



Universidade do Minho

Mestrado Integrado em Engenharia Informática

Ferramentas e técnicas de *Abstract Interpretation*

ENGENHARIA DE SEGURANÇA
PROJETO 2

Grupo 2

Paulo Gameiro - A72067
Pedro Rodrigues - PG41092
Rafaela Soares - A79034

Braga, Portugal
11 de Maio de 2020

Conteúdo

1	Introdução	2
2	<i>Abstract Interpretation</i>	3
3	Ferramentas	5
3.1	Julia Analyser	5
3.1.1	Pré-requisitos de utilização	5
3.1.2	Como se inicializa uma análise?	6
3.1.3	Como são apresentados os resultados?	8
3.2	SPARTA	11
3.2.1	Pré-requisitos de utilização	11
3.2.2	Como se inicializa uma análise e como são apresentados os resultados? . . .	11
3.2.3	Exemplo de utilização	11
3.3	Clang Static Analyser	12
3.3.1	Pré-requisitos de utilização	12
3.3.2	Como se inicializa uma análise e como são apresentados os resultados? . . .	12
3.4	Polyspace Code Prover	13
3.5	Coverity	13
3.6	CodeSonar	13
3.7	C++Test/BugDetective	14
4	Considerações finais	15

1 Introdução

Na maioria dos casos, a validação de *software* parece ser a fase mais dispendiosa e complexa de todo o processo de desenvolvimento de *software*, representando mais de metade do custo total.

As métricas de *softwares* possibilitam realizar uma das atividades mais fundamentais do processo de gestão de projectos - o planeamento. A partir deste, pode-se identificar a quantidade de esforço, de custo e das atividades que serão necessárias para a realização do projecto.

Do ponto de vista de medição, podem ser divididas em duas categorias: medidas diretas e indiretas. Pode-se considerar como medidas diretas do processo de engenharia de *software* o custo e o esforço aplicados ao desenvolvimento e manutenção do *software* e do produto, a quantidade de linhas de código produzidas e o total de defeitos registados durante um determinado período de tempo. Porém, a qualidade e a funcionalidade do *software*, ou a sua capacidade de manutenção, são mais difíceis de serem avaliadas e só podem ser medidas de forma indireta.

No intuito de se amenizar custos, acredita-se que se deva recorrer a métricas de qualidade de *software*, sendo uma delas a interpretação abstracta.

2 *Abstract Interpretation*

Em primeira instância, é necessário compreender o conceito de *Abstract Interpretation*, também conhecido como *Deep Flow Analysis*. Trata-se de um método formal matemático, desenvolvido no final dos anos 70, baseado em técnicas formais de verificação que proporcionam aos programadores abstrair a semântica do *software*. (1)

A verificação de programas consiste na melhoria da semântica do programa que satisfaz a sua especificação. A interpretação abstracta formaliza a ideia de que isto é possível com um determinado nível de abstracção, onde os detalhes envolvendo a semântica e a especificação são ignorados. Um exemplo da abstracção de semântica é a lógica de Hoare, enquanto que os exemplos de abstracção de especificação são o invariante, correcção parcial e total.

De salientar que a abstracção deve ser sólida - a prova de que a semântica abstracta satisfaz a especificação abstracta implica que a semântica concreta também satisfaz a especificação concreta. Uma abstracção fraca pode levar a falsos negativos, ou seja, o programa pode ser dado como correcto em relação à especificação, quando este é, de facto, incorrecto.

Para além disso, a abstracção deve ser também completa e não podem ser deixados de fora aspectos da semântica. Abstracções incompletas levam a falsos positivos, é indicado que a especificação pode ser potencialmente violada por algumas execuções mas não é verdade. Em todos os casos, a abstracção de semântica deve ser sólida, completa e simples, de forma a evitar explosões combinatórias.

Os métodos formais são interpretações abstractas com maneiras diferentes de abstrair as semânticas. Em todos os casos, as semânticas abstractas têm de ser escolhidas para serem representadas por computadores.

- Em *model-checking*, a abstracção de semântica é fornecida manualmente pelo utilizador na forma de um modelo finito de execuções do programa. Em alguns casos, estas execuções podem ser computadas automaticamente.
- Nos métodos dedutivos (*deductive methods*) a abstracção semântica é especificada por condições de verificação e têm de ser fornecidas pelo utilizador na forma de propriedades indutivas (propriedades que se mantêm verdadeiras à medida que o programa é executado) satisfazendo estas condições de verificação. Algumas propriedades indutivas podem ser calculadas automaticamente, através de análise estática.
- Em análise estática, a abstracção de semântica é calculada automaticamente graças a aproximações predefinidas, também possivelmente parametrizadas pelo utilizador.

