Prevenção de Violações de Atomicidade usando Contratos

Diogo G. Sousa, Carla Ferreira e João M. Lourenço *

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
dm.sousa@campus.fct.unl.pt
{carla.ferreira,joao.lourenco}@fct.unl.pt

Resumo A programação concorrente obriga o programador a sincronizar os acessos concorrentes a regiões de memória partilhada, contudo esta abordagem não é suficiente para evitar todas as anomalias que podem ocorrer num cenário concorrente. Executar uma sequência de operações atómicas pode causar violações de atomicidade se existir uma correlação entre essas operações, devendo o programador garantir que toda a sequência de operações é executada atomicamente. Este problema é especialmente comum quando se usam operações de pacotes ou módulos de terceiros, pois o programador pode identificar incorretamente o âmbito das regiões de código que precisam de ser atómicas para garantir o correto comportamento do programa. Para evitar este problema o programador do módulo pode criar um contrato que especifica quais as sequências de operações do módulo que devem ser sempre executadas de forma atómica. Este trabalho apresenta uma análise estática para verificação destes contratos.

Keywords: Verificação de Protocolos, Concorrência, Violações de Atomicidade, Programação por Contrato, *Thread Safety*, Análise Estática, Verificação de *Software*

1 Introdução

O encapsulamento do funcionamento interno da implementação de um módulo de *software* oferece fortes vantagens no desenvolvimento de *software*, pois permite a sua reutilização e facilita a manutenção de código. No entanto esta transparência de utilização pode levar a utilizações incorretas de um módulo devido ao programador não estar consciente dos detalhes de implementação.

Um dos requisitos para o correto comportamento de um módulo é respeitar o seu *protocolo*, que determina quais as sequências de invocações a métodos que são legítimas. Por exemplo, um módulo que ofereça uma abstração para lidar com ficheiros tipicamente exige que o programador comece por chamar o método

* Este trabalho foi parcialmente financiado pela Fundação para a Ciência e Tecnologia (FCT/MCTES), no contexto do projecto de investigação PTDC/EIA-EIA/113613/2009.

```
void schedule() {
    Task t=taskQueue.next();

    if (t.isReady())
        t.run();
}
```

Figura 1: Programa com uma violação de atomicidade.

`open()`, seguido de um número arbitrário de `read()`s ou `write()`s, e por fim de `close()`. Um programa que não cumpra este protocolo está incorreto e deve ser corrigido. Uma solução é usar a metodologia de programação por contrato [19], onde o programador do módulo especifica qual o seu protocolo de utilização. Este contrato, para além de servir como documentação do módulo, pode ser verificado automaticamente, assegurando que um programa cumpre o contrato do módulo [7, 14].

A programação concorrente levanta novos desafios em relação ao protocolo de um módulo. Não só é importante respeitar o protocolo do módulo como é necessário garantir a execução atómica de determinadas sequências de chamadas suscetíveis de causarem violações de atomicidade. Estas violações de atomicidade são possíveis mesmo se os métodos do módulo empregarem mecanismos de controlo de concorrência. Tomemos como exemplo o programa apresentado na Figura 1. Este programa está encarregado de escalonar tarefas. O método `schedule()` obtém uma tarefa e executa-a se esta está pronta para tal. Este programa contem uma potencial violação de atomicidade, pois o método pode executar uma tarefa que não está pronta devido a um *thread* concorrente ter executado a mesma tarefa. Neste caso os métodos `isReady()` e `run()` devem ser executados no mesmo contexto atómico para evitar violações de atomicidade. As violações de atomicidade são um problema comum em programação concorrente [18] e são particularmente propensas a serem introduzidas quando se compõem chamadas a um módulo, por o programador poder não estar consciente da implementação e estado interno do módulo.

Neste artigo propomos estender os protocolos de utilização de módulos com uma especificação das sequências de chamadas ao módulo que devem ser executadas atomicamente, de forma a evitar violações de atomicidade. As contribuições deste trabalho são:

1. Uma análise estática para extrair o comportamento de um programa relativamente às sequências de chamadas que pode executar;
2. Uma análise estática para verificar a atomicidade de sequências de chamadas a um módulo por parte do programa cliente.

