Proposta e Avaliação de uma Abordagem de Desenvolvimento de *Software* Fidedigno por Construção com o Método B*

B. Dantas, D. Déharbe, S. Galvão, A. Martins Moreira, V. Medeiros Júnior

[1]Departamento de Informática e Matemática Aplicada
Programa de Pós-graduação em Sistemas e Computação
Universidade Federal do Rio Grande do Norte
Campus Universitário, Lagoa Nova
59078-970 Natal, RN, Brazil

{bartira,david,stepgalvao,anamaria,junior}@consiste.dimap.ufrn.br

Abstract. *This work describes a model-driven approach to design and develop software from the functional specification level down to assembly. The proposed approach builds upon the B method and provides a methodology to craft* assembly-level *software components in a rigorous way. While the B method is conventionally applied to produce algorithmic level software artifacts for safety-critical systems, it was not originally designed to handle the final transformations to source code and then to assembly. The users of the B method need thus to use code synthesis and compilation tools that do not offer the same rigorousness. Subtle bugs in these final steps may indeed jeopardize the whole engineering process. The approach proposed in the paper extends the B method to covers these last steps and therefore contributes to the scientific grand challenge of Computer Science proposed by Tony Hoare [6]: "The Verifying Compiler".*

Resumo. *O artigo descreve uma abordagem orientada a modelos para o desenvolvimento de componentes de software abrangendo desde a especificação funcional até a produção de código em nível de montagem. A abordagem proposta tem como base o método B e permite projetar e construir componentes de software em assembly de forma rigorosa e comprovadamente correta. Enquanto o método B é tradicionalmente aplicado pela indústria para desenvolver componentes de software para sistemas críticos até um nível algorítmico, ele não foi originalmente concebido para tratar das últimas transformações até a geração de código de montagem ou executável. A abordagem proposta nesse artigo estende o método B para cobrir essas últimas transformações e dessa forma contribui para uma das metas do Grande Desafio 5 que é o "desenvolvimento e adaptação de tecnologias e instrumentos em geral de apoio à implementação e à avaliação de software fidedigno por construção".*

1. Introdução

Um dos cinco Grandes Desafios da Computação identificados pela comunidade científica através da ação promovida pela Sociedade Brasileira de Computação [1] é a construção de sistemas corretos e seguros, uma condição *sine qua non* do "desenvolvimento tecnológico

*O trabalho apresentado recebeu apoio financeiro do CNPq, através dos projetos 485576/2007-4 e 550946/2007-1.

de qualidade". Em particular, um dos tópicos de pesquisa levantados é o projeto de sistemas fidedignos por construção: a engenharia de software é um processo complexo, onde são gerados artefatos em diferentes níveis de abstração e de complexidade e é de suma importância garantir a correspondência entre esses artefatos. Nesse contexto, destaca-se também o desafio científico lançado pelo pesquisador britânico Tony Hoare do compilador verificador [6], o qual tem sido objeto de muita atenção e entusiasmo na comunidade de engenharia de software em geral e de métodos formais em particular. Finalmente, particularmente importante nesse contexto, é a combinação do desafio da obtenção de software fidedigno por construção e com o da evolução desse software que preserve o rigor de seu desenvolvimento inicial.

Na área de métodos formais, diversas pesquisas têm sido realizadas com o intuito de atingir o objetivo do *software* fidedigno por construção. Alguns exemplos são a extração de programas a partir de provas matemáticas da satisfatibilidade da especificação [10], a obtenção de código de refinamentos sucessivos com corretude garantida a priori (cálculo de refinamentos com regras provadas corretas [4]), ou refinamentos *ad hoc* definidos pelo projetista e verificados formalmente a posteriori. O método B [2], que usamos como base para nossa proposta, se encaixa nessa última linha de ação. B e suas ferramentas possuem diversas características adequadas ao tratamento dos desafios propostos e algumas limitações que precisam ser superadas para que esses objetivos sejam efetivamente atingidos. Esse artigo procura, ao mesmo tempo em que apresenta uma proposta de adaptação do método para a obtenção de maior garantia de fidedignidade, mostrar como diversos princípios do método são importantes para a superação dos desafios considerados.

A confiabilidade do resultado de um processo de desenvolvimento de *software* depende, no entanto da confiabilidade de cada etapa. O método B [2], inspirado em técnicas de especificação como VDM [9] e Z [15], e na teoria do refinamento [3], proporciona uma abordagem rigorosa que abrange o desenvolvimento desde a modelagem funcional até o nível algorítmico. Caso todas as verificações pertinentes ao método sejam efetuadas, garante-se que o modelo algorítmico obtido é fidedigno ao modelo abstrato que ele refina. Mas, os últimos passos para uma implementação executável são: a síntese de código em uma linguagem de programação e a compilação desse código para a plataforma computacional alvo. Essas duas últimas transformações não podem ser verificadas com o mesmo nível de rigor que o proporcionado pelo método B: de um lado a síntese de código efetua uma tradução entre linguagens que não possuem a mesma base semântica, e de outro lado, os compiladores, implementados de forma *ad hoc*, efetuam transformações significativas na estrutura do código. Assim, os esforços empreendidos na aplicação do método B podem ser desperdiçados por um defeito em uma dessas etapas finais.

Uma solução possível para esse problema é construir casos de teste a partir do modelo funcional inicial construído no método B [8]. Esses testes podem então ser aplicados à implementação resultante e assim ser verificada a sua compatibilidade com o modelo inicial. Porém, em geral, essa abordagem não oferece garantias de completude.

Uma nova abordagem, proposta em [5], usa técnicas existentes de forma inovadora para estender o alcance do método B até o nível de linguagem de montagem. Dessa forma, são eliminadas as duas etapas de síntese e compilação identificadas como potencialmente menos confiáveis do ponto de vista da correção do resultado. Nessa nova abor-

dagem, são geradas obrigações de prova que, se verificadas, garantem que o programa em nível de montagem implementa o modelo funcional inicial. Assim, a produção de código executável apenas requer uma tradução trivial das instruções de montagem para seus correspondentes binários. Em [5], foi investigada a compilação formal das principais construções algorítmicas (atribuição, sequência, condicional e laço) para instruções da *máquina de acesso randômico* [7], um modelo computacionalmente completo cujo funcionamento se assimila ao dos microcontroladores e microprocessadores modernos.

