Vasco Pessanha

Licenciatura em Engenharia Informática

Verificação Prática de Anomalias em Programas de Memória Transaccional

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João M. S. Lourenço, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutor António M. L. C. A. Ravara
Arguente:	Prof. Doutor Paolo Romano
Vogal:	Prof. Doutor João M. S. Lourenço

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Verificação Prática de Anomalias em Programas de Memória Transaccional

Agradecimentos

Apesar da total dedicação e empenho que coloquei neste trabalho, não teria sido possível terminá-lo sem a ajuda preciosa de algumas pessoas. Este espaço é dedicado àqueles que deram contributos de natureza diversa para que esta dissertação fosse realizada.

Sem qualquer margem para dúvidas, este trabalho não teria ido tão longe se não tivesse sido constantemente apontado para o caminho certo. Por isso, não posso deixar de expressar o meu agradecimento incondicional ao meu orientador, João Lourenço, que tantas vezes me obrigou a parar para assentar ideias, conseguir olhar mais em frente e escolher novos e melhores caminhos. Nunca o seu apoio, atenção e companhia deixaram de estar presentes em momentos (e noites!) de maior pressão e desespero.

Não posso também deixar de agradecer ao Ricardo Dias, por tantas vezes negligenciar o seu próprio trabalho de forma a poder apoiar o desenvolvimento do meu. As suas contribuições para o desenvolvimento deste projecto são inumeráveis, mas gostava de destacar o seu apoio no planeamento e acompanhamento deste trabalho, a orientação na familiarização com a ferramenta Soot e na definição formal das análises descritas, e ainda várias iterações de revisão deste documento.

Gostaria também de expressar um agradecimento especial à Joana Neves e aos meus pais, Tomaz e Hélia, por terem estado sempre presentes tentando acompanhar os desafios, conquistas, alegrias e tristezas inerentes a este trabalho.

Aos meus colegas de trabalho e amigos que desenvolveram a sua tese ao mesmo tempo que eu, por estarem diariamente ao meu lado partilhando cada batalha deste trabalho. Por isto, e porque a produtividade do meu trabalho está fortemente dependente de uns intervalos retemperadores, gostaria de destacar os nomes de Ricardo Alves, João Vaz, Pedro Martins, João Saramago e Valter Balegas.

Apesar de não estarem ligados directamente a esta dissertação, gostaria de agradecer ao meu irmão Miguel e aos meus amigos João Miguel, Mafalda Catrau, João Cunha, Ricardo Miragaia, Miguel Vieira, Jaime Ribeiro, Filipe Louraço, Teresa Andrade, Francisco Mesquita, Catarina Ferreira, Afonso Nascimento, João Horta, entre outros, sem os quais não teria conseguido balancear o esforço exigido por este trabalho e a minha vida social.

Finalmente, o trabalho reportado nesta dissertação foi parcialmente financiado pelo

Centro de Informática em Tecnologias da Informação (CITI/FCT/UNL – PEst-OE/EEI/UI0527/2011) e pela Fundação para a Ciência e Tecnologia (FCT/MCTES), no contexto dos projectos Byzantium (Refª PTDC/EIA/74325/2006), Synergy-VM (Refª PTDC/ EIA-EIA/113613/2009) e Euro-TM COST Action IC1001.

“The only place where success comes before work is in the dictionary."

— *Donald Kendall*

Resumo

A Memória Transaccional (MT) é uma nova abordagem ao controlo de concorrência, baseada no conceito de transacção dos sistemas de gestão de base de dados. Ao contrário dos modelos baseados em *locks*, em MT podem existir diversos processos a aceder optimística e simultaneamente à mesma região crítica. Assim, uma transacção executa como se fosse única no sistema sendo que, no final, os resultados tornam-se permanentes ou são descartados (*rolled-back*) consoante a existência ou não de conflitos. Apesar de potenciar um melhor desempenho e utilidade, a MT é ainda uma tecnologia prematura, carecendo de ferramentas e trabalho que comprovem o seu potencial.

A programação concorrente é difícil e propensa a erros. Muitos destes erros são decorrentes de anomalias relacionadas com o acesso concorrente a dados partilhados. Apesar de alguns autores defenderem que os programas em MT são menos propensos a erros, estes também podem exibir anomalias concorrentes, tais como *high-level dataraces*, i.e., delimitações incorrectas do escopo das transacções, ou *stale-value errors*, que correspondem a transposições erróneas de variáveis de um bloco atómico para outro.

Programas com este tipo de anomalias podem exibir comportamentos imprevisíveis ou erróneos, não cumprindo os objectivos para os quais foram concebidos.

Este trabalho visa o desenvolvimento de algoritmos, baseados em análise estática, para a detecção de anomalias de programas escritos no paradigma transaccional. Estes algoritmos estão consolidados numa infraestrutura — MoTH — que agrega um conjunto de *plugins* que detectam anomalias específicas de programas Java ByteCode.

Com este trabalho pretendemos provar que a utilização de análise estática na detecção de *high-level dataraces* e de *stale-value errors* é uma solução viável que permite obter resultados com um nível de precisão razoável.

Palavras-chave: Violações de Atomicidade, High-Level Datarace, Análise Estática, Concorrência, Memória Transaccional por Software

Abstract

Transactional Memory (TM) is an approach to concurrency control in general purpose programming languages that inherits the concept of transaction from the database setting. Unlike other language constructs such as locks, TM has an optimistic approach to concurrency control by allowing more than one thread to access simultaneously the same critical section. A transaction always executes as if it is alone in the system, and in the end its effects are undone (rolled back) if it conflicts with another concurrent transactions. In spite of the potential for increasing scalability and performance, TM is a recent and developing programming model and still has a very limited impact in real-world applications.

Designing and developing concurrent software is difficult and error prone. Concurrent programs exhibit concurrency anomalies that originate faults and failures. Despite some claims that TM programs are less error prone, they still exhibit concurrency anomalies such as high-level dataraces, i.e., wrong delimitations of transactions' scope, and stale-value errors, that occur when the value of a shared variable jumps from an atomic block to another.

Programs with this kind of anomalies can exhibit unpredictable and wrong behaviour, not fulfilling the goals for which they were conceived.

This work aims the detection of anomalies through static analysis of transactional Java ByteCode programs that execute in strong atomicity. A extensible and flexible framework is proposed, which can be extended with plugins that detect specific types of anomalies.

With this framework we expect to prove that high-level dataraces and stale-value errors can be detected with reasonable precision through static analysis.

Keywords: Atomicity Violation, High-Level Datarace, Static Analysis, Concurrency, Software Transactional Memory

Conteúdo

Lista de Figuras

Listagens

1

Introdução

1.1 Contexto

O aumento da velocidade dos relógios dos processadores parou de aumentar, já que as necessidades de energia resultam na dissipação de muito calor e o sobreaquecimento do material põe em causa o bom funcionamento do sistema. Assim, o desenho de processadores sofreu alterações significativas, passando de uma procura de velocidades de relógios cada vez mais elevadas para a utilização de processamento multi-core. No entanto, para que esta revolução tecnológica seja vantajosa, tem de ocorrer simultaneamente uma revolução do software existente, permitindo um melhor aproveitamento dos recursos disponibilizados.