Na Secção 2 definimos a especificação do contrato e a sua semântica. A Secção 3 contem a metodologia geral da análise. A Secção 4 apresenta a fase da análise que extrai o comportamento do programa cliente e a Secção 5 demonstra como verificar o contrato. Segue-se a Secção 6 com a demonstração experimental do nosso trabalho. O trabalho relacionado é apresentado na Secção 7 e terminamos com as conclusões na Secção 8.

2 Especificação do Contrato

O contrato de um módulo determina quais as sequências de chamadas aos seus métodos públicos que devem ser executadas atomicamente pelo programa cliente de forma a evitar violações de atomicidade. O programador do módulo deve ser responsável por identificar estas sequências de chamadas e definir o contrato.

Definição 1 (Contrato)**.** A especificação do contrato de um módulo com os métodos públicos $m_1, \cdots, m_n$ tem a forma,

$$1.\ e_1$$
$$\vdots$$
$$k.\ e_k.$$

Cada *cláusula* i do contrato é descrita por e_i, uma expressão regular sem fecho de Kleene sobre o alfabeto $\{m_1, \cdots, m_n\}$. Expressões regulares sem fecho de Kleene são expressões regulares que não usam a estrela de Kleene, tendo apenas os operadores de alternativa (|) e concatenação (omisso).

Cada sequência definida em e_i tem de ser executada atomicamente pelo programa cliente do módulo, caso contrário isso representa uma violação do contrato. O contrato especifica um número finito de sequências de chamadas, por ser uma união de linguagens livres de estrela. Por isso é possível ter a mesma expressividade enumerando explicitamente todas as sequências de chamadas, i.e. evitando a utilização do operador de alternativa. Escolhemos oferecer o operador de alternativa para que se possa agrupar na mesma cláusula várias sequências que sejam similares. É um pré-requisito da análise apresentada que o contrato defina um número *finito* de sequências de chamadas.

Exemplo Consideremos a implementação de *arrays* oferecida pela biblioteca *standard* de Java, `java.util.ArrayList`.

O seguinte contrato define duas das possíveis cláusulas desta classe.

1. `contains indexOf`
2. `indexOf` (`set` | `remove` | `get`)

A cláusula 1 do contrato da `ArrayList` indica que a execução do método `contains()` seguido de `indexOf()` deve ser feita de maneira atómica, caso contrário um programa cliente pode confirmar a existência de um objeto no *array* mas falhar ao obter o seu índice devido a uma modificação concorrente. A cláusula 2 representa um cenário semelhante onde adicionalmente se modifica essa posição do objeto obtido.

3 Metodologia

A análise proposta verifica estaticamente se um programa cliente cumpre o contrato de um dado módulo como definido na Secção 2. Para isto verificamos que

cada *thread* que o programa possa lançar executa sempre de forma atómica todas das sequências de chamadas definidas pelas cláusulas do contrato.

Esta análise é composta pelas seguintes fases:

1. Determinar os métodos de entrada de cada *thread* que o programa pode executar.
2. Determinar quais os métodos do programa que são executados atomicamente. Dizemos que um método é *executado atomicamente* se este for atómico[1] ou se for sempre chamado por métodos executados atomicamente.
3. Extrair o comportamento de utilização do módulo sob análise de cada *thread* do programa.
4. Para cada *thread*, verificar que a sua utilização do módulo respeita sempre o contrato, como definido na Secção 2.

Na Secção 4 apresentamos o algoritmo de extração do comportamento do programa cliente relativamente à utilização do módulo. Na Secção 5 definimos a análise que verifica se o comportamento extraído cumpre o contrato.

4 Extração do Comportamento do Programa

O comportamento do programa relativamente à sua utilização de um módulo pode ser visto como o comportamento individual de cada *thread* que o programa possa executar. A utilização de um módulo por um *thread* do programa pode ser descrita por uma linguagem L onde cada palavra $w \in L$ representa um sequência de chamadas do programa a métodos do módulo.