Assim, o objetivo da interpretação abstracta é fornecer uma teoria básica coerente e conceptual, de forma a compreender numa *framework* as diversas ideias, razões, métodos e ferramentas num programa de análise e verificação. Além disto, tem como intento também guiar a forma correcta de ferramentas automáticas para a análise e verificação de programas.

Para se interiorizar melhor esta métrica, foque-se na equação $-3 * 4 * 5 = ?$. Não é necessário que esta seja resolvida para se ter conhecimento de que o resultado é um número negativo, se se recorrer às regras do sinal de multiplicação. Tal poderia ser um exemplo de aplicação de interpretação abstracta, caso o resultado dessa equação fosse um parâmetro para o cálculo da raiz quadrada. Uma vez que se tem conhecimento de que o resultado da equação nunca poderia ser um número positivo, sabe-se que o cálculo da raiz quadrada não seria possível, uma vez que é necessário um número positivo, pelo que, se detecta um erro antes de tal ser feito. (1)

Desta forma, é possível detectar hipotéticos problemas de confiabilidade no *software*, uma vez que existem ferramentas de interpretação abstracta que permitem detectar automaticamente diversos tipos de erros de programação relacionados com o fluxo de controlo de um programa, sem que este seja realmente executado. Exemplos de possíveis erros que podem ser identificados são *overflows* de *buffers*, conexões de bases de dados não fechadas, não referência a *null pointer*, entre outros não menos relevantes. (2)

3 Ferramentas

3.1 Julia Analyser

Desenvolvida pela empresa JuliaSoft, especialista em verificação de *software*, esta ferramenta de interpretação abstrata compromete-se a auxiliar os seus utilizadores a garantir segurança e qualidade dos seus programas, de forma a "reduzir significativamente os seus custos de desenvolvimento e manutenção e eliminar riscos relacionados a vulnerabilidades de segurança e vazamentos de privacidade". (15)

Esta ferramenta de análise estática pode ser utilizada em código Java, Android e .NET(C#) e integra o método científico de interpretação abstrata, de forma a:

- Adaptar-se ao ambiente de desenvolvimento existente, facilitando a remoção de defeitos e vulnerabilidades o mais rápido possível.
- Integrar-se no ambiente DevOps e Integração Contínua em questão, por meio de *scripts* ou conectando-o a consolas de qualidade como o SonarQube.
- Reconstruir o gráfico do programa e examinar todos os caminhos de execução possíveis, sendo capaz de identificar também vulnerabilidades complexas.
- Analisar o bytecode, tornando possível verificar o código proprietário ou aplicações de terceiros.
- Ajudar a corrigir os erros: dispõe de avisos únicos diretamente na linha de código e categorização dos resultados, de acordo com seu contexto e com base no tipo e gravidade do erro.
- Coordenar o esforço, usando o recurso avançado de painel de KPI. (15)

A partir da sua instalação em IDEs como Eclipse, Visual Studio, IntelliJ ou Android Studio, é possível conectar com o servidor de análise e executar várias análises num limite de 10000 LoCs, com a versão gratuita. Podendo estas serem também examinadas posteriormente na *web console* com outros utilizadores. Para além disso, é também possível reorganizar os avisos de acordo com diferentes parâmetros e exportar um arquivo XML de todos os avisos. Tal é possível, pois a ferramenta inclui uma elevada gama de *checkers* para identificar *bugs*, vulnerabilidades e ineficiências, assim como também detalha os avisos de "defeito" ou sugestão de correção de código.

Já a partir da *web console*, é possível consultar análises anteriores e visualizar graficamente os *warnings*, entre outras funcionalidades não menos relevantes.

3.1.1 Pré-requisitos de utilização

- Conhecimento em Java, Android ou .NET(C#).
- Experiência num dos seguintes ambientes de desenvolvimento integrado: Eclipse, Visual Studio, IntelliJ ou Android Studio.
- Criar uma conta em <https://portal.juliasoft.com/>.
- Seguir a documentação do IDE escolhido, disponibilizado em "Quickstart" no menu de utilizador em <https://portal.juliasoft.com/>, no intuito de instalar a ferramenta com sucesso.