Contudo, a programação concorrente é indubitavelmente mais complexa e propensa a erros que a programação sequencial. Muitos destes erros são decorrentes de anomalias relacionadas com o acesso concorrente a dados partilhados e devem-se à dificuldade do programador prever todos os cenários possíveis de execução, i.e., em prever toda a combinatória de possíveis interacções entre os processos que vão executar.

Entre estas anomalias encontram-se os *dataraces* e os *stale-value errors*, que resultam na imprevisibilidade do comportamento do programa. Existem duas classes de *dataraces*: os *low-level dataraces* que ocorrem quando dois processos acedem a uma região partilhada simultaneamente, sendo que pelo menos um dos acessos é uma escrita, e os *high-level dataraces* que estão relacionados com cenários onde pelo menos duas transacções distintas deviam ser encapsuladas numa única. Um *stale-value error* ocorre quando é feita uma cópia do valor de uma variável dentro do escopo de um bloco sincronizado, que é posteriormente utilizada noutro bloco sincronizado. Se, entre estes, o valor da variável original for alterado, então o valor guardado torna-se obsoleto (*Stale-Value*).

Para evitar este tipo de anomalias, os processos têm de ser correctamente sincronizados de forma a protegerem o acesso aos dados partilhados. Actualmente, os *locks* estão entre os mecanismos de sincronização mais utilizados. No entanto, o uso de sistemas baseados em *locks* impõe um compromisso entre *coarse-grained locking*, que simplifica o desenvolvimento dos programas mas que pode limitar consideravelmente a exploração da concorrência, e *fine-grained locking*, que promove a concorrência mas é mais propício a erros de programação necessitando, portanto, de um desenho mais cuidadoso.

A Memória Transaccional (MT) [Gue10, HLR10] é uma abordagem à programação concorrente que tenta conjugar a performance dos sistemas *fine-grained* com a simplicidade dos sistemas *coarse-grained* [RHW10]. A MT utiliza o conceito de transacção dos sistemas de gestão de bases de dados (SGBD) para encapsular um determinado bloco de código que deve ser executado atomicamente. Cada transacção executa e, no final, tenta fazer *commit*, i.e., tornar os seus resultados permanentes em memória verificando um conjunto de propriedades. Consoante a transacção obedece ou não às propriedades exigidas, esta sucede ou as suas acções são retrocedidas (*rolled-back*).

Apesar de ter diversas vantagens sobre os sistemas baseados em *locks*, a MT não garante a ausência de erros no acesso a dados partilhados, nomeadamente de *dataraces* ou *stale-value errors*.

1.2 Problema

Mesmo que os programas executem em ambientes de atomicidade forte [BLM05, BLM06] onde todos os acessos a variáveis globais são encapsulados num bloco sincronizado, estando portanto isento de *low-level dataraces*, é ainda possível que um programa exiba um comportamento anómalo devido à presença de outro tipos de anomalias de alto nível como, por exemplo, *high-level dataraces* ou *stale-value errors*. Estes tipos de anomalias resultam na imprevisibilidade do comportamento dos programas e, por esta razão, devem ser detectados e corrigidos. Ao longo deste documento, por questões de legibilidade e simplicidade, utilizaremos o termo *dataraces* para nos referirmos a este tipo de anomalias de alto nível.

Vejamos o exemplo representado na Listagem 1.1, onde todos os acessos feitos a dados partilhados, ou seja às variáveis x e y, são feitos dentro de blocos sincronizados com o mesmo *lock*. Este programa não tem *low-level dataraces* mas pode, no entanto, exibir um comportamento incorrecto, onde o resultado da execução das operações reset e swap não corresponde ao par $\langle 0, 0 \rangle$. Considere-se o caso em que um processo p_1 executa a operação reset mas que, entre os dois blocos sincronizados do mesmo, outro processo p_2 executa a operação swap. Assim, quando p_1 obtém novamente o *lock*, as variáveis do par já foram trocadas, fazendo com que o segundo bloco sincronizado do método reset não tivesse qualquer tipo de efeito. Esta sequência das operações é ilustrada na Figura 1.1, onde os dois blocos sincronizados do método reset são representados por resetX e resetY, respectivamente.

Listagem 1.1: Exemplo de um *high-level datarace* (retirado de [AHB04])

```
1  public void swap() {
2    int oldX;
3    synchronized (lock) {
4      oldX = coord.x;
5      coord.x = coord.y; // swap X
6      coord.y = oldX; // swap Y
7    }
8  }
9  public void reset() {
10   synchronized (lock) {
11     coord.x = 0;
12   } // inconsistent state (0, y)
13   synchronized (lock) {
14     coord.y = 0;
15   }
16 }
```

Existe já um conjunto diversificado de técnicas e ferramentas que fazem a detecção de *dataraces* em ambientes de atomicidade forte, dirigidos para a programação com *locks*. No entanto, existe ainda uma carência significativa de trabalhos propostos a detectar *dataraces* em programas transaccionais.

$$\begin{array}{ccccccc} & & resetX(p_1) & & swap(p_2) & & resetY(p_1) \\ (x,y) & \longrightarrow & (0,y) & \longrightarrow & (y,0) & \longrightarrow & (y,0) \end{array}$$

Figura 1.1: Exemplo de uma sequência de operações que ilustra o *datarace* do exemplo da Listagem 1.1

Além disso, as ferramentas propostas na literatura pecam tanto na diversidade de anomalias que se propõem a detectar, como na precisão dessa mesma detecção. Por um lado, a maioria das ferramentas apresentadas na literatura detectam apenas um tipo específico de *dataraces* obrigando o utilizador a analisar o seu programa com diversas ferramentas de forma a garantir o bom funcionamento do mesmo. Por outro lado, nas ferramentas que se propõem a resolver esta questão abrangendo um conjunto mais amplo de anomalias, são reportados muitos falsos positivos obrigando o utilizador a verificar manualmente se todos os conflitos reportados correspondem efectivamente a anomalias reais.

1.3 Hipótese

Será que a análise estática é uma abordagem apropriada e viável na detecção de *dataraces* em programas que utilizam o paradigma de Memória Transaccional?

1.4 Aproximação

A detecção de *dataraces* será dirigida a programas cujo controlo de concorrência segue uma semântica transaccional num ambiente de atomicidade forte, através da análise estática do seu Java ByteCode [LY99].

Para tal, serão desenvolvidos algoritmos e técnicas de análise estática que permitirão a detecção de *dataraces*. Para validar estes algoritmos, será desenvolvida uma ferramenta de análise estática genérica e extensível, à qual poderão ser ligados *plugins*, chamados Sensores, desenhados para detectar um determinado tipo de *datarace*. Assim, numa primeira fase, a ferramenta analisará o código compilado do programa recolhendo e apurando um conjunto amplo de informação que será posteriormente disponibilizado como *input* para os Sensores. Nesta fase, identificamos os diferentes processos do programa concorrente, utilizamos uma extensão do algoritmo de *view consistency* [AHB03] para computar as suas *views* e analisamos as dependências entre as variáveis do mesmo. Depois, numa segunda fase, toda esta informação será disponibilizada a cada Sensor que executará um algoritmo de detecção de um *datarace* específico.

