UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

PROGRAMAÇÃO DE REDES SENSORES BASEADA EM EVENTOS

Bruno Alexandre Valdez Fernandes

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

PROGRAMAÇÃO DE REDES SENSORES BASEADA EM EVENTOS

Bruno Alexandre Valdez Fernandes

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Francisco Cipriano da Cunha Martins

2014

Agradecimentos

Aos meus avós por me terem dado a oportunidade de poder ter tirado tanto o Mestrado como a Licenciatura. Sem eles este meu percurso académico não seria possível. Por isso a sua ajuda foi muito importante.

Ao meu irmão por ser um chato e assim conseguir distrair-me da faculdade sempre que eu precisei. Mas apesar disso vai continuar a perder comigo. À minha mãe por me ter sempre apoiado ao longo destes anos e dado a sua força. Ao *Buddy* e ao *Rex* por terem sido os meus companheiros de estudo durante estes anos. À minha tia Susana também por me ter apoiado.

Ao Professor Francisco por ter contruibuído para a minha aprendizagem ao longo deste ano, pela sua disponibilidade sempre que eu necessitei e pela sua camaradagem. Ele fez-me perceber o que é realmente trabalhar num projeto em informática e fez-me crescer como um profissional na área de informática. Ao Carlos por me ter ajudado sempre que precisei e ter comparticipado neste meu trabalho.

À família da parte do meu pai por me terem ajudado e apoiado sempre que eu precisei ao longo destes cinco anos.

Ao Fábio por me ter acompanhado desde o primeiro ano de faculdade. Aprendemos juntos e partilhámos muito desde o primeiro ano. Que continuemos a trabalhar juntos ao longo de vários anos. Aos meus restantes colegas por me terem acompanhado durante os diversos projetos e por terem contibuído para o meu percurso académico.

Resumo

As redes de sensores são constituídas por dispositivos com recursos muito limitados e na maior parte dos casos, utilizam pilhas com baixa capacidade como fonte de alimentação. Muitos destes dispositivos dispõem de uma pequena quantidade de armazenamento – apenas 2 *KB* de memória *RAM* – o que ilustra bem as suas grandes limitações computacionais. Ainda assim, estes dispositivos são utilizados em diversas áreas. A título de exemplo podemos salientar aplicação nas áreas da medicina, monitorização de incêndios, em operações militares e até em alarmes de incêndios dentro dos edifícios.

A programação destes dispositivos está relacionado com o seu microcontrolador, sendo assim dependente do fabricante que produz o dispositivo. Tendo em conta que os fabricantes constroem dispositivos com diferentes microcontroladores, são usadas diversas linguagens de programação. A linguagem que propomos é baseada em eventos, ou seja, reage a estímulos. Exemplificando, se tivermos um dispositivo que esteja a fazer monitorização de incêndios e que determine que existe um fogo, este dispositivo vai reagir a esse estímulo, executando a parte do programa correspondente a este evento.

Para a linguagem *Macaw*, foi desenvolvido um compilador, de forma a receber como *input* um programa sintáticamente válido. Este programa vai ser validado pelo sistema de tipos e processado por um conjunto de algoritmos de otimização. Por fim, vai ser convertido para *bytecode*, de modo a poder ser executado na Máquina Virtual. Com este compilador pretende-se calcular em tempo de compilação o tamanho máximo que a aplicação vai ocupar no dispositivo. Assim, permite ao programador saber se o dispositivo tem espaço de armazenamento suficiente para armazenar a aplicação, permitindo contornar uma das limitações destes dispositivos.

Com esta linguagem, é possível resolver alguns dos problemas existentes relativamente a outras linguagens de programação. Ao poder ser utilizada em diversos dispositivos, permite que não seja necessário ter o conhecimento de todos os microcontroladores disponíveis. Uma outra vantagem importante, consiste em o compilador com o seu sistema de tipos identificar diversos erros cometidos pelo programador antes do programa ser executado.

Palavras-chave: Redes Sensores, Macroprogramação, Eventos, Compiladores, Algoritmos de Otimização

Abstract

Sensor networks consist of devices with limited resources and in most cases use batteries with lower capacity for power source. Many of these devices have a small amount of storage - only 2 KB of RAM - which illustrates their large computational limitations. Yet, these devices are used in many areas. For example, we can stress application in medicine, fire monitoring, military operations and even fire alarms within buildings.

The programming of these devices is related to its microcontroller, and thus dependent on the manufacturer that produces the devices. Given that manufacturers build devices with different microcontrollers are used several programming languages. The language we propose is based on events, or responds to stimuli. For example, if we have a device that is to do monitoring of fires and to determine that there is a fire, this device will react to this stimulus, the running of the program corresponding to this event.

For Macaw language, a compiler is designed so as to receive as input syntactically valid program. This program will be validated by the type system and processed by a set of optimization algorithms. Finally, it will be converted to bytecode, so it can be run in Virtual Machine. With this compiler is intended to calculate at compile time the maximum size that the application will occupy on the device. Thus allows the programmer to determine if the device has enough storage to store the application, allowing around one of the limitations of these devices.

With this language, it is possible to solve some of the problems relative to other programming languages. Power to be used on various devices, lets not necessary to have knowledge of all microcontrollers available. Another important advantage consists of the compiler with your system to identify various types of mistakes made by the programmer before the program run.

Keywords: Sensors Networking, Macroprogramming, Events, Compilers, Optimization Algorithms

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As redes de sensores são compostas por um conjunto de dispositivos capazes de medir variáveis do ambiente e tomar algumas ações relativamente a essas medições. São também capazes de transmitir dados entre dispositivos presentes na mesma rede. Estes dispositivos têm muitas limitações de processamento, de memória e de bateria. A possibilidade de algumas redes de sensores serem colocadas em zonas com pouca acessibilidade expõe ainda mais as dificuldades destes dispositivos e a resolução de possíveis problemas que possam existir. Se uma rede de sensores, é colocada num local em que a acessibilidade é escassa e o programa a ser executado tem uma falha, o acesso físico a esses sensores para a resolução desse problema pode apresentar custos muito elevados.

Atualmente existem muitas linguagens de programação para redes de sensores. Desde linguagens de alto nível que abstraem a informação de *hardware* dos programadores, a linguagens de baixo nível em que o programador necessita de definir o comportamento do *hardware* na rede. Outra característica das linguagens de programação, consiste em estas permitirem definir o comportamento geral da rede ou o comportamento de cada dispositivo individualmente. A maior dificuldade no uso destas linguagem consiste em garantir o bom funcionamento destas nas redes. Isto deve-se ao facto dos mecanismos de depuração ainda não estarem muito desenvolvidos.

Esta dissertação pretende desenvolver uma linguagem de programação de alto nível baseada em eventos que verifique estaticamente, em tempo de compilação, erros que possam dar origem a problemas de execução do programa. Este tipo de linguagem permite diminuir significativamente o tempo de desenvolvimento das aplicações, a sua manutenção e assim diminuir a possibilidade dos erros que possam acontecer durante a execução do programa. Para facilitar a depuração das aplicações desenvolvidas pela linguagem, é gerado em tempo de compilação um grafo que apresenta os vários estados em que o programa pode estar durante a sua execução.

Este trabalho foi realizado no âmbito do Projeto em Engenharia Informática do Mestrado em Engenharia Informática e também no âmbito do projeto *MACAW* do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa. Este projeto

é financiado pela Fundação para a Ciência e Tecnologia (PTDC/EIA-EIA/115730/2009). Além do desenvolvimento do compilador para a linguagem *Macaw*, o projeto também tem como objetivo desenvolver a máquina virtual para a execução da linguagem.

1.1 Motivação

As redes de sensores têm cada vez mais utilização, tanto pelo seu potencial, como pela facilidade de aplicação em diversos ambientes, sendo que muitos destes ambientes são críticos. Estas redes são utilizadas muitas vezes para controlar condições ambientais. As primeiras aplicações para redes de sensores começaram por ser aplicadas no ambiente militar, mas hoje em dia a sua utilização em diversos ambientes está a aumentar [20]. O uso destas redes num futuro próximo vai-se tornar cada vez mais importante, pois muitos dos processos que atualmente necessitam de ser controlados por um humano podem tornar-se automáticos devido à monitorização destes dispositivos.

Outra grande vantagem que estas redes de sensores têm é a facilidade com que se pode construir uma rede. Muitos dos dispositivos são modulares, ou seja, depois de se obter o corpo principal pode-se adicionar módulos a este corpo. Estes módulos podem ser desde receptores *wi-fi* para a comunicação com outros sensores, ou módulos com uma porta *HDMI* para a visualização dos dados no dispositivo. Um exemplo de um dispositivo com estas características é o *Raspberry Pi* [16].

Um grande conjunto das linguagens de programação para sensores são de baixo nível, muito próximas do *hardware*, o que dificulta a programação, pois é necessário ter em conta detalhes de baixo nível dos dispositivos que se está a programar. Outro problema que estas linguagens de programação apresentam é o facto de serem desenvolvidas para um *hardware* específico, tornando a sua aplicabilidade confinada aos dispositivos em questão. O leque de linguagens de programação para redes de sensores é muito grande.

A capacidade de processamento destes dispositivos é outro problema, pois possuem recursos escassos. É nossa convicção que num futuro próximo, quando estas limitações forem ultrapassadas, a utilização destes dispositivos vai aumentar exponencialmente. Estes podem vir a ser utilizados em ambientes adversos, onde os humanos não têm qualquer acesso e obter informações sobre esses locais. Com o aumento da aplicabilidade destes dispositivos, torna-se cada vez mais importante que o seu comportamento seja o esperado, trazendo assim uma maior confiança na sua utilização. Não se quer que um equipamento que esteja a fazer monitorização de uma situação crítica pare a meio porque a bateria sua terminou ou porque se esgotou o espaço na memória para armazenar os dados que está a controlar.

Por isso, é cada vez mais importante ter linguagens de programação que acompanhem a evolução das redes de sensores e que forneçam as abstrações adequadas a este modelo de programação. Assim, é relevante ter linguagens fáceis de entender e que permitam um

desenvolvimento rápido de aplicações, mas que também facilitem a fase de teste e que sejam conscientes das limitações físicas destes dispositivos. Isto contribui para a garantia de qualidade das aplicações.

Como foi referido anteriormente esta proposta pretende, desenvolver uma linguagem de programação que seja de alto nível, ou seja, possa abstrair o *hardware* e facilitar o trabalho de programação, tentando diminuir o número de erros que a aplicação possa ter. Também pretende aumentar a qualidade das aplicações desenvolvidas para estes dispositivos, facilitando a sua depuração. Devido aos dispositivos pertencentes na rede de sensores serem reativos a linguagem desenvolvida é baseada em eventos. Estes dispositivos são reativos pois reagem a estímulos externos. Pode ser desde a receção de uma mensagem de outro dispositivo na rede, a um fator ambiental que este dispositivo esteja a monitorizar.

Esta linguagem vai utilizar a noção de macroprogramação para abstrair o programador do *hardware* e os detalhes de rede. Outra verificação que vai fazer em tempo de compilação é o espaço total que a aplicação vai ocupar no dispositivo. Isto vai servir para que o compilador, quando for executado possa avisar o programador que o dispositivo não tem memória suficiente para essa aplicação. Devido a este problema de memória que os dispositivos têm, foram desenvolvidos algoritmos de optimização de forma a diminuir o tamanho do *bytecode* gerado pelo compilador. Aumentando assim, o leque de possíveis programas que podem ser definidos com a linguagem. Alguns dos algoritmos de optimização utilizados foram: *Conditional Constant Propagation*, *Constant Folding*, *Constant Propagation*, etc.

Uma vantagem desta linguagem é o facto de se poder fazer chamadas a funções de *hardware* definidas na Máquina Virtual. Para que esta funcionalidade seja possível, basta que a Máquina Virtual e o compilador contenham os mesmos dados relativamente à configuração do *hardware*. Esta característica facilita que a linguagem possa ser aplicada sobre vários tipos de *hardware*. Sempre que for necessário aplicar a linguagem a um novo *hardware*, basta apenas ser definido um novo ficheiro contendo as configurações de *hardware* que se vai utilizar. O facto da linguagem poder ser aplicada a vários tipos de *hardware* é uma das suas grandes vantagens.

1.2 Objetivos

O objetivo desta dissertação consiste em desenvolver uma linguagem de programação baseada em eventos que resolva alguns dos problemas identificados na secção anterior. Um dos objetivos específicos consiste em desenhar a linguagem e a sua semântica operacional. Para o desenho da linguagem, teve-se em conta que os dispositivos onde esta linguagem vai ser executada têm muitas limitações.

O desenho e a implementação de um sistema de tipos é outro objetivo específico. Esta característica da linguagem permite diminuir o número de erros que possam acontecer em tempo de execução. Este aspeto traz uma maior confiança na utilização da linguagem desenvolvida.

O terceiro objetivo consiste em determinar em tempo de compilação o tamanho máximo que a pilha de operandos vai ter durante a execução do programa. Por isso, desenvolvemos um analisador sintático para cálcular o tamanho máximo da pilha. Este analisador sintático é desenvolvido a partir da *AST* (*Abstract Syntax Tree*) gerada inicialmente pelo *plugin* utilizado nesta dissertação.

O desenho do *bytecode* teve de se analisar quais os tipos de instruções necessárias para a execução da linguagem no compilador, inclusive o formato do *bytecode*, que além das instruções do *bytecode*, tem de ter outras informações necessárias para o funcionamento da máquina virtual, sendo este o quarto objetivo.

O quinto objetivo consiste em executar o código gerado em diversos tipos de dispositivos, onde cada um deles tem características específicas. Assim, é necessário que o *bytecode* gerado seja o menor possível, aumentando assim a possibilidade de a linguagem poder ser executado em diversos dispositivos mesmo naqueles que têm menor capacidade. Teve de se ter em conta que algumas optimizações diminuem o código gerado mas outras aumentam o código gerado só para facilitar o trabalho da Máquina Virtual.

O sexto objetivo consise em desenvolver um grafo de estados do *hardware*, utilizando a especificação de *hardware* desenvolvida, de modo a que os programadores consigam analisar o comportamento do *hardware*. Este grafo permite identificar comportamentos errados por parte do *hardware* e assim ajudar na depuração das aplicações.

O último objetivo deste trabalho passa por desenvolver testes unitários para garantir o bom funcionamento do compilador. Os testes desenvolvidos utilizaram cobertura por grafos. Em métodos sem ciclos foi feita a cobertura de todos os caminhos do grafo, caso contrario foi feita cobertura dos caminhos primos. Estes testes foram feitos ao sistema de tipos da linguagem, aos algoritmos de optimização e a geração de código. Um outro objetivo é o de averiguar a expressividade da linguagem utilizando casos de uso com dimensão apreciável.

1.3 Calendarização

Esta dissertação teve a seguinte calendarização:

- Até fim de Outubro elaborar um estudo do estado da arte em programação de sensores e linguagem de programação baseada em eventos;
- Até fim de Janeiro elaborar o desenho da linguagem e implementação do analisador sintático;
- Até fim de Fevereiro elaborar o desenho do *bytecode*;
- Até fim de Março desenvolvimento de algoritmos de optimização no compilador;
- Até fim de Abril geração de código e desenvolvimento de testes unitários;
- Até fim de Maio implementação de um caso de uso para validação da solução desenvolvida;
- Até fim de Junho escrita da tese.

1.4 Contribuições

Foi desenvolvido um compilador para a linguagem *Macaw* que tem como objetivo diminuir o número de erros que possam acontecer em tempo de execução, diminuir ao máximo o tamanho do *bytecode* de modo a aumentar o leque de programas válidos para a linguagem, entre outros. Outra vantagem que o compilador apresenta, é o facto de calcular em tempo de compilação o tamanho de utilização de memória dos programas. O compilador e a Máquina Virtual da linguagem permitem que a linguagem possa ser executada em diversos tipos de *hardware*, o que é uma vantagem comparativamente às outras linguagens já existentes.

As contribuições deste trabalho são:

- Desenvolvimento de uma linguagem de programação baseada em eventos para sensores;
- Compilador utiliza algoritmos de optimização para diminuição do código gerado;
- Possibilidade de chamada de funções nativas do *hardware* na linguagem, o que permite aumentar a aplicabilidade da linguagem;
- Criar um compilador que permita, em tempo de compilação, determinar se o sensor tem espaço de memória suficiente para armazenar o programa ou se tem bateria suficiente para executar o mesmo;

- Desenvolvimento de um grafo de estados onde é possível identificar os diversos estados em que o *hardware* pode estar;
- Utilizando o resultado final dos algoritmos de otimização, é demonstrado no *Eclipse IDE* quais as mudanças possíveis no código fonte de modo a apróximar o código fonte do *bytecode* gerado pelo compilador.

1.5 Estrutura do documento

O documento está organizado em cinco capítulos para além desta introdução. O Capítulo 2 – Trabalho Relacionado – que descreve as linguagens de programação com intuitos semelhantes a esta proposta e faz um resumo de dois casos práticos onde as redes de sensores foram aplicadas. O Capítulo 3 – Linguagem de programação *Macaw* – apresenta uma descrição dos componentes principais da linguagem. O Capítulo 4 – Implementação Compilador *Macaw* – descreve os detalhes de implementação do compilador. O Capítulo 5 – Algoritmos de Optimização – descreve os algoritmos de otimização e a sua aplicação. O Capítulo 6 – Testes – descreve os testes desenvolvidos no compilador, os seus resultados e os erros encontrados. O Capítulo 7 – Exemplos Práticos – explica alguns exemplos práticos da linguagem *Macaw* e o Capítulo 8 – Conclusão – faz uma análise do trabalho desenvolvimento e são descritas as conclusões finais do projeto.

Capítulo 2

Trabalho relacionado

Neste capitulo faz-se uma análise das linguagens de programação para redes de sensores existentes tendo em conta o tipo de comunicação e o paradigma de programação das mesmas. Vai ser apresentada uma descrição prévia da linguagem de programação *Callas*, devido aos princípios desta linguagem serem semelhantes à linguagem desenvolvida nesta dissertação. Também vão ser apresentadas algumas plataformas que permitem a depuração e simulação de aplicações para redes de sensores. Para finalizar, vai ser feita uma análise das aplicabilidade das redes de sensores onde são apresentados casos de uso.

2.1 Tipo de comunicação

A comunicação entre sensores pode ser definida como um conjunto de nós da rede de sensores que trocam informação entre si para concretizar uma determinada aplicação. Há três abordagens principais: "comunicação entre vizinhos", "grupos *multi-hop*" e "comunicação entre todo o sistema" [28].

As linguagens de programação para a abordagem "comunicação entre vizinhos" permitem que os dados possam ser trocados apenas por sensores que estejam em contato direto entre si (utilizando por exemplo o sistema de rádio dos sensores).

A linguagem *nesC* e *ActiveMessages* são exemplos deste tipo de abordagem. As aplicações desenvolvidas com o *nesC* [18] são construídas interligando os sensores da rede através de interfaces. Nestas interfaces podem-se definir comandos ou eventos, onde os comandos são utilizados para começar operações e os eventos são utilizados para obter os resultados de forma assíncrona. O comportamento da linguagem *ActiveMessages* é um pouco diferente do *nesC*, pois esta linguagem utiliza mensagens com um identificador que especifica qual o sensor que deve receber a mensagem, tendo assim um comportamento semelhante aos *sockets* [28].

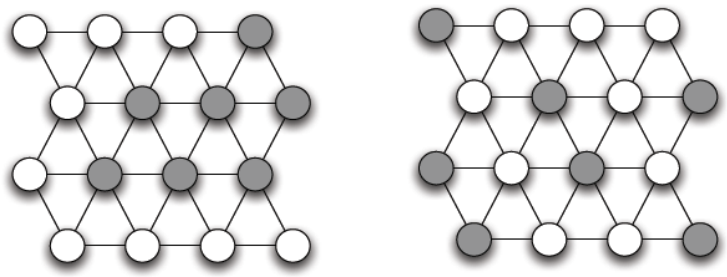
Os "grupos *multi-hop*" são conjuntos de sensores dentro de uma rede e podem ser divididos em dois grupos: "grupos *multi-hop*" ligados e não ligados. A diferença entre ambos está rebresentado na Figura 2.1 [28]. Os "grupos *multi-hop*" ligados permitem

a troca de mensagens entre sensores que não estejam ligados entre si desde que estejam dentro do mesmo *hop*. De realçar, que quaisquer dois nós que estejam neste grupo estão ligados, utilizando outros nós que pertençam ao mesmo grupo. Uma linguagem de programação utilizada para este caso é o *EnviroSuite*, é uma plataforma orientada a objetos, direcionada para fazer monitorização e aplicações para rastrear eventos. Os objetos nesta linguagem representam as entidades físicas (sensores) na rede. Estes objetos são criados dinamicamente quando as entidades são detetadas na rede [28].

Os sensores pertencentes aos "grupos *multi-hop*" não ligados entre si não têm qualquer assunção sobre o local dos nós que pertencem ao grupo. Uma linguagem utilizada neste caso é o *Logical Neighborhoods*. Esta permite aos programadores redefinir os vizinhos baseando-se nas propriedades lógicas dos sensores na rede [28].

Na "comunicação entre todo o sistema" todos os sensores na rede podem estar envolvidos na troca de dados (também pode ser definido como macroprogramação). Uma linguagem utilizada para este caso é o *TinyDB* [25]. O utilizador introduz expressões *SQL* na estação base da rede e esta envia para todos os sensores essas consultas. Quando um sensor recebe a consulta, este processa-a fazendo algumas leituras do ambiente, e se for necessário encapsula os dados enviando-os de novo para a estação base. A estação base depois de receber os dados de todos os sensores faz o tratamento destes e devolve o resultado ao utilizador que inseriu as consultas.

Todos estes tipos de comunicação têm as suas vantagens e desvantagens. Quando se quer desenvolver alguma aplicação para redes de sensores deve-se escolher o mais indicado para a aplicação pretendida. Algumas das linguagens definidas anteriormente são de baixo nível (como por exemplo o *nesC*), enquanto outras são de alto nível (como por exemplo o *EnviroSuite*).



(a) *Multi-Hop* ligados (b) *Multi-Hop* não ligados entre si

Figura 2.1: Grupos *Multi-Hop*

A linguagem que vai ser desenvolvida incorporar-se neste último grupo "comunicação entre todo o sistema", pois vai ser possível definir o comportamento de toda a rede de sensores e não individualmente. Uma das grandes vantagens deste tipo de comunicação consiste no programador não ter de se preocupar em programar cada um dos nós da rede (se for uma rede muito grande ter de programar cada um dos nós pode ser complicado) e

assim trazer alguma abstração no programa.

2.2 Paradigmas de programação

As linguagens de programação utilizadas nas redes de sensores podem-se dividir em três paradigmas diferentes: imperativo, declarativo e híbrido.

Em relação ao paradigma de programação imperativo o processamento da aplicação é expresso através de instruções que indicam claramente como mudar o estado do sistema. Este paradigma pode ser dividido em duas categorias: sequencial e baseado em eventos. Como vimos anteriormente o *nesC* é baseado em eventos e um exemplo de linguagem sequencial é *Pleiades*. Esta linguagem é uma extensão ao *C* em que é possível indicar quais são os nós na rede e aceder ao seu estado local [28]. As linguagens de programação neste paradigma têm a vantagem de que o programador consegue controlar facilmente o comportamento do sistema.

No paradigma de programação declarativo o objetivo da aplicação é descrito sem indicar como é que é cumprido. Este paradigma pode ser dividido em três categorias: funcional, baseado em regras e utilizando consultas *SQL*. Como já vimos anteriormente o *TinyDB* é um exemplo de uma linguagem que utiliza consultas *SQL* no seu funcionamento. Uma linguagem que é do tipo funcional é o *Regiment* [29]. Nesta linguagem os programadores manipulam conjuntos de dados chamados sinais. Esta linguagem também tem conhecimento sobre regiões e aplica as funções programadas aos sinais em certas regiões da rede. Um exemplo de uma linguagem utilizada baseada em regras é o *Snlog* [6]. Esta utiliza predicados, tuplos, factos e regras. Os predicados correspondem a esquemas para representar os dados, os tuplos são utilizados para a transmissão de dados entre os sensores, os factos são tuplos particulares que são instanciados durante o início do sistema e as regras consistem em ativar os factos da linguagem [28]. As linguagens de programação indicadas anteriormente têm uma desvantagem relativamente às linguagens do paradigma imperativo. Esta desvantagem deve-se ao facto de no paradigma declarativo não se conseguir controlar o comportamento do sistema, dificultando assim o processo de depuração das aplicações definidas neste paradigma.

O paradigma híbrido pode combinar diversos tipos de paradigmas de programação. Temos como exemplo a linguagem *Abstract Task Graph* [4] que utiliza o paradigma declarativo e imperativo.

A linguagem que vai ser desenvolvida é do tipo imperativo baseada em eventos como já foi referido. Basear-se-á em eventos para tomar ações. Em termos de compreensão do programa e do seu controlo de fluxo a programação baseada em eventos é um paradigma fácil de entender, em comparação com o paradigma orientado a regras, que torna complicado especificar as regras da rede e entender o controlo de fluxo dos programas definidos com esta linguagem.

2.3 Linguagem de programação *Callas*

A linguagem de programação *Callas* tem um objetivo semelhante ao objetivo da linguagem *Macaw* desenvolvida nesta dissertação. Esta linguagem foi desenvolvida com o intuito de facilitar o desenvolvimento de aplicações para redes de sensores.

A linguagem *Callas* é baseada em eventos e é compilada por um compilador "*type-safe*" e "*subject reduction*", permitindo assim diminuir o número de erros que possam existir durante a execução da linguagem [26]. A grande diferença que esta linguagem tem em comparação com a desenvolvida nesta dissertação, é relativo ao tipo de comunicação que é permitida pela linguagem. A linguagem *Callas* não utiliza o conceito de macroprogramação, permitindo ao programador não só definir o comportamento de cada sensor na rede, mas também o protocolo de rede que vai ser utilizado. Uma outra diferença que esta linguagem tem relativamente à linguagem *Macaw* é que permite programação recursiva.

A linguagem *Callas* consegue determinar em tempo de compilação se existe algum problema no protocolo utilizado na comunicação entre os sensores, permitindo assim diminuir alguns erros que possam existir. Um sensor de rede nesta linguagem é representado por um processo que está a correr no sensor, uma fila que contém os eventos que vão ser executados no sensor, o programa que vai ser executado no sensor (sendo este guardado em memória) e por fim, uma tabela com temporizadores para as chamadas de funções. Devido ao tipo de comunicação desta linguagem, estes dados têm de ser definidos para todos os sensores da rede [26].

O compilador desta linguagem gera dois tipos de *bytecode*. Um tipo para os sensores "*sink*" – estes sensores são ligados por exemplo a um computador permitindo obter e manipular dados do ambiente. O outro *bytecode* é criado para dispositivos do tipo "*samplers*". Depois de este *bytecode* ser gerado, é transformado num ficheiro *.jar* ou *.suite* que pode ser executado nos sensores da rede. Este *bytecode* vai ser executado no sensor alvo utilizando o *SunSpot*. Assim que o *SunSpot* começar a sua execução, a máquina virtual da linguagem *Callas* começa a executar o *bytecode* gerado por parte do compilador [27].

2.4 Testes e depuração de aplicações

Os testes e a depuração de aplicações é uma área ainda muito pouco explorada na programação em sensores. Algumas das soluções apresentadas no estado de arte das redes de sensores oferecem algumas ideias para testar aplicações de redes de sensores, mas estas são dependentes de uma arquitetura de sensores, o que limita bastante a utilização destas soluções. Mesmo assim, nestes exemplos apresentados é mostrado ao programador um erro, mas não dão nenhuma pista sobre a razão de o programa ter parado a sua execução [28]. Atualmente o modo mais utilizado para testar o comportamento das aplicações numa rede de sensor é simular o comportamento da rede. Mas esta aproximação não é suficiente, pois alguns erros só são encontrados em casos específicos de utilização. De resto, como

se constatou nos capítulos anteriores, as linguagens apresentadas estão mais preocupadas na gestão da rede e nos algoritmos a utilizar (por exemplo, difusão de mensagens) do que nestes pormenores de garantia de qualidade.

Isto dificulta o trabalho dos programadores pois estes têm dificuldades em garantir que as aplicações que estão a desenvolver têm o comportamento desejado, o que traz diversos problemas. Se o problema for identificado só quando a rede de sensores tiver sido construída, num local com pouco acesso, o deslocamento até ao local para a resolução do problema pode apresentar custos muito elevados.

A linguagem que vai ser desenvolvida de modo a facilitar a depuração das aplicações apresenta um grafo, onde indica como o estado do *hardware* dos seus componentes se vai comportar ao longo da execução do programa. Por exemplo, quando um evento é executado num certo estado, este grafo demonstra o que vai acontecer aos módulos de *hardware* a seguir ao evento ser executado. Isto facilita a identificação de qualquer problema que a aplicação possa ter em tempo de execução. De seguida vão ser apresentados alguns exemplos de linguagens e aplicações que pretendem facilitar o trabalho de depuração das aplicações.

2.4.1 *NodeMD*

NodeMD [21] é uma aplicação de *deployment* que permite analisar alguns problemas que podem acontecer durante a execução de programas num sensor da rede. O objetivo desta aplicação é identificar o erro antes que este incapacite totalmente o sensor. Depois de os problemas serem identificados, esta aplicação elabora um diagnóstico que permite aos programadores resolverem o problema encontrado, permitindo assim diminuir os custos relativos à necessidade de ir ao local onde o sensor está colocado, para este ser substituído. Os autores desta aplicação pretendem que esta analise e diagnostico afete o menos possível o comportamento da rede de sensores.

Esta aplicação tem três subsistemas diferentes [21]. O primeiro é relativo à detecção de erros na execução de um determinado sensor. Os erros identificados são: *Stack Overflow*, *DeadLock* e *LiveLock* [21]. O algoritmo para verificar *Stack Overflow* consiste em comparar a pilha da *Thread* que está a ser executada atualmente com o *Stack Pointer* depois de um procedimento ser chamado [21]. Se o *Stack Pointer* exceder o tamanho de pilha atual isso vai causar *Stack Overflow*. *DeadLock* consiste em dois ou mais processos que estão à espera um do outro para poder continuar a sua execução, mas nenhum deles termina. Esta condição de *DeadLock* é muito semelhante ao *LiveLock*, só que no caso do último os processos não estão bloqueados. Cada um dos processos está a mudar de estado para deixar o outro que está à espera avançar, mas ambos mudam ao mesmo tempo. Para determinar casos de *DeadLock* e *LiveLock*, foi desenvolvido um algoritmo que introduz *checkpoints* no código para determinar se as *Threads* do sistema está a ter o comportamento desejado [21].

O segundo subsistema, consiste em notificar ao programador qual a falta que aconteceu durante a execução do programa num determinado sensor da rede. O sumário que é desenvolvido e posteriormente enviado para o programador consiste num histórico de execução dos eventos no sistema [21].

O terceiro subsistema, permite que os programadores consigam interagir diretamente com o nó que teve a falta, permitindo assim que o programador consiga resolver o problema. Os autores desta aplicação pretendem que quando exista uma falta no código poder atualizar o código do sensor que contém a falta [21].

2.4.2 *Clairvoyant*

Clairvoyant [32] é uma aplicação de depuração para redes de sensores. Este liga-se a um sensor da rede usando a conexão wireless do dispositivo, e permite executar comandos de depuração *standard*. Além destes comandos *standard* também existem comandos que permitem não só testar o *hardware* do dispositivo, como por exemplo, o acesso ao *LED* do dispositivo. Podem também permitir, aceder ao comportamento total da rede de sensores. Esta aplicação pretende que a sua execução não traga qualquer problema nem distorção do comportamento do sensor que vai ser testado [32]. Esta técnica utilizada por o *Clairvoyant* consiste em depuração remota, ou seja, permite testar dispositivos sem que estes estejam fisicamente conectados.