Esse artigo prossegue nessa direção, propondo agora investigar a aplicabilidade da abordagem no contexto de uma plataforma computacional industrial, no caso, o microcontrolador PIC16C432, um equipamento de baixo custo que tem sido usado para executar *software* embarcado em aplicações críticas. Por um lado, o conjunto de instruções do PIC16C432 foi modelado em B. Por outro lado, realizou-se o desenvolvimento, utilizando o método B, de um sistema reativo simples (um semáforo) até o nível de abstração de linguagem de programação imperativa. A partir desse artefato, foi construído um programa de montagem PIC16C432 correspondente. Esse programa foi codificado em B, utilizando o modelo de instruções construído, e a sua conformidade com o modelo funcional inicial foi provada em ferramentas de suporte ao método B.

Estrutura do artigo. A seção 2 fornece informações acerca do método B necessárias ao entendimento da abordagem e do exemplo proposto. A seção 3 provê uma visão geral da abordagem proposta para construção de *software* correto por construção. Na seção 4.1, é apresentado a modelagem do conjunto de instruções do PIC16C432. A seção 5 apresenta o estudo de caso do semáforo, desde o modelo funcional incial até a implementação em nível de montagem. As conclusões são apresentadas na seção 6.

2. O método B

O método B[2, 13] oferece uma abordagem orientada a modelos para o desenvolvimento de componentes de software. Possui uma linguagem própria, chamada de Notação de Maquinas Abstratas (*Abstract Machine Notation* - AMN), que permite tanto expressar um modelo funcional de alto nível, quanto os refinamentos sucessivos desse modelo. Esses refinamentos compõem os passos realizados até que se chegue a um modelo suficientemente concreto, a partir do qual o código fonte do componente pode ser gerado automaticamente. A base matemática do método B (lógica de primeira ordem, aritmética inteira e teoria dos conjuntos) proporciona uma grande similaridade com a notação Z[15]. Entretanto, com a intenção de ser facilmente compreendido e utilizado fora do mundo acadêmico, o método B possui uma estruturação rigorosa e estritamente relacionada às construções das linguagens de programação imperativa.

Existem, no mercado, diversas ferramentas de apoio ao desenvolvimento de *software* implementando o método B. Essas ferramentas dão suporte às tarefas de especificação, animação, verificação, síntese e gerenciamento de projetos.

2.1. Etapas do método B

Uma especificação B é estruturada em módulos. Um módulo define um conjunto de estados válidos, um subconjunto desses estados que são os estados iniciais possíveis, e

operações que podem ocasionar uma transição entre estados. O processo de desenvolvimento inicia com um módulo que define o modelo funcional de alto nível do sistema. Em B, esse modelo inicial é chamado de "máquina" (MACHINE). Nessa fase de modelagem pode-se utilizar técnicas semi-formais tais como UML que permitem a transição do documento em linguagem natural para a notação formal do método B [14, 11]. O método B impõe que seja provado que, em uma máquina, todos os estados iniciais são válidos, e que as operações não definem transições de um estado válido para um estado inválido.

Uma vez estabelecido o modelo funcional inicial, o método B propicia construções para definir "refinamentos" (REFINEMENT). Em B, um refinamento é sempre associado a outro modelo (mais abstrato), e especifica uma decisão de projeto, ora sobre a representação concreta do estado, ora sobre a operacionalização de uma operação por um algoritmo. O método B requer que, quando é realizado um refinamento, seja provada a sua conformidade com o módulo refinado.

Existe um tipo especial de refinamento, chamado de "implementação" (IMPLEMENTATION), que é usado quando o nível de abstração é equivalente ao de uma linguagem de programação. Esse tipo de módulo é classificado como *modelo algorítmico* nesse artigo. Implementações podem ser escritas em um subconjunto da AMN chamado de B0 e que restringe a definição a construções algorítmicas sequenciais e determinísticas. A partir de um módulo implementação, existe a possibilidade de gerar código fonte em linguagens de programação como C ou ADA.

As diversas etapas do método B são ilustradas graficamente na figura 1, com as seguintes convenções: retângulos representam etapas; retângulos inclinados correspondem ao processos humanos e computacionais; retângulos cinzas são etapas geradas pelo método B; rótulos V enfatizam que a verificação formal é aplicada para a etapa correspondente; a área cinza claro é o foco desse trabalho.

A garantia de fidedignidade provida pelo método é resultado das provas realizadas em cada etapa (consistência do modelo, conformidade dos refinamentos). As etapas finais de geração de código em linguagem de programação e de compilação devem ser verificadas com o uso de dados de teste gerados a partir do modelo funcional inicial. Essa verificação não é completa, pois, como é de conhecimento geral, embora os testes possam revelar a presença de erros, não podem garantir sua ausência. É portanto importante procurar diminuir a dependência dessa fase de testes, onde entra nossa proposta de extensão do método B ao nível de linguagem de montagem.

Por outro lado, as ferramentas de apoio ao método B procuram contribuir para o quesito evolutibilidade, através da preocupação com a gestão modular de projetos e de suas provas, de maneira a que provas possam por exemplo ser re-aproveitadas ou re-aplicadas quando da evolução da especificação. Assim, apenas as partes da especificação que foram efetivamente alteradas necessitarão de um novo esforço de verificação. No entanto, o esforço relacionado à evolução ainda é grande dada a necessidade de realização/adaptação manual dos refinamentos. Essa é certamente uma linha de trabalho essencial para transformar o método em uma solução para o desafio do desenvolvimento rigoroso fidedigno e evolutivo.