Finalmente, esta ferramenta será implementada através do desenvolvimento de uma extensão da ferramenta Soot [Soo], uma das ferramentas de análise de Java ByteCode mais maduras e complexas.

1.5 Contribuições

Entre as contribuições deste trabalho destacam-se:

- Extensão do formalismo de *view consistency* [AHB03], de forma a incorporar a distinção entre acessos de leitura e de escrita.

- Algoritmos e técnicas de análise estática que permitem apurar dependências de dados e de controlo entre as variáveis globais de um programa Java ByteCode.

- Uma infraestrutura extensível de análise estática de programas escritos em Java ByteCode, com os *plugins* necessários à detecção de *high-level dataraces* e *stale-value errors*.

- Um *benchmark* concorrente, Store, que permite a validação de ferramentas de detecção de *dataraces*.

- Um repositório organizado [Rep] com um conjunto amplo de testes, desenvolvidos por nós ou utilizados na literatura para validar trabalhos relacionados com a detecção de *dataraces*.

1.6 Publicações

A descrição e avaliação de uma versão inicial deste trabalho foi já publicada em *9th International Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (PADTAD 2011) [PDL+11]. Um segundo artigo focado nas análises de dependências desenvolvidas foi submetido ao *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP 2012), encontrando-se actualmente em avaliação. Um repositório organizado com o código fonte do protótipo desenvolvido e dos testes utilizados na validação do nosso trabalho é disponibilizado em [Rep].

1.7 Estrutura do Documento

No Capítulo 1 fez-se uma pequena introdução ao tema desta dissertação. Começou-se por fazer um breve enquadramento do conceito de Memória Transaccional e foram referidas algumas anomalias inerentes à programação concorrente. Depois, foram apontados e discutidos os problemas relacionados com a detecção deste tipo de anomalias e foi descrita uma possível abordagem para os resolver bem como as contribuições esperadas do nosso trabalho.

No Capítulo 2 será apresentado o trabalho relacionado desta tese. Começar-se-á por aprofundar o conceito de Memória Transaccional, estudando algumas propriedades e conceitos a ter em conta no desenho deste tipo de sistemas que serão úteis para este trabalho. Serão estudadas mais pormenorizadamente algumas anomalias inerentes à programação concorrente. Introduzir-se-á o conceito de análise de programas, diferenciando-se análise dinâmica e estática, e será feito um estudo de algumas técnicas de análise estática que serão utilizadas neste trabalho. Finalmente, será apresentado um estudo comparativo de algumas ferramentas de análise de programas que trabalham sobre Java ByteCode, que nos permitiu escolher a ferramenta que serviu como base do protótipo desenvolvido no contexto deste trabalho.

No Capítulo 3 apresentaremos a ferramenta MoTH que permite a detecção estática de *dataraces* em programas transaccionais escritos em Java ByteCode. Primeiro, apresentaremos a estrutura da ferramenta identificando as duas fases da detecção de *dataraces*. Depois, para cada uma das fases da análise da ferramenta, apresentaremos uma descrição do seu funcionamento.

No Capítulo 4 será apresentada a validação da ferramenta MoTH. Para tal, descreveremos o funcionamento da ferramenta, seguindo passo a passo a análise de um dos testes utilizados nesta validação, e apresentaremos todos os resultados obtidos.

Finalmente, tendo sido apresentado todo o trabalho, no Capítulo 5 é feito um resumo do mesmo sendo apresentadas as conclusões. Ainda neste capítulo, apresentaremos algumas propostas de trabalho futuro que permitirão colmatar algumas limitações deste trabalho.

2

Trabalho relacionado

2.1 Introdução

Para entender o conteúdo deste documento, é necessária a compreensão de alguns temas e áreas relevantes ao nosso trabalho. Neste capítulo será feita uma breve descrição de um conjunto de tópicos relevantes, que nos ajudará a contextualizar e introduzir o nosso trabalho, bem como das abordagens relevantes e relacionadas com o nosso trabalho.

Na Secção 2.2, começaremos por abordar o conceito de Memória Transaccional (MT), discutindo algumas questões de desenho e implementação de sistemas de MT, já que este constitui a base do nosso trabalho. Depois, na Secção 2.3, abordaremos algumas das anomalias relacionadas com acessos concorrentes a dados partilhados, nomeadamente os *dataraces*. Apesar destas anomalias serem discutidas de forma genérica neste capítulo, o seu estudo pode ser aplicado ao conceito de MT. Finalmente, na Secção 2.4, serão estudadas algumas técnicas de análise de programas que permitem a detecção deste tipo de anomalias, sendo feito também um estudo das ferramentas actuais de análise de programas.

2.2 Memória Transaccional

Apesar de, nos últimos anos, a performance dos processadores ter vindo a crescer exponencialmente, o processamento sequencial poderá vir a tornar-se obsoleto. O aumento continuado da velocidade do relógio dos processadores tem-se tornado inviável, já que as elevadas necessidades de energia resultam na dissipação de muito calor e o sobreaquecimento do material põe em causa o bom funcionamento do sistema. Além disto, a performance dos sistemas sequenciais depende da velocidade dos seus processadores

e, por esta razão, tem vindo a estagnar. Assim, como as exigências de poder computacional por partes dos utilizadores continuam a crescer, os sistemas concorrentes (quer multi-core, quer multi-processadores) apresentam-se como uma boa aproximação para responder a estas necessidades.

No entanto, apesar da necessidade da programação concorrente estar a aumentar, esta é indubitavelmente mais difícil do que a programação sequencial, tanto na escrita de programas como no seu teste, validação e debugging. A existência de múltiplos cenários de execução e de acessos a dados partilhados por parte de diferentes processos concorrentes, faz com que a execução seja não-determinística. Sendo assim, um erro de programação num determinado cenário de execução pode permanecer indetectável por bastante tempo se esse cenário não for executado.

Desde os anos 60, os sistemas de gestão de bases de dados (SGBD) fazem a gestão de acessos concorrentes a bases de dados, permitindo aos utilizadores especificar o que querem fazer de uma forma intuitiva, minorando o problema inerente à concorrência de acessos. Isto faz do modelo transaccional usado nos SGBD um bom modelo para controlo de acessos concorrentes a dados partilhados.

Memória transaccional (MT) [Gue10, HLR10] é uma abordagem optimista à programação concorrente, que fornece controlo de acessos a dados partilhados baseando-se no conceito de transacção dos SGBD. Uma transacção em memória consiste num conjunto de operações que é executado atomicamente, parecendo instantâneo e indivisível para o resto do sistema.

O uso de sistemas baseados em *locks* impõe um compromisso entre *coarse-grained locking*, que pode reduzir significativamente a concorrência entre os processos, e *fine-grained locking* que é mais propício a erros de programação e que necessita de um desenho mais cuidadoso. A MT tenta conjugar a performance dos sistemas *fine-grained* com a simplicidade da programação *coarse-grained* [RHW10]. Para isso, a MT oferece ao programador construções linguísticas que identificam um bloco transaccional, relegando a sua implementação, incluindo a detecção e resolução de conflitos, para o motor transaccional. Tal como nos sistemas de bases de dados, uma transacção executa e tenta suceder (fazer *commit*) verificando um conjunto de propriedades. Consoante a transacção obedece ou não às propriedades exigidas, o estado resultante torna-se permanente ou é retrocedido (*rolled-back*).