Esta plataforma tem as suas vantagens e desvantagens. A principal vantagem é que o código fonte a ser executado no sensor não precisa de ser alterado para que possa ser testado. Uma desvantagem desta aplicação é o facto de este ter de ser instalado previamente no sensor. Este ocupa 32 *KB* da memória do programa e 1 *KB* de memória dos dados. Devido à limitação de memória que os sensores apresentam, se um programa necessitar de ocupar toda a memória do sensor, este programa não vai poder ser testado [32].

Alguns dos comandos existentes no *Clairvoyant* permitem, adicionar *breakpoint* no código e eliminar o mesmo, produzir *step* no código para analisar instrução a instrução do código, entre outros. O comando *stop* permite que o sensor entre em modo de depuração, Além destes comandos para analisar o código do sensor existem certos comandos que permitem aceder ao *hardware* do sensor. Por exemplo, o comando *interrupts* identifica as interrupções que acontecem durante a execução do código. Este comando permite, identificar condições de corrida que possam existir no código [32].

Devido à grande facilidade de comunicação entre os sensores numa rede, é importante conseguir testar o comportamento da rede como um todo. E para poder realizar estes testes o *Clairvoyant* apresenta comandos para os efetuar. O comportamento destes testes é semelhante ao serem aplicados a um sensor individual, só que são aplicados à rede toda [32].

2.4.3 *TOSSIM*

A ferramenta *TOSSIM* [23] permite simular o comportamento de redes de sensores que utilizem o sistema operativo *TinyOS* [22]. Este simulador tem um comportamento bastante fiável, quer para uma rede com poucos sensores, quer para uma rede com centenas de sensores. Como já vimos anteriormente, as redes de sensores têm características únicas que trazem incerteza à execução de um programa e que podem levar ao mau funcionamento das aplicações, tais como: factores ambientais extremos, problema de comunicação entre sensores, etc. Devido a isto, é muito importante que antes da rede de sensores seja colocado no seu local de ação esteja correta sem qualquer erro.

A simulação é uma forma de teste bastante usada para testar aplicações pois permite encontrar erros no comportamento da rede que vai ser simulada. No caso do *TinyOS* isto é importante pois utiliza linguagem baseada em eventos. A rede de sensores na plataforma *TOSSIM* é representado por um grafo, onde os nós do grafo correspondem a um sensor e os vértices à ligação entre os nós da rede [23]. Devido às características do *TinyOS*, o simulador *TOSSIM* tem de ter quatro características: escalabilidade, completude, fidelidade e *bridging* [23].

As redes de sensores construídas com o *TinyOS* são facilmente escaláveis. Por isso, o simulador tem de conseguir simular uma grande quantidade de sensores na rede. Este aspeto traz escalabilidade ao simulador. A completude do simulador é necessária para poder simular o máximo de interações possíveis entre os nós da rede. A fidelidade do sistema de simulação é garantido se este conseguir simular sem qualquer erro o comportamento num nó individual e entre nós. A capacidade de *bridging* permite que os programadores consigam depurar o código implementado que vai ser executado nos sensores [23].

Este simulador permite simular, por exemplo, o *hardware* de um sensor na rede. Entre este *hardware* está o relógio interno do sensor, a sequencia de inicio do sensor, entre outros. Permite também simular a comunicação entre os sensores da rede, definindo a eficácia de transmissão dos dados entre os sensores da rede [23]. Esta característica é importante para testar a comunicação em ambientes mais complexos.

Esta plataforma tem a vantagem de permitir que outras aplicações se conectem a ela através do uso de *socket TCP/IP*. Isto permite que, uma aplicação de depura-ção se possa ligar a este e possa testar o código que esteja a ser executado. Uma aplicação que faz uso destas funcionalidades é o *TinyViz*, esta é uma aplicação gráfica que permite visualizar, analisar e controlar a execução do simulador [23]. A desvantagem que este sistema apresenta é a suposição de que todos os nós da rede estão a executar o mesmo *software*.

2.5 Aplicação das redes de sensores

Esta secção é constituída por duas subsecções. Na primeira é feito um pequeno resumo dos diversos tipos de aplicações em que as redes de sensores podem ser aplicadas. Relativamente à segunda secção, são exemplificados exemplos práticos de aplicações de redes de sensores.

2.5.1 Classificação da aplicabilidade de redes de sensores

As redes de sensores são aplicadas em diversas áreas e existem muitos casos em que investigadores desenvolveram linguagens de programação específicas para monitorizar certos aspetos. Devido à grande variedade de ambientes onde as redes de sensores podem ser aplicadas, é necessário fazer uma avaliação dos tipos de requisitos que a rede de sensores vai precisar. Utilizando a figura 1 em [28], a aplicabilidade das redes de sensores pode ser classificada usando os seguintes aspectos: Objetivo da rede de sensores, padrão de interação, mobilidade da rede, o espaço e o tempo.

O objetivo da rede é uma característica muito importante quando se quer aplicar uma rede de sensores num determinado ambiente. A rede pode ter como objetivo apenas obter os dados relativos ao ambiente em que está a ser aplicada. Uma rede que esteja a monitorizar a meteorologia para que depois os meteorólogos possam analisar os dados, é um exemplo de uma rede que tem este objetivo. Outro objetivo é obter os dados e tomar certas ações relativamente aos dados obtidos. Os sensores que estão nos prédios a monitorizar fumos e possíveis incêndios, são um exemplo deste tipo de redes. Isto porque, depois de analisarem se existe a possibilidade de haver um incêndio, os aspersores são ligados para o apagar [28].

O padrão de interação de uma rede de sensores pode ser dividido em três categorias: "um para muitos", "muitos para muitos" e "muitos para um" [28]. A primeira categoria ("um para muitos"), é importante para ambientes em que seja necessário ter um sensor que envie certos comandos para os outros sensores da rede [28]. A categoria "muitos para muitos", mais conhecida como macroprogramação, permite que todos os sensores na rede possam interagir entre si. Esta categoria, tem a vantagem de facilitar o trabalho dos programadores, pois podem definir o comportamento da rede como um todo em vez de o fazerem individualmente [28]. Por fim, "muitos para um" é muito importante em ambientes onde os sensores recebem dados do ambiente e enviam os dados para um sensor central. Este sensor posteriormente vai analisar os dados recebidos e tomar alguma ação relativamente a esses dados [28].

As redes de sensores podem ser estáticas em que os sensores estão estáticos desde o momento em que a rede é colocada no ambiente. Esta modalidade é a mais utilizada

atualmente. As redes também podem ter alguns dos sensores da rede dinâmicos, podendo estes sensores estar colocados em animais ou *robots* [24]. Os "*mobile sink*" podem ser ou estáticas ou móveis. O papel mais importante destes sensores é a forma com os dados são obtidos. Os dados iram ser obtidos quando estes "*mobile sink*" se aproximam dos sensores [28]. Entre estas três categorias, as redes de sensores estáticas são as mais utilizadas. A sua utilização prende-se por ser mais fácil de ser aplicada no ambiente de utilização. Sensores que sejam colocados em animais ou *robots* são claramente muito complicados de se controlar.

Como já foi referido anteriormente, as aplicações desenvolvidas para redes de sensores podem ser locais na rede, ou seja, podem envolver apenas alguns dos sensores na rede. Para este caso pode-se utilizar algumas linguagens de programação, tais como: *EnviroSuite* e *Logical Neighborhoods*. Mas também podem envolver todos os sensores na rede.

As aplicações desenvolvidas para redes de sensores podem ser baseada em eventos onde os sensores ficam a aguardar que algum evento aconteça para poder tomar uma ação. É de realçar que a linguagem desenvolvida nesta dissertação se encontra neste tipo de aplicabilidade. As aplicações também podem ter uma aplicação continua. Isto é, os sensores na rede vão estar continuamente a processar dados, e se for o caso, executar ações relativas aos dados obtidos.

2.5.2 Exemplos práticos da aplicabilidade das redes de sensores

Um grupo de investigadores aplicou um conjunto de sensores para avaliar o nível de deterioração da torre Aquila em Itália. Esta torre está localizada na entrada principal da cidade e devido ao constante fluxo de carros a entrar e sair da cidade, trouxe problemas para a estrutura desta torre. Para diminuir este problema foi aprovada a construção de um túnel subterrâneo para diminuir o fluxo de carros. Então este grupo de investigadores quis avaliar com uma rede de sensores qual o impacto na estrutura da torre deste túnel [13].

Na floresta do Arizona estes sensores também têm um papel importante. Uma rede de sensores foi colocada numa região remota que tem grandes dificuldades de acesso. Sendo assim, se ocorresse um incêndio nessa área, esta iria arder durante horas até que fosse possível identificar o incêndio. Com a colocação de uma rede de sensores no local, assim que um sensor determinar um possível fogo, comunica com os outros sensores da rede de modo a formar um perímetro à volta desse local. Depois de este perímetro estar definido, envia um sinal via Internet para os bombeiros mais perto do local. Quando os bombeiros chegam ao local, injetam na rede agentes de procura para identificar se existe alguém em perigo devido ao incêndio. Os sensores voltam ao seu estado normal assim que o incêndio é extinto, ou seja, ficam a monitorizar as variáveis do ambiente de modo a identificar se existe algum novo problema [15].

Um outro caso, mas este mais simples que os anteriores, em que podem ser utilizados

é nos sensores que são acionados quando existe um fogo num edifício e começa a apagar o incêndio com água. Com estes exemplos consegue-se perceber a versatilidade das redes de sensores e a sua importância. Podem ajudar em muitos casos e resolver situações complicadas [20].

2.6 Considerações finais

Devido à grande complexidade das redes de sensores, hoje em dia é cada vez mais importante o desenvolvimento de linguagens de programação que facilitem o desenvolvimento de aplicações. Os custos relacionados com o desenvolvimento e colocação das redes nos ambientes alvo são muito elevados por isso este aspeto é muito importante.

Atualmente existem muitas aplicações que podem facilitar a depuração de aplicações desenvolvidas para redes de sensores. O aspeto negativo destas aplicações é que estão desenvolvidas para uma única arquitetura de sensores. Torna-se assim complicado, pois sempre que se queira desenvolver uma aplicação para um sensor diferente, tem de se conhecer a aplicação que irá ser necessária para depuração da aplicação. Se existir uma única aplicação que seja genérica o suficiente para conseguir funcionar para todas as arquiteturas de sensores, é um aspeto positivo. Mas estes aspetos negativos não são só para aplicações de depuração de linguagens, mas também para as linguagens. As linguagens para sensores são desenvolvidas para um *hardware* específico. Os programadores têm de ter conhecimento suficiente sobre o estado da arte das linguagens de programação, para que se for necessário alterar os dispositivos em que estão a trabalhar estes saberem como os programar.

Capítulo 3

Linguagem de programação *Macaw*

O desenho de uma linguagem de programação é um processo desafiante. Em cada momento debatemo-nos com quais as funcionalidades a incluir em função da expressividade que pretendemos, dos recursos de computação que estão disponíveis e quais as garantias que pretendemos que os programas satisfaçam.

Ao termos a sintaxe definida foi necessário desenvolver a sua semântica operacional e o seu sistema de tipos.

O formato do *bytecode* e a linguagem intermédia trouxeram novos problemas relativamente à limitação de memória que o *hardware* apresenta. Foi necessário perceber que informação do código fonte é que seria necessário utilizar para o bom funcionamento da Máquina Virtual.

Este capitulo apresenta ao detalhe o processo de desenvolvimento da linguagem *Macaw*.

3.1 Sintaxe da linguagem

O desenho da sintaxe da linguagem teve de ter vários fatores em causa. Um deles foi relativamente ao tipo de *hardware* onde a linguagem vai ser utilizada. Devido aos diversos problemas que estes dispositivos apresentam, a linguagem teve de ter certos cuidados. Não se podia utilizar instruções que pudessem vir a esgotar rapidamente a memória do sensor. Isto faria com que a aplicabilidade da linguagem fosse bastante pequena. Uma outra funcionalidade que esta linguagem não poderia permitir é a programação recursiva pois, se isto fosse possível, a memória do sensor iria-se esgotar facilmente e teríamos novamente o problema de memória dos sensores. Como um dos objetivos da linguagem consiste em identificar alguns consumos energéticos estes dados estão presentes na linguagem para fazer essa análise.

Um programa escrito em linguagem *Macaw* é constituída por dois ficheiros, um *.macawh* e outro *.macaw*. O ficheiro *.macawh*, é utilizado para descrever o *hardware* dos sensores em que a linguagem vai ser aplicada. Esta definição de *hardware* consiste em

definir os módulos de *hardware* do dispositivo que vai ser utilizado. Por exemplo, se o dispositivo que vai ser utilizado tem um módulo de *GPRS* e um módulo *Solenoid* e ambos vão ser utilizados para a programação da rede, então estes têm de ser definidos na secção "*Hardware States*". Se fosse o caso de o módulo *GPRS* não ser utilizado, então não seria necessário ser definido nesta secção. Associado à definição de um módulo está consumo energético, indicando o valor normal e o máximo que esse módulo irá consumir.

A segunda secção ("*Boot*") é utilizada para definir quais os módulos que vão estar ligados ou desligados quando a aplicação arrancar. Os módulos aqui indicados são aqueles que foram definidos na secção anterior.

A terceira e última secção indica quais os eventos e as funções de *hardware* que vão ser utilizadas. Um evento ocorre quando uma ação externa é executada. Por exemplo, se existe um evento chamado *buttonPressed()*, este vai ser acionado quando alguém carregar no botão do sensor. Para estes eventos é necessário indicar a sua assintaura. Relativamente às funções de *hardware*, tal como os eventos, é necessário ser definido a sua assinatura. O seu desenvolvimento é produzido na máquina virtual da linguagem. Estas são as funções nativas do *hardware* utilizado. Uma outra diferença relativamente a estas funções nativas comparativamente aos eventos é que podem retornar valores. Quer para os eventos quer para as funções de *hardware* é possível definir cláusulas "*when*", "*exec*" e "*turns*". A cláusula "*when*" serve para determinar quando um certo evento ou função de *hardware* podem ser executadas, identificando o estado em que os módulos de *hardware* necessitam de estar para que este possa serem executados. A cláusula "*exec*", é utilizada para determinar os consumos energéticos, o valor indicado consiste no consumo energético do *hardware* ao executar o determinado evento ou função. A cláusula "*turns*", identifica em que estado o *hardware* vai ficar depois de o evento ou a função de *hardware* ser executado. As cláusulas "*when*" e "*turns*" podem ser vistas como uma pré-condição e uma pós-condição do método, respetivamente. Estas três últimas cláusulas ("*when*", "*exec*" e "*turns*") são opcionais. De realçar que os eventos têm de ser definidos antes das funções de *hardware*.

Este ficheiro permite que o programador, que vá desenvolver o *software* das redes de sensores não necessite de ter o conhecimento total do *hardware* do sensor que vai ser utilizado. Também permite, que este ficheiro seja desenvolvido pela mesma pessoa que desenvolveu a Máquina Virtual, permitindo assim garantir que este ficheiro é desenvolvido de acordo com a especificação utilizada na Máquina Virtual. A sintaxe detalhada para este ficheiro pode ser consultada na Figura A.1.

O ficheiro *.macaw* permite desenvolver o *software* que vai ser utilizado na rede de sensores. Inicialmente, é necessário indicar qual o ficheiro *.macawh* que vai ser utilizado. Esta localização é indicada na secção "*hardware description*". Posteriormente a esta secção, é possível indicar as variáveis globais necessárias para o desenvolvimento do programa. A novidade em relação às variáveis globais é o facto da inicialização de *array*

poder ter "...". Ao definir esta "*string*", as restantes posições do *array* vão ser preenchidas automáticamente. Estas posições vão ser preenchidas com o valor definição para o tipo do *array*. Por exemplo, se um *array* for do tipo "*int8*" com 50 posições e apenas forem preenchidas 2 posições, as restantes posições vão ser preenchidas com o valor 0. Os membros do programa são de dois tipos diferente, funções ou "*handler*". As funções têm um comportamento esperado de uma função, enquanto que os "*handler*" são apenas chamados quando um evento é acionado. Os membros têm de ter o seu corpo bem definido. Inicialmente, é necessário indicar as variáveis locais do membro, seguindo-se as instruções e por fim depois de estas estarem definidas pode-se indicar o comando de retorno. A definição de variáveis locais é feito exatamente da mesma forma que as variáveis globais.

As instruções que se podem utilizar são as atribuições de valores a variáveis, a chamada de funções, o comando "*attach*" que permite fazer a ligação entre um evento e um "*handler*" do programa, ou seja, quando o evento for acionado o "*handler*" correspondente vai ser acionado, o comando "*unfold*" permite executar o seu corpo um número determinado de vezes, o comando "*if*" é um comando condicional que permite avaliar expressões e por fim, o comando "*ifStateContains*" permite avaliar se o estado do *hardware* naquele ponto é verdadeiro ou falso.

As expressões que podem ser utilizadas são as que são utilizadas em todas as linguagens do tipo imperativo. Pode-se utilizar expressões relacionais, aditivas e multiplicativas.

Relativamente aos tipos da linguagem, de realçar que o *array*, apenas permite um inteiro de 8 *bits* para representar o seu tamanho, esta decisão teve em base não só a memória disponível por parte dos dispositivos da rede, mas também por acharmos que para o tipo de *hardware* em que a linguagem vai ser executada um *array* com 128 posições é suficiente. Como tipos também temos os tipos constantes, estes são definidos apenas uma vez e o seu valor não pode ser alterado novamente ao longo do programa.

Os tipos primitivos "*int8*", "*int16*" e "*int32*" são *signed*, ou seja, podem estar entre os valores indicados na tabela 3.1. O tipo "*boolean*" e o tipo "*float*" podem ter os valores esperados. A sintaxe detalhada para este ficheiro pode ser consultada na Figura A.2.

Uma das características que se queria desta linguagem é que fosse simplista o suficiente para que qualquer pessoa com pouco conhecimento de programação a pudesse utilizar. E por isto mesmo, as instruções e características da linguagem foram mantidas o mais simples possível. Uma outra vantagem que esta linguagem apresenta é o facto de ser facilmente utilizada em vários tipos de sensores. Estas funções só necessitam de estar definidas na Máquina Virtual da linguagem e depois consiste apenas em chamar essas mesmas funções.

Tipo	Valor Mínimo	Valor Máximo
int8	-128	127
int16	-32 768	32 767
int32	-2 147 483 648	2 147 483 647

Tabela 3.1: Diferença de valores de inteiros

3.2 Semântica operacional

O desenho da semântica operacional da linguagem *Macaw* foi um aspeto necessário de modo a provar que o funcionamento da linguagem é correto.

A semântica operacional consiste em desenvolver provas de modo a validar o comportamento correto da linguagem. Estas provas podem ser desenvolvidas de uma forma informal utilizando código máquina por exemplo, mas também podem ser representados de uma forma formal utilizando uma semântica operacional ou interpretador da linguagem. Nestes modelos formais é definido um estado do programa, e as provas consistem em alterar esses estados tendo em conta alguns comandos da linguagem [14]. Por exemplo, a regra *Assign-Simple-Global*, consiste em alterar o estado do programa quando existe uma atribuição de valor a uma variável global.

O primeiro passo para desenvolver uma semântica operacional da linguagem consiste em representar o que é um estado de execução da linguagem. Este estado é definido utilizando a sintaxe de ambiente de execução da linguage *Macaw* representada na figura B.1. Esta sintaxe é constituída por um dispositivo, contexto global do programa, pelo código do programa, por um contexto local, uma pilha de chamadas, uma pilha de eventos e por fim a tabela de despacho.

- O dispotivo é utilizado para representar um estado de execução da linguagem *Macaw*;
- O contexto global (D) do programa consiste em todas as definições de variáveis globais do programa, sendo estas representadas pelo seu nome (x), a sua posição (i) e o seu valor (v);
- Os membros (T) correspondem a todos os membros definidos no programa. São representados pelo seu nome (y), por uma lista de variáveis ($\vec{x}$) e pelo corpo do membro (c);
- A pilha de chamadas (S), representa os membros que são chamados, este é representado por vários códigos locais. O código local corresponde à execução de comandos do corpo do membro, este é constituído pelo corpo do método (c) e pelo contexto local do membro (D);

- A pilha de eventos (Q) vai conter os eventos que foram definidos no ficheiro *.macawh*. Estes eventos são representados pelo seu nome (x) e por uma lista de valores ($\vec{v}$) referente aos tipos dos seus parâmetros;

- A tabela de despacho (H) liga os eventos com os seus "*handler*", esta ligação é produzida entre o nome único do evento (x) e o nome do "*handler*" (y) que vai ser executado quando o evento for acionado.

Tendo em conta o estado de execução da linguagem *Macaw*, definiu-se o estado inicial da linguagem *Macaw*. Este estado está descrito na figura B.2.

O estado inicial da linguagem é definido com todas as variáveis globais do programa (D_0), contém todos os membros definidos do programa (T_0), inicialmente não tem nenhum membro a ser executado, contém o evento inicial que vai iniciar o programa, este evento não tem qualquer parâmetro por isso a sua lista de valores está vazia e a tabela de despacho inicialmente faz a ligação entre o evento de arranque da aplicação e o " *handler*" *main*.

A primeira regra (*Bind*), identifica a mudança de estado quando ocorre um comando "*attach*" durante a execução de um membro. Quando este comando é executado vai ser introduzida uma ligação entre o evento x e o "*handler*" y na tabela de despacho. Este comando não altera nada no domínio local do membro depois de ser executado.

A regra *unfold-out*, demonstra o comportamento do comando "*unfold*" quando a execução deste comando termina. Ao avaliar a expressão do comando "*unfold*", se o resultado for zero, significa que o comando terminou a sua execução e por isso mesmo o resto do código (c_2) vai ser executado. Este comando "*unfold*" depois de ser executado não faz qualquer outra alteração no estado da linguagem.

A regra *unfold-in*, prova o comportamento do comando "*unfold*" quando este está a ser executado. Se a expressão do comando quando for avaliada for maior que zero, isto significa que o comando ainda está a ser executado. O comportamento consiste em executar o corpo do "*unfold*" (c_1) uma vez, e decrementar em uma unidade a expressão do comando.

A quarta regra *if-then-else-false*, prova o comportamento do comando "*if*" quando a sua expressão tem o resultado falso. A expressão do comando "*if*" ao ser avaliada, se o seu resultado for falso, então vai ser executado o código referente ao comando "*else*". Sendo assim, c_2 vai ser executado seguido de c_3 e do resto do corpo do membro.

A regra *if-then-else-true*, tem o mesmo objetivo de comprovar o comando "*if*", mas esta regra consiste no caso de a expressão ser avaliada como verdadeira. A diferença relativamente à regra anterior, é que em vez de ser executado o código referente ao "*else*", vai ser executado o código referente ao "*then*". Sendo assim, a sequencia de execução irá ser c_1 sendo posteriormente executado c_3 e o resto do membro.

Tendo em conta as duas regras anteriores (*if-then-else-false* e *if-then-else-true*), as

duas seguintes regras (*IfStateContains-True* e *IfStateContains-False*) têm o mesmo objetivo. A diferença consiste em que estas são utilizadas para comprovar o comportamento do comando "*ifStateContains*".

A regra *Call*, prova o comportamento da linguagem quando existe uma chamada a um membro. Quando existe uma chamada a uma função no código que está a ser executado, o código local desse membro vai ser adicionado ao topo da pilha de chamadas, sendo esse membro o que irá ser executado. Este código local, vai ter o código do membro e o domínio irá ter os valores atribuídos na chamada do membro.

A regra *Return*, comprova o comportamento da linguagem no fim da execução de um membro quando é encontrado um comando "*return*". Ao ser encontrado este comando, o código local referente a esse membro vai ser retirado da pilha de chamadas, sendo executado o seguinte código local que está presente na pilha de chamadas.

A regra *event*, consiste em adicionar um evento à pilha de eventos quando um evento é encontrado. Adicionado o seu identificador único e os tipos dos seus parâmetros.

A regra *Fire*, prova o comportamento da linguagem quando um evento é acionado. Inicialmente o evento está na pilha de eventos, este ao ser acionado vai adicionar ao código local o corpo referente ao seu "*handler*" e o domínio local vai conter os parâmetros do "*handler*".

A regra *Decl-Simple-Local*, prova o comportamento da linguagem ao declarar uma variável local. Ao declarar uma variável, se esta pertencer ao domínio local (*D*') então o seu nome e valor vão ser adicionados ao domínio local.

A regra seguinte (*Assign-Simple-Local*), tem como objetivo provar o comportamento da linguagem numa atribuição de valor a uma variável local. A mudança de estados é semelhante à regra anterior, ou seja, o nome e o valor da mesma vão ser adicionados ao domínio local do membro.

A regra *Assign-Simple-Global*, pretende comprovar o comportamento da linguagem, quando é atribuído um valor a uma variável global. A diferença desta regra relativamente à regra anterior, consiste em a variável ter de pertencer ao contexto global e não ao contexto local do programa.

Por fim, a regra *Assign-Array-Global*, consiste em comprovar o comportamento de atribuir um valor a uma variável do tipo *array*. Ao atribuir um valor a um *array*, se este pertencer ao domínio global do programa e v_2 que indica a posição de atribuição do valor, estiver correta, então vai-se adicionar o novo valor v_1 ao *array* global.

Com estas provas consegue-se validar o bom comportamento da linguagem diminuindo assim o número de erros que a linguagem possa ter.

3.3 Sistema de tipos

O sistema de tipos é uma parte integrante da análise semântica do compilador. O sistema de tipos pretende diminuir os erros que possam existir durante a execução do programa. Por exemplo, ao não definir uma regra de sistema de tipos que indique que os tipos inteiro e booleano não são equivalentes, o programador por engano poderia fazer essa equivalência. Este erro causaria uma falha na execução do programa na Máquina Virtual, o que traria diversos problemas na rede de sensores. Como um dos objetivos é diminuir o máximo de erros possíveis que possam existir em tempo de execução, o sistema de tipos é um passo importante no desenvolvimento da linguagem. As regras do sistema de tipos não são utilizadas apenas para validar se os tipos estão corretos, mas também para validar outros aspetos, como por exemplo, se a estrutura de um programa está correta. As regras dos sistema de tipos estão descritas na secção C.

Cada uma das regras do sistema de tipos é dividida em duas partes. A parte superior da regra é vista como uma pré-condição da regra que tem de ser satisfeita de modo a que a parte inferior da regra possa ser executada.

A regra *T-Programa* verifica se um programa *Macaw* está bem tipificado. Um programa *Macaw* é considerado bem tipificado se os seus recursos de *hardware*, as variáveis globais e os seus membros estiverem bem tipificados. Além dessas verificações, a regra também tem como objetivo verificar se o *main* do programa está bem tipificado e se este existe. O *main* é considerado bem tipificado se o seu tipo de retorno for do tipo "*handler*" e a sua lista de parâmetros estiver vazia. As regras *T-Globais* e *T-Funções* são utlizadas para verificar se cada variável global do programa e se cada membro respetivamente, estão bem tipificados. Estas regras também permitem validar se não existem variáveis globais e membros com o mesmo nome. A "," da parte inferior da regra significa que existe uma união disjunta, fazendo com que não possam existir elementos com nomes repetidos.

A regra *T-Recursos* tem como objetivo validar a secção "*hardware states*", "*boot*" e "*actions*". Estas três secções são consideradas bem tipificadas se cada uma das secções estiver bem tipificada. De forma a validar a secção "*hardware states*" são utilizadas duas regras (*T-HardRess* e *T-HardRes*). A regra *T-HardRess*, é utilizada para percorrer cada um dos módulos de *hardware* e validar se cada um deles está bem tipificado. Esta regra é utilizada também para verificar se não existem módulos repetidos. Um módulo de *hardware* é considerado bem tipificado, se o valor energético indicado em "*normal*" for menor que o valor energético indicado em "*max*". Um módulo de *hardware* ao estar bem tipificado vai ser adicionado ao Γ, como sendo do tipo *mod*.

A regra que vai verificar se a secção "*boot*" está bem tipificada é a *T-BootSemOr*. Como se pode verificar não existe uma regra para o "*or*", isso deve-se ao facto de não ser

possível utilizar este comando secção "*boot*". O "*and*" é considerado bem tipificado se a expressão esquerda e direita estiverem bem tipificadas. Um módulo de *hardware* que tenha "*on*" ou "*off*" é bem tipificado se o nome indicado for do tipo *mod*.

A secção "*actions*" é considerada bem tipificada se os seus eventos e funções de *hardware* estiverem bem tipificadas. A regra para verificar se os eventos estão bem tipificados (*T-TransiçãoEventos*), antes de considerar o evento bem tipificado e assim adicionar o mesmo ao Γ com os seus parâmetros, necessita de verificar certas condições. A regra tem de verificar se cada um dos parâmetros do evento têm um tipo válido e se as cláusulas "*when*" e "*turns*" estão bem tipificadas. De realçar que na cláusula "*turns*" não é válido utilizar o comando "*or*", mas na cláusula "*when*" já é válido. A regra que verifica as funções de *hardware* (*T-TransiçãoFunções*) tem um objetivo semelhante à regra que valida os eventos. A única diferença consiste em validar que o tipo de retorno de estas funções é válido. Estas regras vão ser utilizadas para validar o ficheiro *.macawh*.

De modo a validar as variáveis globais foram definidas quatro regras. A regra *T-GVar* é utilizada para validar uma variável global que não foi inicializada. A regra vai adicionar ao Γ o nome da variável com o seu tipo correspondente. Se a variável global for inicializada, a regra é um pouco diferente, só vai ser adicionado ao Γ o tipo e o nome da variável se o tipo da expressão for equivalente ao tipo da variável. A regra *T-GArrVar* é utilizada para um *array* que não seja inicializado. Neste caso vai ser adicionado ao Γ o nome do *array*, o seu tipo e o tamanho do *array*. No caso do *array* ser inicializado, antes de ser adicionado ao Γ a sua informação, é necessário verificar se cada elemento do *array* tem o tipo correto e se o número de elementos do *array* está de acordo com o tamanho indicado. Em relação à inicialização de *array* em que se utiliza a "*string* " "..." é utilizada a regra *T-GArrVarInitPontos*. A unica diferença relativamente à regra anterior é o facto de o número de elementos descritos ter de ser menor que o tamanho do *array*. De notar que em todas as regras é utilizada a união disjunta de forma a garantir que não existe nenhuma variável com o mesmo nome.

A regra *T-Membro* é utilizada para verificar se um membro do programa está bem tipificado. Um membro é considerado bem tipificado se tiver um tipo de retorno válido, se todos os seus parâmetros tiverem um tipo válido e se o seu corpo for válido. Por fim, o seu corpo também tem de estar bem tipificado. O corpo de um membro é considerado bem tipificado se as variáveis definidas, as suas instruções e o comando de "*return*" estiverem bem tipificadas.

Em todas as regras do corpo do membro é utilizado um Γ_x em vez do Γ. Este Γ_x é referente ao membro que está a ser verificado. As variáveis locais vão ser armazenadas neste Γ_x. Assim é possível identificar se uma variável local pertence ou não ao membro que está a ser verificado.

