Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um estudo sobre verificação formal de sistemas concorrentes

João Paulo Carvalho Colu de Queiroz

Brasília
2012

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um estudo sobre verificação formal de sistemas concorrentes

João Paulo Carvalho Colu de Queiroz

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Computação

Orientador
Prof. Dr. Flávio Leonardo Cavalcanti de Moura

Brasília
2012

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Computação

Coordenadora: Prof. Dr. Mylene Christine Queiroz de Farias

Banca examinadora composta por:

Prof. Dr. Flávio Leonardo Cavalcanti de Moura (Orientador) — CIC/UnB
Prof. Dr. Daniel Lima Ventura — INF/UFG
Prof. Dr. Mauricio Ayala Rincón — CIC/UnB

CIP — Catalogação Internacional na Publicação

Queiroz, João Paulo Carvalho Colu de.

Um estudo sobre verificação formal de sistemas concorrentes / João Paulo Carvalho Colu de Queiroz. Brasília : UnB, 2012.
73 p. : il. ; 29,5 cm.

Tese (Mestrado) — Universidade de Brasília, Brasília, 2012.

1. Verificação Formal, 2. Lógica de Hoare, 3. Coq, 4. Linguagens Imperativas, 5. Java, 6. JML, 7. Krakatoa

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um estudo sobre verificação formal de sistemas concorrentes

João Paulo Carvalho Colu de Queiroz

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Computação

Prof. Dr. Flávio Leonardo Cavalcanti de Moura (Orientador)
CIC/UnB

Prof. Dr. Daniel Lima Ventura
INF/UFG

Prof. Dr. Mauricio Ayala Rincón
CIC/UnB

Prof. Dr. Mylene Christine Queiroz de Farias
Coordenadora do Mestrado em Computação

Brasília, 14 de junho de 2012

Abstract

This work presents a study of methodologies to formally verify applications developed with imperative languages, specially with the Java language. The theoretical formalisms shown include Hoare Logic, which is used to sketch properties on imperative languages, and JML constructions (based on Hoare Logic), which is a specification language used to specify the expected behavior from Java programs. The tools shown are the Krakatoa system, used to convert JML specifications into proof obligations, and the Coq interactive proof environment, used to verify proof obligations. Finally, this paper presents a case study that employs the theoretical and practical proposed framework.

Keywords: Formal Verification, Hoare Logic, Coq, Imperative Languages, Java, JML, Krakatoa

Resumo

Este trabalho apresenta um estudo de metodologias para verificação formal de aplicativos desenvolvidos em linguagens imperativas, em especial, na linguagem Java. Os formalismos teóricos mostrados incluem a Lógica de Hoare, usada para representar propriedades de aplicações imperativas, e construções da linguagem de especificação JML (baseada na Lógica de Hoare), usada para especificar o comportamento esperado de aplicações codificadas em Java. As ferramentas mostradas são o sistema Krakatoa, usado para converter especificações JML em obrigações de prova, e o ambiente interativo de provas Coq, usado para verificar obrigações de prova. Finalmente, exibe-se um estudo de caso que utiliza o ferramental teórico e prático proposto.

Palavras-chave: Verificação Formal, Lógica de Hoare, Coq, Linguagens Imperativas, Java, JML, Krakatoa

Sumário

Lista de Figuras viii

Lista de Tabelas ix

1 Introdução 1

2 Fundamentação Teórica 3

2.1 Lógica de Hoare 3

2.1.1 O sistema dedutivo de Hoare 3

2.2 A linguagem JML 5

2.3 O *software* Krakatoa 7

2.4 O ambiente de provas Coq 8

2.4.1 Expressões, fórmulas lógicas e novas constantes 8

2.4.2 Proposições e provas 9

3 Estudo de Caso 11

3.1 Vida Ambiente-Assistida 11

3.1.1 Conexão com os fundamentos teóricos pesquisados 12

3.1.2 Agente de Emergência 12

3.1.3 Componentes não verificados 13

3.2 Especificações JML 14

3.2.1 Classe principal 14

3.2.2 Classe `Agent` 16

3.2.3 Classes usadas para enfileiramento de emergências 17

3.2.4 Classes usadas para geração de novo comportamento 19

3.2.5 Classes usadas para sinalizar emergências 20

3.2.6 Classes herdadas pelos comportamentos 21

3.2.7 Classe `estados` 22
3.2.8 Outras classes 23
3.3 Obrigações de provas 23
3.3.1 Objetivos de prova principais 25
3.3.2 Objetivos triviais 35
3.3.3 Objetivos não provados 36
3.4 *Feedback* do uso das ferramentas 36
3.4.1 *Workspace* Java 36
3.4.2 Programação genérica 38
3.4.3 *Model fields* 39
3.4.4 Classes internas 40
3.4.5 Teorias e construções adicionais 41
3.4.6 Provador automático Yices 43
3.4.7 Objetivos de prova de outros arquivos 43

4 Conclusão **47**
4.1 Trabalhos futuros 48

Referências **49**

A Códigos Anotados **52**
A.1 `MonitorAcuteEmergencyBehaviour.java` 52
A.2 `Agent.java` 54
A.3 `Queue_Emergency.java` 55
A.4 `ThreadedBehaviourFactory.java` 56
A.5 `ThreadedBehaviourWrapper.java` 57
A.6 `HashMap.java` 57
A.7 `Behaviour.java` 58
A.8 `estados.java` 59

I Trechos da axiomatização Java *memory heap* **60**

Lista de Figuras

2.1 Esquema de geração de condições de verificação; adaptado de [18]. 8

3.1 Tela do `why3ide` com obrigações de prova geradas pelo Krakatoa 24

Lista de Tabelas

2.1 Provadores suportados por Why3 7

3.1 Objetivos de prova principais 26

3.2 Objetivos de prova triviais 45

3.3 Objetivos de prova triviais (cont.) 46

3.4 Objetivos de prova não provados 46

3.5 Objetivos de prova exclusivos da classe `estados` 46

Capítulo 1

Introdução

A necessidade de verificação da correção de um determinado algoritmo apresenta-se ao longo de toda a história do cálculo: após a criação de um procedimento computacional, o desenvolvedor ou os usuários interessam-se em saber se o produto atende aos requisitos originais.

Os métodos de verificação formal têm recentemente sido empregados pela indústria e obtido sucesso em diferentes níveis de utilização: compiladores [31], sistemas operacionais [25], controle de tráfego aéreo [9, 30], *smart cards* [34]. Esta utilização crescente gera uma demanda por profissionais aptos a aplicá-los [24].

Este trabalho apresenta ferramentas teóricas e computacionais que apoiem ações de verificação formal, em especial, de linguagens de uso comercial, tais como Java e C; além de um estudo de caso em linguagem Java. O trabalho está dividido em dois capítulos principais que apresentam respectivamente aspectos teóricos e a experiência com o estudo de caso.

No Capítulo 2, apresentam-se:

- a Lógica de Hoare (Seção 2.1), usada para especificar formalmente aplicações desenvolvidas com linguagens de programação imperativa, em especial, linguagens que admitem "efeitos colaterais" encontrados, por exemplo, em operações de atribuição;
- a linguagem de especificação JML (Seção 2.2), a qual baseia-se na Lógica de Hoare, e que foi usada para especificar as propriedades esperadas do estudo de caso Java;
- o sistema Krakatoa (Seção 2.3), que possibilitou converter especificações JML e código Java em obrigações de prova;
- o ambiente interativo de provas Coq (Seção 2.4), que foi usado para verificar as obrigações de prova geradas pelo Krakatoa.

No Capítulo 3, exibe-se o caso prático em que foi empregado esse ferramental: verificação de parte de um sistema multiagente codificado em linguagem Java, o qual atua no domínio de aplicações de Vida Ambiente-Assistida (VAA), intentando criar um ambiente doméstico que monitore a saúde pessoal dos assistidos. Criaram-se então anotações JML

diretamente no código-fonte Java dessa aplicação VAA, gerando as obrigações de prova através do sistema Krakatoa. As obrigações de prova geradas foram analisadas no Coq.

Este trabalho espera contribuir como um exemplo real do uso das ferramentas; além de discutir alguns pontos de atenção (Seção 3.4) quanto ao seu uso, através da experiência adquirida durante a verificação do sistema VAA. Esses pontos, por vezes, requereram uma adequação na estratégia de anotação ou de prova. Em particular, exibe-se:

- o assistente de prova Coq (Seção 2.4) sendo usado como plataforma de convergência de objetivos de prova gerados;
- o uso de provadores automáticos para diminuir a intervenção manual em obrigações de prova triviais;
- documentação formal em JML das partes das APIs (Application Programming Interfaces) usadas pelo sistema VAA.

Pressupõe-se do leitor deste texto conhecimento das construções básicas da linguagem Java, uma vez que apenas algumas nuanças dessa linguagem são aqui realçadas. Pressupõe-se também experiência com algum provador interativo, pois o provador Coq é aqui brevemente apresentado.

Capítulo 2

Fundamentação Teórica

2.1 Lógica de Hoare

Uma linguagem de programação imperativa caracteriza-se por conter sequências de comandos que podem modificar o estado de execução do programa, ou seja, admite comandos com "efeitos colaterais"[1][35], tal como uma atribuição, onde a operação $\{x := 1\}$ modifica o estado de memória na região que acomoda o valor da variável x. Triplas de Hoare permitem especificar programas com essa característica, possibilitando expressar pré-condições e pós-condições [24, p. 224], como será mostrado na seção seguinte (Seção 2.1.1).

Floyd [19] propôs uma técnica de modelagem de aplicações imperativas através de fluxogramas (grafos dirigidos) e a geração de condições de verificação ao longo do fluxograma. Influenciado por esse trabalho, Hoare [22] apresentou um sistema dedutivo composto de axiomas e regras de inferência. Dijkstra75 [14] criou um cálculo para a Lógica de Hoare, posteriormente denominado de Cálculo da Pré-condição mais Fraca [2]. Esses trabalhos referem-se a um subconjunto de instruções presente em todas as linguagens imperativas, conquanto não tratam de uma linguagem de programação específica.

O sistema dedutivo proposto por Hoare é usado como fundamento de diversas linguagens de especificação para linguagens imperativas, tais como as linguagens JML e ACSL, descritas na Seção 2.2.

2.1.1 O sistema dedutivo de Hoare

O sistema dedutivo de Hoare é baseado na noção de *triplas de Hoare* usada para estabelecer a conexão entre uma pré-condição ϕ, um algoritmo P e uma pós-condição (descrição de resultado) ψ:

$$\phi \; \{P\} \; \psi \tag{2.1}$$

[1] Tradução do termo original, em inglês: *side-effects statements.*

[2] *Weakest Precondition Calculus*

Esta notação deve ser interpretada como "se a asserção ϕ for verdadeira antes do início do programa P, então a asserção ψ será verdadeira quando esse programa terminar".

A operação de atribuição de uma variável (por exemplo, $x := f$) é representada por um axioma:

Axioma 1. $\vdash \psi_0$ *{x := f}* ψ *onde x é um identificador de variável; f é uma expressão, eventualmente contendo x; e ψ_0 é obtido a partir de ψ substituindo-se todas as ocorrências de x por f.*

Este axioma, formulado originalmente em [22], foi apresentado de forma mais geral em [8]:

Axioma 2. $\phi\ \{x := f\}\ \psi \equiv \phi \Rightarrow \psi[x := f]$

onde, a substituição $\psi[x := f]$ denota uma expressão que é igual a ψ, exceto pelo fato de que todas as aparições de x em ψ são substituídas por f.

Villamizar [8, p. 66 *et seq.*] descreveu sua formulação do axioma da atribuição da seguinte forma:

> Em vez de explicar a atribuição pela forma em que se executa, pode-se entendê-la em termos de substituição textual ou sintática. A ideia chave envolve raciocinar "para trás"; das pós-condições para as pré-condições.
>
> [...] como, "após" a atribuição, x recebe o valor que tinha a expressão f "antes" da atribuição, este axioma expressa que, para que ψ aplique-se a x "após" a atribuição, esta asserção (ψ) deveria valer para f (ψ aplicada a f) "antes" da atribuição. Portanto, a pré-condição ϕ deve implicar $\psi[x := f]$.
>
> [...] Aplicando o axioma da atribuição ao caso particular $\phi \equiv \psi[x := f]$ obtemos $\psi[x := f]$ {x := f} ψ. Afirmamos que $\psi[x := f]$ é a "pré-condição mais fraca"[3] (*weakest precondition*) para que a execução de {x := f} termine com ψ verdadeiro.

Nota-se que a partir desse axioma mais geral, pode-se obter o axioma original:

$$\psi[x := f]\ \{x\ :=\ f\}\ \psi \tag{2.2}$$

Um programa imperativo consiste de uma sequência de comandos que são escritos um após o outro. Na "linguagem imperativa" definida por Hoare, comandos são encadeados usando-se um ponto-e-vírgula: $(P_1; P_2; \ldots; P_n)$. O conjunto de regras de inferência a seguir permite expressar operações de linguagens imperativas, como por exemplo o encadeamento de programas (Regra de Inferência 1) ou consequências (Regra de Inferência 2 e Regra de Inferência 3):

Regra de Inferência 1. *Se* $\vdash \phi\{P_1\}\psi_1$ *e* $\vdash \psi_1\{P_2\}\psi$ *então* $\vdash \phi\{(P_1; P_2)\}\psi$

Regra de Inferência 2. *Se* $\vdash \phi\{P\}\psi$ *e* $\vdash \psi \rightarrow \chi$ *então* $\vdash \phi\{P\}\chi$

Regra de Inferência 3. *Se* $\vdash \phi\{P\}\psi$ *e* $\vdash \chi \rightarrow \phi$ *então* $\vdash \chi\{P\}\psi$

[3] um predicado ψ diz-se "mais fraco" que outro ϕ, se $\phi \Rightarrow \psi$ vale.

Dentre outras construções de linguagens imperativas, é importante a capacidade de expressar iterações, ou seja, executar uma porção de programa (P) repetidamente até uma determinada condição (β) tornar-se falsa, sendo que, ao longo de toda a execução, o predicado ϕ (conhecido como **invariante**) é verdadeiro. Assim, pode-se caracterizar o programa *while* β *do* P por:

Regra de Inferência 4. *Se* $\vdash \phi \wedge \beta\{P\}\phi$ *então* $\vdash \phi\{\text{while } \beta \text{ do } P\}\neg\beta \wedge \phi$

2.2 A linguagem JML

As linguagens ACSL e JML são linguagens de especificação de propriedades inspiradas no formalismo de Hoare. Com essas linguagens, pode-se anotar aplicações escritas respectivamente em linguagem C e Java. Além de serem baseadas na mesma lógica, elas possuem um cerne de construções comum, uma vez que ACSL (ANSI/ISO C Specification Language) é uma adaptação (para a linguagem C) de JML (Java Modeling Language).

As especificações (ou anotações) são feitas usando pré-condições, pós-condições e invariantes. Essas anotações são feitas no próprio código-fonte, dentro de comentários, e portanto sua introdução nesses arquivos não afeta o comportamento do *software* gerado. As anotações utilizam uma formatação especial: `/*@ anotação */` ou `//@ anotação`. Somente algumas poucas construções comuns são aqui apresentadas.

Consideremos o exemplo abaixo de uma função `maximo()` usando ponteiros (codificada em linguagem C):

```
/*@
  @ requires \valid(x) && \valid(y) && \valid(res);
  @ ensures *res >= *x && *res >= *y;
  @ ensures *res == *x || *res == *y;
  @ ensures \result == *res;
  @ ensures *x == \old(*x) && *y == \old(*y);
  @*/
int maximo(int *x, int *y, int *res)
{
    if (*x > *y)
        *res = *x;
    else
        *res = *y;

    return *res;
}
```

A especificação da função está escrita imediatamente acima da declaração da função. Neste exemplo, o contrato (especificação) contém pré-condições (cláusula `requires`) quanto à validade dos ponteiros usados e pós-condições (cláusulas `ensures`) que garantam a correção da informação em `*res` e do retorno da função (ambas retornam o máximo),

além de assegurar que `*x` e `*y` não venham a ser modificadas. Para tanto, usaram-se as seguintes construções:

- `\valid()` gerará condição de verificação para a certificação de que o ponteiro parametrizado tenha sido previamente alocado (válido); o que equivale a testar se o conteúdo é diferente de `null`, por exemplo, `p` é válido se `p != null`, caso contrário `p` será inválido;
- `\result` denota o resultado de uma função (retorno de um procedimento);
- `\old()` indica o valor inicial de um parâmetro, ou seja, o valor originalmente recebido pela função;
- os operadores aritméticos `>=` e `==` têm o significado usual;
- `&&` e `||` correspondem respectivamente aos operadores lógicos conjunção e disjunção;
- duas cláusulas `ensures` estão relacionadas através de uma conjunção, ou seja, todas as asserções constantes nas cláusulas do exemplo devem ser válidas ao término da execução da função.

