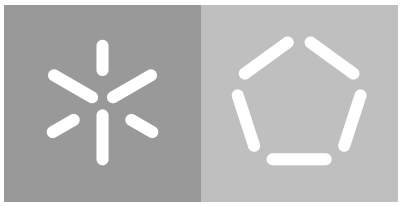


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rafael Alexandre Antunes Barbosa

Verificação Funcional de Controladores Arduino

Outubro 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rafael Alexandre Antunes Barbosa

Verificação Funcional de Controladores Arduino

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação supervisionada por
Jorge Miguel de Matos Sousa Pinto
Cláudio Filipe Belo da Silva Lourenço

Outubro 2018

AGRADECIMENTOS

Para começar, gostaria de agradecer aos meus orientadores Jorge Sousa Pinto e Cláudio Lourenço pela disponibilidade e ensinamentos durante a dissertação.

De seguida, gostaria de agradecer aos meus pais José e Cristina, ao meu irmão Gabriel e à minha família pelo apoio e motivação.

Por último, gostaria de agradecer à Elsa pelo amor e dedicação, pela disponibilidade e paciência, por acreditar. Obrigado por tudo.

ABSTRACT

In some contexts, namely in the development of autonomous vehicles in robotics, the exploitation of the Arduino platform for the development of embedded systems allows a simpler and faster implementation of the desired functionality than would be possible using the devices known as *Programmable Logic Controllers (PLCs)*, typically used. However, the software certification is a much more complicated task than the PLCs certification, due mainly to the complexity of the software and its difficult analysis. Although the software testing is the most used technique to validate it, it can not be used to prove the absence of errors. On the other hand, the formal verification of software is usually used to demonstrate the correctness of critical systems as, for example, aerospace industry and banking industry, where a failure means the loss of human lives or high monetary values. As the embedded systems are increasingly used in robotic systems, whose criticality is increasingly high, it is important to ensure their correctness.

With this, the intervention area focused on the communication protocols because they have an appropriate dimension given the scope of the dissertation. These are quite important, particularly in the context of robotic, since embedded systems are in constant communication with several peripherals, being fundamental the correction of the communication protocols. This dissertation aims at the application of formal verification tools to communication libraries of the Arduino platform. The adopted verification process comprises the application of the Frama-C tool, together with the WP plug-in, to perform a static analysis of annotated ANSI-C code with ACSL properties. Furthermore, another objective is the exploration of the experimental Frama-Clang plug-in because of the widespread interest in achieving C++ code certification.

KEYWORDS Arduino, Safety-Critical Embedded Systems, Formal Verification of Software, Communication Protocols, Frama-C, Frama-Clang

RESUMO

Em alguns contextos, nomeadamente no desenvolvimento de veículos autónomos em robótica, a exploração da plataforma Arduino para desenvolvimento de *embedded systems* possibilita uma implementação mais simples e rápida da funcionalidade desejada do que seria possível utilizando os dispositivos conhecidos como *Programmable Logic Controllers (PLCs)*, tipicamente utilizados. Contudo, a certificação de software é uma tarefa bastante mais complicada do que a certificação de PLCs, devido principalmente à complexidade do software e à sua difícil análise. Embora o teste de software seja a técnica mais utilizada para o validar, esta não pode ser utilizada para provar a ausência de erros. Por outro lado, a verificação formal de software é normalmente utilizada para demonstrar a correção de sistemas críticos como, por exemplo, indústria aeroespacial e indústria bancária, onde uma falha significa a perda de vidas humanas ou de valores monetários elevados. Como os *embedded systems* são cada vez mais utilizados em sistemas robóticos, cuja criticalidade é cada vez mais elevada, é importante garantir a sua correção.

Com isto, a área de intervenção incidiu sobre os protocolos de comunicação porque estes apresentam uma dimensão adequada dado o âmbito da dissertação. Estes são bastante importantes, em particular no contexto da robótica, uma vez que os *embedded systems* estão em constante comunicação com diversos periféricos, sendo fundamental a correção dos protocolos de comunicação. Esta dissertação tem como objetivo a aplicação de ferramentas de verificação formal a bibliotecas de comunicação da plataforma Arduino. O processo de verificação adotado compreende a aplicação da ferramenta Frama-C, em conjunto com o *plug-in* WP, para realizar uma análise estática de código ANSI-C anotado com propriedades ACSL. Para além disso, é ainda objetivo a exploração do *plug-in* experimental Frama-Clang devido ao interesse generalizado em alcançar a certificação de código C++.

PALAVRAS-CHAVE Arduino, *Safety-Critical Embedded Systems*, Verificação Formal de Software, Protocolos de Comunicação, Frama-C, Frama-Clang

CONTEÚDO

1 INTRODUÇÃO 1
- 1.1 Contextualização 1
- 1.2 Motivação 2
- 1.3 Objetivos e Contribuições 2
- 1.4 Organização do Documento 3

2 ESTADO DA ARTE 5
- 2.1 Arduino 5
- 2.2 Protocolos de Comunicação 7
- 2.3 Verificação Formal de Software 9
- 2.4 Frama-C 9
- 2.5 Trabalho Relacionado 10

3 TUTORIAL DE FRAMA-CLANG 14
- 3.1 Plug-in Frama-Clang 14
- 3.2 Exemplo de Utilização 15

4 CONHECIMENTO DA BIBLIOTECA WIRE 20
- 4.1 Biblioteca Wire 20
- 4.2 Ficheiro de Código C++ 21
- 4.3 Ficheiro de Código C 23

5 VERIFICAÇÃO DA BIBLIOTECA WIRE 25
- 5.1 Configuração e Parametrização do Frama-C 25
- 5.2 Anotações e Limitações 30
- 5.3 Análise de Resultados e Discussão 38

6 CONCLUSÃO 43
- 6.1 Conclusões 43
- 6.2 Trabalho Futuro 44

LISTA DE FIGURAS

Figura 1	Plataforma Arduino Uno	6
Figura 2	Arduino IDE - *Blink Example*	7
Figura 3	Topologia do Protocolo de Comunicação I2C	8
Figura 4	Topologia do Protocolo de Comunicação SPI	8
Figura 5	Frama-C - *Swap Example*	11
Figura 6	Condições de Verificação do Método *difference()*	19
Figura 7	*Call Graph* da Biblioteca *Wire*	21

LISTA DE TABELAS

Tabela 1	Condições de Verificação do Ficheiro *Wire.cpp*	39
Tabela 2	Condições de Verificação dos Ficheiros *twi.c* e *twi-volatile.c*	41

LISTA DE LISTAGENS

3.1 Classe *Rectangle* em Código C++ com Anotações em ACSL++ 16
3.2 Estrutura *Rectangle* em Código C com Anotações em ACSL 16
3.3 Método *difference()* em Código C++ com Anotações em ACSL++ 17
3.4 Função *difference()* em Código C com Anotações em ACSL 18
3.5 Estruturas para Representação de Métodos Virtuais 18

5.1 Axioma *physical_address* para Validação de Endereços Físicos de Memória . . 29
5.2 Implementação e Anotações do Método *write()* do Ficheiro *Wire.cpp* 30
5.3 Implementação e Anotações da Função *twi_transmit()* do Ficheiro *twi.c* 34
5.4 Implementação e Anotações da Função *twi_transmit()* do Ficheiro *twi-volatile.c* 38

ACRÓNIMOS

ACSL ANSI/ISO C Specification Language.

EVA Evolved Value Analysis.

FRAMA-C Framework for Modular Analysis of C.

I2C Inter-Integrated Circuit.

IDE Integrated Development Environment.

ISR Interrupt Service Routine.

PLC Programmable Logic Controller.

ROS Robotic Operating System.

SCL Serial Clock.

SDA Serial Data.

SPI Serial Peripheral Interface.

WP Weakest Precondition.

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Os *Programmable Logic Controllers (PLCs)* são dispositivos especialmente adaptados para sistemas de automação industrial, sendo frequentemente utilizados como controladores de segurança em sistemas críticos, uma vez que são fáceis de verificar e certificar devido à sua simplicidade. A aplicabilidade destes controladores é extensa e pode variar entre dispositivos muito básicos, que simplesmente lêem informação de um sensor e realizam uma ação, até dispositivos mais complexos, que envolvem várias componentes interligadas entre si. Os PLCs assumem particular relevância na área da robótica. Um exemplo tipicamente utilizado, é a aplicação de PLCs para analisar informações de um sensor que quando deteta humanos em redor desliga o robô, podendo implicar o corte de corrente elétrica, para que este não coloque em risco vidas humanas. De sublinhar que estes dispositivos funcionam de forma independente do software que controla o robô, que é normalmente muito complexo, o que faz com que sejam de facto eficientes.

A expansão contínua de sistemas de automação, em particular da robótica, para ambientes mais desestruturados e uma maior interação humana requer a existência de sistemas modernos altamente configuráveis e adaptáveis e, consequentemente, com modos de funcionamento flexíveis. Assim, são requisitados sistemas de controlo de segurança mais avançados e mais fáceis de configurar e de modificar. Está em curso uma mudança de paradigma que procura substituir os PLCs por microcontroladores do tipo Arduino [1] na implementação de controladores de segurança, uma vez que a utilização de PLCs para desenvolvimento destes sistemas é dificultada pelo poder de processamento limitado e pela dificuldade de programação dos mesmos.

O Arduino é uma plataforma eletrónica *open-source* que disponibiliza um microcontrolador possível de manipular para a construção de dispositivos interativos. A utilização de microcontroladores Arduino e afins acontece em diversas áreas, algumas das quais críticas e com elevados níveis de exigência. Deste modo, é fundamental alcançar a certificação dos *embedded systems* desenvolvidos com base nestes microcontroladores, em particular, a certificação do software neles presentes. De salientar que um *embedded system* representa

um sistema informático dentro de um sistema elétrico, utilizado para a automatização de tarefas seguindo um programa de computador.

1.2 MOTIVAÇÃO

Com a mudança de paradigmas, as funcionalidades começaram a ser desenvolvidas através de software devido à facilidade de implementação e alteração das mesmas, à medida que os requisitos vão sendo alterados. A realização de modificações em software é mais acessível, visto que não é necessária a construção de diferentes circuitos para aplicações específicas. Em contrapartida, o processo de verificação e certificação de software é bastante mais complicado do que a validação de PLCs.

Os Arduinos são dispositivos baratos e de fácil programação, sendo muito utilizados no desenvolvimento de projetos informais, pelo que não existiu até agora uma preocupação com a verificação e certificação de software desenvolvido para sistemas Arduino. Assim sendo, existe uma necessidade de estudar métodos e técnicas de verificação e certificação para tornar estes dispositivos mais previsíveis e seguros para que possam ser utilizados em sistemas críticos.

Tendo em conta o âmbito e duração desta dissertação, existiu a necessidade de escolher uma área de intervenção adequada, pelo que foram escolhidos os protocolos de comunicação devido à sua importância neste tipo de sistemas. Os *embedded systems* estão em constante comunicação com diversos periféricos, com o intuito de permutar a informação necessária à automatização das tarefas. Assim, a correção das bibliotecas de comunicação é essencial para conseguir a validação do software utilizado em sistemas Arduino.

As bibliotecas dos protocolos de comunicação *Inter-Integrated Circuit (I2C)* [2] e *Serial Peripheral Interface (SPI)* [3] são exemplos de bibliotecas constantemente utilizadas em *embedded systems*.

1.3 OBJETIVOS E CONTRIBUIÇÕES

Esta dissertação explora a aplicação de verificação formal de software a sistemas Arduino. Com esta abordagem, pretende-se aumentar a qualidade do software para que este possa, posteriormente, ser utilizado em sistemas críticos.

Em particular, um dos objetivos é a verificação funcional da biblioteca *Wire* [4] que possibilita aos dispositivos Arduino a aplicação do protocolo de comunicação I2C. O processo de verificação utilizado nesta dissertação recorre à análise estática de código baseada em técnicas de verificação dedutiva, sendo utilizada a *Framework for Modular Analysis of C (Frama-C)* [5] em conjunto com o *plug-in Weakest Precondition (WP)* [6]. Para isto, são acrescentadas à implementação da biblioteca *Wire* especificações escritas através da linguagem *ANSI/ISO C*

Specification Language (ACSL) [7] para, posteriormente, a ferramenta Frama-C desempenhar a correspondente verificação. A presente dissertação procura apresentar algumas características da biblioteca *Wire* para, posteriormente, demonstrar as especificações introduzidas e os resultados alcançados desta verificação, sendo referidas algumas particularidades e limitações genéricas e/ou específicas da implementação.

Como o Frama-C está apenas vocacionado para a verificação de código C e a biblioteca *Wire* é constituída por ficheiros de código C e C++, outro objetivo é a exploração do *plug-in* experimental Frama-Clang [8] para possibilitar a verificação de código C++ através do Frama-C. Para além disso, como este *plug-in* é bastante recente e, consequentemente, existe pouca documentação, são apresentados os conhecimentos adquiridos durante a exploração e a aplicação do Frama-Clang, com destaque para o processo de tradução de código C++ em código C e ainda para as especificações ACSL++. Apesar da utilização do *plug-in* Frama-Clang na verificação da biblioteca *Wire* ser por si só uma contribuição evidente desta dissertação, este estudo exploratório é ainda fundamental para a evolução da verificação de código C++. De sublinhar que embora exista uma escassez de ferramentas para este tipo de verificação, existe um interesse generalizado na certificação de código C++.

1.4 Organização do Documento

O Capítulo 2 disponibiliza os conceitos necessários para a leitura desta dissertação. Este começa com a apresentação da plataforma Arduino, dos protocolos de comunicação I2C e SPI e ainda da ferramenta de verificação Frama-C, terminando com a descrição breve de alguns casos de estudo onde é utilizada a verificação formal para validar e certificar o software de sistemas críticos.

No Capítulo 3 são retratados os conhecimentos adquiridos na exploração do *plug-in* Frama-Clang, com destaque para a exemplificação do processo de tradução de código C++ para código C. Para além disso, são demonstradas as especificações disponibilizadas pela linguagem ACSL++.

O Capítulo 4 apresenta algumas características importantes para a utilização da biblioteca *Wire*. Como a biblioteca está disponível através de uma interface em C++, primeiro são apresentados os métodos de C++ e depois as correspondentes funções de baixo nível implementadas em C. A ligação entre as camadas de C++ e C é demonstrada a partir de um *call graph* simplificado.

No Capítulo 5 é descrita a configuração do Frama-C para posterior verificação da biblioteca *Wire*. De seguida, são apresentadas algumas das anotações para retratar o trabalho desenvolvido e ainda as correspondentes limitações genéricas. A secção termina com a apresentação de uma análise aos resultados da verificação, sendo mencionados alguns problemas específicos e pormenores da implementação.

