de l'exécution apparaît (par exemple, une exception est levée). Ainsi, ce mécanisme assure la présence d'un comportement anormal au moins dans les derniers événements enregistrés.

Afin de pouvoir appliquer les méthodes par différence de spectres, notre deuxième contribution part du principe que les exécutions des cycles d'une même trace sont indépendantes. Notre approche et donc basée sur la *détection de cycles* et l'utilisation les « méthodes par différence de spectres » sur

la trace découpée. Le cœfficient utilisé par défaut dans notre approche est Ochiai. Cette sélection est basée sur nos résultats expérimentaux concernant les différents cœfficients présentés dans cette thèse, et qui montrent l'efficacité du cœfficient Ochiai, qui fournit dans plus de 86% des cas un meilleur classement des instructions suspectes.

L'évaluation expérimentale concernait six méthodes de localisation de fautes. Ces sixméthodes sont : « le voisin le plus proche », « Tarantula », « Ample », « Jaccard », « Ochiai » et l'utilisation du cœfficient O^p. Chaque méthode a été donc appliquée sur 13 traces de différents programmes embarqués. Pour chaque trace, un diagnostic est généré pour chaque application de l'une de ces six méthodes. Ainsi, les six diagnostics générés pour chaque trace sont comparés entre eux pour évaluer les performances de leurs méthodes respectives à localiser les fautes. La métrique utilisée pour évaluer les six méthodes est appelée *effort*, qui se définit comme étant le nombre d'instructions analysées pour identifier la faute, par rapport au nombre total d'instructions dans le code source.

D'après les résultats de l'évaluation, dans 85% des cas, la localisation de fautes en utilisant les techniques basées sur les scores de suspicion nécessitent moins d'effort que la méthode du « plus proche voisin ». Le décalage d'effort entre les méthodes utilisant des scores de suspicion et la méthode du « plus proche voisin » se justifie par le manque du classement des instructions suspectes dans cette dernière. Aussi, dans 75% des cas, pour la localisation de fautes, moins de 20% du code source est analysé. Cependant, même dans le pire des cas, le code n'est jamais analysé à 100%.

Dans nos évaluations, nous avons observé que de nombreux programmes embarqués sont des programmes (quasiment) sans mémoire ou bien avec des variables locales au corps de la boucle principale. C'est-à-dire que les cycles ont des exécutions indépendantes. Notre deuxième contribution est basée donc sur l'analogie entre les exécutions du programme et les cycles. Cependant, il existe également des cas où les multiples cycles d'une même exécution interagissent entre eux de plusieurs façons. Notre approche de localisation de faute risque alors d'être peu efficace pour ce type de cas.

Ainsi, afin de répondre au besoin de faire de la localisation de faute pour des exécutions où les multiples cycles interagissent entre eux, nous nous sommes intéressés à la fouille de données, et plus précisément à *la recherche de règles d'association*.

Dans notre travail, les règles d'association nous ont permis d'exprimer les liens entre les instructions, cela en utilisant les cycles d'une même trace d'exécution. Par conséquent, comme pour nos deux premières contributions, notre dernière approche est également basée sur la détection de cycles dans la trace. Ainsi, dans le but de rechercher les règles d'association, les cycles

sont séparés en deux groupes. Le premier groupe concerne les cycles les moins ressemblant au cycle fautif, ils sont considérés comme corrects. Le deuxième groupe concerne les cycles les plus ressemblant au cycle fautif, ils sont considérés comme fautifs. La principale motivation de l'utilisation de la fouille de données est donc d'extraire des informations de comportements considérés comme correctes et des comportements considérés comme fautifs, afin de les comparer et localiser la faute.

Une évaluation expérimentale de notre approche de localisation de faute en utilisant les règles d'association a été menée en utilisant 13 traces de différents programmes embarqués. Pour chaque trace, trois ensembles de règles sont générés : un ensemble de règles *fiables*, un ensemble de règles *moyennement fiables* et un ensemble de règles *peu fiables*. L'ingénieur commence par analyser les règles d'association considérées comme peu fiables en premier. Si cet ensemble ne lui permet pas d'identifier l'instruction fautive, il passe à l'analyse de l'ensemble de règles moyennement fiables. S'il l'ensemble des règles d'association moyennement fiables ne permet pas à son tour d'identifier l'instruction fautive, la localisation de faute est alors considérée comme un échec. Nous estimons à ce stade que la localisation de faute est un échec, car notre but est de réduire l'effort d'analyse, ainsi nous souhaitons éviter à l'ingénieur l'analyse des règles d'association considérées comme étant fiables.