Essas linguagens de especificação são essencialmente baseadas em lógica de primeira ordem, assim como outras linguagens de especificação comportamental frequentemente usadas, tal como Z [39]. Não obstante, as linguagens de especificação aqui estudadas suportam também:

- recursividade: o fornecimento de medidas para verificação da terminação de funções recursivas, inclusive no caso mais geral de recursividade mútua; para tanto deve-se fornecer a expressão de decrescimento através da cláusula `decreases`;
- predicados indutivos, construções como `inductive P(x1, ... ,xn) { case c1 : p1; ... case ck : pk; }`
- tipos polimórficos (EXPERIMENTAL), por exemplo, pode-se considerar a declaração `//@ type list <A>;`
- construções de ordem superior (EXPERIMENTAL), através de termos como `\lambda T1 x1, ... , Tn xn ; t`, que equivaleria ao λ-termo tipado $\lambda x_1 : T_1, ..., x_n : T_n.t$.

No estudo de caso apresentado (Capítulo 3), estas construções não foram empregadas, pois, com exceção dos tipos polimórficos, a modelagem da aplicação Java verificada não as requereu: as partes verificadas não possuem métodos recursivos, tratamento de casos ou, por exemplo, métodos que recebam métodos como argumento (ordem superior). Quanto à definição de tipos polimórficos, algumas restrições foram encontradas nesse recurso experimental impedindo a definição de novos tipos (Seção 3.4.5).

Referências para as linguagens de especificação ACSL e JML podem ser encontradas respectivamente em [3] e [27].

2.3 O *software* Krakatoa

O Krakatoa [26] é uma plataforma de análise estática de código-fonte escrito em linguagem Java, voltado para as ações de inferência lógica. Aceita especificações expressas em JML. O Frama-C [20] é sistema análogo voltado para a verificação de programas escritos em linguagem C e anotados com ACSL.

As ferramentas analisam o código e as especificações de comportamento fornecidas, validam-nos quanto a erros sintáticos e os convertem para uma linguagem comum, que será entrada para o *software* intermediário Why [38]. Esta aplicação age como uma *interface* para o Krakatoa (e o Frama-c) e os provadores, facilitando o acoplamento de novos ambientes de prova (automáticos ou manuais), bem como de novos tradutores de linguagens imperativas e linguagens de especificação. Atualmente são suportadas as referidas traduções de Java/JML e C/ACSL. A relação dos provadores suportados pode ser encontrada na Tabela 2.1.

Interativos	**Automáticos**
Coq	Alt-Ergo
PVS (em progresso)	CVC3
	E-prover
	Gappa
	Simplify
	SPASS
	Vampire
	veriT
	Yices
	Z3

Tabela 2.1: Provadores suportados por Why3

As condições de verificação serão então traduzidas para a linguagem específica de um dado provador de escolha do usuário, e que seja suportada pela plataforma Why. Em seguida o provador é invocado pelo sistema.

Como será visto no caso trabalhado (Capítulo 3), mesmo que o foco de uma ação de verificação seja o uso de um determinado provador interativo, o uso de provadores automáticos mecaniza a prova de um grande volume de obrigações triviais, de forma que o provador interativo pode ser utilizado somente para as obrigações não-triviais. Um esquema geral do processo de tradução das especificações ACSL/JML para os ambientes de prova (automáticos ou interativos), passando pelo Why, pode ser visto na Figura 2.1. Manuais de operação destes *softwares* podem ser encontradas em [33], [13] e [7].

Todos os *softwares* descritos nesta seção são de código aberto, licenciados sob a GNU LGPL v2. Há versões disponíveis para Linux, Mac OS X e Windows, contudo não há homologações para versões específicas desses sistemas operacionais.

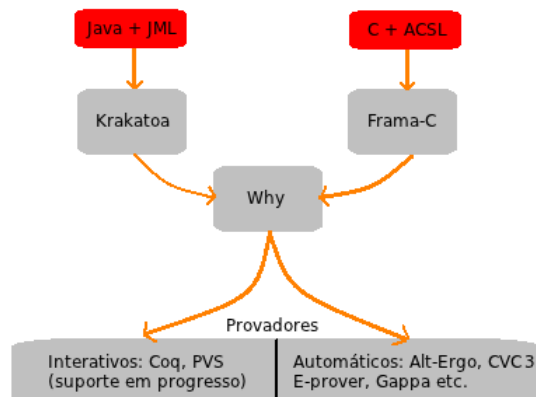


Figura 2.1: Esquema de geração de condições de verificação; adaptado de [18].

2.4 O ambiente de provas Coq

O assistente de prova Coq [12] é um sistema que permite expressar especificações, desenvolver programas ou realizar provas matemáticas baseadas em uma lógica de ordem superior (Cálculo de Construções Indutivas, formalismo derivado do Cálculo-λ com tipos). Adicionalmente, este ambiente pode ser usado como estrutura lógica para que alguém forneça axiomas para novas lógicas e realizar provas nesses novos sistemas. Por exemplo, pode ser usado para implementar lógicas modais, temporais ou pode ser usado para raciocinar sobre programas imperativos [10].

Assim como os *softwares* da Seção 2.3, Coq é licenciado sob a GNU LGPL v2 e encontra-se disponível para Linux, Mac OS X e Windows.

A seguir, são brevemente apresentados alguns dos recursos e características do Coq introduzidos por [5]. Para uma discusão ampla da ferramenta e do formalismo subjacente, pode-se consultar [6].

2.4.1 Expressões, fórmulas lógicas e novas constantes

A notação $A : B$ é usada para indicar que a expressão A tem como tipo a expressão B. Entre essas expressões, algumas podem ser interpretadas, por exemplo, como proposições (têm tipo `Prop`), outras como números naturais (têm tipo `nat`) e outras como elementos de estruturas de dados mais complexas. Fórmulas lógicas complexas podem ser construídas combinando proposições com conectivos lógicos ou outras expressões. Pode-se também construir uma nova função com a palavra-chave `fun`, a qual substitui o símbolo λ do Cálculo-λ. Para conferir se uma fórmula é bem-formada, usa-se o comando `Check`, por exemplo:

```
Check (fun x:nat => x = 3).
fun x : nat => x = 3 : nat -> Prop

Check (forall x:nat, x < 3 \/ (exists y:nat, x = y + 3)).
```

```
forall x : nat, x < 3 \/ (exists y : nat, x = y + 3) : Prop

Check (let f := fun x => (x * 3, x) in f 3).
let f := fun x : nat => (x * 3, x) in f 3 : nat * nat
```

A intuição do comando `let` é a de substituição, por exemplo: `let x := y in t` é equivalente ao termo `t`, exceto pelo fato de que as ocorrências de `x` serão substituídas pelo termo `y`.

2.4.2 Proposições e provas

O sistema Coq baseia-se no isomorfismo de Curry-Howard [23]: tipos correspondem a proposições, e provas a termos. Assim, a notação $A : B$ denota que o termo A é do tipo B ou ainda que A é uma prova da fórmula lógica B.

Um teorema que prova uma implicação $A \Rightarrow B$ pode ser interpretado como um termo em que seu tipo expressa uma função (tipo funcional), a qual recebe um argumento do tipo A produzindo um retorno do tipo B. Esse tipo funcional é expresso em Coq pela notação *A* `->` *B*. Assim, uma função f desse tipo (denotada por *f* `:` *A* `->` *B*) pode ser aplicada a um termo t do tipo A (denotado por $t : A$), o que permitiria deduzir B, pois o retorno da aplicação de f a t seria desse tipo, ou seja, $f\ t : B$. Essa dinâmica corresponde à regra de inferência *modus ponens*.

A construção de provas dessa forma, ou seja, apresentando manualmente termos que possuam determinado tipo, torna-se inviável para tipos (proposições) não-triviais. A abordagem usualmente empregada envolve o uso de táticas: programas que, em determinados casos, permitem a construção automática de trechos de provas. A seguir, usa-se a conhecida propriedade sobre números naturais $\forall n \in \mathbb{N}, 2 * \sum_{i=0}^{n} i = n * (n + 1)$ para ilustrar uma prova em Coq usando táticas:

```
Fixpoint somar_ate_n (n:nat) : nat :=
    match n with
              | 0 => 0
              | S n' => S n' + somar_ate_n n'
    end.

(* Carregar aritmética básica de Peano *)
Require Import Arith.

Theorem somar_ate_n_prop:
   forall n:nat, 2*(somar_ate_n n) = n*(S n).
Proof.
   intro n.
   induction n.

       (* Base de indução *)
       simpl.
```

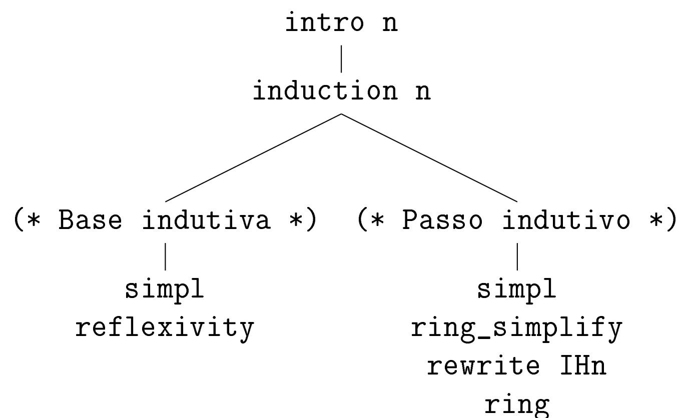
```
        reflexivity.

        (* Passo indutivo *)
        simpl.
        ring_simplify.
        rewrite IHn.
        ring.
Qed.
```

Apresenta-se a intuição das táticas apresentadas no *script* acima, conforme descrito por Filliâtre *et al.* [17]:

- `intro`: introduz antecedentes de implicações e parâmetros de `forall`;
- `induction`: inicia uma prova por indução;
- `simpl`: simplifica um termo usando computação, por exemplo, aplicada em um objetivo do tipo `1 + 1` produzirá um objetivo do tipo `2`;
- `reflexivity`: soluciona objetivos que tenha a forma `t = u`, verificando se `t` é conversível em `u`, caso em que soluciona o objetivo com sucesso;
- `ring`: soluciona equações contendo expressões polinomiais da estrutura algébrica anel (ou semi-anel);
- `ring_simplify`: opera em objetivos simultaneamente como `ring` e `simpl`, ou seja, atual como `ring` normalizando (simplificando) os objetivos;
- `rewrite termo`: reescreve o objetivo de acordo com uma igualdade fornecida; então o parâmetro `termo` deve ter uma igualdade em seu tipo; exemplo, `rewrite t`, onde `t : x = 5`, substitui as ocorrências de `x` no objetivo por `5`.

Observe ainda que a prova do teorema exemplo `somar_ate_n_prop` (acima) pode ser vista ainda em uma estrutura de árvore, representando os dois subobjetivos da prova por indução:



Capítulo 3

Estudo de Caso

3.1 Vida Ambiente-Assistida

Vida Ambiente-Assistida (VAA) é um domínio de soluções de monitoramento que tenciona a criação de um ambiente doméstico inteligente. As soluções que compõem esse domínio usualmente envolvem dispositivos de *hardware* e *software*, além de um conjunto de serviços integrados que provejam assistência a pessoas. Por exemplo, em um ambiente em que viva uma pessoa idosa ou com alguma necessidade especial, sensores podem captar a ocorrência de atividades anormais, como uma queda, e enviar um alerta para uma central de atendimento a incidentes.

Inserido nesse contexto, o EMERGE [16] é um projeto que visa detectar situações de emergência, propondo o uso de sensores embutidos na residência da pessoa assistida para a captura de diversos dados do ambiente. Estes dados devem ser processados por um sistema de *software* que identifique situações de emergência, tratando-as adequadamente. O EMERGE propõe uma especificação da arquitetura e dos requisitos desse sistema, envolvendo a combinação de sistemas multiagentes, porém não implementa de fato o sistema.

O projeto apresentado em [36] implementa a parte do sistema responsável pelo encaminhamento das situações de emergência. Para a análise e desenho dos agentes, foi usada a metodologia GAIA [36, p. 59] e, para sua implementação, a linguagem Java. Para esboçar determinadas propriedades de *liveness* [1], isto é, propriedades que devem ser cumpridas caso algum evento específico aconteça, empregou-se um método semiformal[2] que documentou o comportamento esperado (comportamento de intenção) do sistema. Além disso, para expressar o conteúdo das mensagens usou-se um padrão[3] que define mensagens a partir da combinação de termos e predicados.

[1] Também denominadas de responsabilidades de sobrevivência.
[2] Pertencente à metodologia GAIA de desenvolvimento de sistemas multiagentes.
[3] FIPA ACL

3.1.1 Conexão com os fundamentos teóricos pesquisados

A linguagem Java, usada pelo sistema, compatibiliza-se com os fundamentos dispostos pela Lógica de Hoare (Seção 2.1) e, dessa forma, usando a linguagem JML (Seção 2.2), especifica-se o comportamento esperado para o código Java a partir de tais predicados. Com o *software* Krakatoa (Seção 2.3), as obrigações de prova para as Triplas de Hoare (especificação de pré-condições e pós-condições em JML, juntamente com o código Java) são descarregadas para o ambiente de provas Coq (Seção 2.4). Os lemas triviais são provados com o auxílio de algum verificador automático (CVC3, Ergo, *SMT provers*), apenas para efeito de mecanização de provas, uma vez que todos os lemas provados automaticamente também podem ser provados de forma interativa (manual).

3.1.2 Agente de Emergência

Os comportamentos selecionados para verificação foram extraídos de [36, p. 98 *et seq.*]. A seleção ocorreu mediante entrevista com o autor, tendo este, por juízo próprio, considerado essa seleção como a parte mais crítica do sistema. Cabe salientar que, de 109 arquivos ".java", apenas três referem-se ao "Agente de Emergência", que implementa o papel "Assistente de Emergência".

As classes Java que compõem esse assistente são:

- `RequestConfirmationBehaviour`;
- `MonitorLongTermDeviationBehaviour`;
- `MonitorAcuteEmergencyBehaviour`.

Essas classes (arquivos) encontram-se na seguinte subpasta do *workspace* do projeto:

`EmergencyAgentBundle/src/br/unb/cic/aal/behaviour`

Esses comportamentos foram implementados tomando como base expressões *liveness* do agente. A responsabilidade de sobrevivência que o `EmergencyAgent` (agente de emergência) deve atender está especificada conforme abaixo:

$$(\underline{InvocarDetectarAlarme}|\underline{InvocarDetectarEmergencia})^{\omega}$$
$$.PedirConfirmacao.[\underline{InvocarEnviarDadosDispatcher}.EnviarNotificacao] \ ||$$
$$(\underline{InvocarDetectarDesvios})^{\omega}.\underline{InvocarEnviarDadosSSM}.EnviarNotificacao$$

As regras de formação de expressões *liveness* (como a expressão acima) estão detalhadas em [36, p. 62 *et seq.*].

Além disso, o autor do sistema separou porções dessa expressão para cada comportamento (classe/arquivo ".java") do agente. Relacionam-se abaixo as subexpressões *liveness* que cada comportamento/classe do agente `EmergencyAgent` deve atender:

- `MonitorAcuteEmergencyBehaviour`:
  $(\underline{InvocarDetectarAlarme}|\underline{InvocarDetectarEmergencia})^{\omega}$
- `RequestConfirmationBehaviour`:
  $PedirConfirmacao.[\underline{InvocarEnviarDadosDispatcher}.EnviarNotificacao]$
- `MonitorLongTermDeviationBehaviour`:
  $(\underline{InvocarDetectarDesvios})^{\omega}.\underline{InvocarEnviarDadosSSM}.EnviarNotificacao$

Destes três comportamentos, o comportamento `MonitorAcuteEmergencyBehaviour` é o responsável pela detecção de emergências agudas, tendo sido o foco das ações de verificação.