Esta fase da análise gera uma gramática livre de contexto que representa esta linguagem L a partir de um *thread* t do programa cliente, representado pelo seu *control flow graph* (CFG) [1]. O CFG do *thread* t representa todas os possíveis caminhos que o *control flow* pode tomar na sua execução. Em concreto a análise gera uma gramática G_t tal que, se for possível alguma execução de t correr a sequência de chamadas $m_1, \cdots, m_n$, então $m_1 \cdots m_n \in \mathcal{L}(G_t)$. ($\mathcal{L}(G)$ denota a linguagem representada pela gramática G.)

Definição 2 (Gramática do Comportamento de um *Thread* do Programa)**.** A gramática $G_t = (N, \Sigma, P, S)$ é criada a partir do CFG do *thread* t do programa cliente. Definimos,

- N, o conjunto de não-terminais, como o conjunto de nós do CFG mais um conjunto de não-terminais que representam cada um dos método do programa cliente (representados em fonte caligráfica);
- Σ, o conjunto de terminais, como o conjunto dos identificadores dos métodos públicos do módulo sob análise (representado a negrito);
- P, o conjunto das produções, como descrito em baixo, pelas as Regras 1–5.
- S, o símbolo inicial da gramática, como o não-terminal que representa o método de entrada do *thread* t.

[1] Um método atómico é um método onde explicitamente se aplica algum mecanismo de controlo de concorrência que lhe garante atomicidade.

Para cada método `f()`, potencialmente usado pelo *thread* t, que representamos por $\mathcal{F}$, adicionamos a P as produções que respeitam as restrições das Regras 1–5. Denotamos um nó do CFG por $\alpha : [\![v]\!]$, onde α é o não-terminal que representa o nó e v o seu tipo. Distinguimos os seguintes tipos de nó: *entry*, o nó de entrada do método $\mathcal{F}$; *module.h()*, uma chamada ao método `h()` do módulo sob análise; *g()*, uma chamada a um outro método do programa cliente; e *return*, o ponto de retorno do método $\mathcal{F}$. A função $succ : N \to \mathcal{P}(N)$ é usada para obter os sucessores de um dado nó do CFG.

$$\text{se } \alpha : [\![\text{entry}]\!], \quad \{\mathcal{F} \to \alpha\} \cup \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \tag{1}$$

$$\text{se } \alpha : [\![\text{module.h()}]\!], \quad \{\alpha \to \mathbf{h}\,\beta \mid \beta \in succ(\alpha)\} \subset P \tag{2}$$

$$\text{se } \alpha : [\![\text{g()}]\!], \quad \{\alpha \to \mathcal{G}\,\beta \mid \beta \in succ(\alpha)\} \subset P \text{ onde } \mathcal{G} \text{ representa g()} \tag{3}$$

$$\text{se } \alpha : [\![\text{return}]\!], \quad \{\alpha \to \epsilon\} \subset P \tag{4}$$

$$\text{se } \alpha : [\![\text{c.c.}]\!], \quad \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \tag{5}$$

Mais nenhuma produção pertence a P.

As Regras 1–5 capturam a estrutura do CFG sob a forma de uma linguagem livre de contexto. A Regra 1 adiciona a produção que relaciona o não-terminal $\mathcal{F}$, que representa o método `f()`, ao nó de entrada do CFG de `f()`. As chamadas ao módulo sob análise são registadas na gramática (Regra 2). A Regra 3 lida com as chamadas a um outro método `g()` dentro do programa cliente (o método `g()` terá o seu não-terminal $\mathcal{G}$ adicionado pela Regra 1). O ponto de retorno do método, adiciona uma produção ϵ (Regra 4). Todos os outros tipos de nó do CFG são tratados uniformemente, apenas fazendo a ligação para os nós sucessores (Regra 5). É de notar que apenas o código do programa cliente é analisado.

Intuitivamente esta gramática representa o *control flow* do *thread* t do programa, ignorando o que não é relevante relativamente à sua utilização do módulo. Por exemplo, se $\mathbf{f}\ \mathbf{g} \in \mathcal{L}(G_t)$ então o *thread* t pode ter uma execução onde executa consecutivamente os método `f()` e `g()` do módulo.

