UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Geração Automática de Teste Unitário com Cobertura de Código Otimizada
Integração no Frama-C

Nuno Miguel Apolinário Leitão

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa

Covilhã, Junho de 2014

Agradecimentos

O sucesso no desenvolvimento desta dissertação, bem como noutros aspetos de desenvolvimento pessoal, apenas foi possível dado a contribuição e ajuda prestada por diversas pessoas que, direta ou indiretamente, interviram de forma muito positiva neste processo.

O meu maior agradecimento vai para os meus pais, Maria João e Manuel Leitão, e irmã, Mariana Leitão, que incansavelmente me aturam todos os dias, deram-me bons exemplos e apoiam-me sempre nas minhas escolhas. Graças a eles hoje estou aqui.
Existem ainda mais dois grandes modelos que também sempre acreditaram em mim: os meus avós maternos, Isabel Leitão e José Apolinário. Preocupados por natureza e ainda que com poucos conhecimentos, tentam sempre que faça boas opções na vida, dão tudo o que têm e não têm para que tenha sucesso. Todos os dias dão um pouco de si para tornar o dia de outra pessoa melhor, tudo o que fazem é sempre com vontade e carinho.

Deixo ainda um grande agradecimento aos restantes familiares que sempre me deram apoio e ajudam quando necessário.

Existe ainda um grupo que me acompanhou ao logo de todo o ano e que todas as semanas permitia usufruir uma hora e trinta minutos de pura diversão, relaxamento e convívio: o grupo de futebol das quintas-feiras às 19h, e todos os seus elementos que, incansavelmente, jogaram semana após semana sempre com vontade e boa disposição.

Uma das pessoas que também me ajudou a crescer pessoalmente, um grande amigo de infância, Mário Pereira, e que eventualmente voltei a encontrar na Universidade. Já várias vezes lhe agradeci e vou voltar a fazê-lo pois foi uma pessoa que marcou a minha vida pessoal e académica.
Desde que entrámos para a Universidade sempre tive o seu apoio e, inicialmente, foi ele que me encaminhou para a vertente de investigação e que acabou por me apresentar o orientador de projeto de licenciatura e de dissertação de mestrado, o Professor Doutor Simão de Sousa.

O Professor Doutor Simão Melo de Sousa, orientador desta dissertação, foi o maior contributo direto para o desenvolvimento desta. Como profissional de investigação e mentor de diversos projetos, manifestou-se uma pessoa deveras incansável e sempre disponível, estando em todos os momentos receptível a prestar ajuda e/ ou esclarecimentos diversos, mesmo em períodos em que a quantidade de trabalho limitava em muito a sua disponibilidade.
Quero ainda agradecer ao Professor Doutor Simão Melo de Sousa pela oportunidade que me deu de lecionar aulas de formação. Foi um grande contributo para o meu desenvolvimento pessoal. Lecionar aulas revelou-se uma paixão, algo que adorei fazer e muitas das vezes, bastava umas horas de formação para sair da sala com um sorriso. Obviamente que para se gostar de lecionar aulas também é preciso ter excelentes pessoas como alunos, deixando também o meu agradecimento aos alunos do curso TPSI e TPSI1 pelas excelentes pessoas que são.
Agradeço ainda ao Joel Carvalho, Rui Braz e ao João Oliveira. Foi e é um enorme prazer trabalhar com eles pois permitiram-me crescer partilhando as suas experiências e conhecimentos na área da formação profissional.

Não posso deixar também de referir o Professor Doutor Paul Crocker e o Professor Doutor João Paulo Fernandes que, embora não tenham contribuído diretamente para o desenvolvimento da dissertação, mostraram-se sempre dispostos a esclarecer qualquer questão ou a prestar apoio.

Esta dissertação foi realizada no laboratório *RELiablE And SEcure Computation Group (RELEASE)*. Foi um enorme prazer trabalhar na companhia destas pessoas que constituem este grande grupo de investigação. A todas elas o meu obrigado.

Existem ainda alguns amigos que diariamente me dão um pouco do seu dia para melhorar o meu: Nuno Pereira, Diogo Tavares, Nuno Santos, Eunice Martins e Christophe Soares.
Gostaria de agradecer ao Nuno Pereira a sua boa disposição diária que alegra o dia de qualquer pessoa, estando sempre disposto a ajudar, independentemente das horas ou situação. Não posso deixar de referir as maravilhosas aulas de Zumba, Step e Yoga para o qual me aconselhou a ir e certamente não me arrependo.
O Diogo Tavares, também um grande amigo de longa data, com o qual posso sempre contar. Embora agora esteja em Lisboa, é aquela pessoa com a qual se pode sempre falar ou pedir qualquer coisa.
O Nuno Santos, outro grande amigo de longa data, o rapaz do desporto e da guitarra! É um grande exemplo que as pessoas podem mudar. Tornou-se uma motivação diária para um estilo de vida saudável.
A Eunice Martins, uma amizade relativamente recente mas que proporciona momentos diários de descontração durante as horas de trabalho, momentos em que também é necessário ter uma conversa sem sentido e rir um pouco da situação. Uma pessoa sempre com boa disposição, vontade de ajudar e aprender.
O Christophe Soares, o homem normalmente desaparecido, mas uma excelente pessoa para ter conversas sérias e também sempre disponível a ajudar.

Gostaria de deixar ainda um breve agradecimento ao André Neves, grande amigo de longa data, considerado um irmão, que sabe muito bem o impacto que teve na pessoa que hoje sou.

Houve ainda uma pessoa, que embora tenha ido para outra Universidade, foi uma das grandes amizades que surgiu nestes anos, o Filipe Oliveira. Aquela pessoa sempre disposta a opinar sobre qualquer assunto com conselhos deveras honestos e fundamentados. Uma excelente pessoa com grandes ambições e potencial.

Agradeço ainda ao João Casteleiro, atual presidente do Núcleo de Informática da UBI (NINF), pela oportunidade de me deixar ingressar como elemento da direção. Foi, não só uma novidade, como me permitiu adicionar algumas caraterísticas à minha lista de competências. Não posso deixar de agradecer também a todos os elementos que fizeram parte desta direção com a qual partilhei várias horas semanais em prol do desenvolvimento do núcleo.

Podia continuar a agradecer a todas as pessoas que contribuíram para esta dissertação e/ ou crescimento pessoal, mas essa lista seria infindável.
Embora possa haver pessoas que aqui não estejam mencionadas, elas certamente influenciaram em algum ponto fulcral durante estes anos, sejam os pequenos cafés para descontrair, as saídas à noites, as jogatanas de ténis, ou simples conselhos, ajudas ou conversas, deixando também a essas pessoas o meu sincero agradecimento.

Resumo

O teste de *software* é o principal modo de verificar a correção deste. Normalmente constitui 50% do seu custo de desenvolvimento. Existem diversas estratégias de teste de software mas, nesta dissertação, é essencialmente abordado o teste unitário, uma forma de teste que permite a análise de pequenos componentes individuais do sistema (funções, classes ou módulos). O seu principal objetivo é detetar erros lógicos nos componentes, verificar *corner cases* e tentar obter a maior cobertura de código testado possível. O grande problema desta abordagem é ser, normalmente, um processo manual, o que por sua vez o torna dispendioso, propício a erros e bastante demorado. Portanto seria uma mais valia conseguir automatizar o processo de geração e execução do teste unitário com uma elevada cobertura de código.

Nesta dissertação é apresentado e explorado o conceito de testes *Concolic* com o intuito de permitir ultrapassar algumas das limitações anteriormente mencionadas. O teste *Concolic* junta duas técnicas de teste de *software*: teste com valores concretos e simbólicos simultaneamente. Atualmente existem várias ferramentas que permitem realizar este tipo de teste de *software* e esta dissertação inspira-se sobre uma delas.

O principal foco desta dissertação foi a exploração e possível integração de uma ferramenta de teste *Concolic* no *Frama-C*. O *Frama-C*, implementado em *Objective Caml (Ocaml)*, disponibiliza um conjunto de ferramentas dedicadas a análise de código C. Fornece várias técnicas de análise estática numa única *framework* e é usado atualmente por empresas como *Airbus* e *Dassault Aviation*.
Deste modo foram analisadas e estudadas um conjunto de ferramentas que permitissem realizar teste *Concolic*. Posteriormente foi escolhida uma ferramenta (*Crest*) com o intuito adaptar os seus processo à plataforma *Frama-C* e à respetiva implementação em *Ocaml*. Esta ferramenta permite a execução de testes *Concolic* de programas desenvolvidos na Linguagem C e contempla várias etapas:

1. Inicialmente é feita automaticamente a instrumentação do código, isto é, altera-se o código original para um código que realize chamadas a funções que permitam a sua execução simbólica. Se o programa for fornecido com valores concretos, este produz o mesmo output que o original. Esta etapa é feita através do *Common Intermediate Language (CIL)*.

2. Finda a instrumentação é feito um ciclo de execução em que são armazenadas um conjunto de restrições das várias condições encontradas. Estas condições são armazenadas e posteriormente resolvidas, fazendo uso do *Yices*, e assim gerar novos valores para, na próxima execução, percorrer diferentes caminhos.

O resultado final desta dissertação contempla a adaptação dos principais funamentos do *Crest* a uma implementação *OCaml* e respetiva integração em *Frama-C*. Todas as etapas foram implementadas desde as várias etapas da instrumentação à execução dos testes, geração automática de inputs e integração no *Frama-C*.

Palavras-chave

Teste *Concolic*, Teste *Software*, Frama-C, Execução Simbólica, Execução Concreta, Pesquisa de *Bugs*, Teste Unitário.

Abstract

Software testing is the main mechanism for verifying the soundness of any software. There are several strategies for doing it. This thesis focuses, primarily, on unit testing, which allows the verification of the smallest components of the system (functions, classes or modules). The main objective of this type of test is to detect logical errors in those components, to verify corner cases and to try to obtain the largest coverage of tested code as possible. However, there are problems associated. Usually, the biggest issue involved is that this type of testing has to be done manually, which makes the process expensive, error prone and time consuming. Therefore, it would be extremely useful to automatize the generation and execution process of unit testing.

This thesis will present and explore the concept of unit testing with the purpose of allowing to overcome some of the above mentioned limitations of this activity. Concolic testing establishes a connection between two software testing techniques: the testing with concrete and symbolic values, simultaneously. Currently, there are several tools available which allow this type of testing and this thesis is inspired in one of these tools.

The main focus of this thesis was the exploration and possible integration of a Concolic testing tool in Frama-C. The Frama-C, implemented in OCaml, offers a range of tools dedicated to the analyse of C code. It provides several analysis techniques in one framework and it's used nowadays by companies such as Airbus and Dassault Aviation.
Thus, a set of tools that make it possible the execution of Concolic tests was studied and analysed. Later, Crest was chosen as a reference implementation of a Concolic test tool with the purpose of implementing it in OCaml and to integrate in Frama-C. The latter makes it possible the execution of Concolic tests of programs developed in C language and contemplates several stages:

1. Initially, we proceed to the code instrumentation, which means that the original code is changed for another one that performs calls to functions that allow the symbolic execution. If the program is furnished with concrete values, this produces the same output as the original does. This step is done using Common Intermediate Language (*CIL*).

2. When the instrumentation process ends, a new cycle of execution is done, where a whole set of constraints of the conditions that the code may contain are stored and solved, through Yices, to make it possible the generation of new values and go through different paths in the next execution.

The final result of this thesis contemplates the adaptation of the main fundamentals of Crest to an OCaml implementation and its respective integration in Frama-C. All the stages were implemented from the various instrumentation steps until tests execution, automatic generation of inputs and integration in Frama-C.

Keywords

Concolic Testing, Software Testing, Frama-C, Symbolic Execution, Concrete Execution, Bug Finding, Unit Testing.

Índice

1 Introdução **1**

1.1 Motivação e Objetivos 1
1.2 Contribuições 2
1.3 Organização do Documento 3

2 Engenharia de Teste **5**

2.1 Teste de Software 5
2.1.1 Estratégias de teste 5
2.1.2 Níveis e Categorias de Teste 6
2.1.3 Geração de Inputs De Teste 8
2.2 Trabalhos Relacionados 10
2.3 Conclusão 11

3 FACT: Arquitetura, Desenvolvimento e Implementação **13**

3.1 Ferramentas e Tecnologias Utilizadas 13
3.1.1 Yices e Ocaml Yices 13
3.1.2 Why3 14
3.1.3 CIL 14
3.1.4 Frama-C 14
3.1.5 Crest 15
3.2 Desenho do FACT 17
3.2.1 Execução Concreta 17
3.2.2 Execução Simbólica 17
3.3 Arquitetura do Sistema 19
3.3.1 Estruturas de Dados 19
3.4 Detalhes de Implementação 23
3.4.1 Instrumentação 23
3.4.2 Ligação Entre C e OCaml 33
3.4.3 Execução do FACT 36
3.4.4 Fluxo Genérico de Execução 37
3.4.5 Ciclo de Execução BDFS 38
3.4.6 Incorporação no Frama-C 38
3.5 Conclusão 40

4 Exemplos **41**

4.1 Objetivos 41
4.2 Conjunto Testes do Crest 41
4.2.1 Output de Uma Execução *Concolic* 41
4.2.2 Primeiro Teste 42
4.2.3 Segundo Teste 42
4.2.4 Terceiro Teste 43
4.2.5 Quarto Teste 43
4.2.6 Quinto Teste 44
4.3 Teste de Dimensões Representativas 46

4.4 Conclusão 50

5 Conclusão **53**
5.1 Reflexão Crítica e Trabalho Futuro 53

Bibliografia **55**

A Manual de Utilização **59**
A.1 Requisitos 59
A.2 Compilação e Instalação 59
A.3 Utilização 60
A.4 Documentação 61

B Exemplo Completo de Código Instrumentado **63**

C Conjunto de Testes do Crest **73**
C.1 Primeiro Teste 73
C.2 Segundo Teste 74
C.3 Terceiro Teste 75
C.4 Quarto Teste 76
C.5 Quinto Teste 76

Lista de Figuras

3.1 Estrutura do Crest. 15
3.2 Nova Estrutura do *FACT*. 19
3.3 Diagrama das Principais Classes. 20
3.4 Fluxo de Execução do FACT. 37

4.1 Outputs do segundo teste para o *FACT*. 44
4.2 Outputs do segundo teste para o *Crest*. 45
4.3 Resultados de cobertura obtidos para a leitura de ficheiro e navegação entre menus. 48
4.4 Resultados de cobertura obtidos para a leitura de ficheiro, navegação entre menus e funções de impressão. 49
4.5 Exemplo de execução do teste *Concolic*. 49
4.6 Resultados de cobertura obtidos para a leitura de ficheiro, navegação entre menus, funções de impressão e infrações. 50
4.7 Resultados de cobertura obtidos para o programa completo. 50

A.1 Opções disponíveis para a execução do *FACT*. 61

Lista de Tabelas

2.1 Tabela comparativa das diferentes ferramentas de teste *Concolic*. 11

4.1 Resultado do *Crest* para o primeiro teste. 42
4.2 Resultados do *FACT* para o primeiro teste. 42
4.3 Resultados do *FACT* para o segundo teste. 43
4.4 Resultados do *FACT* para o terceiro teste. 43
4.5 Resultados do *FACT* para o quarto teste. 44
4.6 Resultados do *FACT* para o quinto teste. 45

Listings

2.1 Programa com ramo condicional. 8
3.1 Exemplo de restrições do caminho de execução. 18
3.2 Simplificação de referências de memória complexas. 23
3.3 Resultado da simplificação de referências de memória complexas. 24
3.4 Simplificação de uma instrução for. 24
3.5 Resultado da simplificação de uma instrução for. 24
3.6 Simplificação de uma instrução switch. 25
3.7 Resultado da simplificação de uma instrução switch. 25
3.8 Simplificação de uma instrução While. 26
3.9 Simplificação de uma instrução Do While. 26
3.10 Resolução de um simplificação de uma instrução While. 27
3.11 Resolução da simplificação de uma instrução do While. 27
3.12 Simplificação para apenas um ponto de retorno. 28
3.13 Resultado da simplificação para apenas um ponto de retorno. 28
3.14 Adição de um else a uma instrução if. 29
3.15 Resultado da adição de um else a uma instrução if. 29
3.16 Simplificação de expressões condicionais. 29
3.17 Resultado da simplificação de expressões condicionais. 29
3.18 Formato dos dados do ficheiro “branches”. 30
3.19 Instrumentação da expressão “a*b > 3+c”. 32
3.20 Instrumentação da expressão de uma instrução *if*. 32
3.21 Instrumentação de uma instrução de atribuição. 32
3.22 Instrumentação do retorno de uma função com argumento. 33
3.23 Instrumentação de uma função que soma dois argumentos. 33
3.24 Inicialização do *OCaml runtime*. 34
3.25 Definição de uma função no OCaml. 35
3.26 Pesquisa da função declarada no OCaml. 35
3.27 Processamento dos argumentos para o Ocaml. 35
3.28 Definição da *interface* para a função “__Load”. 36
4.1 Comando para Executar o FACT. 41
4.2 Exemplo de leitura dos dados simbólicos. 47
B.1 Exemplo de programa completo em linguagem C. 63
B.2 Exemplo de programa completamente instrumentado. 65
C.1 Pesquisa de elementos num vector. 73
C.2 Utilização de uma variável simbólica em funções. 74
C.3 Programa com expressões matemáticas complexas. 75
C.4 Programa com operações de shift. 76
C.5 Programa com índices de *arrays* simbólicos 77

Lista de Acrónimos

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CIL	C Intermediate Language
CUTE	Concolic Unit Testing Engine
DART	Directed Automated Random Testing
DFS	Depth First Search
EXE	EXecution generated Executions
FACT	FrAmac Concolic Testing
Ocaml	Objective Caml
(REL)EASE	RELiablE And SEcure Computation Group
RWSet	Read-Write Set
SAGE	Scalable Automated Guided Execution
SMART	Systematic Modular Automated Random Testing
UBI	Universidade da Beira Interior

Capítulo 1

Introdução

O teste de *software* é a principal estratégia usada para encontrar *bugs* de *software*. Normalmente é feito manualmente e representa aproximadamente 40-50% [Som10] do preço final do desenvolvimento do *software*. Ainda assim, este está sujeito a erros, é caro e certamente não exaustivo. No entanto, *software* sem testes acarreta ainda mais custos, sendo uma necessidade constante a correção de erros.

Uma das principais técnicas usadas no teste de *software* é o teste aleatório, pois é rápido, é fácil gerar *inputs* aleatórios e muitas vezes são formas de atingir *corner cases*, isto é, casos que normalmente são esquecidos e podem causar problemas. A dificuldade surge em gerar estes *inputs* problemáticos, isto é, *inputs* que possam originar problemas durante a execução. Outro problema que surge aquando da utilização de teste aleatório é que normalmente não são alcançados todos os ramos do programa, pois muitos dos *inputs* gerados vão originar o mesmo caminho de execução. Existe ainda outro problema associado ao teste aleatório, mais concretamente ao teste de caixa preta, que será discutido no capítulo 2. Para este tipo de teste é necessário um oráculo, um sistema que saiba as respostas corretas para cada *input* gerado e, deste modo, consiga verificar, ou não, a correção do programa.

Para resolver o problema da descoberta de caminhos, surgiram as técnicas de execução simbólica, em que um programa é executado usando variáveis simbólicas em vez de valores concretos. Deste modo é possível descobrir os diversos caminhos de execução possíveis e a partir destes gerar valores concretos para percorrer os novos caminhos descobertos.
Novamente estas técnicas também têm os seus problemas, nomeadamente a eficácia do *Satisfiability Modulo Theories (SMT) Solvers* que resolvem as restrições dos caminhos de execução. Quando a combinação de caminhos é enorme, torna-se computacionalmente dispendioso e inviável resolver as restrições e continuar a execução.

O teste *Concolic* realiza execução simbólica e concreta simultaneamente. A grande vantagem deste tipo de teste sobre execução puramente simbólica é a existência de valores concretos que podem ser usados para refletir sobre estruturas de dados complexas assim como simplificar restrições quando estas já não estão ao alcance do *SMT*.

1.1 Motivação e Objetivos

O principal objetivo desta dissertação é a construção de uma aplicação que permita realizar testes *Concolic* para um programa desenvolvido em Linguagem C e a respetiva integração em *Frama-C* [frab].
A grande vantagem de incorporar esta aplicação no *Frama-C* é a possibilidade de permitir que o programador tenha todo um leque de possibilidades para analisar o seu software numa única ferramenta que, atualmente, já é utilizada por grandes empresas, facilitando também o pro-

cesso de divulgação desta ferramenta.

Este projeto pode ser visto como a continuação do projeto de licenciatura intitulado *TESTA - Geração Automática de Testes Unitários* [Per13], embora este ainda estivesse numa fase embrionária de desenvolvimento.

É ainda importante referir que este trabalho foi desenvolvido em parceria entre o grupo de investigação (REL)EASE (RELiablE And SEcure Computation Group) da Universidade da Beira Interior (UBI) e a empresa Educed no âmbito do projeto PROVA. Este trabalho foi financiado por uma bolsa de investigação e espera-se que posteriormente seja incorporado no PROVA.