Na linguagem *Macaw* é possível fazer chamadas a outros membros numa expressão ou no corpo do membro. Para se poder fazer uma chamada no corpo do membro esse membro

tem de ser do tipo "*void*". Enquanto que se esse membro for chamado numa expressão o tipo de retorno tem de estar de acordo com o tipo de expressão onde foi chamada. Uma chamada a um membro tem de garantir que a expressão de cada argumento está de acordo com o tipo desse argumento. De modo a verificar se uma atribuição de valor é válida é necessário verificar se o tipo da expressão é equivalente ao tipo da variável.

Um comando "*attach*" é considerado bem tipificada se o evento e o "*handler*" tiverem o mesmo número de argumento e se estes argumentos forem do mesmo tipo. Os comandos "*unfold*", "*if*" e "*ifStateContains*" são consideradas bem tipificadas se a sua expressão for do tipo correto ("*int8*" para o comando "*unfold*" e "*boolean*" para os comandos "*if*" e "*ifstateContains*") e se o seu corpo estiver bem tipificado.

O comando "*return*" está bem tipificado se o tipo da sua expressão estiver de acordo com o tipo de retorno do membro que está a ser validado.

Como se vai poder verificar nas secções sobre a implementação da linguagem vai ser utilizado uma tabela de símbolos no compilador. O Γ é referente à tabela de símbolos global do programa, enquanto que o Γ_x é referente à tabela de símbolos local de cada membro.

3.4 Sintaxe da linguagem intermédia

Uma linguagem intermédia é uma versão mais concisa da *AST* gerada inicialmente pelo compilador. Esta representação intermédia permite remover dados que não são necessários para a geração de código, facilitando o processo de geração. Este processo é facilitado, pois o número de nós existentes na representação intermédia é claramente inferior ao número de nós da *AST* gerada inicialmente. Esta técnica é muito utilizada por compiladores. Existem dois exemplos claros sobre esta utilização, o compilador de *C++* e *Latex*. Os primeiros compiladores para a linguagem *C++* não convertiam diretamente o código fonte para o código máquina. Antes, o código fonte em *C++* era traduzido para a linguagem *C*. O compilador começava por utilizar o pré-processador *cpp* e posteriormente utilizando o *cfront* para traduzir o programa para a linguagem *C*. Neste caso, a linguagem que é gerada por o *cfront* pode ser visto como uma representação intermédia da linguagem *C++* [14].

Um outro caso é a linguagem *Latex*, esta linguagem não gera diretamente o *pdf* a partir do código fonte inserido por parte do utilizador. O texto fonte é inicialmente traduzido para uma linguagem chamada *Tex*, que depois vai ser traduzida para uma representação intermédia mais simples chamada *dvi*. Posteriormente a este passo, a representação intermédia vai passar por mais dois passos até chegar finalmente ao *pdf* final [14].

Para o caso desta linguagem, a geração de código não necessita de aceder aos nós relativamente às funções de *hardware* da *AST* original, entre outros. Assim, com esta representação intermédia, estes dados não iram lá constar. Evitando visitas desnecessárias

a esses nós. Uma outra razão para gerar esta representação intermédia, é o fato de se querer calcular em tempo de compilação o tamanho da pilha de chamadas e operandos da Máquina Virtual. Para efetuar este cálculo é necessário saber qual o *bytecode* gerado pelo compilador. Pretende-se também que esta representação fosse o mais semelhante possível ao *bytecode* gerado por parte do compilador.

A sintaxe desta representação intermédia é bastante simples comparativamente com a sintaxe inicial da linguagem, como pode ser comprovado na Figura D.1. Esta representação é constituída por um cabeçalho, por uma definição de módulos, por um conjunto de variáveis globais se o programa as tiver, as funções do programa e uma secção de Mapa.

O cabeçalho é definido por seis valores inteiros, cada um com um significado diferente. Este cabeçalho contém informação sobre o tamanho das variáveis globais do programa, o tamanho que as funções do programa vão ocupar, o número de eventos que foram definidos no ficheiro *.macawh*, o tamanho da pilha de chamada e operandos, o tamanho da fila de eventos da Máquina Virtual e por fim, o tamanho total do *bytecode* gerado por parte do compilador.

Os módulos podem ser representados apenas por um valor inteiro. Este valor indica se o módulo de *hardware* quando a aplicação é iniciada está ligado ou não.

As variáveis globais têm a representação do seu valor. No caso de variáveis globais que sejam do tipo *array*, estas precisam de ter mais um valor. Este valor é correspondente ao tamanho desse *array*. Este valor permite que a Máquina Virtual possa validar o acesso ao *array*. Este valor é armazenado na posição a seguir ao último valor do *array*. Os valores destas variáveis globais podem ser inteiros, do tipo *float* ou *booleanos*.

As funções são representadas por um nome, ou seja, o nome da função e contém uma lista de comandos. Estes comandos correspondem ao *bytecode* gerado para essa mesma função. Este *bytecode* contém comandos que permitem fazer somas, subtrações, saltos, aceder a variáveis globais, entre outros. O funcionamento destes comandos vai ser explicado posteriormente.

Por fim, a secção Mapa permite identificar a posição do "*handler*" *main* no *bytecode* gerado. Esta secção permite que este "*handler*" possa ser definido em qualquer posição do programa. Sem esta secção o "*handler*" *main* tinha de ser definido sempre na mesma posição do programa.

3.5 Formato do *bytecode* gerado

Como foi referido anteriormente, um dos grandes problemas apresentados por estes dispositivos é o facto destes terem uma memória bastante reduzida. Por esta razão, para o desenvolvimento do *bytecode* foi necessário ter em conta este fator, sendo assim tentou-se abstrair a maior informação possível do código fonte.

O formato do *bytecode* é bastante semelhante à representação intermédia apresentada na secção anterior. Relativamente à secção de cabeçalho, cada um destes valores vai ser representado por dois *bytes* cada um. A escolha por dois *bytes* foi necessária, pois representar estes valores apenas com um *byte* tornava o tamanho do *bytecode* bastante limitativo.

Uma das principais diferenças é relativamente à declaração dos módulos. Enquanto que a representação intermédia indica para cada módulo se este está ligado ou não quando a aplicação arranca, no *bytecode* é usado um *byte* para representar um conjunto de oito módulos. Ou seja, um *byte* corresponde a oito *bits*, cada um desses *bits* vai ser utilizado para indicar se o módulo está ligado ou não. Se o módulo estiver ligado este vai ter o seu *bit* com o valor 0, caso contrário, vai ter o seu *bit* representado com 1.

No exemplo apresentado na figura G.4 estão definidos quatro módulos de *hardware*, mas apenas um deles está ligado quando a aplicação arranca. Os restantes módulos por omissão estão todos desligados. Com este cenário, em que existem menos de oito módulos definidos, é apenas necessário um *byte* para representar um módulo. Em que os primeiros 4 *bits* de este *byte* vai ter dados sobre os módulos de *hardware*. Isto é exemplificado com a Figura 3.1.

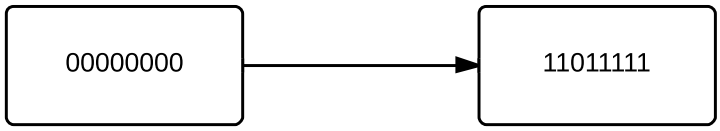


Figura 3.1: Transformação para *bytecode*

Como se pode verificar, o módulo *mcu328Sleep* corresponde ao primeiro *bit* do *byte*, o módulo *power_light* corresponde ao segundo *bit*, o módulo *mcu328p* ao terceiro *bit* e por fim, o módulo *button_light* corresponde ao quarto *bit*.

Se fossem definidos mais que oito módulos de *hardware* já iria ser necessário usar mais que um *byte* para a representação dos módulos. Neste caso os primeiros oito módulos

de *hardware* iriam ser representados no primeiro *byte*, os oito módulos seguintes iriam ser representados no segundo *byte* e os seguintes iriam ser representados da mesma forma.

A definição de variáveis globais no *bytecode* é feita da mesma forma independentemente do tipo primitivo da variável. A principal diferença é se esta é do tipo *array* ou não. Se a variável for do tipo "*int8*" ou "*boolean*", o seu valor vai ocupar um *byte*, se a variável for do tipo "*int16*" esta vai ocupar dois *bytes* e se a variável for do tipo "*int32*" ou "*float*" esta vai ocupar 4 *bytes*. No caso de a variável ser do tipo *array*, os seus valores vão ser convertidos de acordo com as regras apresentadas anteriormente e vai ser utilizado mais um *byte* para converter o valor relativo ao tamanho do *array*.

Outra diferença apresentada comparativamente com a representação intermédia é relativamente à definição de membros, sendo necessário apresentar no *bytecode* informação sobre o tamanho das variáveis locais e o tamanho dos parâmetros do membro. Estes valores são representados utilizado dois *bytes*. Fizemos uma análise e chegamos à conclusão de que é necessário ter mais espaço para variáveis locais do que para parâmetros. Assim sendo, desses dois *bytes* são utilizados dez *bits* para representar o tamanho das variáveis locais, o que permite que as variáveis locais do método ocupem 1024 *bytes* e os restantes seis *bits* são utilizados para representar o tamanho dos parâmetros, o que permite que os parâmetros ocupem 64 *bytes* de memória. Estas duas informações são representadas utilizando um *byte* para cada uma delas, sendo estes *bytes* colocados antes das instruções de *bytecode*.

Na tabela 3.2 está apresentado o *bytecode* que não tem qualquer parâmetro, estas instruções são maioritariamente instruções de cálculo de expressões.

A memória ocupada por estas instruções é igual para todos, ou seja, todos estes *bytecode* apenas necessitam de guardar a informação relativa ao tipo de *bytecode*. Assim sendo, é utilizado apenas um *byte* para representar cada um destes *bytecode*.

Alguns dos *bytecode* apresentados na tabela 3.2 podem ser divididos em três *bytecode*. Por exemplo, o *bytecode add* tem três variações diferentes, uma para calcular o resultado da soma entre dois valores do tipo *int8* (*add8*), outra variação para calcular a soma entre dois valores do tipo *int16* (*add16*) e por fim uma variação que permite calcular a soma entre dois valores do tipo *int32* (*add32*). Esta situação aplica-se a outras instruções da tabela, tais como: *sub*, *mul*, *div*, *rem*, *slt*, *ret* e *dup*. Mesmo estas variações tendo três tipos diferentes, estas vão ocupar um *byte* de memória do *bytecode*.

Na tabela 3.3 estão apresentados o *bytecode* que recebem parâmetros. Nesta tabela além da explicação do significado de cada *bytecode*, também é apresentado uma coluna referente ao tamanho que esse parâmetro vai ocupar. De realçar, que de novo todas as instruções de *bytecode* vão ocupar um *byte*. Dos comandos apresentados à exceção dos comandos *jmp*, *brd*, *castf*, *cast32* e *lnk* os outros apresentam três variações. Para os tipos *int8*, *int16* e *int32*.

A instrução *lnk* é diferente das outras sendo a única que recebe dois parâmetros. Neste

Tipo	Significado
add	Soma valores inteiros
sub	Subtrai valores inteiros
mul	Multiplica valores inteiros
div	Divide valores inteiros
rem	Resto da divisão entre valores inteiros
slt	Compara dois valores inteiros
dup	Produz uma duplicação
ret	Retorna um valor
call	Produz uma chamada de função
retv	Comando de retorno Membros "*void*" ou "*handler*"
fadd	Soma valores do tipo "*float*"
fsub	Subtrai valores do tipo "*float*"
fmul	Multiplica valores do tipo "*float*"
fdiv	Divide valores do tipo "*float*"
fslt	Compara dois valores do tipo "*float*"
and	Produz operação "*and*" numa expressão "*boolean*"
or	Produz operação "*or*" numa expressão "*boolean*"
not	Nega uma expressão "*boolean*"
hdl	Efetua a chamada de "*handler*" quando o evento é acionado
cast16	Produz o *cast* de uma variavel para o tipo "*int16*"
ldbit	Carrega o *bit* referente à posição do módulo de *hardware*

Tabela 3.2: Instruções *bytecode* sem parâmetros

caso cada um dos parâmetros desta instrução ocupam um *byte*.

Por fim, a secção *Map* no *bytecode* identifica a localização do *main* no *bytecode* gerado. Esta secção pode ser traduzida utilizando dois *bytes* de modo a representar essa localização.

O facto da linguagem *Macaw* ser orientada a eventos significa que esta tem de ter um modo "adormecido" em que fica à espera que um evento ocorra. Este modo pode ser visto como um ciclo infinito que está constantemente a verificar se ocorreu algum evento. Se tiver ocorrido algum evento, então vão ser executadas as ações relativas a esse evento. De modo a que a linguagem *Macaw* tenha este ciclo, no "*handler*" *main* é sempre gerado *bytecode* de modo a que este fique em ciclo infinito no fim da sua execução. O ciclo é constituído pelas instruções *HDL*, *CALL* e *JMP*. São executadas por esta ordem, em que a instrução *JMP* faz com que esse *bytecode* seja executado infinitamente. Estas instruções permitem que um evento quando for acionado, a instrução *HDL* coloque os dados na pilha e de seguida o comando *CALL* faça a chamada desse "*handler*".

Tipo	Memória ocupada	Significado
ldc	1,2 ou 4 *bytes*	Carrega uma constante para a pilha de chamadas e operandos
ld	2 *bytes*	Carrega uma variável para a pilha de chamadas e operandos
st	2 *bytes*	Guarda o valor que está no topo da pilha no local indicado
beq	1 *byte*	Compara dois valores e produz um salto
bne	1 *byte*	Negação do resultado da instrução *beq*
cast32	1 *byte*	Produz um *cast* para "*int32*"
castf	1 *byte*	Produz um *cast* para "*float*"
ldg	2 *bytes*	Carrega para a pilha o valor de uma variável global
stg	2 *bytes*	Guarda o valor da variável global no espaço de memória indicado
lda	2 *bytes*	Carrega na pilha o valor armazenado na posição do *array*
sta	2 *bytes*	Guarda o valor da posição do *array* no espaço de memória indicado
brd	1 *byte*	Permite fazer chamadas de funcções de *hardware*
jmp	2 *bytes*	Produz um salto para uma posição do *bytecode*
lnk	2 *bytes*	Faz a ligação entre um evento e o seu "*handler*"

Tabela 3.3: Instruções *bytecode* com parâmetros

Capítulo 4

Implementação compilador *Macaw*

No desenvolvimento do compilador começou por se definir o fluxo de informação do mesmo. Este compilador, tal como todos os compiladores modernos atualmente são orientados para a sintaxe [14]. Estes compiladores necessitam que a estrutura sintática do código fonte seja validada pelo analisador. Se o código fonte estiver correto, este pode ser convertido para a *AST* (*Abstract Syntax Tree*) [14]. A vantagem desta conversão é abstrair do código fonte informação desnecessária, por exemplo, no caso da secção "*hardware states*", não é necessário guardar a "*string*" "*min*" e "*max*". Esta análise é desenvolvida pela plataforma *Xtext*. Depois deste processo e a *AST* estar contruída esta vai ser analisada com o sistema de tipos. Como foi explicado na secção 3.3, a função do sistema de tipos é validar que a semântica estática de cada nó da *AST* está correta, ou seja, se cada nó da *AST* está correto (verificar se o tipo de uma variável é correto, se não existem variáveis repetidas, etc.) [14]. Se existir algum erro na semântica da linguagem, então vai ser lançado um erro a explicar o problema encontrado. O desenvolvimento deste sistema de tipos está de acordo com as regras apresentadas no Apêndice C.

O processo explicado anteriormente é igual quer para a compilação do ficheiro *.macaw*, quer para o ficheiro *.macawh*. A partir deste passo o fluxo de informação do compilador é diferente para cada um dos ficheiros.

O ficheiro *.macaw*, depois de estar validado por o sistema de tipos, o código fonte vai ser transformado para as optimizações poderem ser aplicadas. A *AST* do código fonte vai ser transformada em um grafo de fluxo de controlo e posteriormente transformado num grafo "*static single assignment form*". As optimizações vão ser aplicadas diretamente sobre este último grafo desenvolvido. Este processo vai ser explicado detalhadamente na secção 5. Posteriormente a este passo, o grafo "*static single assignment form*" vai ser traduzido numa linguagem intermédia. Esta representação intermédia, foi desenvolvida para facilitar o processo de geração de código e os cálculos necessários para o cabeçalho do mesmo. Esta passo permite abstrair certos aspetos da *AST* que não são necessários para a geração de código. A construção da linguagem intermédia é desenvolvida pelo gerador de código. Por fim, o mesmo gerador de código desenvolve dois tipos de ficheiros

diferentes. O ficheiro *.asm* permite visualizar o *bytecode* gerado de forma textual e assim identificar algum problema que possa existir. O ficheiro *.bc* é exatamente o mesmo que este anterior, a grande diferença é que este ficheiro é destinado para a Máquina Virtual da linguagem. Estes são os passos necessários para compilar um ficheiro *.macaw*, sendo este processo esquematizado na figura 4.1.

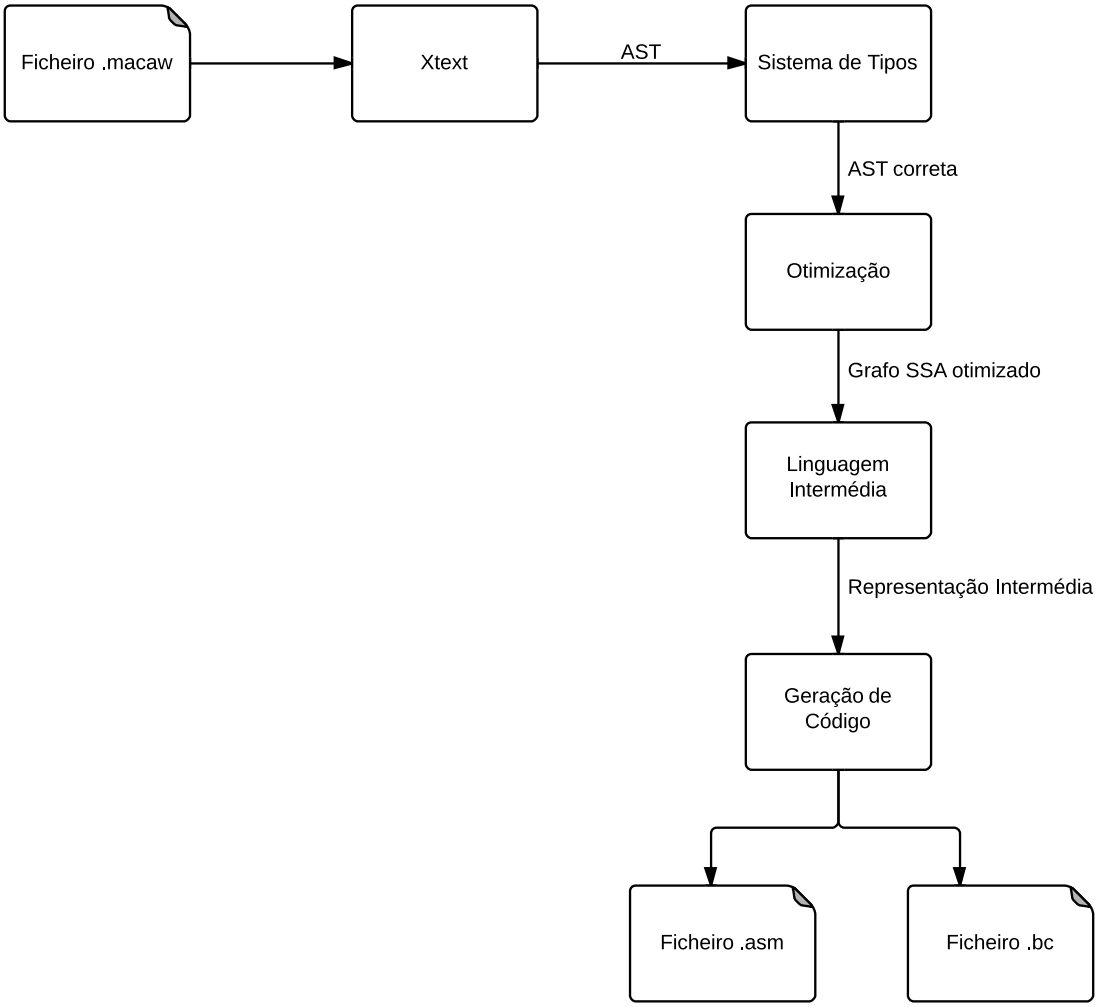


Figura 4.1: Fluxo de informação do compilador *Macaw* - *Parte I*

O ficheiro *.macawh*, tem um processo de compilação bastante mais simples que o ficheiro apresentado anteriormente. Depois do sistema de tipos ser executado sobre o ficheiro, o gerador de código é logo executado. O gerador de código vai gerar um *pdf* com um grafo que permite visualizar os estados em que o programa vai estar quando for executado. Este grafo é útil para verificar se o comportamento da linguagem e do *hardware* está de acordo com o esperado inicialmente. Para o desenvolvimento deste grafo utilizou-se a aplicação *Graphviz* [7] e a sua facilidade de desenvolvimento de grafos deste tipo. O algoritmo utilizado para desenvolver este ficheiro vai ser explicado na secção 4.6. O processo de compilação explicado anteriormente está representado na Figura 4.2.

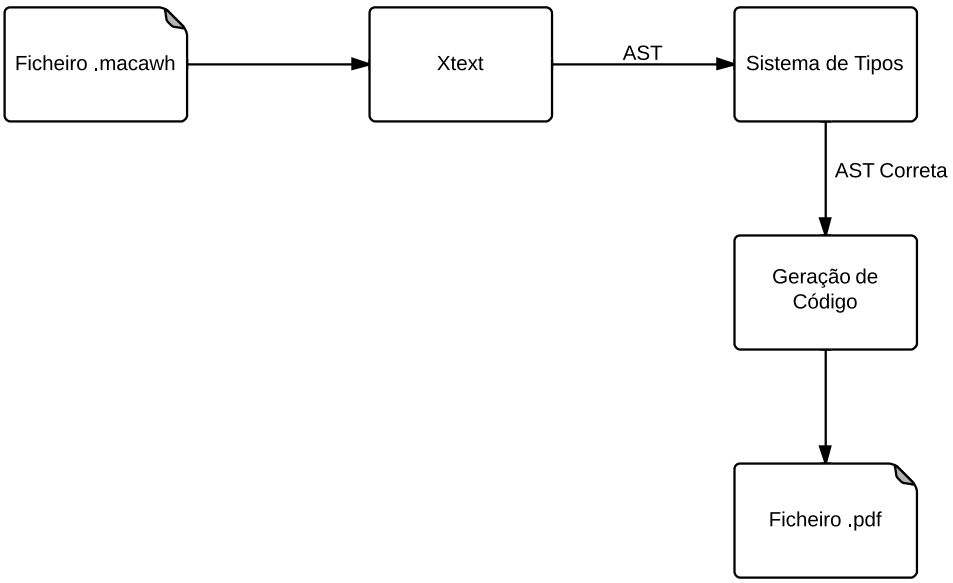


Figura 4.2: Fluxo de informação do compilador *Macaw - Parte II*

O compilador desenvolvido apresenta algumas características que estão omissas para o utilizador. Uma dessas característica que o compilador apresenta, é o facto de todos as variáveis serem inicializadas. Mesmo as variáveis que não são inicializadas por parte do programador, vão ser inicializados por um valor de definição. Esse valor para cada tipo de variável é indicado na tabela 5.1.

Tipo	Valor por Definição
int8	0
int16	0
int32	0
float	0.0
boolean	*false*

Tabela 4.1: Valores por definição

A linguagem *Macaw* apresenta uma hierarquia de tipos primitivos. Ou seja, permitem que tipos inferiores na hierarquia possam ser convertidos para um tipo superior. As regras de conversão de tipos são apresentadas na figura seguinte.

A Figura 4.3 apresenta a função "*max*" que permite determinar qual dos tipos dados está num nível superior da hierarquia de tipos.

Como se pode verificar pelas regras, o tipo "*int8*" é o tipo primitivo mais simples, e assim sendo pode ser convertido para qualquer um dos outros tipos primitivos ("*int8*", "*int16*", "*int32*" e "*float*"). O tipo primitivo "*int16*", pode ser convertido para "*int32*" e "*float*". Por fim, o tipo "*int32*", pode ser convertido apenas para "*float*". O "*float*", é o tipo

$$\max(\textbf{int8}, \textbf{int16}) = \textbf{int16} \qquad \text{(T-MaxInt1)}$$

$$\max(\textbf{int8}, \textbf{int32}) = \textbf{int32} \qquad \text{(T-MaxInt2)}$$

$$\max(\textbf{int16}, \textbf{int32}) = \textbf{int32} \qquad \text{(T-MaxInt3)}$$

$$\frac{t : \textbf{int8}, \textbf{int16}, \textbf{int32}, \textbf{float}}{\max(t, t) = t} \qquad \text{(T-MaxIgualdade)}$$

$$\frac{t_1, t_2 : \textbf{int8}, \textbf{int16}, \textbf{int32}, \textbf{float}}{\max(t_1, t_2) = \max(t_2, t_1)} \qquad \text{(T-MaxTransitividade)}$$

$$\frac{t : \textbf{int8}, \textbf{int16}, \textbf{int32}, \textbf{float}}{\max(t, \textbf{float}) = \textbf{float}} \qquad \text{(T-MaxFloat)}$$

Figura 4.3: Hierarquia de tipos da linguagem *Macaw*

primitivo do topo da hierarquia, assim sendo este não pode ser convertido para qualquer outro tipo. Uma outra regra apresentada na figura, é relativamente à transitividade dos tipos. O resultado da função "*max*" é igual mesmo que os tipos que recebem estejam com ordem diferente. Esta funcionalidade permite que existam *cast* implícito na linguagem para um tipo superior. *Cast* implícito significa que não é necessário que o programador indique para que tipo é necessário fazer o *cast*. Esta conversão é produzida internamente pelo compilador. Por exemplo, se uma variável for do tipo "*int32*" e tiver o valor zero, este valor apesar de ser um "*int8*" vai ser automaticamente convertido para um "*int32*".

Todas as acções definidas no ficheiro *.macawh*, têm um identidicador único para fazer a ligação entre as funções e eventos definidos no compilador e os que são definidos na Máquina Virtual. Para que o programador não tivesse de indicar estes valores à mão individualmente, estes são adicionados pelo compilador. O que diminui a probabilidade de erro que poderia trazer ao introduzir estes valores manualmente.

A linguagem *Macaw* permite fazer a ligação a três eventos predefinidos, estes eventos são: "*onBoot*", "*Idle*" e "*onError*". O evento "*onStart*", é acionado quando o sensor inicia a execução da aplicação, permitindo ao programador definir um comportamento inicial do sensor. Este evento não recebe nenhum parâmetro. O evento "*Idle*", é acionado quando o sensor está "adormecido". Mais uma vez este evento não recebe nenhum parâmetro. Por fim, o evento "*onError*", é lançado quando existe um erro na aplicação. Por exemplo, uma divisão por zero, ou um acesso incorreto a um *array*. Permite assim, ao programador definir o que irá acontecer quando existe um erro. Este evento recebe como parâmetro um "*int16*" relativo ao erro que aconteceu durante a execução do programa.

4.1 *Xtext*

O *Xtext* [12] é uma plataforma que permite o desenvolvimento de linguagens "*domain-specific*", esta plataforma é utlizada com o *Eclipse IDE* [11]. Linguagens "*domain-specific*" permitem resolver um problema específico, ao contrario de linguagens de programação como *Java* ou *C*, estas linguagens são conhecidas como linguagens de propósito geral [5]. Esta plataforma utiliza a linguagem de programação *Java* e *Xtend*. Esta última linguagem, tem algumas características que não estão presentes na linguagem *Java*. Entre elas, está o uso de expressões *lambda*, também permite inferência de tipos, ou seja, não é necessário indicar explicitamente o tipo de uma variável ou um tipo de retorno de uma função e permite o uso expressões multi-linha [5]. Além destas características, o *Xtend* permite utilizar todas as bibliotecas que estão disponíveis na linguagem de programação *Java*.

Esta plataforma tem como objetivo facilitar o desenvolvimento de compiladores. O *Xtext* desenvolve os primeiros passos para o desenvolvimento de um compilador. Tendo já implementado o *scanner*, o *parser* e apresenta um sistema de tipos por definição. O primeiro passo que é necessário fazer quando se compila um programa, é verificar se esse programa respeita a sintaxe da linguagem. Para que isto seja possível, o programa tem de ser dividido em *tokens*. Cada *token* é um elemento atómico da linguagem. Este processo é chamado de análise léxica e o programa que faz esta análise é chamado de analisador léxico, léxico ou *scanner* [5]. Mas, apenas ter uma sequência de *tokens* não é suficiente para garantir que o programa é sintaticamente válido. É necessário validar que o programa apresenta uma estrutura sintática válida. Esta validação é chamada análise sintática e é produzida pelo *parser* da linguagem [5]. Se esta validação correr bem, o *parser* vai construir a *AST* do programa. Esta é construída para facilitar a análise semântica do programa, sendo esta guardada em memória para não ser necessário de correr o *parser* sempre que se queria analisar o programa [5]. Por fim, o *Xtext* tem um sistema de tipos por definição que verifica a completude de certos tipos. Esta fase de compilação é conhecida como análise semântica. Ao desenvolver esta fase de compilação, o *Xtext* facilita em muito o processo de compilação, o desenvolvimento de estes algoritmos para linguagens bastante complexas é um processo bastante complicado. Esta explicação está representada gráficamente na figura 4.4.

Outra vantagem do *Xtext*, é apresentar integração com o *Ecplise IDE*. Todas as *DSL* hoje em dia apresentam uma integração deste género.

Para começar a desenvolver basta definir o nome que a linguagem vai ter e o *Xtext* cria quatro pacotes principais. No nosso caso, o nome da linguagem é *macaw*, por isso vai criar os seguintes pacotes: *macawc*, *macawc.sdk*, *macawc.ui* e *macawc.tests*.

O pacote *macawc*, é utilizado para implementar as funcionalidades do compilador. Ou seja, é utilizado para definir a sintaxe da linguagem, o sistema de tipos no qual se pode indicar erros e *warnings* que vão ser lançados no *plugin* e o gerador de código. No caso da linguagem *macaw*, este pacote, foi utilizado também para desenvolver as otimizações do código e o gerador do grafo de estados do programa.

O pacote *macawc.ui*, é utilizado para alterar a interface do *plugin* gerado pelo *Xtext*. É permitido alterar o *outline* apresentado por parte do *Eclipse IDE*, apresentar mensagens para corrigir erros lançados por parte do compilador, permite que se possa fazer "*auto-complete*" quando se escreve o nome de uma variável definida anteriormente, entre outros. Neste pacote também é permitido alterar a cor que os erros e os *warnings* vão ter quando forem lançados.

O *Xtext* permite que a linguagem possa ser testada, e para tal é criado o pacote *macaw.tests*. É permitido testar todas as funcionalidades do compilador, desde os erros que são lançados por parte do mesmo, ao código que é gerado por parte do mesmo. Além disto, também é permitido testar a interface gerada pelo *Xtext*. Para produzir estes testes, é utilizado o *JUnit* [19] aproveitando as suas capacidades.

A geração do *plugin* da linguagem para poder ser posteriormente instalado num *Eclipse IDE* é produzido utilizando o pacote *macawc.sdk*. Para gerar este *plugin*, é necessário ter instalado previamente a aplicação *Buckminster* [10]. Este *plugin* tem de ser instalado num *Eclipse IDE* que tenha a plataforma *Xtext* instalada.

A plataforma *Xtext* é tão versátil, que permite que o código gerado seja compilado utilizando a *Virtual Machine* do *Java*. Para utilizar essa funcionalidade basta que o código gerado seja um programa *Java* válido [5].