Para finalizar, o Capítulo 6 apresenta as conclusões acerca dos resultados e das limitações da verificação, assim como uma avaliação sobre a utilização do Frama-C e do *plug-in* Frama-Clang. Ainda são delineadas algumas contribuições futuras, sendo referidos exemplos que podem beneficiar dos resultados desta dissertação.

2

ESTADO DA ARTE

2.1 ARDUINO

O Arduino tem origens no projeto Wiring [9]. A primeira plataforma disponibilizada por este projeto suportava o microcontrolador ATmega128 [10], porém ainda não era suficientemente acessível para a maioria das pessoas. Com o intuito de alcançar uma plataforma mais razoável, foi adicionado suporte para o microcontrolador ATmega8 [11] e, consequentemente, replicado o código fonte do projeto Wiring, de onde resultou um projeto separado denominado por Arduino [12].

O Arduino é uma plataforma eletrónica *open-source* utilizada para a construção de dispositivos interativos independentes ou colaborativos que executam certas aplicações. As plataformas Arduino são capazes de obter diversos *inputs* a partir de sensores ou outros dispositivos e, posteriormente, determinar os respetivos *outputs*, sendo definido um conjunto de instruções para delinear o comportamento destas plataformas, com recurso à linguagem de programação Arduino. À semelhança do projeto Wiring, o projeto Arduino foi criado no *Interaction Design Institute Ivrea* como uma ferramenta de prototipagem rápida concebida para iniciantes e profissionais, sendo utilizado em objetos do quotidiano e instrumentos científicos complexos. As plataformas Arduino são inteiramente *open-source*, por forma a possibilitar aos utilizadores a adaptação destas plataformas às suas necessidades e, para além disso, assegurar a existência de uma enorme quantidade de conhecimento disponibilizado pelas contribuições da comunidade.

Geralmente, o termo Arduino é tanto utilizado para a linguagem de programação como para o dispositivo físico, contudo esta distinção deverá ser clara a partir do seguinte contexto: um dispositivo Arduino executa código da linguagem de programação Arduino, por sua vez, o código Arduino pode ser escrito, compilado e carregado para o dispositivo a partir do Arduino *Integrated Development Environment (IDE)* [13].

O projeto Arduino disponibiliza várias plataformas independentes com características diferentes, sendo considerada para a presente dissertação apenas a plataforma denominada por Uno [14], visto que é um modelo simples e acessível. A plataforma Uno, que está retratada na Figura 1, contém um microcontrolador ATmega328P [15], detém 14 pinos digitais

de *input/output* e 6 pinos analógicos de *input*, assim como todos os componentes essenciais para um correto funcionamento do microcontrolador e o estabelecimento de comunicação com um computador. De salientar que os pinos são o mecanismo utilizado para troca de informações entre dispositivos. O microcontrolador ATmega328P suporta os protocolos de comunicação I2C/TWI e SPI, sendo disponibilizados pelo Arduino IDE através das bibliotecas *Wire* [4] e *SPI* [16], respetivamente.

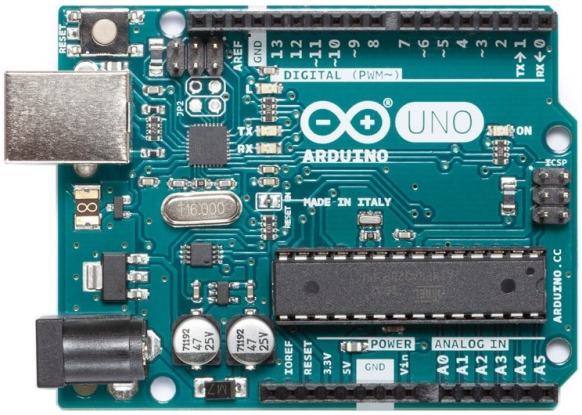


Figura 1: Plataforma Arduino Uno

O projeto Arduino disponibiliza um IDE, baseado em Processing [17], que permite aos utilizadores escrever programas para a plataforma Arduino, com recurso à linguagem de programação Arduino, baseada em Wiring. Os programas escritos com o Arduino IDE são denominados *sketches* e têm a extensão de ficheiro *.ino* [18]. Um *sketch* necessita apenas das funções *setup()* e *loop()* para compilar corretamente. A função *setup()* é invocada apenas no início da execução do *sketch* depois da plataforma Arduino iniciar, sendo utilizada para inicializar variáveis e bibliotecas, e definir os modos dos pinos. A seguir à função *setup()*, a função *loop()* é executada repetidamente, por forma a controlar ativamente o comportamento do dispositivo Arduino [19]. A Figura 2 apresenta a interface do Arduino IDE com um exemplo denominado *Blink*, cujo único objetivo é ligar e desligar repetidamente a LED presente no dispositivo Arduino.

Para o dispositivo Arduino proceder à execução de um *sketch*, precisam de ocorrer operações intermédias de pré-processamento, compilação e carregamento para o dispositivo. Inicialmente, é efetuado um pré-processamento para converter o código do *sketch* escrito na linguagem Arduino num programa C, sendo este compilado pelo *avr-gcc*. De seguida, o código é combinado com as bibliotecas *standard* do Arduino, por forma a produzir um único ficheiro *Intel HEX*. Com isto, o ficheiro resultante é carregado para o dispositivo através da ligação USB, com recurso à ferramenta *avrdude* [20].



Figura 2: Arduino IDE - *Blink Example*

2.2 PROTOCOLOS DE COMUNICAÇÃO

Para evitar a utilização de protocolos extremamente complexos e sofisticados, foram desenvolvidos os protocolos de comunicação I2C e SPI. Estes são maioritariamente adequados para comunicações entre circuitos integrados como, por exemplo, a comunicação entre um microcontrolador e um conjunto de periféricos.

O protocolo I2C [2], cuja topologia é retratada na Figura 3, representa um mecanismo de comunicação síncrona com arquitetura *multi-master/multi-slave* e ligações *half-duplex*, onde a transmissão e receção de informação ocorre alternadamente. Independentemente do número de dispositivos ligados ao canal de comunicação, o protocolo I2C requer apenas duas linhas denominadas *Serial Data (SDA)* e *Serial Clock (SCL)*, sendo todos os dispositivos identificados com um endereço único de 7-bits. De salientar que é possível expandir estes endereços para 10-bits. Contudo, este protocolo disponibiliza apenas velocidades de 10 Kbps, 100 Kbps, 400 Kbps, 1 Mbps e 3.4 Mbps. Por forma a permutar informação, o *master* envia uma condição de *start* e um endereço com indicação de leitura ou escrita, que é verificado por todos os dispositivos. Caso exista correspondência de endereço, o dispositivo responde com *acknowledgement*, caso contrário, o dispositivo apenas ignora e aguarda uma

condição de *stop*. Quando o *master* recebe o *acknowledgement*, este inicia a transmissão ou receção de informação e submete uma condição de *stop* ao terminar, por forma a assinalar a libertação do canal de comunicação.

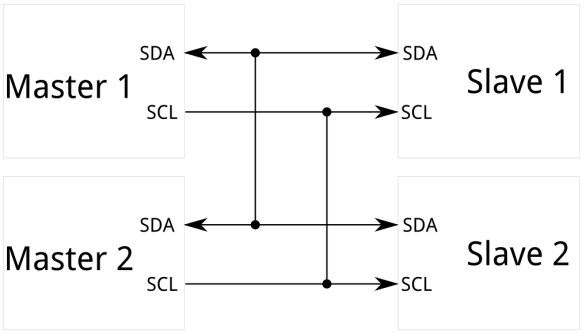


Figura 3: Topologia do Protocolo de Comunicação I2C

Por sua vez, o protocolo SPI [3], cuja topologia é apresentada na Figura 4, descreve uma comunicação síncrona com arquitetura *single-master/multi-slave* e ligações *full-duplex*, sendo permitida a comunicação simultânea em ambas as direções. O protocolo SPI requer a utilização de, pelo menos, quatro linhas denominadas MISO, MOSI, SCK e SS. Quando o *master* pretende transmitir informação, este seleciona um dispositivo ao atribuir o valor *LOW* à respetiva linha SS. Caso contrário, a linha SS permanece *HIGH*, por forma a indicar ao dispositivo para ignorar o *master*. O protocolo SPI disponibiliza quatro modos de comunicação definidos através dos parâmetros *CPOL* e *CPHA*, sendo necessário um par *master/slave* utilizar os mesmos parâmetros, pelo que *slaves* com configurações diferentes requisitam a reconfiguração do *master* sempre que comunica com um *slave* diferente. De salientar que este protocolo não impõe uma velocidade máxima, pelo que consegue alcançar velocidades superiores a 10 Mbps.

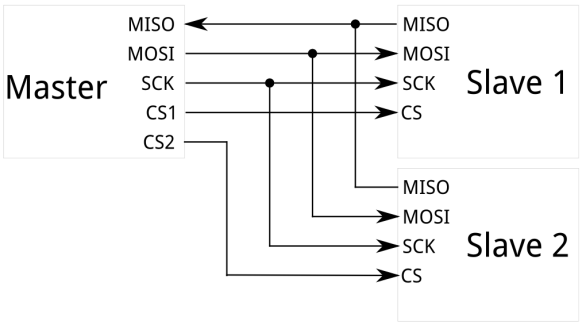


Figura 4: Topologia do Protocolo de Comunicação SPI

Com isto, os protocolos de comunicação supracitados são opções populares a utilizar em qualquer *embedded system*. O protocolo I2C é mais adequado para aplicações que reque-

rem uma arquitetura *multi-master*, enquanto o protocolo SPI é preferível em aplicações que requisitam a transferência de informação.

2.3 VERIFICAÇÃO FORMAL DE SOFTWARE

Por forma a verificar a correção de programas, é prática comum recorrer a teste de software, porém este mecanismo não garante a ausência de erros.

> *Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.* (Edsger Dijkstra, 1972)

A verificação formal de software consiste na aplicação de métodos formais durante o desenvolvimento de software. A aplicação de métodos de verificação formal é uma prática recomendada quando o nível de exigência da correção de código é elevado como, por exemplo, quando o software utilizado pode colocar em risco vidas humanas ou valores monetários elevados.

Uma abordagem possível na verificação formal de software é a verificação dedutiva que representa o processo de geração de um conjunto de obrigações de prova matemática, cuja validade demonstra a conformidade do programa com a respetiva especificação [21, 22]. A demonstração destas obrigações de prova é realizada através de *interactive theorem provers* [23, 24], *automatic theorem provers* [25] e *satisfiability modulo theories solvers* [26]. A principal desvantagem desta abordagem é a necessidade do utilizador compreender minuciosamente o funcionamento do programa assim como a metodologia de verificação, por forma a anotar as especificações de verificação das funções.

Outra abordagem possível é a utilização de *software model checking* que se caracteriza pela exploração exaustiva dos estados e das transições presentes no modelo matemático [27]. Ao contrário da abordagem supracitada, esta técnica de verificação é muitas vezes automática, porém existe a possibilidade da ocorrência de explosão do espaço de estados. Uma primeira solução para este problema é a utilização de *software model checking* baseado em interpretação abstrata [28], tendo esta a desvantagem de introduzir falsos positivos, ou seja, às vezes a ferramenta pode indicar erros impossíveis de ocorrer no sistema real. Uma segunda solução é a aplicação de *bounded model checking* de software [29], cuja a desvantagem é não capturar o comportamento completo do sistema, pelo que podem existir erros que nunca serão detetados.

2.4 FRAMA-C

O Frama-C é uma plataforma *open-source* que disponibiliza um conjunto de ferramentas dedicadas à análise estática de código fonte escrito na linguagem C [5, 30, 31].

A plataforma Frama-C está organizada numa arquitetura modular, isto é, os *plug-ins* correspondentes aos analisadores estabelecem uma ligação com um *kernel* comum, cujo objetivo é centralizar a informação e coordenar a análise. A arquitetura retratada contribui para a robustez do Frama-C, enquanto que possibilita um vasto leque de funcionalidades. Os principais *plug-ins* disponibilizados pelo Frama-C são *Evolved Value Analysis (EVA)*, Jessie [32] e WP [33]. O *plug-in* EVA determina um conjunto de possíveis valores para cada variável do programa, através de técnicas de interpretação abstrata. Por sua vez, os *plug-ins* Jessie, que está descontinuado, e WP permitem demonstrar a conformidade das funções com as propriedades especificadas, com recurso a técnicas de verificação dedutiva. Para além destes, existem diversos *plug-ins* adicionais como, por exemplo, Impact Analysis, Slicing, Mthread e Frama-Clang [34].

A verificação formal de software para sistemas críticos pode ser realizada através da plataforma Frama-C, visto que esta permite verificar a conformidade de uma implementação com uma especificação formal. As propriedades funcionais são escritas pelos utilizadores com recurso à linguagem de especificação ACSL e utilizadas pelas várias ferramentas de verificação estática. A linguagem ACSL é uma linguagem de especificação comportamental baseada na noção de contratos das funções, ou seja, os utilizadores precisam de especificar as pré-condições/pós-condições a satisfazer antes/depois da invocação das funções, respetivamente [7, 35].

A computação da pré-condição mais fraca pretende reduzir os problemas de verificação dedutiva à demonstração da validade das fórmulas de primeira ordem denominadas condições de verificação [36]. Com isto, é possível a utilização de *automatic theorem provers* como, por exemplo, Alt-Ergo [37] e Z3 [38] para a demonstração das condições de verificação produzidas pelos *plug-ins* Jessie e WP, tal como é retratado na Figura 5. Quando uma condição de verificação é desconhecida pode ser devido a uma de duas razões. Ou a condição de verificação está errada, isto é, o comportamento do programa não está de acordo com a especificação, ou o *automatic theorem prover* não é capaz de provar a condição de verificação. Na segunda hipótese, é possível recorrer a um *interactive theorem prover*.

2.5 TRABALHO RELACIONADO

Esta secção descreve brevemente alguns casos de estudo onde é aplicada a ferramenta Frama-C para a validação e certificação de software de sistemas críticos como, por exemplo, indústria aeroespacial e indústria bancária. De salientar que o caso de estudo da indústria bancária é o mais relevante para a presente dissertação, uma vez que utiliza a ferramenta Frama-C para a certificação de um dispositivo de assinaturas digitais desenvolvido a partir de uma plataforma Arduino.