Os objetivos previstos no plano de trabalho são os seguintes:

- Revisão da literatura sobre a engenharia dos testes de *software*;
- Estudo e análise de ferramentas de teste *Concolic*;
- Escolha de uma ferramenta capaz de realizar testes *Concolic* e respetivo estudo do seu funcionamento;
- Estudo do desenvolvimento de *plugins* para *Frama-C*;
- Desenho de uma solução de teste *Concolic* para *Frama-C*;
- Desenvolvimento e implementação da ferramenta de teste *Concolic* com base nos princípios na ferramenta estudada;
- Incorporação da ferramenta desenvolvida no *Frama-C*;
- Realização de um caso de estudo que comprove o correto funcionamento da ferramenta desenvolvida.

1.2 Contribuições

Dado o estado atual das capacidades das ferramentas de execução de teste *Concolic*, não foram feitas grandes contribuições ao estado da arte destas no âmbito desta dissertação. Mas considera-se uma etapa importante de entender e conseguir utilizar os conceitos atualmente existentes para construir uma base robusta e funcional que, eventualmente, permita a continuação e melhoria no âmbito deste tipo de teste.
Por outro lado, a principal contribuição, além de utilizar estes conceitos com sucesso numa ferramenta funcional, o *FACT*, esta foi implementada em *Ocaml* que, por sua vez, abre um leque enorme de possibilidades para o seu crescimento. Um dos fatores que diretamente resultou desta escolha foi a integração do *FACT* em *Frama-C* que possibilita outras análises para além do teste *Concolic*. No contexto da Engenharia de teste, no nosso entender, tal contribuição tem a sua relevância!

Tendo como ponto de partida o *FACT*, torna-se possível testar e implementar novas estratégias de pesquisa, métodos de execução, expandir para outros *SMT Solvers* através do *Why3* ou até mesmo utilizar informação de outros *plug-ins* do *Frama-C* em prol do teste *Concolic*.

1.3 Organização do Documento

O restante documento encontra-se dividido em quatro capítulos:

- O capítulo 2 descreve sumariamente a engenharia do teste de *software*. São ainda apresentadas as várias ferramentas utilizadas durante o desenvolvimento da dissertação e finalmente diversos trabalhos cujo foco de desenvolvimento foi o mesmo que o apresentado nesta dissertação;
- O capítulo 3 apresenta a ferramenta desenvolvida e o modo como esta se relaciona com todas as tecnologias utilizadas para o seu correto funcionamento.
- O capítulo 4 apresenta um conjunto de resultados realizado com o intuito de provar o correto funcionamento da ferramenta desenvolvida.
- O capítulo 5 apresenta as conclusões do trabalho desenvolvido. Refere ainda alguns possíveis aspetos a melhorar.
- O apêndice A apresenta o manual de utilização da ferramenta desenvolvida. Este contempla os requisitos, instruções de compilação, instalação e modo de utilização da ferramenta.
- O apêndice B apresenta um programa implementado em linguagem C e respetivo código instrumentado.
- O apêndice C apresenta cinco exemplos, considerados relevantes, para demonstrar as principais funcionalidades da ferramenta desenvolvida.

Capítulo 2

Engenharia de Teste

Este capítulo introdutório visa apresentar diversas definições relativas à área do teste de *software* e diversas ferramentas que atualmente permitem realizar teste *Concolic*.
Na primeira seção é apresentada a definição de teste de *software* e qual a sua importância no processo de desenvolvimento de software.
Na segunda seção é feita a transição do teste concreto e simbólico para o teste concreto. Finalmente, são apresentadas algumas vantagens e limitações deste tipo de teste.
Na ultima seção são apresentadas as considerações finais referentes a este capítulo.

2.1 Teste de Software

O teste de *software* é o principal modo de verificar a correção deste em prática industrial. Normalmente constitui 40-50% [Som10] do custo do desenvolvimento do software.
Num mundo ideal, queremos testar todas as combinações possíveis de *inputs* e caminhos para um determinado programa. Na esmagadora parte dos casos, isto não é possível. Até um simples programa com um ciclo e algumas condições já tem centenas de milhares de possibilidades de *inputs* que por sua vez levam a um número enorme de combinações de caminhos. Testar todos estes casos torna-se praticamente impossível, não só ao nível de recursos humanos, mas também ao nível económico.
Torna-se então necessário definir corretamente o conceito de teste de software. Como apresentado em [MS11]:

"Teste é o processo de executar um programa com o intuito de encontrar erros."

Quando se testa um programa deve-se ter a finalidade de adicionar algum valor, isto é, devemos encontrar erros. Logo, não se deve testar um programa para mostrar que ele funciona, mas sim para a encontrar o maior número possível de erros.
Dada esta definição de teste de *software* é importante definir quando, e se é possível encontrar *todos* os erros de um determinado *software*. A resposta óbvia é negativa, é normalmente impraticável, muitas vezes impossível, encontrar todos os erros de um programa.

2.1.1 Estratégias de teste

De modo a responder a alguns dos desafios do teste de *software*, surgiram estratégias de teste. Duas das principais incluem o teste de caixa preta e o teste de caixa branca, que vão ser apresentados de seguida.

2.1.1.1 Teste de caixa preta

O teste de caixa preta é uma técnica de teste que ignora o funcionamento interno do sistema e foca-se apenas na verificação do *output* gerado tendo em conta um determinado *input*. O

objetivo é ignorar completamente o comportamento interno do sistema e concentrar o teste em encontrar circunstâncias na qual o programa não produz o *output* esperado.
Nesta técnica de teste, os dados para os testes são derivados unicamente da especificação do programa, isto é, não é feito qualquer uso da sua estrutura interna.

Nestas circunstâncias, para encontrar todos os erros no programa, o melhor critério é encontrar um conjunto de testes de *input* exaustivo, isto é, criar um caso de teste para cada *input* possível, não apenas para cada *input* válido, que por si só já seria normalmente um conjunto de testes extremamente grande.

Isto revela que o teste por exaustão é impraticável e portanto reforça a ideia de que um programa testado *não é* um programa sem erros. Ainda assim é possível a automação deste tipo de teste obrigando, deste modo, a definir um oráculo que permite verificar se os *outputs* produzidos se encontram de acordo com o previsto.

2.1.1.2 Teste de caixa branca

O teste de caixa branca é uma técnica de teste que tem em conta a estrutura do programa. Ao contrário do teste de caixa preta, o teste de caixa branca obtém os seus dados de *input* com base da lógica do programa. Neste tipo de técnica o objetivo é teste por exaustão de caminhos e não de inputs. Ainda assim existem duas falhas:

- O número de caminho possíveis é, normalmente, extremamente grande. O que torna, assim como no teste exaustivo de *inputs*, este tipo de teste impraticável;
- O teste de todos os caminhos possíveis significa que foi feito um teste completo, ainda assim o programa pode conter erros em que este funciona corretamente mas, não cumpre o propósito para o qual foi desenvolvido.

2.1.2 Níveis e Categorias de Teste

Como já referido, testar *software* não é apenas verificar que o código executa corretamente, é preciso também garantir que funciona de acordo com um conjunto de especificações estabelecidas e, se o propósito do *software* é ser incorporado num outro sistema, torna-se necessário verificar se os dois comunicam corretamente. Outro aspeto bastante importante é ainda o *feedback* obtido pelo utilizador final relativo à usabilidade do software.
Para responder a todas estas diferentes necessidades são utilizados diversas categorias de teste [MS11]:

- Teste de Funcionalidade *(Functional testing)*- assegura que as funcionalidades descritas nas especificações se encontram implementadas;
- Teste de *Stress* - usado para determinar a estabilidade do um sistema quando submetido a grandes cargas de dados;
- Teste de Usabilidade - é feito na perspetiva do cliente, tenta avaliar a facilidade com que este consegue adaptar-se e utilizar o *software*;
- Teste de Segurança - pretende verificar os diversos aspetos e robustez do software relativamente à sua segurança;

- Teste de *Performance* - usado para verificar se o *software* cumpre os requisitos de *performance* estabelecidos nas especificações;

- Teste de Armazenamento - é usado para garantir que o sistema apenas faz uso dos recursos estritamente essenciais para, deste modo, ser executado sem comprometer a performance de outras aplicações;

- Teste de Regressão - verifica que novas alterações mantêm o funcionamento correto de outras funcionalidades não alteradas;

- Teste de Aceitação - o cliente testa e fornece *feedback* do *software*;

- Teste de Instalação - visa verificar erros que possam surgir durante o processo de instalação do sistema;

- Teste *Alfa* - feito por potenciais utilizadores ou uma equipa independente para que este possa ser aprovado para teste *Beta*.

- Teste *Beta* - feito pelo utilizador com o intuito de descortinar erros menos evidentes do programa.

Os níveis de teste, por sua vez, servem para responder às diversas necessidades associadas à realização dos testes de *software* nas diversas etapas de desenvolvimento deste.
Nesta dissertação foi essencialmente explorado o conceito de teste unitário, portanto este vai ser apresentado com maior detalhe.

Existem essencialmente três níveis de testes:

- Teste unitário - usado para testar pequenas unidades de código, tais como classes ou funções;

- Teste de integração - é o tipo de teste que visa verificar as ligações entre diversos componentes distintos mas que no sistema final vão comunicar entre si;

- Teste de sistema - tem como objetivo testar o sistema final, uma versão em que todos os componentes já se encontram corretamente interligados e devidamente testados.

2.1.2.1 Teste Unitário

O teste unitário é o processo de testar componentes individuais do programa. Em vez deste ser testado como um todo, o teste é focado em pequenas partes deste. Algumas vantagens desta abordagem:

1. O teste torna-se mais simples pois apenas é trabalhada uma unidade de cada vez;

2. Como consequência da alínea anterior, a tarefa de encontrar erros torna-se claramente mais fácil pois o componente a ser testado não tem qualquer tipo de dependência e então sabemos em que unidade se encontra o erro;

3. É ainda possível introduzir paralelização na execução dos testes de diferentes unidades.

2.1.3 Geração de Inputs De Teste

Como apresentado, nos testes de caixa branca e de caixa preta, existem vários conjuntos de técnicas de exploração e geração de valores de input para os testes. No âmbito desta dissertação vão ser abordados dois conceitos genéricos de geração e execução de casos de teste: execução concreta e simbólica.

2.1.3.1 Testes Concretos

A execução de testes com valores concretos, como o próprio nome indica, é uma técnica que executa o programa a testar atribuindo valores efetivos às variáveis. Estes valores podem ser gerados a partir de várias técnicas. Uma das técnicas mais utilizadas, embora não muito eficaz, é geração automática. Esta é largamente utilizada pois é fácil de implementar e bastante rápida. O grande problema é que se torna complicado gerar valores específicos de modo a que seja possível percorrer os dois ramos de uma condição.

```
int main ()
{
        int val;
        scanf("%d", &val);
        if (val == 1025)
                printf("1º Ramo.");
        else
                printf("2º Ramo.");
}
```

Listing 2.1: Programa com ramo condicional.

O exemplo apresentado no *listing* 2.1, com teste aleatório, a probabilidade de percorrermos os dois ramos da condição é extremamente baixa, o segundo ramo é facilmente executado mas, gerar o número "1025" num universo de números tão grande é muito pouco provável e portanto o primeiro ramo, muito provavelmente, não seria executado.
Esta é uma das principais limitações do teste aleatório que, por sua vez, se reflete no teste concreto. A questão é: como gerar *inputs* que permitam ter a maior cobertura de código possível? Para dar resposta a esta questão surgiu a execução simbólica.

2.1.3.2 Testes Simbólicos

A execução de testes com *inputs* simbólicos é uma técnica que executa um programa apenas com base nas condições que são encontradas ao longo da execução, isto é, em vez de ter *inputs* concretos, são fornecidos símbolos que representam valores arbitrários. Este tipo de teste implica que seja utilizado teste de caixa branca.
Este conjunto de símbolos permite construir e resolver, se possível, um conjunto de restrições que irá permitir saber quais os valores que permitem percorrer um determinado caminho de execução no programa.
Tomando como exemplo o programa anteriormente apresentado, a execução simbólica iria manter a variável "x" sem um valor e quando chegasse à condição, esta seria guardada "val == 1025". O programa iria então fornecer a restrição a uma ferramenta de resolução de restrições e esta iria indicar que o valor para a qual esta é verdadeira seria quando "val = 1025". Deste modo já existia o conhecimento de como percorrer o primeiro ramo da condição, para saber

como percorrer o segundo bastaria negar a restrição e voltar a fornece-la à ferramenta de resolução de restrições que, como se pode verificar, seria verdade desde que "val != 1025".
Nesta situação já se encontram algumas semelhanças com o teste *Concolic*, mas na execução puramente simbólica, o programa não chega a ser executado com os valores que originaram da resolução das restrições, apenas são indicados. Enquanto que na execução *Concolic* esses valores vão automaticamente ser utilizados para descobrir novos caminhos.

Como era de esperar, a execução simbólica também tem um grande problema, esta está completamente dependente do poder do *SMT Solver*. Para este exemplo relativamente simples, não haveria qualquer problema, mas para programas consideravelmente maiores e com condições aninhadas, a performance do *SMT Solver* começa-se a degradar. Para tirar alguma carga do *SMT Solver* foi então apresentada o conceito de teste *Concolic*.

2.1.3.3 Teste Concolic

O teste *Concolic* surgiu da junção da ideia de execução concreta e simbólica *(**Conc**rete + Symb**olic**)*.
Combina teste aleatório e execução simbólica para remover as limitações de ambos: os valores concretos do teste aleatório são usados para ultrapassar as limitações da execução simbólica e, a execução simbólica é usada para gerar *inputs* concretos que fornecem melhor cobertura do que o teste aleatório.
O principal objetivo é gerar *inputs* que permitam exercitar todos os caminhos de execução possíveis (até um determinado critério de paragem ser atingido).

A execução simbólica obtém restrições em cada ramo possível de divisão encontrado ao longo do caminho de execução. No fim da execução, o algoritmo calculou uma sequência de restrições simbólicas correspondentes a cada ramo. A conjunção destas restrições chama-se restrições de caminho (*Path Constraint*). Essas restrições são resolvidas e os valores obtidos irão servir para executar novamente o programa permitindo exercitar diferentes caminhos de execução.
A primeira execução do teste *Concolic*, é feita com valores aleatórios para *inputs* primitivos e *NULL* para apontadores. Posteriormente o algoritmo realiza um ciclo de execução (genérico) que contempla as seguintes etapas:

- Executa o código com o *input* concreto gerado;
- No fim da execução, uma restrição simbólica no *path constraints* é negada e resolvida usando um *SMT Solver* para gerar um novo *input* que leve o programa por outro caminho;
- O ciclo de execução repete-se com outros *inputs* e continua até explorar todos os caminhos possíveis. Este baseia-se numa estratégia de pesquisa e um critério de paragem.

Tendo novamente em consideração o exemplo apresentado no *listing* 2.1, num teste *Concolic*, inicialmente seria gerado um valor aleatório, diga-se 234, e o programa seria executado. Enquanto a execução do código ocorria, são guardadas as *path constraints* encontradas e nesta situação surgia "val == 1025". Dado que o valor gerado era 234, na primeira execução seria executado o ramo da instrução *else*. Na execução seguinte a *path constraints* era resolvida e facilmente se verifica que para o ramo da instrução *if* ser verdadeiro, a variável "val" tem de ter o valor 1025. Portanto, esse valor seria introduzido como o valor dessa variável e, desde modo, o primeiro ramo era executado.

O problema pode surgir quando o *SMT Solver* não é poderoso o suficiente para calcular valores concretos que satisfaçam as restrições. Para facilitar isto, as restrições simbólicas vão sendo simplificadas, substituindo-se valores simbólicos por valores concretos.

2.2 Trabalhos Relacionados

Existem várias categorias de geração de *inputs*, tais como teste aleatório, execução simbólica, teste *Concolic* e teste baseado em modelos (*model based testing*).
O teste baseado em modelos é muito popular mas muitas vezes impraticável dado que muitos poucos programas são desenvolvidos com modelos ou especificações formais.
O *QuickCheck* [qc] é uma das ferramentas mais utilizadas para geração de *inputs* com teste de caixa branca. Originalmente implementado para testar programas em *Haskell* [has] mas, atualmente, já existe possibilidade de o utilizar para testar outras linguagens de programação.
O *QuickCheck* necessita que o programador forneça especificações ao programa na forma de propriedades, as quais as funções devem satisfazer e o *QuickCheck* testa essas propriedades com base num grande número de casos de teste gerados automaticamente. Embora o teste seja gerado automaticamente, é necessário a correta definição das especificações. Ao realizar teste aleatório, este acarreta alguns dos problemas já referidos no capítulo 1.

Atualmente já existem alguns trabalhos realizados na área de teste *Concolic*. A ferramenta pioneira na área foi o *Directed Automated Random Testing (DART)* [GKS05]. Este deteta erros considerados normais, tais como erros em tempo de execução, violação de asserções e não terminação.
Posteriormente surgiu o *Systematic Modular Automated Random Testing (SMART)* [God07], este faz uso dos mesmos mecanismos que o *DART* mas implementa um algoritmo de pesquisa mais eficiente.
Um grande avanço foi o *Scalable Automated Guided Execution (SAGE)* [GLM12] que, embora tenha as mesmas caraterísticas do *DART*, difere num aspeto inovador: consegue operar sobre código máquina podendo assim, ser executado sobre qualquer linguagem.
Surgiram posteriormente algumas ferramentas com mais algumas funcionalidades: *Concolic Unit Testing Engine (CUTE)* [SMA05], jCUTE [SA06] e *CREST* [BS08] que são independentes mas baseiam-se nos mesmos princípios.
Existe ainda o *EXecution generated Executions (EXE)* [CGP+06], *Klee* [CDE08], *Read-Write Set (RWset)* [BCE08], *JFuzz* [JHG], *Pex* [TDH08] e *PathCrawler* [Wil10].
Deste grupo de ferramentas as únicas que são disponibilizadas publicamente é o *Pex*, *Crest* e *Klee*.

Na tabela 2.1 [QR11] é apresentado mais algum detalhe relativo ás várias ferramentas enumeradas anteriormente. Na primeira coluna encontra-se o nome da ferramenta, na segunda coluna pode-se observar a linguagem que a ferramente suporta, na terceira a plataforma em que esta pode ser instalada e configurada e finalmente a ferramenta de resolução de restrições usado em cada uma.
Embora algumas destas ferramentas possuam algumas caraterísticas inovadores relativamente a outras, todas elas se baseiam nos mesmo princípios e sofrem dos mesmos problemas:

- *Multithreading* - nenhuma destas ferramentas suporta *Multithreading*, embora atualmente

Ferramenta	Linguagem	Plataforma	Ferramenta Resolução de Restrições
DART	C	NA	lp_solver
SMART	C	Linux	lp_solver
CUTE	C	Linux	lp_solver
jCUTE	Java	Linux	
CREST	C	Linux	Yices
EXE	C	Linux	STP
KLEE	C	Linux	STP
Rwset	C	Linux	STP
JFuzz	Java	Linux	JPF
PathCrawler	C	NA	NA
Pex	.NET	Windows	Z3
SAGE	código máquina	Windows	Disolver

Tabela 2.1: Tabela comparativa das diferentes ferramentas de teste *Concolic*.

já comecem a surgir algumas técnicas [FHRV13] para ultrapassar esta limitação;

- Algumas ferramentas de resolução de restrições não suportam variáveis do tipo *float* ou *double* e operações de aritmética não linear;
- Algumas ferramentas de teste *Concolic* não suportam apontadores (*DART*) nem operações de *shifting*;
- Estas ferramentas funcionam extremamente bem em código relativamente pequeno, isto é, bibliotecas e funções, mas para sistemas industriais é necessário dividir o sistema em pequenas partes, daí serem apenas usadas em teste unitário. Frequentemente, o próprio processo de divisão tem um custo temporal muito elevado o que leva, muitas vezes, o teste unitário a ser ignorado.

Com base no *survey* já referido, a ferramenta que apresentou melhores resultados e mais funcionalidades, das que são disponibilizadas publicamente, o *Crest* revelou-se a mais completa. Não só pelo número de caraterísticas implementadas como também pelo número de estratégias de pesquisa implementadas.
Todo o trabalho realizado nesta dissertação baseou-se na implementação do *Crest*, sendo o objetivo principal desenvolver uma ferramenta que contemple todas as suas caraterísticas e esteja integrada no *Frama-C*. Deste modo, é possível expandir e adicionar novas funcionalidades.

2.3 Conclusão

Neste capítulo foram abordados vários conceitos relativos à engenharia de teste de software. Verificou-se que não existe uma técnica ótima e que funcione sempre. Existem sim, várias alternativas para diferentes situações.

Foi ainda apresentada o conceito e evolução dos testes *Concolic* e respetivas ferramentas que foram surgindo. Atualmente ainda não existe uma ferramenta ideal, isto é, uma ferramenta que funcione para qualquer situação e que funcione sempre, ainda assim nota-se uma crescente melhoria nas ferramentas de teste *Concolic*. Como resultado do estudo da literatura, o *Crest* revelou-se a ferramenta publicamente disponível com funcionalidades superiores.
No seguinte capítulo, é apresentado o funcionamento detalhado do *Crest*.