La métrique utilisée pour évaluer notre approche est donc l'*effort* d'analyse, qui se définit comme étant le nombre de règles analysées avant l'identification de l'instruction fautive, par rapport au nombre total de règles générées.

D'après les résultats expérimentaux, pour les 13 programmes, la localisation de fautes en utilisant les règles d'association nécessite un effort d'analyse inférieur à 42%. Ces résultats sont prométeurs. Cependant, nous pensons qu'il est encore possible de réduirel'effort, comme il est expliqué dans la section suivante.

Perspectives

Les perspectives de notre travail concernent les trois approches présentées dans cette thèse.

Pour l'aide à la compréhension de la trace d'exécution, nous souhaitons améliorer la compression générée par notre algorithme Cyclitur, en essayant de garantir une seule représentation pour chaque répétition. Il est donc nécessaire d'explorer les différentes approches proposées dans la littérature, comme par exemple l'approche définie par Kieffer et Yang dans [43], qui consiste à représenter en premier les répétitions les plus longues.

Une perspective parallèle à celle de la compression est l'amélioration de

la visualisation de la trace en proposant une animation des composants dans le temps, comme il est proposé par Wettel et al. dans [73]. Nous pensons que cela permettra d'avoir une meilleure description du comportement du microcontrôleur.

La majorité des techniques type SBFL fournissent un bon résultat dans le cas où il n'y a qu'une seule défaillance. Cependant, elles ne traitent pas le cas de multiples défaillances, qui est généralement beaucoup plus compliqué. Dans ce genre de situation, certaines défaillances peuvent masquer ou camoufler d'autres. Ainsi un échec observé peut être dû à l'interaction entre une série de fautes. Une perspective importante pour notre seconde contribution concerne donc la localisation de plusieurs fautes simultanément, en utilisant les nouvelles techniques proposées pour remédier à ce problème. Dans ce but nous souhaitons adapter et évaluer les techniques proposées dans la littérature [78, 12, 11], comme le débogage parallèle [40] par exemple. Nous nous intéressons particulièrement au travail de Cellier et al. [22], qui permet de faire de la localisation de fautes en utilisant les règles d'association et l'analyse formelle de concepts. Nous souhaitons également définir un cœfficient de similarité pour les programmes cycliques, afin d'augmenter la pertinence du diagnostic.

Les résultats de l'évaluation de notre troisième contribution, ont montré que l'utilisation des règles d'association pour la localisation de faute dans des programmes où les multiples cycles d'une même exécution peuvent interagir entre eux, nécessite un effort d'analyse moyen. Cependant, nous pensons qu'il est encore possible de réduire cet effort en appliquant des filtres de sélection sur les règles d'association à vérifier. Par exemple en ignorant l'analyse des règles d'association contenues dans une règle plus grande. Étant donné que les mêmes instructions apparaissant dans les règles plus petites apparaissent dans la règle la plus grande, analyser seulement la plus grande servira à réduire l'effort pour la localisation de faute. Une autre perspective concerne l'évaluation de notre approche pour la localisation de multiples fautes.

Concernant nos contributions, nous souhaitons également élargir nos évaluations en appliquant nos approches sur d'autres programmes embarqués.

Annexe A

Annexes

global: $\mathcal{J}, \mathcal{DJ}$ /* Global sets of lists */

Algorithm LCM()
1. $X := I(\mathcal{T}(\emptyset))$ /* The root $\perp$ */
2. **For** $i := 1$ **to** $|E|$
3. **If** $X[i]$ satisfies (cond2) and (cond3) **then**
   Call LCM_Iter($X[i], \mathcal{T}(X[i]), i$) or
   Call LCMd_Iter2($X[i], \mathcal{T}(X[i]), i, \mathcal{DJ}$)
   based on the decision criteria
4. **End for**