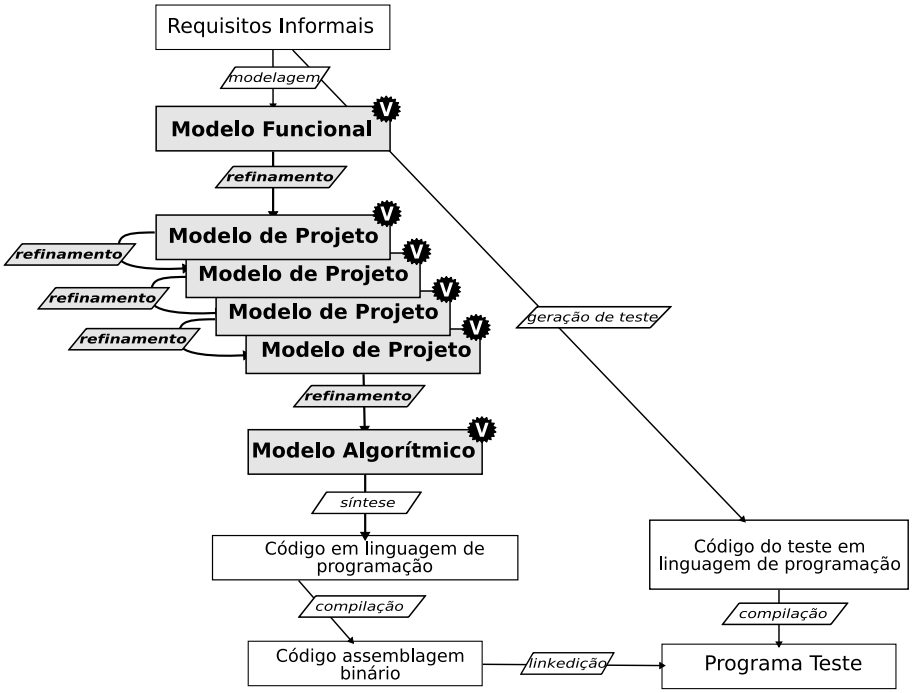

Figura 1. Visão geral de um processo de engenharia de software baseado no método B

2.2. A notação B

Na sua essência, um módulo B é composto por duas partes principais: a definição do estado e das operações. Além desses elementos essenciais, ainda há cláusulas auxiliares como parâmetros de módulo, constantes, asserções, etc. que propiciam maior reusabilidade e modularidade, embora não estendam estritamente o poder expressivo da notação. Nós discutiremos aqui estritamente o cerne das cláusulas de um módulo B.

A definição dos estados dos componentes é realizada através das cláusulas VARIABLES e INVARIANT. A primeira enumera os componentes de estados; já a segunda, restringe as possibilidades de valores que eles podem assumir.

Para a especificação da inicialização, assim como para as operações, B oferece um conjunto de construções denominadas *substituições generalizadas*. Algumas dessas construções se assemelham a construções de linguagens de programação imperativa, e outras são mais abstratas, permitindo, por exemplo, não determinismo. A semântica é definida através do *cálculo de substituições*, um conjunto de regras que define como as diferentes substituições reescrevem fórmulas da lógica de primeira ordem. Como notação, temos que $[S]E$ denota o resultado da aplicação da substituição S a uma expressão E. Por exemplo, a operação que incrementa a variável v pode ser definida através da substituição simples $v := v + 1$. Assim, $[v := v + 1]v < 0$ é $v + 1 < 0$.

A notação B fornece também construções mais elaboradas. A *substituição não determinística* $\mathbf{ANY}\ v\ \mathbf{WHERE}\ C\ \mathbf{THEN}\ S\ \mathbf{END}$ aplica a substituição S com a variável v assumindo qualquer valor que satisfaça a condição C. A substituição $v :\in V$, onde V é um conjunto, é equivalente a $\mathbf{ANY}\ x\ \mathbf{WHERE}\ x \in V\ \mathbf{THEN}\ v := x\ \mathbf{END}$. A *substituição paralela* $[S \parallel S']$ aplica as duas substituições S e S' simultaneamente. A

$$
\begin{array}{ll}
\textbf{MACHINE}\ \mathit{traffic\_light} & \mathit{advance} = \\
\textbf{SETS}\ \mathit{COLOR} = \{\mathit{green}, \mathit{yellow}, \mathit{red}\} & \quad \textbf{CASE}\ \mathit{color}\ \textbf{OF} \\
\textbf{VARIABLES}\ \mathit{color} & \quad \textbf{EITHER}\ \mathit{green}\ \textbf{THEN}\ \mathit{color} := \mathit{yellow} \\
\textbf{INVARIANT}\ \mathit{color} \in \mathit{COLOR} & \quad \textbf{OR}\ \mathit{yellow}\ \textbf{THEN}\ \mathit{color} := \mathit{red} \\
\textbf{INITIALISATION}\ \mathit{color} :\in \mathit{COLOR} & \quad \textbf{OR}\ \mathit{red}\ \textbf{THEN}\ \mathit{color} := \mathit{green} \\
\textbf{OPERATIONS} &
\end{array}
$$

Figura 2. Exemplo de modelo funcional em B

substituição com pré-condição $\textbf{PRE}\, C\, \textbf{THEN}\, S\, \textbf{END}$ é usada para especificar uma operação com a pré-condição de aplicação C. Por exemplo, a operação parcial que incrementa v até o valor *top* pode ser especificada como $\textbf{PRE}\, v < \mathit{top}\, \textbf{THEN}\, v := v + 1\, \textbf{END}$.

2.3. Exemplo de modelo funcional

O exemplo da figura 2 mostra as cláusulas mais básicas para a construção de um modelo funcional de um semáforo[1]. O nome do modelo é $\mathit{traffic\_light}$, sendo definido na cláusula MACHINE. Um conjunto COLOR é definido, composto pelas três cores possíveis do semáforo. O estado é composto por uma única variável, chamada color, cujo valor deve pertencer a COLOR e inicializada não deterministicamente com um dos elementos desse conjunto. Em seguida, é especificada a operação *advance*, que opera uma transição do semáforo.