Para fazer a verificação do comportamento `MonitorAcuteEmergencyBehaviour` isoladamente, foram realizadas anotações JML referentes à semântica de intenção dos agentes (solicitações à camada JADE) e a consecução das respectivas obrigações de prova no ambiente Coq. Ao todo, 19 classes necessitaram de algum tipo de anotação para que `MonitorAcuteEmergencyBehaviour` tivesse seu comportamento especificado, como será mostrado na seção seguinte (Seção 3.2):

```
Agent.java
Behaviour.java
ConcurrentLinkedQueue_Emergency.java
ConcurrentLinkedQueue.java
CyclicBehaviour.java
DataStore.java
DetectEmergencyService.java
Emergency.java
estados.java
HashMap.java
MonitorAcuteEmergencyBehaviour.java
Observable.java
Observer.java
Queue_Emergency.java
Queue.java
RequestConfirmationBehaviour.java
SimpleBehaviour.java
ThreadedBehaviourFactory.java
ThreadedBehaviourWrapper.java
```

3.1.3 Componentes não verificados

As classes de comportamento fazem uso de JADE (Java Agent DEvelopment), *framework* que fornece funcionalidades de *middleware* para agentes. Assim a aplicação em estudo não tratou de detalhes físicos concernentes ao estabelecimento de canais de comunicação; tais funcionalidades foram delegadas pela aplicação à camada JADE.

Dessa forma, no que tange à verificação formal do sistema, confia-se na implementação dos *frameworks* usados pelo sistema, apenas especificando-se o seu comportamento conforme sua documentação oficial. Para que se possa garantir o funcionamento correto dessas ferramentas, estas devem ser objeto de ação de verificação específica, o que não faz parte do escopo do presente estudo.

3.2 Especificações JML

O processo de execução das anotações deu-se em duas etapas:

1. anotação da classe principal (`MonitorAcuteEmergencyBehaviour`) do comportamento do agente;
2. anotação de classes periféricas à classe principal (apenas código usado por essa classe).

Para cada classe, sempre usou-se o mesmo método para criação das anotações: analisou-se o comportamento de intenção documentado [36, 4, 1] e criou-se especificações JML que dele representassem uma abstração lógica. O código-fonte das classes Java com suas respectivas anotações JML pode ser encontrado em
`https://lambda.cic.unb.br/svn/mestrado/jp/vida-assistida/codigo-anotado`.

3.2.1 Classe principal

A classe `MonitorAcuteEmergencyBehaviour`, pertencente ao `EmergencyAgent`, responsabiliza-se por monitorar eventos de emergência. No caso de ocorrência de um destes eventos, deve requisitar a outro agente (`ElderlyAgent`) que realize a ação de confirmar ou não a emergência junto à pessoa assistida. Esta interação do `EmergencyAgent` com o `ElderlyAgent` é representada pela inclusão de `RequestConfirmationBehaviour` na lista de comportamentos que o agente `EmergencyAgent` pode ter (executar).

A classe `MonitorAcuteEmergencyBehaviour` foi codificada sendo "derivada" da classe `CyclicBehaviour`, classe da API JADE [4], e tendo "implementado a *interface*" `Observer`, classe da API Java SDK [1]. Essa classe possui três métodos:

- `onStart()`: esse método é executado na inicialização do comportamento; é executado apenas uma vez, o que ocorre imediatamente antes da primeira chamada ao método `action()` do comportamento;
- `action()`: para classes "derivadas" de `CyclicBehaviour` (que é o caso da classe `MonitorAcuteEmergencyBehaviour`), este método é executado continuamente, de forma cíclica;
- `update()`: classes que "implementam a *interface*" `Observer` são notificadas da ocorrência de eventos externos por meio do método `update()`. Mais informações sobre esse mecanismo de notificação podem ser obtidas na Seção 3.2.5.

Por fim, é requisito do agente que as confirmações de emergência ocorram de forma assíncrona, ou seja, deve haver um enfileiramento de notificações e confirmações de emergência, não bastando modelar de forma booleana essas informações (emergência sinalizada ou não), mas sim quantificadas (rastrear a quantidade de emergências sinalizadas).

Anotações

Para armazenar a informação da quantidade de emergências detectadas, foi criada a classe `estados`, não usada pelo código, mas tão somente pelas anotações (vide Seção 3.2.7), contendo o atributo `qtde_emergencias`.

O recebimento das notificações das emergências, atividade delegada ao `update()`, é anotado em função do tipo (cláusula `instanceof`) de seu parâmetro `arg` e deve expressar a possibilidade de atualizar `estados.qtde_emergencias`:

```
/*@
  @ assigns estados.qtde_emergencias;
  @ behavior enfileirar_emergencia:
  @     assumes arg instanceof Emergency;
  @     ensures estados.qtde_emergencias == \old(estados.qtde_emergencias)
  @                                         + 1;
  @ behavior nao_enfileirar_emergencia:
  @     assumes ! (arg instanceof Emergency);
  @     ensures estados.qtde_emergencias == \old(estados.qtde_emergencias);
  @*/
public void update(Observable o, Object arg) {
...
}
```

Nesta anotação, nota-se a especificação de dois possíveis comportamentos (fluxos) para este método: `enfileirar_emergencia` e `nao_enfileirar_emergencia`. Esses comportamentos do método ocorrem, respectivamente, de acordo com o tipo de seu parâmetro `arg` ser `Emergency` ou não, o que é representado na cláusula `assumes` (pré-condição). A cláusula `ensures` especifica o comportamento do enfileiramento de mais uma emergência, se for o caso.

Detectadas as emergências através do método `update()`, a inclusão de um novo objeto do tipo `RequestConfirmationBehaviour` na lista de comportamentos que o agente `EmergencyAgent` deve possuir é modelada pelo vetor lógico (somente existente nas anotações) `lista_behaviours` e seu controle de tamanho `qt_behaviours` (vide Seção 3.2.2). Essa responsabilidade é delegada ao método `action()`:

```
/*@
  @ requires threadFactory != null;
  @ assigns estados.qtde_emergencias;
  @ behavior emergencia:
```

```
  @     assumes estados.qtde_emergencias >= 1;
  @     ensures
  @         estados.qtde_emergencias == \old(estados.qtde_emergencias) - 1
  @         && myAgent.qt_behaviours == \old(myAgent.qt_behaviours) + 1
  @         && myAgent.lista_behaviours[\old(myAgent.qt_behaviours)]
  @            instanceof ThreadedBehaviourWrapper
  @         && ((ThreadedBehaviourWrapper)
  @            myAgent.lista_behaviours[\old(myAgent.qt_behaviours)])
  @            .myBehaviour instanceof RequestConfirmationBehaviour;
  @ behavior nenhuma_emergencia:
  @     assumes estados.qtde_emergencias == 0;
  @     ensures estados.qtde_emergencias == 0;
  @*/
public void action() {
...
}
```

Nota-se que é pré-condição para a execução do método `action()` que o atributo `threadFactory` esteja inicializado (`threadFactory != null`), o que será assegurado pelo método `onStart()`. A possibilidade de atribuição de `estados.qtde_emergencias` é especificada na cláusula `assigns` do contrato. São especificados também dois fluxos de execução para o método, `emergencia` e `nenhuma_emergencia`, respectivamente sob as pré-condições de haver alguma emergência enfileirada (`estados.qtde_emergencias >= 1`) ou não (`estados.qtde_emergencias == 0`).

Finalmente o método `onStart()`, responsável pela inicialização do comportamento, deve fazer a alocação do objeto `threadFactory`:

```
/*@
  @ ensures threadFactory != null;
  @*/
public void onStart() {
```

A anotação completa da classe `MonitorAcuteEmergencyBehaviour` pode ser encontrada no Apêndice A.1.

3.2.2 Classe `Agent`

A classe `Agent`, componente da API JADE [4, p. 10 *et seq.*], é uma classe base comum a todo agente que venha a ser definido pelo usuário. No projeto verificado, as classes `EmergencyAgent` e `ElderlyAgent` derivam de `Agent` e representam os dois principais agentes participantes do sistema. Esses agentes são instanciados e registrados em `StartAgent`, componente dos pacotes de carga (inicialização) do sistema, fora de objetivo de verificação (não foram alvo de verificação). `MonitorAcuteEmergencyBehaviour` é o comportamento verificado, que por sua vez é registrado para execução por `EmergencyAgent` em tais pacotes de carga, sendo o fato de sua execução admitida.

Não obstante, MonitorAcuteEmergencyBehaviour precisa interagir com o agente que o hospeda (EmergencyAgent) para indicar a necessidade de execução de um novo comportamento, quando da detecção de uma nova emergência. Isso é possível através de objeto que MonitorAcuteEmergencyBehaviour possui internamente (myAgent), e do uso do método addBehaviour() da classe Agent. O objeto myAgent é acessado através de herança (vide Seção 3.2.6).

Anotações

Para representar a lista de comportamentos do agente, modelou-se um vetor, fazendo uso do recurso JML *model fields* [28, p. 11]. Cabe salientar que essa construção existe apenas nas anotações, não fazendo parte do código de fato:

```
//@ model integer qt_behaviours = 0;
//@ model Behaviour [] lista_behaviours;
```

O vetor de comportamentos lista_behaviours, do tipo Behaviour [] (ou seja, comporta elementos do tipo Behaviour), é usado para hospedar (na camada lógica) a informação de cada novo comportamento lançado; a variável qt_behaviours, para contabilizar esse vetor. O método addBehaviour() é então especificado de forma a representar a adição de um elemento a esse vetor:

```
/*@
  @ requires b instanceof ThreadedBehaviourWrapper;
  @ ensures qt_behaviours == \old(qt_behaviours) + 1 &&
  @         lista_behaviours[\old(qt_behaviours)] == b;
  @*/
public void addBehaviour(Behaviour b);
```

O sistema tem como requisito o lançamento de um novo comportamento com execução paralela. De acordo com a documentação de JADE, esse efeito é obtido com o uso de comportamentos encapsulados por um objeto do tipo ThreadedBehaviourWrapper (Seção 3.2.4) e, portanto, é um pré-requisito (cláusula requires) que o comportamento seja desse tipo.

A classe sob estudo apenas lança comportamentos, mas não os remove do contexto, portanto o método removeBehaviour() não foi anotado. Busca-se verificar que os novos comportamentos tenham sido corretamente iniciados apenas.

A anotação completa da classe Agent pode ser encontrada no Apêndice A.2.

3.2.3 Classes usadas para enfileiramento de emergências

A classe ConcurrentLinkedQueue e a *interface* Queue, componentes da API Java SDK [1], são usadas pelo sistema para enfileiramento de emergências, por sua vez representadas pela classe Emergency. Os trechos de código em verificação apenas usam

informações (providas pelos métodos da API) referentes ao fato do enfileiramento e, portanto, as emergências são apenas quantificadas (vide Seção 3.4.3).

Essas classes usam "programação genérica" (vide Seção 3.4.2) e, portanto, foram adaptadas para trabalhar exclusivamente com emergências (classe `Emergency`), resultando nas classes `ConcurrentLinkedQueue_Emergency` e `Queue_Emergency`.

Anotações

O código cliente faz uso dos métodos responsáveis por incluir e remover emergências da fila, respectivamente os métodos `add()` e `poll()` da interface `Queue_Emergency`.

O método `add()` meramente adiciona uma emergência ao contexto:

```
/*@
  @ assigns estados.qtde_emergencias;
  @ ensures estados.qtde_emergencias == \old(estados.qtde_emergencias) + 1;
  @*/
boolean add(Emergency e);
```

O código cliente despreza o retorno de `add()`, assim o retorno do método (que é do tipo `boolean`) não foi especificado.

O método `poll()`, porém, conforme documentação [1], além de retirar uma emergência do contexto, caso a fila esteja vazia, retorna `null` (indicador de objeto não alocado); caso contrário, algo diferente de `null`:

```
/*@
  @ assigns estados.qtde_emergencias;
  @ behavior fila_com_conteudo:
  @     assumes estados.qtde_emergencias >= 1;
  @     ensures \result != null &&
  @             estados.qtde_emergencias == \old(estados.qtde_emergencias)
  @             - 1;
  @ behavior fila_vazia:
  @     assumes estados.qtde_emergencias == 0;
  @     ensures \result == null &&
  @             estados.qtde_emergencias == \old(estados.qtde_emergencias);
  @*/
Emergency poll();
```

Esses dois aspectos do método estão especificados nas anotações acima respectivamente como `fila_com_conteudo` e `fila_vazia`.

A anotação completa da interface `Queue_Emergency` encontra-se no Apêndice A.3.

3.2.4 Classes usadas para geração de novo comportamento

`ThreadedBehaviourFactory` é a classe pertencente à API JADE [4, p. 29 *et seq.*] usada para executar comportamentos JADE em uma *thread* Java dedicada. Qualquer tipo de comportamento JADE (Seção 3.2.6) pode ser executado em uma linha de execução paralela (*threaded*) através da classe `ThreadedBehaviourFactory`.

Para operacionalizar a nova linha de execução, o programador deve alocar um novo objeto do tipo `ThreadedBehaviourFactory` e passar como parâmetro para seu método `wrap()` o comportamento JADE que se deseja executar paralelamente. O método `wrap()` então criará e retornará um novo objeto do tipo `ThreadedBehaviourWrapper`. Ao adicionar esse novo objeto à lista de comportamentos executados por um agente (classe `Agent`), através de seu método `addBehaviour()`, o *framework* JADE perceberá o tipo especial do comportamento (`ThreadedBehaviourWrapper`) e dedicará uma *thread* a ele.

Para troca de mensagens entre comportamentos, é usado o objeto `myStore`, acessado através de herança (vide Seção 3.2.6). Este objeto é do tipo `DataStore`, classe derivada de `HashMap`, componente da API Java SDK [1], que provê um par de métodos (`put()` e `get()`) para salvar e recuperar as informações armazenadas no objeto `myStore`. As informações são acessadas através de um índice (*hash*). Assim o *hash* foi modelado como um vetor, acessado a partir do índice mediante mapeamento entre o índice e uma posição do vetor.

Anotações

Para usar o método `wrap()` de `ThreadedBehaviourFactory`, é necessário (pré-condição) que o objeto do tipo `Behaviour` que esteja sendo passado como parâmetro tenha sido alocado previamente (seja diferente de `null`):

```
/*@
  @ requires b != null;
  @ ensures \result instanceof ThreadedBehaviourWrapper
  @         && \result != null
  @         && ((ThreadedBehaviourWrapper)\result).myBehaviour == b;
  @*/
public Behaviour wrap(Behaviour b);
```

Caso essa pré-condição seja satisfeita, o comportamento esperado do médoto `wrap()` é que ele aloque um novo comportamento em um objeto do tipo `ThreadedBehaviourWrapper` e que seu atributo interno `myBehaviour` aponte para o comportamento fornecido através do parâmetro `b`.

Na API original JADE [4, p. 29], `ThreadedBehaviourWrapper` é uma classe interna à classe `ThreadedBehaviourFactory`, porém, foi criada uma classe exclusiva para acomodá-la (vide Seção 3.4.4), que teve seu construtor anotado como uma mera atribuição ao atributo interno `myBehaviour`:

```
//@ ensures myBehaviour == b;
```

```
private ThreadedBehaviourWrapper(Behaviour b);
```

Para os métodos `put()` e `get()` de `HashMap`, poder-se-ia conectar as anotações JML com alguma implementação lógica genérica, tal como fez Dross *et al.* [15] para o provador Coq (especificação de *container* do tipo *map*). Obstando-se a esta abordagem, uma limitação encontrada no Krakatoa (vide Seção 3.4.5) obrigou o uso de uma especificação restrita ao projeto (não genérica, conforme a referida seção), estabelecendo correspondência entre o *hash* usado pelo agente e um índice fixo:

```
//@ model Object [] vetor;

/*@
  @ ensures key == RequestConfirmationBehaviour.EVENT_KEY
  @         ==> vetor[0] == value;
  @*/
public Object put(Object key, Object value);

/*@
  @ ensures key == RequestConfirmationBehaviour.EVENT_KEY
  @         ==> \result == vetor[0];
  @*/
public Object get(Object key);
```

A cada novo *hash* necessário, nessa abordagem, deve-se estender as pós-condições, incluindo nova tradução entre índice (parâmetro `key`) e posição de `vetor`.