A engenharia de testes é uma área extremamente vasta e nesta dissertação foi feita uma contribuição que incide essencialmente na utilização da técnica de teste de caixa branca aliado ao teste unitário fazendo uso do teste *Concolic*.

Capítulo 3

FACT: Arquitetura, Desenvolvimento e Implementação

Este capítulo apresenta a ferramenta desenvolvida denominada de *FrAma-c Concolic Testing (FACT)*. Como o próprio nome indica, o *FACT* permite realizar teste *Concolic* através do uso do *Frama-C*.
Inicialmente são descritas as várias ferramentas e tecnologias necessárias para o correto funcionamento do *FACT*. Posteriormente serão abordados os diversos detalhes de implementação do *FACT*, desde o processo de instrumentação, à execução e geração dos casos de teste.

3.1 Ferramentas e Tecnologias Utilizadas

Durante o desenvolvimento do *FACT*, surgiu a necessidade de aprender e explorar várias tecnologias. Para tornar possível realizar teste *Concolic*, é necessário realizar teste simbólico e concreto. O teste concreto é naturalmente executado quando se executa um programa mas, no teste simbólico, existem mais alguns cuidados, nomeadamente recolher as restrições de caminhos *(path constraints)*.
Tendo como base algum do trabalho desenvolvido no *TESTA*, a melhor maneira de obter estas restrições é através do uso do *C Intermediate Language (CIL)* [cil]. Tendo estas restrições, é ainda necessário resolvê-las com uma ferramenta de resolução de restrições. Para ultrapassar este problema foi escolhido o *Yices* [yic].

A escolha do *Frama-C* surgiu como uma questão de interesse e relevância que este atualmente tem na indústria dos testes. Dado que o desenvolvimento de um *plugin* para *Frama-C* tem que ser feito em *OCaml*, o *FACT* foi também implementado em *OCaml*.

3.1.1 Yices e Ocaml Yices

Ao realizar teste *Concolic*, existe a necessidade da utilização de um *SMT solver* para resolver as restrições que são recolhidas durante a execução para, deste modo, ser possível explorar novos caminhos.

Dado que o *Crest* faz uso das capacidades do *Yices*, também se adotou este na implementação do *FACT* pois inicialmente tornaria mais simples e compreensível o funcionamento do programa. Dado o *FACT* ser implementado em *OCaml*, surgem problemas de comunicação com o *Yices*, pois este encontra-se implementado em C. Para ultrapassar este problema, foi utilizado o *OCamlYices* [ocaa].
O *OCamlYices* permite o acesso à *Application programming interface (API)* do *Yices* através da linguagem *OCaml*.

3.1.2 Why3

O *Why3* [why], reimplementação do *Why*, é uma plataforma para verificação de programas. Fornece uma linguagem para especificação e programação, denominada de *WhyML*, que faz uso de *theorem provers* externos.

O *Why3* contempla uma biblioteca *standard* de teorias lógicas e estruturas de dados para programação. Um utilizador pode escrever um programa diretamente em *WhyML* e obter, por intermédio de uma mecanismo automático de extração, um programa em *OCaml*.

O *Why3* permite a fácil integração de *theorem provers* como o *Alt-Ergo*, *CVC4*, *Simplify*, *Yices*, *Z3* entre outros que podem ser consultados na página oficial [why].
Este fornece ainda uma *API* que permite usar o *Why3* como uma biblioteca. Ao integrar o *Why3* no *FACT* é possível ter acesso não só ao *Yices*, mas a todas os *theorem provers* instalados a partir do *Why3*.
Embora este não se encontre na versão atual do *FACT*, dada a estrutura e tecnologias utilizadas, torna-se relativamente simples, adaptar a *API* do *Why3* ao *FACT*.

3.1.3 CIL

O *CIL* disponibiliza um conjunto de ferramentas que permitem realizar facilmente análise e transformações de código fonte escrito em linguagem C para código transformado em linguagem C.

No contexto desta dissertação o *CIL* é usado para pré-processar o código original. Entenda-se por pré-processar as etapas de inserir as chamadas às funções que vão realizar a execução simbólica, para recolher as restrições dos caminhos, e trabalhar o código de modo a torná-lo mais simples para, posteriormente, ser executado sem problemas e conseguir manter a coerência entre as instruções utilizadas. Estas etapas vão ser descritas com maior detalhe na seção 3.4.

Outro aspeto importante a referir é que o *Frama-C* contempla a integração de uma versão do *CIL* adaptada. Para manter o número de dependências do *FACT* ao minimo, foi utilizada a versão do *CIL* embutida no *Frama-C*. Esta contempla algumas diferenças relativamente à versão original mas, que no âmbito desta dissertação, não são relevantes mencionar.

3.1.4 Frama-C

O *Frama-C* disponibiliza um conjunto de ferramentas dedicadas a análise de *software* implementado em C. Fornece várias técnicas de análise estática numa única *framework*. A grande vantagem de possuir todas estas técnicas num local só, é que os resultados de uma determinada ferramenta podem ser utilizados noutra sem qualquer problema de compatibilidade.

O *Frama-C* encontra-se organizado numa arquitetura de *plug-in*. Existe um *Kernel* comum que centraliza informação e realiza a análise, permitindo que, deste modo, os *plug-ins* interajam entre si. Esta arquitetura garante ainda robustez no desenvolvimento do *Frama-C* e uma fácil adição de funcionalidades.

3.1.5 Crest

O *Crest* foi a ferramenta escolhida como base de estudo para a implementação do *FACT*. O *Crest* encontra-se implementado em linguagem C++ e permite realizar teste *Concolic* a programas implementados em linguagem C.

3.1.5.1 Arquitetura do Crest

O *Crest*, como se pode observar na figura 3.1, é constituído por quatro diretorias principais:

- "base" – esta diretoria contém a declaração e especificação de todas as classes e tipos necessários para o *Crest*. Estes irão ser apresentados em maior detalhe na seção 3.3;
- "libcrest" – esta diretoria contém apenas um ficheiro onde se encontram implementadas todas as funções que irão ser disponibilizadas ao utilizador de modo a que este consiga instrumentar corretamente o código a testar;
- "process_cfg" – esta diretoria também contém apenas um ficheiro e o seu propósito é processar o *Control Flow Graph (CFG)* do programa a ser testado. Este apenas é utilizado se a estratégia de pesquisa fizer uso do *CFG*;
- "run_crest" – esta contém os principais ficheiros do programa, o ficheiro "run_crest" é onde a execução do *Crest* é instanciada e o ficheiro "concolic_search" é o ficheiro que contém as diferentes estratégias de pesquisa implementadas e, na prática, é o que permite realizar o teste *Concolic*.

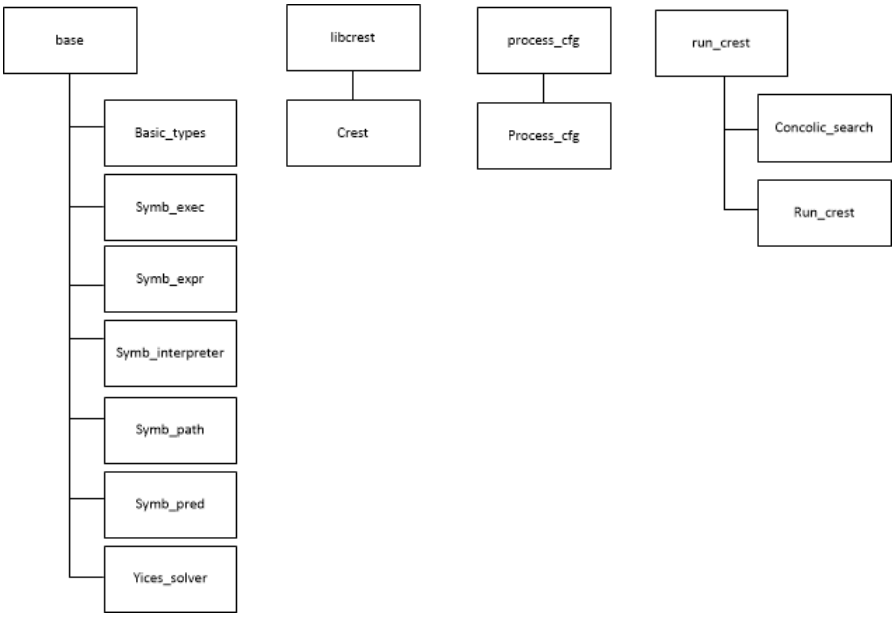

Figura 3.1: Estrutura do Crest.

3.1.5.2 Estratégias de Pesquisa

As estratégias de pesquisa são um fator de extrema importância para um ótimo desempenho da ferramenta, não só a nível de performance mas também se encontra relacionado com a cobertura de código e a capacidade de exploração dos possíveis caminhos de execução.

Uma das estratégias de pesquisa mais utilizadas no âmbito do teste *Concolic* é o *Bounded Depth-First Search (BDFS)*. Esta é semelhante à tradicional pesquisa em profundidade exceto no aspeto em que limita a profundidade de pesquisa evitando, por exemplo, ciclos infinitos.

O *Crest* implementa um conjunto razoável de estratégias de pesquisa:

- Pesquisa aleatória de caminhos.
- Pesquisa aleatória de *inputs*.
- Pesquisa com *BDFS*;
- Pesquisa com *CFG*;
- Pesquisa híbrida [MS07] — este tipo de pesquisa é potencialmente útil quando o programa possui um ciclo infinito, pois torna o número de caminhos de execução também infinito. Inicialmente, esta estratégia de pesquisa começa com teste aleatório para aumentar a cobertura. Quando o teste satura, isto é, deixa de descobrir novos caminhos no fim de um determinado número de etapas, o algoritmo automaticamente muda para execução *Concolic* de modo a realizar *BDFS* com o intuito de descobrir um novo caminho. Quando este é descoberto, o algoritmo volta à execução concreta.

As estratégias de pesquisa podem ditar a eficácia da ferramenta. Dado que grande parte das ferramentas de teste *Concolic* fazem uso de *BDFS*, optou-se por apenas implementar esta estratégia no *FACT*.

3.1.5.3 Utilização do Crest

Para testar um programa fazendo uso das capacidades do *Crest*, torna-se necessário, para o programador, realizar três etapas distintas:

- Indicar as variáveis nas quais vai ser necessária realizar execução simbólica. São suportados três tipos primitivos de variáveis: *int*, *short* e *char*. Para esse propósito são disponibilizadas as seguintes funções:
  - *CREST_int(var)*;
  - *CREST_short(var)*;
  - *CREST_char(var)*;
  - *CREST_unsigned_int(var)*;
  - *CREST_unsigned_short(var)*;
  - *CREST_unsigned_char(var)* ;
- Instrumentação automática do código:
  - *crestc program.c*
- Execução do programa instrumentado:
  - *run_crest ./ program <num_iter> -<estratégia_pesq>*.

3.2 Desenho do FACT

Antes de inciar o desenvolvimento do *FACT* é necessário considerar várias caraterísticas necessárias no teste *Concolic*. Como já referido, é necessário conseguir realizar execução simbólica, concreta e posteriormente integrar os dois.
Além destas necessidades, surgem ainda outros desafios necessários resolver de modo a garantir que o *FACT* está apto a realizar teste *Concolic*: é necessário resolver as restrições recolhidas com a execução simbólica, e é preciso conseguir integrar o programa como um todo no *Frama-C*. Surgiu ainda uma outra dificuldade adjacente ao facto de o *FACT* estar implementado em *Ocaml* e o programa em C necessitar de chamar algumas funções deste. As principais dificuldades neste âmbito, foi realizar a execução simbólica e concreta simultaneamente, ainda que esta tarefa tenha sido simplificada ao utilizar os princípios presentes no *Crest*.

3.2.1 Execução Concreta

A execução concreta permite que o programa seja executado com *inputs* concretos e, deste modo, garantir que o código é testado através da execução deste, ao contrário do que acontece com a execução puramente simbólica em que o código acaba por apenas ser analisado e, eventualmente é produzido um *output* com os possíveis valores de *input* para a qual irá ser percorrido um determinado caminho de execução.

Esta etapa não é claramente alvo de muita dificuldade dado que qualquer programa a realiza automaticamente. No entanto, existe um cuidado a considerar: quando se pretendem percorrer diferentes caminhos de execução é necessário, de algum modo, permitir que os *inputs* que garantam esses caminhos de execução sejam passados para o programa e posteriormente atribuídos às variáveis correspondentes. Esta situação é resolvida com a criação de um ficheiro temporário, denominado de "input", para cada execução do programa a testar. Ao ser iniciada a execução do programa é automaticamente feita uma chamada a uma função, introduzida no processo de instrumentação, apresentado na seção 3.4.1, que irá ler os *inputs* e atribui-los às variáveis correspondentes.

Naturalmente é feita a execução concreta do programa mas, para ser possível perceber quais as restrições para determinado caminho de execução, é necessário realizar execução simbólica.

3.2.2 Execução Simbólica

É a partir da execução simbólica que é possível recolher as restrições encontradas durante a execução concreta de um programa. Esta é claramente a etapa de implementação de maior dificuldade mas, ao seguir o principio de execução do *Crest*, a tarefa tornou-se relativamente mais simples, não ao nível de implementação, mas sim de planeamento e estruturação da solução.

Para realizar a execução simbólica durante a execução do código original, este necessita de ser modificado de modo a incorporar funções que, de certo modo, consigam entender o código que as rodeia e assim armazenar a informação necessária. Essa informação é posteriormente passada para o *FACT* para ser processada e, deste modo, gerar novos *inputs* que permitam exercitar novos caminhos de execução.
As restrições de caminhos que podem ser encontradas no programa refletem-se através do uso

de instruções *if ... else* em que tendo em conta determinada condição, o programa segue um determinado ramo de execução. A condição que se encontra dentro do *if* é a restrição de caminho que se quer obter. Obviamente que para conseguir resolver essa restrição é preciso saber qual foi o percurso pelas quais as variáveis que esta contém percorreram e as transformações a que estas estiveram sujeitas. Veja-se o exemplo apresentado no *listing* 3.1 em que a condição que se encontra na instrução *if ... else* é *val == 1025*.

```
int main ()
{
        int val;
        scanf("%d", &val);
        for (i = 0; i < 10; i++)
        {
                val++;
        }
        if (val == 1025)
                printf("1º␣Ramo.");
        else
                printf("2º␣Ramo.");
}
```

Listing 3.1: Exemplo de restrições do caminho de execução.

Se apenas fossem recolhidas informações relativas às instruções condicionais, o *FACT* não iria ter muito sucesso para descobrir novos caminhos pois, se a variável "var" fosse considerada simbólica, o *FACT* iria gerar um caso em que a variável tivesse o valor de "1025" e outro diferente com o intuito de percorrer os dois ramos. Mas, se o valor "1025" fosse o valor inicial da variável, esta iria atingir o ramo da instrução *else* dado que antes de alcançar a instrução *if*, existe um ciclo que incrementa o valor da variável por 10, sendo o valor final da variável "1035".
Deste modo, é necessário considerar todo o código existente no programa para a execução simbólica, o que torna esta tarefa mais complexa. No entanto, existe ainda outro problema: o uso de funções externas ao programa a testar, por exemplo, "printf(), strcmp()", entre outras. O programador, na prática, não tem acesso ao código destas funções, o que torna muito complicado o manipular o *output* destas tendo em conta que o seu conteúdo não pode ser processado tornado-se assim, uma das limitações da execução simbólica.

Existem ainda outros fatores, refentes à linguagem C, que necessitam de algum cuidado no processamento das instruções. Por exemplo, existem três tipos de ciclos diferentes em linguagem C e podem haver vários níveis de acesso de apontadores. Estes e outros fatores irão ser discutidos com maior detalhe na seção 3.4.1.

Para realizar a execução simbólica, foi definido um conjunto de instruções que fazem uso de uma pilha de execução com o intuito de simular o comportamento e execução para todas as instruções que um programa em linguagem C possa conter. Estas funções são abordadas com maior detalhe na seção 3.4.1.1.

Deste modo, resta apenas conseguir introduzir essas funções nos sítios corretos. A esta etapa é dado o nome de *instrumentação do código*. Esta torna possível realizar execução simbólica de modo a armazenar todas as restrições para cada caminho de execução e variáveis recolhidas

durante a execução concreta do programa. Depois de obter toda esta informação, é apenas necessário processá-la e posteriormente dar início a outra execução com valores concretos que permitam executar caminhos de execução diferentes.

3.3 Arquitetura do Sistema

No que respeita à arquitetura do *FACT*, esta sofreu pequenas modificações, presentes na figura 3.2 relativamente à estrutura original do *Crest*:

- Dado que o *Crest*, assim como o *Frama-C*, também integra uma versão do *CIL* com algumas extensões utilizadas no processo de instrumentação, foi necessário transformar e adaptar essas extensões para o *FACT*. Essas extensões encontram-se sobre a forma de três ficheiros: "FactInstrument", "OneReturn" e "Simplemem", que são descritos em maior detalhe na seção 3.4;
- A pasta "lib" contém o código *OCaml* das funções de instrumentação e respetivo código para estas serem executadas a partir da *interface* C.
- Foi ainda adicionada a pasta "c_files" que contém o ficheiro com o código que permite a interligação das funções de instrumentação executadas no código C, com as funções que se encontram no ficheiro "lib".

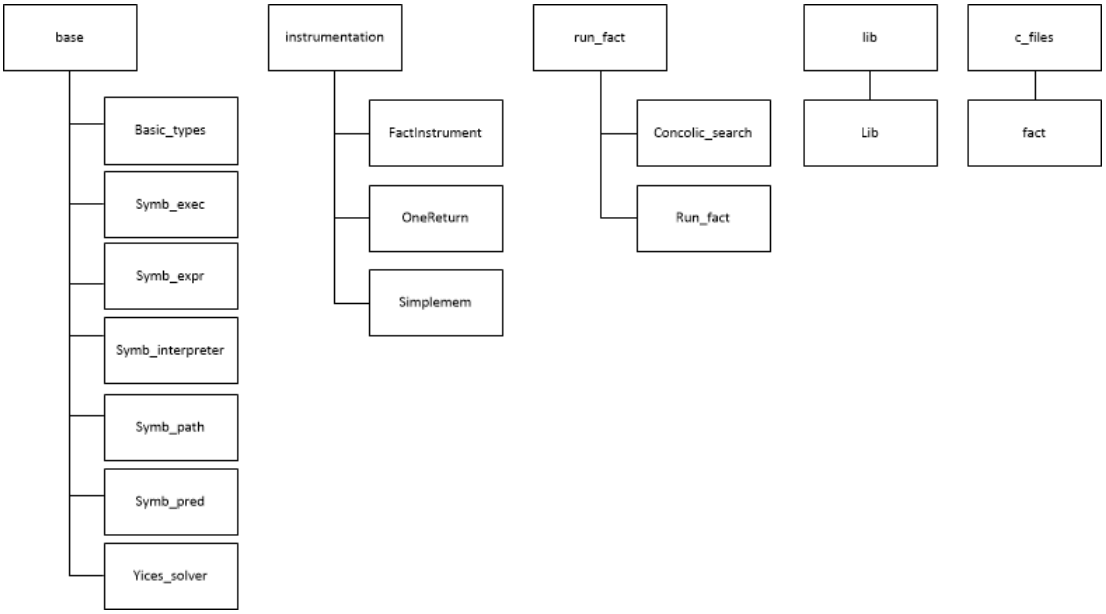

Figura 3.2: Nova Estrutura do *FACT*.

3.3.1 Estruturas de Dados

Como apresentado na Figura 3.2, a pasta "base" contempla cinco ficheiros com a implementação das estruturas que servem de suporte ao *FACT*.
Pode-se ainda verificar na Figura 3.3 quais as relações entre as principais classes. É importante referir que todas elas fazem uso dos tipos básicos definidos no ficheiro "base". Nas seguintes sub-seções são apresentadas com maior detalhe o propósito de cada uma destas estruturas de dados.

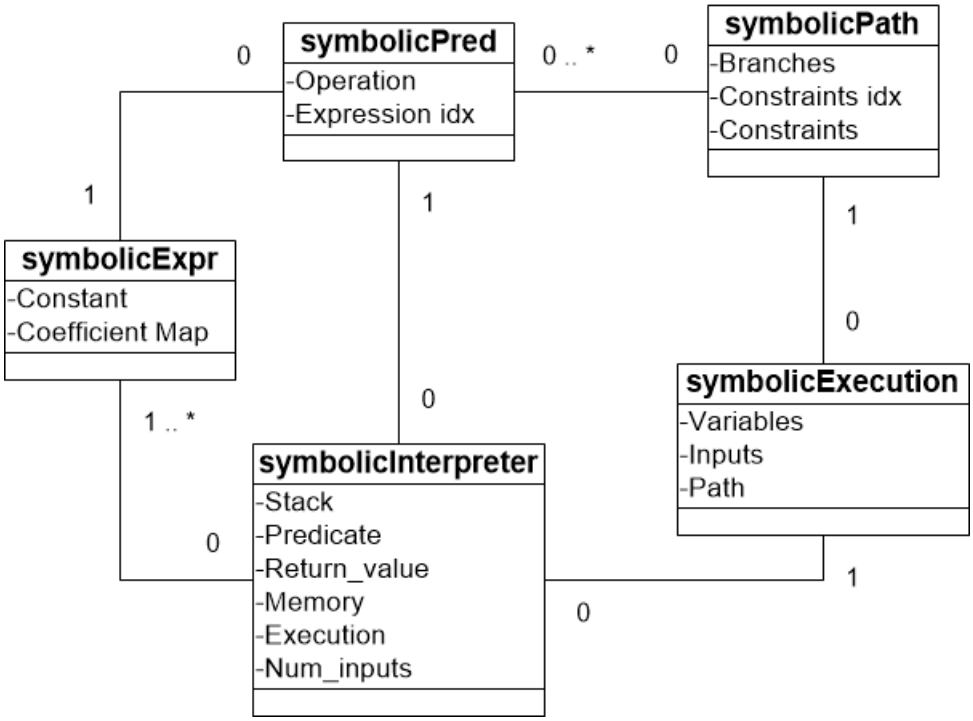

Figura 3.3: Diagrama das Principais Classes.