3.1.2 Como se inicializa uma análise?

A título demonstrativo, será apresentada a sua utilização no Android Studio, contudo em todas as documentações das IDEs referidas anteriormente, existe uma breve explicação. Para se proceder à análise com Julia, é necessário:

1. Escolher os módulos de código a serem analisados.

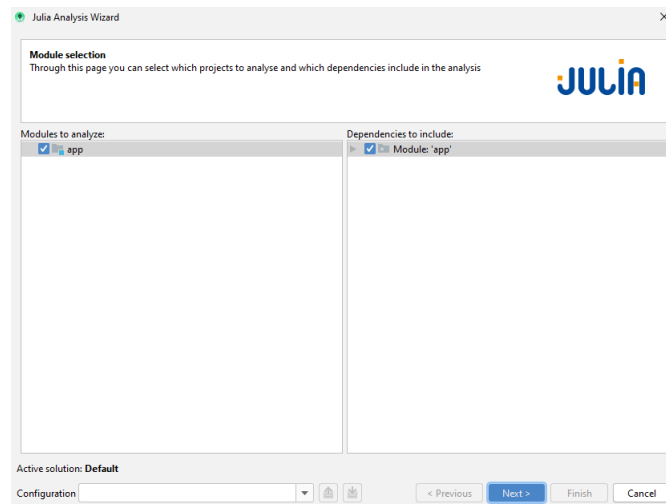


Figura 1: Escolha de módulos de código a serem analisados no Julia Analyser

2. Escolher a opção desejada de análise.

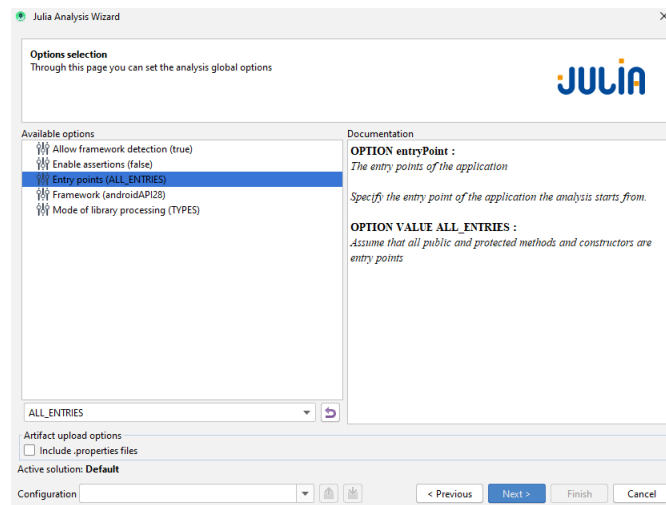


Figura 2: Escolha da opção desejada de análise no Julia Analyser

3. Escolher os *checkers* de análise.

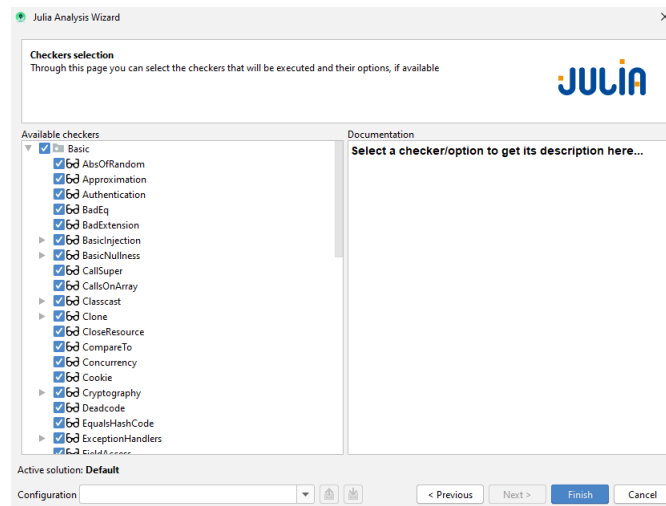


Figura 3: Escolha dos *checkers* de análise no Julia Analyser

4. Proceder à análise e esperar os resultados. O acompanhamento da realização da análise pode ser feita através do Event Log.

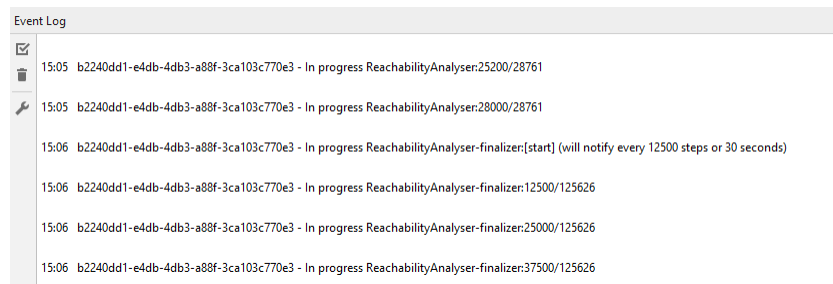


Figura 4: Acompanhamento da realização da análise

3.1.3 Como são apresentados os resultados?

O IDE é bastante intuitivo na representação dos resultados da análise realizada. Este identifica, ao longo do código, todos os *warnings* detetados, com cores distintas. Estes podem ter cor verde, amarelo e vermelho, representando baixa, média e elevada gravidade, respetivamente.