Na Secção 2.2.1 fazer-se-á uma pequena comparação entre os SGBDs e os sistemas de MT. Depois, na Secção 2.2.2, serão discutidos tipos de controlo de concorrência pessimistas e optimistas para suporte de MT. Na Secção 2.2.3, serão discutidos os conceitos de Atomicidade, Opacidade e Isolamento de transacções, relevantes para garantir que as transacções obedecem às propriedades requeridas pelo sistema. Finalmente, na Secção 2.2.4, será feito um breve resumo dos assuntos abordados.

2.2.1 SGBDs e sistemas de MT

Nos sistemas de bases de dados, todas as transacções obedecem a quatro propriedades conhecidas, denominadas propriedades ACID, que podem ser adaptadas a sistemas de MT:

- Atomicidade – Todas as transacções obedecem à política "all or nothing": ou todas ou nenhuma operação é executada. Por outras palavras, todos os resultados de uma transacção sucedida têm de ser visíveis pelo resto do sistema, enquanto os efeitos de uma transacção abortada permanecem invisíveis. Quando respeitada, esta propriedade previne que uma actualização ocorra apenas parcialmente deixando os dados inconsistentes.

- Consistência – Esta propriedade garante que o sistema se apresenta consistente antes e depois de cada transacção (independentemente de esta suceder ou não). Apesar de poder executar num estado inconsistente, uma transacção tem de deixar o sistema consistente quando sucede.

- Isolamento – Tal como o nome sugere, o isolamento dita que uma transacção se comporta como se fosse única no sistema, i.e., sem se aperceber de outras que estejam a executar. Assim, para uma transacção, só são visíveis os resultados de transacções que já sucederam.

- Durabilidade – Durabilidade é a propriedade que garante que, se uma transacção sucede, então os resultados são permanentes mesmo que o sistema falhe (i.e., os dados são guardados de forma permanente em memória não-volátil, como um disco).

Os SGBD constituem a base e motivação dos sistemas de MT. No entanto, como estes sistemas trabalham sobre tipos de memórias com diferentes características, as técnicas dos SGBD não são totalmente portáveis para os sistemas de MT. Estas diferenças têm de ser consideradas na implementação dos sistemas de MT. São enumeradas de seguida algumas das mais importantes:

- O tempo de acesso a dados é significativamente diferente nos sistemas de MT e nos SGBD. Sendo as bases de dados geralmente armazenadas em disco (cujo tempo de acesso é sensivelmente entre 5 e 10 milissegundos), os SGBD podem chegar a executar milhões de instruções enquanto fazem uma petição de dados. Como os sistemas de MT trabalham sobre memória, não é possível fazer qualquer tipo de computação durante o acesso a dados. Para além disso, para suportar a detecção de conflitos, o próprio sistema de MT faz leituras e escritas adicionais em memória sempre que é feito um acesso no programa, penalizando negativamente o desempenho do sistema.

- Como a MT lida maioritariamente com acessos a memória volátil, a propriedade de Durabilidade não se aplica nesse contexto.

- Nos SGBD qualquer tipo de acesso à base de dados é feito através de uma transacção, mesmo que esta não tenha sido declarada explicitamente. No entanto, em alguns modelos de Memória Transaccional, é possível ter acessos a dados partilhados fora do escopo transaccional.
- Ao contrário dos SGBD, os sistemas de MT têm de coexistir de forma consistente com diferentes tipos de linguagens de programação, paradigmas, software e sistemas de operação. Para que os programadores não tenham dificuldades em usar modelos de MT, as alterações nas linguagens de programação têm de ser minimizadas.
- As operações suportadas pelos SGBD estão apenas relacionadas com leituras e escritas de dados, sendo relativamente fácil fazer *roll-back* caso a transacção aborte. No entanto, nos sistemas de MT, podemos implementar operações mais complexas (por exemplo operações I/O, comunicações numa rede, acessos a bibliotecas externas, etc.), que podem ser difíceis ou mesmo impossíveis de reverter caso a transacção aborte.

2.2.2 Controlo de concorrência optimista e pessimista

Existem dois tipos de abordagens no que diz respeito ao controlo de concorrência: pessimistas e optimistas. No primeiro caso, quando existe a possibilidade de ocorrência de um conflito, esta é imediatamente detectada e o conflito é evitado. Por outro lado, uma abordagem mais optimista pode permitir que dois ou mais processos executem como se fossem únicos no sistema, deixando a detecção e resolução de conflitos para o momento em que os resultados se tornam permanentes.

Apesar de geralmente os modelos de MT serem optimistas quando comparados com modelos baseados em *locks*, é possível distinguir implementações de MT pessimistas e optimistas. Se o sistema detecta um conflito entre duas ou mais transacções, pode abortar imediatamente uma das transacções conflituosas, ou permitir que estas prossigam resolvendo o conflito quando uma delas tentar suceder. Na primeira abordagem, pode ocorrer o caso em que o sistema aborta uma transacção porque está em conflito com outra que já ia abortar independentemente da primeira. Por outro lado, na segunda abordagem, ao deixarmos que duas transacções conflituosas continuem a executar, podemos estar a efectuar computação desnecessária (se a transacção vai ser abortada, pode ser preferível que aborte assim que o conflito seja detectado não consumindo mais recursos).

Os modelos pessimistas e optimistas têm vantagens e desvantagens, sendo que a escolha do tipo de modelo a utilizar depende das características do sistema em questão. Uma política pessimista pode ser mais adequada num ambiente com elevada contenção e de grande probabilidade de conflitos (provavelmente o sistema aborta uma transacção que iria ser abortada mais tarde de qualquer maneira). Por outro lado, uma política optimista pode ser bastante eficaz em ambientes pouco conflituosos permitindo uma maior concorrência entre as transacções.

Existem ainda soluções híbridas que tentam combinar as vantagens das duas abordagens, por exemplo, usando uma política pessimista nas operações de escrita e uma política optimista nas operações de leitura [SATH+06].

2.2.3 Atomicidade, Opacidade e Isolamento

Tal como foi visto anteriormente, uma das grandes diferenças entre os sistemas de MT e os SGBD consiste no facto de alguns dos primeiros terem de considerar interacções entre acessos transaccionais e não-transaccionais.

Nos sistemas de bases de dados, todos os acessos são feitos dentro do escopo de uma transacção, mesmo que esta não seja declarada explicitamente. Assim, um programa é um conjunto de transacções, em oposição a um conjunto de instruções. No entanto, em alguns sistemas de MT uma transacção tem de ser declarada explicitamente, caso contrário os acessos são considerados não-transaccionais. Como muitos destes sistemas não fazem detecção de conflitos entre acessos transaccionais e não-transaccionais, podem ocorrer comportamentos inesperados.