3.3.1.1 Basic Types

Este ficheiro contempla os tipos básicos necessários para o correto funcionamento do *FACT*. Encontram-se definidos os seguintes tipos:

- *id_t* — esta definição representa uma variável do tipo inteiro *(int)* que representa os identificadores das chamadas às funções de instrumentação;
- *branch_id_t* — esta definição representa uma variável do tipo inteiro *(int)* que representa os identificadores dos possíveis blocos de código condicionais;
- *function_id_t* — esta definição representa uma variável do tipo inteiro *(int)* que representa os identificadores de funções;
- *var_t* — esta definição representa uma variável do tipo inteiro *(int)* que representa os identificadores das variáveis consideradas simbólicas;
- *value_t* — esta definição representa uma variável do tipo inteiro de 64 *bits (int64)* que representa o valor de uma variável;
- *addr_t* — esta definição representa uma variável do tipo inteiro de 64 *bits (int64)* que representa o endereço de uma variável;

Por uma questão de leitura de código e facilidade de implementação, foram definidas operações binárias e unárias. Para as funções de comparação, tais como igualdade, negação, sinais de maior, menor e maior ou igual e menor ou igual, foi definida a variável *compare_op_t*.
Para as restantes operações binárias, tais como soma, subtração, multiplicação, *shift* ou qualquer outra operação que não seja suportada, foi definida a variável *binary_op_t*. Finalmente foram definidas as operações unárias, tais como negação e complemento lógico, na variável *unary_op_t*.

Uma das limitações das ferramentas de teste *Concolic*, nomeadamente das ferramentas de resolução de restrições é a não capacidade de trabalhar com números reais, portanto foram definidos os seguintes tipos de variáveis que podem ser consideradas simbólicas: *char, short, int* e as respetivas sem sinal *(unsigned)*. Ao serem consideradas simbólicas, torna-se possível recolher informações relativas à execução das variáveis e como consequência, é possível gerar novos valores para diferentes execuções de modo a calcular e percorrer diferentes caminhos de execução. Caso uma variável não seja simbólica, esta vai ter sempre o mesmo valor inicial em todas as execuções do programa a testar.

3.3.1.2 Symbolic Expression

A classe *symbolicExpr* representa uma expressão simbólica e é constituída por três variáveis distintas:

- *const_* — variável do tipo *int64* em que se uma expressão é simbólica então o seu valor é 0, caso contrário terá o valor respetivo.
- *coeff_* — esta variável é do tipo *VartValuetMap*. No caso da expressão ser simbólica contém a equivalência entre o identificador das variáveis e o valor temporário destas.

3.3.1.3 Symbolic Predicate

A classe *symbolicPred* representa uma predicado simbólico, isto é, representa uma função cujo *output* é verdade ou falso.
Esta classe contém duas variáveis:

- *op_* — a operação de comparação que o predicado contém: igualdade, negação, maior, menor, maior e igual ou menor e igual;
- *expr_* — representa a expressão associada ao predicado.

Quando surge uma operação binária aplicada a um valor simbólico, é adicionado o predicado simbólico às restrições do caminho percorrido.

3.3.1.4 Symbolic Path

A classe *symbolicPath* é usada para representar um conjunto de restrições encontradas ao longo do caminho de execução.
Esta classe contempla três variáveis:

- *branches_* — é uma lista de tipo *branch_id_t* que representa o conjunto de ramos na qual já foram executados, sejam chamadas a funções ou casos condicionais (instrução *if*);
- *constraints_idx_* — é uma lista de tipo inteiro *(int)* em que cada posição contém o número total elementos da lista *branches_* cada vez que é adicionado um novo predicado simbólico.
- *constraints_* — é uma lista de tipo *symbolicPred* em que cada posição contém um predicado simbólico para cada ramo encontrado (instrução *if*).

Se apenas for feita a chamada a uma função, apenas é adicionado o id do ramo ao caminho. Caso surja uma instrução *if* é adicionada o predicado que permitiu ao caminho de execução continuar por um determinado ramo.

3.3.1.5 Symbolic Execution

A classe *symbolicExecution* reflete uma execução simbólica do programa instrumentado. Existem essencialmente três aspetos a salvaguardar durante uma execução simbólica:

- As variáveis simbólicas declaradas que, por sua vez, são armazenadas num mapa cujo tipo é *VartTypetMap*, chamada de *vars_*, em que para cada variável é guardado o seu tipo;
- Os valores dos *inputs* dados ao programa. A variável *inputs_* do tipo lista de *value_t* guarda os valores dos *inputs* que são processados pelo programa;
- O caminho percorrido. Para tal foi criada uma variável chamada *path_* do tipo *symbolicPath*.

É criada uma instância desta classe cada vez que o programa instrumentado é novamente executado e posteriormente é devolvida a respetiva informação para o *FACT* esta informação será utilizada para gerar os novos *inputs* e dar continuidade à execução.

3.3.1.6 Symbolic Interpreter

Esta classe representa basicamente um interpretador para uma execução simbólica. A implementação de cada uma das funções de instrumentação encontram-se nesta classe. Além de implementarem estas funções, a classe *symbolicInterpreter* possui ainda um conjunto de variáveis que permite simular o processo de execução do programa:

- *stack_* — esta variável representa a pilha de execução do programa, encontra-se definida como uma lista de tuplos de expressão simbólica e o valor da expressão.
- *pred_* — esta variável do tipo *symbolicPred* armazena temporariamente um predicado simbólico cada vez que é encontrado um ramo condicional ou é aplicada uma operação binária;
- *return_value_* — esta variável apenas vai indicar se existe ou não valor de retorno de uma função. Caso o valor exista, este pode ser encontrado na pilha;
- *mem_* — é necessário manter armazenado as variáveis definidas. Esta tem uma correspondência entre os endereços das variáveis com a respetiva expressão;
- *ex_* — finalmente existe a variável que serve para guardar o resultado de uma execução simbólica. A variável *ex_* é do tipo *symbolicExecution*.

3.3.1.7 Yices Solver

A classe *yicesSolver* foi criada com o intuito de conter as funções que permitem resolver as várias restrições encontradas durante a execução. Esta é compostas por duas funções:

- *incrementalSove* — esta função visa resolver apenas uma restrição (a última encontrada) por cada execução. Trabalha as restrições a resolver de modo a que estas possam ser depois fornecidas como *input* para a função *solve*.
  Inicialmente é criado um grafo de dependências em que para cada identificador de uma variável (*var_t*) é atribuído os restantes identificadores das variáveis na qual a restrição depende.
  De seguida é criada uma pilha com cada um dos identificadores das variáveis da restrição a

resolver e um mapa, com as variáveis dependentes, que permite fazer a correspondência entre esses elementos da pilha e os respetivos tipos.
Finalmente, o grafo inicialmente criado é utilizado para completar as variáveis dependentes. Por uma questão de otimização são novamente verificadas as restrições que fazem uso dessas variáveis, e apenas estas serão passadas para a função *solve* de modo a simplificar os cálculos do *Yices*.

- *solve* — a função *solve* aceita como argumentos as variáveis e respetivos tipos e também a lista de restrições a resolver (do tipo *symbolicPred*). Dados estes argumentos, esta função faz uso das funcionalidades do *Yices* para as resolver. Nesta fase são necessárias algumas considerações e conhecimento do funcionamento do *OCamlYices*. É necessário passar toda a informação recebida para um formato que o *Yices* consiga entender e trabalhar. É então necessário declarar as variáveis, transformar as restrições com base na lista recebida e finalmente solicitar ao *Yices* que as resolva. Se houver solução, então são registados os valores encontrados para cada variável permitindo, deste modo, obter o valor para cada variável das restrições e assim dar continuidade à execução do programa.

3.4 Detalhes de Implementação

Esta seção descreve os detalhes de implementação do *FACT*. A primeira etapa na realização do teste *Concolic* é a fase de instrumentação, a fase em que o código original é trabalhado e aprimorado para a execução *Concolic*.
Dado que o *FACT* foi desenvolvido na integra em *OCaml* é também necessário criar a ligação entre o código C e respetivas funções de instrumentação para cada uma das suas implementações. Finalmente é ainda referido como se encontra implementado o ciclo de execução do *FACT* e a respetiva estratégia de pesquisa implementada *(BDFS)*.

3.4.1 Instrumentação

O processo de instrumentação é a parte que requer mais cuidados. Portanto, são tidos em conta diversos cuidados para garantir que o programa a ser testado executa sem problemas. Todas as etapas aqui apresentadas foram baseadas na implementação do *Crest*.
Para a execução desta etapa foi utilizado o *CIL*, já incluído no *Frama-C*, para percorrer e alterar o código fonte do programa de modo a torná-lo mais simples, mantendo a coerência entre diversas instruções C, e garantir a capacidade de execução simbólica.

Todo este processo de preparação e simplificação do código contempla 9 etapas:

- Simplificação de expressões com referências de memória complexas. O objetivo é simplificar o processo de instrumentação pois deste modo quando uma variável simbólica é introduzida na memória, na classe *symbolicInterpreter* apenas são introduzidas endereços diretos para estas. No exemplo apresentado no *listing* 3.2 é apresentada uma função que recebe como parâmetro uma variável do tipo *int***. A função limita-se a imprimir o valor que se encontra na segunda linha, terceira coluna.

```
void simpleMemory(int** m)
{
  printf("Function␣%d\n", m[2][3]);
```

```
}
```

Listing 3.2: Simplificação de referências de memória complexas.

A ideia é desdobrar o acesso à variável e, como apresentado no *listing* 3.3, são criadas novas variáveis em que são posteriormente incrementadas tantas posições quantas as necessárias para aceder à posição de memória pretendida.
Esta etapa é necessária pois um endereço de memória considerado complexo, não passa de vários apontadores. Deste modo, torna-se simples e direto indicar qual o endereço de uma determinada variável simbólica.

```
void simpleMemory(int **m)
{
  int *mem__4;
  int **mem_;
  mem_ = m + 2;
  mem__4 = *mem_ + 3;
  printf("Function␣%d\n",*mem__4);
  return;
}
```

Listing 3.3: Resultado da simplificação de referências de memória complexas.

- Conversão de instruções *for* e *switch* para *goto*, *if* e *while*. Esta etapa permite manter a coerência das chamadas às funções de instrumentação, visto que, apenas é necessário instrumentar um tipo de ciclo (*while*). No exemplo apresentado no *listing* 3.4, a função contém um ciclo *for* que imprime os valores de 0 até uma dimensão *dim*.

```
void simpleFor(int dim)
{
  int i;
  for(i = 0; i < dim; i++)
    printf("Iteration:␣%d\n",i);
}
```

Listing 3.4: Simplificação de uma instrução for.

Como apresentado no *listing* 3.5, para transformar este ciclo num *while* é introduzido um *if* com a condição de paragem do *for* e, quando esta é atingida, a execução do código é lançada para fora do *while* através de um *goto*. Enquanto esta não é atingida o corpo do *for* é executado normalmente, dentro do *while*.

```
void simpleFor(int dim)
{
  int i;
  i = 0;
  while (1) {
    while_0_continue: /* internal */ ;
    if (i < dim) {
      ;
    }
```

```
    else {
      goto while_0_break;
    }
    printf("Iteration: %d\n",i);
    i ++;
  }
  while_0_break: /* internal */ ;
  ;
  return;
}
```

Listing 3.5: Resultado da simplificação de uma instrução for.

No exemplo apresentado no *listing* 3.6 encontra-se uma função com uma instrução *switch* com dois casos específicos e o caso por omissão.

```
void simpleSwitch(int dim)
{
  switch (dim)
    {
    case 0:
      printf ("Case 0\n");
      break;
    case 1:
      printf ("Case 1\n");
      break;
    default:
      printf ("Default Case\n");
      break;
    }
}
```

Listing 3.6: Simplificação de uma instrução switch.

O objetivo, como se verifica no *listing* 3.7, passa por transformar o *switch* num conjunto de instruções *if*. Quando uma determinada condição é atingida, a execução do programa é direcionada para o ramo correspondente.

```
void simpleSwitch(int dim)
{
  if (dim == 0) {
    goto switch_0_0;
  }
  else {
    if (dim == 1) {
      goto switch_0_1;
    }
    else {
      goto switch_0_default;
      goto switch_0_break;
```

```
    }
  }
  switch_0_0: /* internal */
  printf("Case␣0\n");
  goto switch_0_break;

  switch_0_1: /* internal */
  printf("Case␣1\n");
  goto switch_0_break;

  switch_0_default: /* internal */ ;
  printf("Default␣Case\n");
  goto switch_0_break;

  switch_0_break: /* internal */ ;
  ;
  return;
}
```

Listing 3.7: Resultado da simplificação de uma instrução switch.

Neste próximo exemplo, são apresentados dois ciclos: um ciclo *while* e *do while* cujo propósito é imprimir os valores entre 0 e a variável *dim*.
A diferença entre os dois é o local do teste do critério de paragem, no exemplo apresentado no *listing* 3.8, o teste, é feito à cabeça e no outro exemplo apresentado no *listing* 3.9, o teste, é feito apenas no fim garantindo que pelo menos uma execução é realizada.

```
void simpleWhile(int dim)
{
  int i=0;
  while(i < dim){
    printf("Iteration:␣%d\n",i);
    i++;
  }
}
```

Listing 3.8: Simplificação de uma instrução While.

```
void simpleDoWhile(int dim)
{
  int i=0;
  do{
    printf("Iteration:␣%d\n",i);
    i++;
  }while(i < dim);
}
```

Listing 3.9: Simplificação de uma instrução Do While.

Como se pode constatar, ambas as soluções são muito semelhantes, alternando apenas a instrução *if* que verifica se o critério de paragem foi, ou não, atingido.
Neste primeiro exemplo, apresentado no *listing* 3.10, a instrução *if* aparece logo no início do ciclo e apenas depois é executado o código, enquanto que no segundo, apresentado no *listing* 3.11, primeiro é executado o código e posteriormente é feito o teste ao critério de paragem.

```
void simpleWhile(int dim)
{
 int i;
 i = 0;
  while (1) {
    while_0_continue: /* internal */ ;
    if (i < dim) {
      ;
    }
    else {
      goto while_0_break;
    }
    printf("Iteration: %d\n",i);
    i ++;
  }
  while_0_break: /* internal */ ;
  ;
  return;
}
```

Listing 3.10: Resolução de um simplificação de uma instrução While.

```
void simpleDoWhile(int dim)
{
  int i;
  i = 0;
  while (1) {
    while_0_continue: /* internal */ ;
    printf("Iteration: %d\n",i);
    i ++;
    if (i < dim) {
      ;
    }
    else {
      goto while_0_break;
    }
  }
  while_0_break: /* internal */ ;
  ;
  return;
}
```

Listing 3.11: Resolução da simplificação de uma instrução do While.

- Transformação de funções para terem apenas um ponto de *retorno*. O objetivo é simplificar o código de modo a garantir que uma função tem apenas um ponto de saída. Este aspeto pode revelar-se útil caso seja necessário introduzir um código que seja sempre executado quando a função terminar, que é o caso para a função de instrumentação "return ()".
  Embora este seja um aspeto não mandatório, garante maior legibilidade ao código instrumentado e permite que a função de instrumentação de retorno de uma função apareça apenas uma vez por cada função.

```
int oneReturn(int el)
{
  if (el== 5)
    return 0;
  else
    return 1;
  return 5;
}
```

Listing 3.12: Simplificação para apenas um ponto de retorno.

A função apresentada no *listing* 3.12 tem três pontos de retorno distintos.
Como se pode verificar no *listing* 3.13, é criada uma variável de retorno que dependo da condição, esse valor é atualizado e posteriormente são introduzidas instruções *goto* para o ponto de retorno.

```
int oneReturn(int el)
{
  int __retres;
  if (el == 5) {
    __retres = 0;
    goto return_label;
  }
  else {
    __retres = 1;
    goto return_label;
  }
  __retres = 5;
  return_label: /* internal */
  return __retres;
}
```

Listing 3.13: Resultado da simplificação para apenas um ponto de retorno.

- Garantir que todas as instruções *if* possuem o respetivo *else*. Esta é a melhor maneira de garantir que quando o código executa um ramo tem sempre uma alternativa, caso contrário, o *FACT* não consegue decidir se é ou não justificável revolver uma restrição para

eventualmente percorrer um caminho que nem existe. Deste modo, torna-se imperativo que todas as instruções *if* tenham o respetivo *else*, ainda que este último se encontre vazio. Este aspeto vai ainda revelar-se útil na execução simbólica para manter uma lista de quais os caminhos que já foram percorridos.
Para demonstrar esta situação é apresentado o *listing* 3.14

```
void addElse(int el)
{
  if (el == 5)
    printf("5");

}
```

Listing 3.14: Adição de um else a uma instrução if.

Nesta situação, como se pode verificar no *listing* 3.15, é simplesmente adicionado uma instrução *else* sem conteúdo.

```
void addElse(int el)
{
  if (el == 5) {
    printf("5");
  }
  else {
    ;
  }
  ;
  return;
}
```

Listing 3.15: Resultado da adição de um else a uma instrução if.

- Transformar expressões condicionais em predicados. A ideia é simplificar expressões do tipo $if(!x)$ para expressões $if(x == 0)$. Este aspeto vai de encontro aos operadores de comparação definidos pela variável *compare_op_t* (seção 3.3.1.1). Esta simplificação é usada para tornar mais fácil o uso das funções de instrumentação. Veja-se o exemplo apresentado no *listing* 3.16.

```
void simplePred(int el)
{
  if (el)
    printf("true");
  else
    printf("false");
}
```

Listing 3.16: Simplificação de expressões condicionais.

O resultado obtido para este exemplo é apresentado no *listing* 3.16.

```
void simplePred(int el)
```

```
{

  if (el != 0) {
    printf("true");
  }
  else {
    printf("false");
  }
  ;
  return;
}
```

Listing 3.17: Resultado da simplificação de expressões condicionais.

- Instrumentar o código com as funções de execução simbólica. Estas funções permitem que seja feita a execução simbólica com a utilização das classes anteriormente apresentadas. Estas funções serão apresentadas com maior detalhe na seção 3.4.1.1;
- Adicionar a função de inicialização do *FACT*. Quando o programa instrumentado é iniciado, torna-se necessário executar algumas etapas de inicialização de variáveis para garantir que o processo de teste executa como previsto.
- Escrever informação auxiliar produzida durante as etapas anteriores nos respetivos ficheiros. Durante as etapas da instrumentação são mantidas algumas informações:
  - O último identificador utilizado numa chamada a uma função de instrumentação;
  - O último identificador utilizado numa chamada a uma instrução;
  - O último identificador atribuído a uma função;
  - O conjunto de ramos percorridos pelo *CIL*.

  As três primeira informações são simplesmente valores inteiros e portanto são guardados como simples inteiros em ficheiros distintos.
  Como se verifica no *listing* 3.18, o conjunto de ramos é guardado de forma diferente.

```
1 0
2 1
24 26
```

Listing 3.18: Formato dos dados do ficheiro “branches”.

O primeiro valor inteiro da primeira linha indica o identificador atribuído à função. O segundo valor indica quantas condições contém essa função. Neste caso a função 1 não tem condições.
Na segunda linha repete-se, a função dois tem uma condição e portanto tem dois ramos. Abaixo de uma nova função existem tantas linhas quantas condições. Neste caso os ramos são os que têm como identificador 24 e 26.

Esta informação irá ser utilizada para saber o total de ramos existentes no programa e eventualmente dar *feedback* ao programador de qual a cobertura de código alcançada no teste *Concolic*.

3.4.1.1 Funções de Instrumentação

As funções de instrumentação são automaticamente introduzidas durante a etapa de instrumentação apresentada na seção 3.4.1. Estas permitem realizar a execução simbólica, analisando as instruções C do código. À medida que o *CIL* percorre o código fonte, são introduzidas as chamadas às funções tendo em conta o código C.