LCM_Iter($X, \mathcal{T}(X), i(X)$) /* occurrence deliver */
1. **output** X
2. **For each** $T \in \mathcal{T}(X)$
   For each $j \in T, j > i(X)$, insert t to $\mathcal{J}[j]$
4. **For** each $j, \mathcal{J}[j] \neq \emptyset$ in the decreasing order
5. **If** $|\mathcal{J}[j]| \geq \alpha$ and (cond2) holds **then**
   LCM_Iter($\mathcal{T}(\mathcal{J}[j]), \mathcal{J}[j], j$)
6. Delete $\mathcal{J}[j]$
7. **End for**

LCM_Iter2($X, \mathcal{T}(X), i(X), \mathcal{DJ}$) /* diffset */
1. **output** X
2. **For** each $i, X[i]$ is frequent
3. **If** $X[i]$ satisfies (cond2) **then**
4. **For** each $j, X[i] \cup \{j\}$ is frequent,
   $\mathcal{DJ}'[j] := \mathcal{DJ}[j] \setminus \mathcal{DJ}[i]$
5. **LCM_Iter2**($\mathcal{T}(\mathcal{J}[j]), \mathcal{J}[j], j, \mathcal{DJ}'$)
6. **End if**
7. **End for**

FIGURE A.1 – L'algorithme LCM

Liste des figures

Liste des tableaux

Bibliographie

[1] bibliothèque qt. http://doc.qt.digia.com/. (Cité en page 4.)

[2] Keil UlinkPro. http://www.keil.com/ulinkpro/. (Cité en pages vii, 6, 54, 103, 109, 134 et 145.)

[3] Labview. http://www.ni.com/labview/f/. (Cité en page 4.)

[4] Le microcontroleur. http://oboulo.pagesperso-orange.fr/files/TP/Le-microcontroleur.pdf. (Cité en page 2.)

[5] Mysql. http://www.mysql.fr/. (Cité en page 140.)

[6] ST Microelectronics. http://www.st.com/internet/evalboard/product/219866.jsp. (Cité en pages 6, 109 et 145.)

[7] Trace macrocells (ETM). http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php. (Cité en pages vii, 6 et 15.)

[8] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. (Cité en page 4.)

[9] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing : Academic and Industrial Conference Practice and Research Techniques - MUTATION. TAICPART-MUTATION 2007*, pages 89 –98, sept. 2007. (Cité en pages viii, 9, 80, 83, 84, 89, 104 et 110.)

[10] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society. (Cité en pages 80 et 82.)

[11] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society. (Cité en pages 88 et 153.)

[12] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, June 2009. (Cité en pages 88 et 153.)

[13] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2) :207–216, June 1993. (Cité en pages 9, 118 et 121.)

[14] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. (Cité en pages 121 et 124.)

[15] Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, and Lydie Bousquet. Compressing microcontroller execution traces to assist system analysis. In *International Embedded Systems Symposium*. Springer Berlin Heidelberg, June 2013. (Cité en pages v, vii, viii, 8, 13, 17 et 59.)

[16] Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, and Lydie du Bousquet. Single-trace fault localization in embedded software. In *International Symposium on Software Reliability Engineering*. IEEE, November 2013. (Cité en pages vi, viii, 8, 9, 65, 69, 90 et 110.)

[17] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 49–60, New York, NY, USA, 2010. ACM. (Cité en pages 9, 68 et 83.)

[18] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2) :255–264, June 1997. (Cité en page 122.)

[19] Patrick Cégielski, Irène Guessarian, Yury Lifshits, and Yuri Matiyasevich. Window subsequence problems for compressed texts. In *Proceedings of the First international computer science conference on Theory and Applications*, CSR'06, pages 127–136, Berlin, Heidelberg, 2006. Springer-Verlag. (Cité en pages 18 et 19.)

[20] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Formal concept analysis enhances fault localization in software. In *Int. Conf. Formal Concept Analysis*, LNAI 4933. (Cité en pages 9 et 118.)

[21] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. DeLLIS : A data mining process for fault localization. In *Int. Conf. Software Engineering (SEKE)*, pages 432–437. Knowledge Systems Institute Graduate School, 2009. (Cité en pages 9, 118 et 146.)