Apesar dos pontos positivos apresentados anteriormente, esta plataforma não está isenta de problemas. Alguns problemas são relativos à execução do *plugin* gerado pelo *Xtext* enquanto outros são relativos à execução dos testes sobre a linguagem desenvolvida.

Por vezes, quando se corre o *plugin* e compila um ficheiro *macaw* com o compilador desenvolvido este dá exceções, mas se o mesmo código for executado de novo a exceção anterior não volta a acontecer. Este é o erro mais comum que o *Xtext* costuma apresentar.

Relativamente aos testes do compilador, o grande problema consiste em que o código macaw em alguns casos não é executado da mesma forma como se fosse executado pelo *plugin* gerado pelo *Xtext*. O *parsing* de um ficheiro *macaw* que contenha um "*ifStateContains*" com módulos de *hardware* é diferente do *parsing* por parte do *plugin*. Fazendo com que este comando não consiga ser testado na sua plenitude.

Apesar de estes erros apresentados anteriormente a ferramenta *Xtext* é muito poderosa, contendo muitas funcionalidades para o desenvolvimento de linguagens "*domain-specific*".

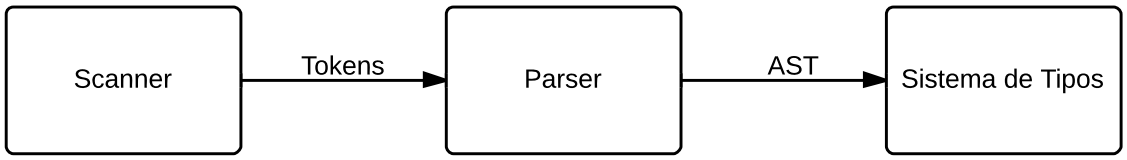


Figura 4.4: Fluxo de informação *Xtext*

4.2 Tabela de símbolos

A tabela de símbolos, é um mecanismo que permite que informação sobre o código fonte, seja associado a símbolos para que estes possam ser usados em certas fases do compilador. Sempre que se quiser obter um dado sobre um símbolo basta aceder à tabela para obter a sua informação [14]. Esta tabela de símbolos é usada maioritariamente na verificação do sistema de tipos, mas pode ser utilizada em qualquer fase do compilador. No caso da linguagem *Macaw*, a tabela de símbolos é construída à medida que os tipos da linguagem vão sendo verificados, sendo esta posteriormente utilizada em todas as fases do compilador, pois é mais fácil aceder à mesma do que fazer uma visita à *AST* de modo a encontrar a informação que se pretende.

O primeiro passo para o desenvolvimento desta tabela, consistiu em identificar qual a informação que era necessária para o seu desenvolvimento. Certas informações, como os consumos energéticos dos módulos, ou mesmo as expressões de certas variáveis não necessitam de constar na tabela.

A tabela da linguagem *Macaw*, segue o que foi dito anteriormente, ou seja, faz a ligação entre os seus símbolos e as entradas da tabela que contêm informação sobre o código fonte. Os símbolos da tabela correspondem ao nome da variável, módulo, membro ou funções de *hardware*. Estes símbolos fazem ligação a diversas entradas. Estas entradas podem ser para os módulos, variáveis, membros e constantes. Todas estas entradas, além de armazenarem dados importantes relativamente aos símbolos, também armazenam a posição de memória que os símbolos vão ocupar no *bytecode*, que irá ser posteriormente gerado.

A entrada para os módulos armazena se o módulo irá estar ligado ou não no arranque da aplicação. Além deste valor, também irá ser armazenado o *byte* e o *bit* que este módulo irá ocupar no *bytecode* gerado.

Para as variáveis que não sejam do tipo *array*, vai ser armazenado o seu tipo. Este tipo permite que posteriormente no sistema de tipos seja validado certas regras. Por exemplo, se uma dada expressão for equivalente ao tipo da variável então não existe nenhum erro, caso contrário pode-se lançar um erro a avisar que os tipos não são equivalentes.

A entrada para variáveis do tipo *array*, têm um objetivo semelhante às variáveis apresentadas anteriormente. Mas esta entrada armazena também o tamanho do *array*.

A entrada para os membros pode ser dividida em três entradas diferentes. Uma para

os "*handler*", outra para as funções de *hardware* e outra para as funções definidas pelo programador. Independentemente do tipo de entrada de membros, todas estas têm uma tabela de símbolos local. Esta tabela de símbolos é utilizada para armazenar as variáveis locais do método. Permitindo assim, fazer uma divisão entre as variáveis locais a um método e as variáveis globais do programa. Outra informação que guardam é se o membro é puro ou não. Um membro é considerado puro, se não atribuir valores a qualquer variável global e se não fizer qualquer chamada a uma outra função que seja impura. Sendo assim, um membro para ser considerado impuro basta ter uma das características apresentadas anteriormente. Todas as funções de *hardware* da linguagem são consideradas impuras. As últimas informações armazenadas nestas entradas, correspondem ao tamanho ocupado pelas variáveis locais para esse membro e o tamanho ocupado pelos seus parâmetros. Como as funções de *hardware* não são necessárias para o *bytecode*, o tamanho das variáveis locais e dos parâmetros não são necessárias para estas funções.

Para as entradas do tipo "*handler*" e funções não existem grandes diferenças. Ambas armazenam uma lista com as variáveis que recebem por parâmetro. A principal diferença, consiste em que as funções necessitam de guardar o tipo de retorno das mesmas. A entrada para as funções de *hardware* além de armazenarem os parâmetros da função, também armazena o identificador único de estas funções. Estes identificadores permitem fazer a ligação com as funções da Máquina Virtual como foi explicado anteriormente.

Relativamente à entrada das constantes, é armazenado na tabela a expressão dessa variável. Permitindo assim que as constantes não ocupem qualquer valor no *bytecode*. Sempre que a constante seja utilizada no código fonte, basta apenas trocar o seu uso por esta mesma expressão. Esta entrada na tabela de símbolos é usada maioritariamente na geração de código.

4.3 Sistema de tipos

A construção do sistema de tipos consistiu em implementar as regras apresentadas em 3.3. Para o desenvolvimento do sistema de tipos utilizou-se o padrão visitante.

O padrão visitante, é uma operação que pode ser aplicada sobre os elementos de uma estrutura, permitindo definir uma operação sobre esta estrutura sem alterar os elementos contidos nesta [17]. Como vai ser necessário produzir operações sobre a *AST* gerada por parte do *Xtext*, a utilização deste visitante vai facilitar o desenvolvimento do sistema de tipos, podendo assim visitar o conteúdo de cada nó da *AST*. Posteriormente, este visitante vai ser utilizado em todas as fases de compilação. De modo a abstrair o comportamento do visitante, e assim este poder ser aplicado em todas as várias fases de compilação foi desenvolvida uma classe que produz a visita na *AST*. Para cada nó da *AST* vai haver três visitas diferentes. A primeira visita é feita ao inicio do nó, a segunda ao nó da *AST* e a terceira ao fim do nó da *AST*. Por exemplo, a visita a um nó da *AST* que corresponda a

uma definição de variável local é feita da seguinte forma: a visita começa por visitar o nó da *AST* antes de os nós filhos de esse nó serem visitados, a seguir a essa visita vão ser visitados os nós filhos de esse nó e por fim visita-se o nó depois da visita aos seus nós filhos. Este processo é o mesmo para todos os nós existentes na *AST*. O *Xtext* por definição, apresenta uma classe que produz o padrão visitante, mas este para cada nó visita apenas os seus nós filhos, o que torna a visita bastante limitada.

A utilização deste padrão facilita muitos aspetos no desenvolvimento de compiladores. Ao utilizar este padrão, sempre que seja necessário adicionar uma nova funcionalidade à linguagem, basta apenas acrescentar a esta mesma classe as três visitas a esse nó [17].

O Xtext permite que sejam lançados erros e *warnings* para a *UI* sempre que exista um erro na validação de um programa. Assim sendo, sempre que um erro é encontrado durante esta validação, é lançado uma mensagem de erro de modo a avisar o programador. De realçar que o *Xtext* lança alguns erros por defeito relacionados com a utilização de variáveis e módulos que não existem.

O desenvolvimento do sistema de tipos foi dividido em cinco classes principais, *ModuleBootValidator*, *ActionsValidator*, *GlobalValidator*, *MemberValidator* e o *MemberBodyValidator*. De modo a validar cada nó da *AST*, estes validadores utilizam a classe desenvolvida com o padrão visitante. Como já foi referido, para a validação existe uma tabela de símbolos global para o programa, e uma tabela de símbolos local a cada membro. Ao ter uma tabela de símbolos local a cada membro, permite identificar se a utilização de uma variável é local ou global. Outra vantagem de ter a tabela de símbolos local, é não permitir que variáveis locais a um membro sejam utilizados por outro membro.

O validador *ModuleBootValidator*, além de validar os módulos de *hardware* e a secção "*boot*", à medida que os módulos vão sendo visitados, calcula a posição do *bit* de cada um dos módulos e o seu *byte* correspondente. Além de estes cálculos, os módulos visitados são adicionados à tabela de símbolos global.

O *ActionsValidator*, à medida que vai visitando a secção "*actions*" e validando-a, vai atribuir a cada evento e função de *hardware* um identificador único de modo a fazer a ligação entre o compilador e a Máquina Virtual. Neste validador, os eventos e funções de *hardware* são adicionados à tabela de símbolos.

Em relação ao *GlobalValidator*, este consiste em verificar se as variáveis têm os tipos corretos, se não existem variáveis globais repetidas, se não existem chamadas de funções, etc. Para a verificação dos tipos é utilizada uma pilha. Por exemplo, uma definição de variável global *int8 max = 1 + 2*, a visita vai percorrer o lado esquerdo e direito da expressão colocando na pilha os seus tipos ("*int8*" para cada uma das constantes). Depois de essa visita terminar, vai-se comparar os dois tipos, colocando na pilha um novo tipo equivalente a esses dois ("*int8*"). Este tipo será posteriormente comparado com o tipo da variável global ("*int8*"). Se, um dos valores da expressão fosse do tipo booleano iria ser lançado um erro ao programador por os tipos não serem equivalentes. Estas variáveis à

medida que são visitadas são colocadas na tabela de símbolos global.

A validação de membros é feita utilizando o *MemberValidator*, este validador consiste em verificar se o membro tem o tipo de retorno correto, se os parâmetros estão corretos e se este não é repetido. Posteriormente a este validação, é criada a tabela de símbolos local onde as variáveis locais vão ser armazenadas.

O *MemberBodyValidator* verifica o corpo de cada um dos membros do programa. Verificando as variáveis locais, os comandos utilizados e se o comando "*return*" é valido tendo em conta o tipo de retorno do membro. Para este validador também é utilizado uma pilha de forma a validar se os tipos de expressões estão corretos.

Devido à linguagem *Macaw* ser dividida em dois ficheiros (*.macaw* e *.macawh*) a validação de sistema de tipos varia para cada um dos ficheiros. Ao validar o ficheiro *.macawh* apenas vão ser executados os validadores *ModuleBootValidator* e *ActionsValidator*. Ao tentar validar o ficheiro *.macaw* todos os validadores vão ser executados. Ou seja, primeiro o ficheiro *.macawh* indicado vai ser validado e posteriormente o próprio *.macaw* irá ser validado.

4.4 Linguagem intermédia

A implementação da sintaxe da linguagem intermédia teve em conta dois objetivos principais. O primeiro objetivo, como já foi dito, é relativamente ao facto desta *AST* ser bastante mais simples que a *AST* gerada inicialmente pelo *Xtext*. O segundo objetivo, é relativamente ao calculo da pilha de chamada e operandos da linguagem. Com a *AST* gerada, é possível percorrer esta e calcular o tamanho total de esta pilha, este processo vai ser explicado na secção 4.8.

A implementação da estrutura da *AST* é bastante simples, esta implementação foi baseada na figura D.1. Para o lado esquerdo de cada regra da sintaxe foi criada uma classe e esta classe é estendida pelas classes do lado direito da regra. Por exemplo, a classe Programa é estendida pelas classes, Cabeçalho, Módulos, Globais, Funções e Mapa. A classe Cabeçalho, Módulos e Mapa não são estendidas por nenhuma visto estas utilizarem valores inteiros, sendo estes armazenados diretamente na classe. A classe Globais, é estendida por duas classes, uma relativa à atribuição de varíaveis do tipo *array* e outra relativa a variáveis que não sejam do tipo *array*. A classe Funções, é estendida pela classe Comandos. De modo a facilitar a construção da *AST* esta classe Comandos é estendida por três classes, uma para comandos sem parâmetros, outra para comandos com um parâmetro, e por fim, uma classe para comandos com dois parâmetros.

Para a geração de *bytecode* foi desenvolvido um visitante a esta *AST* com o mesmo objetivo do apresentado em 4.3, de modo a facilitar não só a geração de *bytecode*, mas também o cálculo do tamanho da pilha de chamada e operandos.

4.5 Gerador de código

O gerador de código é uma peça importante na fase de compilação, sendo este que vai transformar o código fonte em instruções que podem ser usadas e executadas pela Máquina Virtual da linguagem. O gerador de código da linguagem *Macaw* é dividido em duas fases. O primeiro passo consiste em criar a *AST* intermédia e o segundo passo consiste em visitar essa *AST* e gerar os dois ficheiros de *bytecode*. A necessidade de criar estes dois passos, deveu-se ao facto de se querer de calcular em tempo de compilação o tamanho total da pilha de chamada e operandos. Este valor só pode ser calculado depois de o *bytecode* ser gerado. As duas fases de geração de código vão ser explicados nas subsecções seguintes. No exemplo G.3, está representado um excerto do *bytecode* relativo ao exemplo *blink.macaw*.

4.5.1 Construção *AST* intermédia

De modo a construir a *AST* intermédia, fez-se uma visita ao grafo *SSA* gerado através das otimizações. À medida que os nós do grafo *SSA* são visitados, para cada elemento no nó, vai ser gerado o nó da AST intermédia correspondente a esses elementos. Além deste passo, também vão ser calculados os tamanhos indicados no cabeçalho do *bytecode* e a secção mapa que indica a posição do *main* no programa.

O *offset* do segmento de dados é sempre o mesmo para qualquer programa, ou seja, é representado pela constante doze. Esta constante representa o tamanho total do *bytecode*, que será sempre o mesmo valor independentemente do programa, permitindo à Máquina Virtual saber onde começa os dados no *bytecode* do programa. Relativamente ao *offset* do segmento de texto, consiste em somar o *offset* do segmento de dados, com o tamanho ocupado pelas variáveis globais do programa. Nestas variáveis globais também está incluído o tamanho ocupado pelos módulos de *hardware*. Este *offset* permite à Máquina Virtual identificar onde se inicia o código do programa. O *offset* do segmento do mapa "*Handler*", permite identificar a localização da secção Mapa no *bytecode*. Este é calculado somando o *offset* do segmento de texto, mais o tamanho ocupado pela tradução dos membros do programa. O tamanho ocupado pela tradução dos membros, é calculado à medida que a tradução do código fonte para instruções é produzida. O *offset* do segmento da pilha, é calculado somando o *offset* do segmento do "*handler*" mapa, com o tamanho de mapa de eventos, sendo este mapa de eventos explicado na secção 4.7. O *offset* do segmento de fila é calculado somando o *offset* do segmento da pilha, com o tamanho da pilha de chamadas e operandos. Por fim, o tamanho total do cabeçalho é calculado somando o offset do segmento da pilha mais uma constante de dois. Esta constante corresponde ao tamanho ocupado pela secção mapa, ou seja, ocupa dois *bytes*. A secção Mapa que identifica a localização do *main* no *bytecode*, é construída quando o *main* é traduzido para *bytecode*, identificando assim a sua posição.

4.5.2 Gerador *bytecode*

O objetivo do gerador de *bytecode* é gerar os dois ficheiros de *bytecode* (*.asm* e *.bc*). Ambos os ficheiros vão ter a mesma sequência de instruções, só que um é em formato de texto que permite ao utilizador identificar o que é gerado e o outro ficheiro gerado em *bytecode* é utilizado para executar a Máquina Virtual. Esta geração de código está de acordo com as funções de tradução indicadas na secção F.

Para a geração de ambos os ficheiros faz-se uma visita à *AST* intermédia utilizando o padrão visitante. A geração do ficheiro *.asm* consiste apenas em visitar a *AST* intermédia e gerar o código correspondente a cada nó visitado. A representação do ficheiro *.asm* vai ser equivalente à apresentada na figura E.1.

A construção do *bytecode* para o ficheiro *.bc* é diferente do *.asm*. Não só porque vai conter o *bytecode* em vez da representação textual, mas porque o *bytecode* para cada instrução é gerado tendo em conta o tipo da instrução. Como foi explicado na secção 3.5, uma instrução de *bytecode* pode ter três tipos diferentes. Por exemplo, a instrução *ADD* tem três tipos diferentes (*ADD8*, *ADD16* e *ADD32*). Para cada uma destas instruções, o *bytecode* vai ter de ser diferente para a Máquina Virtual poder identificar qual é o tipo da instrução. No caso do comando *ADD*, para todos os seus tipos, o *bytecode* vai ser *0x00*, de forma a diferenciar o tipo, é feito um *left shift* de duas casas para adicionar o tipo da instrução aos primeiros dois *bits* do *bytecode*. Se for o *ADD8* vai ser adicionado o inteiro 8, se for o *ADD16* vai ser adicionado o inteiro 16 e se for o *ADD32* vai ser adicionado o inteiro 32. Este processo vai ser aplicado a todos as instruções com mais de um tipo. Se a instrução não tiver mais de um tipo, vai ser adicionado o valor 0. Cada instrução do *bytecode* tem um *bytecode* único, isto poderia ser aplicado também a cada tipo de *bytecode*. Por exemplo, o *ADD8* tinha um *bytecode*, o *ADD16* tinha outro e o *ADD32* outro. Mas isto faria com que o leque de *bytecode* disponível para representar as instruções de *bytecode* diminui-se. A Máquina Virtual, quando lê o *bytecode* vai fazer o processo inverso, ou seja, um *right shift* de forma a obter a instrução de *bytecode*. Para obter o tipo da instrução do *bytecode* lê-se os primeiros dois *bits* que contem o tipo da instrução.

Cada elemento do cabeçalho é representado por dois *bytes*. Como já foi referido, cada conjunto de oito módulos é representado por um *byte*. Inicialmente todos os *bits* desse *byte* estão com o valor um, e sempre que um dos módulos que pertence a esse conjunto de oito esteja ligado, o seu *bit* correspondente é colocado a zero. As variáveis globais vão ser representados por o número de *bytes* correspondente ao seu tipo, ou seja, se a variável for do tipo "*int8*" ou "*boolean*" vai ser representado por um *byte*, se for do tipo "*int16*" vai ser representado por dois *bytes* e se for do tipo "*int32*" ou "*float*" vai ser representado por 4 *bytes*. Se a variável global for do tipo *array*, cada elemento do *array* vai ser representado da forma explicada anteriormente. Por exemplo, se o *array* for do tipo "*int32*" cada um dos seus elementos vai ser representado por 4 *bytes*.

4.6 Algoritmo de geração do grafo de estados

Uma das funcionalidades apresentadas pelo compilador da linguagem *Macaw*, é o facto de gerar um ficheiro *pdf* com o grafo de estados do *hardware*. O ficheiro *pdf* é gerado utilizando a aplicação *graphviz*. Este grafo, apresenta todas as possibilidades de estados que o *hardware* pode ter durante a execução de um programa. A sua utilização tem a vantagem de poder facilitar a depuração do comportamento do *hardware* nas aplicações. O utilizador que desenvolva a especificação de *hardware* do programa, pode verificar com este grafo se existe algum estado de *hardware* em que o *hardware* não pode estar. Por exemplo, no grafo apresentado na figura 4.5, referente ao código G.4, o utilizador que especificou o *hardware* pode não querer que o módulo de *hardware* "*power_light*" e o módulo "*button_light*" estejam ligados ao mesmo tempo. Assim teria de alterar a especificação de *hardware* de modo a resolver esse problema. Como se pode verificar no grafo, cada nó corresponde a um estado do *hardware* e cada aresta corresponde a uma ação definida no programa.

O algoritmo para desenvolver este grafo tem como estado inicial a informação relativa à secção "*boot*" do programa. No caso do grafo apresentado, o nó inicial vai conter o estado "*mcu328p*". O passo seguinte consiste em verificar a cláusula "*when*" de cada ação presente na secção "*actions*" e verificar se essa cláusula é válida de acordo com o nó inicial. Tendo em conta que o estado inicial é "*mcu328p*" vai-se verificar todos as ações que tenham pelo menos esse estado, ou seja, "*on mcu328p*" ou que não contenham um "*when*" definido. Ações que não tenham um "*when*" definido é equivalente a ter "*when true*", o que significa que essa ação é válida para todos os estados. Depois deste passo, sabe-se quais são as ações válidas para esse estado. Para cada uma destas ações vai-se criar os novos estados de acordo com cada "*turns*" dessas ações. Por exemplo, a função de *hardware* "*turnOnButtonLight*" tem "*turns on button_light*" sendo que o novo estado irá ser "*mcu328p button_light*". No caso de o "*turns*" não existir e se o "*when*" for válido para um certo estado, esse estado vai ter uma aresta para ele próprio. Depois deste processo terminar, vai-se executar exatamente o mesmo processo só que vai ser aplicado a cada um dos estados do grafo. O algoritmo vai ser executado até não existir mais nenhuma alteração a fazer no grafo. Quando o algoritmo terminar a sua execução, o grafo vai ser transformado numa "*string*", tendo em conta a sintaxe do *graphviz*, de modo a poder transformar o grafo numa represensentação visual com esta aplicação. A sintaxe do *graphviz* consiste inicialmente em definir os nós do grafo, e de seguida definir cada uma das arestas do grafo.

O grafo permite identificar claramente os estados que o *hardware* pode ter durante a execução do programa. Por exemplo, no estado "*mcu328p button_light*" são possíveis executar quatro ações diferentes em que três delas mudam o estado do *hardware*. Se a função de *hardware* "*turnOnPowerLight*" for executada vai passar para o estado "*mcu328p button_light power_light*". Enquanto que ao executar a função de *hardware* "*turnOffButton-*

Light" o estado vai ser alterado para "*mcu328p*". Em casos especiais, pode acontecer que pela especificação apresentada todos os módulos de *hardware* estejam desligados, e por isso, no grafo é apresentado o estado "*Empty*" que representa esse caso.

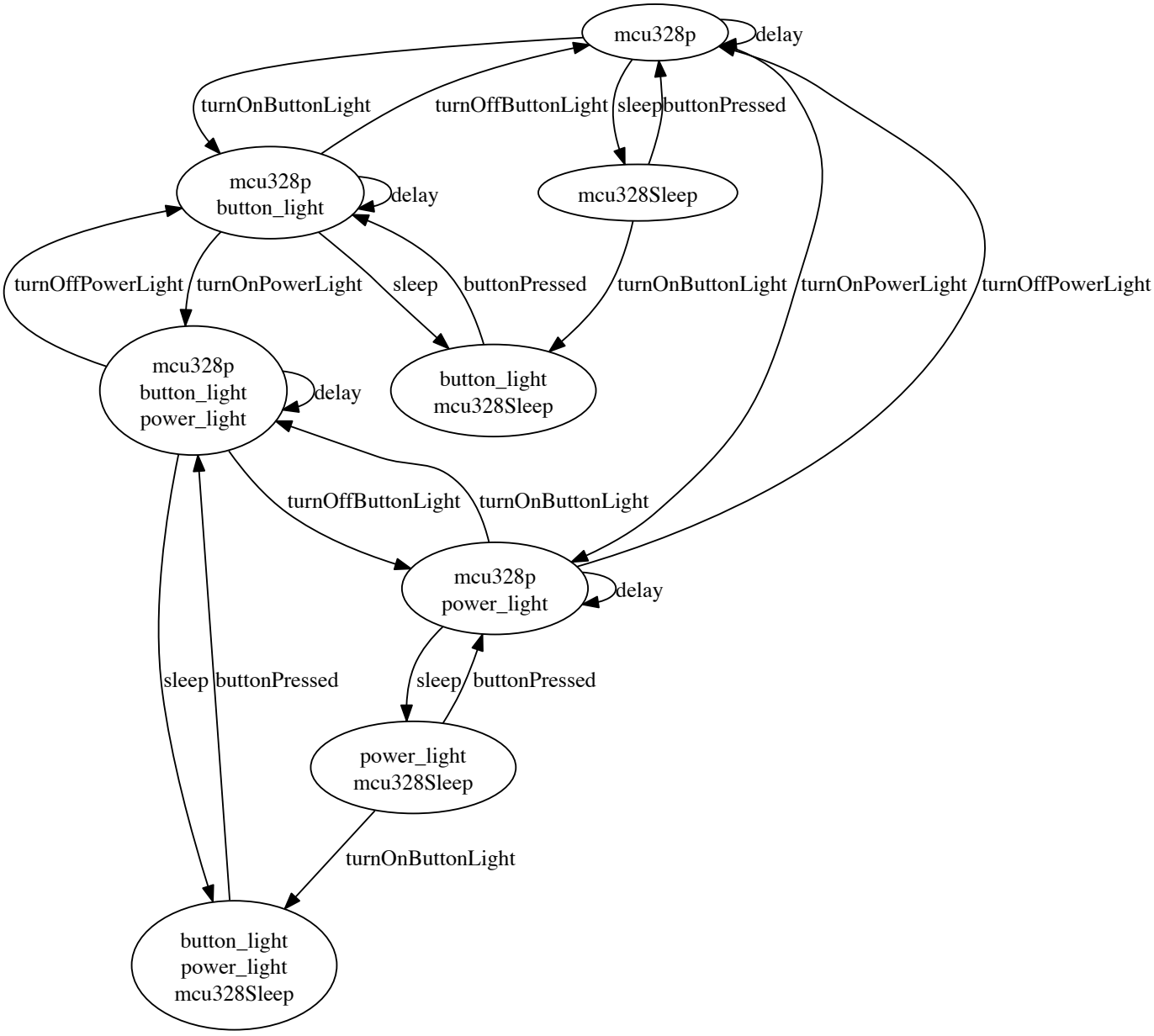


Figura 4.5: Grafo de estados do programa *blink-button-pressed*

4.7 Cálculo do tamanho do mapa de eventos

A estrutura mapa de eventos na Máquina Virtual permite fazer a ligação entre os eventos do programa e os seus "*handlers*". O facto de em tempo de compilação se conseguir calcular o tamanho deste mapa, permite que a Máquina possa alocar apenas a memória necessária para esta estrutura. Sem esta informação, havia duas maneiras de desenvolver esta estrutura, ou se alocava um número fixo de memória para esta estrutura, ou o Máquina Virtual calculava este tamanho antes de começar a execução.

O cálculo do tamanho do mapa de eventos é bastante simples, consiste em calcular o número total de eventos do programa (tendo em conta que neste valor vão estar incluídos os três eventos predefinidos), multiplicando este valor pelo tamanho ocupado em memória de cada entrada no mapa (este valor é três, pois no mapa vai ser armazenado o identificador único e a posição de memória do "*handler*" no *bytecode*). Ao resultado de este cálculo vai ser somado de novo o número de eventos do programa.

Para calcular este valor, utilizou-se o padrão visitante para visitar a secção "*actions*" no ficheiro *.macawh*. À medida que é encontrado um evento o contador global de eventos é incrementado em uma unidade. No fim, este contador vai acrescentar mais três eventos relativo aos três eventos predefinidos ("*Idle*", "*onBoot*" e "*onError*").

4.8 Cálculo do tamanho da pilha de chamada e operandos

O cálculo da pilha de chamadas foi dividido em dois passos, o primeiro consiste em construir o grafo de chamadas do programa, e o passo seguinte passa por percorrer esse grafo e calcular o tamanho da pilha de chamadas e operandos da Máquina Virtual.

O grafo de chamadas é apenas construído depois das otimizações serem executadas. Isto deve-se ao facto da possibilidade de algumas optimizações eliminarem chamadas de membros e/ou eliminarem os próprios membros. Por exemplo, os algoritmos podem identificar que uma chamada de um membro seja código inacessível, sendo essa chamada eliminada. Assim, elimina-se arestas desnecessárias do grafo de chamadas. Utilizou-se o algoritmo *depth first search* para visitar os nós do grafo *SSA*.

O grafo de chamadas é calculado a partir de cada "*handler*", ou seja, para cada um dos "*handler*" vai-se verificar que membros chamam. Por exemplo, o "*handler*" *main* chama o membro *f()* e esse membro *f()* chama o membro *g()*. Assim sendo, o grafo de chamadas vai ter uma aresta do "*handler*" *main* para *f()* e uma aresta de *f()* para *g()*. Este processo vai ser igual para todos os "*handler*" do programa. A única limitação, é se essa chamada for a uma função de *hardware*. Como não se consegue saber como estas foram

desenvolvidas, estas chamadas não são tidas em conta. Mesmo na Máquina Virtual estas funções só iram colocar na pilha o resultado da chamada se esta retornar alguma coisa.

O processo de cálculo da pilha de chamadas e operandos, consiste em calcular a pilha para cada um dos "*handler*" do programa e verificar qual tem o tamanho da pilha maior. Para o cálculo da pilha, é necessário usar o visitante desenvolvido para visitar a *AST* intermédia. Para este cálculo, vai-se visitar o *bytecode* que foi gerado pelo gerador de código contido na *AST* intermédia, analisando o espaço ocupado por cada uma das instruções na pilha. Por exemplo, se o "*handler*" *main* não fizer nenhuma chamada de membro, o seu código vai ser percorrido, calculando o tamanho ocupado na pilha por cada instrução de *bytecode* gerado para esse "*handler*", de modo a calcular o tamanho total da pilha. De forma a ter o resultado o mais aproximado possível com o da Máquina Virtual, sempre que existe uma chamada de função, vai ser calculado o tamanho da pilha relativa a esse membro chamado e quando este cálculo terminar, continua-se com o cálculo do membro que fez essa chamada. Por exemplo, o "*handler*" *main* chama o membro *f()*, ao percorrer o código gerado para o *main*, quando se encontrar o *bytecode CALL*, o código referente ao membro *f()* vai ser percorrido de modo a calcular o tamanho da sua pilha, quando este terminar, continua-se com o cálculo da pilha do "*handler*" *main*.

4.9 Plataforma *UI*

O *Xtext* apresenta por definição uma plataforma *UI*, mas esta é muito limitada. O *Content Assist*, por exemplo sugere variáveis que estão definidas num outro ficheiro, podendo levar o programador ao engano. Assim, alterou-se não só o *Content Assist*, como o *Outline* e o *Quick Fix* do *plugin*. Além destas características, o *plugin* tem uma preferência onde é possível colocar o caminho para o ficheiro *.dot* no sistema. Como foi dito anteriormente, este ficheiro é necessário para que o grafo de estados do programa seja gerado. Esta preferência encontra-se em: *Eclipse* > *Preferences* > *Macaw*.

Este *plugin* tem de ser instalado em um *Eclipse IDE* que contenha o *Xtext* previamente instalado. Dentro do projeto e da pasta *src* cria-se os ficheiros com extensão *.macaw* e *.macawh*. Os três ficheiros gerados (*.asm*, *.bc* e *.pdf*) vão ser gerados na pasta *src-gen*.