A gramática G_t pode ser ambígua, ou seja, oferecer várias derivações diferentes para a mesma palavra. Cada ambiguidade no *parsing* de uma sequência de chamadas $m_1 \cdots m_n \in \mathcal{L}(G_t)$ representa diferentes contextos onde essas chamadas podem ser executadas pelo *thread* t. É portanto desejável manter a gramática ambígua para que a verificação do contrato tenha em conta todas as ocorrências das sequências de chamadas no programa cliente.

A linguagem $\mathcal{L}(G_t)$ pode conter sequências de chamadas que o programa não executa, por exemplo, considerando um caminho no CFG que nunca é executado. No entanto a linguagem $\mathcal{L}(G_t)$ contem todas as sequências de chamadas que o programa pode executar, não produzindo falsos negativos.

Exemplo Consideremos o programa apresentado na Figura 2. Este programa exemplifica como a análise lida com ciclos e recursividade. O programa tem apenas o método `f()`, sendo este o ponto de entrada to *thread* em questão. O módulo sob análise é representado pelo objeto `m`. O CFG deste método está

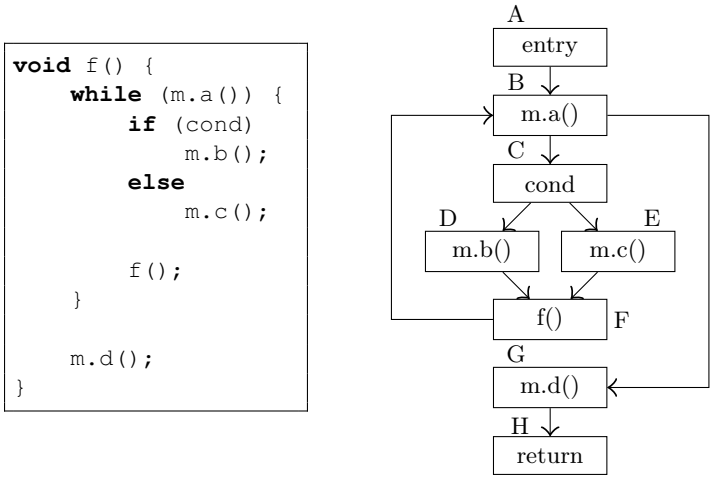

Figura 2: Programa que usa o módulo `m` (esquerda) e respetivo CFG (direita).

representado na Figura 2 (direita). Segundo a Definição 2 construimos a gramática $G = (N, \Sigma, P, S)$, onde $N = \{\mathcal{F}, A, B, C, D, E, F, G, H\}$, $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$ e $S = \mathcal{F}$. As produções P da gramática são,

$$\begin{array}{lll} \mathcal{F} \to A & C \to D \mid E & F \to \mathcal{F}\, B \\ A \to B & D \to \mathbf{b}\, F & G \to \mathbf{d}\, H \\ B \to \mathbf{a}\, C \mid \mathbf{a}\, G & E \to \mathbf{c}\, F & H \to \epsilon. \end{array}$$

5 Verificação de Contratos

A verificação de um contrato garante que todas as sequências de chamadas definidas por este são executadas de forma atómica em todos os *threads* do programa cliente. Visto que existe um número finito de sequências definidas pelo contrato verificamos cada uma das sequências para verificar o contrato.

O Algoritmo 1 apresenta o pseudo-código que verifica um contrato contra um programa cliente. Para cada *thread* t do programa cliente é necessário saber onde ocorrem, no programa, cada uma das sequências $w = m_1, \cdots, m_n$ definidas pelo contrato (linha 4). Para isso fazemos o *parsing* destas sequências na gramática G'_t (linha 5). A gramática G'_t inclui todas as palavras e sub-palavras de G_t. As sub-palavras têm de ser incluídas pois é necessário ter em conta traços de execução parciais do *thread* t. A gramática G'_t pode ser ambígua e cada árvore de *parsing* distinta representa diferentes execuções de $m_1, \cdots, m_n$ que podem ocorrer no *thread* t. A função `parse()` devolve o conjunto destas árvores de *parsing*. Se subirmos na árvore de *parsing* iremos encontrar o nó que representa o método sobre o qual todas as chamadas aos métodos $m_1, \cdots, m_n$ são executadas. Este nó é o *lowest common ancestor* dos terminais $m_1, \cdots, m_n$ na árvore de *parsing* (linha 7). Temos portanto que se o *lowest common ancestor* obtido for executado atomicamente (linha 8) então toda a sequência de chamadas é executada sobre

Algoritmo 1 Algoritmo de verificação do contrato.