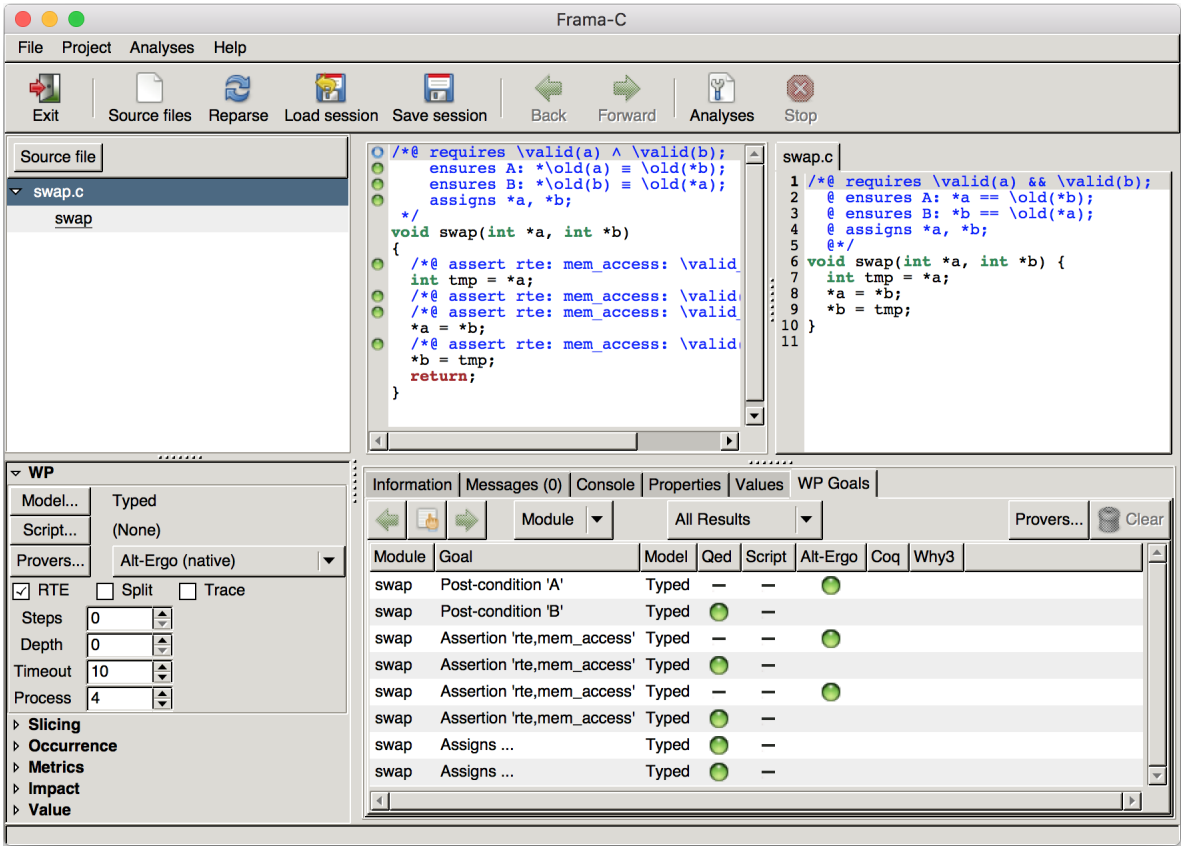


Figura 5: Frama-C - *Swap Example*

DISPOSITIVO DE ASSINATURAS DIGITAIS A utilização do *malware* denominado *High Roller* para a realização de ataques automáticos às transações bancárias dos clientes causou enormes prejuízos, pelo que a agência europeia de cibersegurança emitiu um aviso e uma chamada para ação contra esses ataques. A evolução dos *smartphones* motiva a realização de transações bancárias online pelos clientes, porém estes continuam vulneráveis. Com isto, a solução proposta por Trommler [39] é a criação de um dispositivo de assinaturas seguro a utilizar em conjunto com o *smartphone*, por forma a permitir a produção de assinaturas digitais para as transações verificadas e aprovadas pelo utilizador, sendo possível comprovar a veracidade das informações financeiras. De salientar que este artigo assume um cenário onde o *smartphone* está comprometido e totalmente controlado pelo atacante, pelo que a informação apresentada não é confiável e a informação introduzida é submetida para o atacante. O protótipo do dispositivo de segurança é baseado no *smartphone Jolla* que disponibiliza uma contracapa substituível e possível de equipar com um dispositivo inteligente. Para o desenvolvimento do dispositivo de segurança é utilizada a plataforma Arduino, sendo a ligação com o *smartphone* estabelecida através do protocolo de comunicação I2C. Os requisitos são limitados ao mínimo, por forma a reduzir a superfície de ataque. Contudo, a segurança está dependente da correta implementação do software, sendo o ob-

jetivo evitar a disponibilização de funcionalidades indesejadas, a partir das quais é possível comprometer a integridade do dispositivo de segurança. Assim, a verificação destas funções foi realizada através da ferramenta Frama-C, pelo que foram especificados contratos que representam os seus possíveis comportamentos. Estes contratos foram verificados com recurso ao *plug-in* WP e as condições de verificação demonstradas pelo Alt-Ergo.

BIBLIOTECA KLIBC Com o desenvolvimento das ferramentas de análise estática, é expectável uma exploração progressiva destas ferramentas para verificação de código fonte. Contudo, existe uma ausência de documentação a demonstrar a correção de código de sistemas reais, visto que as empresas que utilizam estas ferramentas não têm interesse em publicar documentação sobre as suas investigações. Devido à existência de diversas aplicações amplamente utilizadas que dependem de bibliotecas *standard*, a verificação estática do código destas bibliotecas apresenta uma enorme relevância. Posto isto, Carvalho et al. [40] apresenta os resultados alcançados pela verificação formal da biblioteca *kLIBC*, com recurso ao *plug-in* WP da plataforma Frama-C, sendo demonstrada a aplicabilidade da verificação de software em código de sistemas reais.

SOFTWARE AERONÁUTICO O software aeronáutico de segurança crítica representa um domínio de aplicação natural da verificação formal. Como explicado por Dordowsky [41], é possível averiguar os resultados de um estudo experimental acerca da aplicação da verificação formal de software num projeto aeronáutico real, cujo objetivo principal era analisar a viabilidade de expressar formalmente os requisitos de um projeto real. Por outras palavras, a partir de um conjunto de critérios predefinidos, o objetivo era verificar a possibilidade de utilização dos métodos formais na indústria, sendo considerada a experiência adquirida pela formalização da especificação de uma funcionalidade de software real. O processo de formalização dos requisitos foi realizado através da linguagem ACSL e o código fonte foi parcialmente verificado pelo Frama-C, em conjunto com o *plug-in* WP. Através deste estudo foi possível concluir que a formalização dos requisitos representa uma ótima ferramenta para analisar as especificações e, consequentemente, melhorar a qualidade do código fonte.

SOFTWARE AEROESPACIAL O software aeroespacial é extremamente importante, sendo necessária uma verificação rigorosa para assegurar a satisfação dos seus requisitos e, consequentemente, prevenir a ocorrência de acidentes [42]. Contudo, a sua certificação é uma tarefa desafiante devido à complexidade resultante das restrições em tempo real. Os testes e as simulações permitem apenas uma verificação parcial do comportamento destes sistemas críticos, sendo recomendada a utilização de verificação formal de software como método complementar. Com isto, Busquim e Silva et al. [43] explora duas abordagens de verifica-

ção formal de software denominadas interpretação abstrata e verificação dedutiva, sendo utilizados os *plug-ins* EVA e Jessie do Frama-C, respetivamente. Os resultados adquiridos demonstram que ambas as abordagens desempenham um papel importante na verificação de software aeroespacial.

FERRAMENTA IKOS Por forma a reduzir os custos e aumentar a eficiência dos processos de verificação de sistemas críticos de voo, a NASA introduziu ferramentas para as diferentes etapas de desenvolvimento de software [44]. Em particular, a NASA disponibilizou uma ferramenta *open-source* denominada IKOS que permite realizar a análise estática de código C/C++, com recurso ao Frama-C. A ferramenta IKOS foi desenvolvida para a verificação de *flight critical embedded systems*, sendo considerados para a sua validação controladores utilizados por veículos aéreos não tripulados como, por exemplo, o *AeroQuad* e o *DIY AutoPilot*. De salientar que o *AeroQuad* utiliza microcontroladores Arduino, enquanto os sistemas do *DIY AutoPilot* incluem o *ArduPilot*.

3

TUTORIAL DE FRAMA-CLANG

O presente capítulo contém uma introdução ao *plug-in* Frama-Clang, uma vez que este *plug-in* é bastante recente e, consequentemente, existe pouca documentação. Por esta razão, a utilização do Frama-Clang exigiu um estudo exploratório, sendo aqui retratados os conhecimentos adquiridos durante o processo de aprendizagem, para além deste *plug-in* ser utilizado nos capítulos seguintes, existe um interesse generalizado em explorar ferramentas para verificação de código C++.

O capítulo é constituído pelas secções *Plug-in Frama-Clang* e *Exemplo de Utilização*. A secção *Plug-in Frama-Clang* apresenta algumas características do Frama-Clang como, por exemplo, as vantagens de utilização, a constituição do *plug-in* e a tradução de código C++ para código C. Enquanto a secção *Exemplo de Utilização* apresenta alguns fragmentos de código e as correspondentes anotações para esclarecer a tradução realizada pelo Frama-Clang, sendo o exemplo *Rectangle.cpp* disponibilizado em `https://bitbucket.org/rafalexbar/arduino`. De sublinhar que é assumido o conhecimento prévio dos conceitos básicos das linguagens C, C++ e ACSL.

3.1 PLUG-IN FRAMA-CLANG

O Frama-C representa uma plataforma dedicada à verificação de programas escritos na linguagem C. Assim sendo, é estudado o *plug-in* experimental Frama-Clang [8, 45, 46] para possibilitar a verificação de código C++ como, por exemplo, ficheiros com extensão *.cpp*, *.c++*, *.C*, *.cxx*, *.cc* e *.ii*.

O Frama-Clang utiliza o Clang para produzir uma AST que, seguidamente, atravessa um processo de simplificação e reordenação para originar uma AST intermédia corretamente estruturada. A partir desta AST intermédia é realizada a tradução de código C++ para C, sendo depois utilizado o Frama-C *standard* para realizar a verificação.

O *plug-in* Frama-Clang é acompanhado da linguagem de especificação ACSL++ que suporta as habituais funcionalidades disponibilizadas pelo ACSL e ainda a anotação de propriedades relativas aos objetos característicos de código C++. Por outras palavras, o ACSL++

permite a especificação de contratos de funções, asserções e anotações de ciclos, para além de permitir complementar as classes com predicados, funções lógicas e invariantes.

A tradução de código C++ para código C representa a conversão das classes em estruturas de dados, onde os atributos e os métodos característicos destas classes são interpretados como variáveis de estrutura e um conjunto de funções, respetivamente. As funções produzidas detêm um argumento *this* predefinido que representa um apontador para a correspondente estrutura de dados. De salientar que o Frama-Clang utiliza um conjunto de especificações provenientes do *Itanium C++ ABI* [47] durante a descodificação de nomes para representar todos os símbolos em C++ com designações inequívocas e válidas em C.

A linguagem C++ permite a definição de métodos virtuais, isto é, a declaração de funções hereditárias e substituíveis, por forma a alcançar o polimorfismo característico da programação orientada aos objetos. Assim sendo, durante a tradução de código pelo Frama-Clang ocorre a criação de uma tabela de métodos virtuais para cada uma das classes existentes, sendo os métodos virtuais representados através de apontadores para funções.

Apesar da necessidade de aperfeiçoar as funcionalidades disponibilizadas como, por exemplo, expandir as especificações possíveis em ACSL++ e finalizar a manipulação dos métodos virtuais, o Frama-Clang compreende as principais funcionalidades do código C++ e suporta as anotações essenciais em ACSL++, sendo praticável a verificação de programas com dimensão moderada. Em suma, o Frama-Clang representa um *plug-in* de grande importância devido ao interesse generalizado em certificar código C++.

3.2 Exemplo de Utilização

Com o objetivo de melhor compreender o *plug-in* Frama-Clang, isto é, a tradução de código C++ para código C e as especificações disponibilizadas pela linguagem ACSL++, são esclarecidos alguns fragmentos ilustrativos de código e as correspondentes anotações.

A Listagem 3.1 apresenta a declaração da classe *Rectangle* que detém dois atributos privados denominados *width* e *height*. Para além disso, esta classe detém um construtor parametrizado e três métodos públicos denominados *area()*, *difference()* e *undefined()*. A linguagem ACSL++ permite a anotação de invariantes de classe, sendo apontado o invariante *positive* para garantir que ambos os atributos da classe *Rectangle* têm sempre um valor positivo. Contudo, a ferramenta Frama-C apresenta a seguinte mensagem a avisar que o invariante *positive* é ignorado porque ainda não é verificado pelo *plug-in* WP, apesar do *plug-in* Frama-Clang interpretar corretamente este invariante.

```
Rectangle.cpp:3:[wp] warning:
  Type invariant not handled yet ('_ZN9RectangleE8positive' ignored).
```

Para além disso, a linguagem ACSL++ suporta a declaração de predicados associados às classes como, por exemplo, o predicado *bigger* cujo objetivo é comparar as áreas de dois retângulos adquiridos como parâmetros.

```
1  class Rectangle {
2    /*@ class invariant positive =
3      @   width > 0 && height > 0;
4      @ predicate bigger(Rectangle * r1, Rectangle * r2) =
5      @   r1->width * r1->height > r2->width * r2->height;
6      @*/
7    private:
8      int width, height;
9    public:
10     Rectangle(int, int);
11     int area(void);
12     int difference(Rectangle * r);
13     virtual int undefined(void);
14 };
```

Listagem 3.1: Classe *Rectangle* em Código C++ com Anotações em ACSL++

```
1  struct Rectangle {
2    struct _frama_c_vmt * pvmt;
3    int width, height;
4  };
5  /*@ type invariant positive(struct Rectangle \this) =
6    @   \this.width > 0 /\ \this.height > 0;
7    @ predicate bigger{L}
8    @   (struct Rectangle \this, struct Rectangle * r1, struct Rectangle * r2) =
9    @   \at(r1->width * r1->height > r2->width * r2->height, L);
10   @*/
11 void Rectangle::Ctor(struct Rectangle * this, int w, int h);
12 int area(struct Rectangle * this);
13 int difference(struct Rectangle * this, struct Rectangle * r);
14 int undefined(struct Rectangle * this);
```

Listagem 3.2: Estrutura *Rectangle* em Código C com Anotações em ACSL

A Listagem 3.2 apresenta o resultado de traduzir a classe *Rectangle* através do *plug-in* Frama-Clang. A tradução da classe *Rectangle* representa a declaração da estrutura *Rectangle* constituída pelas variáveis *width*, *height* e *pvmt*, sendo a variável *pvmt* um apontador para a correspondente tabela de métodos virtuais.