A anotação completa da classe `ThreadedBehaviourFactory`, bem como das classes `ThreadedBehaviourWrapper` e `HashMap` pode ser encontrada nos Apêndices A.4, A.5 e A.6.

3.2.5 Classes usadas para sinalizar emergências

A API Java SDK [1] possui duas classes para implementação de mecanismo de notificação de eventos: `Observable` e `Observer`.

`Observer` é uma *interface* que deve ser "implementada" por classes que desejam ser informadas sobre modificações em objetos observáveis (de tipo, ou seja, da classe `Observable`). Além disso, classes que implementem essa *interface* devem também implementar o método `update()`, que será executado sempre que objetos observados sofrerem alterações.

No sistema, emergências são objetos observáveis e têm seu mecanismo de observação codificado dentro da classe `DetectEmergencyServiceImpl`, derivada de `Observable`. `DetectEmergencyServiceImpl` tem seus procedimentos de instanciação e registro realizados pela classe `StartService`. Ambas as classes, componentes dos pacotes de carga do sistema, estão fora do objetivo de verificação. `MonitorAcuteEmergencyBehaviour` é registrado como um observador de emergências por `DetectEmergencyServiceImpl`. Esse

processo de catalogação de componentes é então admitido, o que implica que as classes do sistema responsáveis por essas inicialização não foram verificadas, não sendo possível afirmar que tal carga do sistema foi feita corretamente.

`MonitorAcuteEmergencyBehaviour`, comportamento verificado, por sua vez implementa a *interface* `Observer`, o que requer anotações referentes ao método `update()` de `Observer`.

Anotações

Uma vez que o mecanismo de armazenamento da informação da quantidade de emergências detectadas envolveu a criação de uma classe dedicada (`estados`), não usada pelo código, mas tão somente pelas anotações (vide Seção 3.4.3), as anotações referentes ao método `update()` concentraram-se na própria classe `MonitorAcuteEmergencyBehaviour` (Seção 3.2.1). A *interface* `Observer` e a classe `Observable` não tiveram anotações adicionais significantes.

3.2.6 Classes herdadas pelos comportamentos

Os comportamentos de usuário, tal como `MonitorAcuteEmergencyBehaviour`, são construídos através da derivação de alguma das classes de comportamento disponíveis na API JADE. Esses comportamentos básicos, por sua vez, derivam da classe abstrata `Behaviour` [4, p. 23].

O sistema possui comportamentos derivados das classes JADE `CyclicBehaviour` e `Behaviour`. A propriedade de herança da orientação a objetos, a qual Java está subordinado, garante que métodos e atributos declarados dentro da classe básica `Behaviour` e que estejam com sua cláusula de visibilidade marcadas como `public` ou `protected` estejam acessíveis às classes derivadas, ou seja, aos comportamentos de usuário.

Dos atributos derivados de `Behaviour`, o sistema manipula direta ou indiretamente (via métodos) os objetos internos `myAgent` e `myStore`. Este último é declarado com visibilidade `private` e, portanto, para ser acessado pelos comportamentos de usuário, necessita de métodos *getter and setter*: `getDataStore()` e `setDataStore()`. Por ser atributo protegido, `myAgent` pode ser acessado diretamente pelas classes derivadas de `Behaviour`.

`ThreadedBehaviourWrapper` também é derivada de `Behaviour`, e é usada no processo de geração de novos comportamentos paralelos (Seção 3.2.4).

Anotações

Um dos construtores de `Behaviour` armazena a informação do agente instanciador do comportamento:

```
/*@
  @ requires a != null;
```

```
  @ ensures myAgent == a;
  @*/
public Behaviour(Agent a);
```

É pré-condição que a informação do agente seja válida (`a != null`).

O método `getDataStore()`, deve então retornar o atributo `myStore`, atributo do tipo `DataStore`, alocado para troca de informações durante a geração de novo comportamento (Seção 3.2.4):

```
//@ ensures \result == myStore && myStore != null;
public DataStore getDataStore();
```

Caso o atributo `myStore` não contenha nenhum objeto, seu conteúdo será `null`, o que indicará que ainda não foi alocado e salvo um objeto nesse atributo. Se for esse o caso, `getDataStore()` deve alocar e salvar um objeto do tipo `DataStore` nesse atribuito. O fato de que o objeto retornado por `getDataStore()` sempre será válido está expresso na pós-condição `myStore != null`, no lado direito da conjunção da cláusula `ensures`, e deve-se a essa propriedade do método de alocar um novo objeto quando, antes da execução do método, `myStore` não for válido.

`setDataStore()` não possui essa responsabilidade de alocação, bastando apenas a atribuição do objeto parametrizado a `myStore`.

```
/*@
  @ requires ds != null;
  @ ensures myStore == ds;
  @*/
public void setDataStore(DataStore ds);
```

Sua única pré-condição é que o parâmetro recebido (`ds`) seja válido (`ds != null`).

A anotação completa da classe base `Behaviour` encontra-se no Apêndice A.7.

3.2.7 Classe `estados`

O mecanismo de armazenamento da informação da quantidade de emergências necessitou (vide Seção 3.4.3) da criação de uma classe dedicada exclusivamente à acomodação de um contador de emergências (atributo `qtde_emergencias`).

Essa classe (`estados`) não é usada pelo código-fonte do projeto em verificação, porém é usada pelas anotações para representar o acúmulo ou consumo de emergências. Esta alternativa não impactou a formalização realizada, pois a classe foi usada em substituição ao recurso *model fields* (Seções 3.2.2 e 3.4.3) meramente para acomodar variáveis e suas consequentes mudanças de estados.

Além do atributo, a classe possui um construtor *default* para garantir a correta inicialização do atributo com valor zero.

Anotações

O construtor precisa de uma anotação para expressar o fato da inicialização:

```
/*@
  @ assigns qtde_emergencias;
  @ ensures qtde_emergencias == 0;
  @*/
estados() {
```

Além do construtor, a classe possui uma invariante que expressa que o atributo, apesar de inteiro, nunca será negativo.

```
//@ invariant qtde_positiva: qtde_emergencias >= 0;
```

Essa invariante é inclusa pelo ambiente de geração de obrigações de prova (Krakatoa) automaticamente como pós-condição partícipe de cada teorema que envolva manipulação desse atributo (Seção 3.3).

A versão completa da classe `estados` foi inserida no Apêndice A.8.

3.2.8 Outras classes

As classes `RequestConfirmationBehaviour` e `DetectEmergencyService` são manipuladas apenas basicamente pela classe principal `MonitorAcuteEmergencyBehaviour`. Nenhum de seus métodos é acessado de forma que interfira no funcionamento da aplicação [4]. Seus atributos são apenas lidos.

Anotações

Assim nenhuma anotação significativa foi feita para essas classes.

3.3 Obrigações de provas

Com o código-fonte Java anotado (Seção 3.2), executou-se o *software* Krakatoa (Seção 2.3) para a classe principal:

```
krakatoa MonitorAcuteEmergencyBehaviour.java
```

Com a execução acima, o Krakatoa descarregou então 71 obrigações de prova a serem provadas. O *software* invoca automaticamente a interface gráfica (`why3ide`) do aplicativo Why (Seção 2.3), a qual centraliza as obrigações de prova geradas (Figura 3.1).

[4] `DetectEmergencyService` tem o método `register()` invocado por `onStart()`, ou seja, durante o processo de inicialização, admitido, conforme explanado nas Seções 3.2.2 e 3.2.5.

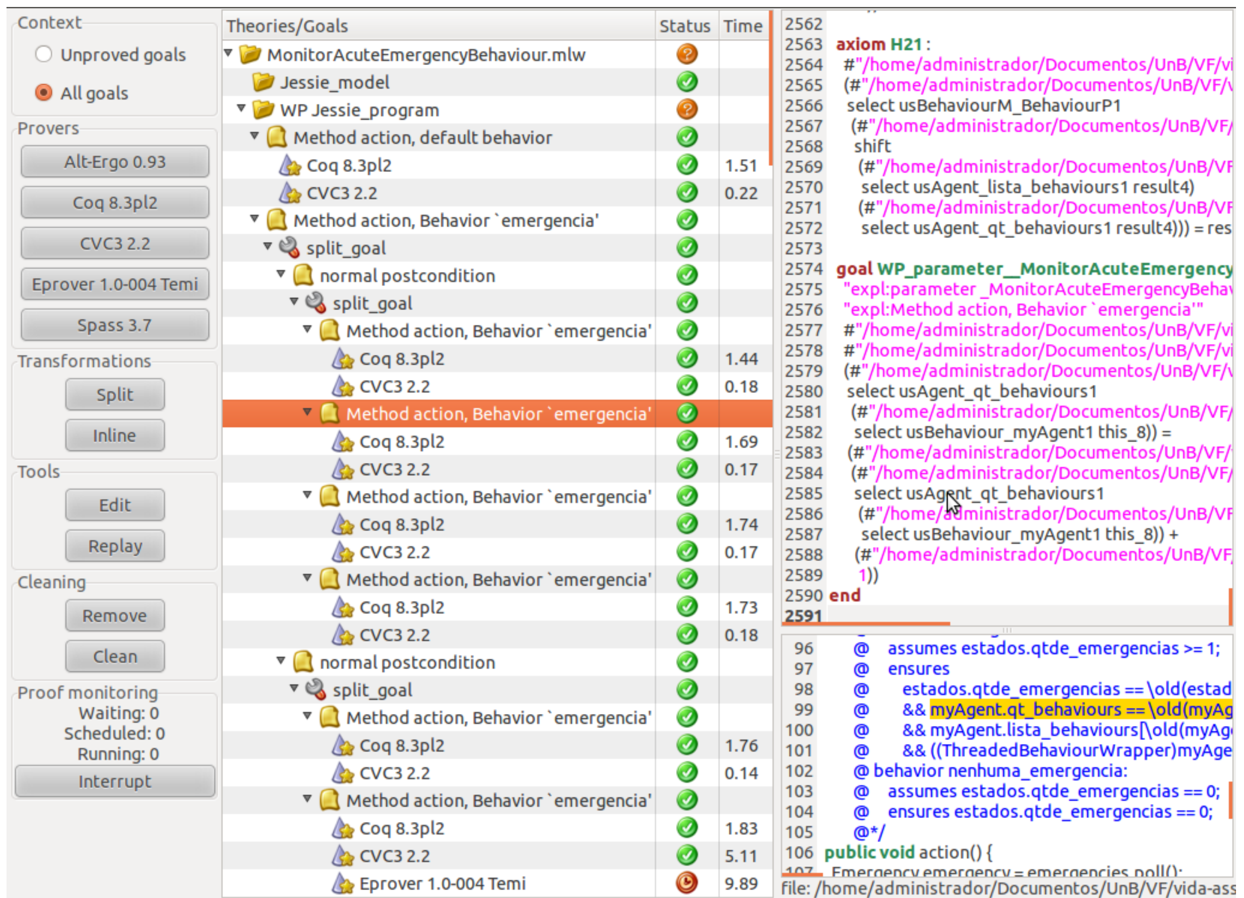


Figura 3.1: Tela do `why3ide` com obrigações de prova geradas pelo Krakatoa

Na barra lateral esquerda, aparecem, dentre outros itens, os provadores instalados (interativos e automáticos) e transformações possíveis para objetivos. À medida que se clica em itens do painel central (obrigações de prova), os dois painéis mais à direita são atualizados: o inferior (direito), com o código-fonte anotado e um realce (amarelo) no trecho da anotação referente ao objetivo de prova; o superior (direito), com o teorema a ser provado, codificado em linguagem Why.

Seleciona-se então um provador no painel esquerdo e o Why transformará o teorema para a linguagem de entrada específica do provador, por exemplo, para a linguagem do provador interativo Coq (Seção 2.4). Além disso, nessa operação, o Why invoca o provador escolhido fornecendo como entrada o arquivo traduzido, abrindo o ambiente de prova interativo, se for o caso. Caso seja selecionado um provador automático, a tradução e disparo serão transparentes, ou seja, não será exibido *front-end* do provador, por padrão.

Nesta seção, algumas obrigações de prova serão analisadas, sempre na linguagem do provador interativo Coq (Seção 2.4). Será usado o acrônimo OP, em vez da expressão "obrigação de prova".

Os arquivos Coq contendo as respectivas OPs são precedidos de preâmbulo contendo axiomatização referente ao modelo de memória de programas Java, modelagem descrita por Marché [32, p. 12 *et seq.*]. Essa axiomatização será suprimida dos *scripts* apresentados nesta seção, contudo pode ser encontrada no Anexo I.

3.3.1 Objetivos de prova principais

Das 71 OPs geradas, 19 apresentaram teoremas não triviais. A Tabela 3.1 exibe um sumário do estado de prova de 16 desses objetivos, onde:

- a primeira coluna corresponde ao número da prova, para referência neste texto;
- a segunda coluna contém o nome da OP, gerada pelo Why;
- as colunas subsequentes referem-se ao estado da prova em cada provador, podendo apresentar uma das seguintes informações:
  - tempo de verificação de uma prova, caso a coluna possua informação numérica;
  - falha na tentativa de prova, caso a coluna possua coloração rosa;
  - objetivo não tentado, caso a coluna possua coloração roxa.

Nota-se que o provador automático CVC3 provou praticamente todos os objetivos, com exceção do objetivo #11, que não foi provado por nenhum provador automático testado. Para esse objetivo de prova, foi necessária a construção de uma prova interativa, com o auxílio do Coq.

Comentam-se a seguir algumas dessas 16 OPs.

OPs #02 a #09

Estes objetivos referem-se às anotações do método `action()` da classe principal (Seção 3.2.1). Naquela anotação, haviam sido especificadas duas seções *behavior* em JML: `emergencia` e `nenhuma_emergencia`. Como pode-se ver na Tabela 3.1, essas OPs referem-se às anotações da seção `emergencia`.

Em verdade, as ferramentas produziram apenas um OP: `Method action, Behavior ‘emergencia’`. Não obstante, a interface gráfica do Why permite a aplicação da transformação `Split`, que basicamente busca conjunções na implicação final do teorema a ser provado, dividindo o teorema em tantos outros teoremas quantas forem as conjunções integrantes.

Pela Tabela 3.1 e Figura 3.1, percebe-se então que aplicou-se uma vez a transformação à OP original, produzindo dois novos teoremas (nomeados ambos de `normal postcondition`) e, em cada um desses novos teoremas, aplicou-se novamente a transformação o que produziu dois conjuntos de quatro novos teoremas, dessa vez, numerados pela ferramenta.

“`Method action, Behavior ‘emergencia’`”, a OP produzida originalmente pela ferramenta, não pôde ser provada diretamente por nenhum provador automático. Após as transformações, o provador CVC3 provou os oito teoremas.

Para a prova no Coq, a transformação tem pouco efeito, pois um simples uso do *tactical* “`repeat split; intros`” gera os oito `subgoals` correspondentes às OPs transformadas. O único efeito prático foi a geração de oito arquivos Coq separados, cada um para sua respectiva OP.