[22] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *Int. Conf. on Software Engineering & Knowledge Engineering*, pages 238–243, 2011. (Cité en page 153.)

[23] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *Information Theory, IEEE Transactions on*, 51(7) :2554 –2576, july 2005. (Cité en page 16.)

[24] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint : problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595 – 604, 2002. (Cité en pages vi, 9, 80 et 81.)

[25] Hacène Cherfi and Yannick Toussaint. Adéquation d'indices statistiques à l'interprétation de règles d'association. In P. Sébillot A. Morin, editor, *6èmes Journées internationales d'Analyse statistique des Données Textuelles - JADT 2002*, pages 233–244, Saint-Malo, France, 2002. none. (Cité en page 122.)

[26] Neva Cherniavsky and Richard Ladner. Grammar-based compression of dna sequences, 2004. (Cité en pages 16 et 21.)

[27] Francisco Claude, Antonio Farina, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proceedings of the 2010 IEEE International Conference on Bioinformatics and Bioengineering*, BIBE '10, pages 86–91, Washington, DC, USA, 2010. IEEE Computer Society. (Cité en page 18.)

[28] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *In Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 528–550. Springer-Verlag, 2005. (Cité en pages vi, 9, 68, 72, 80 et 82.)

[29] Hermelin Danny, M. Landau Gad, Landau Shir, and Weimann Oren. A unified algorithm for accelerating edit-distance computation via text-compression. *CoRR*, 2009. (Cité en page 19.)

[30] Carl G. de Marcken. Unsupervised language acquisition. Technical report, 1996. (Cité en page 19.)

[31] C.T. De Oliveira, F. Theoleyre, and A. Duda. Connectivity in multi-channel multi-interface wireless mesh networks. In *International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 35–40, 2011. (Cité en page 56.)

[32] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces. a re-interpretation of Jones, Harrold and Stasko test information visualization. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005. (Cité en page 69.)

[33] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Advances in knowledge discovery and data mining. chapter From data mining to knowledge discovery : an overview, pages 1–34. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996. (Cité en page 117.)

[34] Gemma C. Garriga, Roni Khardon, and Luc De Raedt. Mining closed patterns in relational, graph and network data. *Annals of Mathematics and Artificial Intelligence*, pages 1–28, 2012. (Cité en page 125.)

[35] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1) :4–12, January 2002. (Cité en pages 4, 67 et 150.)

[36] Rw Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2) :147–160, 1950. (Cité en pages 9, 92 et 128.)

[37] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10 :2000, 2000. (Cité en pages 8, 68, 71 et 72.)

[38] Shunsuke Inenaga and Hideo Bannai. Finding characteristic substrings from compressed texts. In *Proceedings of the Prague Stringology Conference 2009*, pages 40–54, Czech Technical University in Prague, Czech Republic, 2009. (Cité en page 19.)

[39] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. (Cité en page 82.)

[40] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 16–26, New York, NY, USA, 2007. ACM. (Cité en pages 88 et 153.)

[41] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM. (Cité en pages vi, 9, 68, 80, 104 et 110.)

[42] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM. (Cité en pages viii, 65, 68, 69, 72, 77, 80 et 89.)

[43] J.C. Kieffer and En hui Yang. Grammar-based codes : a new class of universal lossless source codes. *Information Theory, IEEE Transactions on*, 46(3) :737–754, 2000. (Cité en page 152.)

[44] J.C. Kieffer and En-Hui Yang. Grammar-based codes : a new class of universal lossless source codes. *Information Theory, IEEE Transactions on*, 46(3) :737 –754, may 2000. (Cité en page 16.)

[45] J.C. Kieffer, En-Hui Yang, G.J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *Information Theory, IEEE Transactions on*, 46(4) :1227 –1245, jul 2000. (Cité en page 16.)

[46] Sebastian Kreft and Gonzalo Navarro. Self-indexing based on lz77. In *Proceedings of the 22nd annual conference on Combinatorial pattern matching*, CPM'11, pages 41–54, Berlin, Heidelberg, 2011. Springer-Verlag. (Cité en page 18.)

[47] Gopal Lakhani and Radhakrishnan Sethurama. Using partial-matching approach with sequitur for contact-based coding. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, page 548, march 2004. (Cité en page 19.)