A tradução dos métodos da classe *Rectangle* corresponde à criação de funções cujo primeiro argumento representa um apontador *this* predefinido, por forma a possibilitar o

acesso às variáveis da estrutura *Rectangle*. Por outro lado, a tradução dos invariantes e predicados de classe compreende a introdução do objeto *\this* como argumento.

```
1  /*@ requires
2    @      0 < width  <= 10
3    @   && 0 < height <= 10
4    @   && 0 < r->width  <= 10
5    @   && 0 < r->height <= 10
6    @   && \valid(r)
7    @   && bigger(this, r);
8    @ assigns
9    @   \nothing;
10   @ ensures
11   @   \result == \old(width) * \old(height) - \old(r->width) * \old(r->height);
12   @*/
13 int Rectangle::difference(Rectangle * r) {
14   return area() - r->area();
15 }
```

Listagem 3.3: Método *difference()* em Código C++ com Anotações em ACSL++

A Listagem 3.3 apresenta a implementação do método *difference()* e as correspondentes anotações em ACSL++. O método *difference()* permite calcular a diferença entre as áreas de dois retângulos, com recurso à invocação do método *area()* sobre ambos os retângulos.

A pré-condição do método *difference()* compreende a informação relevante para satisfazer a pré-condição do método *area()*, ou seja, contém condições sobre os intervalos de valores válidos dos parâmetros *width* e *height* dos dois retângulos. Para garantir a utilização de apontadores com endereços de memória válidos, é acrescentada à pré-condição do método *difference()* uma condição de validação relativa ao retângulo recebido como argumento. Para além disso, é requisitada a validação do predicado *bigger* de modo a garantir um resultado sempre positivo. Nenhuma variável é modificada durante a execução do método *difference()*, visto que apenas é determinada a diferença entre os valores das áreas de dois retângulos, sendo estes valores obtidos através do método *area()*, que também não altera qualquer variável. O ACSL++ não possibilita a invocação de métodos, sendo impossível a consideração de uma pós-condição simplificada, ou seja, para retratar o resultado do método *difference()* é necessário substituir as invocações do método *area()* pelas correspondentes operações.

A Listagem 3.4 demonstra o resultado de traduzir o método *difference()* através do *plug-in* Frama-Clang. A tradução do método *difference()* representa a criação da função *difference()*, sendo acrescentado um argumento adicional correspondente ao apontador *this* predefinido. O processo de tradução introduz automaticamente uma condição de validação, por forma a garantir um endereço de memória válido.

A verificação das condições de *overflow* relativas à operação de multiplicação depende dos valores dos atributos *width* e *height* definidos pela pré-condição. Por outro lado, a verificação das condições de *overflow* correspondentes à operação de subtração depende das pós-condições que retratam os resultados das áreas.

```
/*@ requires \valid(this);
  @ requires
  @   0 < this->width ≤ 10 ⋀ 0 < this->height ≤ 10 ⋀
  @   0 < r->width ≤ 10 ⋀ 0 < r->height ≤ 10 ⋀ \valid(r) ⋀
  @   bigger(* this, this, r);
  @ ensures
  @   \result ≡
  @   \old(this->width) * \old(this->height) -
  @   \old(r->width) * \old(r->height);
  @ assigns \nothing;
  @*/
int difference(struct Rectangle * this, struct Rectangle * r) {
  int tmp = area(this);
  int tmp_0 = area(r);
  /*@ assert rte: signed_overflow: -2147483648 ≤ tmp - tmp_0; */
  /*@ assert rte: signed_overflow: tmp - tmp_0 ≤ 2147483647; */
  int __retres = tmp - tmp_0;
  return __retres;
}
```

Listagem 3.4: Função *difference()* em Código C com Anotações em ACSL

```
struct _frama_c_rtti_name_info_node _frama_c_rtti_name_info =
  { .name                   = "Rectangle",
    .base_classes           = (struct _frama_c_rtti_name_info_content *) 0,
    .number_of_base_classes = 0,
    .pvmt                   = & _frama_c_vmt_header };
struct _frama_c_vmt _frama_c_vmt_header =
  { .table      = _frama_c_vmt,
    .table_size = 1,
    .rtti_info  = & _frama_c_rtti_name_info };
struct _frama_c_vmt_content _frama_c_vmt[1] =
  { { .method_ptr = (void (*)(void))(& undefined), .shift_this = 0 } };
```

Listagem 3.5: Estruturas para Representação de Métodos Virtuais

A Listagem 3.5 apresenta as estruturas produzidas pelo Frama-Clang para a representação dos métodos virtuais da classe *Rectangle*. Com isto, para a classe *Rectangle* é produzida a estrutura *_frama_c_rtti_name_info* que demonstra a ausência de classes superiores e referencia a estrutura *_frama_c_vmt_header* que, por sua vez, apresenta uma tabela de métodos

virtuais denominada *_frama_c_vmt* apenas constituída por um apontador para função correspondente ao método virtual *undefined()*.

Através destas anotações em ACSL++ são satisfeitas as condições de verificação relativas à classe *Rectangle* e respetivos métodos, com exceção de muitas das condições de verificação correspondentes à representação de métodos virtuais, que demonstra a necessidade de aperfeiçoar esta funcionalidade. A partir da Figura 6 são demonstrados os resultados das condições de verificação do método *difference()*.

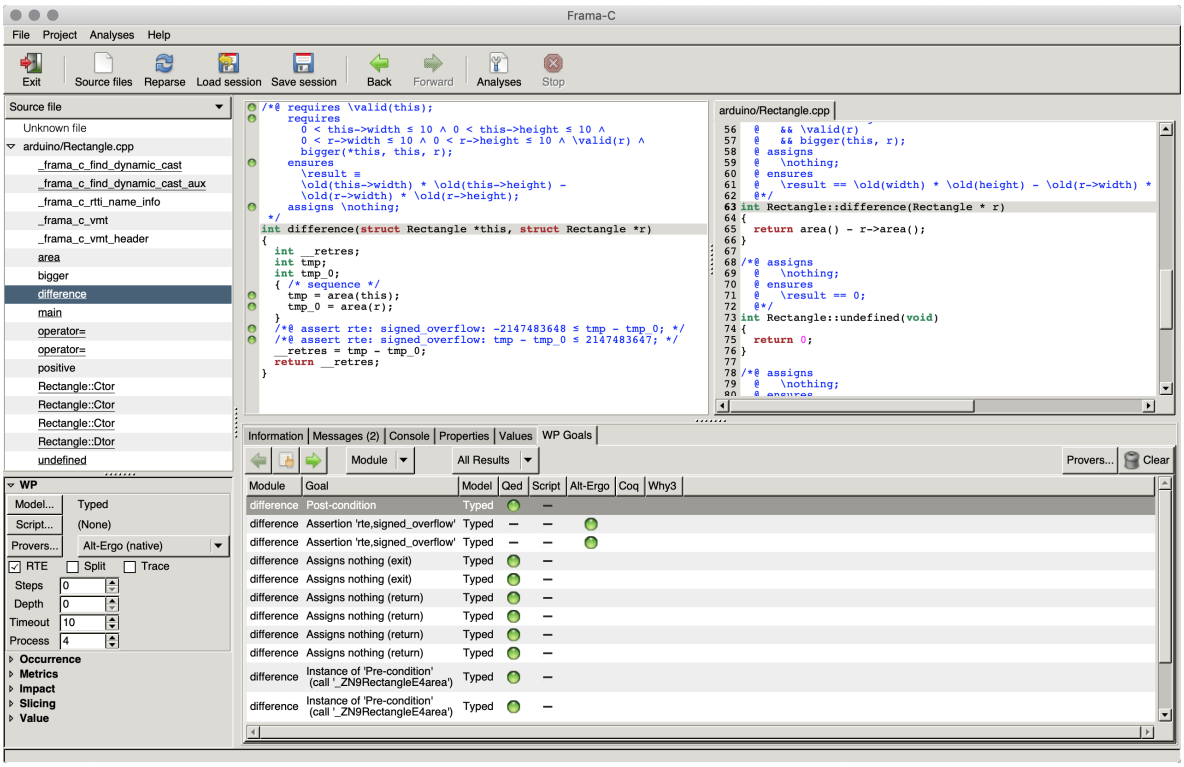


Figura 6: Condições de Verificação do Método *difference()*

4

CONHECIMENTO DA BIBLIOTECA WIRE

O presente capítulo apresenta uma introdução à biblioteca *Wire* para permitir a obtenção do conhecimento necessário à utilização do protocolo de comunicação I2C em dispositivos Arduino. O capítulo é constituído por três secções complementares denominadas *Biblioteca Wire*, *Ficheiro de Código C++* e *Ficheiro de Código C*. A secção *Biblioteca Wire* sublinha algumas características importantes desta biblioteca e, seguidamente, apresenta um *call graph* para demonstrar a ligação entre os métodos e as funções pertencentes aos ficheiros de C/C++ que compõe a biblioteca. Enquanto as secções *Ficheiro de Código C++* e *Ficheiro de Código C* descrevem, sucintamente, os métodos de C++ e as funções de C fundamentais à utilização desta biblioteca, respetivamente.

4.1 BIBLIOTECA WIRE

A biblioteca *Wire* [4, 48, 49] desempenha um papel fundamental, uma vez que possibilita a utilização do protocolo de comunicação I2C em dispositivos Arduino. Para aplicação do protocolo de comunicação I2C são apenas necessárias as ligações SDA e SCL disponibilizadas pelos pinos A4 (SDA) e A5 (SCL) da plataforma Arduino Uno. Os dispositivos participam numa comunicação I2C como dispositivos *master* ou *slave*. Os dispositivos *master* requisitam informação aos dispositivos *slave* e ainda controlam a velocidade da comunicação, enquanto os dispositivos *slave* apenas respondem aos pedidos dos dispositivos *master*. Assim sendo, ambos conseguem transmitir informação pelo canal de comunicação, porém a informação transmitida é sempre controlada pelos dispositivos *master*.

Uma característica importante da biblioteca *Wire* é a manipulação de apenas endereços de 7-bits, apesar dos endereços 0 a 7 estarem reservados. Para além disso, qualquer transmissão detém um limite máximo de 32 *bytes* de informação devido à dimensão dos *buffers* da biblioteca *Wire*, sendo descartados todos os *bytes* adicionais presentes numa transmissão. De acrescentar que apesar das operações do microcontrolador serem controladas através de interrupções, sendo possível uma aplicação continuar a executar enquanto ocorre a comunicação, as operações implementadas pela biblioteca *Wire* são bloqueantes, ou seja, as aplicações são suspendidas durante a utilização do canal de comunicação I2C.

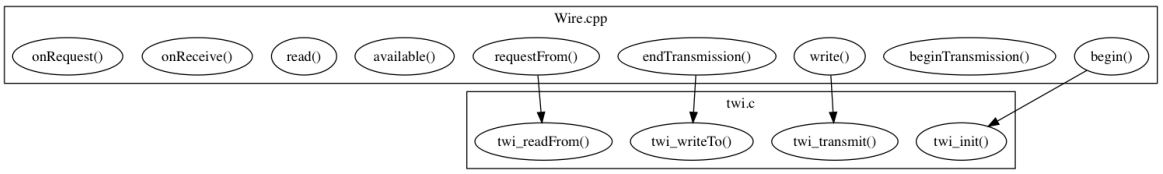


Figura 7: *Call Graph* da Biblioteca *Wire*

A biblioteca *Wire* é constituída por um ficheiro escrito em código C++ denominado *Wire.cpp* e outro ficheiro em código C designado *twi.c*, sendo ambos acompanhados pelo correspondente *header file*. Através da Figura 7 é apresentado o *call graph* da biblioteca *Wire*, onde estão retratados os métodos indispensáveis do ficheiro *Wire.cpp* e as respetivas invocações de funções do ficheiro *twi.c*. De sublinhar que os métodos esclarecidos na secção *Ficheiro de Código C++* estão ordenados segundo uma utilização exemplo desta biblioteca.

4.2 FICHEIRO DE CÓDIGO C++

A presente secção esclarece os principais métodos do ficheiro *Wire.cpp*, sendo demonstradas as assinaturas disponibilizadas e ainda clarificados os parâmetros correspondentes.

```
void begin(void);
void begin(uint8_t address);
void begin(int address);
```

O método *begin()* inicia o protocolo de comunicação I2C e acrescenta o dispositivo Arduino ao canal de comunicação. Caso seja especificado um *address* de 7-bits, o Arduino é adicionado como dispositivo *slave*, caso contrário é introduzido como dispositivo *master*. O método *begin()* não devolve um resultado.

```
void beginTransmission(uint8_t address);
void beginTransmission(int address);
```

Através do método *beginTransmission()* é estabelecido o dispositivo *slave* para qual será transmitida a informação, sendo necessária a especificação do correspondente parâmetro *address* de 7-bits. O método *beginTransmission()* também não devolve um resultado.

```
size_t write(uint8_t data);
size_t write(const uint8_t * data, size_t quantity);
```

O método *write()* permite ao dispositivo *slave* transmitir a informação requisitada pelo dispositivo *master* ou possibilita ao dispositivo *master* escrever a informação num *buffer* para, posteriormente, transmitir ao dispositivo *slave*. O método *write()* aceita como parâmetros

um único *byte* ou um conjunto de *bytes* acompanhado pela quantidade de *bytes* existentes nesse conjunto. De salientar que as transmissões detêm um limite máximo de 32 *bytes* de informação devido aos tamanhos dos *buffers* da biblioteca *Wire*. Para além disso, este método devolve o número de *bytes* escritos com sucesso.

```
uint8_t endTransmission(void);
uint8_t endTransmission(uint8_t sendStop);
```

Com recurso ao método *endTransmission()* é comunicada ao dispositivo *slave* a informação escrita, anteriormente, pelo dispositivo *master* num *buffer*. Caso o valor do argumento *sendStop* seja *true*, é remetida uma mensagem de paragem para libertar o canal de comunicação depois da transmissão. Caso este argumento seja *false*, é enviada uma mensagem para reiniciar após a transmissão, com o intuito de permitir ao dispositivo *master* diversas transmissões. De salientar que o valor predefinido deste argumento é *true*. O método *endTransmission()* devolve um *byte* que caracteriza o estado da transmissão cujo o valor informa sucesso ou a ocorrência de erro.

```
uint8_t requestFrom(uint8_t address, uint8_t quantity);
uint8_t requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop);
uint8_t requestFrom(int address, int quantity);
uint8_t requestFrom(int address, int quantity, int sendStop);
```

O método *requestFrom()* permite requisitar ao dispositivo *slave* uma determinada quantidade de informação. Os argumentos *address* e *quantity* são utilizados para especificar o endereço de 7-bits do dispositivo *slave* e a quantidade de informação a requisitar, respetivamente. Caso o valor do argumento *sendStop* seja *true*, é remetida uma mensagem de paragem para libertar o canal de comunicação depois do pedido. Caso este argumento seja *false*, é enviada uma mensagem para reiniciar após o pedido, com o intuito de permitir ao dispositivo *master* diversos pedidos de informação. De salientar que o valor predefinido deste argumento é *true*. O método *requestFrom()* devolve a quantidade de *bytes* recebidos do dispositivo *slave*.

```
int available(void);
```

Através do método *available()* é conhecida a quantidade de informação disponível para consequente operação de leitura. Um dispositivo *master* utiliza o método *available* depois de solicitar informação ao dispositivo *slave*, enquanto um dispositivo *slave* manipula este método para adquirir a informação transmitida pelo dispositivo *master*. O método *available()* devolve a quantidade de *bytes* disponíveis para leitura.

```
int read(void);
```

O método *read()* possibilita a leitura *byte* a *byte* da informação existente no *buffer*, sendo utilizado sempre que um dispositivo *master* transmite informação ao dispositivo *slave* ou quando um dispositivo *slave* comunica a informação requisitada pelo dispositivo *master*. O método *read()* devolve o próximo *byte* de informação.

```
void onReceive(void (* function)(int));
```

O método *onReceive()* permite atribuir um comportamento específico ao dispositivo *slave* para quando estiver a receber informação do dispositivo *master*. O parâmetro *function* representa o apontador da função a invocar quando o dispositivo *slave* adquire a informação que, por sua vez, apresenta como parâmetro um inteiro correspondente ao número de *bytes* obtidos e não devolve qualquer valor. O método *onReceive()* não devolve um resultado.

```
void onRequest(void (* function)(void));
```

O método *onRequest()* permite atribuir um comportamento específico ao dispositivo *slave* para quando o dispositivo *master* estiver a requisitar informação. O parâmetro *function* representa o apontador da função a invocar quando é requisitada informação ao dispositivo *slave* que, por sua vez, não detém qualquer parâmetro nem resultado. O método *onRequest()* também não devolve um resultado.