Blundell et al. [BLM05, BLM06] introduz os termos de *atomicidade fraca* e *atomicidade forte*. Enquanto a primeira garante a semântica transaccional apenas entre transacções, a atomicidade forte também garante que todas as variáveis partilhadas são forçosamente acedidas num contexto transaccional. Por vezes, os termos isolamento fraco/forte são também utilizados na literatura, em vez de atomicidade fraca/forte. O uso da palavra *isolamento* refere-se ao facto de, se uma transacção consegue ver um estado intermédio de outra que está a executar, então não está isolada dos efeitos ou resultados dessa mesma transacção. Por outro lado, o termo *atomicidade* refere-se ao facto de nesse caso a transacção não ser vista como um átomo ou instrução única.

Em sistemas de atomicidade fraca podem ocorrer diferentes tipos de problemas ou anomalias relacionados com interacções entre acessos transaccionais e não-transaccionais. A título de exemplo, considere-se uma transacção que lê por diversas vezes o valor de uma determinada variável dentro de uma transacção. Como as transacções executam como se fossem únicas no sistema, espera-se que o valor lido seja sempre o mesmo. No entanto, se entre duas leituras houver uma actualização dessa variável fora do escopo transaccional, então essa alteração será observada pela transacção. Estes conflitos dão pelo nome de *low-level dataraces* e são aprofundados na Secção 2.3.

Como foi visto anteriormente, a propriedade de isolamento garante que uma transacção que sucede não vê estados intermédios de outras transacções. No entanto, segundo esta propriedade, uma transacção pode ser afectada por estados intermédios de outras transacções, desde que no final aborte e seja *rolled-back*.

Finalmente, a opacidade é uma propriedade das transacções mais restrita que o isolamento, que dita que qualquer transacção executa sem ter acesso a qualquer estado intermédio de outra, independentemente de suceder ou não. Assim, apesar de um sistema que garante a opacidade entre transacções garantir o isolamento destas, o contrário não

se verifica necessariamente.

2.2.4 Sumário

Nesta secção introduzimos o conceito de Memória Transaccional que representa a base do nosso trabalho. Diferenciámos abordagens pessimistas e optimistas e estudámos algumas propriedades importantes no desenho de um sistema de MT, que serão úteis tanto na compreensão do problema a que nos propomos resolver, como na compreensão da sua resolução em si.

A ferramenta MoTH propõe-se a fazer a detecção de *dataraces* em programas com a semântica transaccional. No entanto, os conflitos abordados neste documento são aplicáveis a outros tipos de mecanismos de controlo de concorrência. Assim, o conteúdo das próximas secções será apresentado de uma forma o mais genérica possível, ainda que possa sempre ser aplicado no âmbito específico da Memória Transaccional.

2.3 Anomalias

A programação concorrente é mais complexa do que a sequencial, gerando um novo leque de possíveis anomalias relacionadas com a má programação de acessos concorrentes a dados partilhados. Se, na programação sequencial, podemos considerar que os valores das variáveis de um programa correspondem aos valores observados no final da execução da última instrução, o mesmo não se verifica na programação concorrente. Isto deve-se à possibilidade de ocorrência de trocas de contexto, onde outro processo executa entre duas instruções do primeiro, alterando o estado das variáveis do programa. Estas trocas de contexto obrigam o programador a prever todos os cenários de execução, fazendo com que a sincronização de processos concorrentes seja extremamente complicada.

Um exemplo deste tipo de anomalias são os *dataraces* que têm sido alvo de intenso estudo na literatura. Dentro desta categoria de anomalias é possível distinguir os *low-level dataraces* e os *high-level dataraces*. Os primeiros, ocorrem apenas em ambientes de atomicidade fraca e estão relacionados com a falta de isolamento no acesso a dados partilhados. Os segundos estão associados a acessos parciais a conjuntos de variáveis que deveriam ser acedidos como um todo.

Mesmo que execute num ambiente de atomicidade forte [AHB03] e esteja isento de *high-level dataraces*, um programa pode ainda exibir erros de concorrência, como por exemplo *stale-value errors*.

A existência de *dataraces* ou *stale-value errors* pode resultar na imprevisibilidade do comportamento dos programas, que pode ser prejudicial para o bom funcionamento dos mesmos. Uma anomalia desta natureza pode fazer com que um programa entre num ciclo infinito, modifique o valor esperado de uma determinada variável, ou não ter qualquer tipo de efeito se, por exemplo, o cenário de execução à qual está associada não ocorrer.

Nas Secções 2.3.1 e 2.3.2 serão discutidas mais pormenorizadamente as classes de *low-level dataraces* e *high-level dataraces* respectivamente e, na Secção 2.3.3, o mesmo será feito com os *stale-value errors*. Em cada uma destas secções, serão discutidos os trabalhos relacionados mais relevantes na definição e detecção deste tipo de anomalias. Finalmente, na Secção 2.3.4, será feito um breve sumário dos assuntos abordados nesta secção.

2.3.1 Low-Level Dataraces

Introdução e Definição

Um *low-level datarace*, muitas vezes referido na literatura apenas como *datarace*, é um conflito que ocorre entre dois acessos a uma região crítica, quando pelo menos um deles é uma escrita e não existe sincronização entre eles. Formalmente, segundo a literatura, este tipo de anomalias ocorre quando as seguintes condições são verificadas entre dois acessos [CLL+02, OC03, SBN+97]:

1. Os dois acessos são feitos à mesma posição de memória e pelo menos um deles é uma escrita;
2. Os dois acessos são feitos por processos diferentes;
3. Não existe nenhum *lock* comum entre os dois processos quando os acessos são feitos;
4. Não existem garantias que um dos acessos ocorreu antes do outro, tornando-os potencialmente simultâneos.

A verificação da última condição baseia-se no estudo das relações *May-Happens-In-Parallel* [DS91], que procuram encontrar relações causais entre dois eventos (neste caso, entre os dois acessos a dados). Por exemplo, se um bloco sincronizado começar a ser executado depois de outro terminar, então acessos destes dois blocos não poderão estar envolvidos num conflito.

Actualmente é reconhecida a utilidade das técnicas e ferramentas de análise e detecção de *low-level dataraces* [CLL+02, OC03]. Assim, a criação deste tipo de ferramentas tem sido alvo de atenção da comunidade científica [CLL+02, TLS10]. Apesar deste tipo de anomalias ser referido neste documento por uma questão de completude, no nosso trabalho apenas nos propomos a detectar anomalias em ambientes de atomicidade forte excluindo, por isso, a possibilidade de ocorrência de *low-level dataraces*.

Exemplo

Como foi visto, para garantir que dois processos acedem sincronizadamente a um conjunto de variáveis partilhadas, tem que existir pelo menos um *lock* comum aos dois processos no momento em que os acessos são feitos. Na Listagem 2.1, x é uma variável partilhada e $l1$ e $l2$ são *locks* partilhados pelos blocos concorrentes B1, B2 e B3. O bloco

B1 está sincronizado correctamente com qualquer um dos outros dois, já que tem um *lock* comum com ambos. No entanto, o mesmo não se passa com os blocos B2 e B3 que não têm nenhum *lock* comum gerando, por isso, um *low-level datarace*.