2.4. Exemplo de refinamento

O modelo anterior pode ser refinado por um módulo que possui como estado um único valor inteiro. A figura 3 mostra um refinamento B da máquina $\mathit{traffic\_light}$. O nome do refinamento é $\mathit{traffic\_light\_data\_refinement}$ e o modelo refinado é especificada na cláusula REFINES. A cláusula CONSTANTS declara duas constantes funcionais cujas definições são dadas na cláusula PROPERTIES. A cláusula VARIABLES declara o único componente do estado do refinamento. A cláusula INVARIANT estabelece a relação entre o estado do refinamento e o estado do módulo refinado: em qualquer momento, o valor de count deve ser igual ao resultado da aplicação da função $\mathit{color\_refine}$ à variável abstrata color. O refinamento do estado inicial é especificado na cláusula INITIALISATION: é realizada uma atribuição do valor 0 à variável `count`. O refinamento da operação é então realizado na seção OPERATIONS. A seção 5 provê um exemplo adicional de refinamento, que resulta em uma implementação, ou modelo algorítmico, do semáforo.

2.5. Obrigações de prova

No método B, para garantir a correção do desenvolvimento, deve-se verificar que cada modelo funcional é coerente, e que cada refinamento é consistente com o modelo que refina. Para isso, devem ser gerados e verificados diferentes tipos de obrigações de prova, detalhadas a seguir.

[1]Por limitação de espaço, os BEGINs e ENDs da sintaxe de B foram omitidos aqui.

REFINEMENT $traffic\_light\_data\_refinement$
REFINES $traffic\_light$
CONSTANTS $color\_refine, color\_step$
PROPERTIES
$color\_refine \in COLOR \longrightarrow NATURAL \wedge$
$color\_refine = \{green \mapsto 0, yellow \mapsto 1, red \mapsto 2\} \wedge$
$color\_step \in 0..2 \longrightarrow 0..2 \wedge color\_step = \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0\}$
VARIABLES $count$
INVARIANT $count \in NATURAL \wedge count \in 0..2 \wedge count = color\_refine(color)$
INITIALISATION $count := 0$
OPERATIONS $advance = count := color\_step(count)$

Figura 3. Exemplo de refinamento em B

A coerência de um modelo funcional é estabelecida quando as ações de inicialização colocam a máquina em um estado válido e quando nenhuma operação, quando aplicada dentro do seu domínio, pode levar a máquina de um estado válido para um estado inválido. Assim, a substituição da inicialização S estabelece o invariante, ou seja $[S]INV$, e, para cada operação com pré-condição PRE e substituição S, deve se verificar que a seguinte fórmula é válida: $PRE \wedge INV \Rightarrow [S]INV$.

Considerando o modelo da figura 2, um exemplo de obrigação de prova (inicialização) é:

$$\begin{aligned} & [color :\in COLOR] color \in COLOR \\ \equiv \; & \forall x \bullet (x \in COLOR \Rightarrow [color := x] color \in COLOR) \\ \equiv \; & \forall x \bullet (x \in COLOR \Rightarrow x \in COLOR) \end{aligned}$$

No caso de um refinamento, seja INV_R o invariante do refinamento, INV_M o invariante do modelo refinado, a consistência de um refinamento com relação ao modelo que refina é garantida quando:

- A inicialização do refinamento, denotada $INIT_R$, deve garantir que, todo estado inicial concreto refina algum estado inicial abstrato. Se $INIT_M$ denota a inicialização do modelo refinado, essa propriedade é expressa da seguinte forma no cálculo de substituições:

$$[INIT_R]\neg[INIT_M]\neg INV_R.$$

- Para as operações, três propriedades devem ser garantidas: (1) a operação do refinamento OP_R deve ser aplicável sempre que a operação abstrata OP_M o for, logo a sua pré-condição PRE_R deve ser mais fraca que a pré-condição PRE_M da operação abstrata; (2) toda alteração do estado concreto corresponde a alguma transição do modelo abstrato; (3) para entradas iguais, as saídas são compatíveis. O cálculo de substituições provê a seguinte formalização dessa propriedade:

$$INV_M \wedge INV_R \wedge PRE_M \Rightarrow PRE_R \wedge [OP_R]\neg[OP_M]\neg INV_R.$$

Dois tipos de ferramentas são empregadas para realizar a verificação dos módulos B. O *gerador* das obrigações de prova de um módulo, que é totalmente automático, aplica as regras do cálculo de substituições e eventuais simplicações. Os *provadores* que são semi-automáticos. Geralmente, boa parte das obrigações de prova é simples o suficiente para ser provada sem intervenção humana. Para as demais obrigações de prova, o usuário pode interagir com o verificador para selecionar as hipóteses relevantes, instanciar fórmulas quantificadas, realizar simplificações, e outras operações que permitem, ora provar a obrigação de prova, ora descobrir que a obrigação de prova não é válida e que há uma falha no módulo que deve então ser corrigido.

3. Visão geral da abordagem proposta

O *elo fraco* da produção de componentes de *software* com o método B é a síntese de software em uma linguagem de programação e sua compilação em linguagem de montagem da plataforma. Como a linguagem B0 (ver seção 2.1) é próxima as construções de programação, a síntese de código geralmente é considerada segura; entretanto, se a linguagem alvo usa construções não suportadas pela linguagem B0 (por exemplo, orientação a objetos), essa transformação pode não ser tão simples.

Este trabalho propõe aplicar os conceitos do método B para gerar artefatos de software em nível de montagem. A abordagem é dividida em: (1) modelagem da plataforma alvo, e (2) refinamento do modelo algorítmico para uma implementação baseada no modelo da plataforma.

A plataforma alvo pode ser modelada com a notação de máquina abstrata de B: o estado da máquina representa o estado da plataforma (isto é, registradores e memória), e cada operação representa uma instrução de montagem. É necessário fazer isso uma única vez para uma determinada plataforma. Detalhes adicionais são fornecidos na seção 4.