Figura 5: Excerto de código com Julia *warnings*

Abaixo do código, no painel Julia Analyses, os *warnings* detetados são apresentados detalhadamente e podem ser identificados no código, através de duplo clique sobre o *warning* em questão.

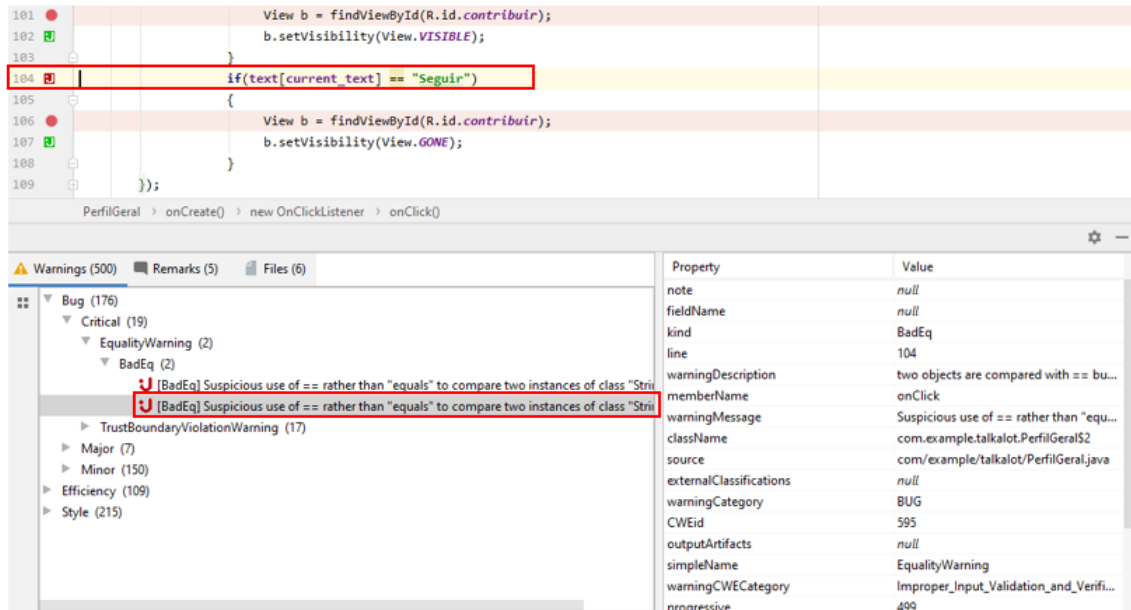


Figura 6: Exemplo de um *warning* crítico detetado

Fite-se o *warning* cima. Este claramente é um *bug* crítico, uma vez que não é aconselhado a utilização do operador relacional `==`, quando se pretende comparar *strings*, uma vez que este operador compara se ambas as *strings* se referem ao mesmo local de memória. Tal retorna a comparação como falsa, mesmo que o conteúdo de `text[currentText]` seja "Seguir".

No sentido de prevenir tal facto, o Julia Analyser sugere a utilização do método `equals`, uma vez que este compara os valores de dois objetos, independentemente de onde estão localizados na memória.

Outro exemplo de um *warning* é o apresentado abaixo. Este apresenta baixa gravidade, pois não compromete o programa em si, uma vez que o parâmetro relativo às permissões está definido e não é *null* (`String[] Permissions`). Contudo, como não é verificado se o parâmetro relativo às *permissions*, no método `requestPermissions`, é ou não *null*, podia existir a possibilidade de o método ser invocado em vão, não sendo dada nenhuma permissão, caso esse parâmetro fosse *null*.