4.9.1 *Outline*

O *Outline* do *plugin* apresenta a informação global do programa. A apresentação do *Outline* é importante, permitindo ao programador obter informação global do programa sem ter de se deslocar à procura dessa mesma informação. O *Outline* pode ser visto como um resumo do programa. O *Outline* permite que ao clicar em alguma informação, este irá mostrar o local no código dessa informação. No caso do ficheiro *.macaw*, o *Outline* vai apresentar o ficheiro colocado na secção "*hardware description*", as variáveis globais

definidas e os membros. No caso do ficheiro *.macawh*, o *outline* vai apresentar todos os módulos definidos e a lista de "*actions*" definida.

Na figura 4.8, está apresentado o *Outline* de um programa *.macaw*. Como se pode verificar é apresentado o ficheiro utilizado no "*hardware description*" e os membros do programa. Para cada variável global é indicado o tipo da mesma e para cada membro é indicado o tipo dos parâmetros e o tipo de retorno do membro.

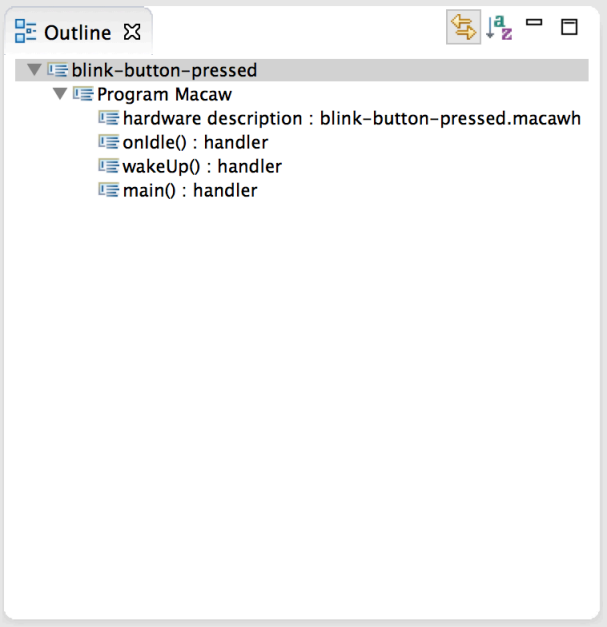


Figura 4.6: *Outline* exemplo de um ficheiro *.macaw*

Na figura seguinte, está apresentado o *Outline* exemplo de um ficheiro *.macawh*. No *Outline* deste ficheiro vai ser apresentado cada um dos módulos do ficheiro e também informação sobre a secção "*actions*". Para cada elemento desta secção, vai ser apresentado os seus parâmetros e o tipo de retorno do mesmo.

A plataforma *Xtext* facilita o desenvolvimento deste *Outline*. Como foi dito anteriormente, o *Xtext* cria automáticamente o pacote *macawc.ui*. Este pacote apresenta uma classe que se tem de alterar de modo a alterar o *Outline* apresentado por defeito.

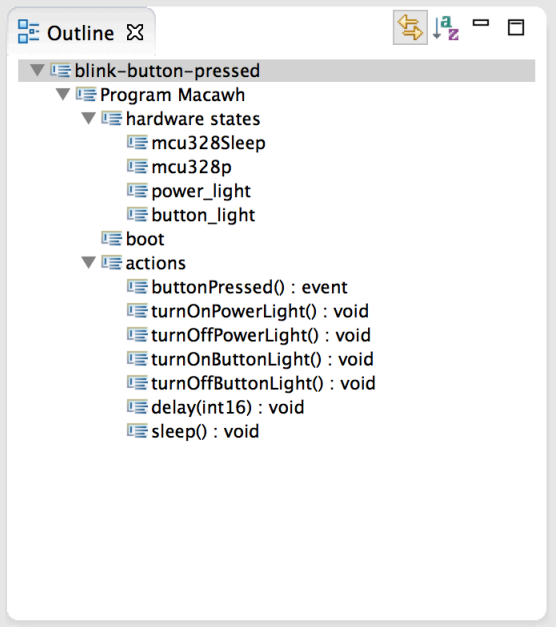


Figura 4.7: *Outline* exemplo de um ficheiro *.macawh*

4.9.2 *Quick fixes*

O *Quick Fixes* tem como objetivo ajudar, corrigindo certos erros causados pelo programador. Por exemplo, o programador ao definir duas variáveis com o mesmo nome é lançado um erro e o *Quick Fixes* apresenta uma solução para resolver esse problema. Neste caso, a solução seria apagar uma das variáveis repetidas.

De modo a tentar ajudar o mais possível os programadores da linguagem, tentou-se desenvolver um *Quick Fix* para cada um dos erros e *warnings* lançados pelo compilador. Mas, alguns erros não são possíveis de se resolver. Por exemplo, o erro devido ao caminho para ficheiro *.macawh* indicado na secção "*hardware description*" estar mal formatado, não é possível de apresentar uma solução para esse erro.

Mais uma vez, o *Xtext* facilita o desenvolvimento desta funcionalidade. No pacote *macawc.ui*, está incluído uma classe que se tem de programar com estas funcionalidades. Nesta classe, tem de se identificar todos os métodos com a *tag* @*Fix*, de modo a solucionar um erro, esta *tag* vai conter o nome do erro que se quer resolver. As alterações que se querem produzir, são aplicadas diretamente na *AST* gerada por parte do *Xtext*. Assim que é produzida uma alteração na *AST*, o *Xtext* atualiza o ficheiro referente a essa *AST*.

Na figura seguinte, é apresentado o funcionamento de um *Quick Fix*. Nesta figura, está presente um erro devido ao "*handler*" *main* ter um parâmetro, o que não é possível. A solução apresentada consiste em eliminar o parâmetro de este "*handler*". Ao clicar na solução apresentada, o parâmetro vai ser eliminado e o erro vai desaparecer.

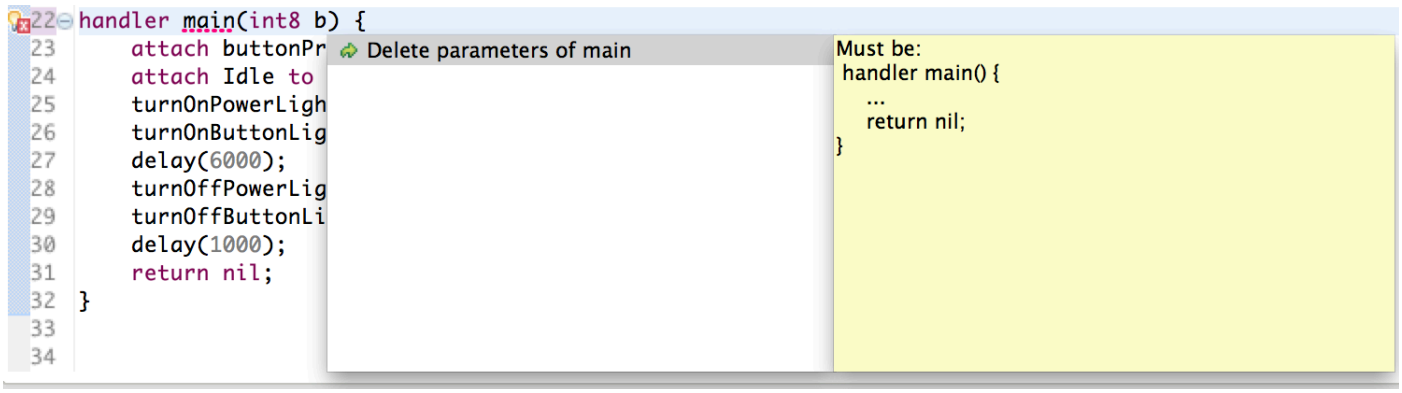


Figura 4.8: *Quick Fix* apresentada para solucionar erro de parâmetros "*handler*" *main*

Os elementos (*Quick Fixes*) que foram desenvolvidos são os seguintes:

- Sugestão de um tipo de retorno válido para os eventos ou funções de *hardware*. Este *Quick Fix* é apresentado quando é definido um evento ou uma função de *hardware* com um tipo de retorno inválido;
- Eliminação de módulos de *hardware* que estejam repetidos. Apenas vai ficar uma das definições repetidas do módulo;
- Se na secção "*boot*" for definido o comando "*and*" na expressão de *hardware*, é possível trocar este comando por o comando "*or*";
- Reordenação dos eventos e funções de *hardware* de modo a que os eventos fiquem antes das funções de *hardware*;
- Um *Quick Fix* que troque um tipo de parâmetro inválido por um válido. Por exemplo, se um parâmetro de um membro for do tipo "*void*" vão aparecer todos os tipos válidos como sugestão;
- Retificação de um "*main*" que tenha um tipo de retorno inválido;
- Eliminação de membros repetidos. Um membro é considerado repetido se existir mais que um com o mesmo nome;
- Eliminação de variáveis locais que sejam do tipo *array*;
- Criação do "*handler*" *main* quando este não é definido;
- Se na definição do "*handler*" *main* forem definidos parâmetros, estes podem ser eliminados;
- Se um membro que não seja do tipo "*void*" seja chamado fora de uma expressão, é possível trocar o seu tipo de retorno para "*void*";

- Ao existir uma chamada a um membro que não existe, esse membro vai ser criado com um tipo de retorno equivalente ao local onde está a ser utilizado. Se a chamada for numa expressão esse membro vai ter o tipo da expressão, caso contrário vai ter o tipo de retorno "*void*";
- Troca do tipo de retorno do *main* se este for definido com um tipo de retorno inválido;
- Remoção de atribuições de valor a variáveis que sejam constantes;
- Um membro ao não ser utilizado em qualquer local do programa pode ser removido;
- Eliminação de variáveis que nunca sejam utilizadas. Quer as variáveis globais, quer as variáveis locais;
- Eliminação de comandos "*attach*" em que o "*handler*" utilizado não tenha *side effect*;
- Eliminação de variáveis locais e globais que sejam repetidas;
- Alterar uma expressão de constantes pelo seu resultado. Por exemplo, se uma expressão for *2 + 2*, trocar essa expressão pelo valor 4.

4.9.3 *Content assist*

O *Xtext* também permite redefinir o *Content Assist* que vem por definição [9]. O *Content Assist* permite fornecer conteúdo sensível de forma a completar o contexto a pedido do utilizador. São apresentadas *popup* que fornecem opções de modo a completar uma frase [9]. O utilizador pode escolher uma das opções apresentadas. Este *Content Assist* pode ser usado não só para completar uma frase. Por exemplo, se se ativar o *Content Assist* num espaço vazio, este vai sugerir todos os membros que tenham o tipo de retorno "*void*", variáveis globais, variáveis locais entre outros.

O *content assist* por definição do *Xtext* apresentava variáveis globais e módulos de *hardware* de outros ficheiros. Para resolver isto, além de apresentar a informação necessário também se teve de restringir essa informação de outros ficheiros. O facto de o *Xtext* apresentar informação de outros ficheiros deve-se ao facto de usar o *Global Scoping*, este *scoping* permite fazer a ligação de variáveis entre ficheiros. Este *scoping* é muito útil para linguagens que queiram aceder a variáveis globais de outros ficheiros. A elaboração do *Content Assist*, consiste em visitar a *AST* do programa e procurar os elementos que se querem apresentar. Por exemplo, para apresentar todos os membros que tenham o tipo de retorno "*int8*", tem de se fazer uma visita à *AST* do ficheiro *.macawh* de modo a verificar as funções de *hardware*, mas também visitar a *AST* do ficheiro *.macaw* para verificar quais os membros com o tipo de retorno indicado.

Na figura 4.9, está apresentado o exemplo de uma *popup* apresentada pelo *content assist* relativamente aos membros que podem ser usados no comando "*attach*". São apresentados três "*handler*" que podem ser utilizados. Mas, se já tivesse escrito a *string* "*o*" apenas iria aparecer o "*handler*" "*onIdle*" no *Content Assist*.

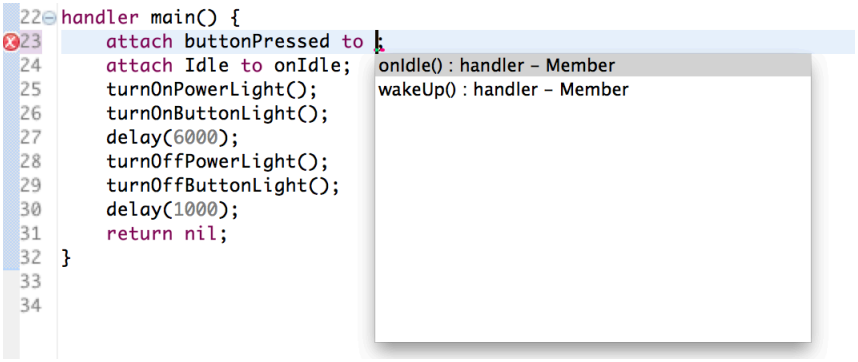


Figura 4.9: Exemplo do *Content Assist* da linguagem *Macaw*

4.9.4 Formatação

A formatação por defeito apresentada pelo *Xtext* teve de ser alterada. Esta formação colocava todo o código do programa numa linha. Ou o utilizador da linguagem formatava o programa à sua maneira ou então não havia solução.

Para resolver esse problema desenvolveu-se um algoritmo para formatar os dois ficheiros da linguagem (*.macaw* e *.macawh*). O desenvolvimento deste algoritmo teve de utilizar a *API* fornecida por parte do *Xtext* para o desenvolvimento. A *API* fornece métodos que permitem verificar certas condições das instruções da linguagem. Por exemplo, quando se inicializa uma variável local o que fazer a seguir ao "*;*", ou o que fazer a seguir ao "=", se existir uma atribuição de valor. De forma a desenvolver a formatação global do programa teve de se definir para cada nó da *AST*, como esse nó tem de ser formatado.

Capítulo 5

Algoritmos de otimização

A limitação de memória por parte destes dispositivos é um dos seus grandes problemas, podendo limitar bastante os programas que podem ser executados nestes dispositivos. Para resolver este problema, decidiu-se desenvolver algoritmos de otimização de forma a diminuir o *bytecode* gerado por parte do compilador. Teve de ser feita uma análise de formar a identificar quais os algoritmos que fariam com que o tamanho do *bytecode* pudesse ser diminuído. Por exemplo, o algoritmo *loop unrolling*, não é recomendável para a nossa linguagem, devido a este aumentar o código dos ciclos de forma a facilitar o seu processamento [1].

Os algoritmos escolhidos para as otimizações são aqueles que muito provavelmente vão diminuir o código gerado. Outro objetivo destes algoritmos, é não só ajudar os programadores, mas também aproximar o código fonte o mais possível do *bytecode* gerado. Estes algoritmos foram também utilizados para lançar *warnings* ou erros utilizando a *UI* do *Eclipse IDE*.

5.1 Processo de construção

Para a linguagem *Macaw* existiam duas possibilidades de aplicar os algoritmos de otimização. Aplicando diretamente na *AST*, ou construir um grafo *static-single assignment form*. Como alguns dos algoritmos necessitam de grafo *static-single assignment form* para poderem ser desenvolvidos, optou-se então por desenvolver este grafo. O processo de desenvolvimento deste grafo é muito complexo e passa por várias fases, o seu desenvolvimento seguiu o processo apresentado em [2].

O primeiro passo de desenvolvimento do grafo *SSA*, consiste em transformar a *AST* do código fonte num grafo de fluxo de controlo. Depois deste passo, usando os dominadores do grafo desenvolvido no passo anterior, a árvore de dominadores vai ser calculada. Para o desenvolvimento da fronteira de dominadores é necessário utilizar tanto o grafo fluxo de controlo como a árvore desenvolvida no passo anterior. Depois de estes estarem desenvolvidos pode-se construir o grafo *static single-assignment form*.

5.1.1 Grafo fluxo de controlo

O desenvolvimento do Grafo Fluxo de Controlo para a linguagem *Macaw* é bastante mais simples que para uma linguagem como o *Java*. Por exemplo, só o facto da instrução *return* na linguagem *Macaw* só poder ser utilizada no fim de um membro, facilita bastante o desenvolvimento do algoritmo.

No algoritmo desenvolvido utilizou-se o visitante explicado na secção 4.3, de forma a facilitar o desenvolvimento do algoritmo. O algoritmo consiste em fazer uma visita à *AST* do membro que está a ser otimizado. Este armazena sempre o nó corrente que está a ser calculado. Se o nó da *AST* que está a ser visitado for um nó do tipo "*attach*", uma inicialização de variável local, uma chamada de função ou uma atribuição de valor, estes elementos vão ser adicionados ao nó corrente do grafo. A diferença consiste nos comandos "*if*", "*unfold*" e "*ifStateContains*", pois estes comandos têm *joint points*, ou seja, dependendo da sua condição podem ir para nós diferentes do grafo. De modo a facilitar o algoritmo nesta fase, estes comandos têm estruturas predefinidas. Assim sendo, quando se faz a visita a um destes nós da *AST*, vai-se tentar criar essas estruturas predefinidas. Se por acaso um dos nós da estrutura não for possível preencher, então esse nó vai ficar vazio. Por exemplo, um nó "*if*" que não tenha "*else*", vai ter o nó referente ao "*else*" vazio.

Posteriormente a esta fase do algoritmo estar concluída, o grafo vai ser analisado de forma a eliminar os nós que estão vazios. Este processo consiste em ligar as arestas de entrada do nó vazio às arestas de saída do nó vazio. Este processo, faz com que o grafo gerado fique desordenado numericamente. Para resolver este problema, vai ser feita uma nova visita ao grafo de modo a ordenar os nós numericamente.

No desenvolvimento deste algoritmo encontrou-se outra limitação por parte do *Xtext*. Para os nós que têm uma condição, é necessário que as suas arestas guardem informação relativa à condição desse comando. Para isto, a forma mais simples de fazer, seria copiar diretamente a expressão do comando "*if*", "*unfold*" ou "*ifStateContains*". Mas, ao fazer isto, o elemento que é copiado é colocado a *null* na *AST*, se depois for necessário obter essa expressão, esse valor vai estar a *null*. Isto deve-se ao facto de o *Xtext* não permitir ciclos dentro da *AST*. Ao colocar essa expressão num outro local da *AST*, faria com que a *AST* deixa-se de ser uma árvore e passa-se a ser um grafo. Para resolver este problema, teve de ser desenvolvido uma classe para as Expressões e para as Expressões de *Hardware* onde iriam ser guardadas informações sobre a expressão original.

Na figura 5.1, está apresentado o grafo de Fluxo de Controlo do "*handler*" *main* do exemplo *blink* apresentado em G.2, este exemplo está explicado na secção 7.1. Nesta figura para cada nó está representado o pedaço de código correspondente. Enquanto o valor do comando "*unfold*" for menor que dez, este vai ser executado. Por fim, vai acabar no nó final do grafo, ou seja, o nó com o número seis.

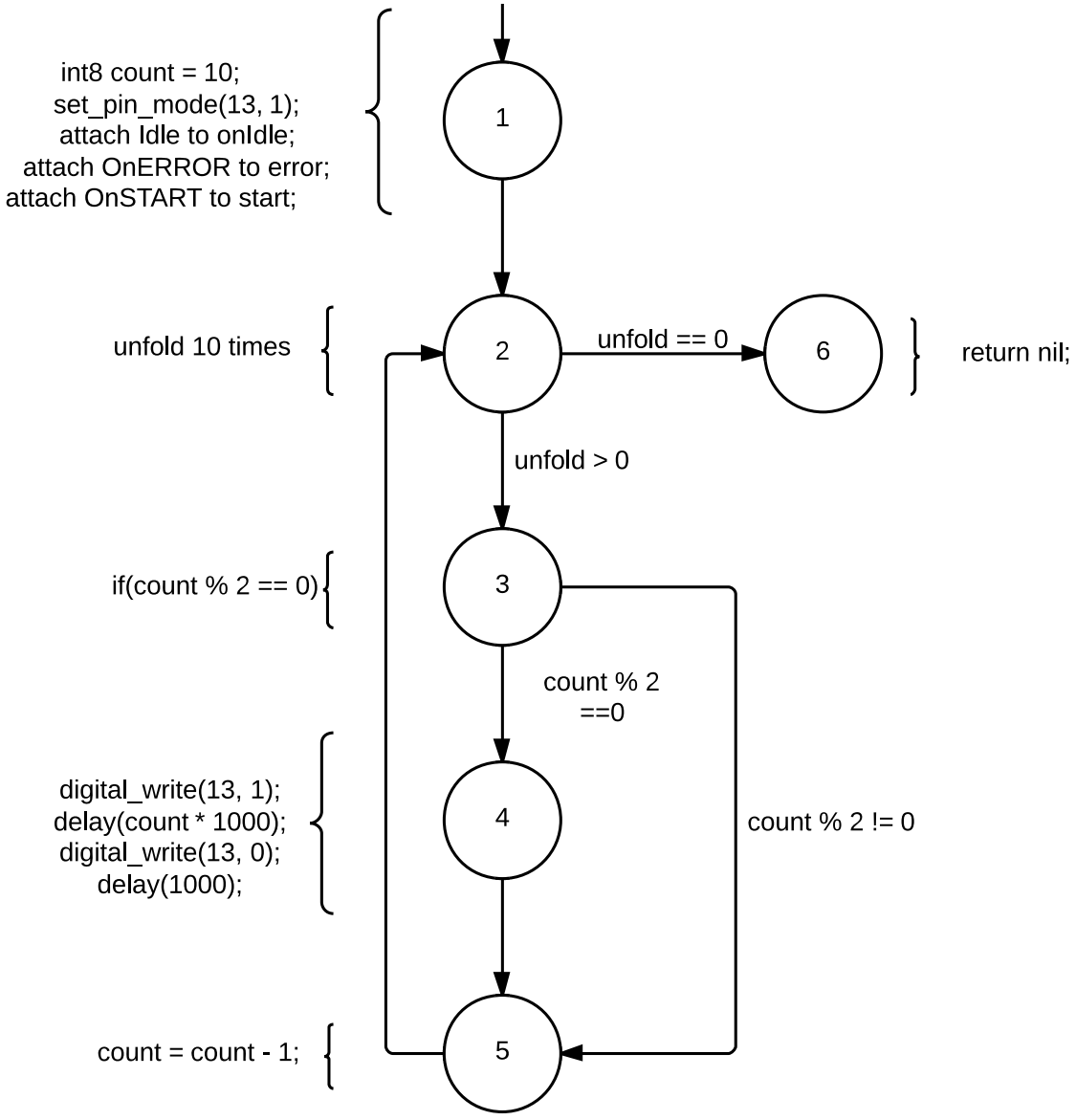


Figura 5.1: Grafo de fluxo de controlo exemplo *main blink*

5.1.2 Árvore dominadores

O desenvolvimento da Árvore de Dominadores é dividido em três fases. A primeira consiste em calcular os dominadores do grafo, posteriormente a este passo vai-se identificar quais são os nós estritamente dominadores de modo a poder desenvolver a árvore.

Um nó *x* é dominado por um nó *y* se todos os caminhos que vão para *x* passam por *y*. Por exemplo, para o grafo da figura 5.1, o nó 2 domina os nós 2, 3, 4, 5 e 6. Este resultado deve-se ao facto de todos os caminhos para esses nós terem de passar pelo nó 2. O algoritmo para calcular os dominadores é bastante simples de se entender. Faz-se *n* visitas ao grafo fluxo de controlo, em que *n* é o número de nós do grafo. E em cada visita do grafo o nó pelo qual se quer identificar os dominadores é marcado como não visitável. Ao marcar o nó desta forma, quando a visita chega a este nó, não continua. Depois da visita ao grafo terminar, pode-se identificar quais foram os nós que não foram visitados, assim consegue-se determinar os nós que são dominados por esse nó.

Como foi dito anteriormente, o passo seguinte consiste em calcular os nós estritamente dominadores. Um nó *x* é estritamente dominado por um nó *y*, se *x* for dominado por *y* e se *x* for diferente de *y*. Isto impede que um nó se possa dominar a ele próprio. Por exemplo, o nó 2, neste caso, domina estritamente os nós 3, 4, 5 e 6. Para o cálculo destes dominadores basta alterar os dominadores calculados no processo anterior.

O último passo consiste em calcular a árvore de dominadores. A árvore de dominadores é constituída por todos os nós do grafo fluxo de controlo. A diferença consiste na ligação entre esses nós. Um nó na árvore de dominadores está ligado aos seus dominadores estritos, desde que exista uma aresta para esses mesmos nós no grafo fluxo de controlo. Como se pode verificar na figura 5.2, apesar do nó 2 dominar estritamente os nós 3, 4, 5 e 6, este apenas vai estar ligado ao nó 3 e 6 por haver uma aresta do nó 2 para estes no grafo fluxo de controlo.

A vantagem de utilização desta árvore consiste em que para cada nó, os seus nós filho vão ser dominados imediatamente por estes.

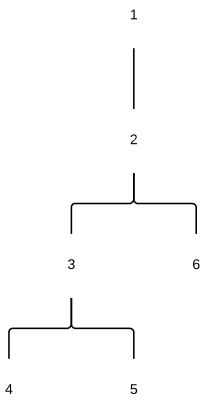


Figura 5.2: Árvore de dominadores do exemplo *main blink*

5.1.3 Fronteira dominadores

O principal objetivo da fronteira de dominadores, é não só identificar quais os nós do grafo que necessitam de colocar funções ϕ, mas também determinar quantos argumentos esta função irá ter. Esta função vai ser colocada em locais do código onde não seja possível determinar qual a variável que vai ser executada. Por exemplo, se num comando "*if*" uma variável *x* é utilizada tanto no "*then*" como no "*else*", permite identificar se esta variável é utilizada no "*then*" ou no "else". Esta função ϕ vai ser explicada na secção 5.1.4 ao detalhe.

Um nó *x* é considerado pertencer à fronteira de um nó *y* se forem verificadas duas condições. O nó *x* não pode dominar estritamente o nó *y* e *x* domina um dos nós que precedem o nó *y* [2].

O cálculo das fronteiras de dominadores para cada nó utilizou o algoritmo apresentado em [2]. Este algoritmo utiliza o grafo fluxo de controlo e a árvore de dominadores desenvolvidos nos passos anteriores. Este é desenvolvido utilizando duas listas, uma lista que contém os nós que são sucessores de um nó *n* no grafo fluxo de controlo, que não sejam estritamente dominados por *n*. A outra lista, vai conter os nós que estão na fronteira de dominadores de *n*, ou seja, que não são dominados estritamente por um dominador imediato do nó *n*. Esta lista apenas pode conter os nós que são filhos do nó *n* na árvore de dominadores.

Ao fazer uma união destas duas últimas listas, consegue-se obter a fronteira de dominadores para o nó *n*.

A tabela resultante de aplicar o algoritmo utilizando a árvore de dominadores e o grafo de fluxo de controlo apresentados anteriormente é o seguinte:

n	*DF(n)*
1	{ }
2	{2}
3	{2}
4	{5}
5	{2}
6	{ }

Tabela 5.1: Fronteira dominadores exemplo *main blink*

5.1.4 *Static Single-Assignment Form*

O grafo *SSA* é uma melhoria à cadeia de usos definições. Estas cadeias são utilizadas por muitos algoritmos de modo a encontrar os usos para cada definição de variável. O grafo *SSA*, é considerado como uma representação intermédia. Este grafo apresenta várias vantagens para o desenvolvimento de otimizações.

A primeira vantagem, é relativamente ao facto de cada variável apenas ser definida uma única vez, este aspeto facilita o desenvolvimento de alguns algoritmos de otimização. Para garantir esta propriedade, é necessário que internamente o compilador renomeie todas as variáveis definidas. Por exemplo, o algoritmo *Dead Code Elimination* desenvolvido utiliza esta característica, o que o torna muito mais simples de desenvolver. Ao ter uma variável que é definida e nunca é usada, essa definição pode ser marcada como código "morto". Este aspeto é uma técnica semelhante à numeração de valores.

Outra característica importante é a utilização de funções ϕ, sendo estas colocadas nos comandos "*if*", "*unfold*" e "*ifStateContains*". Enquanto que se o programa for sequencial, não há dificuldades em identificar qual o caminho de execução, se este tiver mais que um caminho possível, a identificação do caminho de execução do programa não é óbvio. As funções ϕ ao serem colocadas nestes comandos, torna possível a certos algoritmos identificar qual os nós destes comandos que vão ser executados, identificando o caminho de execução do programa. Por exemplo, o algoritmo *Conditional Constant Propagation* ao usar estas funções consegue, em alguns casos, identificar se é o "*then*" ou o "*else*" do "*if*" que vai ser executado.

O desenvolvimento deste grafo é dividido em duas fases. A primeira consiste em identificar quais os nós que necessitam de ter as funções ϕ e posteriormente renomear as varáveis de modo a que cada variável seja definida apenas uma vez.

Como foi referido anteriormente, as funções ϕ vão ser colocadas em nós que tenham *joint points*. Para facilitar o processo, podia-se colocar em todos os *joint points* estas funções para todas as variáveis, mas isto é desnecessário. Só ia fazer com que os algoritmos tivessem mais trabalho a fazerem os seus cálculos. Por exemplo, no exemplo apresentado na figura 5.3 o comando "*if*" não tem função ϕ. Isto deve-se ao facto de esse comando não definir qualquer variável no seu corpo. Se este definisse alguma variável, então seria necessário ser colocado uma função ϕ. A função ϕ, antes do comando "*unfold*" significa o seguinte: no inicio da execução do comando "*unfold*", $count_2$ pode ter o valor de $count_1$ ou de $count_3$.

De modo a identificar os locais onde é necessário colocar as funções ϕ utiliza-se a fronteira de dominadores e o grafo fluxo de controlo. O algoritmo desenvolvido foi baseado no algoritmo apresentado em [2]. Este algoritmo consiste inicialmente em identificar todas as variáveis que são definidas no método. Para procurar todas estas definições fez-se

uma visita ao grafo fluxo de controlo, à medida que uma definição de uma variável é encontrada esta é armazenada, como também o nó em que esta ocorre. Depois deste passo, para cada definição de variável, vai-se adicionar uma função ϕ nos nós que fazem parte da fronteira de dominadores do nó onde ocorreu essa definição. O número de argumentos de uma função ϕ, depende do número de saídas de cada nó da fronteira de dominadores. Para o exemplo dado na figura 5.3, a variável *count* é definida no nó 5, ao aceder à tabela de fronteira de dominadores, o nó 2, é o único pertencente à fronteira por isso este vai ter a função ϕ. Se por acaso existisse outro nó na fronteira, este iria ser adicionado. Devido ao nó 2 apenas ter dois sucessores, esta função apenas vai ter dois argumentos.

O último passo para a construção do grafo *SSA*, consiste em renomear as variáveis por parte do compilador. Como foi dito anteriormente, se o programa fosse sequencial não haveria qualquer dificuldade em renomear estas variáveis. Este algoritmo consiste em fazer uma visita recursiva à árvore de dominadores. O algoritmo tem um contador para cada variável, ao encontrar uma definição de valor esse contador é incrementado, ao encontrar um uso de uma variável, essa variável vai obter o valor do contador de forma a fazer uma ligação à sua definição. Este procedimento é exatamente igual para as funções ϕ. Garantido assim que cada variável é definida apenas uma vez. O grafo *SSA* desenvolvido vai conter as arestas do grafo fluxo de controlo, mas também arestas de cada definição para cada um dos seus usos (arestas *SSA*). Estas arestas que fazem a ligação entre os nós de definição e uso dessa variável, vão sendo contruídas à medida que as variáveis vão sendo renomeadas. Uma outra vantagem que este grafo apresenta, é o facto de se poder fazer uma visita semelhante à visita da *AST* apresentada na secção 4.3.

O grafo *SSA* equivalente ao grafo fluxo de controlo apresentado na figura 5.1 é apresentado na figura 5.3. De realçar, que este apenas apresenta as arestas referentes ao grafo fluxo de controlo. O grafo gerado pelo compilador iria ter uma aresta da definição de $count_1$ para o seu uso na função ϕ. Para $count_2$ e $count_3$ as arestas iriam ser construídas da mesma forma.