Estas funções encontram-se implementadas na classe *symbolicInterpreter*. Existe um total de onze funções de instrumentação:

- load (id: id_t) (addr: addr_t) (value:value_t) — esta função é utilizada cada vez que uma variável vai ser usada, seja numa condição ou para posterior atribuição, basicamente insere a variável com o endereço *addr* e valor *value* na pilha *(stack_)*;
- store (id: id_t) (addr: addr_t) — armazena uma variável com endereço *addr* que já se encontre na pilha para a memória *(mem_)*.
- clearStack (id: id_t) — elimina o conteúdo da pilha. Esta função é chamada cada vez uma função termina, seja uma função instrumentada, ou uma função do C, por exemplo, "printf", "strcmp", etc. Esta função tem como objetivo garantir que os dados da pilha são removidos após uma função ser chamada, deste modo, garante que não fica informação por processar.
- apply1 (id: id_t) (op: unary_op_t) (value: value_t) — utilizada quando é feita a chamada a operadores unários. Introduz na pilha o resultado da aplicação da operação unária (*op*) ao último elemento que se encontra na pilha. A variável *value* contém o resultado da operação.
- apply2 (id: id_t) (op: binary_op_t) (value: value_t) — utilizada quando é feita a chamada a operadores binários ou de comparação. Introduz na pilha o resultado da aplicação da operação binária (*op*) aos dois últimos elementos que se encontram na pilha. A variável *value* contém o resultado da operação;
- branch (id: id_t) (bid: branch_id_t) (pred_value: bool) — função usada quando surge uma instrução *if*. O predicado é guardado na variável *pred_* e posteriormente, este é inserido juntamente com o identificador do ramo no caminho de execução (variável *path_* da classe *symbolicExecution*);
- call (id: id_t) (fid: function_id_t) — função usada quando surge uma chamada a uma função instrumentada. Assim como na função *branch*, introduz o identificador da função chamada no caminho de execução;
- return (id: id_t) — para indicar que a função vai terminar. É introduzido o identificador de retorno da função no caminho de execução. Ao introduzir os identificadores no caminho percorrido permite obter a cobertura do código atual;
- handleReturn (id: id_t) (value: value_t) — processamento dos valores de retorno de uma função. Se o valor de retorno é igualado a uma variável, esse valor é introduzido na pilha.

Estas funções encontram-se no código C instrumentado mas, precisam de um ponto de entrada, isto é, torna-se necessário criar um ponto em que uma variável da classe *symbolicInterpreter* é instanciada. Para tal existem ainda duas funções que permitem realizar ações quando o programa é iniciado e quando é terminado:

- init () — esta função é a primeira função chamada no código a ser testado. Esta vai ler a lista de *inputs* para as variáveis simbólicas de modo a percorrer diferentes caminhos de execução. É ainda nesta função que a variável do tipo *symbolicInterpreter* é instanciada e posteriormente são introduzidos esses *inputs* na respetiva variável;
- atExit () — quando o processo de teste é terminado, torna-se necessário fornecer os resultados recolhidos ao *FACT*. Para tal, a informação é escrita num ficheiro que será posteriormente lido e processado no *FACT*.

De seguida encontram-se alguns exemplos com código C e respetivo código instrumentado.
Uma expressão C gera uma série de chamadas às funções *Load* e *Apply* correspondente a posterior avaliação necessária da expressão. Por exemplo a expressão "a*b > 3+c" iria gerar a instrumentação apresentada no *listing* 3.19.

```
Load(&a, a)
Load(&b, b)
ApplyBinOp(MULTIPLY, a*b)
Load(0, 3)
Load(&c, c)
ApplyBinOp(ADD, 3+c)
ApplyBinOp(GREATER_THAN, a*b > 3+c)
```

Listing 3.19: Instrumentação da expressão "a*b > 3+c".

Cada chamada a um *Load* ou *Apply* inclui um valor concreto, seja apenas carregado ou calculado e depois carregado. As constantes são consideradas como tendo endereço "0".

Ao entrar num bloco de um *if* ou *else* é gerada uma chamada à função *Branch* para indicar qual o ramo executado. Por exemplo, a instrução "if(a*b > 3+c)" gera uma série de chamadas ás funções *Load* e *Apply*, como apresentado anteriormente, mais uma chamada para cada ramo como apresentado no *listing* 3.20.

```
Branch(true_id,  1)
Branch(false_id, 0)
```

Listing 3.20: Instrumentação da expressão de uma instrução *if*.

Quando é feita uma atribução, é gerada uma chamada à função *Store*, a indicar que o valor deve sair da pilha e ser armazenado num determinado endereço.
Tendo em consideração a instrução "a = 3 + b", o resultado é apresentado no *listing* 3.21.

```
Load(0, 3)
Load(&b, b)
ApplyBinOp(ADD, 3+b)
Store(&a)
```

Listing 3.21: Instrumentação de uma instrução de atribuição.

Para todas as chamadas de funções, os argumentos são colocados na pilha. Se o corpo da função se encontrar instrumentado, os valores destes argumentos são armazenados na respetiva estrutura de execução simbólica.
No corpo da função chamada, a instrução "return e" gera uma expressão de instrumentação para a expressão "e", seguida de uma chamada à função *Return*. Uma instrução *return void*

apenas gera a chamada à função *Return*.
Se o valor de retorno é atribuído a uma variável, por exemplo "z = max(a,7)", então são geradas as chamadas apresentadas no *listing* 3.22. Caso o valor de retorno seja ignorado, por exemplo "max(a, 7);" apenas é gerada uma chamada à função *ClearStack*.

```
HandleReturn([valor concreto devolvido])
Store(&z)
```

Listing 3.22: Instrumentação do retorno de uma função com argumento.

No *listing* 3.23 encontra-se a instrumentação completa do seguinte exemplo: "add(x, y) { return x+y; }" e o respetivo resultado armazenado numa variável "z".

```
Load(&a, a)
Load(0, 7)
Call(add)
Store(&y)
Store(&x)
Load(&x, x)
Load(&y, y)
ApplyBinOp(ADD, x+y)
Return()
HandleReturn(z)
Store(&z)
```

Listing 3.23: Instrumentação de uma função que soma dois argumentos.

No apêndice B encontra-se um programa em C e respetiva instrumentação final que reflete todas as etapas e detalhes descritos no âmbito do processo de instrumentação.

3.4.1.2 Funções de Declaração de Variáveis

Existem ainda as funções que o programador usa para indicar quais as variáveis a tratar simbolicamente. Para esse efeito são disponibilizadas seis funções:

- UChar — indica que a variável do tipo *unsigned char* é simbólica;
- UShort — indica que a variável do tipo *unsigned short* é simbólica;
- UInt — indica que a variável do tipo *unsigned int* é simbólica;
- Char — indica que a variável do tipo *char* é simbólica;
- Short — indica que a variável do tipo *short* é simbólica;
- Int — indica que a variável do tipo *int* é simbólica;

Estas funções de instrumentação têm de ser manualmente introduzidas pelo utilizador para o *FACT* saber como as processar.

3.4.2 Ligação Entre C e OCaml

Dado que o *FACT* foi implementado em *OCaml*, assim como as funções de instrumentação anteriormente apresentadas, tornou-se necessário conseguir fazer a chamada das funções de instrumentação a partir do código C [cam].

Para fazer esta comunicação foi criada uma biblioteca com uma *interface* para cada uma das funções de instrumentação. Este ficheiros encontram-se na diretoria "c_files" apresentada na figura 3.2.

Esta biblioteca contém dezoito funções definidas que vão chamar as funções referidas na seção 3.4.1.1. Para haver uma correta comunicação entre as funções C e *OCaml* são necessários alguns cuidados:

1. Inicializar o *OCaml runtime*;
2. Verificar se a função pretendida se encontra definida no ficheiro em *OCaml*;
3. Se necessário processar os argumentos em C de modo a que estes possuam equivalência nos tipos do *OCaml*;
4. Por último resta chamar a função pretendida com os argumentos necessários.

Finda as etapas de programação é ainda necessário ter em consideração alguns detalhes no processo de compilação.
Nas seguintes sub-seções são apresentadas cada uma das etapas individualmente.

3.4.2.1 Inicialização OCaml

Para chamar as funções é necessário inicializar o *OCaml runtime*. Para tal é disponibilizada a função *caml_startup(char **)*. O parâmetro desta função serve apenas para inicializar o *Sys.argv* caso seja necessário passar argumentos para o *OCaml*.
No exemplo apresentado no *listing* 3.24 pode-se verificar as etapas necessárias para a inicialização. Dado que não são necessários quaisquer parâmetros adicionais é apenas instanciado o argumento.

```
...
char *argv[2];
argv[0] = malloc(1);
argv[0][0] = 0;
argv[1] = NULL;
caml_startup(argv);
 ...
```

Listing 3.24: Inicialização do *OCaml runtime*.

3.4.2.2 Procurar Definição da Função

Para conseguir encontrar uma função definida em *OCaml* através do C, é necessário definir inicialmente a função num programa *OCaml* e posteriormente registá-la como uma *callback*.
As funções que servem de interface para o *OCaml* e posteriormente chamam a variável do tipo *symbolicInterpreter* encontram-se implementadas no ficheiro *lib.ml* que se encontra na diretoria "lib". Para definir cada uma destas funções como um *callback* basta fazer uso da função *register* disponível no módulo *callback* do *OCaml*. No exemplo apresentado no *listing* 3.25 é criada uma função *OCaml* e registada como uma *callback* que faz uso da função *load*.

```
let si = new symbolicInterpreter

let Load (id: int) (addr: int64) (value: int64) : unit =
        si#load id addr value

let () =
        Callback.register "Load" Load
```

Listing 3.25: Definição de uma função no OCaml.

Depois de criadas e registadas todas as funções e de inicializado o *OCaml runtime* é necessário encontrar a função em *OCaml* através código C. No exemplo apresentado no *listing* 3.26 é feita a procura da função *load* previamente registada no *OCaml*:

```
...
static value *load_closure = NULL;
if (load_closure == NULL) {
        load_closure = caml_named_value("Load");
        }
...
```

Listing 3.26: Pesquisa da função declarada no OCaml.

Depois de encontrado o apontador para a função pretendida, apenas é necessário processar os argumentos, se estes existirem, e finalmente chamar a função.

3.4.2.3 Processar Argumentos

Caso a função necessite de argumentos, torna-se necessário algum cuidado pois é preciso manter a compatibilidade entre os tipos do *OCaml* e do C.
Para tal são disponibilizadas algumas funções que permitem realizar este tipo de operações. Todos os tipos que serão necessários podem ser transformados em inteiros, de 32 ou 64 *bits*. Para este efeito são utilizadas duas funções:

- caml_copy_int64 — para transformar e copiar um valor para um *int64* em *OCaml*;
- Val_int — para transformar e copiar um valor para um *int* ou *char* em *OCaml*.

No exemplo apresentado no *listing* 3.27, é processado como parâmetro um inteiro de 32 *bits* (variável *id*) e dois inteiros de 64 *bits* (variáveis *addr* e *val*) para a variável "args".

```
...
value args[3];
args[0] = Val_int(id);
args[1] = caml_copy_int64(addr);
args[2] = caml_copy_int64(val);
```

Listing 3.27: Processamento dos argumentos para o Ocaml.

3.4.2.4 Chamar a Função

Finalmente pode ser feita a chamada à função pretendida, para tal basta usar a função $caml\_callbackN(*load\_closure, n, args);$ em que o primeiro argumento é o apontador para a função, o segundo o número de parâmetros dos argumentos e a terceira variável é a lista de argumentos. O resultado final da função *load* encontra-se no *listing* 3.28.

```
void __Load(int id, unsigned long int addr, long long int val) {
        static value *load_closure = NULL;
        value args[3];
        if (load_closure == NULL) {
                load_closure = caml_named_value("Load");
        }
        args[0] = Val_int(id);
        args[1] = caml_copy_int64(addr);
        args[2] = caml_copy_int64(val);
        caml_callbackN(*load_closure, 3, args);
        return;
}
```

Listing 3.28: Definição da *interface* para a função "__Load".

Como já referido, todas as funções de instrumentação encontram-se definidas na classe *symbolicInterpreter* e as respetiva *interface* em C no ficheiro *fact.c* e em *OCaml* no ficheiro *lib.ml*.

3.4.2.5 Compilação

Findo o processo de programação é ainda necessário compilar corretamente os ficheiros de modo a que estes consigam comunicar entre si.

O processo de compilação é sempre um dos grandes desafios, não só nesta etapa, mas também na compilação do *FACT* como um *plug-in*.
Inicialmente compila-se a *interface* C com o *GNU Compiler Collection (GCC)* [gcc]. Posteriormente compilam-se todos os ficheiros necessários à interface *OCaml* com a opção *-output-obj* que permite gerar um objeto para ser posteriormente executado.
Deste modo já se encontram compilados todos os ficheiros necessários para fazer uso do *FACT*, é apenas necessário a vinculação (*linking*) entre as interfaces e o programa a testar. Esta etapa é automaticamente feita quando é executado o teste *Concolic* através do *Frama-C*.

3.4.3 Execução do FACT

Até agora apenas foram abordados os processos de preparação do código para a realização do teste simbólico em que é fornecido um conjunto de *inputs* ás variáveis simbólicas e é recolhida informação relativa à execução do programa. Depois de recolhida a informação, esta necessita de ser processada e eventualmente voltar a executar a aplicação a testar com novos *inputs* até ser atingido um determinado critério de paragem.

Esta seção foca-se essencialmente em como é iniciado e realizado o processo de execução de modo a concretizar o teste *Concolic*. Existem dois ficheiros que refletem estas etapas. Ambos encontram-se na diretoria "run_crest" e são:

- concolic_search.ml — este ficheiro contempla todas as ferramentas necessárias para a implementação das estratégias de pesquisa assim como a implementação do *BDFS*;
- run_fact.ml — este ficheiro é o ponto de entrada para o *FACT*, e contempla algumas etapas para este funcionar corretamente como um *plug-in*, estas serão discutidas na seção 3.4.6. Aqui também é inicializada a estratégia de pesquisa passada como argumento aquando da execução do programa.

Quando o *FACT* é executado torna-se imperativo realizar a instrumentação do código e respetiva vinculação entre as *interfaces* e a aplicação a ser testada.

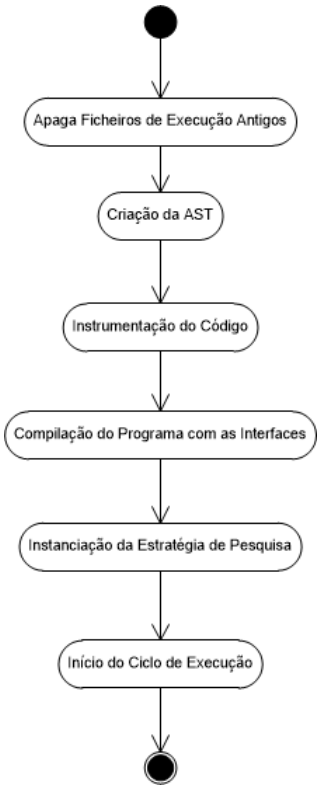

Figura 3.4: Fluxo de Execução do FACT.

Como se pode verificar na figura 3.4, inicialmente são apagados dados que possam ser de execuções anteriores. Depois é criada a *Abstract Syntax Tree (AST)*, funcionalidade automaticamente disponibilizada pelo *CIL*, e fazendo uso desta, o código C é instrumentado como apresentado na seção 3.4.1.
Deste modo o código instrumentado já tem capacidade de realizar uma execução simbólica e necessita apenas de ser compilado com a *interface*. Para compilar é ainda necessário a inclusão da biblioteca do *OCaml runtime* para tornar possível a execução do objeto resultante da compilação da *interface OCaml*.
Se todas as etapas forem corretamente executadas, é verificada qual a estratégia de pesquisa escolhida pelo utilizador, e é então instanciado um objeto dessa estratégia de modo a dar inicio à execução do teste *Concolic*.

3.4.4 Fluxo Genérico de Execução

Embora o fluxo de execução do teste *Concolic* dependa da estratégia de pesquisa utilizada ainda existem algumas etapas que são sempre realizadas:

- A primeira execução é normalmente com *inputs* gerados aleatoriamente;
- Os *inputs* a serem passados para o programa a testar são escritos num ficheiro *inputs*. Este ficheiro será lido na função de inicialização ("init()").
- O programa instrumentado é executado;
- Finda a execução do programa, os resultados são escritos no ficheiro "szd_execution" para serem processados pelo programa principal;
- O programa lê os dados gerados a partir da execução e processa-os de modo a gerar novos *inputs* que permitam exercitar diferentes caminhos de execução.

O modo como é feito a exploração dos novos caminhos vai-se refletir pela estratégia de pesquisa utilizada.

3.4.5 Ciclo de Execução BDFS

O fluxo de exploração dos possíveis caminhos de execução pode ter algumas variações dependendo da estratégia de pesquisa utilizada. Dado que apenas foi implementada a estratégia *BDFS* esta é brevemente discutida nesta seção:

- Inicialmente é criada uma nova instância da estratégia de pesquisa escolhida pelo utilizador;
- É feita a contabilização dos ramos e funções do programa a partir dos ficheiros gerados na etapa da instrumentação;
- De seguida é então criada uma nova instância da classe *SymbolicExecution* que servirá para processar e guardar a informação das execuções do código a testar;
- É feita uma execução inicial com a lista de *inputs* aleatórios;
- Finda a execução do programa é feito a atualização da cobertura dos ramos;
- Se tudo correu como planeado são repetidas as etapas até alcançar uma determinada profundidade ou até não haverem restrições a resolver.
- Caso alguma coisa tenha corrido mal, é ignorado o ramo e é feita uma chamada recursiva aumentando também a profundidade.

Quando o ciclo terminar são apresentados os resultados da cobertura alcançados.

3.4.6 Incorporação no Frama-C

Uma das principais contribuições é a possibilidade de realizar teste *Concolic* no *Frama-C* e portanto torna-se necessário integrar o *FACT* no mesmo. Para tal são disponibilizados pelo *Frama-C* alguns módulos que permitem uma fácil integração do *FACT* como um *plug-in* [fraa].

3.4.6.1 Desenvolvimento

Para integrar um programa com o *Frama-C* é necessário criar um ponto de entrada para o programa. Neste caso foi criada a função "run () : unit" que contempla as etapas descritas na seção 3.4.3. O *Frama-C* disponibiliza uma *API* que permite que este seja facilmente expandido. Para tal efeito é disponibilizada a função "Db.Main.extend", tornando possível executar o *FACT* como um *script* externo ao *Frama-C*.

Dado que o objetivo é integrar completamente o *FACT* com o *Frama-C* como um *plug-in*, torna-se necessário registar o *script* como um *plug-in*. Para esse efeito é disponibilizado o *functor Plugin.Register* em que dados vários parâmetros, permite registar o *plug-in*. Deste modo, quando o comando "frama-c -help" é executado o *FACT*, se devidamente instalado, já deve aparecer na lista dos *plug-ins* disponíveis.

Dado que o *FACT* aceita como possíveis argumentos a estratégia de pesquisa, o número de iterações e também argumentos extra para a instrução de execução do programa a testar, uma *string*, um valor inteiro e outra *string* respetivamente, ainda é necessário fazer uso do *functor* "Self.String" e "Self.Int" para aceitar os respetivos parâmetros. Estes *functores* têm como argumentos o nome da opção, o valor por omissão, o nome dado ao argumento e uma mensagem de ajuda que será disponibilizada através do comando "frama-c -crest-help".

Para executar o *plug-in* apenas quando este é requisitado é ainda preciso usar mais um *functor*, "Self.False", de modo a indicar que apenas é executado caso a opção passada no parâmetro *option_name* apareça, sendo neste caso a opção é "-fact". Isto é necessário, caso contrário o *plug-in* seria automaticamente executado cada vez que o *Frama-C* era executado.

Deste modo, é possível obter uma integração total do *FACT* como um *plug-in* para o *Frama-C*. É ainda necessário não esquecer que este *plug-in* tem de ser compilado e instalado.

3.4.6.2 Compilação e Instalação do Plug-in

O *Frama-C* disponibiliza um modelo de *Makefile* que permite compilar *plug-ins*. Dadas as dimensões do *FACT*, de este depender do *Yices* e de se encontrar dividido em várias diretorias e ficheiros é necessário ter algum cuidado na criação do *Makefile*.

Inicialmente é necessário ter em consideração as várias bibliotecas do *Frama-C* e portanto são definidos os caminhos para estas no inicio do ficheiro. Ainda existem vários parâmetros que são necessários configurar para uma correta compilação:

- *PLUGIN_NAME* — indica o nome do *plug-in*;
- *PLUGIN_CMO* — indica os vários ficheiros *CMO* que contemplam o *FACT*;
- *PLUGIN_DOCFLAGS* — possui opções adicionais que possam ser úteis para gerar a documentação automaticamente;
- *PLUGIN_BFLAGS* — opções adicionais para o compilador de *bytecode*;
- *PLUGIN_OFLAGS* — opções adicionais para o compilador nativo;
- *PLUGIN_EXTRA_BYTE* — ficheiros de *bytecode* adicionais para a vinculação;

- *PLUGIN_EXTRA_OPT* — ficheiros de compilação nativos adicionais para a vinculação;

Na variável "PLUGIN_CMO" são incluídos todos os ficheiros apresentados na seção 3.2.
Nas opções adicionais para o compilador (*PLUGIN_OFLAGS* e *PLUGIN_EXTRA_BYTE*) são adicionados também a diretoria do *OcamlYices*, assim como os respetivos ficheiros de implementação do *OcamlYices* nos ficheiros para adicionar na vinculação (*PLUGIN_EXTRA_BYTE* e *PLUGIN_EXTRA_OPT*).