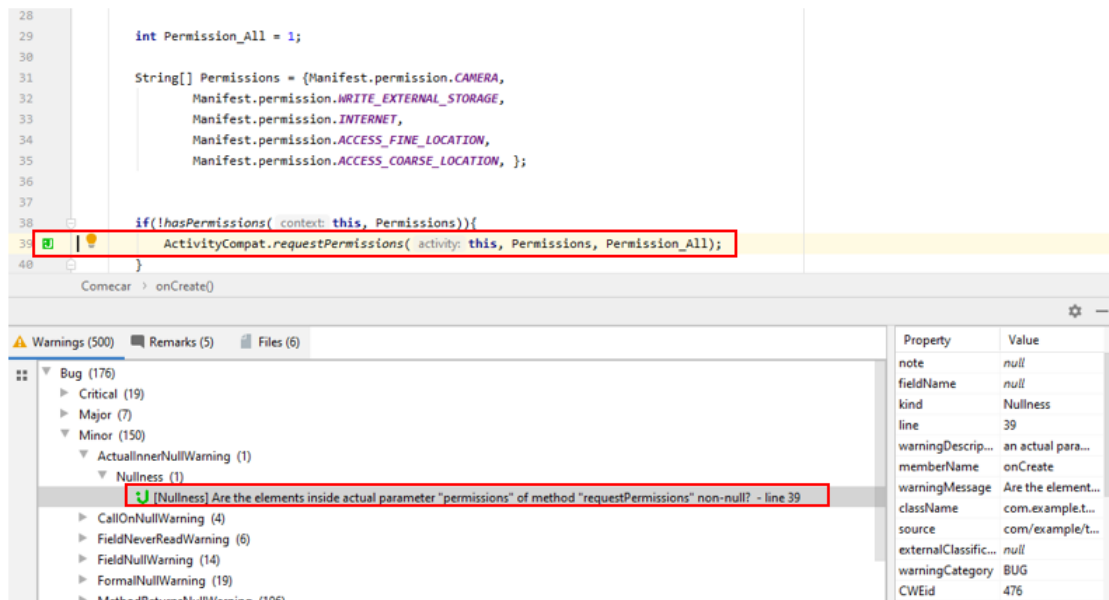


Figura 7: Exemplo de um *warning* de baixa gravidade detetado

De referir, ainda, que todos os *warnings* detetados podem ser agrupados, da seguinte forma:

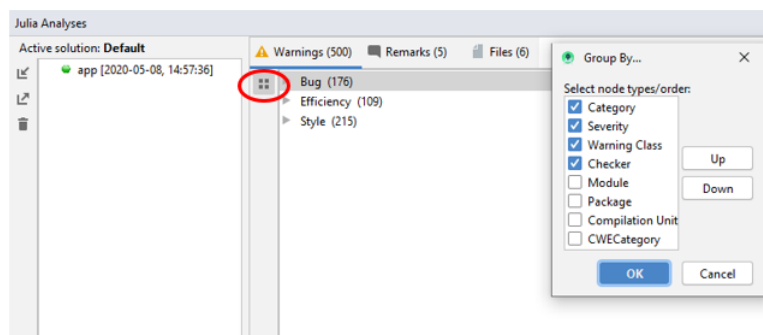


Figura 8: Implementação

Para além destas funcionalidades no IDE, é possível usufruir de diversas outras na *web console*, especificamente em "Solutions" na "Dashboard". Esta possibilita aos utilizadores analisarem, em versão *web*, todo o resultado obtido, podendo este ser extraído. Estes podem consultar os *checkers* utilizados, os ficheiros analisados, o total de LoCs gasto, os *warnings* e os gráficos gerados, entre outros não menos relevantes.

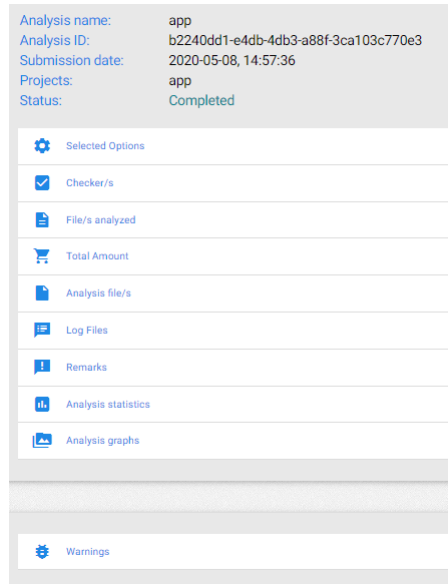


Figura 9: Funcionalidades da *web console*

É interessante constatar, através de uma representação visual mais apelativa, os tipos de *warnings* detetados, assim como a gravidade associada aos mesmos. Tal pode permitir aos programadores terem em atenção potenciais erros ou lapsos no futuro.

No programa alvo de análise, cerca de 35% dos *warnings* detetados são *bugs*, o que é uma percentagem considerável, assim como os *warnings* de eficiência, representando cerca de 22%.