Require: P, o programa cliente;
C, o contrato do módulo (conjunto de sequências).

```
1: for t ∈ threads(P) do
2:     G_t ← make_grammar(t)
3:     G'_t ← subword_grammar(G_t)
4:     for w ∈ C do
5:         T ← parse(G'_t, w)
6:         for τ ∈ T do
7:             N ← lowest_common_ancestor(τ, w)
8:             if ¬run_atomically(N) then
9:                 return ERROR
10: return OK
```

o mesmo contexto atómico, respeitando o contrato. A árvore de *parsing* tem informação da localização no código onde pode ocorrer um violação do contrato e podemos oferecer instruções ao utilizador de onde esta ocorre e como a evitar.

A gramática G_t pode usar toda a expressividade oferecida pelas linguagens livres de contexto. Por isso não é suficiente usar o algoritmo $LR(\cdot)$ [16] para a fase de *parsing*, visto que este não lida com ambiguidades na gramática. Para contornar isto podemos usar um *parser GLR* (*Generalized LR parser*), que explora todas as ambiguidades que podem dar origem a diferentes árvores de derivação. Tomita apresenta um *parser GLR* [21], uma versão não-determinista do algoritmo $LR(0)$, bem como algumas otimizações na representação da *stack* de *parsing* que reduzem a complexidade temporal e espacial da fase de *parsing*.

Outro aspeto importante a ter em conta é que o número de árvores de *parsing* pode ser infinito. Isto é devido a ciclos na gramática, ou seja, derivações de um não-terminal para si próprio ($A \Rightarrow \cdots \Rightarrow A$), o que é comum em G_t. Por este motivo a função `parse()` deve detetar ciclos redundantes e terminar a exploração desse ramo de *parsing*. Para isto o algoritmo deve detetar um ciclos na lista de reduções aplicadas e abortar o ramo atual de *parsing* se este ciclo não tiver contribuído para o reconhecimento de um novo terminal.

Exemplos A Figura 3 apresenta um programa (esquerda), que usa o módulo `m`, com três métodos, onde o método `run()` é o ponto de entrada de um *thread* t e é atómico. No centro da figura apresentamos a gramática G_t simplificada. (Não apresentamos a gramática G'_t por motivos de brevidade.) Os métodos `run()`, `f()` e `g()` são representados na gramática pelos não-terminais $\mathcal{R}$, $\mathcal{F}$ e $\mathcal{G}$ respetivamente. Se aplicarmos o Algoritmo 1 a este programa com o contrato $C = \{\mathbf{a\ b\ b\ c}\}$ a árvore de *parsing* obtida (representada por τ na linha 6 do algoritmo) é apresentada na Figura 3 (direita). Para verificar que todas as chamadas representadas nesta árvore de *parsing* são executadas atomicamente o algoritmo de verificação obtém o *lowest common ancestor* de **a b b c** na árvore (linha 7), neste caso $\mathcal{R}$. Como $\mathcal{R}$ é executado atomicamente (**`atomic`**) este contrato está a ser respeitado pelo programa cliente do módulo.

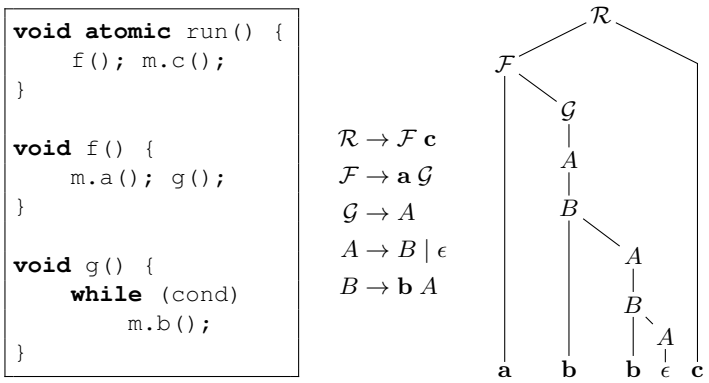

Figura 3: Programa (esquerda), gramática simplificada (centro) e árvore de *parsing* de **a b b c** (direita).