#	Proof obligations	CVC3	Coq	Eprover	Spass
01	*Method action, default behavior*	0.22	1.51		
	Method action, Behavior 'emergencia'				
	normal postcondition				
02	*1*	0.18	1.44		
03	*2*	0.17	1.69		
04	*3*	0.17	1.74		
05	*4*	0.18	1.73		
	normal postcondition				
06	*1*	0.14	1.76		
07	*2*	5.11	1.83	◷	◷
08	*3*	5.34	1.98	◷	◷
09	*4*	5.23	1.70	◷	◷
10	*Method action, Behavior 'nenhuma_emergencia'*	0.15	2.08		
11	*normal postcondition*	◷	1.63	◷	◷
12	*Method onStart, Safety*	0.11			
13	*Method update, default behavior*	0.11			
14	*Method update, Behavior 'enfileirar_emergencia'*	0.12	1.49		
15	*normal postcondition*	0.11	1.51		
16	*downcast*	0.09	1.54		

Tabela 3.1: Objetivos de prova principais

OP #06

A OP #06, após a aplicação da tática `intros` (introdução das implicações e quantificadores universais), apresenta o seguinte formato:

```
Theorem
  WP_parameter_...._action_ensures_emergencia :
...
... (SUPRIMIDO)
...
1 subgoal
this_8 : pointer usObject
estados_qtde_emergencias1 : int32
usMonitorAcuteEmergencyBehaviour_threadFactory1 : memory usObject
                                                       (pointer usObject)
usObject_alloc_table1 : alloc_table usObject
H : 1 <= integer_of_int32 estados_qtde_emergencias1 /\
    valid_struct_Object this_8 0 0 usObject_alloc_table1 /\
    usNon_null_Object
      (select usMonitorAcuteEmergencyBehaviour_threadFactory1 this_8)
```

```
      usObject_alloc_table1
estados_qtde_emergencias2 : int32
result : pointer usObject
H0 : (integer_of_int32 estados_qtde_emergencias1 = 0 ->
      result = (null:pointer usObject) /\
      integer_of_int32 estados_qtde_emergencias2 =
      integer_of_int32 estados_qtde_emergencias1) /\
     (1 <= integer_of_int32 estados_qtde_emergencias1 ->
      usNon_null_Object result usObject_alloc_table1 /\
      integer_of_int32 estados_qtde_emergencias2 =
      integer_of_int32 estados_qtde_emergencias1 - 1)
result1 : bool
H1 : result1 = true /\ offset_max usObject_alloc_table1 result = 0 \/
     result1 <> true /\ result = (null:pointer usObject)
H2 : result1 <> true
______________________________________(1/1)
integer_of_int32 estados_qtde_emergencias2 =
integer_of_int32 estados_qtde_emergencias1 - 1
```

Essa OP origina-se de uma das proposições integrantes da cláusula `ensures` (pós-condição) do referido método `action()`:

```
estados.qtde_emergencias == \old(estados.qtde_emergencias) - 1
```

Nota-se que os diversos estados das variáveis são traduzidos para o Coq como diferentes variáveis, numeradas de acordo com o fluxo do código imperativo. Em especial:

- `\old(estados.qtde_emergencias)` (conteúdo da variável imediatamente antes da chamada ao método) foi traduzido como `estados_qtde_emergencias1`;
- `estados.qtde_emergencias` (conteúdo da variável imediatamente após o encerramento do método), como `estados_qtde_emergencias2`.

Além disso, o contexto (premissas) de prova foi produzido tendo como base um cálculo da "pré-condição mais fraca" (Seção 2.1) implementado pela ferramentas. Para tanto, a ferramenta considerou:

- o código-fonte Java do método (Apêndice A.1);
- as anotações JML feitas para cada uma das dependências funcionais desse código imperativo (em especial, `Queue_Emergency.poll()`);
- além das pré-condições JML especificadas para o próprio método `action()`.

Após examinar o contexto, nota-se que manipulações simples envolvendo as conjunções contidas nas hipóteses H0 e H conduzem à prova do teorema. O *script* Coq de construção da prova reflete essas observações:

```
Proof.
  intros.
  elim H0; clear H0; intros.
  elim H3; clear H3; intros.
    assumption.

    elim H; clear H; intros.
    assumption.
Qed.
```

OP #07

A OP #07, após a aplicação da tática `intros`:

```
Theorem
  WP_parameter__MonitorAcuteEmergencyBehaviour_action_ensures_emergencia :
...
... (SUPRIMIDO)
...
1 subgoal
this_8 : pointer usObject
estados_qtde_emergencias1 : int32
usMonitorAcuteEmergencyBehaviour_threadFactory1 : memory usObject
                                                    (pointer usObject)
usBehaviour_myAgent1 : memory usObject (pointer usObject)
usAgent_qt_behaviours1 : memory usObject Z
usObject_alloc_table1 : alloc_table usObject
H : 1 <= integer_of_int32 estados_qtde_emergencias1 /\
    valid_struct_Object this_8 0 0 usObject_alloc_table1 /\
    usNon_null_Object
      (select usMonitorAcuteEmergencyBehaviour_threadFactory1 this_8)
      usObject_alloc_table1
estados_qtde_emergencias2 : int32
result : pointer usObject
H0 : (integer_of_int32 estados_qtde_emergencias1 = 0 ->
      result = (null:pointer usObject) /\
      integer_of_int32 estados_qtde_emergencias2 =
      integer_of_int32 estados_qtde_emergencias1) /\
     (1 <= integer_of_int32 estados_qtde_emergencias1 ->
      usNon_null_Object result usObject_alloc_table1 /\
      integer_of_int32 estados_qtde_emergencias2 =
      integer_of_int32 estados_qtde_emergencias1 - 1)
result1 : bool
H1 : result1 = true /\ offset_max usObject_alloc_table1 result = 0 \/
     result1 <> true /\ result = (null:pointer usObject)
H2 : result1 <> true
```

```
____________________________________(1/1)
select usAgent_qt_behaviours1 (select usBehaviour_myAgent1 this_8) =
select usAgent_qt_behaviours1 (select usBehaviour_myAgent1 this_8) + 1
```

A OP advém de outra parte da pós-condição do método `action()`:

```
myAgent.qt_behaviours == \old(myAgent.qt_behaviours) + 1
```

Dessa vez, a variável deve ser acessada através de um atributo interno (`myAgent`) à classe `Behaviour`. Esse atributo é alocado dinamicamente, portanto o acesso às variáveis envolvidas não é mais direto, mas sim representado pelo uso da função `select`, pertencente à axiomatização citada (Anexo I).

Porém, nota-se que H0 é proveniente da extração das anotações do método `poll()` [5] (Apêndice A.3). Além disso, H1 representa as possibilidades do comando `if` constante do corpo de `action()`:

```
public void action() {
        Emergency emergency = emergencies.poll();
        if (emergency != null) {
...
...
        }
}
```

Note que `result1` (em H1) é booleano que equivale exatamente à expressão `emergency != null`. Por H2, percebe-se que essa OP corresponde à extração de comportamento em que o `if` falha: `result1 <> true`, ou `result1 = false`, ou `emergency = null`.

Além disso, em H0, `result` representa exatamente o retorno do método `poll()` que, no código acima, é armazenado em `emergency`.

Por outro lado, percebe-se pelas anotações de `action()` que a seção *behavior* `emergencia` tem como pré-condição (cláusula `assumes`) que `estados.qtde_emergencias >= 1`, o resultado de `poll()` deveria conduzir a `emergency != null`.

De fato, em H0, a asserção `usNon_null_Object result usObject_alloc_table1` ratifica a contradição no contexto, ou seja, um fluxo imperativo que não ocorre nessas condições de estado das variáveis envolvidas.

Voltando a atenção novamente para o objetivo, constata-se que seria impossível construir uma prova direta, pois implicaria no seguinte teorema ser provado (note o formato do objetivo do teorema):

```
forall x:Z, x = x + 1
```

O *script* Coq de construção da prova trabalha em busca de recompor no ambiente do provador essas contradições, fechando a OP:

[5] `poll()` é método usado no corpo do método `action()`

```
Proof.
  intros.
  elim H1; clear H1; intros.
    elim H1; clear H1; intros.
    contradiction.

  elim H1; clear H1; intros.
  elim H0; clear H0; intros.
  elim H4; clear H4; intros.
    unfold usNon_null_Object in H4.
    rewrite H3 in H4.
    generalize null_pointer; intros.
    elim (H6 usObject usObject_alloc_table1); clear H6; intros.
    generalize
      (Zle_trans 0 (offset_max usObject_alloc_table1 null) (- (2)) H4 H7);
      intros.
    info auto with zarith.

  elim H; clear H; intros.
  assumption.
Qed.
```

OP #10

À OP #10, assim como a todas as outras OPs, não é aplicada nenhuma transformação Why (tal como a transformação `Split` mencionada anteriormente). Essa OP é denominada pela ferramenta `Method action, Behavior 'nenhuma_emergencia'`, pois refere-se às anotações do método `action()`, *behavior* (seção da anotação JML) `nenhuma_emergencia`.

Após a aplicação da tática `intros` e `repeat split; intros`, o teorema apresenta dois subobjetivos, dos quais reproduz-se apenas parte do contexto do primeiro subobjetivo:

```
Theorem
  WP_parameter__..._action_ensures_nenhuma_emergencia :
...
... (SUPRIMIDO)
...
2 subgoals
...
... (SUPRIMIDO)
...
H : integer_of_int32 estados_qtde_emergencias1 = 0 /\
    valid_struct_Object this_8 0 0 usObject_alloc_table1 /\
    usNon_null_Object
      (select usMonitorAcuteEmergencyBehaviour_threadFactory1 this_8)
      usObject_alloc_table1
```

```
estados_qtde_emergencias2 : int32
result : pointer usObject
H0 : (integer_of_int32 estados_qtde_emergencias1 = 0 ->
      result = (null:pointer usObject) /\
      integer_of_int32 estados_qtde_emergencias2 =
      integer_of_int32 estados_qtde_emergencias1) /\
     (1 <= integer_of_int32 estados_qtde_emergencias1 ->
      usNon_null_Object result usObject_alloc_table1 /\
      integer_of_int32 estados_qtde_emergencias2 =
      integer_of_int32 estados_qtde_emergencias1 - 1)
result1 : bool
H1 : result1 = true /\ offset_max usObject_alloc_table1 result = 0 \/
     result1 <> true /\ result = (null:pointer usObject)
H2 : result1 = true
...
... (SUPRIMIDO)
...
H5 : usRequestConfirmationBehaviour_EVENT_KEY =
     usRequestConfirmationBehaviour_EVENT_KEY ->
     select usObjectM_ObjectP1 (shift (select usHashMap_vetor1 result3) 0) =
     result
...
... (SUPRIMIDO)
...
______________________________________(1/2)
integer_of_int32 estados_qtde_emergencias2 = 0


______________________________________(2/2)
integer_of_int32 estados_qtde_emergencias2 = 0
```

A OP provém do único predicado constante da pós-condição dessa seção de anotação do método:

`estados.qtde_emergencias == 0`

Os dois subobjetivos referem-se aos fluxos possíveis do comando `if` constante do corpo de `action()`: o primeiro subobjetivo, por H2 (`result1 = true`), indica o caso `then`; e o segundo subobjetivo possui hipótese que indica (`result1 <> true`) o caso `else`.

Nos dois casos, não há a necessidade de recorrer a axiomas ou lemas do prelúdio (teoria carregada dentro do *script* pelas ferramentas). O aproveitamento de simples fatos (hipóteses) do contexto, como H e H0, permite a prova dos subobjetivos, o que pode ser visto no *script* de construção da prova:

```
Proof.
  intros.
  repeat split; intros.
```

```
    elim H; clear H; intros.
    elim H0; clear H0; intros.
    elim H0; clear H0; intros.
      rewrite H10.
      assumption.

      assumption.

    elim H; clear H; intros.
    elim H0; clear H0; intros.
    elim H0; clear H0; intros.
      rewrite H5.
      assumption.

      assumption.
Qed.
```

OP #11

A OP #11 não pôde ser fechada por nenhum provador automático, tendo sido mandatória a construção de uma prova interativa, no caso, com o Coq.

Esta OP foi denominada pela ferramenta `normal postcondition`, nome padrão, uma vez que refere-se ao método `onStart()`, o qual não foi especificado com as cláusulas *behavior*, dessa forma, não possuindo uma seção nomeada na anotação.

Após a aplicação da tática `intuition`, o teorema apresenta um único objetivo:

```
Theorem
  WP_parameter__MonitorAcuteEmergencyBehaviour_onStart_ensures_default :
...
... (SUPRIMIDO)
...
1 subgoal
this_16 : pointer usObject
usMonitorAcuteEmergencyBehaviour_threadFactory1 : memory usObject
                                                      (pointer usObject)
usObject_alloc_table1 : alloc_table usObject
H : valid_struct_Object this_16 0 0 usObject_alloc_table1
usObject_tag_table1 : tag_table usObject
usObject_alloc_table2 : alloc_table usObject
result : pointer usObject
H1 : strict_valid_struct_Object result 0 (1 - 1) usObject_alloc_table2
H0 : alloc_extends usObject_alloc_table1 usObject_alloc_table2
H2 : alloc_fresh usObject_alloc_table1 result 1
H4 : instanceof usObject_tag_table1 result usThreadedBehaviourFactory_tag
usMonitorAcuteEmergencyBehaviour_threadFactory2 : memory usObject
```

```
                                                        (pointer usObject)
H3 : usMonitorAcuteEmergencyBehaviour_threadFactory2 =
     store usMonitorAcuteEmergencyBehaviour_threadFactory1 this_16 result
______________________________________(1/1)
usNon_null_Object
  (select usMonitorAcuteEmergencyBehaviour_threadFactory2 this_16)
  usObject_alloc_table2
```

Esse objetivo concerne à única pós-condição do método `onStart()`:

```
ensures threadFactory != null
```

Para a consecução da prova, é importante notar um dos fatos inclusos pela axiomatização a cerca de armazenamento e recuperação de informação dinâmica (ponteiros):

```
Axiom select_store_eq : forall (t:Type) (v:Type), forall (m:(memory t v)),
  forall (p1:(pointer t)), forall (p2:(pointer t)), forall (a:v),
  (p1 = p2) -> ((select (store m p1 a) p2) = a).
```

Esse axioma deve assim ser interpretado: se dois ponteiros `p1` e `p2` são iguais, então, caso seja armazenado (`store`) algum valor `a` dentro da região de memória `m` na posição `p1` e, posteriormente, caso seja recuperado (`select`) o valor armazenado nesta mesma região de memória `m` na posição `p2` (igual a `p1`, por hipótese), o valor recuperado será o próprio valor `a`, armazenado inicialmente.

Além disso, observa-se que H3 representa modificação do estado de `threadFactory` (variável presente no código-fonte imperativo). Percebendo-se que esse armazenamento ocorre na mesma posição (`this_16`) em que o objetivo recupera a informação, pode-se rescrever H3 no objetivo e aplicar o teorema, simplificando-o para:

```
usNon_null_Object result usObject_alloc_table2
```

Para a prova deste objetivo, tem-se que expandir (`unfold` em Coq) as definições de `strict_valid_struct_Object` (veja hipótese H1) e `usNon_null_Object` (objetivo), também constantes da axiomatização.

Por fim, a aplicação do axioma `select_store_eq` gerou um segundo subobjetivo, o qual exige que se mostre que as posições envolvidas (ponteiros) são iguais, ou seja, que `this_16 = this_16`, resolvido de maneira trivial.

O *script* de construção da prova reflete essas observações:

```
Proof.
  intuition.
  subst usMonitorAcuteEmergencyBehaviour_threadFactory2.
  simpl in H1.
  rewrite select_store_eq.
    unfold strict_valid_struct_Object in H1.
```

```
    unfold usNon_null_Object.
    elim H1; clear H1; intros.
    rewrite H3.
    auto with zarith.

    trivial.
Qed.
```

OP #16

As ferramentas envolvidas também geram OPs referentes à segurança dos métodos. Uma delas (OP #16) refere-se à segurança da operação Java denominada *downcast*.

A operação de *downcast* é descrita formalmente por Pierce [37, p. 215] como um *ascription*, ou seja, uma conversão entre tipos relacionados dentro de uma hierarquia de subtipos [6]. Pierce atenta ainda para o fato de que os *downcasts*, em oposição aos *upcasts*, representam a conversão de um tipo para um subtipo seu, o que não necessariamente sucede e, por isso, em seu formalismo (assim como em Java), não são verificados estaticamente, necessitando uma verificação de tipos em tempo de execução [7].

A OP #16, de fato nomeada `downcast` pela ferramenta, após aplicação da tática `intros`, apresenta-se como abaixo:

```
Theorem
  WP_parameter__MonitorAcuteEmergencyBehaviour_update_safety :
...
... (SUPRIMIDO)
...
1 subgoal
this_4 : pointer usObject
o_0 : pointer usObject
arg_0 : pointer usObject
usObject_tag_table1 : tag_table usObject
usObject_alloc_table1 : alloc_table usObject
H : left_valid_struct_Object arg_0 0 usObject_alloc_table1 /\
    left_valid_struct_Object o_0 0 usObject_alloc_table1 /\
    valid_struct_Object this_4 0 0 usObject_alloc_table1
result : bool
H0 :result = true /\ instanceof usObject_tag_table1 arg_0 usEmergency_tag
    \/
   result <> true /\ ~instanceof usObject_tag_table1 arg_0 usEmergency_tag
```

[6] Em Java, esta hieraquia refere-se ao relacionamento de **herança** (derivação) entre as classes, onde pelo menos duas classes são partícipes: superclasse e subclasse, respectivamente, tipo e subtipo.