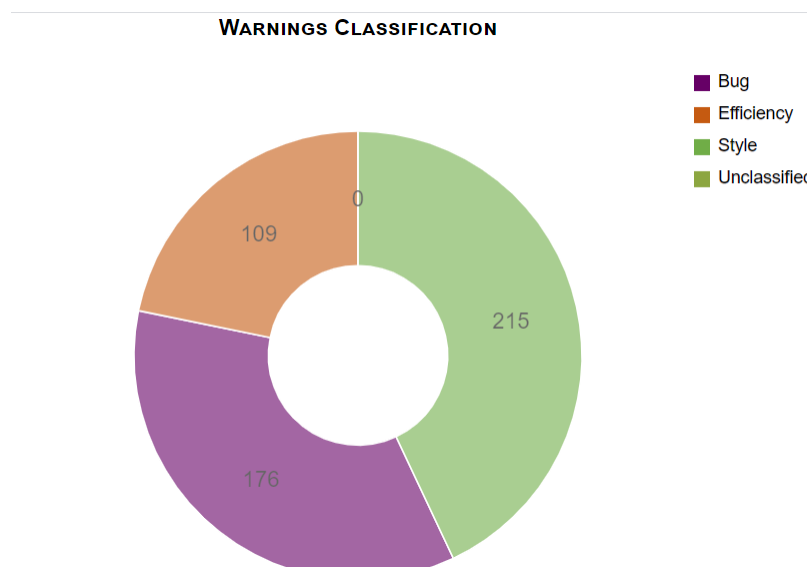


Figura 10: Classificação dos *warnings*

3.2 SPARTA

O SPARTA é uma biblioteca de componentes de *software*, direcionada para C++, e que permite construir analisadores estáticos que podem ser executados num ambiente de produção.

Esta biblioteca oferece uma série de componentes que contêm uma API simples e que podem ser facilmente montados para construir um analisador estático pronto para a produção, de forma a que um engenheiro, que utilize esta biblioteca, possa focar-se exclusivamente na lógica.

Nesse sentido, permite que a ferramenta do programador se foque nos três eixos fundamentais na produção de uma análise, que são as propriedades do programa a analisar (extensão de valores numéricos, por exemplo), a granularidade das propriedades (sensível ao contexto, interprocedimento, entre outros não menos relevantes) e a representação das propriedades do programa (gráficos, intervalos, etc...). De referir também que permite obter os resultados da análise de uma forma mais eficiente e escalável.

3.2.1 Pré-requisitos de utilização

- Conhecimento em C++.
- Instalação da biblioteca SPARTA, através de <https://github.com/facebookincubator/SPARTA>.

3.2.2 Como se inicializa uma análise e como são apresentados os resultados?

De forma a realizar a análise, é necessário colocar os ficheiros que se pretende testar na diretoria **test** e executar o ficheiro **sparta_test**, que retorna o resultado da execução da análise em cada um dos ficheiros. O SPARTA irá representar os testes bem sucedidos a verde e os testes mal sucedidos a vermelho.

```
Running main() from gmock_main.cc
[=====] Running 87 tests from 22 test cases.
[-----] Global test environment set-up.
[-----] 7 tests from DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0, where TypeParam
= sparta::DisjointUnionAbstractDomain<sparta::ConstantAbstractDomain<int>, sparta::ConstantAbstrac
tDomain<std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >>
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Basics
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Basics (0 ms)
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.JoinMeetBounds
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.JoinMeetBounds (0 ms)
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Idempotence
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Idempotence (0 ms)
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Reflexivity
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Reflexivity (0 ms)
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Commutativity
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Commutativity (0 ms)
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Absorption
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Absorption (0 ms)
[ RUN ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Relations
[ OK ] DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0.Relations (0 ms)
[-----] 7 tests from DisjointUnionAbstractDomain/AbstractDomainPropertyTest/0 (0 ms total)
```

Figura 11: Resultados de um exemplo de análise com a ferramenta SPARTA

3.2.3 Exemplo de utilização

O otimizador de código de Android **ReDex** é baseado nesta biblioteca e consegue, devido ao SPARTA, expressar estas análises enquanto mantém um alto nível de performance. A *liveness analysis* do **ReDex** pode ser realizada em menos de 60 linhas de código. Ainda assim, o resultado do analisador pode facilmente escalar para métodos com milhares de blocos e variáveis, devido à performance dos algoritmos do SPARTA.

Apesar da sua utilização no ReDex, o SPARTA não contém lógica especificamente direcionada para Android.

3.3 Clang Static Analyser

Clang Static Analyser é uma ferramenta que tem como objetivo encontrar *bugs* em C, C++ e em programas Objective-C. Esta pode ser utilizada através do terminal ou através de within Xcode, em caso de macOS.

3.3.1 Pré-requisitos de utilização

- Conhecimento em C ou C++.
- Instalação do Clang, através de <http://clang-analyzer.llvm.org/>.

3.3.2 Como se inicializa uma análise e como são apresentados os resultados?

Apesar do seu uso no terminal não ser tão intuitivo face ao uso de um editor que identifique tais *bugs*, através da **documentação** disponibiliza por esta ferramenta, o seu uso é acessível.