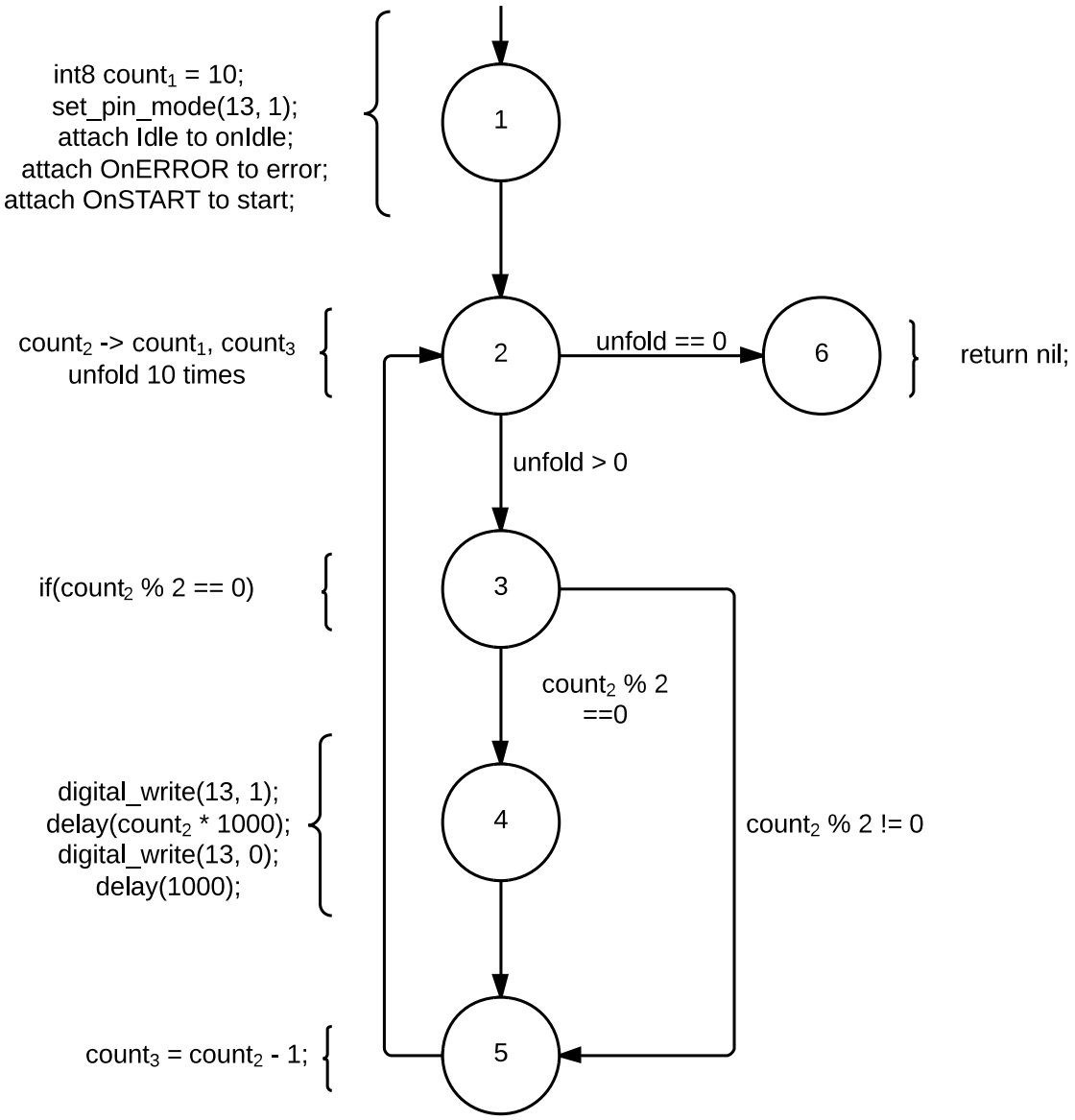


Figura 5.3: Grafo *SSA* do exemplo *main blink*

5.2 *Copy propagation*

O algoritmo *Copy Propagation* consiste em propagar os valores de uma definição de variável para cada um dos seus usos. Esta definição tem de cumprir certos requisitos para que possa ser propagada. Tem de cumprir um dos requisitos seguintes:

1. Atribuição de valor à definição tem de ser um valor constante;
2. Atribuição de valor à definição tem de ser uma variável.

Como se pode verificar pelos requisitos, não se pode aplicar este algoritmo se a atribuição de valor for uma chamada de função ou uma expressão matemática.

O algoritmo consiste em percorrer o grafo *SSA*, visitando cada nó do grafo, e em cada definição de valor verificar se é possível propagar o seu valor segundo os requisitos apresentados. Se isto for possível, então vai-se identificar todas as arestas *SSA* dessa definição para cada uso sendo esses usos trocados pelo valor da definição da variável. Devido a este processo, estas arestas, da definição para cada um dos seus usos vão ser eliminadas. Por exemplo, se for atribuído o valor constante um a uma variável x, todos os seus usos, vão ser substituídos por este valor. A vantagem deste algoritmo é poder facilitar alguns cálculos, ao propagar esta constante pode permitir que uma expressão possa vir a ser calculada com outros algoritmos.

5.3 *Constant folding*

O algoritmo *Constant Folding* permite efetuar cálculos de expressões. Estas expressões para poderem calculadas pelo algoritmo têm de ter apenas valores constantes. Por exemplo, se a expressão for um mais dois, esta expressão pode ser calculada, mas se esta expressão contiver uma variável não pode ser calculada. Este algoritmo também calcula as expressões dos comandos "*if*" e "*unfold*". Como também as expressões no comando "*return*".

O algoritmo desenvolvido consiste em percorrer o grafo *SSA* visitando cada um dos seus nós. Sempre que for encontrada uma expressão na visita vai-se tentar calcular o seu valor. Se esta expressão for calculada, então a expressão vai ser substituída por o resultado calculado. Uma expressão ao ser calculada com este algoritmo, vai diminuir consideravelmente o tamanho do *bytecode* que vai ser gerado. A conjugação de o algoritmo *Copy Propagation* e este algoritmo aumenta o número de expressões que podem ser calculadas. Se o algoritmo *Copy Propagation* propagar um valor constante referente a uma variável, pode fazer com que este algoritmo possa calcular uma expressão que anteriormente não era possível calcular.

5.4 *Conditional constant propagation*

O algoritmo *Conditional Constant Propagation* permite calcular algumas expressões e identificar código inacessível. Código inacessível corresponde a nós no grafo fluxo de controlo que nunca vão ser executados em todas as possibilidades de execução do programa. Este código pode ser removido, diminuindo assim o tamanho do *bytecode*.

O objetivo deste algoritmo, consiste em descobrir os valores que são constantes em todas as possibilidades de execução do programa e propagar essas constantes o mais possível no programa [30].

O resultado deste algoritmo consiste em um reticulado para presentar cada uma das definições de valor em cada nó do grafo *SSA*. O reticulado usado neste algoritmo é apresentado na Figura 5.4 [31]. O reticulado é uma estrutura algébrica que permite fazer relações entre os seus elementos. No caso do reticulado deste algoritmo, este é constituído por três níveis, o valor mais alto é $\top$, os valores intermédios consistem num número infinito de constantes que uma definição de variável pode ter e por fim o nível mais baixo é o $\bot$ [30]. No algoritmo não é possível a uma variável que suba de nível no reticulado, ou seja, não é possível uma variável ao estar no nível $\bot$ subir para um outro nível do reticulado. Na mesma lógica, se o reticulado estiver no nível das constantes este já não vai subir para o nível $\top$ do reticulado [30]. No fim da execução do algoritmo, se o reticulado de uma variável tiver um valor constante significa que em todas as execuções possíveis do programa essa variável vai ter sempre o mesmo valor [30]. Se o valor for $\bot$, significa que não se consegue garantir que essa variável vá ter um valor constante e se o valor for $\top$, significa que naquele momento não é possível determinar se o valor vai ser constante ou não [30]. No inicio de execução do algoritmo, o reticulado de cada um das variáveis vai ter o valor $\top$.

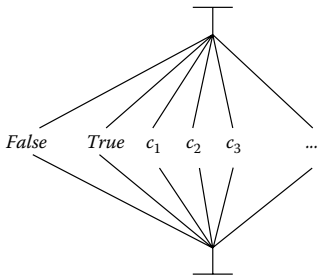


Figura 5.4: Reticulado do algoritmo *Conditional Constant Propagation*

A execução do algoritmo consiste em baixar o nível do reticulado para todas as variáveis consoante a informação que vai recebendo. Esta informação vai ser adicionada de acordo com as regras apresentadas na figura 5.5 [31]. O *any* nestas regras, consiste em qualquer valor constante que esteja no segundo nível do reticulado. Para a avaliação dos comandos "*if*" e "*unfold*" é útil utilizar as regras apresentadas na figura 5.6 [31]. Estas

regras permitem calcular as expressões destes comandos, aumentando a probabilidade de encontrar código inacessível.

$$any \sqcap \top = any \tag{5.1}$$

$$any \sqcap \bot = \bot \tag{5.2}$$

$$\varphi_i \sqcap \varphi_j = \varphi_i \; \textit{if} \; \varphi_i = \varphi_j \tag{5.3}$$

$$\varphi_i \sqcap \varphi_j = \bot \; \textit{if} \; \varphi_i \neq \varphi_j \tag{5.4}$$

Figura 5.5: Regra para o operador $\sqcap$

$$true \vee any = true \tag{5.5}$$

$$any \vee true = true \tag{5.6}$$

$$false \wedge any = false \tag{5.7}$$

$$any \wedge false = false \tag{5.8}$$

Figura 5.6: Regra para expressões

O algoritmo utiliza as arestas do grafo *SSA* correspondentes ao fluxo de controlo, como as arestas de cada definição de variável para os seus usos (arestas *SSA*). As arestas de fluxo de controlo são utilizadas para identificar código inacessível, enquanto que as arestas *SSA* são utilizadas para propagar as constantes pelo código. O algoritmo utiliza duas listas, em que uma guarda os nós do grafo *SSA* que são visitados e a outra armazena as arestas *SSA* quando existe uma alteração de valor [31].

As arestas relativas ao fluxo de controlo do grafo *SSA* são adicionadas à lista sempre que um nó do grafo tenha apenas um sucessor. Outra situação é quando encontra um nó com *joint points*. Ao encontrar um *joint point* (significa que é um "*unfold*", "*ifStateContains*" ou "*unfold*") a sua expressão vai ser avaliada adicionando à lista o ramo correspondente [31]. Por exemplo, se ao avaliar um "*if*" o resultado da expressão for verdadeira, então o ramo verdadeiro do "*if*" vai ser adicionado. Se o resultado for falso, então vai ser adicionado o ramo falso. Por fim, se o resultado for $\bot$, tanto o ramo verdadeiro como falso vão ser adicionados à lista.

As arestas *SSA* são adicionadas à lista de arestas *SSA*, sempre que existe uma mudança de valor nessa definição. Garantindo assim, que sempre que uma definição de variável te-

nha o seu valor de reticulado alterado todos os seus usos vão ter em conta essa mudança [31]. No fim da execução do algoritmo, se um dos nós do grafo *SSA* não for visitado significa que este nó e todo o código contido nele pode ser marcado como código inacessível.

5.5 *Dead code elimination*

O grafo *SSA* facilita o desenvolvimento do algoritmo *Dead Code Elimination*. O desenvolvimento deste algoritmo sem este grafo iria ser muito mais complicado. Este algoritmo consiste em analisar o grafo e verificar se existe código "morto". Uma definição de variável quando tem a sua lista de usos vazio é considerado "morto", essa variável ao nunca ser usada, pode ser eliminada visto não ter grande utilidade no programa desenvolvido [2]. Ao eliminar este código, estamos a otimizar o tamanho do *bytecode*, como também a fazer com que a Máquina Virtual não tenha de executar instruções desnecessárias. A definição de uma variável global é uma exceção, pois ao aplicar o algoritmo a um membro do programa não se sabe se essa definição não é importante num outro local do programa, assim sendo não pode ser eliminada. Uma definição do tipo $x = a + b;$ ao ser eliminada, as definições de a e b deixam de ter uma aresta para esta expressão. Se a variável a ou b apenas fossem usadas nesta expressão, pode fazer com que estas também se tornem código morto. Para resolver este problema, o algoritmo desenvolvido utiliza uma lista de modo a verificar se na execução anterior do algoritmo algo mudou no grafo *SSA*.

Como foi referido, o algoritmo desenvolvido tem uma lista de modo a verificar as alterações produzidas no grafo *SSA*. Esta lista inicialmente contém todas as variáveis definidas no membro que esta a ser otimizado. O algoritmo consiste em tirar um elemento de cada vez de esta lista e verificar se tem algum uso, se tiver, então não faz nada, caso contrario, vai eliminar essa definição do grafo e todas as variáveis que estão na sua expressão vão ser adicionadas à lista [2]. Devido a utilizar o grafo *SSA*, estes usos são fáceis de verificar acedendo apenas às arestas do grafo *SSA*. O algoritmo vai ser executado até que não exista nenhuma variável na lista. Neste algoritmo teve de ser adicionado uma verificação de modo a que definições de variáveis globais não sejam eliminadas.

Este algoritmo permite também que algumas funções ϕ sejam eliminadas. Permitindo assim, que o algoritmo *Conditional Constant Propagation* possa identificar mais facilmente código inacessível.

5.6 Outros algoritmos de otimização

Além dos algoritmos apresentados anteriormente, foram desenvolvidos outros que achamos serem necessários. Mais uma vez o objetivo de estes é diminuir o *bytecode* gerado e também ajudar o programador.

1. Algoritmo que permite eliminar "*unfold*", "*if*" e "*ifStateContains*" vazios. Este algoritmo consiste em fazer uma visita ao grafo *SSA* e analisar se existe algum destes comandos com o corpo vazio. É pouco provável que um programador introduza estes comandos vazios, mas ao executar outras optimizações pode acontecer que o código todo no corpo destes comandos seja eliminando, tornado desnecessário ter estes comandos;

2. Algoritmo que permite eliminar comandos "*unfold*" sem *side effect*. Um comando "*unfold*" é considerado sem *side effect*, se não existir nenhuma definição de variável global no seu corpo, se não fizer nenhuma chamada de função a um membro impuro e se nenhuma das variáveis definidas no "*unfold*" sejam utilizadas fora de este. O algoritmo desenvolvido consiste em analisar o grafo *SSA* para o membro que está a ser analisado e verificar se existe algum "*unfold*" que tenha estas características;

3. Algoritmo que elimina comandos "*if*" em que a sua expressão ou o resultado da expressão seja *true* e comando "*unfold*" que a sua expressão ou o resultado dela seja um. Para estes dois casos não é necessário criar o *bytecode* relativamente ao cabeçalho destes comandos, basta gerar código referente ao corpo de cada um destes. Este algoritmo é utilizado depois de o algoritmo *Conditional Constant Propagation* ser executado e consiste em verificar se os comandos "*if*" e "*unfold*" cumprem estas condições;

4. Algoritmo que calcula as expressões das variáveis globais, permitindo assim que seja apenas necessário gerar código para o resultado das mesmas. A ideia do algoritmo é bastante semelhante ao *Constant Folding* só que é aplicado às variáveis globais;

5. Algoritmo que verifica se alguma das variáveis globais não é usada. Depois de todos os algoritmos de otimização serem executados, vai-se percorrer todos os membros do programa de modo a verificar se existe alguma variável global que não seja usada. Se alguma não for usada, não é necessário gerar código para a mesma;

6. Algoritmo que elimine membros puros que tenham tipo de retorno "*void*" ou "*handler*". Membros com estas características podem ser eliminados visto não trazerem

nada de novo ao programa devido a não terem *side effects*. O algoritmo consiste em analisar para cada um dos membros "*void*" e "*handler*" se estes são puros. Se assim forem, são eliminados e o código não é gerado. Este algoritmo é executado depois das otimizações, pois inicialmente um certo membro pode ser impuro mas depois de as otimizações tornar-se um membro puro;

7. Algoritmo que elimina funções que retornem constantes e que não tenham *side effect*. O algoritmo ao analisar que uma função retorna uma constante, então todas as chamadas a essa função podem ser trocadas por esse mesmo valor constante. Esta função tem de ser pura para que este algoritmo possa ser aplicado;

8. Algoritmo que substitui todos os usos de uma variável constante pela sua definição. O uso de este algoritmo permite que as variáveis globais que sejam do tipo constante não ocupem memória no *bytecode*;

9. Algoritmo que elimina membros que nunca são usados. Este algoritmo consiste em fazer uma visita ao grafo de chamadas verificando quais são os membros que nunca são chamados. Ao eliminar estes membros permite diminuir o *bytecode* gerado.

Capítulo 6

Testes

A fase de testes é muito importante no desenvolvimento de um projeto. O desenvolvimento de testes permite escrever código com qualidade em que quase todos os erros conseguem ser verificados. Uma das vantagens presentes em desenvolver testes consiste na sua fácil utilização, onde basta carregar num único botão para correr a bateria de testes. Sem uma bateria de testes, sempre que uma nova funcionalidade for adicionada à linguagem tem de se verificar manualmente se essa funcionalidade funciona e se o resto da linguagem está a funcionar de forma esperada. O processo de verificação manual é bastante dispendioso. Ter uma bateria de testes permite que sempre que seja lançada uma nova versão da linguagem ou alguma nova funcionalidade seja implementada, basta correr os testes e verificar se essa atualização à linguagem trouxe algum erro. Se ao testar essa alteração aparecer um teste que falhe então é necessário verificar se esse erro é suposto acontecer com a nova versão. Se existir um erro os testes têm de ser atualizados, caso contrário tem de se verificar qual é o erro que a nova atualização tem.

O *Xtext* permite o desenvolvimento desta bateria de testes utilizando a versão 4 do *JUnit* [5]. Os testes em quase todos os casos de uso são executados pelo compilador como se tivessem a ser executados no *plugin* gerado pelo *Xtext*. Existiram certas exceções onde o comportamento do compilador, com um teste era diferente do comportamento do compilador a ser executado no *plugin*. Por exemplo, não se conseguiu testar o comando "*ifStateContains*" devido ao *Xtext* lançar um erro a dizer que o módulo de *hardware* utilizado neste comando não existe, mesmo este estando definido no ficheiro *.macawh* indicado no teste. Enquanto que se o mesmo código utilizado no teste fosse executado no *plugin* esse erro já não era lançado.

Nas secções seguintes vai ser explicado como foi desenvolvida a bateria de testes e os erros que foram encontrados com esta bateria de testes.

6.1 Testes realizados

A bateria de testes foi desenvolvida de forma modular, ou seja, foi produzido um conjunto de testes específicos para o sistema de tipos, outro conjunto para os algoritmos de optimização e outro conjunto para a geração de código. Para não ser necessário correr cada um destes conjuntos individualmente, para cada um deles foi criado um *test suite*. Um *test suite* permite executar automaticamente um conjunto de testes. Este *test suite* foi desenvolvido para cada um dos conjuntos indicados anteriormente e foi desenvolvido um *test suite* para cada um dos *test suite*. Isto permite que sempre que uma funcionalidade seja alterada se possa testar essa funcionalidade individualmente, mas se for necessário testar o sistema global isso também seja possível. Por exemplo, se uma regra do sistema de tipos for alterada, basta testar o *test suite* correspondente ao sistema de tipos. Por outro lado, se for lançada uma nova versão da linguagem é melhor executar o *test suite* que execute todos os testes da linguagem.

Os testes desenvolvidos tiveram como requisito de teste cobertura de todos os caminhos do *CFG* e caminhos primos. Para cada método testado foi desenvolvido o seu *CFG* e definiu-se um dos critérios de cobertura para desenvolver os testes. Em métodos com ciclos foi utilizado o critério de cobertura de caminhos primos e em métodos sem ciclos foi utilizado a cobertura de todos os caminhos. Testes com cobertura de todos os caminhos do *CFG*, como o nome indica, os testes vão cobrir todos os caminhos. Se esse *CFG* tiver um ciclo irá fazer com que o número de testes seja infinito, e por isso, este critério só é utilizado em métodos sem ciclos. Testes com cobertura de caminhos primos consistem em testar caminhos simples que não são subcaminhos de nenhum caminho simples. Um caminho é considerado simples, se nesse caminho um nó não aparecer mais que uma vez, a exceção é se o mesmo nó aparecer no inicio e no fim desse caminho.

Os testes efetuados de forma a validar o sistema de tipos tencionam verificar se os erros são lançados de acordo com o esperado. Tenciona também em verificar se, ao executar um programa válido, se não é lançado nenhum erro pelo sistema de tipos do compilador. Foi desenvolvido um teste individual para cada uma das classes pertencentes ao sistema de tipos. O *test suite* consiste em executar estes todos de seguida.

Os testes relativos aos algoritmos de otimização pretendem verificar se os *warnings* são lançados de acordo com o esperado. Para validar alguns algoritmos, teve de ser testado o código gerado depois de o algoritmo ser executado e comparar se o código gerado está de acordo com o esperado. Foi produzido um teste para testar individualmente cada um dos algoritmos desenvolvidos.

Para testar o gerador de código foram criados testes com cada uma das instruções permitidas na linguagem *macaw*. Para verificar se o código gerado estava de acordo com

o esperado comparou-se com esse código gerado com as funções de tradução em F. Além de estes testes também se testou o cabeçalho gerado.

O *Xtext* permite testar a *AST* gerada de modo a verificar se esta está a ser bem gerada. Mas achamos que isso não seria necessário visto que se essa *AST* fosse mal gerada os outros testes verificariam esse mesmo problema. O sistema de tipos apresentado por definição pelo *Xtext* também é possível testar.

Os testes desenvolvidos têm um total de 381 testes.

6.2 Erros encontrados

Ao executar os testes indicados na secção anterior, foram encontrados erros de desenvolvimento em todos os testes. Alguns porque o comportamento não era esperado e outros por má programação o que levavam a exceções por parte do compilador. Para todos os erros encontrados foi encontrada uma solução. Foram encontrados outros erros durante o desenvolvimento do compilador, mas estes testes ainda não tinham sido desenvolvidos.

Os erros encontrados foram os seguintes:

- Era possível fazer negação de expressões com o operador errado. Por exemplo, podia-se fazer *not(1+1)* ou *–true*;
- O comando "*ifStateContains*" estava a gerar *bytecode* duplicado. O método que gerava o código deste comando estava a ser executado mais que uma vez;
- As variáveis globais do programa podiam ser definidas com o tipo "*void*";
- A secção de "*hardware description*" podia ser identificada no ficheiro *.macawh*;
- A utilização de variáveis com o tipo *array* que tivessem sido definidas no cabeçalho dava exceção;
- O sistema de tipos ao validar os tipos do evento e do "*handler*" permitia que os tipos dos parâmetros não fossem iguais;
- O *bytecode* relativo às variáveis locais do tipo booleano não estava a ser gerado;
- Era permitido fazer a ligação entre um evento e um *handler* que tivessem um número de argumentos diferente;
- A geração do grafo de chamadas não tinha em conta os membros que eram utilizados no cabeçalho do comando "*unfolfd*" e do comando "*if*";
- Era possível definir variáveis do tipo *array* com o mesmo nome nos parâmetros de um membro;

- Os eventos predefinidos do compilador não estavam a ser carregados, o que fazia com que pudessem ser definidos eventos por parte do utilizador com este nome. Nem se conseguia fazer a ligação entre estes eventos e um "*handler*";
- O sistema de tipos não validada a comparação de valores booleanos. Ou seja, comparar o valor "*true*" ou "*false*";
- A definição de variável poderia ser usada na mesma atribuição de variável. Por exemplo, *int8 j= j + 1;*. O sistema de tipos não impedia este erro;
- O algoritmo *Dead Code Elimination* estava a eliminar funções ϕ que não devia. Por causa de este erro eliminava ciclos que não poderiam ser eliminados;
- O algoritmo *Conditional Constant Propagation* calculava expressões que usavam variáveis do tipo *array* atribuindo a este uso o valor 0;
- As funções ϕ estavam a ser mal calculadas com o algoritmo *Conditional Constant Propagation*. Este acedia mal às arestas do grafo *SSA* para obter o valor dos parâmetros das funções ϕ.

Capítulo 7

Exemplos práticos

Neste capitulo vão ser explicados alguns dos exemplos desenvolvidos com a linguagem *Macaw*. O primeiro exemplo (*blink*), permite que o pino 13 do sensor pisque cinco vezes, mas o tempo em que a luz está ligada vai diminuindo. O segundo exemplo (*blink-button-pressed*), permite novamente que o pino pisque, mas este é um exemplo mais avançado que tem em conta certos eventos.

Os exemplos de *blink* foram testados em dispositivos *Arduino* [3] e tiveram o comportamento esperado. Para verificar o seu comportamento, executou-se o *bytecode* gerado pelo compilador nestes dispositivos. O resultado da execução de estes programas consiste em verificar visualmente se o comportamento é o esperado para o *bytecode* gerado.

7.1 Exemplo *blink*

O exemplo *blink*, permite que o pino 13 de um dispositivo *Arduino* pisque cinco vezes, em que o tempo que a luz do pino 13 emite vá diminuindo ao longo do tempo. O objetivo deste programa é validar o comportamento da linguagem com o *hardware* existente no *Arduino*. Depois de piscar as cinco vezes, o *main* vai entrar no seu ciclo infinito. O ficheiro *.macawh* está representado na figura G.1 estando o ficheiro *.macaw* representado na figura G.2

O ficheiro *.macawh* para este exemplo é bastante simples. Apresenta apenas um módulo de *hardware* relativo à *board* do *Arduino*. Esta, como é óbvio, inicialmente vai estar ligada. As três ações definidas não necessitam de ter qualquer definição de *hardware*, por isso mesmo apenas têm o cabeçalho destas funções definido. A função "*set_pin_mode*", permite especificar qual o modo de um determinado pino do *Arduino*. A função "*digital_write*" permite escrever uma mensagem para um pino e a função "*delay*" permite que exista uma pausa na execução do programa.

A execução do programa, começa por se definir uma variável auxiliar e indicar qual o modo que se quer para o pino 13 do *Arduino*. O ciclo "*unfold*" vai ser executado dez vezes. O comando "*if*" permite que o tempo de duração que o pino 13 está ligado seja um

número par devido à sua condição. O corpo deste comando "*if*" consiste em ligar o pino 13 e fazer um *delay* de *count * 1000* milisegundos de modo a que o pino fique com a luz ligada durante esse tempo. Depois desse passo, desliga-se o pino 13 e faz-se um *delay* de apenas um segundo.

O comportamento visual que o *Arduino* irá ter com este programa é bastante simples, consiste em ligar a luz do pino 13 inicialmente dez segundos, depois desliga a luz durante um segundo, novamente liga a luz oito segundos e depois desliga a luz durante um segundo e por aí em diante.

7.2 Exemplo *blink-button-pressed*

O exemplo *blink-button-pressed* é um exemplo mais avançado comparativamente ao exemplo apresentado anteriormente. O *Arduino* para este exemplo, é constituído por dois *LED* (um vermelho e outro verde) em que o *LED* vermelho pisca quando o dispositivo está no estado *Idle*, mas ao carregar num botão é lançado um evento e o *LED* verde começa a piscar. Quando a execução do evento terminar, o *LED* vermelho volta a piscar ficando o *LED* verde desligado. A especificação de *hardware* está representada na figura G.4, sendo o programa representado na figura G.5. Com este exemplo, é possível identificar o comportamento da linguagem *Macaw* relativamente aos eventos.

A especificação de *hardware* define quatro módulos de *hardware*, em que dois são relativos ao *LED* vermelho ("*power_light*") e ao *LED* verde ("*button_light*") e os restantes módulos são relativos à *board* do *Arduino*. Um representa a *board* quando esta está ligada ("*mcu328p*") e outra é relativa à *board* desligada ("*mcu328Sleep*"). No início da execução do programa apenas o módulo "*mcu328p*" está ligado. As ações contêm um evento relativo ao premir o botão. Quando o botão do *hardware* for premido, vai ser laçado um evento com estas características. As funções de *hardware*, "*turnOnPowerLight*" e "*turnOnButtonLight*" permitem ligar o *LED* vermelho e o *LED* verde respetivamente, enquanto que as funções" *turnOffPowerLight*" e "*turnOffButtonLight*" permitem desligar o *LED* vermelho e o *LED* verde respetivamente. A função "*delay*", tem como objetivo parar a execução do *hardware* durante *x* tempo e por fim, a função "*sleep*", coloca o *hardware* num estado *Idle*. Todas estas ações indicam o estado em que o *hardware* tem de estar para estas serem executadas e qual o estado em que estes vão ficar posteriormente à sua execução.

O programa inícia com o "*handler*" *main* a ligar os eventos com os seus "*handler*". Posteriormente a este passo, os dois *LED* vão ser ligados simultaneamente, fazendo uma pausa de seis segundos. Depois desta pausa os dois *LED* vão ser desligados de novo. Após a pausa de um segundo se nenhum evento externo foi lançado, o evento *Idle* vai ser acionado e o *hardware* vai entrar no estado *Idle*. O "*handler*" do evento *Idle* consiste em ligar o *LED* vermelho durante seis segundos, sendo este desligado ao fim de esse tempo.

Quando o evento relativo ao pressionar o botão for acionado, o *LED* verde vai ser ligado durante seis segundos, após este tempo o *LED* vai ser desligado durante seis segundos. Estes passos vão ser repetidos durante três iterações referente ao ciclo do comando "*unfold*".

Com este exemplo, consegue-se identificar que a linguagem *Macaw* tem uma grande aplicabilidade. Este exemplo, poderia ser aplicado em grande escala a um sistema de alarme de uma escola por exemplo. Em que quando botão do *hardware* fosse premido, iria ser lançado um sinal sonoro aos alunos de modo a estes serem avisados de um perigo.

Capítulo 8

Conclusão

O compilador da linguagem *Macaw* apresenta algumas vantagens relativamente ao estado de arte das linguagens de programação para sensores. Alguns dos problemas apresentados no estado de arte, consistem em as linguagens de programação para sensores não ajudarem a diminuir os erros nos programas. Alguns dos erros só são encontrados quando os sensores já estão colocados no local de funcionamento. Outra limitação consiste em as linguagens apresentadas não serem aplicadas na prática nos sensores

O facto de a linguagem poder ser utilizada em diversos tipos de dispositivos aumenta a portabilidade e a aplicabilidade da linguagem. Esta funcionalidade permite que um programa possa ser executado em diversos dispositivos mesmo se estes tiverem *hardware* diferente, basta que a especificação de *hardware* no ficheiro *.macawh* seja adaptado a cada um dos sensores. Ao dividir o programa em dois ficheiros permite que o programador que programe a rede de sensores não necessite de ter conhecimentos avançados de *hardware*, visto bastar aceder à especificação de *hardware* que foi posteriormente desenvolvida.

Pretendeu-se com esta linguagem diminuir o maior número de erros que possam acontecer durante a execução do programa. O facto do ficheiro *pdf* conter os diversos estados em que o *hardware* pode estar, permite identificar se o *hardware* vai entrar em algum estado inválido fazendo com que esse *hardware* falhe. O compilador permite identificar em tempo de compilação o tamanho que o *bytecode* vai ocupar no sensor, de modo a diminuir os erros que possam acontecer durante a execução do programa no sensor.

Os algoritmos de otimização permitem que o *bytecode* gerado seja o mínimo possível, aumentando assim o leque de programas possíveis que se podem desenvolver nesta linguagem.