4.3 FICHEIRO DE CÓDIGO C

A presente secção explicita as principais funções do ficheiro *twi.c*, sendo apresentadas as assinaturas e esclarecidos os argumentos correspondentes.

```
void twi_init(void);
```

A função *twi_init()* implementa um conjunto de configurações características de cada dispositivo Arduino para inicializar o protocolo de comunicação I2C. Em particular, são preparadas as ligações SDA e SCL necessárias numa comunicação I2C e ainda é estabelecida a velocidade do canal de comunicação. A função *twi_init()* não devolve um resultado.

```
uint8_t twi_transmit(const uint8_t * data, uint8_t length);
```

Através da função *twi_transmit()* é preenchido o *buffer* de transmissão do dispositivo *slave* com informação para, posteriormente, comunicar ao dispositivo *master*. Os parâmetros da função *twi_transmit()* são um conjunto de *bytes* acompanhado pela quantidade de *bytes* existentes nesse conjunto. A função *twi_transmit()* devolve um *byte* que caracteriza o estado da operação de escrita cujo o valor indica sucesso ou a ocorrência de erro.

```
uint8_t twi_writeTo(uint8_t address, uint8_t * data, uint8_t length,
  uint8_t wait, uint8_t sendStop);
```

A função *twi_writeTo()* comunica determinada quantidade de informação ao dispositivo *slave*, assim que alcançada a permissão de *master* do canal de comunicação. O parâmetro *address* especifica o endereço de 7-bits do dispositivo *slave*. De seguida, os parâmetros *data* e *length* indicam qual o conjunto de *bytes* a transmitir e o número de *bytes* desse conjunto, respetivamente. Por outro lado, o parâmetro *wait* informa esperar ou não pela conclusão da transmissão e o parâmetro *sendStop* indica enviar ou não uma mensagem de paragem depois da transmissão. De salientar que o valor predefinido dos parâmetros *wait* e *sendStop* é *true*. Por último, a função *twi_writeTo()* devolve um *byte* que caracteriza o estado da transmissão cujo o valor informa sucesso ou a ocorrência de erro.

```
uint8_t twi_readFrom(uint8_t address, uint8_t * data, uint8_t length,
  uint8_t sendStop);
```

Com recurso à função *twi_readFrom()* é requisitada determinada quantidade de informação ao dispositivo *slave*, assim que alcançada a permissão de *master* do canal de comunicação. O parâmetro *address* especifica o endereço de 7-bits do dispositivo *slave*. De seguida, os parâmetros *data* e *length* indicam onde armazenar os *bytes* requisitados e o número de *bytes* a requisitar, respetivamente. Por outro lado, o parâmetro *sendStop* indica enviar ou não uma mensagem de paragem depois da transmissão. De salientar que o valor predefinido deste parâmetro é *true*. Por último, a função *twi_readFrom()* devolve a quantidade de *bytes* recebidos do dispositivo *slave*.

5

VERIFICAÇÃO DA BIBLIOTECA WIRE

O presente capítulo descreve o esforço de verificação da biblioteca *Wire* utilizando a ferramenta Frama-C e o *plug-in* Frama-Clang. O capítulo é constituído pelas secções *Configuração do Frama-C*, *Anotações e Limitações* e *Análise de Resultados e Discussão*. A secção *Configuração do Frama-C* demonstra os diversos passos indispensáveis para alcançar a configuração necessária da ferramenta Frama-C. De seguida, a secção *Anotações e Limitações* apresenta algumas anotações para exemplificar o trabalho desenvolvido e ainda as correspondentes limitações genéricas. Para finalizar, a secção *Análise de Resultados e Discussão* apresenta os resultados das condições de verificação, assim como algumas limitações e/ou particularidades específicas de determinados métodos e/ou funções. Os vários ficheiros de código com as correspondentes anotações são disponibilizados em `https://bitbucket.org/rafalexbar/arduino`.

5.1 CONFIGURAÇÃO E PARAMETRIZAÇÃO DO FRAMA-C

O repositório da dissertação contém os ficheiros de código que constituem a biblioteca *Wire*. Contudo, esta biblioteca recorre a outros *header files* disponibilizados pelas bibliotecas do Arduino, sendo necessário realizar a invocação da ferramenta Frama-C com a localização destes *header files*. Com isto, para ser possível a verificação da biblioteca *Wire* é fundamental configurar corretamente o Frama-C, sendo demonstrados no decorrer desta secção os vários passos para alcançar uma configuração que permita a verificação desta biblioteca.

Como a biblioteca *Wire* é constituída pelos ficheiros *twi.c* e *Wire.cpp*, a presente secção está dividida em duas partes complementares, isto é, para começar são demonstrados os passos para verificação do ficheiro de código C e, posteriormente, explicados os passos correspondentes ao ficheiro de código C++. A referida divisão de tarefas advém da particularidade do Frama-C suportar apenas a verificação da linguagem C, sendo necessária a instalação do *plug-in* Frama-Clang para permitir a verificação da linguagem C++.

Configuração e Parametrização para Verificação de Código C

Para a reprodução dos diversos passos da configuração é necessária a instalação antecipada do Arduino IDE disponibilizado em https://www.arduino.cc/en/Main/Software e da ferramenta Frama-C a partir de https://frama-c.com/download.html. A elaboração do espaço de trabalho compreende a criação da pasta principal *safe-wire* constituída pela cópia do repositório *arduino* da dissertação assim como uma cópia da pasta *hardware* disponibilizada pelo Arduino IDE. A cópia do repositório da dissertação é adquirida através do comando *git clone https://rafalexbar@bitbucket.org/rafalexbar/arduino.git*, enquanto a cópia da pasta *hardware* é obtida em sistemas macOS a partir da localização */Applications/Arduino.app/Contents/Java/hardware*. A pasta *hardware* é indispensável porque contém as várias bibliotecas e configurações do Arduino. De sublinhar que a execução dos comandos apresentados é sempre realizada a partir da pasta principal *safe-wire*.

A execução típica da ferramenta Frama-C para a verificação do ficheiro *twi.c* seria normalmente realizada através do simples comando:

```
frama-c-gui -wp -wp-rte arduino/twi.c
```

O comando *frama-c-gui* permite a execução do Frama-C com interface gráfica, sendo que a opção *-wp* possibilita a utilização do *plug-in* WP e a opção *-wp-rte* produz as condições de verificação de segurança como, por exemplo, ausência de *overflows* e *array out-of-bounds*, e a localização *arduino/twi.c* assinala o ficheiro *twi.c* para verificação. Contudo, a utilização deste comando provoca a seguinte mensagem de erro:

```
arduino/twi.c:23:10: fatal error: ‘avr/io.h’ file not found
```

A mensagem de erro advém da ausência de informação acerca da localização do ficheiro *avr/io.h*, sendo necessário indicar ao Frama-C onde encontrar este *header file* dentro da pasta *hardware*. Com isto, depois de determinada a localização do ficheiro *avr/io.h* é aplicado o seguinte comando:

```
frama-c-gui -wp -wp-rte
-cpp-extra-args="-I hardware/tools/avr/avr/include" arduino/twi.c
```

Através da opção *-cpp-extra-args* é revelada ao Frama-C a localização *hardware/tools/avr/avr/include* para procurar o ficheiro *avr/io.h*. No entanto, este comando ainda não apresenta toda a informação indispensável para a verificação do ficheiro *twi.c*, sendo apresentada a seguinte mensagem de erro:

```
arduino/twi.c:26:10: fatal error: ‘Arduino.h’ file not found
```

À semelhança da mensagem de erro anterior, esta mensagem requisita a especificação da localização do ficheiro *Arduino.h*, sendo acrescentada à opção *-cpp-extra-args* a localização

hardware/arduino/avr/cores/arduino. Para além disso, a utilização do último comando apresentado manifesta ainda a seguinte mensagem de aviso:

```
hardware/tools/avr/avr/include/avr/io.h:623:6:
warning: "device type not defined"
```

A mensagem de aviso comunica que não está estabelecido um dispositivo Arduino, porém é importante estabelecer um dispositivo para possibilitar a configuração específica dos *header files* e das variáveis. Para a resolução desta mensagem de aviso são adicionadas à opção *-cpp-extra-args* as constantes *-D __AVR_ATmega328P__* e *-D __AVR_DEVICE_NAME__= atmega328p* para indicar a consideração do dispositivo Arduino Uno, que detém um microcontrolador ATmega328P. Ainda é necessário acrescentar a constante *-D F_CPU=16000000L* para informar a frequência de 16 MHz característica deste microcontrolador. As constantes e os valores apresentados advêm da consulta dos ficheiros da pasta *hardware*. Com isto, é alcançado o seguinte comando:

```
frama-c-gui -wp -wp-rte -cpp-extra-args="-D __AVR_ATmega328P__
    -D __AVR_DEVICE_NAME__=atmega328p -D F_CPU=16000000L
-I hardware/tools/avr/avr/include -I hardware/arduino/avr/cores/arduino"
                            arduino/twi.c
```

Novamente, o comando indicado não contém a informação necessária para a verificação do ficheiro *twi.c*, uma vez que é apresentada a mensagem de erro:

```
hardware/arduino/avr/cores/arduino/Arduino.h:257:10:
fatal error: ‘pins_arduino.h’ file not found
```

Como esta mensagem de erro é idêntica às mensagens anteriores, para a resolução desta mensagem é acrescentada à opção *-cpp-extra-args* a localização *-I hardware/arduino/avr/variants/standard* do ficheiro *pins_arduino.h*, sendo adquirido o comando:

```
frama-c-gui -wp -wp-rte -cpp-extra-args="-D __AVR_ATmega328P__
    -D __AVR_DEVICE_NAME__=atmega328p -D F_CPU=16000000L
-I hardware/tools/avr/avr/include -I hardware/arduino/avr/cores/arduino
    -I hardware/arduino/avr/variants/standard" arduino/twi.c
```

De salientar que a pasta *standard* desta localização é proveniente da intenção de empregar o dispositivo Arduino Uno, dado que existe uma pasta específica com o ficheiro *pins_arduino.h* para cada modelo do Arduino. A execução do Frama-C através do comando supracitado apresenta a seguinte mensagem de erro:

```
syntax error at hardware/tools/avr/avr/include/avr/pgmspace.h:1529:14-38,
                        before or at token: src
```

A mensagem de erro aponta um problema de sintaxe resultante da utilização do tipo de dados *uint_farptr_t* no ficheiro *pgmspace.h*, porém a solução não é alcançada através da adição de informação ao comando. O erro aparece porque são considerados por defeito os ficheiros *inttypes.c* e *inttypes.h* presentes na localização *$(frama-c -print-path)/libc* do Frama-C em detrimento dos ficheiros *inttypes.c* e *inttypes.h* da pasta *hardware* que, por sua vez, detêm o tipo de dados requisitado. Portanto, para forçar a utilização dos cabeçalhos necessários podem, por exemplo, ser escondidos os ficheiros *inttypes.c* e *inttypes.h* do Frama-C, de modo que o Frama-C apenas consiga localizar os ficheiros da pasta *hardware*.

Com todas as configurações supracitadas aparenta ser praticável a verificação do ficheiro *twi.c*, porém durante a verificação é descoberta uma inconsistência no modelo de memória do *plug-in* WP resultante da necessidade de validação de endereços físicos de memória. A inconsistência advém da representação dos endereços físicos de memória, por exemplo, a validação do endereço de memória $0xBC$ origina $shift(global(0), 0xBC)$ e a definição da função *shift()*, que está presente em *$(frama-c -print-path)/wp/ergo/Memory.mlw*, é:

```
function shift(p: addr, k: int) : addr =
  { base = (p).base; offset = ((p).offset + k) }
```

Assim sendo, é produzido o resultado $addr = \{base = 0; offset = 0xBC\}$. No entanto, como o predicado *valid_rw* compreende a condição $0 < (p).base$, a avaliação deste predicado origina de forma errada uma inconsistência no modelo de memória. Com isto, para a resolução[1] desta inconsistência é inevitável a modificação do modelo de memória, sendo necessário substituir a declaração da função global presente em *$(frama-c -print-path)/wp/ergo/Memory.mlw* para:

```
logic global_base : int
function global(b : int) : addr = { base = global_base + b; offset = 0 }
```

Contudo, não existe qualquer garantia de que a modificação supracitada não provoque outros problemas no modelo de memória. De salientar que durante a dissertação é apenas utilizado o Alt-Ergo, sendo preservados os modelos de memória das ferramentas Why3 e Coq. A utilização destas ferramentas implicaria a modificação dos correspondentes modelos de memória.

Ainda relativamente à validação dos endereços físicos de memória, o ficheiro *twi.c* compreende a especificação do axioma *physical_address* apresentado na Listagem 5.1. A utilização do axioma *physical_address* permite considerar como válidos os vários endereços físicos de memória utilizados no decorrer do ficheiro *twi.c* e ainda assumir que a verificação destes endereços de memória ocorra externamente ao Frama-C.