O modelo algorítmico tem que ser refinado para o modelo a nível de montagem. Esse último modelo é definido sobre o modelo de plataforma alvo discutido anteriormente. Uma estratégia geral desse refinamento é mapear as variáveis de estado do modelo algorítmico para diferentes endereços de memória da plataforma, e traduzir as operações a nível algorítmico para combinações de operações definidas no modelo de plataforma correspondente às instruções de linguagem de montagem. O refinamento resultante leva à geração e verificação das obrigações de prova correspondentes. Nós então obtemos um artefato de *software* no nível de montagem que é comprovadamente compatível com o modelo fucional inicial.

Essa abordagem fornece uma extensão do método B como mostrado à esquerda da Figura 4. Contudo, o método B clássico tem algumas limitações que não nos permite aplicar essa então chamada “estratégia” ideal, a saber: primeiro, construções de repetição em B podem ser usadas somente em um módulo de implementação; e, segundo, um módulo de implementação não pode ser refinado. Portanto o método B não fornece subsídios para construir o refinamento de um algoritmo que utiliza construções de repetição. Uma solução seria remover essa limitação do método B, mas iria requerer a atualização de ferramentas de terceiros.

Felizmente, é possível elaborar outra solução para contornar essa limitação sem modificar o método B. Essa solução é mostrada à direita da Figura 4: ao invés de estabelecer uma relação de refinamento entre os modelos de montagem e o algorítmico,

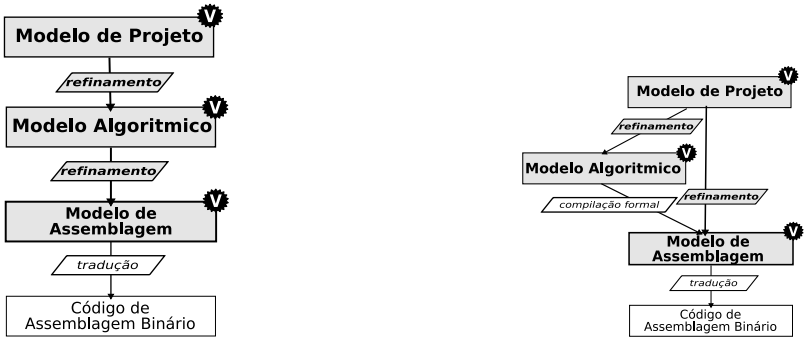

Figura 4. Aplicando o método B até o nível de montagem: ideal (esquerda) e atual (direita).

consideraremos um refinamento do modelo de projeto imediatamente anterior ao modelo algorítmico no processo de refinamento. A construção da implementação da montagem a partir da implementação algorítmica poderá ser formalizada por um conjunto de regras que podem ser implementadas para construir um compilador formal baseado em B, mas a verificação pode ser feita normalmente em um refinamento B, com respeito ao modelo de projeto. A seção 5 provê um exemplo simples, demostrando a correspondência lógica entre as implementações algorítmica e de montagem.

4. Modelagem de uma plataforma computacional

Nesse trabalho, a plataforma computacional modelada é a do PIC16C432, um microcontrolador com um barramento de 8 bits, 35 instruções e 8 níveis de pilha em hardware. Esse artefato, bastante simples, tem um custo muito baixo, pode ser utilizado para executar *software* embarcado em diversos tipos de aplicações e tem uma boa difusão no mercado de microcontroladores para sistemas embarcados.

A modelagem da plataforma PIC é estruturada em diferentes módulos [2]:

- O módulo PIC contém a especificação do estado do microcontrolador e do seu conjunto de instruções. Informações detalhadas são apresentadas na seção 4.1.
- O módulo ALU provê as definições das diferentes funções lógicas e aritméticas que são usadas na especificação das instruções da PIC.
- O módulo $TYPES$ fornece as definições dos diferentes tipos de dados que são usados na plataforma PIC: os valores possíveis de uma palavra de dados, de um endereço de memória, etc.

4.1. O modelo funcional do microcontrolador PIC16C432

O modelo PIC é apresentado com a sua modelagem em B intercalada com comentários informais. Os módulos contendo as definições auxiliares usadas na especificação do estado e das operações são importados através da cláusula **SEES**:

MACHINE PIC
SEES $ALU, TYPES$

[2] O leitor interessado nos detalhes da especificação completa ou em outros estudos de caso é convidado a visitar o repositório dos autores no endereço `http://b2asm.googlecode.com`.

A semântica do conjunto de instruções da plataforma pode ser definida com base na alteração do valor dos seguintes componentes, que formam o estado do modelo da PIC: o registro de trabalho w, o bit de teste de nulidade z, o bit de vai-um c, o contador de programa pc, o ponteiro de pilha sp, a pilha de execução $stack$ e a memória de dados mem:

VARIABLES $w, z, c, pc, sp, stack, mem$
INVARIANT
$w \in WORD \wedge z \in BOOL \wedge c \in BOOL \wedge$
$mem \in REGISTER \rightarrow WORD \wedge pc \in INSTRUCTION \wedge$
$stack \in \mathbb{N} \rightarrow INSTRUCTION \wedge sp \in \mathbb{N} \wedge \text{dom}(stack) = 0..(sp-1)$

Nessa definição, $WORD$, $REGISTER$ e $INSTRUCTION$ representam respectivamente as palavras de dados, os endereços da memória de dados e os endereços da memória de instruções. Suas definições encontram-se no módulo $TYPES$.

Cada instrução de montagem é modelada na forma de uma operação no modelo B. São classificadas em: cópia de dados, operações lógicas e aritméticas, operações sobre bits e de alteração do fluxo de execução. Essa apresentação se limita a um subconjunto representativo das instruções. A operação $MOVWF$ modela a instrução que copia a palavra armazenada no registrador de trabalho para um dado endereço na memória de dados. A operação $MOVLW$ (omitida aqui) modela a cópia de um valor dado para o registrador de trabalho w. Ambas incrementam o contador de programa.