O desenvolvimento deste compilador com a plataforma *Xtext* permitiu desenvolver certas funcionalidades da *UI* que permitem ajudar os utilizadores de esta linguagem. Com o lançamento de erros e a possibilidade do *plugin* sugerir correção para os mesmos facilita no uso de esta linguagem.

Os exemplos práticos foram desenvolvidos com o objetivo de validar a linguagem *Macaw* e verificar que esta funciona e pode ser aplicada a exemplos de grande escala. Claro

que a linguagem ao ser aplicada a exemplos de grande escala é que se pode verificar a robustez da mesma. A dificuldade centrou-se mais relativamente à construção do *hardware* do que no desenvolvimento do *software*.

A linguagem *Macaw* traz assim uma maior confiança na programação para redes de sensores. Num futuro próximo estes dispositivos vão ser mais utilizados e provavelmente utilizados em situações criticas, por isso mesmo o mau funcionamento destes pode ter consequências catastróficas.

8.1 Trabalho futuro

A proposta de trabalho futuro é não só constituída por novas funcionalidades, mas também com algumas alterações necessárias na linguagem *Macaw* de modo a cumprir certos objetivos da linguagem.

Uma nova funcionalidade que se pretende adicionar ao compilador, consiste em fazer uma análise do programa a partir do grafo de estados e verificar se o uso do *hardware* está de acordo com a especificação de *hardware*. Por exemplo, ao fazer uma chamada a uma função de *hardware* é possível verificar se o estado global do *hardware* está de acordo com a cláusula "*when*" dessa função. Se o estado global do *hardware* não estiver de acordo com a cláusula "*when*" dessa função, então é lançado um erro ao utilizador de modo a este corrigir o problema. O comando "*ifStateContains*" vai ser utilizado para este algoritmo. Para o desenvolvimento desta funcionalidade é necessário desenvolver um novo sistema de tipos.

Outra funcionalidade que se pretende desenvolver consiste em utilizar o grafo de estados do programa e verificar os consumos energéticos do *hardware*. O algoritmo consiste em analisar os caminhos do grafo e verificar para todos os caminhos os consumos energéticos (consumos mínimos e máximos). Esta funcionalidade posteriormente iria avisar o utilizador que aquele programa no máximo vai consumir um x de energia e no mínimo vai consumir x de energia. Esta funcionalidade permite identificar qual a longevidade do *hardware* ao executar aquele programa.

Pretende-se aplicar a linguagem *Macaw* a um cenário real de grande escala. O objetivo consiste em facilitar a instalação e a abstração do *hardware*. Um dos exemplos práticos em que se quer aplicar a linguagem é no Sistema de Rega Inteligente [8]. O exemplo, como o nome indica, consiste num sistema de rega inteligente. O objetivo desta aplicação consiste em gerenciar um sistema de rega utilizando planos de rega definidos. Um plano de regra pode consistir, por exemplo, em regar todos os dias a uma determinada hora. Esses planos podem obter dados relativamente ao processo de rega de modo a melhorar os planos e assim consumir menos energia e água.

Ao analisar este exemplo e devido à funcionalidade de consumo energético, chegou-se à conclusão de que é necessário definir uma nova representação do *hardware*. Devido

a isto, as maiores alterações da linguagem seriam feitas no ficheiro *.macawh*. Neste momento a secção onde os módulos de *hardware* são definidos parte do principio que cada módulo de *hardware* apenas tem um estado. Mas isso em alguns casos não é verdade. Por exemplo, o módulo de *hardware GPRS*, pode ter até oito estados de *hardware* diferentes.

A nova sintaxe da linguagem está definida na secção H. Como foi referido as mudanças produzidas na sintaxe foram todas na especificação de *hardware* da linguagem. Como se pode verificar, agora um módulo de *hardware* tem uma lista de estados de *hardware*. E cada estado é definido por um nome e pelo seu consumo energético. Este consumo energético pode ser definido em três unidades diferentes (*mA*, *A* e *uA*). A secção "*boot*" tem o mesmo objetivo que a secção "*boot*" da sintaxe anterior, ou seja, definir quais os módulos de *hardware* que estão ativos no inicio da execução da aplicação. A única diferença é que é necessário indicar o módulo e o estado em que esse módulo está. Assim sendo, o *x* corresponde ao nome do módulo de *hardware* e o *y* corresponde ao estado em que esse módulo está. A definição de eventos e funções de *hardware* é semelhante à anterior. A diferença consiste em que a expressão de *hardware* deixou te ter os comandos "*on*" e "*off*".

8.2 Exemplo sistema de rega inteligente

Este sistema de rega é um bom exemplo para aplicar a linguagem *Macaw* visto ser um sistema de grande escala e permitir analisar a robustez da linguagem. Este sistema pode ser constituído por vários dispositivos e ainda uma estação central que recolhe diversas informações sobre o sistema de rega. Estas informações que vão sendo recolhidas vão ser utilizadas para otimizar os planos de rega que os sensores vão executar.

Devido à dificuldade de representação do *hardware* na primeira sintaxe proposta, este exemplo foi desenvolvido com a nova sintaxe. Este exemplo é uma primeira versão do sistema de rega, e por isso mesmo ainda não foi testado nem colocado nos dispositivos.

No ficheiro *.macawh* são definidos os módulos de *hardware*, e para cada um destes módulos quais os estados. Entre estes módulos, de realçar o módulo *GPRS* e o módulo *Solenoid*. O módulo *GPRS* permite a comunicação entre os dispositivos na rede, enquanto que o módulo *Solenoid* é utilizado para o sistema de rega dos sensores. Na secção "*boot*", como esperado, estão indicados os módulos de *hardware* que estão ligados no inicio da execução da aplicação. Na secção "*actions*", estão definidos os eventos e funções de *hardware* que vão ser utilizadas. Em comentário está descrito a qual módulo de *hardware* um conjunto de funções correspondem. Como se pode verificar pelo ficheiro *.macawh*, com a nova sintaxe é muito mais simples definir o *hardware* dos dispositivos que vão executar a aplicação.

O *main* definido no ficheiro *.macaw* faz a ligação entre cada evento do programa e o seu "*handler*". Como inicialmente o programa não tem nenhum plano de rega definido,

vai ser feita uma ligação ao servidor. Este servidor é a estação central da rede. Esta conexão é feita se o *GPRS* dos dispositivos estiver ligado, e consiste em enviar um identificador ao servidor. O membro "*sleepNow*", permite que os dispositivos fiquem em modo adormecido até receberem uma mensagem da central. Por exemplo, não é necessário que os dispositivos estejam a regar constantemente. Isto permite que a bateria dos dispositivos seja poupada. O membro "*standBy*", permite que um sensor fique num modo de pausa durante um certo tempo. Enquanto está no modo de pausa, é ligado um *LED* de forma a avisar quando começa e quando termina essa pausa. O membro "*scheduleWaterPlanNextExecution*" permite calendarizar um plano de rega para ser aplicado. Quando for a altura de aplicar esse plano de rega vai ser utilizado o membro "*runWaterPlan*". Este membro consiste em executar um plano de rega que está calendarizado. O membro "*executeAction*", consiste em produzir uma ação de acordo com uma mensagem recebida por parte da central. Uma mensagem pode indicar por exemplo, para terminar o plano de rega atual que está a ser executado. Se não existir nenhuma mensagem por parte da central para terminar o plano de rega, quando o plano de regar terminar é lançado um evento que executa o "*handler*" "*waterPlanExecutionTimeout*".

É nossa convicção que este exemplo esteja muito próximo de um programa final que fossa ser executado num Sistema de Rega Inteligente. Este programa permite que um sistema de rega fique em *stand by* até receber um plano de rega. Esse plano de rega pode ser por exemplo, carregar duas horas. Num outro dia o plano de rega pode ser apenas meia hora. Esta informação é enviada pela estação central da rede de sensores. O exemplo desenvolvido com a linguagem *Macaw* está apresentado na secção G.

Apêndice A

Sintaxe da linguagem *Macaw*

ph	$::=$	*Programa Macawh*
	r	
r	$::=$	*Recursos*
	hardware states $\vec{m}$ **boot** b **actions** a	
m	$::=$	*Estados de Hardware*
	x : **normal** `int` **max** `int`	
b	$::=$	*Expressões de Hardware*
	on x | **off** x | b **and** b | b **or** b	
a	$::=$	*Ações*
	$\vec{ae}$ $\vec{ah}$	
ah	$::=$	*Transições de estados Funções*
	t $f(\vec{t})$ **when** `int` **exec** b **turns** `int` b	
ae	$::=$	*Transições de estados Eventos*
	event $f(\vec{t})$ **when** `int` **exec** b **turns** `int` b	

Figura A.1: Sintaxe da linguagem *Macaw* – Parte I

pm ::= *Programa Macaw*

$d\ \vec{g}\ \vec{f}$

d ::= *Descrição Hardware*

hardware description $String$

f ::= *Membro*

$t\ x(t\ \vec{x})\{\ c\}$

c ::= *Corpo do Membro*

$\vec{g}\ \vec{i}$ **return** e

g ::= *Globais*

$t\ x = e\ \mid\ t\ x[\texttt{int}] = \{\vec{e}\}\ \mid\ t\ x[\texttt{int}] = \{\vec{e},\ ...\}$

i ::= *Instruções*

$h = e$

$\mid\ f(\vec{e})$

$\mid$ **attach** x **to** f

$\mid$ **unfold** (e) **times** $\{\vec{i}\}$

$\mid$ **if** $(e)\ \{\vec{i}\}$ **else** $\{\vec{i}\}$

$\mid$ **IfStateContains** $(b)\ \{\vec{i}\}$

v ::= *Valores*

$\texttt{true}\ \mid\ \texttt{false}\ \mid\ \texttt{int}\ \mid\ \texttt{float}\ \mid\ h\ \mid\ f(\vec{e})\ \mid\ nil$

h ::= *Valores Esquerdos*

$x\ \mid\ x[e]$

e ::= *Expressões*

$v\ \mid\ e + e\ \mid\ \ldots$

t ::= *Tipos*

void | **event** | **handler** | u[**int8**] | **const** u | u

u ::= *Tipos Primitivos*

boolean | **int8** | **int16** | **int32** | **float**

Figura A.2: Sintaxe da linguagem *Macaw* – Parte II

Apêndice B

Semântica operacional

$$
\begin{array}{lcll}
N & ::= & (D, T, C, Q, H) & \text{Dispositivo} \\
D & ::= & \{x_1 : (i_1, v_1), \ldots, x_n : (i_k, v_k)\} & \text{Contexto Global} \\
T & ::= & \{y_1 : (\vec{x}_1, c_1), \ldots, y_k : (\vec{x}_k, c_k)\} & \text{Código do Programa} \\
S & ::= & (c, D) & \text{Código Local} \\
C & ::= & S_1 :: \cdots :: S_n & \text{Pilha de Chamadas} \\
Q & ::= & \langle x_1, \vec{v}_1 \rangle :: \cdots :: \langle x_k, \vec{v}_k \rangle & \text{Pilha de Eventos} \\
H & ::= & \{x_1 : y_1, x_2 : y_2, \ldots, x_k : y_k\} & \text{Tabela de Despacho}
\end{array}
$$

Figura B.1: Sintaxe de ambiente de execução Macaw

BOOT

$$
\begin{array}{rl}
 & (D_0, T_0, \epsilon, \langle onBoot, \epsilon \rangle, \{onBoot : main\}) \quad \text{onde,} \\
T_0 = & \{(x : (\vec{x}, c) \mid t\ x(t\ \vec{x})\{\ c\ \} \in \vec{f}\} \quad \text{e,} \\
D_0 = & \{(x : (1, v)) \mid t\ x = e \in \vec{g} \wedge v = eval(e)\ \} \cup \\
 & \{(x : (i, v)) \mid t\ x[i] = e \in \vec{g} \wedge v = eval(e)\ \}
\end{array}
$$

Figura B.2: Estado de *boot* de um programa *Macaw*

BIND

$$(D, T, (\underline{\textbf{attach } x \textbf{ to } y}\ c, D'), Q, H) \rightarrow (D, T, (c, D'), Q, H + \{x : y\})$$

UNFOLD-OUT

$$\frac{v = eval(e) \quad v = 0}{(D, T, (\underline{\textbf{unfold } (e) \textbf{ times } \{c_1\}}\ c_2, D'), Q, H) \rightarrow (D, T, (c_2, D'), Q, H)}$$

UNFOLD-IN

$$\frac{v = eval(e) \quad v > 0}{(D, T, (\underline{\textbf{unfold } (e) \textbf{ times } \{c_1\}}\ c_2, D'), Q, H) \rightarrow (D, T, (\vec{c_1}\ \textbf{unfold } (v-1) \textbf{ times } \{c_1\}\ c_2, D'), Q, H)}$$

IF-THEN-ELSE-FALSE

$$\frac{v = eval(e) \qquad v = \textbf{false}}{(D, T, (\underline{\textbf{if } (e) \{c_1\} \textbf{ else } \{c_2\}}\ c_3, D'), Q, H) \rightarrow (D, T, (c_2\ c_3, D'), Q, H)}$$

IF-THEN-ELSE-TRUE

$$\frac{v = eval(e) \qquad v = \textbf{true}}{(D, T, (\underline{\textbf{if } (e) \{c_1\} \textbf{ else } \{c_2\}}\ c_3, D'), Q, H) \rightarrow (D, T, (c_1\ c_3, D'), Q, H)}$$

IFSTATECONTAINS-FALSE

$$\frac{v = eval(b) \qquad v = \textbf{false}}{(D, T, (\underline{\textbf{IfStateContains } (b) \{c_1\}}\ c_2, D'), Q, H) \rightarrow (D, T, (c_2, D'), Q, H)}$$

IFSTATECONTAINS-TRUE

$$\frac{v = eval(b) \qquad v = \textbf{true}}{(D, T, (\underline{\textbf{IfStateContains } (b) \{c_1\}}\ c_2, D'), Q, H) \rightarrow (D, T, (c_1\ c_2, D'), Q, H)}$$

Figura B.3: Semântica de redução de programas *Macaw* - part I

CALL

$$\frac{\vec{v} = eval(\vec{e}) \qquad T(y) = (\vec{x}, c)}{(D, T, (\underline{y(\vec{e})}\ c_1, D'), Q, H) \rightarrow (D, T, (c_1, \overline{D'}) :: (c, \{\vec{x} : \vec{v}\}), Q, H)}$$

RETURN

$$\frac{v = eval(e)}{(D, T, (c_1, D') :: (\underline{\textbf{return}\ e}\ c_2, D''), Q, H) \rightarrow (D, T, (c_1, D'), Q, H)}$$

EVENT

$$(D, T, C, Q, H) \rightarrow (D, T, C, Q :: \langle i, \vec{v} \rangle, H)$$

FIRE

$$\frac{H(i) = y \qquad T(y) = (\vec{x}, c)}{(D, T, (\epsilon, \_), \langle i, \vec{v} \rangle :: Q, H) \rightarrow (D, T, (c, \{\vec{x} : \vec{v}\}), Q, H)}$$

DECL-SIMPLE-LOCAL

$$\frac{v = eval(e) \qquad x \in dom(D')}{(D, T, (\underline{t\ x = e}\ c, D'), Q, H) \rightarrow (D, T, (c, D' + \{x : v\}), Q, H)}$$

ASSIGN-SIMPLE-LOCAL

$$\frac{v = eval(e) \qquad x \in dom(D')}{(D, T, (\underline{x = e}\ c, D'), Q, H) \rightarrow (D, T, (c, D' + \{x : v\}), Q, H)}$$

ASSIGN-SIMPLE-GLOBAL

$$\frac{v = eval(e) \qquad x \notin dom(D') \qquad x \in dom(D)}{(D, T, (\underline{x = e}\ c, D'), Q, H) \rightarrow (D + \{x : v\}, T, (c, D'), Q, H)}$$

ASSIGN-ARRAY-GLOBAL

$$\frac{v_1 = eval(e) \qquad x \notin dom(D') \qquad x \in dom(D) \qquad 0 \leq v_2 < i}{(D, T, (\underline{x[v_2] = e}\ c, D'), Q, H) \rightarrow (D + \{x : (i, [\ldots, v_2 : v_1, \ldots])\}, T, (c, D'), Q, H)}$$

Figura B.4: Semântica de redução de programas *Macaw* - part II

Apêndice C

Regras do sistema de tipos

$$\frac{\emptyset \vdash r \dashv \Gamma \qquad \Gamma \vdash \vec{g} \dashv \Gamma' \qquad \Gamma' \vdash \vec{f} \dashv \Gamma'' \qquad \Gamma'' \vdash main\colon (\epsilon \to \textbf{handler}\,)}{\vdash r\ \vec{g}\ \vec{f}} \quad \text{(T-Programa)}$$

$$\frac{\Gamma \vdash \vec{g} \dashv \Gamma' \qquad \Gamma' \vdash g' \dashv \Gamma''}{\Gamma \vdash \vec{g}, g' \dashv \Gamma''} \quad \text{(T-Globais)}$$

$$\frac{\Gamma \vdash \vec{f} \dashv \Gamma' \qquad \Gamma' \vdash f' \dashv \Gamma''}{\Gamma \vdash \vec{f}, f' \dashv \Gamma''} \quad \text{(T-Funções)}$$

Figura C.1: Sistema de tipos para um programa *Macaw* – Parte I

$$\frac{\Gamma \vdash \vec{m} \dashv \Gamma' \qquad \Gamma' \vdash_w b \qquad \Gamma' \vdash \vec{a} \dashv \Gamma''}{\Gamma \vdash \textbf{hardware states } \vec{m} \textbf{ boot } b \textbf{ actions } \vec{a} \dashv \Gamma''} \quad \text{(T-Recursos)}$$

$$\frac{\Gamma \vdash \vec{m} \dashv \Gamma' \qquad \Gamma' \vdash m' \dashv \Gamma''}{\Gamma \vdash \vec{m}, m' \dashv \Gamma''} \quad \text{(T-HardRess)}$$

$$\frac{v_1 \leq v_2}{\Gamma \vdash x \ : \ \textbf{normal } v_1 \textbf{ max } v_2 \dashv \Gamma, x\colon \texttt{mod}} \quad \text{(T-HardRes)}$$

$$\frac{\Gamma \vdash x\colon \texttt{mod}}{\Gamma \vdash \textbf{on } x} \quad \frac{\Gamma \vdash x\colon \texttt{mod}}{\Gamma \vdash \textbf{off } x} \quad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \textbf{ and } b_2} \quad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \textbf{ or } b_2} \quad \text{(T-Boot)}$$

$$\frac{\Gamma \vdash_w x\colon \texttt{mod}}{\Gamma \vdash_w \textbf{on } x} \quad \frac{\Gamma \vdash_w x\colon \texttt{mod}}{\Gamma \vdash_w \textbf{off } x} \quad \frac{\Gamma \vdash_w b_1 \quad \Gamma \vdash_w b_2}{\Gamma \vdash_w b_1 \textbf{ and } b_2} \quad \text{(T-BootSemOr)}$$

Figura C.2: Sistema de tipos para um programa *Macaw* – Parte II

$$\frac{\Gamma \vdash a\vec{h} \dashv \Gamma' \qquad \Gamma' \vdash ah' \dashv \Gamma''}{\Gamma \vdash a\vec{h}, ah' \dashv \Gamma''} \quad \text{(T-AçõesFunções)}$$

$$\frac{\Gamma \vdash a\vec{e} \dashv \Gamma' \qquad \Gamma' \vdash ae' \dashv \Gamma''}{\Gamma \vdash a\vec{e}, ae' \dashv \Gamma''} \quad \text{(T-AçõesEventos)}$$

$$\frac{\begin{array}{c} t \neq \textbf{event}, \textbf{handler}, \textbf{const}, \textbf{array} \\ \vec{t} \neq \textbf{event}, \textbf{const}, \textbf{handler}, \textbf{void} \qquad \Gamma \vdash b_1 \quad \Gamma \vdash_w b_2 \end{array}}{\Gamma \vdash t\ x(\vec{t}) \textbf{ when } b_1 \textbf{ exec } \texttt{int} \textbf{ turns } b_2 \ \dashv \Gamma, x\colon (\vec{t} \to t)} \quad \text{(T-TransiçãoFunções)}$$

$$\frac{\begin{array}{c} \vec{t} \neq \textbf{event}, \textbf{const}, \textbf{handler}, \textbf{void} \\ \Gamma \vdash b_1 \quad \Gamma \vdash_w b_2 \end{array}}{\Gamma \vdash \textbf{event } x(\vec{t}) \textbf{ when } b_1 \textbf{ exec } \texttt{int} \textbf{ turns } b_2 \ \dashv \Gamma, x\colon (\vec{t} \to t)} \quad \text{(T-TransiçãoEventos)}$$

Figura C.3: Sistema de tipos para um programa *Macaw* – Parte III

$$\Gamma \vdash u\ x \dashv \Gamma, x\colon t \qquad \text{(T-GVar)}$$

$$\frac{\emptyset \vdash v\colon u}{\Gamma \vdash u\ x = v \dashv \Gamma, x\colon u} \qquad \text{(T-GVarInit)}$$

$$\Gamma \vdash u\ x[\textbf{int8}\,] \dashv \Gamma, x\colon u[\textbf{int8}\,] \qquad \text{(T-GArrVar)}$$

$$\frac{\emptyset \vdash v_i\colon t \qquad 1 \leq i \leq n \qquad n <= maxVal(\textbf{int8}\,)}{\Gamma \vdash u\ x[\textbf{int8}\,] = \{v_1, \ldots, v_n\} \dashv \Gamma, x\colon u[\textbf{int8}\,]} \qquad \text{(T-GArrVarInit)}$$

$$\frac{\emptyset \vdash v_i\colon t \qquad 1 \leq i < n \qquad n <= maxVal(\textbf{int8}\,)}{\Gamma \vdash u\ x[n] = \{v_1, v_2,\ \ldots\} \dashv \Gamma, x\colon u[n]} \qquad \text{(T-GArrVarInitPontos)}$$

Figura C.4: Sistema de tipos para variáveis globais

$$\frac{\begin{array}{c} t_1, \ldots, t_n \neq \textbf{event}\,, \textbf{handler}\,, \textbf{void}\,, \textbf{const} \quad t_x \neq \textbf{array}\,, \textbf{event}\,, \textbf{const} \\ \Gamma, x_1\colon t_1, \ldots, x_n\colon t_n \vdash_x c \end{array}}{\Gamma \vdash t_x\ x(t_1 x_1, \ldots, t_n x_n)\{\ c\ \} \dashv \Gamma, x\colon (t_1, \ldots, t_n \to t_x)} \qquad \text{(T-Membro)}$$

$$\frac{\Gamma \vdash \vec{g} \dashv \Gamma' \qquad \Gamma' \vdash \vec{i} \dashv \Gamma'' \qquad \Gamma'' \vdash e \dashv \Gamma'''}{\vdash \vec{g}\ \vec{i}\ \textbf{return}\ e} \qquad \text{(T-CorpoMembro)}$$

$$\frac{\Gamma \vdash \vec{i} \dashv \Gamma' \qquad \Gamma' \vdash i' \dashv \Gamma''}{\Gamma \vdash \vec{i}, i' \dashv \Gamma''} \qquad \text{(T-Instruções)}$$

$$\frac{\Gamma \vdash y\colon t_1, \ldots, t_n \to t_x \qquad \Gamma \vdash e_i\colon t_i \qquad 1 \leq i \leq n \qquad t_x = \textbf{void}}{\Gamma \vdash_x y(\vec{e})\colon t_x} \qquad \text{(T-Chamada de função)}$$

$$\frac{\Gamma \vdash h\colon t \qquad \Gamma \vdash e\colon t \qquad \Gamma \vdash y\colon t_1, \ldots, t_n \to t_x \qquad \Gamma \vdash e_i\colon t_i \qquad 1 \leq i \leq n}{\Gamma \vdash_x h = y(\vec{e})} \qquad \text{(T-Chamada de função Expressão)}$$

$$\frac{\Gamma \vdash h\colon t \qquad \Gamma \vdash e\colon t}{\Gamma \vdash_x h = e} \qquad \text{(T-Assign)}$$

$$\frac{\Gamma \vdash y\colon \textbf{event}\, t_1, \ldots, t_n \qquad \Gamma \vdash z\colon \textbf{handler}\ t_1, \ldots, t_n}{\Gamma \vdash_x \textbf{attach}\ y\ \textbf{to}\ z} \qquad \text{(T-EventFire)}$$

Figura C.5: Sistema de tipos para membros e código local – Parte I

$$\frac{\Gamma \vdash v\colon \texttt{int} \qquad \Gamma \vdash \vec{i}}{\Gamma \vdash_x \textbf{unfold}\ (v)\ \textbf{times}\ \{\vec{i}\}} \quad \text{(T-Unfold)}$$

$$\frac{\Gamma \vdash e\colon \texttt{bool} \qquad \Gamma \vdash \vec{i}_1 \qquad \Gamma \vdash \vec{i}_2}{\Gamma \vdash_x \textbf{if}\ (e)\ \{\vec{i}_1\}\ \textbf{else}\ \{\vec{i}_2\}} \quad \text{(T-IfElse)}$$

$$\frac{\Gamma \vdash b\colon \texttt{bool} \qquad \Gamma \vdash \vec{i}}{\Gamma \vdash_x \textbf{IfStateContains}\ (b)\ \{\vec{i}\}} \quad \text{(T-IfStateContains)}$$

$$\frac{\Gamma(h) = t}{\Gamma \vdash h\colon t} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash x\colon \_ \to t \qquad \Gamma \vdash e\colon t}{\Gamma \vdash_x \textbf{return}\, e} \quad \text{(T-Ret)}$$

Figura C.6: Sistema de tipos para membros e código local – Parte II

Apêndice D

Sintaxe da linguagem intermédia

$$
\begin{array}{rll}
P & ::= & \textit{Programa Intermédio} \\
 & \quad H\ \vec{M}\ \vec{D}\ \vec{F}\ N & \\
H & ::= & \textit{Cabeçalho} \\
 & \quad \texttt{int int int int int} & \\
M & ::= & \textit{Módulos} \\
 & \quad \texttt{int} & \\
D & ::= & \textit{Globais} \\
 & \quad x = v \ \mid \ x[\texttt{int}] = \{\vec{v}\} & \\
F & ::= & \textit{Funções} \\
 & \quad x\{\vec{c}\} & \\
N & ::= & \textit{Mapa} \\
 & \quad \texttt{int} & \\
v & ::= & \textit{Valores} \\
 & \quad \texttt{true} \mid \texttt{false} \mid \texttt{int} \mid \texttt{float} & \\
c & ::= & \textit{Comandos} \\
 & \quad \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{rem} & \\
 & \mid \mathsf{slt} \mid \mathsf{ldg}\ n \mid \mathsf{stg}\ n \mid \mathsf{lda}\ n \mid \mathsf{sta}\ n & \\
 & \mid \mathsf{ldc}\ n \mid \mathsf{ld}\ n \mid \mathsf{st}\ n \mid \mathsf{dup} \mid \mathsf{ret} & \\
 & \mid \mathsf{call} \mid \mathsf{retv} \mid \mathsf{brd}\ n \mid \mathsf{fadd} \mid \mathsf{fsub} & \\
 & \mid \mathsf{fmul} \mid \mathsf{fdiv} \mid \mathsf{fslt} \mid \mathsf{fabs} \mid \mathsf{not} & \\
 & \mid \mathsf{and} \mid \mathsf{or} \mid \mathsf{jmp}\ n \mid \mathsf{beq8}\ n \mid \mathsf{beq16}\ n & \\
 & \mid \mathsf{beq32}\ n \mid \mathsf{bne8}\ n \mid \mathsf{bne16}\ n \mid \mathsf{bne32}\ n \mid \mathsf{hdl} & \\
 & \mid \mathsf{lnk}\ n_1\ n_2 \mid \mathsf{cast16} \mid \mathsf{cast32}\ n \mid \mathsf{castf}\ n \mid \mathsf{ldbit} &
\end{array}
$$

Figura D.1: Sintaxe da linguagem intermédia *Macaw*

Apêndice E

Formato *bytecode*

p	$::=$	*Programa*
	h	Segmento Cabeçalho
	hm	Segmento Módulos
	d	Segmento Dados
	t	Segmento de Texto (código)
	ma	Segmento Mapa
h	$::=$	*Segmento Cabeçalho*
	n_1	Tamanho total do *bytecode*
	n_2	*Offset* do segmento de dados
	n_3	*Offset* do segmento de texto
	n_4	*Offset* do segmento do mapa de *handler*
	n_5	*Offset* do segmento da Pilha (*runtime*)
	n_6	*Offset* do segmento da Fila (*runtime*)
hm	$::=$	*Segmento de Módulos*
	$\vec{m}$	Lista de Módulos
m	$::=$	*Módulos*
	n	Informação módulos
d	$::=$	*Segmento Dados*
	$\vec{v}$	Valores
t	$::=$	*Segmento de Texto*
	$\vec{f}$	Lista de Funções
f	$::=$	*Funções*
	$n_1\ n_2\ \vec{c}$	Tamanho dos dados e *bytecode*
c	$::=$	*Instruções* bytecode
	$\mid$ add $\mid$ sub $\mid$ mul $\mid$ div $\mid$ rem	
	$\mid$ slt $\mid$ ldg n $\mid$ stg n $\mid$ lda n $\mid$ sta n	
	$\mid$ ldc n $\mid$ ld n $\mid$ st n $\mid$ dup $\mid$ ret	
	$\mid$ call $\mid$ retv $\mid$ brd n $\mid$ fadd $\mid$ fsub	
	$\mid$ fmul $\mid$ fdiv $\mid$ fslt $\mid$ fabs $\mid$ not	
	$\mid$ and $\mid$ or $\mid$ jmp n $\mid$ beq8 n $\mid$ beq16 n	
	$\mid$ beq32 n $\mid$ bne8 n $\mid$ bne16 n $\mid$ bne32 n $\mid$ hdl	
	$\mid$ lnk $n_1\ n_2$ $\mid$ cast16 $\mid$ cast32 n $\mid$ castf n $\mid$ ldbit	
ma	$::=$	*Segmento Mapa*
	n	Posição do *handler main*
v	$::=$	*Valores*
	$integer$	Valores Inteiros
	$\mid$ $float$	Valores *float*
	$\mid$ $boolean$	Valores *boolean*

Figura E.1: Formato do *bytecode*

Apêndice F

Funções de tradução para *bytecode*

$$
\begin{aligned}
[\![if statecontains\ (b)\ i]\!]_f = & [\![b]\!] \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{beq8}\ n \\
& [\![i]\!]_f \\
& \text{where n = } |[\![i]\!]| + 1 \\
[\![b_1\ and\ b_2]\!] = & [\![b_1]\!] \\
& [\![b_2]\!] \\
& \mathsf{and} \\
[\![b_1\ or\ b_2]\!] = & [\![b_1]\!] \\
& [\![b_2]\!] \\
& \mathsf{or} \\
[\![on\ x]\!] = & \mathsf{ldc8}\ 0 \\
& \mathsf{ldg8}\ n \\
& \mathsf{ldc8}\ m \\
& \mathsf{ldbit} \\
& \text{where n = Table(x).getGlobalPos()} \\
& \text{and m = Table(x).getBitPos()} \\
[\![off\ x]\!] = & \mathsf{ldc8}\ 1 \\
& \mathsf{ldg8}\ n \\
& \mathsf{ldc8}\ m \\
& \mathsf{ldbit} \\
& \mathsf{sub8} \\
& \text{where n = Table(x).getGlobalPos()} \\
& \text{and m = Table(x).getBitPos()} \\
[\![g(e_1, ..., e_n)]\!]_f = & [\![e_1]\!]_{f,t_1} \\
& ... \\
& [\![e_n]\!]_{f,t_n} \\
& \mathsf{ldc8}\ m \text{ if t = HardwareEntry} \\
& \mathsf{brd}\ p \text{ if t = HardwareEntry} \\
& \mathsf{ldc16}\ p \text{ if t = MemberEntry} \\
& \mathsf{call} \text{ if t = MemberEntry} \\
& \text{where m = Table(g).getHardwareId()} \\
& \text{and p = Table(g).getGlobalPos()} \\
& \text{and t = Table(g).getType()} \\
& \text{and } t_1 \text{ = Table(g).getTypeParam(1)} \\
& \text{and } t_n \text{ = Table(g).getTypeParam(n)}
\end{aligned}
$$