Um exemplo prático desta ferramenta pode ser a utilização da *flag* `-fsyntax-only`, que executa o pré-processor, parser e type checking stages. Fite-se o seguinte programa.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

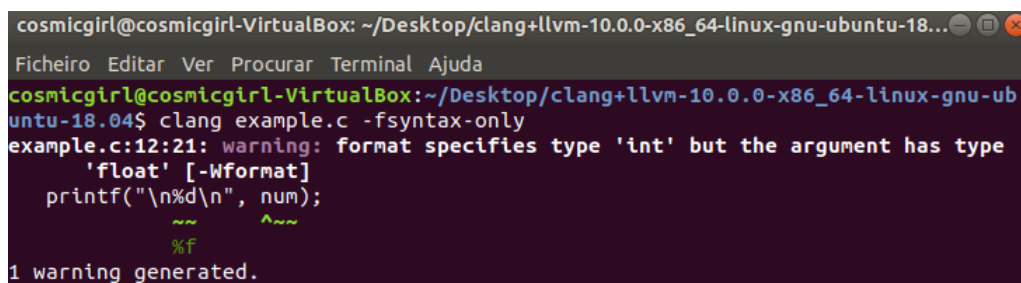
int main() {

    float num = 1.0;

    printf("\n%d\n", num);

    return 0;
}
```

Na imagem abaixo, pode-se evidenciar que a ferramenta identifica e reporta que está a ser referenciado um float, onde deveria de ser referenciado um int, no programa alvo de análise.