$MOVWF(f) =$
PRE $f \in REGISTER$ **THEN**
$mem(f) := w \parallel pc := INSTRUCTION\_NEXT(pc)$

O conjunto de instruções possui operações aritméticas e lógicas de soma, subtração, conjunção, disjunção e disjunção exclusiva, todas são binárias e com especificações similares. Cada operação possui duas versões. A primeira modela a instrução com um único argumento, uma palavra de dados k, que combina k com o registro de trabalho w, e guarda o valor em w. A segunda modela a instrução com dois argumentos f e d: f é um endereço da memória de dados, cujo conteúdo é combinado com o registro de trabalho w. O argumento d é um bit que indica se o resultado deve ser guardado no registro de trabalho ou no endereço f. A modelagem da segunda forma da instrução de adição é mostrada a seguir.

$ADDWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
ANY $result, carry, zero$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
$result, carry, zero = add(mem(f), w)$
THEN
IF $d = 0$ **THEN** $w := result$
ELSE $mem(f) := result$
END $\parallel$
$c := carry \parallel z := zero$
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$

Em ambas versões, os bits z e c são atribuídos para indicar respectivamente se o resultado da combinação foi nulo e se houve estouro (vai-um) . No modelo, a combinação

é realizada através de uma função and, cuja definição é dada no módulo ALU (detalhes são fornecidos na seção 4.2).

Há ainda operações que modelam as instruções que alteram o fluxo de execução. A operação $GOTO$ modela a instrução de salto incondicional e é simplesmente definida da seguinte forma: As operações $CALL$ e $RETURN$ modelam respectivametne as instruções de chamada e de retorno de uma rotina. As operações B correspondentes especificam como o estado da pilha e do contador de programa são alterados por essas instruções. Há ainda instruções de teste para efetuar desvios condicionais. A operação $BTFSC$ modela tal instrução com dois parâmetros: um endereço de memória f, e uma posição b. Se o b-ésimo bit da palavra no endereço f for zero, não executa a instrução seguinte e sim a posterior. Essa definição utiliza a função auxiliar $bitget$, especificada no módulo ALU, apresentado a seguir.

$$
\begin{array}{l}
GOTO(k) = \\
\quad \textbf{PRE}\ k \in INSTRUCTION\ \textbf{THEN}\ pc := k \\
CALL(k) = \\
\quad \textbf{PRE}\ k \in INSTRUCTION\ \textbf{THEN} \\
\quad\quad stack(sp) := INSTRUCTION\_NEXT(pc) \parallel sp := sp + 1 \parallel pc := k \\
RETURN = \\
\quad \textbf{PRE}\ sp > 0\ \textbf{THEN} \\
\quad\quad pc := stack(sp - 1) \parallel stack := 0..(sp - 2) \lhd stack \parallel sp := sp - 1 \\
BTFSC(f, b) = \\
\quad \textbf{PRE}\ f \in REGISTER \land b \in WORD\_POSITION\ \textbf{THEN} \\
\quad\quad \textbf{IF}\ bitget(mem(f), b) = 0\ \textbf{THEN} \\
\quad\quad\quad pc := INSTRUCTION\_NEXT(INSTRUCTION\_NEXT(pc)) \\
\quad\quad \textbf{ELSE} \\
\quad\quad\quad pc := INSTRUCTION\_NEXT(pc)
\end{array}
$$

4.2. O módulo ALU

O módulo ALU define diversas funções matemáticas que são implementadas no microcontrolador PIC16C432. Utiliza o módulo $TYPES$ que contém as definições dos tipos dos parâmetros e resultados dessas funções. Assim, o módulo ALU possui as constantes add e $bitget$, que possuem tipo e valor funcionais; elas são definidas na cláusula $PROPERTIES$ do módulo:

$$
\begin{array}{l}
add \in (WORD \times WORD) \longrightarrow (WORD \times BOOL \times BOOL) \land \\
\forall w_1, w_2, s \bullet \\
\quad (w_1 \in WORD \land w_2 \in WORD \land s \in NATURAL \land s = w_1 + w_2 \Rightarrow \\
\quad\quad ((s \leq 255 \Rightarrow add(w_1, w_2) = (s, FALSE, bool(s = 0))) \\
\quad\quad \land (256 \geq s \Rightarrow add(w_1, w_2) = (s - 256, TRUE, bool(s = 256))))) \land \\
\\
bitget \in WORD \times WORD\_POSITION \longrightarrow BIT \land \\
\forall w, i \bullet (w \in WORD \land i \in WORD\_POSITION \Rightarrow bitget(w, i) = WORD\_TO\_BV(w)(i))
\end{array}
$$

5. Estudo de caso

Nesta seção, o modelo e o refinamento do semáforo, utilizados nas seções 2.3 e 2.4 são objetos de novos refinamentos. Esses refinamentos resultam em uma implementação B tradicional, ou seja um modelo algorítmico, e a uma implementação em nível de montagem da plataforma PIC16C432, utilizando o modelo formal apresentado na seção 4.1.

Primeiro, é apresentada uma implementação do semáforo utilizando os conceitos tradicionais do método B. A variável de refinamento $count$ é implementada com uma instância, chamada $state$, do módulo nat que modela o armazenamento de um valor do tipo natural.

IMPLEMENTATION $traffic\_light\_alg$
REFINES $traffic\_light\_data\_refinement$
IMPORTS $state.nat$
INVARIANT $state.value \in 0..2 \wedge state.value = count$
INITIALISATION $state.set(0)$
OPERATIONS
$advance =$
  IF $state.value = 0$ **THEN** $state.set(1)$
  ELSIF $state.value = 1$ **THEN** $state.set(2)$
  ELSE $state.set(0)$ **END**

A implementação em nível de montagem é mais longa e é apresentada em partes. O estado é composto por uma instância do modelo PIC, de nome m. O invariante estabelece a correspondência entre o estado do módulo refinado (figura 3) com o dessa instância: o valor de $count$ é guardado no endereço 0 da memória de dados. Uma vez definido o mapeamento das variáveis do nível algorítmico para a memória do microcontrolador, o invariante da implementação em linguagem de montagem é igual à substituição das variáveis pelos seus locais de memória correspondentes, no invariante da implementação algorítmica. Assim, a variável $state.value$ é substituída por $m.mem(0)$.