1 A resolução apresentada é sugerida em:
https://stackoverflow.com/questions/45222693/how-to-force-a-memory-location-to-be-valid-in-acsl.

```
/*@ axiomatic physical_address
  @ {
  @   axiom twar: \valid(&TWAR);
  @   axiom twbr: \valid(&TWBR);
  @   axiom twcr: \valid(&TWCR);
  @   axiom twdr: \valid(&TWDR);
  @   axiom twsr: \valid(&TWSR);
  @ }
  @*/
```

Listagem 5.1: Axioma *physical_address* para Validação de Endereços Físicos de Memória

Por último, as variáveis pertencentes ao ficheiro *twi.c* apresentam atributos *volatile*, porém o Frama-C manifesta um escasso suporte para a demonstração das condições de verificação dependentes destas variáveis. Assim sendo, é necessário acrescentar a opção *-wp-no-volatile* para serem ignorados os atributos *volatile* das variáveis, sendo alcançado o comando completo para a verificação do ficheiro *twi.c*:

```
frama-c-gui -wp -wp-rte -wp-no-volatile -cpp-extra-args=
"-D __AVR_ATmega328P__ -D __AVR_DEVICE_NAME__=atmega328p -D F_CPU=16000000L
  -I hardware/tools/avr/avr/include -I hardware/arduino/avr/cores/arduino
        -I hardware/arduino/avr/variants/standard" arduino/twi.c
```

Configuração e Parametrização para Verificação de Código C++

A verificação de código C++ requisita a instalação antecipada do *plug-in* Frama-Clang disponibilizado em https://frama-c.com/frama-clang.html. Com isto, a verificação do ficheiro *Wire.cpp* seria normalmente desempenhada através do simples comando:

```
frama-c-gui -wp -wp-rte arduino/Wire.cpp
```

Contudo, a execução do Frama-C através deste comando apresenta duas mensagens de erro e uma mensagem de aviso:

```
arduino/Wire.cpp:23:12: fatal error: ‘inttypes.h’ file not found
  arduino/Wire.h:24:10: fatal error: ‘Stream.h’ file not found
        hardware/tools/avr/avr/include/avr/io.h:623:6:
            warning: "device type not defined"
```

Como estas mensagens são bastante idênticas aos erros e avisos do ficheiro *twi.c*, as configurações necessárias são também semelhantes. Assim sendo, para realizar a verificação do ficheiro *Wire.cpp* é requisitada a utilização do seguinte comando:

```
frama-c-gui -wp -wp-rte -cpp-extra-args="-D __AVR_ATmega328P__
    -D __AVR_DEVICE_NAME__=atmega328p -D F_CPU=16000000L
-I hardware/tools/avr/avr/include -I hardware/arduino/avr/cores/arduino"
                        arduino/Wire.cpp
```

Sumário

A verificação simultânea do ficheiro *twi.c* e do ficheiro *Wire.cpp* é realizável através da utilização do seguinte comando:

```
frama-c-gui -wp -wp-rte -wp-no-volatile -cpp-extra-args=
"-D __AVR_ATmega328P__ -D __AVR_DEVICE_NAME__=atmega328p -D F_CPU=16000000L
 -I hardware/tools/avr/avr/include -I hardware/arduino/avr/cores/arduino
-I hardware/arduino/avr/variants/standard" arduino/Wire.cpp arduino/twi.c
```

5.2 ANOTAÇÕES E LIMITAÇÕES

Anotações: write()

```
1  /*@ requires
2    @   txBufferIndex == txBufferLength;
3    @ assigns
4    @   txBuffer[0 .. BUFFER_LENGTH - 1],
5    @   txBufferIndex,
6    @   txBufferLength;
7    @ behavior full:
8    @   assumes
9    @       transmitting != 0
10   @     && txBufferLength >= BUFFER_LENGTH;
11   @   assigns
12   @     \nothing;
13   @   ensures
14   @     \result == 0;
15   @ behavior master:
16   @   assumes
17   @       transmitting != 0
18   @     && txBufferLength < BUFFER_LENGTH;
19   @   assigns
20   @     txBuffer[0 .. BUFFER_LENGTH - 1],
21   @     txBufferIndex,
22   @     txBufferLength;
23   @   ensures
24   @     \result == 1;
```

```
25   @   ensures
26   @        txBuffer[\old(txBufferIndex)] == \old(data)
27   @     && txBufferIndex  == \old(txBufferIndex) + 1
28   @     && txBufferLength == txBufferIndex;
29   @ behavior slave:
30   @   assumes
31   @     transmitting == 0;
32   @   assigns
33   @     \nothing;
34   @   ensures
35   @     \result == 1;
36   @ complete behaviors
37   @   full, master, slave;
38   @ disjoint behaviors
39   @   full, master, slave;
40   @*/
41 size_t TwoWire::write(uint8_t data)
42 {
43   if (transmitting) { // master transmitter mode
44
45     // don't bother if buffer is full
46     if (txBufferLength >= BUFFER_LENGTH) {
47       setWriteError();
48       return 0;
49     }
50
51     // put byte in tx buffer
52     txBuffer[txBufferIndex] = data;
53     ++txBufferIndex;
54
55     // update amount in buffer
56     txBufferLength = txBufferIndex;
57
58   } else { // slave send mode
59
60     // reply to master
61     twi_transmit(&data, 1);
62   }
63
64   return 1;
65 }
```

Listagem 5.2: Implementação e Anotações do Método *write()* do Ficheiro *Wire.cpp*

A Listagem 5.2 apresenta a implementação e as anotações do método *write()*. Quando utilizado pelo dispositivo *master*, o método *write()* permite a escrita da informação *data* no *txBuffer* para, posteriormente, transmitir ao dispositivo *slave*, tal como representado desde

a linha 51 à 56. Por outro lado, quando manipulado pelo dispositivo *slave*, o método *write()* possibilita a transmissão da informação *data* requisitada pelo dispositivo *master*, com recurso à função *twi_transmit()* como retratado pelas linhas 60 e 61.

As linhas 1 e 2 especificam a pré-condição do método *write()*, sendo apenas requisitado que os valores das variáveis *txBufferIndex* e *txBufferLength* sejam iguais visto que uma operação de escrita está correta quando a quantidade de informação existente no *txBuffer* corresponde à posição onde escrever a nova informação. Entre as linhas 3 e 6 são indicadas na cláusula *assigns* as únicas variáveis que são modificadas durante a execução do método *write()*, ou seja, as variáveis *txBuffer*, *txBufferIndex* e *txBufferLength*. Desde a linha 7 até à linha 35 são especificados os três comportamentos possíveis do método *write()* denominados *full*, *master* e *slave*.

O comportamento *full* ocorre quando um dispositivo *master* quer escrever informação mas o *txBuffer* está cheio, sendo preservado o valor de todas as variáveis e devolvido o resultado presente na pós-condição para informar zero *bytes* escritos com sucesso, tal como representado entre as linhas 7 e 14. O comportamento *master*, retratado entre linhas 15 e 28, ocorre quando um dispositivo *master* consegue escrever informação no *txBuffer* com sucesso, sendo modificadas as variáveis *txBuffer*, *txBufferIndex* e *txBufferLength* como indicado na cláusula *assigns*. A pós-condição das linhas 23 e 24 retrata a realização da escrita do *byte* com sucesso, enquanto a pós-condição das linhas 25 à 28 representa as alterações que ocorrem sobre as variáveis durante a operação de escrita. A informação *data* é escrita na posição *txBufferIndex* do *txBuffer* e, consequentemente, são incrementadas as variáveis *txBufferIndex* e *txBufferLength* para a atualizar o índice da próxima escrita e a quantidade de informação existente no *txBuffer*. Por outro lado, o comportamento *slave* retratado entre as linhas 29 e 35 ocorre quando um dispositivo *slave* quer transmitir informação, sendo preservado o valor de todas as variáveis do código C++ e sempre devolvido um resultado que indica a transmissão do *byte* com sucesso visto que é descartado o resultado da função *twi_transmit()*, utilizada para realizar a transmissão da informação a partir de código C.

Por último, enquanto as linhas 36 e 37 verificam a ocorrência de um comportamento, as linhas 38 e 39 verificam a existência de comportamentos exclusivos, ou seja, as anotações das linhas 36 à 39 verificam sempre a ocorrência de um e apenas um dos comportamentos enumerados.

Com isto, caso seja alcançada a validação das anotações supracitadas, é garantida a segurança do método *write()*.

Limitações: write()

Ao executar a ferramenta Frama-C para verificar o método *write()* através das anotações demonstradas são produzidas 19 condições de verificação, das quais 12 são válidas e as

restantes 7 desconhecidas. Contudo, ao observar com atenção as condições de verificação desconhecidas, é admissível concluir que estes resultados retratam algumas limitações da exploração do *plug-in* experimental Frama-Clang, sendo necessário o aperfeiçoamento deste *plug-in*. Em particular, os resultados da verificação do método *write()* demonstram problemas relacionados com as cláusulas *assigns* dos vários comportamentos e ainda relativos à principal cláusula *assigns* do método *write()*.

Quanto aos problemas relacionados com as cláusulas *assings* dos diversos comportamentos, o *plug-in* Frama-Clang não está a interpretar e, posteriormente, aplicar acertadamente estas cláusulas durante a verificação do método *write()*. Por outras palavras, para além de ser considerada a cláusula *assings* do último comportamento como *assings* principal do método *write()*, todos os comportamentos perdem as correspondentes cláusulas *assings*. Com isto, é impossível assegurar uma correta modificação das variáveis nos diferentes comportamentos e ainda verificar estas modificações no contexto global, visto que a principal cláusula *assigns* não corresponde às anotações elaboradas. De salientar que esta limitação ocorre em todos os métodos do ficheiro de C++ cujas anotações apresentem diferentes comportamentos, apesar da explicação deste problema recorrer ao método *write()*. Assim, caso seja verificada a principal cláusula *assigns* em qualquer método sobre estas circunstâncias, significa que a cláusula *assigns* estabelecida para o último comportamento desse método é igual ao *assigns* principal, sendo a verificação do *assigns* principal assegurada através desta simples coincidência.

Para além disto, a utilização do método *write()* pelo dispositivo *slave* resulta na invocação da função *twi_transmit()*. Mesmo assumindo a inexistência da limitação anterior, seria impossível a verificação da principal cláusula *assigns* e da cláusula *assigns* do comportamento *slave*, visto que nenhuma destas cláusulas contém as variáveis modificadas pela função *twi_transmit()*, tal como é possível averiguar pelas anotações da função *twi_transmit()* entre as linhas 4 e 6 da Listagem 5.3. Para a resolução deste problema seria necessário acrescentar as variáveis *twi_txBuffer* e *twi_txBufferLength*, que são modificadas durante a execução da função *twi_transmit()*, à principal cláusula *assigns* do método *write()* e à cláusula *assigns* do comportamento *slave*. Porém, esta solução é impraticável porque estas variáveis são apenas acessíveis pelo ficheiro de código C, de modo que o Frama-C e/ou o *plug-in* Frama-Clang não estão capacitados para reconhecer as variáveis utilizadas em diferentes ficheiros. Ainda assim, assumindo a ausência dos problemas das cláusulas *assigns* e da invocação da função *twi_transmit()*, as cláusulas *assigns* do método *write()* são verificadas, sendo possível garantir que este método apenas modifica o valor das variáveis certas, ou seja, o comportamento do método *write()* está de acordo com o esperado. Novamente, esta limitação não existe apenas no método *write()* mas em qualquer método do ficheiro de C++ que utilize funções do ficheiro C.

Anotações: twi_transmit()

A Listagem 5.3 apresenta a implementação e as anotações da função *twi_transmit()*. Através da função *twi_transmit()* é preenchido o *twi_txBuffer* do dispositivo *slave* com a informação armazenada no conjunto *data* para, posteriormente, comunicar ao dispositivo *master*, tal como representado entre as linhas 52 e 67.

```
1  /*@ requires
2    @      \valid(data + (0 .. length - 1))
3    @   && 0 < length <= TWI_BUFFER_LENGTH;
4    @ assigns
5    @   twi_txBuffer[0 .. TWI_BUFFER_LENGTH - 1],
6    @   twi_txBufferLength;
7    @ behavior success:
8    @   assumes
9    @        TWI_BUFFER_LENGTH >= (twi_txBufferLength + length)
10   @     && TWI_STX == twi_state;
11   @   assigns
12   @     twi_txBuffer[0 .. TWI_BUFFER_LENGTH - 1],
13   @     twi_txBufferLength;
14   @   ensures
15   @     \result == 0;
16   @   ensures
17   @        twi_txBufferLength == \old(twi_txBufferLength) + \old(length)
18   @     && \forall uint8_t index ; 0 <= index < \old(length) ==>
19   @     twi_txBuffer[\old(twi_txBufferLength) + index] == data[index];
20   @ behavior full:
21   @   assumes
22   @     TWI_BUFFER_LENGTH < (twi_txBufferLength + length);
23   @   assigns
24   @     \nothing;
25   @   ensures
26   @     \result == 1;
27   @ behavior receiver:
28   @   assumes
29   @        TWI_BUFFER_LENGTH >= (twi_txBufferLength + length)
30   @     && TWI_STX != twi_state;
31   @   assigns
32   @     \nothing;
33   @   ensures
34   @     \result == 2;
35   @ complete behaviors
36   @   success, full, receiver;
37   @ disjoint behaviors
38   @   success, full, receiver;
39   @*/
```

```
40 uint8_t twi_transmit(const uint8_t * data, uint8_t length)
41 {
42   // ensure data will fit into buffer
43   if (TWI_BUFFER_LENGTH < (twi_txBufferLength + length)) {
44     return 1;
45   }
46
47   // ensure we are currently a slave transmitter
48   if (TWI_STX != twi_state) {
49     return 2;
50   }
51
52   // set length and copy data into tx buffer
53
54   /*@ loop invariant
55     @      0 <= i <= length
56     @   && \forall uint8_t k ; 0 <= k < i ==>
57     @   twi_txBuffer[twi_txBufferLength + k] == data[k];
58     @ loop assigns
59     @   i, twi_txBuffer[0 .. TWI_BUFFER_LENGTH - 1];
60     @ loop variant
61     @   (length - i);
62     @*/
63   for (uint8_t i = 0; i < length; ++i) {
64     twi_txBuffer[twi_txBufferLength + i] = data[i];
65   }
66
67   twi_txBufferLength += length;
68
69   return 0;
70 }
```

Listagem 5.3: Implementação e Anotações da Função *twi_transmit()* do Ficheiro *twi.c*

As linhas 1, 2 e 3 apresentam a pré-condição da função *twi_transmit()*, sendo requisitado um conjunto *data* com todas as posições de memória válidas e uma dimensão *length* positiva mas menor ou igual à constante *TWI_BUFFER_LENGTH*, que é utilizada para definir a dimensão dos diversos *buffers* do ficheiro *twi.c*. A cláusula *assigns* detalhada entre as linhas 4 e 6 informa que apenas são modificadas as variáveis *twi_txBuffer* e *twi_txBufferLength* durante a execução da função *twi_transmit()*. Desde a linha 7 até à linha 34 são especificados os três comportamentos da função *twi_transmit()* denominados *success*, *full* e *receiver*.