[7] Em Java, essa verificação em tempo de execução é feita pela JVM (*Java Virtual Machine*). Analogamente, no formalismo de Pierce, os *downcasts* são tratados por regras de *evaluation* (análise dinâmica), não sendo, dessa forma, tratados pelas regras de tipagem (análise estática).

```
H1 : result = true
______________________________________(1/1)
instanceof usObject_tag_table1 arg_0 usEmergency_tag
```

Esse objetivo, como outros objetivos gerados automaticamente pela ferramenta e rotulados com sufixo *safety*, não se refere a uma anotação, mas sim à verificação da segurança do *downcast* (*ascribe*) feito dentro do método `update()`:

```
public void update(Observable o, Object arg) {
        if (arg instanceof Emergency) {
                emergencies.add((Emergency) arg);
        }
}
```

No caso, a conversão sendo verificada é `(Emergency) arg`, ou seja, a conversão do argumento `arg` do tipo `Object` (superclasse) para o tipo `Emergency` (subclasse).

A prova desse objetivo decorre do fato de a conversão estar corretamente protegida dentro do comando `if` que, justamente, testa, antes de executar a conversão, se `arg` é de fato um objeto (instância) da classe `Emergency`.

Além disso, a prova trabalha no sentido de aproveitar a informação extraída do comando `if` (hipótese H0) e da premissa de que o teste ocorre quando o `if` sucede ("caso `then`", reflexo na hipótese H1).

Assim, o *script* de construção da prova "elimina" a disjunção de H0 gerando os dois casos do `if`: no "caso `then`", o fato virá do contexto; no "caso `else`", haverá uma contradição no contexto.

```
Proof.
  intros.
  elim H0; clear H0; intros.
    elim H0; clear H0; intros.
    assumption.

    elim H0; clear H0; intros.
    contradiction.
Qed.
```

3.3.2 Objetivos triviais

Ainda, do total de 71 obrigações de prova, 52 eram objetivos triviais, tais como simples tautologias. Para esses objetivos, empregou-se exclusivamente provadores automáticos; no caso, o CVC3.

Esses objetivos encontram-se sumarizados nas Tabelas 3.2 e 3.3. Nestas e nas próximas tabelas, onde se lê `CC`, entenda-se como `Constructor of class`, ou seja, objetivos de prova gerados para construtores *default* das classes envolvidas.

3.3.3 Objetivos não provados

Três OPs, relacionadas na Tabela 3.4, não foram provadas.

As OPs #01 e #02 foram geradas indevidamente pelas ferramentas devido ao problema descrito na Seção 3.4.7.

A terceira OP refere-se à segurança do método `action()`. Como explanado na Seção 3.2.1, `onStart()` e `action()` são métodos abstratos definidos na classe *Behaviour*, da qual todos os comportamentos de usuário derivam, inclusive a classe principal (`MonitorAcuteEmergencyBehaviour`). Pelo fato de serem abstratos, a implementação desses métodos fica a cargo das classes que derivam de *Behaviour*. Além disso, o *framework* Jade responsabiliza-se por disparar em **tempo de execução** o método `onStart()` antes que qualquer chamada ao método `action()` ocorra.

A OP #03 carece justamente de que seja expressado esse fato que ocorre em tempo de execução, de forma que a informação esteja disponível em tempo de prova: todo novo objeto derivado de *Behaviour* instanciado terá seu método de alocação invocado antes de seu método de ação.

JML permite que os contratos contenham anotações referentes a execução de métodos [28, p. 6] [11, p. 5], porém o Krakatoa usa uma versão modificada de JML [8] que não permite expressar chamadas a construtores ou métodos [33, p. 23] [32, p. 11].

Uma alternativa para contornar a ausência do construto que expresse encadeamento entre métodos ainda não havia sido encontrada pelo autor deste estudo no momento da confecção desta dissertação.

3.4 *Feedback* do uso das ferramentas

Durante a construção das especificações JML (Seção 3.2) e das obrigações de provas (Seção 3.3), algumas questões surgiram como pontos de atenção ou, até mesmo, levando a alguma adequação na estratégia de anotação ou prova. Documentam-se aqui estas questões.

3.4.1 *Workspace* Java

Projetos Java com um grande número de classes/arquivos fazem uso de uma hierarquia de pastas, dividindo o projeto em agrupamentos de acordo com a modelagem de progra-

[8] A versão de JML usada pelo Krakatoa, por vezes, é denominada em sua documentação de KML.

mação empregada. Essa estrutura de pastas, com seus respectivos arquivos, é denominada *workspace* pelas ferramentas de desenvolvimento Java.

O sistema objeto do estudo de caso do presente trabalho não é diferente: além dos 109 arquivos ".java", há 61 diretórios no *workspace* somente para acomodar o código-fonte.

Workspaces Java, além de serem usados pelos programadores como estrutura organizacional de projeto, também evitam colisão entre nomes de diferentes versões de uma mesma classe [9]. No caso em estudo, apesar de as classes selecionadas para especificação não terem mais de uma versão, diversas outras classes do *workspace* possuem coincidência de nomes [10].

Assim, a primeira tentativa de verificação das propriedades especificadas manteve os arquivos em sua estrutura original. Obstando-se a essa alternativa, encontrou-se um mau funcionamento no processamento das diretivas `package` e `import` [11], presentes, como era de se esperar, no código-fonte do projeto.

O sistema possui duas classes interdependentes: `Behaviour` e `Agent`. Assim, uma "importa" a outra.

Apesar de essa ser uma construção válida em Java, o Krakatoa entra em um ciclo infinito ao tentar processar essa dependência. Em verdade, um simples caso de teste como o abaixo reproduz essa situação:

1. crie os diretórios "p1" e "p2";
2. dentro do diretório "p1", crie o arquivo "Classe1.java" com o seguinte conteúdo:

   ```
   package p1;
   import p2.Classe2;
   public class Classe1 {}
   ```

3. dentro do diretório "p2", crie o arquivo "Classe2.java" com o seguinte conteúdo:

   ```
   package p2;
   import p1.Classe1;
   public class Classe2 {}
   ```

4. execute, por exemplo, "`krakatoa p1/Classe1.java`";
5. interrompa a execução (infinita) e inspecione o arquivo de log (`krakatoa.log`):

[9] Essas diferentes versões de uma mesma classe não se referem a diferentes estágios do processo de desenvolvimento de uma classe, mas sim a diferentes visões de uma mesma classe dentro do projeto, ou seja, diferentes colocações funcionais, o que se considera uma forma de polimorfismo.

[10] No sistema estudado, apesar de haver 109 arquivos ".java", há somente 79 nomes de arquivos ".java", caso descontem-se as repetições de nomes. `GetNameBean.java` e `GreetingAction.java` são exemplos de coincidências de nomes de classes.

[11] As diretivas `package` e `import` são usadas para sinalizar ao compilador Java (`javac`), respectivamente, onde os arquivos das classes serão encontrados e quais as dependências interclasses [2].

```
...
...
...
adding java file Classe1 (fullname ./p1/Classe1.java)
importing package java.lang
importing p2.Classe2
importing package java.lang
importing p1.Classe1
importing package java.lang
importing p2.Classe2
importing package java.lang
importing p1.Classe1
importing package java.lang
importing p2.Classe2
importing package java.lang
importing p1.Classe1
importing package java.lang
importing p2.Classe2
...
...
...
```

O contorno usado para continuar a ação de verificação foi desativar as diretrizes `package` e `import` no código-fonte (envolvendo-as com "comentários") e manter todos os arquivos do projeto (usados na verificação) em um único diretório. Como não houve colisão entre os nomes das classes envolvidas, a estratégia sucedeu.

3.4.2 Programação genérica

Uma das construções suportadas pela linguagem Java é denominada *generics* ou "programação genérica" [2]. Basicamente a construção permite que, na definição de uma classe, seja incluído um parâmetro que represente um tipo. A intenção é que seja possível declarar uma classe como `Classe<T>`, em que T é um parâmetro que será usado no corpo da classe onde normalmente são inseridos tipos; posteriormente, instanciações como `Classe<bool> c` ou `Classe<int> c`, possibilitarão que `Classe` comporte-se exatamente como declarada, a menos das respectivas substituições, nesses exemplos, `T == bool` ou `T == int`.

Este tipo de comportamento polimórfico é formalizado por Pierce [37, p. 361 *et seq.*], que o denomina **polimorfismo paramétrico**. A formalização ali apresentada tenciona a inclusão de *universal types*, resultando em um sistema tradicionalmente denominado *System F*.

O caso em estudo faz uso da *interface* `Queue` e da classe `ConcurrentLinkedQueue`, declarados respectivamente como `Queue<E>` e `ConcurrentLinkedQueue<E>`. Assim, fazem uso da programação genérica Java.

Por outro lado, Krakatoa não suporta programação genérica Java [32, p. 16].

A solução dada para criar um ambiente de anotação para estas classes sem interferir em seu comportamento foi a de operar o polimorfismo manualmente, ou seja, criar arquivos com o código original replicado, substituindo manualmente o parâmetro `E` pelos tipos de fato usados no código. Para cada tipo usado, ter-se-ia que criar um arquivo, porém a classe principal selecionada para verificação instancia-os apenas com o tipo `Emergency`.

Dessa forma, `Queue_Emergency` e `ConcurrentLinkedQueue_Emergency` foram criadas, onde cada ocorrência do parâmetro `E`, constante dos arquivos originais, foi substituída pelo tipo (classe) `Emergency`.

3.4.3 *Model fields*

Por vezes, o processo de anotação requer a modelagem de alguma estrutura de dados específica. JML provê um mecanismo de anotação denominado *model fields*, o qual possibilita criar variáveis existentes apenas no escopo das anotações, de forma que esses campos não podem ser acessados pelo código Java [28, p. 11 *et seq.*].

Esse mecanismo pode ser usado para prover abstrações do estado concreto de um objeto, onde cada abstração pode denotar diferentes aspectos de um mesmo objeto [11, p. 10].

Esse tipo de construção foi usado em dois pontos das anotações:

- para a classe `Agent`, quando foi usada para modelar um vetor de comportamentos e manter atualizada a informação sobre o tamanho do vetor:

  ```
  //@ model integer qt_behaviours = 0;
  //@ model Behaviour [] lista_behaviours;
  ```

- para a classe `HashMap`, onde foi empregada para especificar um vetor que acomoda objetos de qualquer tipo (classe `Object` [12]):

  ```
      //@ model Object [] vetor;
  ```

Tentou-se também usar *model fields* para abstrair uma fila de dados, intentando construir uma anotação genérica para a *interface* `Queue_Emergency`, como as anotações em `Agent` e `HashMap`. Porém, ao tentar gerar as obrigações de prova, a ferramenta apresentou um mau funcionamento:

```
File "java/java_typing.ml", line 1109, characters 8-8:
Fatal error: exception Assert_failure("java/java_typing.ml", 1109, 8)
```

Esse erro refere-se a uma exceção inserida dentro do código-fonte do Krakatoa/Why, especificamente no arquivo `java_typing.ml`, conforme excerto do código original (escrito em Objective Caml) abaixo:

[12]Em Java, a classe `Object` é a superclasse de todas as classes [2]. Pierce formalizou essa noção através da constante `Top` como o elemento máximo da relação de subtipo [37, p. 185].

```
   1093 and type_term_field_access t loc id =
   1094   match t.java_term_type with
   1095     | JTYclass(_,c) ->
...
... (SUPRIMIDO)
...
   1108     | JTYinterface _ii ->
   1109         assert false (* TODO *)
   1110     | JTYarray _ ->
...
... (SUPRIMIDO)
...
   1120     | JTYnull | JTYbase _ | JTYlogic _ ->
```

Analisando a linha mencionada (1109), constata-se que faz parte do mecanismo do Krakatoa de tipagem de termos Java durante acesso a campos (*fields*). Além disso, a referida linha é executada quando o termo Java for do tipo `interface` e, de fato, há uma asserção gerando uma falha com um comentário [13] indicando que esse trecho ainda não foi totalmente desenvolvido.

Abaixo o conteúdo do arquivo `C.java` que, após executado com `krakatoa C.java`, reproduz o problema:

```
interface I { /*@ model int a; @*/ }

public class C implements I { }
```

Então, como o recurso (*model fields*) não se encontrava disponível para uso com `interfaces`, além de não se intentar fazer manipulações do código-fonte original que mudassem seu comportamento, optou-se por criar uma nova classe, a qual serviu meramente para acomodar variáveis de estado a serem usadas nas anotações, portanto não sendo referenciada pelo sistema em verificação.

Dessarte simulou-se a funcionalidade *model fields* para `interfaces` através da classe `estados` (Seção 3.2.7).

3.4.4 Classes internas

A linguagem Java permite a definição de uma classe dentro de outra, contrução denominada de "classe interna" (ou "classe aninhada"). Esse recurso é usado para agrupar logicamente classes em um único arquivo, aumentando o encapsulamento [14] [2]. Declarações desse tipo ocorrem com estruturas análogas à que segue:

[13] O comentário em questão é `TODO` comumente usado por programadores de língua inglesa para indicar um "afazer", isto é, algo ainda a ser programado.

[14] Encapsulamento, conceito existente em linguagens orientadas a objeto, refere-se a ocultar os estados internos (variáveis) de objetos (instâncias de classes) e requerer que toda a interação com esses objetos seja realizada através de métodos, que, por sua vez, farão a devida atualização desses estados [2].

```
class ClasseExterna {
    ...
    class ClasseInterna {
        ...
    }
}
```

O sistema faz uso da classe `ThreadedBehaviourFactory`, integrante da API JADE, para geração de novos comportamentos (Seção 3.2.4). Essa classe define também a classe interna `ThreadedBehaviourWrapper`.

Realizadas as anotações no código original (`ThreadedBehaviourFactory.java`), executou-se o Krakatoa para gerar as obrigações de prova, porém a ferramenta não reconheceu a declaração da classe interna:

```
File "./ThreadedBehaviourFactory.java", line 95, characters 39-63:
        typing error: unknown identifier ThreadedBehaviourWrapper
```

Sendo a classe interna necessária às anotações (Seção 3.2.4), contornou-se a situação movendo-a para um arquivo dedicado (`ThreadedBehaviourWrapper.java`), tornando-a uma classe convencional, suscetível então à validação sintática do Krakatoa.

3.4.5 Teorias e construções adicionais

A especificação original de JML prevê determinadas extensões para que seja possível especificar tipos de dados abstratos em anotações. A cláusula `represents`, que permite criar funções envolvendo *model fields* [15] (Seção 3.4.3), e a possibilidade de usar métodos em anotações vão ao encontro desse objetivo [11, p. 10].

Em Krakatoa, tais recursos foram preteridos em lugar da possibilidade de declarar novos predicados e funções lógicas. O guia de referência da ferramenta [33, p. 30] cita a possibilidade dessas e outras extensões de "modelagem avançada", porém não as documenta. Informações sobre esse dialeto da linguagem de especificação do Krakatoa encontram-se esparsas em apresentações e artigos. Marché [32, p. 11] expõe a sintaxe desses recursos e acrescenta a possibilidade de usar predicados indutivos e *Algebraic Data Types* (blocos axiomáticos onde são declarados nomes de tipos e seus axiomas). Giorgetti [21, p. 3] reapresenta essa sintaxe e adiciona a possibilidade de definir teorias (cláusula `theory`), bloco lógico podendo apresentar todas as outras extensões (tipos, predicados, funções e axiomas).

O sistema sob análise formal faz uso de funcionalidades das APIs JADE e Java que proveêm para a aplicação estruturas de dados comumente encontradas em APIs (filas, *hash maps* etc.). Com a intenção de construir anotações genéricas o suficiente para serem reaproveitadas em outras ações de verificação, característica importante para o trabalho, tentou-se usar as extensões axiomáticas do Krakatoa, nos moldes de um ADT (*abstract data type*) de Pierce [37, p. 390 *et seq.*].

[15] Tais funções são denominadas *model methods*.

Ao tentar reproduzir exemplos examinados, alguns problemas impediram sua continuidade, sempre de natureza sintática.