Figura 2.1: Blocos atómicos concorrentes

```
// bloco B1
synchronized (l1,l2){
//acesso a x
}
```

```
// bloco B2
synchronized (l1){
//acesso a x
}
```

```
// bloco B3
synchronized (l2){
//acesso a x
}
```

Relação entre *dataraces* com *locks* e MT

Apesar de, ao longo desta secção, tanto a definição como o exemplo de *dataraces* serem baseados no mecanismo de sincronização com *locks*, o mesmo pode ser aplicado em sistemas de MT. Conceptualmente, uma transacção pode ser implementada através de um *lock* global, já que cada transacção é sincronizada com todas as outras do sistema. Teixeira et al. [TLS10] descreve as condições de um *datarace* em MT, mostrando como cada cenário corresponde a uma situação semelhante num sistema baseado em *locks*:

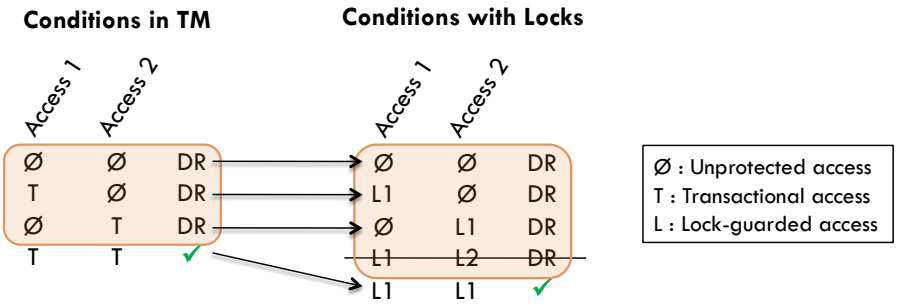


Figura 2.2: Condições de um *datarace* com transacções e com *locks* (retirado de [TLS10])

Assim, a terceira condição para a ocorrência de um *low-level datarace*, que dita que não existe nenhum *lock* comum entre os dois processos quando os acessos são feitos, pode ser adaptada para sistemas de MT. Nestes sistemas, para garantir que dois acessos estão dentro do escopo do *lock* global, ambos os acessos têm que ser feitos dentro do escopo transaccional. Note-se que, tendo isto em conta, sistemas de MT que garantem atomicidade forte garantem consequentemente a ausência de *low-level dataraces*.

2.3.2 High-Level Dataraces

Introdução e Definição

O facto de um programa estar isento de *low-level dataraces*, não implica necessariamente que este não exiba erros de concorrência. Ao contrário dos *low-level dataraces*, os *high-level dataraces* [AHB03, AHB04] não resultam da falta de isolamento do acesso a variáveis.

Este tipo de anomalia está associado a programas que exibem um comportamento errado, apesar de cada um dos acessos a dados partilhados estar protegido por um *lock*, já que operações que deviam ser executadas atomicamente estão a ser intercaladas com a execução de outros processos.

Assim, os *high-level dataraces* estão associados a acessos parciais a conjuntos que deviam ser acedidos de forma atómica, i.e., como um todo. Se tivermos um par de variáveis partilhadas que deve ser acedido atomicamente, acessos a uma única variável desse par podem gerar *high-level dataraces*. Uma das grandes dificuldades na detecção destes conflitos consiste em apurar que conjuntos devem ser acedidos atomicamente.

Exemplo

A Listagem 2.1 foi retirada de [AHB04] e apresenta um exemplo de um *high-level datarace*. Apesar de, no método `reset`, ambos os acessos às variáveis x e y estarem protegidos por um *lock*, é possível que um processo veja o estado (inconsistente) entre as duas actualizações. Assim, depois do valor de x ser actualizado com o valor 0, a operação `swap` pode executar, trocando os valores das variáveis, e x ficará com o valor inicial de y resultando num comportamento errado do programa.

Listagem 2.1: Exemplo de um *high-level datarace* (Réplica da Listagem 1.1)

```
1   public void swap() {
2     int oldX;
3     synchronized (lock) {
4       oldX = coord.x;
5       coord.x = coord.y; // swap X
6       coord.y = oldX; // swap Y
7     }
8   }
9   public void reset() {
10    synchronized (lock) {
11       coord.x = 0;
12    } // inconsistent state (0, y)
13    synchronized (lock) {
14      coord.y = 0;
15     }
16  }
```

Detecção de High-Level Dataraces

Existem já diversas aproximações à detecção de *high-level dataraces* em programas concorrentes, tanto estáticas [TLS10, VTD06, BBA08], dinâmicas [AHB03, WS03], ou híbridas [EQ07]. Algumas destas abordagens serão analisadas com mais detalhe devido à sua relevância e proximidade com a nossa.

Artho et al. [AHB03, AHB04] propõe o uso do conceito de *view consistency* na detecção deste tipo de anomalias. Uma *view* de um bloco sincronizado corresponde ao conjunto de variáveis partilhadas acedidas por este, quer seja através de uma leitura ou de uma

escrita. Por sua vez, as *maximal views* de um determinado processo são definidas como sendo aquelas que não estão totalmente incluídas noutra *view* do mesmo processo. Intuitivamente, uma *maximal view* representa um conjunto de variáveis que devem ser acedidas atomicamente. Assim, se as intersecções das *views* de um processo com a *maximal views* de outro processo não formarem uma cadeia entre si, então estamos perante uma violação do conceito de *view consistency*.

Para uma melhor compreensão deste conceito, voltemos ao exemplo anterior ilustrado na Listagem 2.1. Representamos por $V(F)$ e $M(F)$, respectivamente, os conjuntos de *views* e *maximal views* da função F. Tendo isto, as *views* de cada método correspondem aos conjuntos de variáveis partilhadas que são acedidas em cada bloco sincronizado do mesmo. Por sua vez, as *maximal views* são todas as *views* que não estão contidas noutra. Obtemos portanto: $V(swap) = M(swap) = \{\{x, y\}\}$ e $V(reset) = M(reset) = \{\{x\}, \{y\}\}$. Note-se que, por exemplo, se o segundo bloco sincronizado da função `reset` acedesse também à variável x, obteríamos os conjuntos: $V(reset) = \{\{x\}, \{x, y\}\}$ e $M(reset) = \{\{x, y\}\}$. Finalmente, como as intersecções das duas *views* da função `reset` com a *maximal view* da função `swap` não formam uma cadeia ($\{x\} \not\subseteq \{y\}$ e $\{y\} \not\subseteq \{x\}$), então a função `reset` tem uma anomalia, já que não acede de forma atómica ao conjunto formado pelas variáveis x e y.

Indubitavelmente, este trabalho consiste na base e motivação da nossa abordagem. Apesar da possibilidade de ocorrência de falsos positivos e negativos, os autores têm uma abordagem interessante analisando as relações entre variáveis em vez de interacções entre transacções.

Praun e Gross [vG03] introduzem o conceito de *method consistency*, uma extensão de *view consistency* distinguindo acessos de leitura e escrita. Baseados na intuição de que todas as variáveis acedidas dentro de um bloco sincronizado devem ser acedidas atomicamente, os autores definem o conceito de *method views*, relacionado com as *maximal views* de Artho. Esta abordagem detecta todos os *dataraces* detectados pelo algoritmo de *view consistency*, e outros que ficaram por detectar com este algoritmo. Na validação da ferramenta MoTH foram detectados *dataraces* que o algoritmo de *method consistency* não conseguiu detectar como, por exemplo, o teste *UnderReporting* descrito na Secção 6.12. Conseguimos ainda remover alguns dos falsos positivos gerados por esta abordagem.