O comportamento *full* ocorre quando um dispositivo *slave* tenta escrever demasiada informação no *twi_txBuffer*, sendo preservado o valor de todas as variáveis e devolvido o valor um representado na pós-condição para informar a ocorrência deste erro, tal como apresentado entre as linhas 20 e 26. Por outro lado, o comportamento *receiver* retratado entre as

linhas 27 e 34 acontece quando a tarefa do dispositivo *slave* é adquirir e não transmitir a informação, apesar de tentar escrever uma quantidade aceitável de informação. Assim, também são preservados os valores das variáveis e o valor devolvido é dois, como apresentado na pós-condição para caracterizar a ocorrência deste erro. Por último, assumindo um dispositivo *slave* que pretende escrever uma quantidade válida de informação no *twi_txBuffer* e que, para além disso, está encarregue de transmitir essa informação, é alcançado o comportamento *success* especificado entre as linhas 7 e 19. A cláusula *assigns* deste comportamento indica que são modificadas as variáveis *twi_txBuffer* e *twi_txBufferLength*, enquanto o resultado na pós-condição das linhas 14 e 15 retrata o correto funcionamento da função *twi_transmit()*. Para além disso, a pós-condição entre as linhas 16 e 19 permite verificar a quantidade de informação existente no *twi_txBuffer* após a escrita e comprovar a escrita acertada da informação do conjunto *data* no *twi_txBuffer*.

As cláusulas *complete* e *disjoint* definidas entre as linhas 35 e 38 permitem verificar a ocorrência de sempre um e apenas um dos comportamentos enumerados. Ao contrário do método *write()* apresentado anteriormente, é necessário complementar o contrato da função *twi_transmit()* com anotações auxiliares no decorrer da implementação, devido à existência do ciclo que realiza a operação de escrita no *twi_txBuffer*, tal como representado entre as linhas 54 e 61. Desde a linha 54 à 57 é especificado um invariante de ciclo para verificar a variação de valores do índice utilizado para contar a informação a escrever e ainda averiguar a escrita acertada da informação em todas as iterações do ciclo. As linhas 58 e 59 definem que durante o ciclo apenas são modificadas as variáveis correspondentes ao iterador de ciclo *i* e a variável *twi_txBuffer*. Por último, as linhas 60 e 61 especificam um variante de ciclo para assegurar a terminação do ciclo após a escrita da quantidade *length* de informação.

Com isto, através da validação de todas as anotações supracitadas, é garantida a segurança da função *twi_transmit()*.

Limitações: twi_transmit()

Ao executar a ferramenta Frama-C para verificar a função *twi_transmit()* com recurso às anotações apresentadas são produzidas 41 condições de verificação, sendo todas as condições de verificação válidas. Contudo, é importante recordar a utilização da opção *-wp-no-volatile* durante a execução do Frama-C para contornar as limitações da verificação das variáveis *volatile*, uma vez que a opção *-wp-no-volatile* permite ignorar os atributos *volatile* das variáveis. Em caso de utilização da opção *-wp-volatile* em detrimento da opção *-wp-no-volatile*, ao executar o Frama-C para verificação da função *twi_transmit()* seriam produzidas 41 condições de verificação, das quais 22 seriam válidas e as restantes 19 desconhecidas. A discrepância de resultados entre as duas opções supracitadas advém da impossibilidade de demonstra-

ção das condições de verificação dependentes das variáveis *volatile* quando considerada a opção *-wp-volatile*. Uma solução seria a exploração de código *ghost* para verificação das variáveis *volatile*, porém o suporte disponibilizado pelo Frama-C ainda é escasso. Em particular, o Frama-C ainda não está preparado para verificar variáveis *volatile* através de código *ghost*, apesar de aparentemente a interpretação de código *ghost* ocorrer com sucesso. Com isto, é estudada uma solução alternativa que compreende a criação do ficheiro *twi-volatile.c* para verificar os atributos *volatile* das diversas variáveis, com recurso a variáveis auxiliares e a quantificadores universais. Os resultados desta solução experimental são adquiridos ao considerar o ficheiro *twi-volatile.c* em vez do ficheiro *twi.c* durante a execução do Frama-C, sendo produzidas 43 condições de verificação válidas.

As diferenças entre os ficheiros *twi.c* e *twi-volatile.c* começam com as declarações das variáveis globais auxiliares denominadas *auxiliary_variable*, onde *variable* corresponde aos nomes das variáveis *volatile* como, por exemplo, a variável *auxiliary_state* corresponde à variável *twi_state*. As variáveis auxiliares não apresentam características *volatile*, sendo utilizadas durante a execução das diversas funções e, posteriormente, aproveitadas para atribuir os resultados das operações às variáveis com atributos *volatile*. Para além disso, existem algumas diferenças entre as anotações destes dois ficheiros, uma vez que as anotações do ficheiro *twi-volatile.c* apresentam modificações para suportar as variáveis auxiliares e ainda acrescentam quantificadores universais para demonstrar as condições de verificação correspondentes às variáveis *volatile*.

A Listagem 5.4 apresenta uma versão simplificada da implementação e das anotações da função *twi_transmit()* do ficheiro *twi-volatile.c*. Quanto à implementação, as linhas 28 e 29 demonstram a utilização da variável *auxiliary_txBufferLength* em detrimento da variável *twi_txBufferLength* durante a operação de escrita, enquanto as linhas 32 e 33 apresentam a atualização da variável *auxiliary_txBufferLength* para posterior atribuição deste resultado à variável *twi_txBufferLength*. Quanto às anotações, entre as linhas 1 e 15 são demonstradas modificações ao comportamento *success* como, por exemplo, a consideração das variáveis *auxiliary_txBufferLength* e *auxiliary_state* em várias condições e ainda a adição da variável *auxiliary_txBufferLength* à cláusula *assigns*. De sublinhar que a pós-condição das linhas 12 e 13 recorre ao quantificador universal para verificar o valor das variáveis *auxiliary_txBufferLength* e *twi_txBufferLength*, onde qualquer valor simultaneamente igual às duas variáveis supracitadas corresponda à quantidade atualizada de informação existente no *twi_txBuffer*. Para finalizar, as linhas 21 e 22 apresentam a única modificação às anotações de ciclo, onde é utilizada a variável *auxiliary_txBufferLength* no invariante de ciclo para verificar a escrita durante as várias iterações do ciclo.

```
/*@ behavior success:
  @   assumes
  @       TWI_BUFFER_LENGTH >= (auxiliary_txBufferLength + length)
  @     && TWI_STX == auxiliary_state;
  @   assigns
  @     twi_txBuffer[0 .. TWI_BUFFER_LENGTH - 1],
  @     auxiliary_txBufferLength,
  @     twi_txBufferLength;
  @   ensures
  @     \result == 0;
  @   ensures
  @     (\forall uint8_t x ; (x == twi_txBufferLength && x == auxiliary_tx
  @     BufferLength) ==> x == \old(auxiliary_txBufferLength) + \old(length))
  @     && (\forall uint8_t index ; 0 <= index < \old(length) ==>
  @     twi_txBuffer[\old(auxiliary_txBufferLength) + index] == data[index]);
  @*/
uint8_t twi_transmit(const uint8_t * data, uint8_t length)
{
  /*@ loop invariant
    @      ...
    @   && \forall uint8_t k ; 0 <= k < i ==>
    @   twi_txBuffer[auxiliary_txBufferLength + k] == data[k];
    @ loop assigns
    @   ...
    @ loop variant
    @   ...
    @*/
  for (uint8_t i = 0; i < length; ++i) {
    twi_txBuffer[auxiliary_txBufferLength + i] = data[i];
  }

  auxiliary_txBufferLength += length;
  twi_txBufferLength = auxiliary_txBufferLength;

  return 0;
}
```

Listagem 5.4: Implementação e Anotações da Função *twi_transmit()* do Ficheiro *twi-volatile.c*

5.3 Análise de Resultados e Discussão

A presente secção apresenta uma análise aos resultados da verificação dos ficheiros *Wire.cpp*, *twi.c* e *twi-volatile.c*, sendo esclarecidos alguns problemas e/ou pormenores específicos de determinados métodos e/ou funções.

Função	Válido	Desconhecido	Total
begin(void)	3	2	5
begin(uint8_t)	5	8	13
begin(int)	5	-	5
end(void)	-	2	2
setClock(uint32_t)	1	2	3
requestFrom(uint8_t, uint8_t, uint32_t, uint8_t, uint8_t)	33	8	41
requestFrom(uint8_t, uint8_t, uint8_t)	7	-	7
requestFrom(uint8_t, uint8_t)	7	-	7
requestFrom(int, int)	7	-	7
requestFrom(int, int, int)	7	-	7
beginTransmission(uint8_t)	2	-	2
beginTransmission(int)	5	-	5
endTransmission(uint8_t)	5	2	7
endTransmission(void)	7	-	7
write(uint8_t)	12	7	19
write(const uint8_t *, size_t)	14	4	18
available(void)	4	-	4
read(void)	8	1	9
peek(void)	6	-	6
onReceiveService(uint8_t *, int)	10	10	20
onRequestService(void)	2	8	10
onReceive(void (*)(int))	-	-	-
onRequest(void (*)(void))	-	-	-

Tabela 1: Condições de Verificação do Ficheiro *Wire.cpp*

Análise de Resultados do Ficheiro Wire.cpp

A Tabela 1 apresenta os resultados das condições de verificação dos diversos métodos do ficheiro *Wire.cpp*. O método *begin(uint8_t)* apresenta duas condições de verificação desconhecidas devido à invocação das funções *twi_attachSlaveTxEvent()* e *twi_attachSlaveRxEvent()*, cujas pré-condições aplicam o predicado *\valid_function* para verificar apontadores para funções. Apesar do predicado *\valid_function* estar disponível na linguagem ACSL, a ferramenta Frama-C demonstra apenas capacidade para interpretar este predicado, sendo necessário expandir o suporte do Frama-C para, posteriormente, realizar a correspondente verificação. Para além disso, aparentemente não existe qualquer suporte do *plug-in* Frama-Clang para verificar apontadores para funções, uma vez que as anotações dos métodos *onReceive(void (*)(int))* e *onRequest(void (*)(void))* são descartadas devido à utilização do pre-

dicado *\valid_function* identificado como desconhecido. De salientar que caso sejam omitidas as pré-condições destes dois métodos, as condições de verificação correspondentes às cláusulas *assigns* e às pós-condições são válidas.

As condições de verificação desconhecidas dos métodos privados *onReceiveService(uint8_t *, int)* e *onRequestService(void)* estão, principalmente, relacionadas com as invocações de funções através de apontadores para funções. Em particular, são desconhecidas as condições de verificação relativas às asserções que manipulam o predicado *\valid_function*, assim como as condições de verificação correspondentes às cláusulas *assigns* e às pós-condições dependentes destas invocações de funções, visto que as variáveis modificadas pelos apontadores para funções são desconhecidas para o Frama-C. As restantes condições de verificação apresentadas como desconhecidas na Tabela 1 resultam das limitações explicitadas na secção anterior, ou seja, das limitações correspondentes às cláusulas *assigns* dos comportamentos e dos problemas relativos à principal cláusula *assigns* dos métodos de código C++ que utilizam as funções de código C.

Para finalizar, apesar dos resultados das condições de verificação não retratarem a intervenção exercida sobre o código, os métodos *requestFrom(uint8_t, uint8_t, uint32_t, uint8_t, uint8_t)* e *write(const uint8_t *, size_t)* apresentam pequenas modificações devido à necessidade de informar o *plug-in* Frama-Clang a intenção destes métodos utilizarem especificamente o método *write(uint8_t)* do ficheiro *Wire.cpp*. A modificação do código destes dois métodos resulta da necessidade de evitar os procedimentos utilizados pelo Frama-Clang para a verificação de métodos virtuais, uma vez que o Frama-Clang ainda não está preparado para desempenhar a verificação deste tipo de métodos, apesar de conseguir representar acertadamente métodos com estas características.

Análise de Resultados dos Ficheiros twi.c e twi-volatile.c

A Tabela 2 apresenta os resultados das condições de verificação das várias funções do ficheiro *twi.c* com a opção *-wp-no-volatile* e ainda do ficheiro *twi-volatile.c* com a opção *-wp-volatile* para a execução da ferramenta Frama-C.

Quanto ao ficheiro *twi.c*, apenas a função *__vector_24()* possui condições de verificação desconhecidas, sendo os resultados destas condições de verificação resultantes das invocações de funções através dos apontadores para funções denominados *(* twi_onSlaveTransmit)()* e *(* twi_onSlaveReceive)()*. A função *__vector_24()* representa um *Interrupt Service Routine (ISR)* definido através do operador *switch* cujo objetivo é estabelecer os procedimentos a realizar pelo software quando o hardware estiver a desempenhar a comunicação da informação. Contudo, existem duas situações nos procedimentos como dispositivo *slave* que utilizam os apontadores para funções supracitados, sendo impossibilitada a verificação das correspondentes cláusulas *assigns*, assim como as pós-condições dependentes destas invo-

	twi.c & -wp-no-volatile			twi-volatile.c & -wp-volatile		
Função	**Válido**	**Desconhecido**	**Total**	**Válido**	**Desconhecido**	**Total**
twi_init()	11	-	11	12	-	12
twi_disable()	5	-	5	4	-	4
twi_setAddress()	5	-	5	5	-	5
twi_setFrequency()	4	-	4	4	-	4
twi_readFrom()	54	-	54	61	-	61
twi_writeTo()	58	-	58	65	-	65
twi_transmit()	41	-	41	43	-	43
twi_attachSlaveRxEvent()	2	-	2	2	-	2
twi_attachSlaveTxEvent()	2	-	2	2	-	2
twi_reply()	9	-	9	8	-	8
twi_stop()	5	-	5	4	-	4
twi_releaseBus()	3	-	3	3	-	3
__vector_24()	226	15	241	-	-	-

Tabela 2: Condições de Verificação dos Ficheiros *twi.c* e *twi-volatile.c*

cações de funções, uma vez que o Frama-C não reconhece as modificações realizadas pelos apontadores para funções. Para além disso, existe um caso de utilização do apontador *(* twi_onSlaveTransmit)()* com ausência do operador *break*, sendo influenciados negativamente os resultados das condições de verificação do caso seguinte. De salientar que o Frama-C introduz automaticamente condições de verificação que aplicam o predicado *\valid_function*, porém ainda não está preparado para desempenhar a correspondente verificação.