O exemplo de Giorgetti *et al.* [21] de definição de uma "teoria" rejeita a cláusula `theory`:

```
/*@ theory Th {
  @   type new_type;
  @   logic new_type func1;
  @   logic integer func2(new_type v, integer k);
  @   axiom axiom_name: axiom_body;
  @ }
  @*/

File "Theory.java", line 1, characters 4-10:
        syntax error (parse error in annotation)
```

O exemplo de Marché [32] de axiomatização de listas de inteiros não aceita a definição de um novo tipo (`ilist`):

```
/*@ axiomatic IntLists {
  @   type ilist;
  @   ilist nil();
  @   ilist cons(integer n, ilist l);
  @   ilist append(ilist l1, ilist l2);
  @   axiom append_nil:
  @     \forall ilist l;
  @     append(nil(),l) == l;
  @   axiom append_cons:
  @     \forall integer n;
  @     \forall ilist l1 l2;
  @     append(cons(n,l1),l2) ==
  @     cons(n,append(l1,l2));
  @ }
  @*/

File "IntLists.java", line 3, characters 6-11:
        syntax error (parse error in annotation)
```

Alguns outros exemplos mais simples (encontrados em apresentações) também apresentaram problemas, como abaixo:

```
/*@   logic integer max(integer x, integer y);
  @   axiom max_is_ge :
  @     \forall integer x y; max(x,y) >= x && max(x,y) >= y;
  @   axiom max_is_some :
  @     \forall integer x y; max(x,y) == x || max(x,y) == y;
```

```
  @/

File "Logic.java", line 1, characters 45-46:
        syntax error (parse error in annotation)
```

De fato, algumas construções simples sucedem, como funções lógicas definidas (não axiomatizadas), contudo foram insuficientes para axiomatizar uma fila ou um *hash map*:

```
//@ logic integer id(integer x) = x;

Parsing OK.
Typing OK.
```

No caso da classe `HashMap`, especificou-se uma lista de conversão fixa entre o *hash* e índice do vetor abstrato (Seção 3.2.4).

3.4.6 Provador automático Yices

Inicialmente o provador automático Yices foi também instalado e usado. Entretanto, assim que algumas obrigações de prova começaram a ser descarregadas, percebeu-se que esse provador sempre fechava todas as obrigações, até mesmo quando referiam-se a anotações ainda incipientes.

Após enviar um teste isolado à lista de discussões da ferramenta, recebeu-se resposta de um dos pesquisadores envolvidos na criação da plataforma Why confirmando que havera sido detectado um mau funcionamento no processo de geração de obrigações de prova especificamente para o provador Yices.

Estando a correção disponível somente em uma versão de desenvolvimento da ferramenta e tendo sido instalados outros provadores automáticos (CVC3, Eprover e Spass), optou-se por desabilitar o Yices do ambiente.

3.4.7 Objetivos de prova de outros arquivos

Na Seção 3.3.3, mencionaram-se duas OPs (#01 e #02) que não puderam ser provadas. Analisando os objetivos gerados, nota-se que são obrigações produzidas para demonstrar a correção dos construtores das classes `estados` e `ThreadedBehaviourWrapper`.

A prova para esses objetivos depende de informação adicional que a ferramenta deveria ter gerado. Em verdade, como a verificação é focada na classe principal, esses objetivos não deveriam ter sido gerados pela ferramenta.

Reproduziu-se então a situação em caso de teste isolado, notificando-a na lista de discussões da ferramenta. De fato, o comportamento esperado seria que quando você executar Krakatoa passando como argumento um arquivo `F.java`, então as OPs para os construtores e métodos de classes em `F.java` deveriam ser apresentados e somente esses, não de outros arquivos.

Finalmente, apenas para ratificar que essas OPs somente não puderam ser fechadas devido a uma operação indesejada da ferramenta, executou-se o Krakatoa para a classe `estados`. Foram geradas duas OPs, uma delas sendo a OP gerada inesperadamente na verificação da classe principal; ambas foram provadas, conforme Tabela 3.5.

Proof obligations	CVC3	Coq	Eprover	Spass
CC Agent, default behavior	0.08			
CC Agent, Safety	0.09			
CC Behaviour, default behavior	0.10			
CC Behaviour, Safety	0.09			
CC ConcurrentLinkedQueue_Emergency, default behavior	0.09			
CC ConcurrentLinkedQueue_Emergency, Safety	0.09			
CC CyclicBehaviour, default behavior	0.09			
CC CyclicBehaviour, Safety	0.08			
CC DataStore, default behavior	0.09			
CC DataStore, default behavior	0.12			
CC DataStore, Safety	0.12			
CC DataStore, Safety	0.11			
CC Emergency, default behavior	0.11			
CC Emergency, Safety	0.09			
CC HashMap, default behavior	0.10			
CC HashMap, Safety	0.11			
CC MonitorAcuteEmergencyBehaviour, default behavior	0.10			
CC MonitorAcuteEmergencyBehaviour, Safety	0.09			
CC Object, default behavior	0.09			
CC Object, Safety	0.11			
CC RequestConfirmationBehaviour, default behavior	0.10			
CC RequestConfirmationBehaviour, Safety	0.11			
CC SimpleBehaviour, default behavior	0.10			
CC SimpleBehaviour, Safety	0.09			
CC String, default behavior	0.09			
CC String, Safety	0.09			
CC String, default behavior	0.09			
CC String, Safety	0.10			
CC String, default behavior	0.09			
CC String, Safety	0.09			
CC String, default behavior	0.09			
CC String, default behavior	0.10			
CC String, default behavior	0.08			
CC String, Safety	0.08			
CC String, default behavior	0.11			
CC String, default behavior	0.09			

Tabela 3.2: Objetivos de prova triviais

Proof obligations	CVC3	Coq	Eprover	Spass
CC String, Safety	0.11			
CC String, Safety	0.08			
CC String, Safety	0.08			
CC String, Safety	0.08			
CC String, default behavior	0.09			
CC String, default behavior	0.08			
CC String, Safety	0.09			
CC String, Safety	0.10			
CC String, default behavior	0.10			
CC String, default behavior	0.09			
CC String, Safety	0.12			
CC String, Safety	0.10			
CC ThreadedBehaviourFactory, default behavior	0.10			
CC ThreadedBehaviourFactory, Safety	0.08			
CC ThreadedBehaviourWrapper, Safety	0.08			
CC estados, Safety	0.08			

Tabela 3.3: Objetivos de prova triviais (cont.)

#	Proof obligations	CVC3	Coq	Eprover	Spass
01	*normal postcondition*	⏲			
02	*normal postcondition*	⏲	?		
03	*Method action, Safety*	⏲	?	⏲	⏲

Tabela 3.4: Objetivos de prova não provados

Proof obligations	CVC3	Coq
normal postcondition		1.35
CC estados, Safety	0.06	

Tabela 3.5: Objetivos de prova exclusivos da classe `estados`

Capítulo 4

Conclusão

Este trabalho apresentou métodos formais que podem ser empregados para análise formal de aplicações imperativas não exclusivamente acadêmicas. No caso trabalhado, devido à dependência de diversas bibliotecas, estabeleceu-se uma disciplina de verificação a anotação. O comportamento esperado foi sempre modelado em duas etapas, partindo da especificação do código alvo (classe principal) e culminando com a anotação de classes usadas (bibliotecas de classes), onde apenas o código necessário para esboçar o comportamento do código alvo foi modelado. A cada classe, analisou-se o comportamento de intenção documentado para posteriormente criar as especificações JML correspondentes.

Observou-se que a formalização e verificação de código comercial produzido em massa é um desafio de tamanho análogo à própria construção da aplicação. Não obstante, percebeu-se que a verificação de propriedades específicas de partes críticas de sistemas de interesse é um objetivo tangível e, de fato, já perseguido por algumas iniciativas na indústria.

Dois fatores demonstraram contribuir sobremaneira neste processo: a tradução automatizada de código imperativo para uma axiomatização apropriada e o uso de provadores automáticos para eliminar a maioria das obrigações de prova.

Apesar de a Lógica de Hoare ser um formalismo que originalmente captura as construções básicas de linguagens imperativas, linguagens de programação comerciais como Java possuem diversas nuanças e extensões que, por si só, carecem de uma formalização semântica. Assim, devido ao uso de determinadas funcionalidades disponíveis nessas linguagens, a própria compreensão integral do código produzido é um desafio. Nesse cenário, de forma manual, realizar a tradução *ad hoc* do código-fonte para a linguagem de especificação de provadores como Coq e PVS é uma tarefa que pode limitar o uso em grande escala das soluções envolvidas, por envolver trabalho repetitivo e suscetível a erros. A abordagem de tradução mecanizada possibilita a concentração de esforços na compreensão dos aspectos semânticos envolvidos, bem como de sua representação apropriada em ambientes de prova.

Por outro lado, mesmo trechos reduzidos de código geram um número elevado de obrigações a serem provadas. No caso estudado, apenas dois objetivos de prova tiveram como

condição *sine qua non* o uso de provadores interativos. Assim, a maioria das obrigações pôde ser provada também de forma mecanizada.

Quanto à linguagem de especificação de propriedades, todas as construções apresentadas são de primeira ordem, o que limita a expressividade das aplicações. Apesar disso, no caso apresentado, foi possível cobrir parte importante de uma aplicação comercial, apenas com construções de primeira ordem. Adicionalmente a linguagem de especificação apresentada (JML) tem evoluído para suportar construções de ordem superior, porém de forma ainda experimental.

4.1 Trabalhos futuros

Ambientes de programação voltados ao acoplamento de *frameworks*, presentes em linguagens orientadas a objeto (como Java), levam a uma profusão de APIs que estendem o núcleo básico das linguagens. Documentar formalmente todo esse código é um desafio, mas que simplifica verificações subsequentes de aplicações clientes dessas bibliotecas, uma vez que o comportamento das bibliotecas já estaria especificado. Para tanto, contudo, é necessário que construções que viabilizem a especificação de tipos de dados abstratos funcionem de forma adequada nessas ferramentas.

Apesar de a maioria das obrigações ter sido provada por provadores automáticos, a necessidade de usar provadores interativos mostra que a identificação de padrões estruturais nas provas poderia ser revertida para a criação de novas táticas, por exemplo, para o Coq. A experiência com as obrigações de prova demonstradas em Coq mostra que há recorrências no processo de criação do termo de prova. Assim, podem ser desenvolvidas táticas de prova especializadas na axiomatização produzida pelas ferramentas, aumentando o grau de automatização do processo de verificação.

Para o uso comercial do Krakatoa é importante que sua documentação seja consolidada, uma vez que encontra-se esparsa em artigos e apresentações. Este é um trabalho futuro que pode ser realizado de forma gradual e todo incremento na documentação diminuirá o tempo de aprendizagem de novos "verificadores".

Finalmente, o problema detectado com o provador automático Yices (Seção 3.4.6) mostrou que há uma lacuna a ser preenchida no projeto do Why: o processo de tradução das anotações JML e código imperativo para os diversos provadores não está formalmente definido. Assim, o *software* envolvido, apesar de ter suas entradas e saídas bem definidas, poderia ter formalizada, por exemplo, a conservatividade do processo de tradução. O trabalho de definição formal de JML em Coq de Lehner [29], apesar de não estar diretamente relacionado com as ferramentas, criou um ambiente para discussão formal sobre a semântica de JML, o qual pode ser usado como ponto de partida para definir formalmente pelo menos o processo (realizado pelo Krakatoa) de tradução das especificações JML (apostas nos códigos-fontes) para a linguagem de entrada do Why.

Referências

[1] Oracle and/or its affiliates. Java Platform, Standard Edition 6, API Specification, 2011. 14, 17, 18, 19, 20

[2] Oracle and/or its affiliates. The Java Tutorials, 2012. 37, 38, 39, 40

[3] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language.* CEA LIST and INRIA Saclay, 2009. 6

[4] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, and Giovanni Rimassa. JADE Programmer's Guide, April 2010. 14, 16, 19, 21

[5] Yves Bertot. Coq in a hurry. *ACM Computing Research Repository (CoRR)*, abs / cs / 0603118, 2006. 8

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions.* Springer-Verlag, 2004. 8

[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. *The Why3 platform.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. `http://why3.lri.fr/`. 7

[8] Jaime Alejandro Bohórquez Villamizar. *Diseño Efectivo de Programas Correctos.* Escuela Colombiana de Ingeniería, Bogotá, D.C., agosto 2005. Tercera Edición Preliminar. 4

[9] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Muñoz, editor, *Second NASA Formal Methods Symposium (NFM 2010)*, volume NASA/CP-2010-216215, pages 14–23, Washington D.C. United States, 04 2010. NASA. Hisseo project, funded by Digiteo. 1

[10] Sylvain Boulmé. A tutorial on reflecting in Coq the generation of Hoare proof obligations, 2009. `http://coq.inria.fr/distrib/v8.2/contribs/HoareTut.html`. 8

[11] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. SV, 2006. 36, 39, 41

[12] The Coq proof assistant, 2009. Versão: 8.2pl1, URL: `http://coq.inria.fr/`. 8

[13] Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. *Frama-C User Manual, Beryllium release*. CEA LIST, Software Reliability Laboratory, Saclay, 2009. 7

[14] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. 3

[15] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct Code Containing Containers. In *5th International Conference on Tests & Proofs (TAP'11)*, Zurich, June 2011. 20

[16] EMERGE, 2009. URL: `http://www.emerge-project.eu/`. 11

[17] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The coq proof assistant reference manual. Technical report, INRIA and IST Types working group, 2009. Versão: 8.2. 10

[18] Jean Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th international conference on Computer Aided verification*, CAV'07, pages 173–177, Berlin, Heidelberg, 2007. Springer-Verlag. viii, 8

[19] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967. 3

[20] Frama-C Software Analyzers, 2009. Versão: Beryllium-20090902+dev, URL: `http://frama-c.cea.fr/`. 7

[21] Alain Giorgetti, Claude Marché, Elena Tushkanova, and Olga Kouchnarenko. Specifying generic java programs: two case studies. In Claus Brabrand and Pierre-Etienne Moreau, editors, *LDTA*, page 8. ACM, 2010. 41, 42

[22] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969. 3, 4

[23] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. 9

[24] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000. 1, 3

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM. 1

[26] The Krakatoa verification tool for Java programs, 2009. Versão: 2.18, URL: `http://krakatoa.lri.fr/`. 7

[27] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006. 6

[28] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2006. 17, 36, 39

[29] Hermann Lehner. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking*. PhD thesis, ETH Zurich, Switzerland, 2011. 48

[30] Leonard Lensink, César Muñoz, and Alwyn Goodloe. From verified models to verifiable code. Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009. 1

[31] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43:363–446, December 2009. 1

[32] Claude Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, January 2009. `http://krakatoa.lri.fr/ws/`. 24, 36, 39, 41, 42

[33] Claude Marché. *Krakatoa: Tutorial and Reference Manual*. INRIA Team-Project, 2010. 7, 36, 41

[34] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certication of Java/JavaCard programs annotated in JML. 2003. 1

[35] Michael Marcotty and Henry F. Ledgard. *Programming language landscape: syntax, semantics, and implementation*. SRA School Group, USA, 1986. 3

[36] Vinicius Uriel Cardoso Nunes. Orquestração de serviços por meio de agentes de software no domínio de vida ambiente-assistida. URL: `http://monografias.cic.unb.br/dspace/bitstream/123456789/225/1/monografia.pdf`, Dezembro 2009. Monografia de conclusão de bacharelado em Ciência da Computação, Universidade de Brasília. 11, 12, 14

[37] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 34, 38, 39, 41

[38] The Why verification tool, 2009. Versão: 2.23, URL: `http://why.lri.fr/`. 7

[39] Z specification language, 2009. URL: `http://www.zuser.org/`. 6

Apêndice A

Códigos Anotados

À exceção do arquivo `MonitorAcuteEmergencyBehaviour.java`, o qual é aqui transcrito *ipsis litteris*, apenas os métodos efetivamente anotados constam deste excerto.