No Apêndice A encontram-se descritas as etapas necessárias para a instalação e utilização do *FACT*.

3.5 Conclusão

Neste capítulo foram apresentados os detalhes e processos de execução necessários que permitem ao *FACT* realizar a execução *Concolic*. Embora se tenha tentado manter o número de dependências ao mínimo, este necessita de várias tecnologias para funcionar corretamente, nomeadamente: o *Frama-C*, como era de esperar, e o *Yices* e *OCamlYices*. Estes dois últimos são facilmente ultrapassáveis fazendo uso do *Why3* que além oferecer novas possibilidades já engloba a possibilidade de aceder a todo um conjunto de *SMT Provers* que podem ser facilmente utilizados através da utilização da *API* deste.

Capítulo 4

Exemplos

4.1 Objetivos

Este capítulo pretende demonstrar o *FACT* em funcionamento e provar que este consegue obter uma cobertura de código superior com geração automática de *inputs*.
Para demonstrar o correto funcionamento deste e visto que os fundamentos do *FACT* baseiam-se no *Crest*, torna-se imperativo que este seja pelo menos capaz de apresentar os mesmos resultados que o *Crest*. Como tal, neste capítulo é apresentada o conjunto de testes do *Crest* e os seus resultados, assim como os resultados que o *FACT* obtém no mesmo conjunto de testes.

Para o desenvolvimento deste projeto foi utilizado o Ubuntu, *Release* 12.04 (precise) *32-bit*, *Kernel Linux 3.5.0-48-generic*. A versão do *Frama-C* é *Fluorine-20130601* e o *Yices* foi usado na versão 1.0.39. A versão do *Crest* usada foi a 0.1.2.

4.2 Conjunto Testes do Crest

O conjunto de testes do *Crest* contempla um total de treze exemplos que visam demonstrar a capacidade da ferramenta nas diversas circunstâncias possíveis. Neste capítulo, são apenas apresentados cinco testes, considerados representativos, e uma pequena comparação entre o *Crest* e o *FACT*.
Estes cinco testes e uma pequena descrição relativa a cada um pode ser encontrado no Apêndice C.

4.2.1 Output de Uma Execução *Concolic*

Para executar cada um destes testes apresentados é necessário executar um conjunto de etapas de instalação que pode ser encontrado com maior detalhe no Apêndice A. Depois de instalado e devidamente configurado, cada um dos testes pode ser executado com o comando de compilação apresentado no *listing* 4.1, ao contrário do *Crest* em que é necessário um comando para instrumentar o código e outro para correr o programa.

```
frama-c -fact -search dfs -cpp-extra-args='-I␣$FACTCPATH' <nomeFicheiro>
```

Listing 4.1: Comando para Executar o FACT.

Quando executado, o *FACT* produz um *output* com o seguinte formato:
Iteration 10 (0.428027): covered 7 branches [1 reach funs, 8 reach branches].
Por ordem, os valores representam:

- A iteração atual;
- O tempo que o programa já se encontra em execução (em segundos);

- O número de ramos já alcançados;
- O total de funções que o programa deve conseguir alcançar;
- O total de ramos que o programa contém.

4.2.2 Primeiro Teste

O primeiro teste, apresentado na seção C.1, tem como principal propósito pesquisar por um determinado elemento. Como se pode verificar, este encontra-se instrumentado com a função "unsigned_char".
Depois de instrumentado, o programa fica com um total de seis ramos: dois com o critério de paragem do primeiro ciclo *for*, dois para o critério de paragem para o segundo ciclo *for* e dois para o *if*.

Os *outputs* produzidos pelo *Crest* e pelo *FACT* relativos a este teste encontram-se apresentados nas tabelas 4.1 e 4.2 respetivamente.
É facilmente percetível que o *FACT* alcançou a mesma cobertura que o *Crest*, dos seis ramos possíveis todos eles foram executados no mesmo número de iterações.

Crest				
Iteração	**Tempo**	**Ramos Alcançados**	**Total Funções**	**Total Ramos**
0	0s	0	0	0
1	0s	5	1	6
2	0s	6	1	6

Tabela 4.1: Resultado do *Crest* para o primeiro teste.

FACT				
Iteração	**Tempo**	**Ramos Alcançados**	**Total Funções**	**Total Ramos**
0	0s	0	0	0
1	0.028002s	5	1	6
2	0.200013s	6	1	6

Tabela 4.2: Resultados do *FACT* para o primeiro teste.

Este exemplo inicial demonstra que o *FACT* consegue manipular uma simples restrição de modo a encontrar um valor que a torne verdadeira.

4.2.3 Segundo Teste

O segundo teste, apresentado na seção C.2, tem como propósito garantir que uma variável criada no "main", instrumentada com a função *int*, e posteriormente submetida a um conjunto de restrições em diversas funções também tem solução.
Depois de instrumentado, o programa fica com um total de doze ramos: dois para cada instrução *if* que se encontra no código. Dado que nesta situação não existem ciclos nem instruções *if* sem o respetivo *else*, depois do processo de instrumentação não são adicionados novos ramos.

Os resultados obtidos pelo *Crest* e pelo *FACT* relativos a este teste são, novamente, muito semelhantes pois existe apenas uma pequena diferença de milissegundos na execução do *FACT*, sendo que todas as restantes linhas se mantêm. Estes encontram-se apresentados na tabela

4.3.
Novamente o *FACT* alcançou a mesma cobertura que o *Crest*, dos doze ramos possíveis todos eles foram executados no mesmo número de iterações.

Iteração	Tempo	Ramos Alcançados	Total Funções	Total Ramos
0	0s	0	0	0
1	0.008001s	6	5	12
2	0.036002s	7	5	12
3	0.052003s	8	5	12
4	0.068004s	9	5	12
5	0.084005s	10	5	12
6	0.104007s	11	5	12
7	0.120008s	12	5	12

Tabela 4.3: Resultados do *FACT* para o segundo teste.

Neste exemplo, pode-se verificar que existem diversas chamadas à função "printf" no código. Como cada um destes ramos foi efetivamente executado, então é possível visualizar os *outputs* produzidos. Para o *FACT* foram obtidos os *outputs* apresentados na *figura 4.1* e para o *Crest*, os resultados, encontram-se na *figura 4.2*.
Pode-se verificar que embora ambos tenham alcançado a mesma cobertura, os caminhos de execução diferiram desde o início até ao fim. Tal ocorrência deve-se aos *inputs* aleatoriamente gerados para a primeira execução.

Com este exemplo, fica demonstrado que o *FACT* consegue recolher e resolver restrições de um conjunto de funções.

4.2.4 Terceiro Teste

O terceiro teste, apresentado na seção C.3, tem como principal propósito garantir que o *FACT* tem capacidade de processar funções matemáticas complexas e relacioná-las com várias variáveis simbólicas. Depois de instrumentado, o programa fica com um total de dois ramos, pois apenas existe uma condição *if* no programa.

Os resultados obtidos pelo *Crest* e pelo *FACT* relativos a este teste são, novamente, muito semelhantes. Estes encontram-se apresentados na tabela 4.4. Novamente o *FACT* alcançou a mesma cobertura que o *Crest*, dos dois ramos possíveis, no mesmo número de iterações.

Iteração	Tempo	Ramos Alcançados	Total Funções	Total Ramos
0	0s	0	0	0
1	0.020001s	1	1	2
2	0.040002s	2	1	2

Tabela 4.4: Resultados do *FACT* para o terceiro teste.

4.2.5 Quarto Teste

O quarto teste, apresentado na seção C.4 tem como principal propósito garantir que o *FACT* tem capacidade de resolver restrições com operações de *shift*. Depois de instrumentado, o programa fica com um total de quatro ramos, dois por cada instrução *if* no programa.

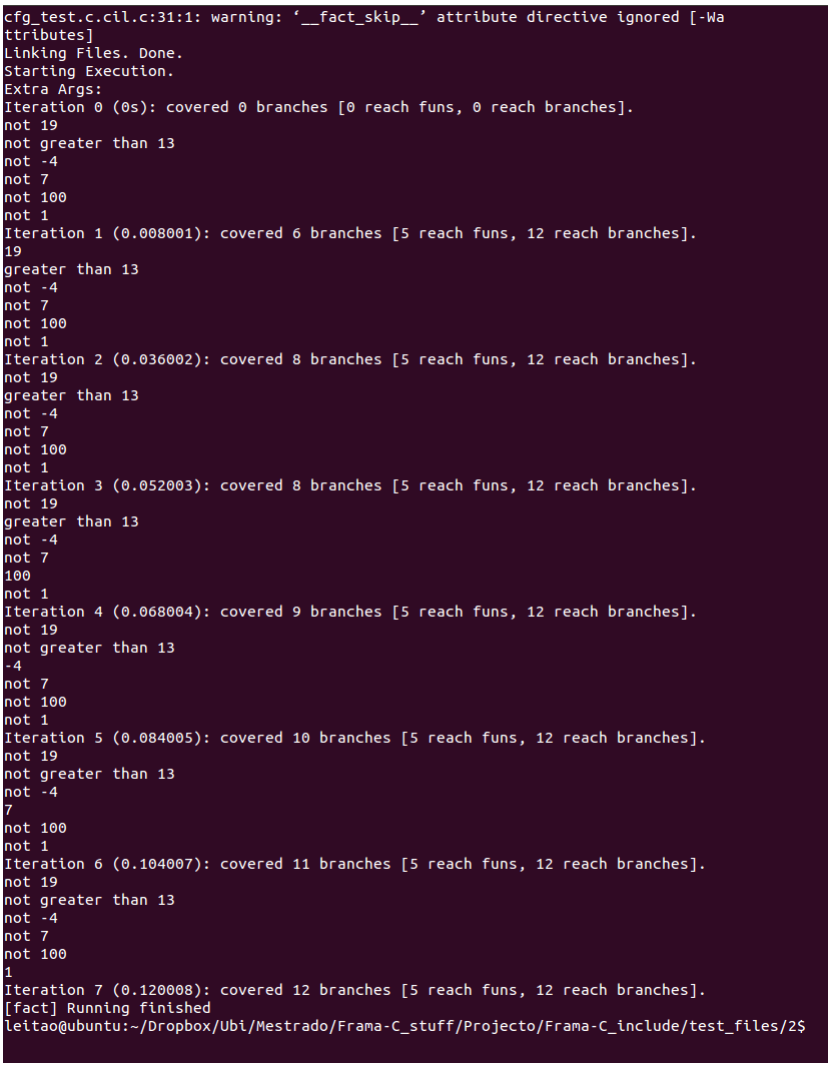

Figura 4.1: Outputs do segundo teste para o *FACT*.

Os resultados obtidos pelo *Crest* e pelo *FACT* relativos a este teste são, novamente, muito semelhantes. Estes encontram-se apresentados na tabela 4.5. Como era de esperar o *FACT* alcançou a mesma cobertura que o *Crest*, dos quatro ramos possíveis todos eles foram executados no mesmo número de iterações.

Iteração	Tempo	Ramos Alcançados	Total Funções	Total Ramos
0	0s	0	0	0
1	0.020002s	1	1	4
2	0.040003s	3	1	4
3	0.056004s	4	1	4

Tabela 4.5: Resultados do *FACT* para o quarto teste.

4.2.6 Quinto Teste

O quinto teste, apresentado na seção C.5, visa apresentar que o *FACT* não suporta índices simbólicos para resolver restrições. Depois de instrumentado, o programa fica com um total de dezoito ramos: quatro para o primeiro *if*, mais quatro para o segundo, dois por cada instrução

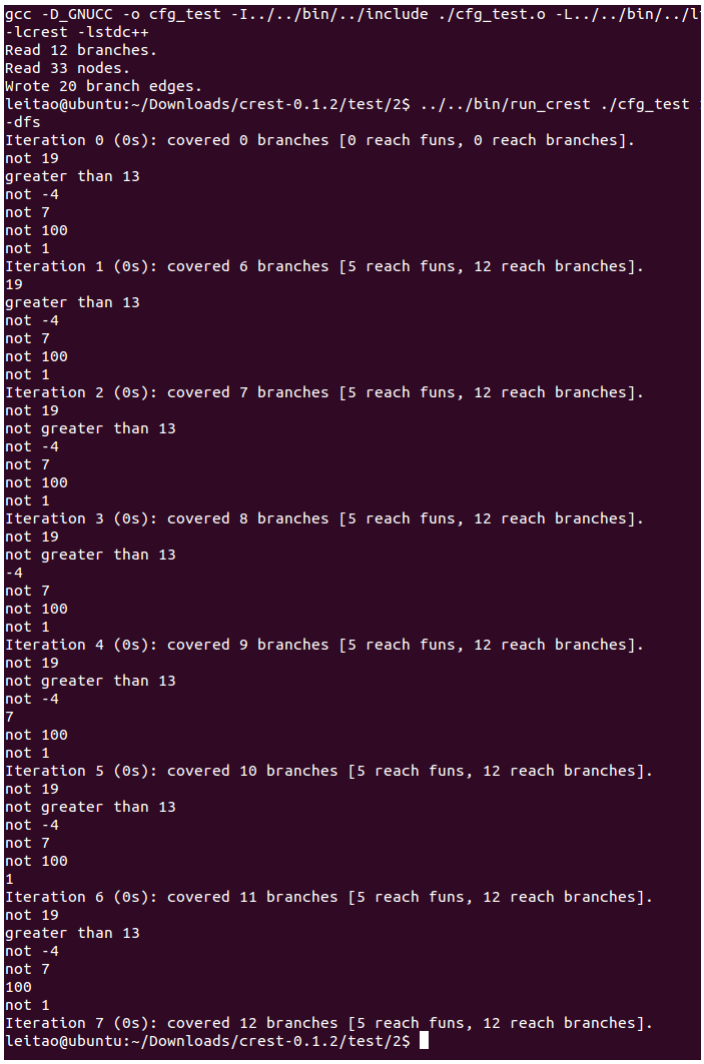

Figura 4.2: Outputs do segundo teste para o *Crest*.

for e dois para as comparações finais. Estas duas últimas comparações espera-se que não sejam atingidas.

Como era espetável, existem dois ramos que não foram atingidos, muito provavelmente as duas últimas condições nunca são verdadeiras.

Iteração	Tempo	Ramos Alcançados	Total Funções	Total Ramos
0	0s	0	0	0
1	0.032002s	2	1	18
2	0.052003s	3	1	18
3	0.064004s	5	1	18
4	0.092006s	15	1	18
5	0.108007s	16	1	18

Tabela 4.6: Resultados do *FACT* para o quinto teste.

Este conjunto de testes serviu como referência para assegurar a qualidade do *FACT*. Embora o conjunto de testes do *Crest* seja constituído por treze testes, não foram todos apresentados aqui. Assim como estes aqui apresentados, em todos eles, o *FACT*, igualou os seus resultados aos produzidos pelo *Cresst*.

4.3 Teste de Dimensões Representativas

O conjunto de testes disponibilizado pelo *Crest* permite ver exatamente o desempenho e capacidades da ferramenta em execução, mas são testes de complexidade relativamente baixa e portanto tentou-se apresentar nesta seção um teste cuja complexidade fosse superior.

A ideia inicial era usar a mesma aplicação utilizada para testar o *Crest*, o *grep*. Na execução do *Crest* como ferramenta de teste *Concolic*, são sempre necessários dois comandos independentes: a etapa de instrumentação que, neste caso, foi implementada como uma extensão ao *CIL* e a etapa da execução do teste.
No *FACT* simplificou-se este processo, sendo apenas necessário executar um comando que posteriormente irá instrumentar e executar automaticamente o código a testar através do *Frama-C*.
Numa fase final desta dissertação, esta caraterística revelou ser uma desvantagem no teste de aplicações mais complexas. Normalmente uma aplicação com um número de linhas de código elevado encontra-se dividida em vários ficheiros e respetivo *Makefile* para a compilação da mesma. Ao realizar a etapa de instrumentação e compilação com o *Frama-C*, apenas é possível processar um ficheiro de cada vez. Visto que normalmente estes têm dependências entre si, torna-se impossível instrumentar um programa com vários ficheiros.
O *Crest*, ao utilizar o *CIL* como uma extensão para instrumentar o código, consegue ultrapassar este problema, pois em vez de utilizar o *gcc* ou outro compilador para compilar o programa, com o *Makefile*, usa o *CIL* com a *flag* correta para instrumentar o código. Ainda assim surgem alguns problemas, pois o *CIL* nem sempre suporta caraterísticas ou funcionalidades extra que determinados programas utilizam.
Deste modo não foi possível utilizar, na versão atual do *FACT*, o *grep* como aplicação de teste. Dado que o *CIL* tem a capacidade de juntar uma aplicação C com múltiplos ficheiros num só ficheiro fonte, ainda se tentou esta abordagem, mas sem sucesso pois havia funções que o *CIL* não conseguia reconhecer. Curiosamente, a versão do *grep* disponibilizada juntamente com o *Crest* também não executava devido a um conflito com a função “printf”.

Tendo em consideração que de momento não poderiam ser utilizados programas com vários ficheiros, escolheu-se uma aplicação que, ainda assim, fosse mais complexa.
O programa escolhido é resultante de um projeto da Licenciatura em Engenharia Informática da Unidade Curricular Estruturas de Dados. O programa consiste em gerir um sistema de portagens que deve disponibilizar as seguintes funcionalidades:

- Carregar o conteúdo dos ficheiros para estruturas de dados apropriadas;
- Registar donos, veículos e passagens;
- Listar todos os dados armazenados em memória;
- Listar ordenadamente o nome e o número de contribuinte dos condutores;
- Listar ordenadamente a matricula, marca e modelo dos veículos;
- Listar os veículos que circularam num determinado período;
- Apresentar um ranking de circulação de cada veículo;
- Listar os veículos que cometeram uma infração num determinado período;
- Apresentar um ranking de infrações de cada veículo;

- Apresentar a marca do carro e o dono que circula a maior velocidade média;
- Apresentar a velocidade média dos condutores com um determinado código postal.

O programa encontra-se dividido em vários ficheiros e estruturas de dados para cada entidade. É relevante referir que este usa memória dinâmica, árvores balanceadas e listas ligadas para armazenar determinadas entidades. Dado que os ficheiros deste programa encontra-se dividido em bibliotecas, a vinculação dos ficheiros é feita automaticamente pelo compilador.

Toda a informação que suporta a aplicação é fornecida por diferentes ficheiros, com um formato específico. É esse formato que a aplicação espera encontrar para conseguir obter uma leitura feita com sucesso. No entanto, tendo em conta que os valores normalmente problemáticos são os que dependem do utilizador, espera-se que estes valores possam ser considerados simbólicos. Com este intuito, o programa foi ligeiramente alterado para não abrir um ficheiro, mas sim, fazer a leitura a partir de valores declarados como simbólicos. No *listing* 4.2, encontra-se um pequeno excerto de código de uma das funções de leitura em que simula quando são encontrados cinco donos de carros cuja informação associada é simbólica.

```
Donos* LerDonos()
{
        Donos *D=NULL;
        FILE *F;
        int totnome,aux=0;
        char c;
        /*F=fopen("donos.txt","r");
        while(!feof(F))*/
        int i;
        for (i = 0; i < 5; i++)
        {
                FACT_int(aux);
                if(aux==0)
                        break;
                D=insertDono(D);
                totnome=1;
                D->numcontri=aux;
                FACT_char(c);
        do{
                        D->nome=(char*)realloc(D->nome,
                                totnome*sizeof(char));
                        D->nome[totnome-1]=c;
                        totnome++;
                        FACT_char(c);
                }while(c!='\t');
                D->nome=(char*)realloc(D->nome,totnome*sizeof(char));
                D->nome[totnome-1]='\0';
                ...
        }
        return(D);
```

Listing 4.2: Exemplo de leitura dos dados simbólicos.

Neste caso mais complexo, o valor a ler é uma *string*, então é percorrida cada posição e indicada que cada uma delas é um caráter simbólico. Caso o valor fosse inteiro bastaria utilizar a função "FACT_int ()".

A aplicação disponibiliza um pequeno menu que permite navegar para outros sub-menus e usufruir das capacidades desta. Como muitos dos campos das estruturas são *strings*, então os algoritmos de ordenação, *rankings* e infrações necessitam de comparar *strings*. Tornando frequente o uso de funções da biblioteca "string.h", tais como "strcmp", e "strlen". Como já referido, estas funções não se encontram instrumentadas e portanto torna-se complicado percorrer os dois ramos onde estas são alvo de comparação pois, com baixa probabilidade é que serão geradas duas *strings* iguais.

Tendo em conta este problema, inicialmente foi feito um teste em que apenas são executadas as funções que permitem preencher os dados e, foi também dada a possibilidade de correr os diferentes menus. Ambas as funcionalidades não usam funções externas. Os resultados desta execução encontram-se na figura 4.3 em que a barra verde corresponde ao total de ramos do programa e a barra vermelha corresponde aos ramos que foram executados. Pode-se ainda verificar no eixo das abcissas o número de iterações necessário para obter determinada cobertura de código. Como esperado, os resultados são extremamente satisfatórios, no total de 132 ramos possíveis, 131 deles foram executados. O tempo de execução associado a este teste é de aproximadamente dois minutos.