IMPLEMENTATION $traffic\_light\_asm\_pic$
REFINES $traffic\_light\_data\_refinement$
IMPORTS $m.PIC$
INVARIANT $m.mem(0) \in 0..2 \wedge m.mem(0) = count$

A inicialização é implementada com a seguinte sequência de instruções, as quais respectivamente atribuem o valor 0 ao registro de trabalho w e copiam o valor do registro de trabalho para o endereço 0 da memória de dados. A prova de consistência desse trecho de código com o modelo de projeto refinado é totalmente automática.

INITIALISATION
  $m.MOVLW(0); \quad m.MOVWF(0)$

A implementação da operação $advance$ é realizada com uma chamada a um bloco contendo 10 instruções de montagem, apresentado a seguir (a semântica de cada instrução foi apresentada na seção 4.1). Essas instruções são obtidas utilizando técnicas clássicas de compilação. Observe que, como o fluxo de execução não é linear e inclui saltos, a implementação B não pode ser simplesmente o sequenciamento dessas instruções [5]. A solução proposta na abordagem apresentada consiste em (1) compor essas instruções em uma instrução condicional que seleciona a instrução a ser executada em função do valor do contador de programa, e (2) executar essa instrução condicional enquanto o contador de programa estiver dentro da faixa correspondente. O programa é gerado assumindo que é carregado no endereço 0 da memória de instruções. Usando a notação B, essa organização pode ser realizada combinando um laço **WHILE** e um condicional **CASE**[3]:

[3] As instâncias das instruções da PIC são sublinhadas para destaque.

$advance =$
$\quad \underline{m.CALL(0);}$
$\quad \mathbf{WHILE}\ m.pc < 10\ \mathbf{DO}$
$\quad\quad \mathbf{CASE}\ m.pc\ \mathbf{OF}$
$\quad\quad\quad \mathbf{EITHER}\ 0\ \mathbf{THEN}\ \underline{m.BTFSC(0,1)}$
$\quad\quad\quad \mathbf{OR}\ 1\ \mathbf{THEN}\ \underline{m.GOTO(8)}$
$\quad\quad\quad \mathbf{OR}\ 2\ \mathbf{THEN}\ \underline{m.BTFSC(0,0)}$
$\quad\quad\quad \mathbf{OR}\ 3\ \mathbf{THEN}\ \underline{m.GOTO(6)}$
$\quad\quad\quad \mathbf{OR}\ 4\ \mathbf{THEN}\ \underline{m.MOVLW(1)}$
$\quad\quad\quad \mathbf{OR}\ 5\ \mathbf{THEN}\ \underline{m.GOTO(9)}$
$\quad\quad\quad \mathbf{OR}\ 6\ \mathbf{THEN}\ \underline{m.MOVLW(2)}$
$\quad\quad\quad \mathbf{OR}\ 7\ \mathbf{THEN}\ \underline{m.GOTO(9)}$
$\quad\quad\quad \mathbf{OR}\ 8\ \mathbf{THEN}\ \underline{m.MOVLW(0)}$
$\quad\quad\quad \mathbf{OR}\ 9\ \mathbf{THEN}\ \underline{m.MOVWF(0)}$
$\quad \mathbf{INVARIANT}\ \ldots$
$\quad \mathbf{VARIANT}\ \ldots$
$\quad \underline{m.RETURN}$

Em B, um laço tem duas anotações: o invariante, que serve a estabelecer a sua pós-condição, e o variante que visa mostrar a término da execução. O invariante do laço estabelece os valores possíveis do contador de programa e do apontador de pilha. Também especifica uma condição invariante do programa em cada ponto possível de execução, ou seja para cada valor possível do contador de programa. Para o ponto inicial e final de execução, o próprio invariante do módulo deve ser estabelecido. Para os demais pontos de execução, e na ausência de laços, o invariante pode ser calculado combinando os invariantes calculados para os pontos de execução anteriores no programa e a semântica de cada tipo de instrução. Quando há um laço no modelo algorítmico, pode e deve-se usar o invariante do próprio laço.

$\mathbf{INVARIANT}$
$0 \leq m.pc \wedge m.pc \leq 10 \wedge m.sp > 0 \wedge$
$m.mem(0) \in 0..2 \wedge$
$(m.pc = 0 \Rightarrow m.mem(0) \in 0..2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 1 \Rightarrow m.mem(0) = 2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 2 \Rightarrow m.mem(0) \neq 2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 3 \Rightarrow m.mem(0) = 1 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 4 \Rightarrow m.mem(0) = 0 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 5 \Rightarrow m.mem(0) = 0 \wedge m.mem(0) = m.count \wedge m.w = 1) \wedge$
$(m.pc = 6 \Rightarrow m.mem(0) = 1 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 7 \Rightarrow m.mem(0) = 1 \wedge m.mem(0) = m.count \wedge m.w = 2) \wedge$
$(m.pc = 8 \Rightarrow m.mem(0) = 2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 9 \Rightarrow (m.mem(0) \in 0..2 \wedge m.mem(0) = m.count \wedge m.w = color\_step(m.count))) \wedge$
$(m.pc = 10 \Rightarrow (m.mem(0) \in 0..2 \wedge m.mem(0) = color\_step(m.count)))$

O variante do laço é uma expressão inteira que deve ser estritamente positiva, e ser tal que seu valor diminui quando o corpo do laço é executado. Essas duas condições são suficientes para garantir o término da execução. No caso do exemplo adotado, como o fluxo de execução é uni-direcional, a expressão do variante laço pode ser simplesmente a diferença entre o valor atual do contador de programa e o endereço da última posição. No caso geral, deve-se usar técnicas de análise estática de código, como a análise de pior tempo de execução [12], para construir essa expressão.