[48] Gopal Lakhani and Radhakrishnan Sethuraman. Context-based coding of sequitur generated grammar for improved text compression. (Cité en page 19.)

[49] J. Kevin Lanctot, Ming Li, and En-hui Yang. Estimating dna sequence entropy. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 409–418, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. (Cité en page 19.)

[50] Jean-Claude Laprie. Dependability : Basic concepts and terminology : In english, french, german, italian and japanese. *Dependable Computing and Fault-Tolerant Systems. Springer*, 1991. (Cité en page 4.)

[51] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proc. IEEE*, pages 296–305. IEEE Computer Society, 1999. (Cité en page 16.)

[52] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM. (Cité en page 19.)

[53] Eric Lehman and Abhi Shelat. Approximation algorithms for grammar-based compression. In *In Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 205–212. ACM/SIAM, 2002. (Cité en page 16.)

[54] Yury Lifshits. Processing compressed texts : a tractability border. In *Proc. CPM 2007*, pages 228–240. Springer, 2007. (Cité en page 19.)

[55] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging : A hypothesis testing-based approach. *IEEE Trans. Software Eng.*, 32(10) :831–848, 2006. (Cité en page 68.)

[56] Heikki Mannila and Hannu Toivonen. Multiple uses of frequent sets and condensed representations. In *In Proc. KDD Int. Conf. Knowledge Discovery in Databases*, pages 189–194. AAAI Press, 1996. (Cité en page 124.)

[57] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10) :900–913, March 2009. (Cité en pages 18 et 19.)

[58] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3) :11 :1–11 :32, august 2011. (Cité en pages vi, 9, 80 et 84.)

[59] Craig G. Nevill-Manning and Craig G. Nevill-manning. Inferring sequential structure. Technical report, 1996. (Cité en page 19.)

[60] Craig G. Nevill-Manning and Ian H. Witten. Compression and explanation using hierarchical grammars, 1997. (Cité en pages 13, 21 et 22.)

[61] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences : a linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1) :67–82, September 1997. (Cité en pages viii, 19, 21, 22 et 31.)

[62] Jivan S. Parab, Vinod G. Shelake, Rajanish K. Kamat, and Gourish M. Naik. *Exploring C for Microcontrollers : A Hands on Approach*. Springer Publishing Company, Incorporated, 1st edition, 2007. (Cité en pages vii et 2.)

[63] Bruno Pôssas, Nivio Ziviani, Wagner Meira, Jr., and Berthier Ribeiro-Neto. Set-based model : a new approach for information retrieval. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '02, pages 230–237, New York, NY, USA, 2002. ACM. (Cité en page 124.)

[64] Uta Priss. Formal concept analysis in information science. *Annual Review of Information Science and Technology*, 40 :521–543, 1996. (Cité en pages 9 et 118.)

[65] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39. IEEE Computer Society, 2003. (Cité en pages viii, 8, 65, 68, 73 et 76.)

[66] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ACM Software Engineering Notes*, pages 432–449. Springer-Verlag, 1997. (Cité en pages 8, 71 et 72.)

[67] A. Rohani and H.R. Zarandi. An analysis of fault effects and propagations in AVR microcontroller ATmega103(L). In *International Conference on Availability, Reliability and Security (ARES)*, pages 166 –172, March 2009. (Cité en page vii.)

[68] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *In Proc. 15th SPIRE, LNCS 5280*, pages 164–175, 2008. (Cité en page 18.)

[69] Alexander Tiskin. Towards approximate matching in compressed strings : local subsequence recognition. In *Proceedings of the 6th international conference on Computer science : theory and applications*, CSR'11, pages 401–414, Berlin, Heidelberg, 2011. Springer-Verlag. (Cité en page 19.)

[70] Takeaki Uno. Fast algorithms for enumerating cliques in huge graphs. In *Research Group of Computation, IEICE, Kyoto University*, 2003. (Cité en page 125.)

[71] Takeaki Uno. A practical fast algorithm for enumerating cliques in huge bipartite graphs and its implementation. In *89th Special Interest Group of Algorithms, Information Processing Society Japan*, 2003. (Cité en page 125.)