```
cosmicgirl@cosmicgirl-VirtualBox: ~/Desktop/clang+llvm-10.0.0-x86_64-linux-gnu-ubuntu-18...
Ficheiro Editar Ver Procurar Terminal Ajuda
cosmicgirl@cosmicgirl-VirtualBox:~/Desktop/clang+llvm-10.0.0-x86_64-linux-gnu-ub
untu-18.04$ clang example.c -fsyntax-only
example.c:12:21: warning: format specifies type 'int' but the argument has type
      'float' [-Wformat]
    printf("\n%d\n", num);
                ^~~~
                %f
1 warning generated.
```

Figura 12: Execução da *flag* `-fsyntax-only`

3.4 Polyspace Code Prover

Desenvolvido por MathWorks, esta ferramenta de análise estática baseada em análise semântica e interpretação abstrata recorre a métodos formais para evidenciar o comportamento interprocessual, o controlo e o fluxo de dados do *software*. Nesse sentido, tem como intuito determinar onde podem ocorrer *overflows*, divisão por zero, acesso a *array* fora dos limites estabelecidos, entre outros erros *run-time* não menos relevantes, nas linguagens C e C++. (3)

Esta ferramenta é bastante intuitiva, uma vez que cada segmento de código é representado por cores diferentes, de forma a estabelecer a seguinte categorização:

- **Verde** - representa que o segmento de código está livre de erros *run-time*.
- **Vermelho** - representa que o segmento de código apresenta falha comprovada, cada vez que a operação é executada.
- **Cinza** - representa que o segmento de código é inacessível, o que pode indicar um problema funcional.
- **Laranja** - representa que o segmento de código não é comprovado para todas as condições de *run-time*.

3.5 Coverity

A Coverity é uma ferramenta paga para *software* de código fechado, mas a sua solução **Coverity Scan** é gratuita para todo o software open-source.

Contém funcionalidades avançadas como a identificação de que testes necessitam de ser executados para cada alteração ao código e podem cobrir tanto o código modificado, como as funções que executam o novo código. Esta ferramenta dispõe ainda de um bom suporte em diferentes compiladores e uma fácil configuração de análises estáticas. Está disponível para diversas linguagens como C/C++, Java, Javascript, entre outras não menos relevantes.

3.6 CodeSonar

Esta ferramenta, desenvolvida pela GrammaTech, permite analisar fluxos de dados e examinar o processamento de todo o programa que está a ser analisado, realizando estes testes em milhões de linhas rapidamente, sendo que a sua análise incremental torna mais eficiente a execução dos testes às modificações diárias do código.

O CodeSonar tem suporte para diversos *standards* de programação como MISRA C:2012, ISO-26262, DO-178B, entre outros.

Recentemente, a GrammaTech anunciou a possibilidade de integração do CodeSonar no Julia Analyser, descrito anteriormente, o que permite uma maior precisão, análises mais avançadas de múltiplas linguagens numa única interface e facilidade de colaboração entre equipas.

3.7 C++Test/BugDetective

O C++Test, também conhecido como cpptest, é uma simples e poderosa *framework* e testes unitários para realizar testes automáticos em C++. Está diretamente focada na sua usabilidade, apesar de que a *review* de alguns utilizadores afirma de que deteta bastantes falsos positivos, tornando a análise mais dificultada.

Relativamente ao BugDetective, desenvolvido pela Parasoft, fornece deteção de problemas em *runtime* e instabilidades na aplicação, tais como *NullPointerExceptions*, *SQL injections*, *leaks* de recursos, entre outros, em funções que executem diversos métodos, classes ou bibliotecas.

O BugDetective está disponível no Parasoft JTest (Java), C++Test (para C e C++), e no .TEST (para .NET).

4 Considerações finais

A criação de *software* abrange o desenvolvimento e implementação de métodos e processos para garantir que o *software* funcione conforme o planeado, livre de defeitos de *design* e falhas de implementação. Nesse sentido, a validação de *software* aponta ser a fase mais dispendiosa e complexa de todo o processo. Posto isto, as métricas de *software* possibilitam avaliar e identificar quantitativamente o esforço e custos das atividades que vão ser realizadas.

Um exemplo destas é a interpretação abstracta. Esta métrica de qualidade de *software* permite reduzir os custos e tem como objectivo fornecer uma teoria básica coerente e conceptual, de forma a que as diversas ideias, razões, métodos e ferramentas possam compreender tudo numa única *framework*. Desta forma, é possível detectar hipotéticos problemas de confiabilidade no *software*, uma vez que existem ferramentas de interpretação abstracta que permitem detectar automaticamente diversos tipos de erros de programação.

Foram enumeradas algumas ferramentas e demonstrado a utilização das mesmas, sendo muitas delas para C, C++ ou C#. As ferramentas variam desde programas com interface a bibliotecas que analisam o código submetido. No entanto, a utilização de algumas ferramentas não foi possível, como foi o caso da Polyspace Code Prover, Coverity, CodeSonar e C++Test/BugDetective, pelo facto de não se ter conseguido obter uma versão gratuita.

Referências

- [1] Boulanger, J-L. (2013). Static Analysis of Software: The Abstract Interpretation, consultado em 29 de abril - https://books.google.pt/books?hl=pt-PT&lr=&id=d0GE2eZIFEAC&oi=fnd&pg=PT8&dq=abstract+interpretation+software&ots=JgyBx0jJf5&sig=4Kw0bx01_PcUwiG92xD-f0olksc&redir_esc=y#v=onepage&q=abstract%20interpretation%20software&f=false
- [2] Jansen, P. (2018). The TIOBE Quality Indicator: A pragmatic way of measuring code quality, consultado a 29 de abril - https://www.tiobe.com/files/TIOBEQualityIndicator_v4_3.pdf.
- [3] Polyspace Code Prover, consultado a 30 de abril de 2020 - <https://www.mathworks.com/products/polyspace-code-prover.html>.
- [4] Abstract Interpretation - Mozilla Wiki, consultado a 02 de Maio de 2020 - https://wiki.mozilla.org/Abstract_Interpretation
- [5] Abstract Interpretation - Departamento de Informática da Escola normal superior de Paris, consultado a 02 de maio de 2020 - https://www.di.ens.fr/~cousot/AI/#tth_sEc1
- [6] Abstract Interpretation - Departamento de Informática da Escola normal superior de Paris, consultado a 02 de maio de 2020 - <https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html>
- [7] Abstract Interpretation - Wikipedia, consultado a 02 de maio de 2020 - https://en.wikipedia.org/wiki/Abstract_interpretation
- [8] JuliaSoft, consultado a 06 de maio de 2020 - <https://juliasoft.com/company/>
- [9] Parasoft Static Analysis and BugDetective Technology, consultado a 10 de maio de 2020 - <https://www.codeproject.com/Articles/26224/Static-Analysis-on-Steroids-Parasoft-BugDetective>
- [10] CppTest: Tutorial, consultado a 10 de maio de 2020 - https://cpptest.sourceforge.io/tutorial.html#tutorial_screenshots
- [11] CppTest, consultado a 10 de maio de 2020 - <https://cpptest.sourceforge.io/>
- [12] GrammaTech Announces Integration of JuliaSoft into CodeSonar, consultado a 10 de maio de 2020 - https://www.eejournal.com/industry_news/grammatech-announces-integration-of-juliasoft-into-codesonar/
- [13] CodeSonar, consultado a 10 de maio de 2020 - <https://www.grammatech.com/products/codesonar>
- [14] Coverity Scan - Projects, consultado a 10 de maio de 2020 - <https://scan.coverity.com/projects>
- [15] Coverity Scan - Static Analysis, consultado a 10 de maio de 2020 - <https://scan.coverity.com/>