$\mathbf{VARIANT}(10 - m.pc)$

A prova de consistência dessa operação foi realizada utilizando um assistente de prova. A grande maioria das obrigações de prova geradas foram verificadas automaticamente, sem assistência do usuário. Para as demais, a interação foi geralmente mínima, requerendo

apenas a instância de um quantificador e a seleção das hipóteses relevantes. Uma reengenharia do assistente de prova certamente permitiria descartar automaticamente essas obrigações de prova. Apenas uma obrigação de prova necessitou uma maior interação, devido à necessidade de provar uma propriedade aritmética bastante simples, mas fora do alcance do provador usado.

6. Conclusão

Esse artigo relata um estudo visando estabelecer uma abordagem inovadora para resolver o problema da geração de *software* correto por construção e, assim, contribuir à resolução de um dos grandes desafios da computação em pesquisa no Brasil e no mundo. Essa proposta baseia-se no método B, o qual fornece o embasamento teórico e o suporte ferramental necessários. O escopo do refinamento no método B foi estendido do tradicional nível algorítmico até o nível de linguagem de montagem, o qual possui um nível de abstração equivalente ao de código executável. Para realizar esse estudo, foi definido o modelo formal, na notação do método B, de uma plataforma computacional existente. Foi realizado o mapeamento de um modelo algorítmico de um sistema reativo simples para a linguagem de montagem dessa plataforma. No contexto teórico e prático do método B, procedeu-se à prova da conformidade do modelo de montagem com o modelo funcional inicial, propiciando assim uma demonstração da viabilidade da abordagem pelo menos para exemplos simples.

O impacto esperado para o projeto iniciado é prover, para uma determinada plataforma alvo (ou conjunto de plataformas), regras de compilação de modelos algorítmicos B para modelos em nível de linguagem de montagem dessa(s) plataforma(s). O resultado dessa compilação poderá ser verificado a posteriori, usando o arcabouço provido pelo método B. Uma meta, a maior prazo, é de, através de uma formalização do cálculo de refinamentos de B, provar a correção de tais conjuntos de regras de tradução e assim ter um mecanismo de produção de código executável correto a priori.

Em relação aos desafios propostos em [1], a abordagem proposta se encaixa prioritariamente no item "Desenvolvimento e adaptação de tecnologias e instrumentos de apoio à implementação [...] de *software* fidedigno por construção", e, por se basear no método B (um método formal de desenvolvimento de *software*), no item "Desenvolvimento e avaliação de modelos e ferramentas de modelagem de sistemas de *software* com base teórica sólida". Finalmente, apesar de não ser o foco dessa pesquisa no seu estágio atual, espera-se também contribuir para o item "Desenvolvimento de ferramentas de apoio ao processo de implementação e de evolução de *software*".

Em grandes linhas, um resultado desse trabalho é uma primeira avaliação do método B como ferramenta para o atendimento dos desafios acima. Pontualmente, observou-se que algumas restrições existentes nas versões atuais do método e de suas ferramentas prejudicam sua utilização para o refinamento de algoritmos em código de montagem. Essas restrições podem ser contornadas, como mostra esse artigo. Seria, por outro lado, interessante avaliar se elas poderiam ser eliminadas do método sem prejuízo para o rigor do desenvolvimento ou para o poder de automatização das ferramentas de suporte.

Por outro lado, outros requisitos mais significativos foram levantados que devem ser atendidos para a superação desses desafios. A maior parte deles é ao menos parcial-

mente satisfeita por B, mas ainda há muito a ser desenvolvido para que um método como B possa ser efetivamente aplicado de maneira mais geral e com menos esforço. Alguns desses requisitos são: deve ser possível e viável verificar formalmente a consistência entre os diferentes artefatos produzidos ao longo do processo de desenvolvimento de software; deve ser estimulada a evolução do software a nível de especificação ou de modelos abstratos, de maneira a garantir-se a manutenção da consistência entre os diferentes artefatos; deve ser possível derivar com um mínimo de esforço, e, tanto quanto possível, automaticamente, novos artefatos e novas verificações; deve haver um bom controle de versões e uma modularidade que permitam a redução da duplicação de esforços durante o desenvolvimento e durante a evolução do software. Concretamente, isso implica na necessidade de avanços significativos nas tecnologias de verificação formal e de geração de refinamentos e de código, pesquisas que certamente continuarão a serem desenvolvidas pelos grupos nas áreas de métodos formais, linguagens de programação e engenharia de software.

Referências

[1] Grandes Desafios da Pesquisa em Computação no Brasil: 2006–2016. `http://www.sbc.org.br`, 2006. Sociedade Brasileira de Computação.

[2] R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge Univ. Press, 1996.

[3] A. Cavalcanti, A. Sampaio, and J. Woodcock. Procedures and recursion in the refinement calculus. *Journal of the Brazilian Computer Society*, 5(1):5–19, 1998.

[4] A. Cavalcanti, A. Sampaio, and J. Woodcock. A refinement strategy for circus. *Formal Aspects of Computing*, 15(2–3):147–181, 2003.

[5] B. Dantas, D. Déharbe, S. Galvão, V. Medeiros Jr, and A. Moreira. Verified compilation based on the B method: an initial appraisal (extended version). Technical Report UFRN-DIMAp-2008-101-RT, UFRN-DIMAp, 2008.

[6] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In *VMCAI*, pages 78–78, 2005.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[8] E. Jaffuel and B. Legeard. LEIRIOS test generator: Automated test generation from B models. In *The 7th International B Conference*, pages 277–280, 2007.

[9] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall Int., 1990.

[10] P. Letouzey. A new extraction for Coq. In *TYPES 2002*, volume 2646 of *LNCS*, 2003.

[11] D. Ossami, J.-P. Jacquot, and J. Souquières. Consistency in UML and B multi-view specifications. In *IFM*, pages 386–405, 2005.

[12] C.Y Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, 1993.

[13] S. Schneider. *The B-Method: An Introduction*. Cornerstones of Computing Series. Palgrave, 2001.

[14] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, 2006.

[15] J. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall International Series in Computer Science. Prentice Hall, 2nd edition, 1992.