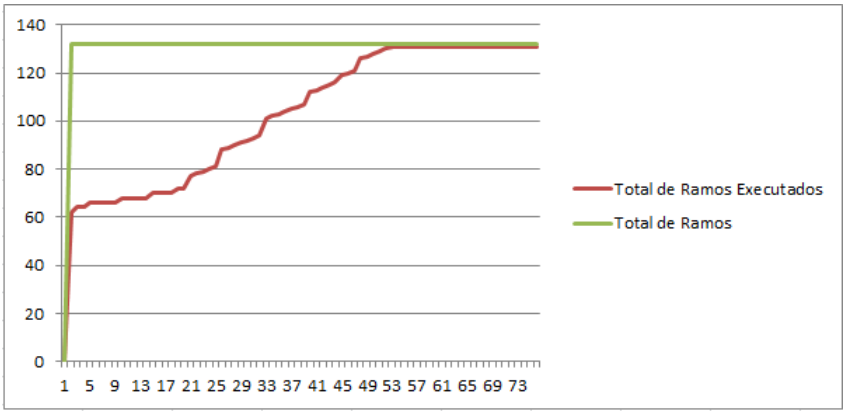

Figura 4.3: Resultados de cobertura obtidos para a leitura de ficheiro e navegação entre menus.

De modo a aumentar a complexidade da execução foram adicionadas as chamadas às funções de impressão que, por sua vez também não fazem uso de nenhuma função de uma biblioteca externa. Como apresentado na figura 4.4 os resultados obtidos também correspondem às expetativas, num total de 142 ramos possíveis, 140 deles foram executados. O tempo de execução associado a este teste é também de aproximadamente dois minutos.
Um aspeto interessante que se destaca na execução de uma aplicação na realização de teste *Concolic*, é que torna-se possível presenciar as diferentes etapas e menus que são percorridos tendo em conta a execução do *FACT*. Na figura 4.5 verifica-se que inicialmente os *inputs* são gerados para cada entidade e depois de apresentado o menu, o *FACT* forneceu um *input* que permite navegar para o menu de registos e, posteriormente registar um novo dono para um

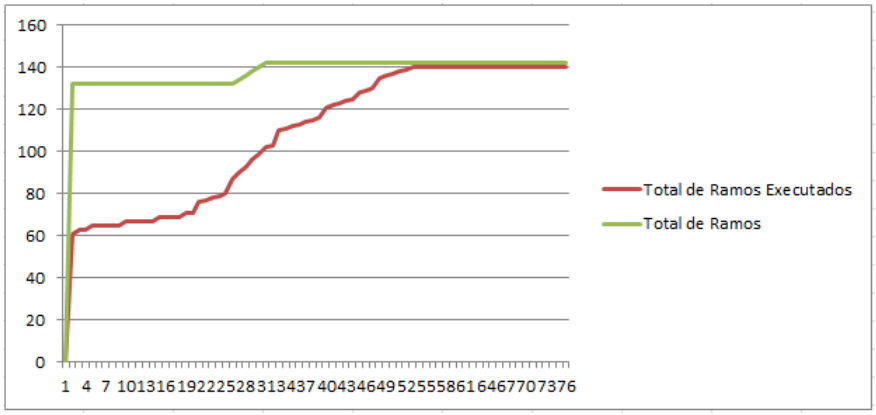

Figura 4.4: Resultados de cobertura obtidos para a leitura de ficheiro, navegação entre menus e funções de impressão.

veiculo. De seguida são pedidos, e automaticamente preenchidos, os dados para esse novo utilizador. Essa execução termina e outra é iniciada.

```
Iteration 20 (57.851616): covered 76 branches [12 reach funs, 162 reach branches].
A ler ficheiro Passagens.txt
A ler ficheiro Sensores.txt
A ler ficheiro Carros.txt
A ler ficheiro Donos.txt
A ler ficheiro Distancias.txt

*******************************
 MENU
*******************************
1:      Registos;
2:      Listagem dos Ficheiros;
3:      Ordenar;
4:      Ranking's;
5:      Infracoes;
6:      Velocidades Medias;
0:      Sair;
*******************************

*******************************
 MENU REGISTOS
*******************************
1:      Registar Dono;
2:      Registar Veiculo;
3:      Registar Passagem;
0:      Voltar atr\E1s;
*******************************
Introduza o numero de contribuinte do novo dono.
Introduza o nome do novo dono.
Introduza o codigo postal do novo dono. No formato: xxxxx-xxx
Iteration 21 (58.007626): covered 86 branches [13 reach funs, 174 reach branches].
```

Figura 4.5: Exemplo de execução do teste *Concolic*.

De modo a testar o comportamento em funções cujo conteúdo usa funções de bibliotecas externas, foram adicionadas as funções de calculo de infrações. Depois de aumentado o número de iterações para 150, o resultado obtido encontra-se apresentado na figura 4.6. O resultado era relativamente promissor até aproximadamente à iteração 50, mas ao ser descoberto um novo ramo e consequentemente mais funções, que possivelmente fazem uso das funções externas, o número de ramos aumentou. Aproximadamente 157 ramos num total de 216 foram cobertos, evidenciado o resultado esperado quado são utilizadas funções externas. O tempo de execução associado a este teste é, assim como nos anteriores, de aproximadamente dois minutos.

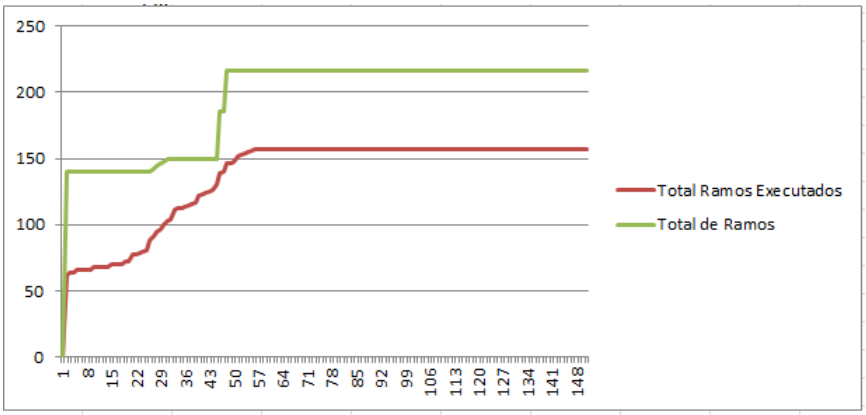

Figura 4.6: Resultados de cobertura obtidos para a leitura de ficheiro, navegação entre menus, funções de impressão e infrações.

Mais por uma questão de curiosidade foi ainda testado o programa completo, contendo aproximadamente 1700 linhas de código. O programa final instrumentado ficou com aproximadamente 7500 linhas de código e, segundo o teste executado, com um total de 452 ramos. Dado que todas as restantes funcionalidades fazem uso das funções da biblioteca "string.h", o resultado obtido não fugiu muitos às expetativas, cada vez que uma nova função era encontrada, surgiam também mais um conjunto de ramos a percorrer. Este pode ser visualizado na figura 4.7. No total foram percorridos 291 dos 452 ramos existentes e, foram também obtidos alguns "segmentation_fault" no menu dos *rankings*, o que indica que está presente um ou mais *bugs* nesta funcionalidade. O tempo de execução associado a este teste é de aproximadamente cinco minutos.
É ainda relevante referir que todo o processo de instrumentação que gera as 7500 linhas de código é praticamente instantâneo (aproximadamente um segundo).

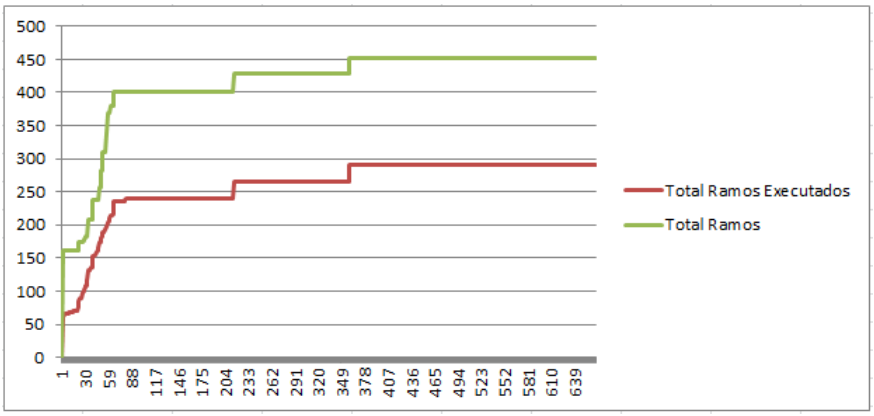

Figura 4.7: Resultados de cobertura obtidos para o programa completo.

4.4 Conclusão

Tendo em conta os resultados apresentados, é possível afirmar que o *FACT* oferece cobertura de código otimizada. De modo a assegurar que no mínimo o *FACT* possui as mesmas caraterísticas do *Crest*, foi executado todo o conjunto de testes proveniente deste sendo que, em todos eles conseguiram-se obter a mesma cobertura de código.

Com o intuito de aplicar o *FACT* a uma aplicação com maior complexidade, foi utilizado um trabalho académico de aproximadamente 1700 linhas de código. Surgiu ainda um problema relativamente óbvio aquando da utilização de funções de bibliotecas externas pois, nesta situação, o *FACT* não consegue saber quais os critérios que condicionam o *output* destas funções. Tirando este detalhe de consideração, o *FACT* conseguiu obter excelentes resultados.

Não foram apresentados testes de ferramentas conhecidas e utilizadas devido ao problema, inesperado, da instrumentação de múltiplos ficheiros. Este será certamente um aspeto a melhorar no futuro.

Capítulo 5

Conclusão

O principal objetivo desta dissertação era obter uma ferramenta que permitisse execução de teste *Concolic*, integrada com o *Frama-C* com o intuito de oferecer um nível de confiança elevado a uma aplicação, tendo como base o total de código executado durante a realização dos testes, isto é, elevada cobertura de código.

O *FACT*, ferramenta desenvolvida para a execução do teste *Concolic*, baseou-se nos princípios e lógica do *Crest*. Foi demonstrado que se conseguiu obter uma eficácia e desempenho semelhante com o *FACT*, estando este totalmente integrado como um *plug-in* do *Frama-C*.

Durante o desenvolvimento desta dissertação, mais concretamente no desenvolvimento do *FACT* surgiram diversos problemas que eventualmente atrasaram o processo de desenvolvimento. Algumas das tecnologias utilizadas não têm propriamente muita documentação, nem uma comunidade muito grande, dificultando, muitas vezes, simples etapas que normalmente são dadas como garantidas, temos por exemplo a compilação e instalação destas ferramentas e das diferentes tecnologias, que nem sempre é tão linear como se espera. Ainda assim, as principais dificuldades foram ultrapassadas, permitindo obter uma ferramenta completamente funcional. No entanto, ainda existem diversos aspetos que podem ser melhorados e novas funcionalidades que irão permitir ao *FACT* obter uma ferramenta com funcionalidades e desempenho superior.

5.1 Reflexão Crítica e Trabalho Futuro

Existe ainda muito trabalho para que o *FACT* consiga obter melhores resultados e seja simples de utilizar para qualquer programador. Neste âmbito, e dado o conhecimento que atualmente detenho na área, é possível afirmar que haveria diversos aspetos a melhorar:

- Uma das melhorias que iriam causar algum impacto e sem dúvida melhorar as capacidades do *FACT* é o uso da *API* do *Why3*. Esta iria permitir usar não só um *SMT Solver*, mas sim todo um conjunto deles que o *Why3* suporta. É importante não esquecer que todos os *SMT Solvers* seriam acessíveis através da *API*, garantindo a coerência da linguagem de *input* para todos eles.
- O *FACT* foi desenvolvido com o foco da linha de comandos mas, o *Frama-C*, também disponibiliza uma *interface* gráfica e o respetivo módulo para expandir essa *interface*. Portanto seria interessante realizar algumas melhorias nesse âmbito, por exemplo, selecionar as variáveis a serem tratadas como simbólicas com apenas um clique no rato.
- O *FACT* apenas contempla uma estratégia de pesquisa. Este aspeto, atualmente, é relativamente simples de contornar visto que todas as funções necessárias para a execução e tratamento dos respetivos resultados já se encontram implementadas, sendo apenas necessário implementar a lógica referente à escolha da ordem de exploração dos caminhos de execução.

- Um dos aspetos que até há algum tempo atrás ainda não tinha sido resolvido, foi a execução de teste *Concolic* em aplicações concorrentes. Entretanto foi apresentada uma solução com o intuito de resolver este problema [FHRV13]. Seria interessante explorar esta solução com o intuito de adicionar esta funcionalidade ao *FACT*.

- Atualmente, o *FACT* contempla um *output* relativamente pobre tendo em conta as suas capacidades dado que, apenas são apresentadas os ramos alcançados e se tudo correu ou não como devia.
  É possível saber quais os ramos alcançados através do ficheiro gerado, "coverage" e os últimos *inputs* da execução, que se encontram no ficheiro "input". No entanto, não é apresentada qualquer tipo de informação acerca dos problemas ou mesmo a execução detalhada. Deste modo, seria interessante gerar um pequeno documento ou fazer uso da *interface* do *Frama-C* aquando do teste de determinada aplicação, em que seja apresentado com exatidão os ramos cobertos e respetivos valores que os permitiram alcançar assim como os respetivos erros e com que valores é que foram atingidos.

- Para ultrapassar o problema da instrumentação em múltiplos ficheiros, podia-se usar uma lógica semelhante ao *Code Digger* [cod], evolução do *Pex* [pex] para as versões mais recentes do *Visual Studio*, em que cada função é testada independentemente das outras, isto é, todos os parâmetros de entrada são considerados variáveis simbólicas. Obviamente que esta abordagem traria outros problemas dada a implementação atual do *FACT*, nomeadamente na instrumentação do código referente à chamada das funções.

- Um outro aspeto ainda interessante para melhorar a performance do *FACT* seria tentar realizar o teste *Concolic* distribuído, seja por várias máquinas ou concorrentemente. Deste modo iria, em principio, melhorar o desempenho para grandes programas. Se fosse adotada a técnica do *Code Digger* a distribuição tornava-se relativamente simples visto que, todas as funções de um programa eram independentes umas das outras.

Atualmente não foram feitas grandes melhorias na área de teste *Concolic*, no âmbito desta dissertação, mas considera-se uma etapa importante de entender e conseguir utilizar os conceitos atualmente existentes para construir uma base robusta e funcional que, eventualmente, permita a continuação e melhoria do *FACT*.

Bibliografia

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *IN TACAS'08: INTERNATIONAL CONFERENCE ON TOOLS AND ALGORITHMS FOR THE CONSTRUCTIONS AND ANALYSIS OF SYSTEMS*, 2008. 10

[BS08] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ASE.2008.69. 10

[cam] Interfacing c with ocaml [online]. Available from: http://caml.inria.fr/pub/docs/manual-ocaml-400/manual033.html [cited 4 Junho 2014]. 33

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=1855741.1855756. 10

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM. Available from: http://doi.acm.org/10.1145/1180405.1180445. 10

[cil] Cil - infrastructure for c program analysis and transformation [online]. Available from: http://www.cs.berkeley.edu/~necula/cil/ [cited 20 Maio 2014]. 13

[cod] Microsoft code digger [online]. Available from: http://visualstudiogallery.msdn.microsoft.com/fb5badda-4ea3-4314-a723-a1975cbdabb4 [cited 15 Junho 2014]. 54

[fac] Fact - frama-c concolic testing [online]. Available from: https://www.dropbox.com/sh/a7jr996go4yzeaw/AAB0qgUI4z_XCJ-v4HU5DtRSa [cited 18 Junho 2014]. 59, 61

[FHRV13] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/SIGSOFT FSE*, pages 37–47. ACM, 2013. Available from: http://dblp.uni-trier.de/db/conf/sigsoft/fse2013.html#FarzanHRV13. 11, 54

[fraa] Frama-c plugin development guide [online]. Available from: http://frama-c.com/download/frama-c-plugin-development-guide.pdf [cited 5 Junho 2014]. 38

[frab] Frama-c software analyzers [online]. Available from: http://frama-c.com/ [cited 20 Maio 2014]. 1

[gcc] Gcc, the gnu compiler collection [online]. Available from: http://gcc.gnu.org/ [cited 19 Junho 2014]. 36

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. Available from: http://doi.acm.org/10.1145/1064978.1065036. 10

[GLM12] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012. Available from: http://doi.acm.org/10.1145/2090147.2094081. 10

[God07] Patrice Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 42(1):47–54, January 2007. Available from: http://doi.acm.org/10.1145/1190215.1190226. 10

[has] The haskell programming language [online]. Available from: http://www.haskell.org/haskellwiki/Haskell [cited 2 Junho 2014]. 10

[JHG] Karthick Jayaraman, David Harvison, and Vijay Ganesh. jfuzz: A concolic whitebox fuzzer for java. 10

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ICSE.2007.41. 16

[MS11] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. ISBN: 978-1-118-03196-4. John Wiley & Sons, 2011. 5, 6

[ocaa] An ocaml binding for yices [online]. Available from: http://github.com/polazarus/ocamlyices [cited 20 Maio 2014]. 13

[ocab] Ocamldoc [online]. Available from: http://docs.camlcity.org/docs/godipkg/3.12/godi-ocaml/man/man1/ocamldoc.1.html [cited 20 Maio 2014]. 61

[Per13] Nuno J. M. Pereira. Testa - geracao automatica de testes unitarios. 2013. 2

[pex] Pex and moles - isolation and white box unit testing for .net [online]. Available from: http://research.microsoft.com/en-us/projects/pex/ [cited 15 Junho 2014]. 54

[qc] Introduction to quickcheck [online]. Available from: http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck1 [cited 2 Junho 2014]. 10

[QR11] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *ESEM*, pages 117–126. IEEE, 2011. Available from: http://dblp.uni-trier.de/db/conf/esem/esem2011.html#QuR11. 10

[SA06] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *In CAV*, pages 419–423. Springer, 2006. 10

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005. Available from: http://doi.acm.org/10.1145/1095430.1081750. 10

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010. 1, 5

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag. Available from: http://dl.acm.org/citation.cfm?id=1792786.1792798. 10

[why] Why3 - where programs meet provers [online]. Available from: http://why3.lri.fr/ [cited 2 Junho 2014]. 14

[Wil10] Nicky Williams. Abstract path testing with pathcrawler. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 35–42, New York, NY, USA, 2010. ACM. Available from: http://doi.acm.org/10.1145/1808266.1808272. 10

[yic] The yices smt solver [online]. Available from: http://yices.csl.sri.com/ [cited 20 Maio 2014]. 13

Apêndice A

Manual de Utilização

Nesta seção pretende-se dar a conhecer ao utilizador quais os requisitos e modo de instalação e utilização do *FACT*. Este encontra-se disponível para *download* na referência [fac].

A.1 Requisitos

Como referido ao longo desta dissertação, o *FACT* tem algumas dependências:

- *Frama-C* — esta é a dependência mais óbvia, pois o programa foi desenvolvido para funcionar como um *plug-in* do *Frama-C*;
- *Yices* e *OCamlYices* — o *Yices* foi o *SMT solver* utilizado e portanto também é necessário estar instalado no computador. Visto que este último não tem ligação direta com o *OCaml*, implica também possuir o *OCamlYices*;
- Outras ferramentas como o *OCaml* e *CIL* assume-se que já estão instaladas pois são necessárias para ter o *Frama-C* a funcionar.

A.2 Compilação e Instalação

Como qualquer outra aplicação, o código precisa de ser compilado e posteriormente instalado no *Frama-C*.
Para facilitar este processo é disponibilizado um *script*, chamado de *make.sh*, que realiza estas tarefas. No entanto é preciso ter o cuidado de alterar a variável *YICES*, definida neste ficheiro, que deve indicar a localização da instalação do *OCamlYices* na máquina a instalar.
O *script* realiza essencialmente três etapas:

- Compila a biblioteca com as funções de instrumentação implementadas em *OCaml*;
- Compila a biblioteca com as funções implementadas em C que permitem fazer a ligação com o *OCaml*;
- Compila o programa como um *plug-in*;
- Se tudo correu bem, o *plug-in* é instalado no *Frama-C*.

A instrução de execução do *script* é a mesma que para qualquer outro programa: "./make.sh".
Por uma questão de facilidade foi também fornecido um *script* para apagar e desinstalar o programa do sistema *(make_clean.sh)*.
Finda a parte de instalação do *script*, é necessário definir duas variáveis de ambiente antes de executar a aplicação:

- "FACTLPATH"— define a localização da biblioteca da aplicação. Exemplo possível do comando:

```
export FACTLPATH=/home/<user>/<dir>/lib/
```

- "FACTCPATH"– define a localização da interface C da aplicação. Exemplo possível do comando:

```
export FACTCPATH=/home/<user>/<dir>/c_files/
```

Estas duas variáveis irão ser utilizadas pelo *Frama-C* quando este necessitar da biblioteca de instrumentação para compilar o programa a testar.