A.1 MonitorAcuteEmergencyBehaviour.java

```
/*KML
package br.unb.cic.aal.behaviour;

import jade.core.behaviours.CyclicBehaviour;

import jade.core.behaviours.ThreadedBehaviourFactory;

import java.util.Observable;
import java.util.Observer;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.Queue_Emergency;
import java.util.concurrent.ConcurrentLinkedQueue_Emergency;

import org.osgi.framework.ServiceReference;

import br.unb.cic.aal.agent.AALAgent;
import br.unb.cic.aal.ontology.Emergency;
import br.unb.cic.aal.service.emergency.detection.DetectEmergencyService;
KML*/

public class MonitorAcuteEmergencyBehaviour extends CyclicBehaviour
        implements Observer {

    /**
     *
```

```
     */
    private static final long serialVersionUID =
        3951408798663833348L;

/*KML
    private Queue<Emergency> emergencies =
        new ConcurrentLinkedQueue<Emergency>();
  KML*/
    private Queue_Emergency emergencies =
        new ConcurrentLinkedQueue_Emergency();
    private DetectEmergencyService detectEmergency = null;
    private ThreadedBehaviourFactory threadFactory = null;

//KML    @Override
    /*@
      @ ensures threadFactory != null;
      @*/
    public void onStart() {
        threadFactory = new ThreadedBehaviourFactory();
/*KML
        ServiceReference reference = null;
        reference =
            ((AALAgent) myAgent).getBundleContext()
            .getServiceReference(
                DetectEmergencyService.class.getName());

        detectEmergency =
            (DetectEmergencyService)((AALAgent) myAgent)
            .getBundleContext().getService(reference);
        detectEmergency.register(this);
  KML*/
    }

//KML    @SuppressWarnings("unchecked")
//KML    @Override
    /*@
      @ requires threadFactory != null;
      @ assigns estados.qtde_emergencias;
      @ behavior emergencia:
      @     assumes estados.qtde_emergencias >= 1;
      @     ensures
      @     estados.qtde_emergencias ==
      @         \old(estados.qtde_emergencias) - 1
      @     && myAgent.qt_behaviours ==
      @         \old(myAgent.qt_behaviours) + 1
      @     && myAgent.lista_behaviours[\old(myAgent.qt_behaviours)]
```

```
      @         instanceof ThreadedBehaviourWrapper
      @     && ((ThreadedBehaviourWrapper)
      @         myAgent.lista_behaviours[\old(myAgent.qt_behaviours)])
      @         .myBehaviour instanceof RequestConfirmationBehaviour;
      @ behavior nenhuma_emergencia:
      @     assumes estados.qtde_emergencias == 0;
      @     ensures estados.qtde_emergencias == 0;
      @*/
    public void action() {
        Emergency emergency = emergencies.poll();
        if (emergency != null) {
            RequestConfirmationBehaviour requestConfirmation =
                new RequestConfirmationBehaviour();
            requestConfirmation.getDataStore()
                .put(RequestConfirmationBehaviour.EVENT_KEY, emergency);
            myAgent.addBehaviour(
                threadFactory.wrap(requestConfirmation));
                //Starts a threaded behaviour
        }
    }

//KML    @Override
    /*@
      @ assigns estados.qtde_emergencias;
      @ behavior enfileirar_emergencia:
      @     assumes arg instanceof Emergency;
      @     ensures estados.qtde_emergencias ==
      @         \old(estados.qtde_emergencias) + 1;
      @ behavior nao_enfileirar_emergencia:
      @     assumes ! (arg instanceof Emergency);
      @     ensures estados.qtde_emergencias ==
      @         \old(estados.qtde_emergencias);
      @*/
    public void update(Observable o, Object arg) {
        if (arg instanceof Emergency) {
            emergencies.add((Emergency) arg);
        }
    }
}
```

A.2 Agent.java

```
/*KML
package jade.core;
```

```
import jade.core.behaviours.Behaviour;
  KML*/


public class Agent {

        /**
         Default constructor.
         */
        public Agent();

        //@ model integer qt_behaviours = 0;
        //@ model Behaviour [] lista_behaviours;

        /*@
          @ requires b instanceof ThreadedBehaviourWrapper;
          @ ensures qt_behaviours == \old(qt_behaviours) + 1 &&
          @         lista_behaviours[\old(qt_behaviours)] == b;
          @*/
        public void addBehaviour(Behaviour b);

//        public void removeBehaviour(Behaviour b);

}
```

A.3 Queue_Emergency.java

```
/*KML
package java.util;

import br.unb.cic.aal.ontology.Emergency;
  KML*/

public interface Queue_Emergency /*KML extends Collection<E> KML*/ {

        /*@
          @ assigns estados.qtde_emergencias;
          @ ensures estados.qtde_emergencias ==
          @     \old(estados.qtde_emergencias) + 1;
          @*/
    boolean add(Emergency e);

//    Emergency remove();

        /*@
```

```
          @ assigns estados.qtde_emergencias;
          @ behavior fila_com_conteudo:
          @     assumes estados.qtde_emergencias >= 1;
          @     ensures \result != null &&
          @             estados.qtde_emergencias ==
          @             \old(estados.qtde_emergencias) - 1;
          @ behavior fila_vazia:
          @     assumes estados.qtde_emergencias == 0;
          @     ensures \result == null &&
          @             estados.qtde_emergencias ==
          @             \old(estados.qtde_emergencias);
          @*/
    Emergency poll();

//    Emergency peek();
}
```

A.4 ThreadedBehaviourFactory.java

```
/*KML
package jade.core.behaviours;

//#MIDP_EXCLUDE_FILE

import jade.core.Agent;
import jade.core.NotFoundException;
import jade.util.Logger;

import java.lang.reflect.Method;
import java.util.Vector;
import java.util.Enumeration;
  KML*/

public class ThreadedBehaviourFactory {
        /*@
          @ requires b != null;
          @ ensures \result instanceof ThreadedBehaviourWrapper
          @         && \result != null
          @         && ((ThreadedBehaviourWrapper)\result)
          @            .myBehaviour == b;
          @*/
        public Behaviour wrap(Behaviour b) ; /*KML {
                return new ThreadedBehaviourWrapper(b);
        } KML*/
}
```

A.5 ThreadedBehaviourWrapper.java

```
/**
 * Inner class ThreadedBehaviourWrapper
 * This class is declared public for debugging purpose only
 */
public class ThreadedBehaviourWrapper extends Behaviour
        /*KML implements Runnable KML*/ {

        private Behaviour myBehaviour;

        //@ ensures myBehaviour == b;
        private ThreadedBehaviourWrapper(Behaviour b);

} // END of inner class ThreadedBehaviourWrapper
```

A.6 HashMap.java

```
/*KML
package jade.util.leap;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.util.Hashtable;
import java.util.Enumeration;
KML*/

/**
   The LEAP (environment-dependent) version of the
   &qote;java.util.HashMap&qote; class.
   This class appears to be exactly the same in J2SE, PJAVA and MIDP.
   The internal implementation is different in the three cases however.

   @author  Nicolas Lhuillier
   @version 1.0, 29/09/00

   @see java.util.HashMap
 */

public class HashMap /*KML implements Map, Serializable KML*/ {
    //@ model Object [] vetor;

    /*@
```

```
      @ ensures key == RequestConfirmationBehaviour.EVENT_KEY ==>
      @         vetor[0] == value;
      @*/
    public Object put(Object key, Object value);

    /*@
      @ ensures key == RequestConfirmationBehaviour.EVENT_KEY ==>
      @         \result == vetor[0];
      @*/
    public Object get(Object key);
}
```

A.7 Behaviour.java

```
/*KML
package jade.core.behaviours;

import jade.util.leap.Serializable;

import jade.core.behaviours.DataStore;
  KML*/

public abstract class Behaviour /*KML implements Serializable KML*/ {
//    void setParent(CompositeBehaviour cb);

//    void setWrappedParent(CompositeBehaviour cb);

//    protected CompositeBehaviour getParent();

//    public Behaviour(){this.onStart();}

    /*@
      @ requires a != null;
      @ ensures myAgent == a;
      @*/
    public Behaviour(Agent a);

    public abstract void action();

    public void onStart();

    //@ ensures \result == myStore && myStore != null;
    public DataStore getDataStore();

    /*@
```

```
      @ requires ds != null;
      @ ensures myStore == ds;
      @*/
    public void setDataStore(DataStore ds);

    protected Agent myAgent;

    private DataStore myStore;
}
```

A.8 estados.java

```
public class estados {
        public static int qtde_emergencias;

        //@ invariant qtde_positiva: qtde_emergencias >= 0;

        /*@
          @ assigns qtde_emergencias;
          @ ensures qtde_emergencias == 0;
          @*/
        estados() {
                qtde_emergencias = 0;
        }
}
```

Anexo I

Trechos da axiomatização Java *memory heap*

Reproduzem-se aqui alguns poucos trechos da axiomatização descarregada pelo Why para os arquivos de prova do Coq.

```
(* This file is generated by Why3's Coq driver *)
(* Beware! Only edit allowed sections below    *)

...
... (SUPRIMIDO)
...

Definition valid (t:Type)(a:(alloc_table t)) (p:(pointer t)): Prop :=
  ((offset_min a p) <= 0%Z)%Z /\ (0%Z <= (offset_max a p))%Z.
Implicit Arguments valid.

...
... (SUPRIMIDO)
...

Axiom address_injective : forall (t:Type), forall (p:(pointer t)),
  forall (q:(pointer t)), (p = q) <-> ((address p) = (address q)).

Axiom address_shift_lt : forall (t:Type), forall (p:(pointer t)),
  forall (i:Z), forall (j:Z), ((address (shift p i)) <  (address (shift p
  j)))%Z <-> (i <  j)%Z.

Axiom address_shift_le : forall (t:Type), forall (p:(pointer t)),
  forall (i:Z), forall (j:Z), ((address (shift p i)) <= (address (shift p
  j)))%Z <-> (i <= j)%Z.

Axiom shift_zero : forall (t:Type), forall (p:(pointer t)), ((shift p
```

```
  0%Z) = p).

Axiom shift_shift : forall (t:Type), forall (p:(pointer t)), forall (i:Z),
  forall (j:Z), ((shift (shift p i) j) = (shift p (i + j)%Z)).

Axiom offset_max_shift : forall (t:Type), forall (a:(alloc_table t)),
  forall (p:(pointer t)), forall (i:Z), ((offset_max a (shift p
  i)) = ((offset_max a p) - i)%Z).

Axiom offset_min_shift : forall (t:Type), forall (a:(alloc_table t)),
  forall (p:(pointer t)), forall (i:Z), ((offset_min a (shift p
  i)) = ((offset_min a p) - i)%Z).

Axiom neq_shift : forall (t:Type), forall (p:(pointer t)), forall (i:Z),
  forall (j:Z), (~ (i = j)) -> ~ ((shift p i) = (shift p j)).

Axiom null_not_valid : forall (t:Type), forall (a:(alloc_table t)),
  ~ (valid a (null:(pointer t))).

Axiom null_pointer : forall (t:Type), forall (a:(alloc_table t)),
  (0%Z <= (offset_min a (null:(pointer t))))%Z /\ ((offset_max a
  (null:(pointer t))) <= (-2%Z)%Z)%Z.

...
... (SUPRIMIDO)
...

Parameter memory : forall (t:Type) (v:Type), Type.

Parameter select: forall (t:Type) (v:Type), (memory t v) -> (pointer t) -> v.

Implicit Arguments select.

Parameter store: forall (t:Type) (v:Type), (memory t v) -> (pointer t)
  -> v -> (memory t v).

Implicit Arguments store.

Axiom select_store_eq : forall (t:Type) (v:Type), forall (m:(memory t v)),
  forall (p1:(pointer t)), forall (p2:(pointer t)), forall (a:v),
  (p1 = p2) -> ((select (store m p1 a) p2) = a).

Axiom select_store_neq : forall (t:Type) (v:Type), forall (m:(memory t v)),
  forall (p1:(pointer t)), forall (p2:(pointer t)), forall (a:v),
  (~ (p1 = p2)) -> ((select (store m p1 a) p2) = (select m p2)).
```

```
...
... (SUPRIMIDO)
...

Definition not_assigns (t:Type) (v:Type)(a:(alloc_table t)) (m1:(memory t v))
  (m2:(memory t v)) (l:(pset t)): Prop := forall (p:(pointer t)), ((valid a
  p) /\ ~ (in_pset p l)) -> ((select m2 p) = (select m1 p)).
Implicit Arguments not_assigns.

Axiom not_assigns_refl : forall (t:Type) (v:Type), forall (a:(alloc_table
  t)), forall (m:(memory t v)), forall (l:(pset t)), (not_assigns a m m l).

Axiom not_assigns_trans : forall (t:Type) (v:Type), forall (a:(alloc_table
  t)), forall (m1:(memory t v)), forall (m2:(memory t v)), forall (m3:(memory
  t v)), forall (l:(pset t)), (not_assigns a m1 m2 l) -> ((not_assigns a m2
  m3 l) -> (not_assigns a m1 m3 l)).

...
... (SUPRIMIDO)
...

Axiom subtag_parent : forall (t:Type), forall (t1:(tag_id t)),
  forall (t2:(tag_id t)), forall (t3:(tag_id t)), (subtag t1 t2) ->
  ((parenttag t2 t3) -> (subtag t1 t3)).

Definition instanceof (t:Type)(a:(tag_table t)) (p:(pointer t)) (t1:(tag_id
  t)): Prop := (subtag (typeof a p) t1).
Implicit Arguments instanceof.

Parameter downcast: forall (t:Type), (tag_table t) -> (pointer t) -> (tag_id
  t) -> (pointer t).

Implicit Arguments downcast.

Axiom downcast_instanceof : forall (t:Type), forall (a:(tag_table t)),
  forall (p:(pointer t)), forall (s:(tag_id t)), (instanceof a p s) ->
  ((downcast a p s) = p).

...
... (SUPRIMIDO)
...

Parameter alloc_extends: forall (t:Type), (alloc_table t) -> (alloc_table
  t) -> Prop.

Implicit Arguments alloc_extends.
```

```
Definition alloc_fresh (t:Type)(a:(alloc_table t)) (p:(pointer t))
  (n:Z): Prop := forall (i:Z), ((0%Z <= i)%Z /\ (i <  n)%Z) -> ~ (valid a
  (shift p i)).
Implicit Arguments alloc_fresh.

Axiom alloc_extends_offset_min : forall (t:Type), forall (a1:(alloc_table
  t)), forall (a2:(alloc_table t)), (alloc_extends a1 a2) ->
  forall (p:(pointer t)), (valid a1 p) -> ((offset_min a1 p) = (offset_min a2
  p)).

Axiom alloc_extends_offset_max : forall (t:Type), forall (a1:(alloc_table
  t)), forall (a2:(alloc_table t)), (alloc_extends a1 a2) ->
  forall (p:(pointer t)), (valid a1 p) -> ((offset_max a1 p) = (offset_max a2
  p)).

...
... (SUPRIMIDO)
...

Axiom usMonitorAcuteEmergencyBehaviour_parenttag_CyclicBehaviour :
  (parenttag usMonitorAcuteEmergencyBehaviour_tag
  usCyclicBehaviour_tag).

Definition usNon_null_BehaviourM(x_4:(pointer usObject))
  (usObject_alloc_table:(alloc_table usObject)): Prop :=
  ((-1%Z)%Z <= (offset_max usObject_alloc_table x_4))%Z.

Definition usNon_null_Object(x_5:(pointer usObject))
  (usObject_alloc_table:(alloc_table usObject)): Prop :=
  (0%Z <= (offset_max usObject_alloc_table x_5))%Z.

Definition usNon_null_ObjectM(x_0:(pointer usObject))
  (usObject_alloc_table:(alloc_table usObject)): Prop :=
  ((-1%Z)%Z <= (offset_max usObject_alloc_table x_0))%Z.

Definition usNon_null_StringM(x_3:(pointer usObject))
  (usObject_alloc_table:(alloc_table usObject)): Prop :=
  ((-1%Z)%Z <= (offset_max usObject_alloc_table x_3))%Z.

Definition usNon_null_byteM(x_1:(pointer usObject))
  (usObject_alloc_table:(alloc_table usObject)): Prop :=
  ((-1%Z)%Z <= (offset_max usObject_alloc_table x_1))%Z.

Definition usNon_null_charM(x_2:(pointer usObject))
  (usObject_alloc_table:(alloc_table usObject)): Prop :=
```

```
  ((-1%Z)%Z <= (offset_max usObject_alloc_table x_2))%Z.

...
... (SUPRIMIDO)
...
(* YOU MAY EDIT THE CONTEXT BELOW *)
```