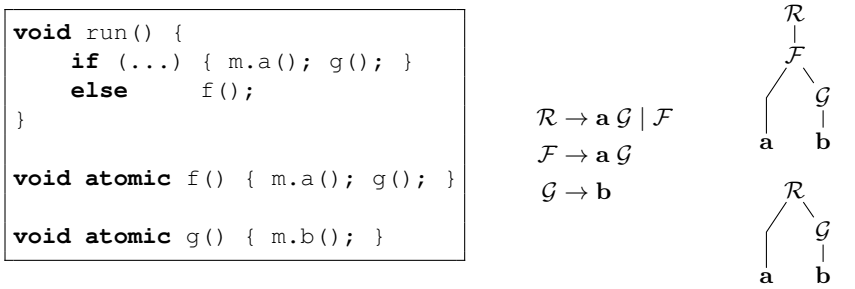

Figura 4: Programa (esquerda), gramática simplificada (centro) e árvores de *parsing* de **a b** (direita).

A Figura 4 exemplifica uma situação onde a gramática gerada é ambígua. Neste caso temos o contrato $C = \{\mathbf{a}\ \mathbf{b}\}$. A figura mostra as duas maneira de fazer *parsing* da palavra **a b** (direita). O Algoritmo 1 vai gerar ambas as árvores. A primeira árvore (cima) tem como *lowest common ancestor* de **a b** o não-terminal $\mathcal{F}$, que corresponde ao método `f()`. Este método é executado atomicamente, respeitando o contrato. Por sua vez a segunda árvore (baixo) tem $\mathcal{R}$ como *lowest common ancestor* de **a b**. Este não-terminal não corresponde a um método executado atomicamente e por isso o contrato não é cumprido.

6 Validação

Foi implementado um protótipo para avaliar a análise apresentada. Esta ferramenta analisa programas *Java* utilizando a *framework* de análise estática Soot [22]. Esta *framework* analisa diretamente *Java bytecode*, permitindo a análise de um programa compilado, sem necessitar de aceder ao seu código fonte.

Para validar a análise proposta foram usados 15 testes tirados da literatura relacionada. Estes testes são programas que contêm violações de atomicidade conhecidas. Modificamos estes testes para um desenho modular e escrevemos

Tabela 1: Resultados de validação.

Teste	***Threads***	**Palavras do Contrato**	**Violações Detetadas**
Account [24]	2	4	2
Allocate Vector [15]	2	1	1
Arithmetic DB [17]	2	2	2
Connection [4]	2	2	2
Coord03 [2]	5	4	1
Coord04 [3]	2	2	1
Elevator [24]	2	2	2
Jigsaw [24]	2	1	1
Knight [17]	2	1	1
Local [2]	2	2	1
StringBuffer [3]	1	1	1
NASA [2]	2	1	1
Store [20]	3	1	1
UnderReporting [24]	2	1	1
VectorFail [20]	2	2	1

contratos para verificar o escopo atómico das chamadas a este módulo. Em alguns testes os contratos contêm cláusulas com sequências de chamadas que os testes não executam, mas que, por completude, devem pertencer ao contrato.

A Tabela 1 apresenta os resultados que validam a correção da análise proposta. As colunas representam o número de métodos de entrada de *threads* (*Threads*); o número de palavras do contrato, ou seja, $|C|$ como definido no Algoritmo 1 (Palavras do Contrato); e o número de violações do contrato detetadas (Violações Detetadas). A nossa ferramenta *detetou todas as violações do contrato* pelo programa cliente, pelo que não existem falsos negativos, o que suporta a *soundness* da análise. Em nenhum dos testes foram reportados falsos positivos. Embora seja possível construir programas que levam a falsas violações, isto tende a não acontecer. Uma versão corrigida dos testes foi verificada e o protótipo identificou corretamente todas as sequências de chamadas do contrato como estando agora a ser executadas atomicamente. Todas as sequências de chamadas dos contratos verificados têm tamanho dois.