A execução do ISR é assíncrona, sendo possível a realização de várias tarefas simultaneamente. Em contrapartida, a ferramenta Frama-C não possibilita a verificação deste modelo de execução, uma vez que está apenas preparada para a verificação de execuções sequenciais. Com isto, apesar das condições de verificação válidas, as funções *twi_readFrom()*, *twi_writeTo()* e *twi_stop()* contêm variáveis cujos valores dependem das modificações assíncronas, sendo impossível para o *Frama-C* alcançar a validação dos valores destas variáveis sem a utilização de um *plug-in* adicional, daí as anotações destas funções não incorporarem as eventuais pós-condições das referidas variáveis.

Uma particularidade das funções *twi_setAddress()*, *twi_readFrom()* e *twi_writeTo()* é a necessidade de aplicação da opção *strategies* disponibilizada pelo Frama-C para, consequentemente, conseguirem a validação das condições de verificação dependentes da operação *shift* à esquerda. Para finalizar, outra particularidade interessante é a manipulação de asserções com *labels* e, posteriormente, a combinação destas *labels* com um quantificador existencial numa pós-condição, para alcançar a verificação do novo valor da variável *TWSR* após a execução da função *twi_init()*.

Quanto ao ficheiro *twi-volatile.c*, as condições de verificação válidas compreendem a avaliação com sucesso das variáveis com atributos *volatile*. A variação da quantidade de condições de verificação está relacionada com a necessidade do Frama-C realizar verificações adicionais devido à introdução das variáveis auxiliares. Porém, também existem situações onde as variáveis auxiliares permitem ao Frama-C descartar algumas das verificações anteriores. Para além disso, o ficheiro *twi-volatile.c* apresenta as particularidades anteriormente apresentadas para o ficheiro *twi.c*. De sublinhar que não é apresentado o número de condições de verificação - Válido, Desconhecido e Total - para a função *__vector_24()*, uma vez que a implementação e as correspondentes anotações são demasiado extensas, pelo que não são incorporadas as modificações requisitadas pela abordagem. Apesar dos resultados positivos para as restantes funções, é admissível concluir que esta abordagem é apenas aceitável para a verificação de funções pequenas e/ou intermédias, uma vez que são necessárias modificações na implementação e nas anotações, pelo que a complexidade desta abordagem está dependente da quantidade de funções e das dimensões correspondentes à implementação e às anotações.

6

CONCLUSÃO

6.1 CONCLUSÕES

A dissertação tinha como principal objetivo a validação e certificação de software utilizado em sistemas Arduino. Uma vez que estão presentes em praticamente todos estes sistemas, as bibliotecas *standard* foram o principal ponto de investimento. Em particular, devido ao contexto em que a dissertação foi realizada, o foco incidiu na verificação da biblioteca de comunicação *Wire*, uma biblioteca que implementa o protocolo de comunicação I2C. Os resultados alcançados são bastante positivos, dado que cerca de 90% das condições de verificação são demonstradas válidas, como seria expectável desta amadurecida biblioteca *open-source* pertencente à numerosa comunidade do Arduino.

A ferramenta Frama-C demonstrou ser bastante robusta, sendo uma excelente opção para realizar a verificação do software utilizado em sistemas críticos. Apesar disso, existem algumas limitações que terão de ser ultrapassadas por forma a permitir a verificação de determinadas características do código C como, por exemplo, a invocação de funções através de apontadores e a utilização de variáveis com atributos *volatile*. De salientar que a abordagem experimentada para a verificação das condições dependentes das variáveis *volatile* apresentou resultados positivos, porém a complexidade desta abordagem aumenta consoante a dimensão do código devido à necessidade de realizar modificações.

Outro objetivo desta dissertação foi a exploração do *plug-in* Frama-Clang, dado que existe um interesse generalizado na certificação de código C++. O *plug-in* Frama-Clang demonstrou ser bastante promissor, uma vez que possibilita a verificação das principais características de código C++ através do Frama-C, embora ainda esteja numa etapa experimental. Apesar da existência de algumas limitações relativas à interpretação das cláusulas *assigns* e à verificação dos métodos com características virtuais, estas limitações não impossibilitaram a verificação da biblioteca *Wire*. Para além disso, as notáveis semelhanças entre as gramáticas das linguagens ACSL e ACSL++ representam uma enorme vantagem para os utilizadores porque proporcionam uma melhor aprendizagem.

Em suma, apesar das limitações do Frama-C e do *plug-in* Frama-Clang, os resultados da verificação da biblioteca *Wire* são bastante positivos. Deve ser considerado o facto da biblioteca escolhida apresentar diversas funcionalidades avançadas que levaram à identificação das limitações referidas. No entanto, acredita-se que não existam outras limitações relevantes no contexto da verificação de *embedded systems*, no sentido em que não é de prever que uma biblioteca diferente levante um conjunto adicional de problemas que leve à identificação de novas limitações.

6.2 TRABALHO FUTURO

Os resultados desta dissertação são bastante positivos, sendo um contributo importante para a confiabilidade dos *embedded systems* desenvolvidos a partir de dispositivos Arduino. A partir desta dissertação existem inúmeras possibilidades para futuras contribuições como, por exemplo, a exploração da verificação formal de software para continuar a certificação de outras bibliotecas de comunicação ou ainda qualquer outra biblioteca popular entre os dispositivos Arduino. Outra possibilidade, dado que a exploração do *plug-in* Frama-Clang decorreu com sucesso, é considerar a utilização do Frama-Clang para a verificação de sistemas críticos desenvolvidos em C++, uma vez que são escassas as ferramentas disponíveis para a realização desta verificação, apesar do interesse generalizado na certificação de código C++. Em particular, a exploração do Frama-Clang para a verificação das componentes importantes do *Robotic Operating System (ROS)* [50] escritas em código C++, devido à sua abundante utilização para o desenvolvimento de diversificados robôs.

Para além disso, existem contribuições futuras relativas às adversidades encontradas durante a dissertação como, por exemplo, efetuar a verificação dos apontadores para funções e das variáveis com atributos *volatile*, apesar da necessidade de aguardar pelo aperfeiçoamento do Frama-C. Quanto às adversidades da verificação resultantes da existência de código com execução assíncrona, uma possível contribuição é a exploração do *plug-in* Mthread [51] desenvolvido com o intuito de possibilitar a análise de programas concorrentes escritos em código C. Outras contribuições dependem do desenvolvimento do *plug-in* Frama-Clang para experimentar a verificação dos métodos com características virtuais e demonstrar a validade das cláusulas *assigns* prejudicadas pela incorreta interpretação.

Em suma, são abundantes as possíveis aplicações da ferramenta Frama-C e do *plug-in* Frama-Clang para a certificação do software de sistemas críticos.

BIBLIOGRAFIA

[1] Arduino. Arduino - Introduction.
https://www.arduino.cc/en/Guide/Introduction. Accessed: 03-09-2018.

[2] SparkFun. Inter-Integrated Circuit (I2C).
https://learn.sparkfun.com/tutorials/i2c. Accessed: 03-09-2018.

[3] SparkFun. Serial Peripheral Interface (SPI). https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi. Accessed: 03-09-2018.

[4] Arduino. Arduino - Wire.
https://www.arduino.cc/en/Reference/Wire. Accessed: 03-09-2018.

[5] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles and Boris Yakobowski. *Frama-C User Manual*. CEA LIST, Sulfur Edition, 2017.

[6] Patrick Baudin, François Bobot, Loïc Correnson and Zaynah Dargaye. *WP Plug-in Manual*. CEA LIST, Sulfur Edition, 2017.

[7] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, Version 1.12, 2016.

[8] CEA LIST. The Frama-Clang Plug-in.
https://frama-c.com/frama-clang.html. Accessed: 23-10-2018.

[9] Wiring. About \ Wiring.
http://wiring.org.co/about.html. Accessed: 03-09-2018.

[10] Atmel Corporation. *ATmega128/L Datasheet*.
Rev. 2467X, 2011.

[11] Atmel Corporation. *ATmega8/L Datasheet*.
Rev. 2486AA, 2013.

[12] Hernando Barragán. The Untold History of Arduino.
https://arduinohistory.github.io. Accessed: 03-09-2018.

[13] Massimo Banzi. *Getting Started with Arduino*. O'Reilly Media, Second Edition, 2011. ISBN: 978-1-449-309879.

[14] Arduino. Arduino Uno Rev3. https://store.arduino.cc/arduino-uno-rev3. Accessed: 03-09-2018.

[15] Atmel Corporation. *ATmega48A/PA/88A/PA/168A/PA/328/P Datasheet*. Rev. 8271J, 2015.

[16] Arduino. Arduino - SPI. https://www.arduino.cc/en/Reference/SPI. Accessed: 03-09-2018.

[17] Processing. Overview \ Processing.org. https://processing.org/overview/. Accessed: 27-10-2018.

[18] Arduino. Arduino - Sketch. https://www.arduino.cc/en/Tutorial/Sketch. Accessed: 03-09-2018.

[19] Arduino. Arduino - Bare Minimum. https://www.arduino.cc/en/Tutorial/BareMinimum. Accessed: 03-09-2018.

[20] Arduino. Arduino - Build Process. https://www.github.com/arduino/Arduino/wiki/Build-Process. Accessed: 03-09-2018.

[21] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming*, pages 125–128, 2013. DOI: 10.1007/978-3-642-37036-6_8.

[22] Rustan Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370, 2010. DOI: 10.1007/978-3-642-17511-4_20.

[23] Lawrence Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989. DOI: 10.1007/BF00248324.

[24] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, Version 8.7.1, 2017.

[25] Donald Loveland. *Automated Theorem Proving: a Logical Basis*. Volume 6 of *Fundamental Studies in Computer Science*. North Holland, First Edition, 1978. ISBN: 0720404991.

[26] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011. DOI: 10.1145/1995376.1995394.

[27] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009. DOI: 10.1145/1592434.1592438.

[28] Edmund Clarke, Orna Grumberg and David Long. Model Checking and Abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, 1992. DOI: 10.1145/143165.143235.

[29] Edmund Clarke, Daniel Kroening and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004. DOI: 10.1007/978-3-540-24730-2_15.

[30] CEA LIST. What is Frama-C. http://frama-c.com/what_is.html. Accessed: 03-09-2018.

[31] Kerstin Hartig, Jens Gerlach, Juan Soto and Jürgen Busse. Formal Specification and Automated Verification of Safety-Critical Requirements of a Railway Vehicle with Frama-C/Jessie. In *Proceedings of the 8th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 145–153, 2010. DOI: 10.1007/978-3-642-14261-1_15.

[32] Claude Marché. Jessie: an Intermediate Language for Java and C Verification. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, pages 1–2, 2007. DOI: 10.1145/1292597.1292598.

[33] CEA LIST. WP Plug-in. https://frama-c.com/wp.html. Accessed: 03-09-2018.

[34] CEA LIST. The Architecture of Frama-C. http://frama-c.com/plugins.html. Accessed: 03-09-2018.

[35] CEA LIST. ANSI/ISO C Specification Language. http://frama-c.com/acsl.html. Accessed: 03-09-2018.

[36] Nikolai Kosmatov and Julien Signoles. Frama-C, a Collaborative Framework for C Code Verification: Tutorial Synopsis. In *Proceedings of the 16th International Conference on Runtime Verification*, pages 92–115, 2016. DOI: 10.1007/978-3-319-46982-9_7.

[37] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer and Alain Mebsout. The Alt-Ergo Automated Theorem Prover. http://alt-ergo.lri.fr/. Accessed: 27-10-2018.

[38] Leonardo De Moura and Nikolaj Bjørner. Z3: an Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. DOI: 10.1007/978-3-540-78800-3_24.

[39] Peter Trommler. A Formally Verified Digital Signature Device for Smartphones. *IADIS International Journal on Computer Science and Information Systems*, 10(2):1–16, 2015. ISSN: 1646-3692.

[40] Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto and Aaron Tomb. Formal Verification of kLIBC with the WP Frama-C Plug-in. In *Proceedings of the 6th International Symposium on NASA Formal Methods*, pages 343–358, 2014. DOI: 10.1007/978-3-319-06200-6_29.

[41] Frank Dordowsky. An Experimental Study using ACSL and Frama-C to Formulate and Verify Low-Level Requirements from a DO-178C Compliant Avionics Project. In *Proceedings of the 2nd International Workshop on Formal Integrated Development Environment*, pages 28–41, 2015. DOI: 10.4204/EPTCS.187.3.

[42] Nancy Leveson. The Role of Software in Spacecraft Accidents. *AIAA Journal of Spacecraft and Rockets*, 41(4):564–575, 2004. DOI: 10.2514/1.11950.

[43] Rovedy Aparecida Busquim e Silva, Nanci Naomi Arai, Luciana Akemi Burgareli, José Maria Parente de Oliveira and Jorge Sousa Pinto. Formal Verification With Frama-C: A Case Study in the Space Software Domain. *IEEE Transactions on Reliability*, 65(3):1163-1179, 2016. ISSN: 1558-1721.

[44] Guillaume Brat. Reducing V&V Cost of Flight Critical Systems: Myth or Reality? In *AIAA Information Systems - AIAA Infotech @ Aerospace, AIAA SciTech Forum*, 2017. DOI: 10.2514/6.2017-0563.

[45] CEA LIST and STANCE. Clang as a C++ Front-End for Software Analysis. https://llvm.org/devmtg/2014-04/PDFs/Posters/FramaC.pdf. Accessed: 23-10-2018.

[46] Virgile Prevosto and Franck Védrine. Frama-Clang, a C++ Front-End for Frama-C. https://frama-c.com/download/framaCDay/FCD16/talk/virgileprevosto_framaclang.pdf. Accessed: 23-10-2018.

[47] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat and SGI. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. Accessed: 23-10-2018.

[48] Arduino. Arduino Playground - Wire Library Detailed Reference. https://playground.arduino.cc/Main/WireLibraryDetailedReference. Accessed: 23-10-2018.

[49] Daniel Hienzsch. The Arduino Wire Library. https://rheingoldheavy.com/arduino-wire-library/. Accessed: 23-10-2018.

[50] ROS. ROS.org | About ROS.
http://www.ros.org/about-ros/. Accessed: 27-10-2018.

[51] CEA LIST. The Mthread Plug-in.
https://frama-c.com/mthread.html. Accessed: 27-10-2018.