Wang e Stoller [WS03] usam o conceito de atomicidade de threads para detectar e prevenir *dataraces*. Note-se que esta *atomicidade* tem um significado diferente da atomicidade das propriedades ACID. Aqui, a atomicidade está relacionada com o conceito de serialização entre transacções, que garante que qualquer execução concorrente de um conjunto de threads é equivalente a pelo menos uma execução sequencial dos mesmos.

Com o objectivo de diminuir o número de falsos positivos da abordagem anterior, Teixeira et al. [TLS10] introduz uma aproximação diferente na detecção deste tipo de anomalias. Motivados pela intuição de que a maioria dos *dataraces* estão relacionados com interacções entre blocos sincronizados consecutivos que deveriam ser encapsulados num único, os autores criam uma abordagem de detecção de *dataraces* baseada em padrões de

acessos anómalos. Assim, são criados os seguintes padrões:

1. **Read-write-Read ou RwR** – um processo lê um estado do sistema em duas ou mais transacções, podendo obter valores inconsistentes resultantes da conjugação de dois estados globais do sistema;

2. **Write-read-Write ou WrW** – um processo actualiza o estado do sistema em duas ou mais transacções, permitindo que outras observem estados intermédios inconsistentes;

3. **Read-write-Write ou RwW** – um processo lê o valor de uma variável partilhada e, com base no valor observado, actualiza o valor de outra. O valor lido inicialmente pode ter sido mudado entretanto por outro processo.

Finalmente, sempre que é verificado um dos padrões de acessos num programa, é reportado um *datarace*. À semelhança do nosso, este trabalho também detecta *dataraces* provenientes de acessos a uma única variável (*stale-value errors*).

Outra abordagem que utiliza o conceito de padrões anómalos na detecção estática de *dataraces* é o trabalho de Vaziri et al. [VTD06]. Os autores criam uma nova definição do conceito de *datarace*, através do levantamento teórico de todos os possíveis padrões anómalos de acessos concorrentes, abrangendo tanto *low-level dataraces* como *high-level dataraces*, provando ainda a completude dessa mesma lista. No entanto, esta abordagem necessita que o programador identifique explicitamente que variáveis devem ser acedidas atomicamente, eliminando um dos maiores desafios na detecção deste tipo de conflitos.

Beckman et al. [BBA08] apresenta uma análise *intra-procedural*, formalizada como um sistema de tipos, usando o conceito de permissões de acessos para detectar este tipo de anomalias em programas transaccionais escritos em Java. No entanto, esta abordagem necessita que o programador declare explicitamente os invariantes e permissões de cada objecto do programa.

Finalmente, Elmas et al. [EQ07] apresenta um sistema que monitoriza dinâmicamente a execução de programas Java lançando uma excepção *DataRaceException* quando um *datarace* está prestes a ocorrer. Este sistema suporta diferentes idiomas de sincronização permitindo, por exemplo, a combinação de transacções e blocos sincronizados com *locks*.

2.3.3 Stale-Value Errors

Introdução e Descrição

Um programa isento de *high-level dataraces* executando em atomicidade forte pode ainda exibir outras anomalias concorrentes de alto nível como, por exemplo, *stale-value errors*. Um *Stale-Value* [BL04] consiste na utilização do valor de uma cópia de uma variável que já não reflecte o seu valor corrente, i.e., está obsoleta. Estas anomalias estão relacionadas com variáveis partilhadas cujo valor sai do escopo de um bloco sincronizado para outro.

Entre os dois blocos, outro processo pode alterar o valor dessa variável fazendo com que o valor copiado pelo primeiro processo fique obsoleto.

Exemplo

A Listagem 2.2 ilustra um exemplo deste tipo de anomalias. Neste exemplo, um programa lê o valor da variável x através dum bloco sincronizado e, com base nesse valor, actualiza essa mesma variável noutro bloco sincronizado. No entanto, entre estes dois blocos, outro processo pode alterar o valor de x, resultando na utilização de um valor obsoleto por parte do primeiro processo.

Listagem 2.2: Exemplo de um *stale-value error*

```
1  public static void main(String[] args) {
2    int x = getX();
3    //x may have a stale-value
4    setX(x*x);
5  }
6
7  @Atomic
8  private int getX() {
9    return x;
10 }
11
12 @Atomic
13 private void setX(int value) {
14   x = value;
15 }
```

Outro exemplo deste tipo de anomalias, retirado de [TLS10], é ilustrado na Listagem 2.3. Neste exemplo é descrita a interface de uma estrutura de dados de capacidade máxima MAX_SIZE, onde é disponibilizado um método attemptToStore que permite a inserção segura de um novo elemento testando previamente se existe espaço livre na mesma. No entanto, apesar de ambas as funções hasSpaceLeft e store serem atómicas, é possível que a estrutura de dados exceda o tamanho máximo se estas forem intercaladas por outro processo que executa o mesmo código. Neste caso, apesar de não existir nenhum valor a passar directamente de um bloco atómico para outro, assume-se intuitivamente que o método store só deve ser chamado caso exista espaço livre na estrutura de dados.

Detecção de Stale-Value Errors

Tal como nos *high-level dataraces*, existem também já diversos trabalhos que abordam a detecção de *stale-value errors* [TLS10, FQ03, FF04, FFY08, BL04, vG03, AHB04, WS03].

Em [AHB04], Artho et al. apresenta uma técnica baseada em análise *Data-Flow* que permite a detecção de *stale-value errors*. Este algoritmo foi implementando usando análise estática e não depende de qualquer tipo de assunção ou anotações do utilizador. Apesar desta abordagem ser correcta segundo os conceitos de correcção (*sound*) e completude

Listagem 2.3: Outro exemplo de um *stale-value error* (retirado de [TLS10])

```
1   @Atomic
2   private boolean hasSpaceLeft() {
3     return (this.list.size() < MAX_SIZE);
4   }
5   @Atomic
6   private void store(Object obj) {
7       this.list.add(obj);
8   }
9   public void attemptToStore(Object obj) {
10    if (this.hasSpaceLeft()) {
11        // list may be full
12      this.store(obj);
13    }
14  }
```

(*completeness*), como aplicados por Flanagan et al. [FLL+02] no contexto de detecção de anomalias, os autores apenas consideram dependências de dados entre variáveis, deixando de fora as dependências de controlo que foram incluídas no nosso trabalho.

Utilizando o conceito de *method consistency* apresentado na Secção 2.3.2, Praun e Gross [vG03] fazem a detecção de *stale-value errors*, tal como de *high-level dataraces*. No entanto, tal como os autores referem no artigo, esta abordagem é incompleta existindo *stale-value errors* que não são detectados pela mesma.

Wang e Stoller [WS03] utilizam o conceito de atomicidade entre *threads* na detecção de *stale-value errors*, assim como de *high-level dataraces*. No entanto, esta abordagem é demasiado abrangente reportando um número significativo de falsos positivos.