A avaliação da performance é apresentada na Tabela 2. As colunas representam o número de linhas do programa cliente (SLOC Cliente); o número de nós do CFG do programa cliente (Nós CFG Cliente); o número de produções da gramática (Produções Gramática); o número de ramos de *parsing* que o *parser* explorou, incluindo ramos que falharam (Ramos de *Parsing*); e o tempo de análise, que inclui a deteção de *threads*, geração da gramática, criação das tabelas de *parsing* e *parsing* (Tempo de Execução). Esta coluna exclui o tempo de inicialização do Soot, que para estes testes foram sempre de 35 ± 5 s. Os resultados de *performance* mostram que a análise é eficiente. O programa Elevator é o teste mais lento, pois o *parser* explora um número razoavelmente elevado de ramos. Isto deve-se à complexidade do CFG deste teste. Em geral o tempo de *parsing* vai dominar a complexidade temporal da análise, ou seja, o tempo de execução

Tabela 2: Resultados de *performance*.

Teste	SLOC Cliente	Nós CFG Cliente	Produções Gramática	Ramos de *Parsing*	Tempo de Execução (ms)
Account	19	36	62	34	20
Allocate Vector	117	197	391	64	73
Arithmetic DB	207	343	732	718	144
Connection	47	69	194	298	51
Coord03	117	112	187	23	27
Coord04	15	33	80	12	20
Elevator	214	458	1018	185418	649
Jigsaw	98	47	105	11	24
Knight	96	209	343	16	59
Local	23	18	46	7	17
StringBuffer	12	24	47	8	19
NASA	74	67	101	6	21
Store	601	559	892	197	136
UnderReporting	20	23	56	5	17
VectorFail	25	51	127	78	31

vai ser proporcional ao número de ramos de *parsing* explorados. A utilização de memória não é um problema pois a complexidade espacial é determinada pelo tamanho da tabela de *parsing* e pelo tamanho da maior árvore de *parsing*. A utilização de memória não é afetada pelo número de árvores de *parsing* porque o nosso *GLR parser* explora os ramos de *parsing* em profundidade em vez de em largura. Isto é possível porque nunca temos árvores de *parsing* de altura infinita devido à deteção de ciclos improdutivos.

7 Trabalho Relacionado

A programação por contrato foi introduzida por Meyer [19] como uma técnica para facilitar a escrita de código robusto e promover a reutilização de código baseada em contratos entre programas e objetos. Estes contratos especificam as condições necessárias para métodos do objeto poderem ser chamados, garantindo a sua correta semântica caso as pré-condições sejam cumpridas.

Cheon et al propôs a utilização de contratos para especificar protocolos de utilização de objetos [7]. O autor apresenta uma análise dinâmica para a verificação destes contratos. Isto contrasta com o nosso trabalho que valida estaticamente os contratos. Beckman et al apresentam uma metodologia baseada em *typestate*, para verificar estaticamente que um protocolo de um objeto é cumprido [4]. Esta aproximação requer que o programador faça explicitamente *unpack* dos objetos antes de invocar os seus métodos. Hurlin [14] estende o trabalho de Cheon para suportar protocolos em cenários concorrentes. São adicionados ao protocolo operadores para permitir especificar que métodos podem ser executados em concorrência, e pré-condições que têm que ser satisfeitas para a execução de determinado método. A análise pode ser verificada estaticamente usando um *theorem prover*. Em geral aproximações baseadas em *theorem proving* são limitadas pois são ineficientes e, normalmente, indecidíveis para lógicas não triviais.