A.3 Utilização

A utilização da ferramenta desenvolvida contempla essencialmente duas etapas:

1. Instrumentação manual do código, como feito no *Crest* – nesta etapa é necessário indicar ao programa quais são as variáveis que devem ser consideradas simbólicas. Para isto são disponibilizadas as seguintes funções:

   - *FACT_int(var)*;
   - *FACT_short(var)*;
   - *FACT_char(var)*;
   - *FACT_unsigned_int(var)*;
   - *FACT_unsigned_short(var)*;
   - *FACT_unsigned_char(var)* ;

   Estas devem ser usadas consoante o tipo da variável em questão. Veja-se o seguinte exemplo em que é declarada uma variável inteira e indicada que a mesma é simbólica:

```
#include <stdio.h>
#include <cFun.h>

int main(void) {
  int x;
  FACT_int(x);

  if (x == 3)
    return 1;
  return 0;
}
```

2. Execução a partir do *Frama-C* – finda a parte de instrumentação dita manual, por parte do programador, pode-se executar imediatamente o programa.
   Assumindo que este já se encontra instalado no *Frama-C*, deve ainda ser possível aceder ao menu de ajuda para visualizar as diferentes opções disponibilizadas com o seguinte comando:

```
frama-c -fact-help
```

Este comando permite visualizar um conjunto de argumentos disponíveis na execução do programa. Como se pode verificar na figura A.1, para executar o *FACT* temos sempre que introduzir a opção "-fact", além desta opção é ainda dada a possibilidade de alterar o número de iterações e a própria estratégia de pesquisa. Atualmente a única estratégia de pesquisa implementada é o *BDFS*.

```
***** LIST OF AVAILABLE OPTIONS:

-extra-args <extra-args>  extra arguments to add to the running code
-fact                     when on (off by default), output a warm welcome message
                          to the user (opposite option is -no-fact)
-num-iters <number>  number of iterations to run the concolic search
                          (default: 100 iterations)
-search <search-type>  type of search used to run the concolic search
                          (default: depth first search (dfs)

*** GETTING INFORMATION

-fact-help                help of plug-in FACT
-fact-h                   alias for option -fact-help
```

Figura A.1: Opções disponíveis para a execução do *FACT*.

Existe ainda uma pequena particularidade necessária na execução da aplicação, é sempre necessário adicionar "-cpp-extra-args=' -I $FACTCPATH"' ao comando de execução. Caso contrário o compilador não irá encontrar a biblioteca com as funções de instrumentação.

Portanto o comando mais simples de execução seria:

```
frama-c -fact -cpp-extra-args=' -I $FACTCPATH' <file_name>.c
```

Posteriormente podem então ser adicionadas as opções apresentadas na figura A.1.

A.4 Documentação

Durante o desenvolvimento do *FACT* existiu o cuidado de comentar, em inglês, todo o código fonte. Deste modo, foi possível gerar com o *OcamlDoc* [ocab] toda a documentação do código na forma de uma página Web. Este encontra-se disponível para *download* na referência [fac].

Apêndice B

Exemplo Completo de Código Instrumentado

Este apêndice tem como propósito apresentar ao leitor um programa implementado em linguagem C que reflete os vários processos da instrumentação do código a testar. O programa apresentado serve apenas para efeitos de exemplificação.
O código C deste programa pode ser encontrado no *listing* B.1 e o respetivo código instrumentado no *listing* B.2.

```
#include<stdio.h>
#include<fact.h>

void simpleMemory(int** m)
{
  printf("Function␣%d\n", m[2][3]);
}

void simpleFor(int dim)
{
  int i;
  for(i = 0; i < dim; i++)
    printf("Iteration:␣%d\n",i);
}

void simpleSwitch(int dim)
{
  switch (dim)
    {
    case 0:
      printf ("Case␣0\n");
      break;
    case 1:
      printf ("Case␣1\n");
      break;
    default:
      printf ("Default␣Case\n");
      break;
    }
}

void simpleDoWhile(int dim)
{
  int i=0;
  do{
```

```
    printf("Iteration: %d\n",i);
    i++;
  }while(i < dim);
}

void simpleWhile(int dim)
{
  int i=0;
  while(i < dim){
    printf("Iteration: %d\n",i);
    i++;
  }
}

int oneReturn(int el)
{
  if (el== 5)
    return 0;
  else
    return 1;
  return 5;
}

void addElse(int el)
{
  if (el == 5)
    printf("5");

}

void simplePred(int el)
{
  if (el)
    printf("true");
  else
    printf("false");
}


int main(){
  int** matrix;
  simpleMemory(matrix);
  int a = oneReturn(5);
  printf("%d\n",a);
  return 0;
}
```

Listing B.1: Exemplo de programa completo em linguagem C.

```
/* Generated by Frama-C */
void __globinit_whole_program(void);

extern void __FactInit(void) __attribute__((__fact_skip__));

extern void __FactHandleReturn(int id, long long val) __attribute__((
__fact_skip__));

extern void __FactReturn(int id) __attribute__((__fact_skip__));

extern void __FactCall(int id, unsigned int fid) __attribute__
((__fact_skip__));

extern void __FactBranch(int id, int bid, unsigned char b)
__attribute__((
__fact_skip__));

extern void __FactApply2(int id, int op, long long val) __attribute__((
__fact_skip__));

extern void __FactApply1(int id, int op, long long val) __attribute__((
__fact_skip__));

extern void __FactClearStack(int id) __attribute__((__fact_skip__));

extern void __FactStore(int id, unsigned long addr)
__attribute__((__fact_skip__));

extern void __FactLoad(int id, unsigned long addr, long long val)
 __attribute__((
__fact_skip__));

extern int printf(char const * __restrict __format , ...);

void simpleMemory(int **m)
{
  int *mem__4;
  int **mem_;
  __FactCall(1,1);
  mem_ = m + 2;
  mem__4 = *mem_ + 3;
  __FactLoad(2,(unsigned long)mem__4,(long long)*mem__4);
  printf("Function␣%d\n",*mem__4);
  __FactClearStack(3);
```

```
  ;
  __FactReturn(4);
  return;
}

void simpleFor(int dim)
{
  int i;
  __FactCall(6,2);
  __FactStore(5,(unsigned long)(& dim));
  __FactLoad(7,(unsigned long)0,(long long)0);
  __FactStore(8,(unsigned long)(& i));
  i = 0;
  while (1) {
    while_0_continue: /* internal */ ;
    __FactLoad(11,(unsigned long)(& i),(long long)i);
    __FactLoad(10,(unsigned long)(& dim),(long long)dim);
    __FactApply2(9,16,(long long)(i < dim));
    if (i < dim) {
      __FactBranch(12,107,1);
      ;
    }
    else {
      __FactBranch(13,108,0);
      goto while_0_break;
    }
    __FactLoad(14,(unsigned long)(& i),(long long)i);
    printf("Iteration:␣%d\n",i);
    __FactClearStack(15);
    __FactLoad(18,(unsigned long)(& i),(long long)i);
    __FactLoad(17,(unsigned long)0,(long long)1);
    __FactApply2(16,0,(long long)(i + 1));
    __FactStore(19,(unsigned long)(& i));
    i ++;
  }
  while_0_break: /* internal */ ;
  ;
  __FactReturn(20);
  return;
}

void simpleSwitch(int dim)
{
  __FactCall(22,3);
  __FactStore(21,(unsigned long)(& dim));
  __FactLoad(25,(unsigned long)(& dim),(long long)dim);
  __FactLoad(24,(unsigned long)0,(long long)0);
```

```
  __FactApply2(23,12,(long long)(dim == 0));
  if (dim == 0) {
    __FactBranch(26,117,1);
    goto switch_0_0;
  }
  else {
    __FactBranch(27,118,0);
    __FactLoad(30,(unsigned long)(& dim),(long long)dim);
    __FactLoad(29,(unsigned long)0,(long long)1);
    __FactApply2(28,12,(long long)(dim == 1));
    if (dim == 1) {
      __FactBranch(31,119,1);
      goto switch_0_1;
    }
    else {
      __FactBranch(32,120,0);
      goto switch_0_default;
      goto switch_0_break;
    }
  }
  switch_0_0: /* internal */ printf("Case␣0\n");
                             __FactClearStack(33);
  goto switch_0_break;
  switch_0_1: /* internal */ printf("Case␣1\n");
                             __FactClearStack(34);
  goto switch_0_break;
  switch_0_default: /* internal */ ;
  printf("Default␣Case\n");
  __FactClearStack(35);
  goto switch_0_break;
  switch_0_break: /* internal */ ;
  ;
  __FactReturn(36);
  return;
}

void simpleDoWhile(int dim)
{
  int i;
  __FactCall(38,4);
  __FactStore(37,(unsigned long)(& dim));
  __FactLoad(39,(unsigned long)0,(long long)0);
  __FactStore(40,(unsigned long)(& i));
  i = 0;
  while (1) {
    while_0_continue: /* internal */ ;
    __FactLoad(41,(unsigned long)(& i),(long long)i);
```

```
    printf("Iteration: %d\n",i);
    __FactClearStack(42);
    __FactLoad(45,(unsigned long)(& i),(long long)i);
    __FactLoad(44,(unsigned long)0,(long long)1);
    __FactApply2(43,0,(long long)(i + 1));
    __FactStore(46,(unsigned long)(& i));
    i ++;
    __FactLoad(49,(unsigned long)(& i),(long long)i);
    __FactLoad(48,(unsigned long)(& dim),(long long)dim);
    __FactApply2(47,16,(long long)(i < dim));
    if (i < dim) {
      __FactBranch(50,141,1);
      ;
    }
    else {
      __FactBranch(51,142,0);
      goto while_0_break;
    }
  }
  while_0_break: /* internal */ ;
  ;
  __FactReturn(52);
  return;
}

void simpleWhile(int dim)
{
  int i;
  __FactCall(54,5);
  __FactStore(53,(unsigned long)(& dim));
  __FactLoad(55,(unsigned long)0,(long long)0);
  __FactStore(56,(unsigned long)(& i));
  i = 0;
  while (1) {
    while_0_continue: /* internal */ ;
    __FactLoad(59,(unsigned long)(& i),(long long)i);
    __FactLoad(58,(unsigned long)(& dim),(long long)dim);
    __FactApply2(57,16,(long long)(i < dim));
    if (i < dim) {
      __FactBranch(60,151,1);
      ;
    }
    else {
      __FactBranch(61,152,0);
      goto while_0_break;
    }
    __FactLoad(62,(unsigned long)(& i),(long long)i);
```

```
    printf("Iteration: %d\n",i);
    __FactClearStack(63);
    __FactLoad(66,(unsigned long)(& i),(long long)i);
    __FactLoad(65,(unsigned long)0,(long long)1);
    __FactApply2(64,0,(long long)(i + 1));
    __FactStore(67,(unsigned long)(& i));
    i ++;
  }
  while_0_break: /* internal */ ;
  ;
  __FactReturn(68);
  return;
}

int oneReturn(int el)
{
  int __retres;
  __FactCall(70,6);
  __FactStore(69,(unsigned long)(& el));
  __FactLoad(73,(unsigned long)(& el),(long long)el);
  __FactLoad(72,(unsigned long)0,(long long)5);
  __FactApply2(71,12,(long long)(el == 5));
  if (el == 5) {
    __FactBranch(74,161,1);
    __FactLoad(76,(unsigned long)0,(long long)0);
    __FactStore(77,(unsigned long)(& __retres));
    __retres = 0;
    goto return_label;
  }
  else {
    __FactBranch(75,163,0);
    __FactLoad(78,(unsigned long)0,(long long)1);
    __FactStore(79,(unsigned long)(& __retres));
    __retres = 1;
    goto return_label;
  }
  __FactLoad(80,(unsigned long)0,(long long)5);
  __FactStore(81,(unsigned long)(& __retres));
  __retres = 5;
  return_label: /* internal */
    __FactLoad(82,(unsigned long)(& __retres),(long long)__retres);
    __FactReturn(83);
    return __retres;
}

void addElse(int el)
{
```

```
  __FactCall(85,7);
  __FactStore(84,(unsigned long)(& el));
  __FactLoad(88,(unsigned long)(& el),(long long)el);
  __FactLoad(87,(unsigned long)0,(long long)5);
  __FactApply2(86,12,(long long)(el == 5));
  if (el == 5) {
    __FactBranch(89,169,1);
    printf("5");
    __FactClearStack(91);
  }
  else {
    __FactBranch(90,170,0);
    ;
  }
  ;
  __FactReturn(92);
  return;
}

void simplePred(int el)
{
  __FactCall(94,8);
  __FactStore(93,(unsigned long)(& el));
  __FactLoad(97,(unsigned long)(& el),(long long)el);
  __FactLoad(96,(unsigned long)0,(long long)0);
  __FactApply2(95,13,(long long)(el != 0));
  if (el != 0) {
    __FactBranch(98,175,1);
    printf("true");
    __FactClearStack(100);
  }
  else {
    __FactBranch(99,176,0);
    printf("false");
    __FactClearStack(101);
  }
  ;
  __FactReturn(102);
  return;
}

int main(void)
{
  int __retres;
  int **matrix;
  int a;
  __globinit_whole_program();
```

```
  __FactCall(103,9);
  simpleMemory(matrix);
  __FactClearStack(104);
  __FactLoad(105,(unsigned long)0,(long long)5);
  a = oneReturn(5);
  __FactHandleReturn(107,(long long)a);
  __FactStore(106,(unsigned long)(& a));
  __FactLoad(108,(unsigned long)(& a),(long long)a);
  printf("%d\n",a);
  __FactClearStack(109);
  __FactLoad(110,(unsigned long)0,(long long)0);
  __FactStore(111,(unsigned long)(& __retres));
  __retres = 0;
  __FactLoad(112,(unsigned long)(& __retres),(long long)__retres);
  __FactReturn(113);
  return __retres;
}

void __globinit_whole_program(void)
{
  __FactInit();
}
```

Listing B.2: Exemplo de programa completamente instrumentado.

Apêndice C

Conjunto de Testes do Crest

Este apêndice tem como propósito apresentar alguns dos testes do conjunto de testes do *Crest*. Este é composto por treze testes distintos que visam refletir as capacidades e falhas que esta ferramenta contém.
Com estes testes espera-se indicar que o *FACT* consegue, pelo menos, igualar as capacidades do *Crest*. Nesta dissertação, apenas são apresentados e comentados cinco testes, considerados relevantes, para provar o funcionamento do *FACT*, caso contrário tornaria a discussão demasiado extensa.

C.1 Primeiro Teste

Este primeiro teste, apresentado no *listing* C.1, inicialmente preenche um vector de duzentos elementos e inicializa-os com o valor 400, exceto a posição 100 que irá ter o valor 13.
A ideia é conseguir demonstrar que é possível gerar um valor que garanta que a mensagem "GOAL!" é alcançada.

```
#include <fact.h>
#include <stdio.h>

int data[200];

int main(void) {
  unsigned char c;
  FACT_unsigned_char(c);

  for (int i = 0; i < 200; i++) {
    data[i] = 400;
  }
  data[100] = 13;

  for (int i = 0; i < 200; i++) {
    if (c == data[i]) {
      fprintf(stderr, "GOAL!\n");
    }
  }

  return 0;
}
```

Listing C.1: Pesquisa de elementos num vector.

C.2 Segundo Teste

Este segundo teste, apresentado no *listing* C.2, cria uma variável "a", que será uma variável simbólica, e é submetida a várias funções que indicam algumas caraterísticas matemáticas sobre esta.
A ideia é conseguir demonstrar que é possível recolher as restrições associadas a esta variável dentro da função para, posteriormente, correr todos os caminhos de execução possíveis.

```
#include <fact.h>
#include <stdio.h>

void f(int);
void g(int);
void h(int);
void i(int);

void f(int a) {
  if (a > 13) {
    printf("greater␣than␣13\n");
  } else {
    printf("not␣greater␣than␣13\n");
  }
}

void g(int a) {
  h(a);

  if (a == 7) {
    printf("7\n");
  } else {
    printf("not␣7\n");
  }

  i(a);
}

void h(int a) {
  if (a == -4) {
    printf("-4\n");
  } else {
    printf("not␣-4\n");
  }
}

void i(int a) {
  if (a == 100) {
    printf("100\n");
```

```
  } else {
    printf("not 100\n");
  }
}

int main(void) {
  int a;
  FACT_int(a);

  if (a == 19) {
    printf("19\n");
  } else {
    printf("not 19\n");
  }

  f(a);

  g(a);

  if (a != 1) {
    printf("not 1\n");
  } else {
    printf("1\n");
  }

  return 0;
}
```

Listing C.2: Utilização de uma variável simbólica em funções.

C.3 Terceiro Teste

Este terceiro teste, apresentado no *listing* C.3, declara cinco variáveis distintas na qual são realizadas operações matemáticas. Caso a igualdade apresentada na instrução *if* seja verdadeira a função retorna o valor de um, caso contrário retorna o valor zero.
A ideia é demonstrar que até para expressões complexas que envolvam várias variáveis, o *Fact*, consegue resolver este tipo de restrições.

```
#include <fact.h>

int main(void) {
  int a, b, c, d, e;
  FACT_int(a);
  FACT_int(b);
  FACT_int(c);
  FACT_int(d);
  FACT_int(e);
```

```
  if (3*a + 3*(b - 5*c) + (b+c) - a == d - 17*e) {
    return 1;
  } else {
    return 0;
  }
}
```

Listing C.3: Programa com expressões matemáticas complexas.

C.4 Quarto Teste

Este quarto teste, apresentado no *listing* C.4, declara três variáveis distintas e aplica-lhes operações de *shift*.
A ideia é demonstrar que as operações de *shift* também são suportadas pelo *FACT*.

```
#include <fact.h>
#include <stdio.h>

int main(void) {
  int x, y, z;

  FACT_int(x);
  FACT_int(y);
  FACT_int(z);

  if (((x + 3) << 2) == 96) {
    printf("A\n");
    if (((y - 15) << x) == z) {
      printf("B\n");
    } else {
      printf("C\n");
    }
  }

  return 0;
}
```

Listing C.4: Programa com operações de shift.

C.5 Quinto Teste

Este quinto teste, apresentado no *listing* C.5, possui uma matriz e um *array* de tamanho fixo inicializados em tempo de execução.
Neste exemplo é apresentada uma limitação do *FACT*, pois não é possível utilizar um índice de uma matriz ou *array* simbolicamente. Portanto as duas últimas condições não serão alcançadas.

```
#include <fact.h>
#include <stdio.h>

int A[100];
char B[12][97];

int main(void) {
  int x, y, i, j;

  FACT_int(x);
  FACT_int(y);

  if ((x < 0) || (x >= 97))
    return 0;
  if ((y < 0) || (y >= 12))
    return 0;

  for (i = 0; i < 100; i++) {
    A[i] = i;
  }

  for (i = 0; i < 12; i++) {
    for (j = 0; j < 97; j++) {
      B[i][j] = i + j;
    }
  }

  if (A[x] == 7) {
    printf("Hello!\n");
  }

  if (B[y][x] == 42) {
    printf("World!\n");
  }

  return 0;
}
```

Listing C.5: Programa com índices de *arrays* simbólicos

Glossário

API	Uma *Application Program Interface* designa um conjunto de métodos ou funções disponibilizadas por um *software* que permitem aceder às funcionalidades deste sem que o utilizador se preocupe com os detalhes de implementação.
Bug	Designa um erro no funcionamento da lógica de um programa.
Callback	Uma *Callback* é uma função passada como parâmetro para outra função, com o intuito desta última executar a função argumento na ocorrência de um evento específico.
Feedback	Designa a reação ou resposta a uma determinada ação.
Framework	É um conjunto de conceitos usados para resolver um problema de um domínio específico. Compreende um conjunto de classes implementadas numa linguagem de programação, usadas para auxiliar o desenvolvimento de *software*.
Functor	Um módulo em *OCaml* é chamado de *Functor* se este é um módulo parametrizado, isto é, aceita outro módulo como parâmetro de entrada.
Input	Representa um conjunto de dados de entrada para um sistema ou *software*.
Interface	Encontra-se normalmente associado à parte gráfica que permite ao utilizador interagir com um determinado sistema.
Kernel	Designa o componente central de um sistema. Normalmente faz a ligação entre as principais funcionalidades deste.
Linking	Termo que designa a ligação de várias bibliotecas ou programas durante o processo de compilação.
Multithreading	É um modelo de programação e execução que permite que várias *threads* existam num contexto de um único processo, isto é, permite a execução concorrente de várias tarefas.
Output	Representa um conjunto de dados de saída para um sistema ou *software*.
Plug-in	É um programa usado para adicionar caraterísticas ou expandir as funcionalidades de um *software* já existente.
Script	Um *script* é uma rotina de código independente que pode ser executada por um *software* para a qual foi desenvolvido.
Software	É o resultado da execução de uma sequência de instruções quando executada num computador. Inclui não só o programa de computador propriamente dito, mas também manuais e especificações.
SMT	*Satisfiability Modulo Theories* é um problema de decisão de fórmulas lógicas.