Figura F.1: Funções de tradução para *bytecode* I

$[\![attach\ x\ to\ g]\!]_f$ = lnk n_1 n_2
where n_1 = Table(x).getHardwareId()
and n_2 = Table(g).getGlobalPos()

$[\![return\ e]\!]_f$ = $[\![e]\!]_{f,t}$
ret8 n; if t = int8
re16 n; if t = int16
ret32 n; if t = int32 || t = float
retv if t = handler || t = void
where t = Table(f).getType()

$[\![unfold\ (e)\ i\ times]\!]_f$ = $[\![e]\!]_{f,t}$
dup8
ldc8 0
beq8 n
$[\![i]\!]_f$
ldc8 1
sub8
jmp m
dup8
beq8 1
where t = int32
and n = 7 + $|[\![i]\!]|$, m = - n

$[\![if\ (e)\ i]\!]_f$ = $[\![e]\!]_{f,t}$
ldc8 0
beq8 n
$[\![i]\!]_f$
where t = boolean
and n = $|[\![i]\!]|$ + 1

$[\![if\ (e)\ i_1\ else\ i_2]\!]_f$ = $[\![e]\!]_{f,t}$
ldc8 0
beq8 n
$[\![i_1]\!]_f$
jmp m
$[\![i_2]\!]_f$
where t = boolean
and n = $|[\![i]\!]_1|$ + 1, m = $|[\![i]\!]_2|$

Figura F.2: Funções de tradução para *bytecode* II

$[\![x_{local} = e]\!]_f$ = $[\![e]\!]_{f,t}$
st8 n; if t = int8
st16 n; if t = int16
st32 n; if t = int32 || t = float
where n = Table(f,x).getFramePos()
and t = Table(f,x).getType()

$[\![x_{global} = e]\!]_f$ = $[\![e]\!]_{f,t}$
stg8 n; if t = int8
stg6 n; if t = int16
stg32 n; if t = int32 || t = float
where n = Table(x).getGlobalPos()
and t = Table(x).getType()

$[\![x[e_1] = e_2]\!]_f$ = $[\![e_2]\!]_{f,t}$
$[\![e_1]\!]_{f,t}$
sta8 n; if t = int8
sta6 n; if t = int16
sta32 n; if t = int32 || t = float
where n = Table(x).getGlobalPos()
and t = Table(x).getType()

$[\![g(e_1, ..., e_n)]\!]_{f,t}$ = $[\![e_1]\!]_{f,t_1}$
...
$[\![e_n]\!]_{f,t_n}$
ldc8 m if t = HardwareEntry
brd p if t = HardwareEntry
ldc16 p if t = MemberEntry
call if t = MemberEntry
$[\![t, t_g]\!]$
where m = Table(g).getHardwareId()
and p = Table(g).getGlobalPos()
and t_g = Table(g).getType()
and t_1 = Table(g).getTypeParam(1)
and t_n = Table(g).getTypeParam(n)

Figura F.3: Funções de tradução para *bytecode* III

$$
\begin{aligned}
[\![e_1 == e_2]\!]_{f,int8} = & [\![e_1]\!]_{f,int8} \\
& [\![e_2]\!]_{f,int8} \\
& \mathsf{sub8} \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{beq8}\ 6 \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{jmp}\ n \\
& \mathsf{ldc8}\ 1 \\
& \text{where n = 12 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 != e_2]\!]_{f,int8} = & [\![e_1]\!]_{f,int8} \\
& [\![e_2]\!]_{f,int8} \\
& \mathsf{sub8} \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{bne8}\ 6 \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{jmp}\ n \\
& \mathsf{ldc8}\ 1 \\
& \text{where n = 12 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 > e_2]\!]_{f,int8} = & [\![e_1]\!]_{f,int8} \\
& [\![e_2]\!]_{f,int8} \\
& \mathsf{sub8} \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{slt8} \\
& \mathsf{not}
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 \leq e_2]\!]_{f,int8} = & [\![e_1]\!]_{f,int8} \\
& [\![e_2]\!]_{f,int8} \\
& \mathsf{sub8} \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{slt8}
\end{aligned}
$$

Figura F.4: Funções de tradução para *bytecode* IV

$$
\begin{aligned}
[\![e_1 == e_2]\!]_{f,int16} = & [\![e_1]\!]_{f,int16} \\
& [\![e_2]\!]_{f,int16} \\
& \mathsf{sub16} \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast16} \\
& \mathsf{beq16}\ 6 \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast16} \\
& \mathsf{jmp}\ n \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast16} \\
& \text{where n = 15 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 != e_2]\!]_{f,int16} = & [\![e_1]\!]_{f,int16} \\
& [\![e_2]\!]_{f,int16} \\
& \mathsf{sub16} \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast16} \\
& \mathsf{bne16}\ 6 \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast16} \\
& \mathsf{jmp}\ n \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast16} \\
& \text{where n = 15 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 > e_2]\!]_{f,int16} = & [\![e_1]\!]_{f,int16} \\
& [\![e_2]\!]_{f,int16} \\
& \mathsf{sub16} \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast16} \\
& \mathsf{slt16} \\
& \mathsf{not}
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 \leq e_2]\!]_{f,int16} = & [\![e_1]\!]_{f,int16} \\
& [\![e_2]\!]_{f,int16} \\
& \mathsf{sub16} \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast16} \\
& \mathsf{slt16}
\end{aligned}
$$

Figura F.5: Funções de tradução para *bytecode* V

$$
\begin{aligned}
[\![e_1 \ == \ e_2]\!]_{f,int32} = & [\![e_1]\!]_{f,int32} \\
& [\![e_2]\!]_{f,int32} \\
& \mathsf{sub32} \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast32}\ 8 \\
& \mathsf{beq32}\ 6 \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast32}\ 8 \\
& \mathsf{jmp}\ n \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast32}\ 8 \\
& \text{where n = 18 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 \ != \ e_2]\!]_{f,int32} = & [\![e_1]\!]_{f,int32} \\
& [\![e_2]\!]_{f,int32} \\
& \mathsf{sub32} \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast32}\ 8 \\
& \mathsf{bne32}\ 6 \\
& \mathsf{ldc8}\ 0 \\
& \mathsf{cast32}\ 8 \\
& \mathsf{jmp}\ n \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast32}\ 8 \\
& \text{where n = 18 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 > e_2]\!]_{f,int32} = & [\![e_1]\!]_{f,int32} \\
& [\![e_2]\!]_{f,int32} \\
& \mathsf{sub32} \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast32}\ 8 \\
& \mathsf{slt32} \\
& \mathsf{not}
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 \leq e_2]\!]_{f,int32} = & [\![e_1]\!]_{f,int32} \\
& [\![e_2]\!]_{f,int32} \\
& \mathsf{sub32} \\
& \mathsf{ldc8}\ 1 \\
& \mathsf{cast32}\ 8 \\
& \mathsf{slt32}
\end{aligned}
$$

Figura F.6: Funções de tradução para *bytecode* VI

$$
\begin{aligned}
[\![e_1 == e_2]\!]_{f,float} = & [\![e_1]\!]_{f,float} \\
& [\![e_2]\!]_{f,floar} \\
& \text{fsub} \\
& \text{ldc8 } 0 \\
& \text{castf } 8 \\
& \text{beq32 } 6 \\
& \text{ldc8 } 0 \\
& \text{castf } 8 \\
& \text{jmp } n \\
& \text{ldc8 } 1 \\
& \text{castf } 8 \\
& \text{where n = 18 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 != e_2]\!]_{f,float} = & [\![e_1]\!]_{f,float} \\
& [\![e_2]\!]_{f,float} \\
& \text{fsub} \\
& \text{ldc8 } 0 \\
& \text{castf } 8 \\
& \text{bne32 } 6 \\
& \text{ldc8 } 0 \\
& \text{castf } 8 \\
& \text{jmp } n \\
& \text{ldc8 } 1 \\
& \text{castf } 8 \\
& \text{where n = 18 + } |[\![e]\!]_1| + |[\![e]\!]_2|
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 > e_2]\!]_{f,float} = & [\![e_1]\!]_{f,float} \\
& [\![e_2]\!]_{f,float} \\
& \text{fsub} \\
& \text{ldc8 } 1 \\
& \text{castf } 8 \\
& \text{fslt} \\
& \text{not}
\end{aligned}
$$

$$
\begin{aligned}
[\![e_1 \leq e_2]\!]_{f,float} = & [\![e_1]\!]_{f,float} \\
& [\![e_2]\!]_{f,float} \\
& \text{fsub} \\
& \text{ldc8 } 1 \\
& \text{castf } 8 \\
& \text{fslt}
\end{aligned}
$$

Figura F.7: Funções de tradução para *bytecode* VII

$$
\begin{aligned}
[\![e_1 \,\%\, e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \text{rem8 if t = int8} \\
& \text{rem16 if t = int16} \\
& \text{rem32 if t = int32 || t = float} \\
[\![e_1 < e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \text{slt8 if t = int8} \\
& \text{slt16 if t = int16} \\
& \text{slt32 if t = int32} \\
& \text{fslt if t = float} \\
[\![e_1 \geq e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \text{slt8 if t = int8} \\
& \text{slt16 if t = int16} \\
& \text{slt32 if t = int32} \\
& \text{fslt if t = float} \\
& \text{not} \\
[\![e_1 \; or \; e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \text{or} \\
[\![e_1 \; and \; e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \text{and} \\
[\![not \; e]\!]_{f,t} = & [\![e]\!]_{f,t} \\
& \text{not}
\end{aligned}
$$

Figura F.8: Funções de tradução para *bytecode* VIII

$$
\begin{aligned}
[\![e_1 + e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \mathsf{add8} \text{ if t = int8} \\
& \mathsf{add16} \text{ if t = int16} \\
& \mathsf{add32} \text{ if t = int32} \\
& \mathsf{fadd} \text{ if t = float} \\
[\![e_1 - e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \mathsf{sub8} \text{ if t = int8} \\
& \mathsf{sub16} \text{ if t = int16} \\
& \mathsf{sub32} \text{ if t = int32} \\
& \mathsf{fsub} \text{ if t = float} \\
[\![e_1 * e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \mathsf{mul8} \text{ if t = int8} \\
& \mathsf{mul16} \text{ if t = int16} \\
& \mathsf{mul32} \text{ if t = int32} \\
& \mathsf{fmul} \text{ if t = float} \\
[\![e_1 \ / \ e_2]\!]_{f,t} = & [\![e_1]\!]_{f,t} \\
& [\![e_2]\!]_{f,t} \\
& \mathsf{div8} \text{ if t = int8} \\
& \mathsf{div16} \text{ if t = int16} \\
& \mathsf{div32} \text{ if t = int32} \\
& \mathsf{fdiv} \text{ if t = float}
\end{aligned}
$$

Figura F.9: Funções de tradução para *bytecode* IX

$[\![t_1, t_2]\!]$ =cast16; if t_1 = int16 && t_2 = int8
cast32 8; if t_1 = int32 && t_2 = int8
cast32 16; if t_1 = int32 && t_2 = int16
castf 8; if t_1 = float && t_2 = int8
castf 16; if t_1 = float && t_2 = int16
castf 32; if t_1 = float && t_2 = int32

$[\![x_{local}]\!]_{f,t}$ =ld8 n; if t_1 = int8
ld16 n; if t_1 = int16
ld32 n; if t_1 = int32 || t_1 = int32
$[\![t, t_1]\!]$
where n = Table(f, x).getFramePos()
and t_1 = Table(f,x).getType()

$[\![x_{global}]\!]_{f,t}$ =ldg8 n; if t_1 = int8
ldg16 n; if t_1 = int16
ldg32 n; if t_1 = int32 || t_1 = float
where n = Table(x).getGlobalPos()
and t_1 = Table(x).getType()

$[\![x[e]]\!]_{f,t}$ =$[\![e]\!]_{f,t}$
lda8 n; if t_1 = int8
lda16 n; if t_1 = int16
lda32 n; if t_1 = int32 || t_1 = float
where n = Table(x).getGlobalPos()
and t_1 = Table(x).getType()

$[\![float]\!]_{f,t}$ =ldc32 n

$[\![int]\!]_{f,t}$ =ldc8 n; if n $\geq$ -128 && n $\leq$ 127
cast16; if t = int16 && n $\geq$ -128 && n $\leq$ 127
cast32 8; if t = int32 && n $\geq$ -128 && n $\leq$ 127
castf 8; if t = float && n $\geq$ -128 && n $\leq$ 127
ldc16 n; if n $\geq$ -32768 && n $\leq$ 32767
cast32 16; if t = int32 && n $\geq$ -32768 && n $\leq$ 32767
castf 16; if t = float && n $\geq$ -32768 && n $\leq$ 32767
ldc32 n; if n $\geq$ -2147483648 && n $\leq$ 2147483647
castf 32; if t = float && n $\geq$ -2147483648 && n $\leq$ 2147483647

$[\![false]\!]_{f,t}$ =ldc8 1

$[\![true]\!]_{f,t}$ =ldc8 0

Figura F.10: Funções de tradução para *bytecode* X

Apêndice G

Programas exemplo

```
hardware states
      BoardActive : normal 65 max 75

boot
      on BoardActive

actions
      void set_pin_mode(int8, int8);

      void digital_write(int8, int8);

      void delay(int16);
```

Figura G.1: Programa exemplo *blink.macawh*

```
hardware description "hardware/blink.macawh";

      handler main() {
            int8 count = 10;
            set_pin_mode(13, 1);
            unfold 10 times {
                  if(count % 2 == 0){
                        digital_write(13, 1);
                        delay(count * 1000);
                        digital_write(13, 0);
                        delay(1000);
                  }
                  count = count - 1;
            }
            return nil;
      }
```

Figura G.2: Programa exemplo *blink.macaw*

```
.header
      total_size: 155
      data_segment_offset: 12
      text_segment_offset: 13
      handler_map_segment_offset: 112
      stack_segment_offset: 130
      queue_segment_offset: 153

.modules
            name: BoardActive
                  state: 0          bit pos: 0  byte pos: 0
.text
      handler main at pos 2 with size param: 0 size locals: 1 {
            LDC8 0
            ST8 0
            LDC8 13
            LDC8 1
            LDC8 3
            BRD 2
            LDC8 10
            DUP8
            LDC8 0
            BEQ8 70
            LD8 0
            LDC8 2
            REM8
            LDC8 0
            SUB8
            LDC8 0
            BEQ8 6
            LDC8 0
            JMP 42
            LDC8 1
            LDC8 0
            BEQ8 31
               ...
            LD8 0
            LDC8 1
            SUB8
            ST8 0
            LDC8 1
            SUB8
            JMP 17
            DUP8
            BEQ8 1
            HDL
            CALL
            JMP 94
}
.map
      2
```

Figura G.3: Excerto de *bytecode* gerado para o exemplo *blink.macaw*

```
hardware states
      mcu328Sleep: normal 10 max 11
      power_light : normal 10 max 11
      mcu328p : normal 25 max 30
      button_light : normal 10 max 11

boot
      on mcu328p

actions
      event buttonPressed()
      when on mcu328Sleep
      turns off mcu328Sleep and on mcu328p;

      void turnOnPowerLight()
      when off power_light and on mcu328p
      turns on power_light;

      void turnOffPowerLight()
      when on mcu328p and on power_light
      turns off power_light;

      void turnOnButtonLight()
      when off button_light
      turns on button_light;

      void turnOffButtonLight()
      when on mcu328p and on button_light
      turns off button_light;

      void delay(int16)
      when on mcu328p;

      void sleep()
      when on mcu328p
      turns off mcu328p and on mcu328Sleep;
```

Figura G.4: Programa exemplo *blink-button-pressed.macawh*

```
hardware description "blink-button-pressed.macawh";

handler onIdle() {
      turnOnPowerLight();
      delay(600);
      turnOffPowerLight();
      delay(1000);
      sleep();
      return nil;
}

handler wakeUp() {
      unfold 3 times {
            turnOnButtonLight();
            delay(6000);
            turnOffButtonLight();
            delay(6000);
      }
      return nil;
}

handler main() {
      attach buttonPressed to wakeUp;
      attach Idle to onIdle;
      turnOnPowerLight();
      turnOnButtonLight();
      delay(6000);
      turnOffPowerLight();
      turnOffButtonLight();
      delay(1000);
      return nil;
}
```

Figura G.5: Programa exemplo *blink-button-pressed.macaw*

```
hardware modules
       mcu {
             active: 2,9mA
             idle: 0,6mA
             power_save: 0,3uA
             power_down: 0,2uA
       }
       transitions {
             soft_wake_up: 10uA
             hard_wake_up: 0,5mA
       }

      led_power {
             on: 20mA
             off: 0mA
             idle: 10uA
             turn_on: 0,5 mA
      }

      led_comm {
             on: 20mA
             off: 0mA
             idle: 10uA
             turn_on: 0,5 mA
      }

      gprs {
             transmit_max: 1,6A
             connected: 63mA
             active: 0,5 mA
             idle: 13mA
             sleep:0.98mA
             power_down: 29uA
             off: 0mA
             startup: 10mA
      }

      solenoid_bistable{
             on: 0
             off: 0
             switch: 100mA
      }

      button {
             idle: 0mA
      }
```

Figura G.6: Programa exemplo *rega.macawh* – Parte I

```
boot
      set mcu.active
      set led_power.on
      set led_comm.off
      set gprs.off
      set button.idle

actions
      //EVENTS
      event ON_BUTTON_PRESSED ()
      when mcu.power_down
      exec mcu.hard_wake_up
      turns mcu.active;

      event ON_GPRS_DISCONNECT(int8)
      when mcu.active and gprs.connected
      exec 100;

      event ON_DATA_READY(int8, int8[50])
      when mcu.active
      exec gprs.transmit_max;

      event ON_WATER_PLAN_EXECUTION_TIMEOUT()
      when mcu.sleep
      exec gprs.transmit_max;

      event ON_WATERING_TIMEOUT()
      when mcu.active
      exec 0;

      //MCU API
      void mcu_delay(int16)
      when mcu.active
      exec 0;

      void mcu_sleep()
      when mcu.active
      exec 0
      turns mcu.power_down;
```

Figura G.7: Programa exemplo *rega.macawh* – Parte II

```
//GPRS API
boolean gprs_start()
when mcu.active and gprs.active
exec gprs.transmit_max
turns gprs.connected;

void gprs_send_data(int8)
when mcu.active and gprs.connected
exec gprs.transmit_max;

void gprs_send_ok()
when mcu.active and gprs.connected
exec gprs.transmit_max;

void gprs_send_unknown(int8)
when mcu.active and gprs.connected
exec gprs.transmit_max;

void gprs_stop()
when mcu.active and gprs.connected
exec 0
turns gprs.active;

void gprs_enable()
when mcu.active and gprs.active
exec gprs.startup
turns gprs.active;

//POWER LED API
void power_led_turn_on()
when mcu.active and !led_power.on
exec led_power.turn_on
turns led_power.on;

void power_led_turn_off()
when mcu.active and (not led_power.off)
exec 0
turns led_power.off;

//COMM LED API
void comm_led_turn_on()
when mcu.active and (not led_comm.on)
exec led_comm.turn_on
turns led_comm.on;

void comm_led_turn_off()
when mcu.active and (not led_comm.off)
exec 0
turns led_comm.off;
```

Figura G.8: Programa exemplo *rega.macawh* – Parte III

```
//SOLENOID API
void solenoid_turn_on(int8)
when mcu.active and solenoid_bistable.off
exec solenoid_bistable.switch
turns solenoid_bistable.on;

void solenoid_turn_off(int8)
when mcu.active and solenoid_bistable.on
exec solenoid_bistable.switch
turns solenoid_bistable.off;

//TEMPERATURE SENSOR API
void temperature_read()
when mcu.active
exec 0;

//WATER MANAGER API
boolean water_manager_cycle_is_completed()
when mcu.active
exec 0;

void water_manager_fill_task(int8[2])
when mcu.active
exec 0;

void water_manager_disable_plan()
when mcu.active
exec 0;

void water_manager_enable_plan()
when mcu.active
exec 0;

void water_manager_update_plan(int8[50])
when mcu.active
exec 0;

void water_manager_time_to_next_cycle()
when mcu.active
exec 0;

//SCHEDULER API
scheduler_add(int8, int8, int8);
when mcu.active
exec 0;
```

Figura G.9: Programa exemplo *rega.macawh* – Parte IV

```
hardware description "rega.macawh";

boolean executePlan;
int8[2] waterTask;
const int8 WATER_PLAN_EXECUTION_TIMEOUT = 0;
const int8 SET_SOLENOID_ON = 1;
const int8 SET_SOLENOID_OFF = 2;
const int8 STOP_ALL_CURRENT_WATERING_NOW = 3;
const int8 GET_TEMPERATURE = 4;
const int8 SET_WATER_PLAN_OFF = 5;
const int8 SET_WATER_PLAN_ON = 6;
const int8 DISCONNECT = 7;

void connectToServer(){
      ifStateContains(gprs.active){
            gprs_enable();
            if(gprs_start())
                  gprs_send(3);
      }
      return nil;
}

handler sleepNow(){
      mcu_sleep();
      return nil;
}

handler standBy(){
      power_led_turn_on();
      ifStateContains(gprs.connected)
            comm_led_turn_on();
      mcu_delay(600);
      power_led_turn_off();
      ifStateContains(gprs.connected)
            comm_led_turn_off();
      mcu_delay(600);
      return nil;
}

void scheduleWaterPlanNextExecution(){
      scheduler_add(WATER_PLAN_EXECUTION_TIMEOUT,
         water_manager_time_to_next_cycle(), -1);
      attach ON_WATER_PLAN_EXECUTION_TIMEOUT to
         waterPlanExecutionTimeout;
      return nil;
}
```

Figura G.10: Programa exemplo *rega.macaw* – Parte I

```
void runWaterPlan(int8 solenoid_id){
      if (!water_manager_cycle_is_completed()){
            water_manager_fill_task(waterTask);
            scheduler_add(WATERING_TIMEOUT, waterTask[0],
               waterTask[1]);
            solenoid_turn_on(solenoid_id);
      } else {
            scheduleWaterPlanNextExecution();
            attach Idle to sleepNow;
      }
      return nil:
}

handler executeAction(int8 actionCode, int8[50]actionData){
      if (actionCode == SET_SOLENOID_ON){
            ifStateContains(solenoid_bistable.off)
                  solenoid_turn_on(actionData[0]);
            ifStateContains(gprs.connected)
                  gprs_send_ok();
      } else if (actionCode == SET_SOLENOID_OFF){
            ifStateContains(solenoid_bistable.on)
                  solenoid_turn_off(actionData[0]);
            ifStateContains(gprs.connected)
                  gprs_send_ok();
      } else if (actionCode == STOP_ALL_CURRENT_WATERING_NOW){
            solenoid_turn_off_all();
            ifStateContains(gprs.connected)
                  gprs_send_ok();
      } else if (actionCode == GET_TEMPERATURE){
            ifStateContains(gprs.connected)
                  gprs_send_data(temperature_read());
      } else if (actionCode == WATER_PLAN_UPDATE){
            water_manager_update_plan(actionData);
            ifStateContains(gprs.connected)
                  gprs_send_ok();
            scheduleWaterPlanNextExecution();
      } else if (actionCode == SET_WATER_PLAN_OFF){
            water_manager_disable_plan();
            ifStateContains(gprs.connected)
                  gprs_send_ok();
      } else if (actionCode == SET_WATER_PLAN_ON){
            water_manager_enable_plan();
            ifStateContains(gprs.connected)
                  gprs_send_ok();
      } else if (actionCode == DISCONNECT){
            ifStateContains(gprs.connected)
                  gprs_stop();
      } else {
            ifStateContains(gprs.connected)
                  gprs_send_unknown(actionCode);
      }
      return nil;
}
```

Figura G.11: Programa exemplo *rega.macaw* – Parte II

```
handler disconnected(int result){
      if(result != 0){
            connectToServer();
      } else {
            if(executePlan)
               runWaterPlan(result);
            else
               attach Idle to sleepNow;
      }
      return nil;
}

handler buttonPressed(){
      attach Idle to standBy;
      connectToServer();
      return nil;
}

handler waterPlanExecutionTimeout(){
      attach ON_IDLE to standBy;
      executePlan = true;
      connectToServer();
      return nil;
}

handler wateringTimeout(int8 solenoid_id){
      solenoid_turn_off(solenoid_id);
      runWaterPlan(solenoid_id);
      return nil;
}

handler main(){
      attach Idle to standBy;
      attach ON_GPRS_DISCONNECT disconnected
      attach ON_BUTTON_PRESSED to buttonPressed;
      attach ON_WATER_PLAN_EXECUTION_TIMEOUT to
         waterPlanExecutionTimeout;
      attach ON_WATERING_TIMEOUT to wateringTimeout
      attach ON_DATA_READY to executeAction;
      executePlan = false;

      connectToServer();

      return nil;
}
```

Figura G.12: Programa exemplo *rega.macaw* – Parte III

Apêndice H

Nova sintaxe da linguagem *Macaw*

ph	$::=$	*Programa Macawh*
	r	
r	$::=$	*Recursos*
	hardware modules $\vec{m}$ **boot** $\vec{b}$ **actions** a	
m	$::=$	*Definição de módulo*
	$x\{\vec{h}\}$	
h	$::=$	*Estados de Hardware*
	x : `int` s	
b	$::=$	*Módulos de Arranque*
	set $x.y$	
s	$::=$	*Unidade*
	$mA \mid A \mid uA$	
eh	$::=$	*Expressões de Hardware*
	$x.y \mid eh$ **and** $eh \mid eh$ **or** eh	
a	$::=$	*Ações*
	$\vec{ae}\ \vec{ah}$	
ah	$::=$	*Transições de estados Funções*
	$t\ f(\vec{t})$ **when** eh **exec** co **turns** eh	
ae	$::=$	*Transições de estados Eventos*
	event $f(\vec{t})$ **when** eh **exec** co **turns** eh	
co	$::=$	*Consumo*
	`int` $\mid x.y$	

Figura H.1: Proposta de nova sintaxe da linguagem *Macaw* – Parte I

pm ::= *Programa Macaw*

$d\ \vec{g}\ \vec{f}$

d ::= *Descrição Hardware*

hardware description $String$

f ::= *Membro*

$t\ x(t\ \vec{x})\{\ c\}$

c ::= *Corpo do Membro*

$\vec{g}\ \vec{i}$ **return** e

g ::= *Globais*

$t\ x = e\ \mid\ t\ x[\texttt{int}] = \{\vec{e}\}\ \mid\ t\ x[\texttt{int}] = \{\vec{e},\ ...\}$

i ::= *Instruções*

$h = e$

$\mid\ f(\vec{e})$

$\mid$ **attach** x **to** f

$\mid$ **unfold** (e) **times** $\{\vec{i}\}$

$\mid$ **if** $(e)\ \{\vec{i}\}$ **else** $\{\vec{i}\}$

$\mid$ **IfStateContains** $(eh)\ \{\vec{i}\}$

v ::= *Valores*

$\texttt{true}\ \mid\ \texttt{false}\ \mid\ \texttt{int}\ \mid\ \texttt{float}\ \mid\ h\ \mid\ f(\vec{e})\ \mid\ nil$

h ::= *Valores Esquerdos*

$x\ \mid\ x[e]$

e ::= *Expressões*

$v\ \mid\ e + e\ \mid\ \ldots$

t ::= *Tipos*

void | **event** | **handler** | u[**int8**] | **const** u | u

u ::= *Tipos Primitivos*

boolean | **int8** | **int16** | **int32** | **float**

Figura H.2: Proposta de nova sintaxe da linguagem *Macaw* – Parte II

Bibliografia

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition edition, 2007.

[2] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in java*. Addison-Wesley, 2nd edition edition, 2002.

[3] Arduino. Website of arduino. `http://www.arduino.cc/`.

[4] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. Berkeley, CA, USA, 2005.

[5] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, Agosto 2013.

[6] David Chu, Arsalan Tavakoli, and Naveen Chauhan. Recent advances and future trends in wireless sensor networks. 2006.

[7] Envisioning connections. Graphviz - graph visualization software. `http://www.graphviz.org/`.

[8] Carlos Jorge Velez Mão de Ferro. Sistema de rega inteligente. Master's thesis, Faculdade de Ciências Universidade do Lisboa, 2013.

[9] Eclipse.org. Eclipse - content assist. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Feditors_contentassist.htm`.

[10] Eclipse.org. Website of buckminster. `http://www.eclipse.org/buckminster/`.

[11] Eclipse.org. Website of eclipse. `https://www.eclipse.org/`.

[12] Eclipse.org. Website of xtext. `https://www.eclipse.org/Xtext/`.

[13] Matteo Ceriotti *et al.* Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. San Francisco, California, USA, Abril 2009.

[14] Charles N. Fischer, Ron K. Cytron, and Jr. Richard J. LeBlanc. *Crafting a Compiler*. Addison-Wesley, Novembro 2009.

[15] Chien-Liang Fok, Gruia-Catalin, Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. Columbus, 2005.

[16] RASPBERRY PI FOUNDATION. Website of raspberry pi. `http://www.raspberrypi.org/`.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and Addison-Wesley. *Design Patterns: Elements of Reusable Object-Oriented Software*.

[18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. *In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 1–11, ACM Press, 2003.

[19] Junit-team. Website of junit. `http://junit.org/`.

[20] Vivek Katiyar, Narottam Chand, Lucian Popa, and Joseph Hellerstein. Entirely declarative sensor network systems. *International Journal of applied engineering reaserch, Dindigul*, Volume 1(No 3), September 2010.

[21] Veljko Krunic, Eric Trumpler, and Richard Han. Nodemd: Diagnosing node-level faults in remote wireless sensor systems. San Juan, Puerto Rico, USA, Junho 2007.

[22] Philip Levis and *et al.* Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[23] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. *SenSys '03 Proceedings of the 1st international conference on Embedded networked sensor systems*, page 126–137, 2003.

[24] Dimitrios Lymberopoulos and Andreas Savvides. Xyz: A motion-enabled, power-aware sensor node platform for distributed sensor network applications. *In Proc. of the 4th Int. Symp. on Information Processing in Sensor Networks (IPSN)*, 2005.

[25] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 2005.

[26] Francisco Martins, Luís Lopes, and João Barros. Towards the safe programming of wireless sensor networks. 2009.

[27] Rui Miguel Ferreira Mendes. Experiments with the callas programming language and its virtual machine. Master's thesis, Faculdade de Ciências Universidade do Porto, 2012.

[28] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. Volume 43(No 13), Abril ACM Press, 2011.

[29] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. *In First International Workshop on Data Management for Sensor Networks (DMSN'04)*, 2004.

[30] Mark N.Wegman and Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181 — 210, 1991.

[31] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook — Optimizations and Machine Code Generation*. CRP Press, 2nd edition edition, Novembro 2009.

[32] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. Sydney, Australia, Novembro 2007.