Muitos trabalhos podem ser encontrados sobre violações de atomicidade. Alguns destes trabalhos podem potencialmente ser usados para a inferência automática dos contratos de módulos que apresentámos neste artigo. Artho et al

definem em [2] a noção de *High-Level Data Races*, que caracteriza sequências de operações atómicas que devem ser corridas de no mesmo contexto atómico para evitar problemas de atomicidade. As *High-Level Data Races* não capturam na totalidade as violações de atomicidade que podem ocorrer num programa. Praun and Gross estendem a aproximação de Artho para detetar potenciais anomalias na execução de métodos de um objeto e aumentam a precisão da análise distinguindo acessos de leitura e escrita sobre variáveis partilhadas entre *threads* [24]. Um refinamento adiciona às *High-Level Data Races* foi introduzido por Pessanha em [9] relaxando as propriedade definidas por Artho o que resulta numa melhor precisão da análise. Farchi et al propõe ainda uma metodologia para a deteção de violações de atomicidade em utilizações de módulos [10], baseado na definição de *High-Level Data Race*. Outro tipo de violações de atomicidade comuns quando se compõe operações executadas atomicamente são os *stale value errors*. Estas anomalias são caracterizadas pela utilização de valores obtidos de forma atómica em múltiplas operações atómicas, podendo estes estar desatualizados e comprometer a correção programa. Várias análises foram desenvolvidas para detetar este tipo de problema [3, 5, 9]. Diversas análises de verificação de violações de atomicidade podem ser encontradas na literatura, baseadas em padrões de acesso a variáveis suscetíveis a causar anomalias [17, 23], sistemas de tipos [6], invariantes semânticos [8], e outras metodologias especificas [11, 12, 13].

8 Conclusões

Neste artigo apresentamos o problema da ocorrência de violações de atomicidade na utilização de um módulo, independentemente dos seus métodos estarem individualmente sincronizados por algum mecanismo de controlo de concorrência. Para isso propomos um solução baseada na metodologia de programação por contrato. Estes contratos determinam quais as sequências de chamadas a métodos do módulo que deve ser executadas de forma atómica.

Apresentamos uma análise estática para a verificação deste contratos. A análise proposta extrai o comportamento do programa cliente em relação à utilização do módulo e verifica que este cumpre o contrato. A fase de extração do comportamento do programa é versátil, podendo ser facilmente adaptada para outras análises baseadas no comportamento do *control flow* do programa.

Foi implementado um prototipo e os resultados experimentais mostram que a análise tem boa precisão e é eficiente.

Referências

1. Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.
2. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, December 2003.
3. Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. *Automated Technology for Verification and Analysis*, pages 150–164, 2004.
4. Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, October 2008.

5. M. Burrows and K.R.M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, 2004.
6. Luís Caires and João C. Seco. The type discipline of behavioral separation. *SIGPLAN Not.*, 48(1):275–286, January 2013.
7. Yoonsik Cheon and Ashaveena Perumandla. Specifying and checking method call sequences of java programs. *Software Quality Control*, 15(1):7–25, March 2007.
8. R. Demeyer and W. Vanhoof. A framework for verifying the application-level race-freeness of concurrent programs. In *22nd Workshop on Logic-based Programming Environments (WLPE 2012)*, page 10, 2012.
9. Ricardo J. Dias, Vasco Pessanha, and João M. Lourenço. Precise detection of atomicity violations. In *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, November 2012. HVC 2012 Best Paper Award.
10. Eitan Farchi, Itai Segall, João M. Lourenço, and Diogo Sousa. Using program closures to make an API implementation thread safe. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2012, pages 18–24, New York, NY, USA, 2012. ACM.
11. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, January 2004.
12. Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, November 2010.
13. Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 43(6):293–303, June 2008.
14. Clément Hurlin. Specifying and checking protocols of multithreaded classes. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 587–592, New York, NY, USA, 2009. ACM.
15. IBM's Concurrency Testing Repository.
16. Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.
17. J. Lourenço, D. Sousa, B. Teixeira, and R. Dias. Detecting concurrency anomalies in transactional memory programs. *Computer Science and Information Systems/ComSIS*, 8(2):533–548, 2011.
18. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, March 2008.
19. Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.
20. Vasco Pessanha. Verificação prática de anomalias em programas de memória transaccional. Master's thesis, Universidade Nova de Lisboa, 2011.
21. Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, January 1987.
22. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.
23. M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGPLAN Notices*, volume 41, pages 334–345. ACM, 2006.
24. C. Von Praun and T.R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.