[72] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. Lcm : An efficient algorithm for enumerating frequent closed item sets. In *In Proceedings of Workshop on Frequent itemset Mining Implementations*, 2003. (Cité en pages vi, 125 et 140.)

[73] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities : a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM. (Cité en page 153.)

[74] Ian H. Witten. Adaptive text mining : Inferring structure from sequences. *J. Discrete Algorithms*, 2 :137–159, 2000. (Cité en page 21.)

[75] W.E. Wong, Tingting Wei, Y. Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *1st International Conference on Software Testing, Verification, and Validation, 2008*, pages 42–51, 2008. (Cité en pages 104 et 110.)

[76] Takanori Yamamoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster subsequence and don't-care pattern matching on compressed texts. In *Proceedings of the 22nd annual conference on Combinatorial pattern matching*, CPM'11, pages 309–322, Berlin, Heidelberg, 2011. Springer-Verlag. (Cité en page 19.)

[77] En-Hui Yang and J.C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. i. without context models. *Information Theory, IEEE Transactions on*, 46(3) :755 –777, may 2000. (Cité en pages 16 et 19.)

[78] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging : simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1105–1112, New York, NY, USA, 2006. ACM. (Cité en pages 88 et 153.)

Résumé

Souvent, dû à l'aspect cyclique des programmes embarqués, les traces de microcontrôleurs contiennent beaucoup de données. De plus, dans notre contexte de travail, pour l'analyse du comportement, une seule trace se terminant sur une défaillance est disponible. L'objectif du travail présenté dans cette thèse est d'aider à l'analyse de trace de microcontrôleurs. La première contribution de cette thèse concerne l'identification de cycles, ainsi que la génération d'une description pertinente de la trace. La détection de cycles repose sur l'identification du loop-header. La description proposée à l'ingénieur est produite en utilisant la compression basée sur la génération d'une grammaire. Cette dernière permet la détection de répétitions dans la trace. La seconde contribution concerne la localisation de faute(s). Elle est basée sur l'analogie entre les exécutions du programme et les cycles. Ainsi, pour aider dans l'analyse de la trace, nous avons adapté des techniques de localisation de faute(s) basée sur l'utilisation de spectres. Nous avons aussi défini un processus de filtrage permettant de réduire le nombre de cycles à utiliser pour la localisation de faute(s). Notre troisième contribution concerne l'aide à l'analyse des cas où les multiples cycles d'une même exécution interagissent entre eux. Ainsi, pour faire de la localisation de faute(s) pour ce type de cas, nous nous intéressons à la recherche de règles d'association. Le groupement des cycles en deux ensembles (cycles suspects et cycles corrects) pour la recherche de règles d'association, permet de définir les comportements jugés correctes et ceux jugés comme suspects. Ainsi, pour la localisation de faute(s), nous proposons à l'ingénieur un diagnostic basé sur l'analyse des règles d'association selon leurs degrés de suspicion. Cette thèse présente également les évaluations menées, permettant de mesurer l'efficacité de chacune des contributions discutées, et notre outil CoMET. Les résultats de ces évaluations montrent l'efficacité de notre travail d'aide à l'analyse de traces de microcontrôleurs.

Abstract

The microcontroller traces contain a huge amount of information. This is mainly due to the cyclic aspect of embedded programs. In addition, in our context, a single trace that ends at the failure is used to analyze the behavior of the microcontroller . The work presented in this thesis aims to assit in analysis of microcontroller traces. The first contribution of this thesis concerns the identification of cycles and the generation of a relevant description of the trace. The detection of cycles is based on the identification of the loop-header. The description of the trace is generated using Grammar-Based Compression, which allows the detection of repetitions in the trace. The second contribution concerns the fault localization. Our approach is based on the analogy between executions and cycles. Thus, this contribution is an adaptation of some spectrum-based fault localization techniques. This second contribution also defines a filtering process, which aims to reduce the number of cycles used by the fault localization. The third contribution considers that the multiple cycles of a same execution can interact together. Our fault localization for this type of cases is based on the use of association rules. Grouping cycles in two sets (suspect cycles and correct cycles), and searching for association rules using those two sets, helps to define the behaviors considered as corrects and those considered as suspects. This thesis presents the experimental evaluations concerning our contributions, and our tool CoMET.