Tal como foi referido, Teixeira et al. [TLS10] utiliza uma abordagem de detecção de padrões para detectar anomalias em ambientes de atomicidade forte. O padrão RwW (Read-Write-Write) foi especialmente desenhado para a detecção de *stale-value errors*. Com este padrão, os autores assumem que sempre que uma variável é lida num bloco atómico e escrita no bloco atómico seguinte, então existe um possível *stale-value error* já que a escrita pode ter sido feita com base no valor lido no bloco anterior. Esta assunção é demasiado forte podendo gerar tanto falsos negativos como positivos.

Em [BL04], Burrows et al. descreve um sistema implementado num compilador que permite detectar este tipo de anomalias. Para cada variável do programa, os autores criam uma segunda variável especial que dita se a primeira pode ou não estar obsoleta. Este sistema não depende de qualquer tipo de anotações por parte do utilizador e, segundo os autores, reporta um número de falsos alarmes suficientemente baixo para serem verificados manualmente pelo utilizador.

Em [FQ03], os autores apresentam um sistema de tipos que verifica a atomicidade de blocos de código para detectar *stale-value errors*. Contudo, o sistema descrito não foi aplicado a qualquer tipo de linguagem ou programa real.

Finalmente, Flanagan e Freund apresentam em [FF04] o Atomizer, um sistema que

usa análise dinâmica para detectar violações do conceito de atomicidade de Wang e Stoller. Em [FFY08] os autores apresentam um novo sistema, Velodrome, que incorpora a primeira análise dinâmica que é correcta e completa, identificando correctamente se uma sequência de operações executadas por diversos processos é, ou não, serializável.

2.3.4 Sumário

Nesta Secção foram referidas algumas anomalias referentes ao acesso concorrente a dados partilhados. Para cada tipo de anomalia, foi apresentada a sua definição bem como exemplos que facilitam a sua compreensão, e foram referidas algumas abordagens existentes na sua detecção.

Todos os algoritmos e técnicas descritas neste documento, bem como o protótipo implementado, assumem que os programas recebidos executam em ambiente de atomicidade forte, i.e., que todos os acessos a variáveis partilhadas são feitos dentro de um bloco atómico. Por esta razão, assumiremos que qualquer programa recebido pela nossa ferramenta está isento de *low-level dataraces*.

Por uma questão de legibilidade, ao longo do documento usaremos o termo *datarace* para referir qualquer tipo de anomalia que possa ocorrer em ambientes de atomicidade forte englobando, por exemplo, *high-level dataraces* e *stale-value errors*.

2.4 Análise de Programas

A programação concorrente pode introduzir erros decorrentes do acesso concorrente a dados partilhados, difíceis de diagnosticar. Esta dificuldade resulta do facto de execuções consecutivas do mesmo programa não obterem necessariamente os mesmos resultados, devido à aleatoriedade no escalonamento dos processos, gerando inúmeros cenários de execução possíveis. Como testar todos estes cenários de execução é frequentemente uma tarefa inviável, alguns cenários erróneos podem nunca ser detectados.

A análise de programas, tal como o nome indica, consiste no processo de analisar o comportamento de programas e, por desempenhar um papel importante na detecção de anomalias em programas concorrentes, será abordado nesta secção de trabalho relacionado. Assim, na Secção 2.4.1, começaremos por apresentar as duas grandes abordagens na análise de programas (estática e dinâmica), dando mais ênfase em análise estática. Em seguida, na Secção 2.4.2, serão abordados alguns tipos de análise de programas, como as análises *Data-Flow* e *Control-Flow* e, na Secção 2.4.3, serão referidas algumas representações de programas usadas em análise estática. Na Secção 2.4.4, serão referidas algumas das ferramentas usadas actualmente na análise de programas e, finalmente, na Secção 2.4.5, será feito um pequeno resumo desta secção sendo apresentada a ferramenta que servirá como base do nosso trabalho.

2.4.1 Análise estática e dinâmica

A análise de programas pode ser dividida em dois grandes grupos: análise estática e análise dinâmica. As duas abordagens não são necessariamente exclusivas e têm diferentes vantagens e desvantagens pelo que, dependendo dos objectivos a atingir e do sistema que queremos analisar, uma poderá ser mais adequada que a outra.

A análise estática permite antecipar o comportamento dos programas antes da sua execução. Assim, consegue detectar potenciais erros mesmo antes destes se manifestarem. Por outro lado, não estando dependente dos *inputs* e cenários de execução possíveis, este tipo de análise pode detectar erros que, pela natureza do programa ou dos dados, nunca viriam a ocorrer, i.e., que não correspondem a erros reais. Nestas situações dizemos que foram gerados *falsos positivos*.

A análise estática aplica algoritmos sobre o código fonte dos programas ou directamente sobre o código binário já compilado. Para facilitar a sua implementação e melhorar o seu desempenho, muitos destes algoritmos actuam, na prática, sobre representações mais abstractas dos programas como, por exemplo, grafos ou árvores. Algumas destas representações são referidas na Secção 2.4.3.

A análise dinâmica de programas, pelo contrário, permite avaliar o comportamento dos programas através da monitorização da sua execução. Numa primeira fase, o código do programa é instrumentado e a execução do programa é feita num ambiente controlado/protegido. Nesta fase são adicionadas instruções no código do programa que permitem a produção de eventos (por exemplo acessos a memória) que serão guardados num ficheiro de registo. Depois, numa segunda fase, esse registo é analisado apurando a relação entre os eventos do mesmo, bem como a sua ordem cronológica.

A eficiência deste tipo de análise está consideravelmente dependente da quantidade e qualidade dos dados de entrada, na medida em que as execuções têm de reflectir cenários reais. Ao necessitar de verificar apenas um cenário de execução específico, a análise dinâmica tem uma implementação significativamente mais simples adicionando, no entanto, um *overhead* à execução do programa. Como este tipo de análise segue um cenário de execução concreto, os erros encontrados correspondem a erros reais, não sendo gerados falsos positivos. No entanto, podem ser gerados *falsos negativos*, i.e., alguns erros podem ficar por detectar, possivelmente porque se encontram noutros cenários de execução que não foram analisados.

Existem ainda soluções híbridas que tentam conjugar as vantagens das análises estáticas e dinâmicas. Este tipo de soluções consiste em fazer uma análise estática antes da execução do programa para detectar possíveis conflitos e, depois, executar o programa com uma análise dinâmica controlando o seu comportamento, de forma a verificar se esses potenciais conflitos correspondem a conflitos reais.

2.4.2 Técnicas de análise estática

Dentro do universo amplo de técnicas de implementação de análise estática, iremos utilizar, no nosso trabalho, análises *Control-Flow* e *Data-Flow*, pelo que serão discutidas e aprofundadas nesta secção.

A análise *Control-Flow* [All70], tal como o nome indica, está relacionada com o fluxo da execução dos programas. Este tipo de análise define os blocos básicos de um programa, estudando depois as relações entre eles, i.e., analisando para cada bloco básico os seus possíveis sucessores e predecessores. Assim, a análise *Control-Flow* interessa-se por perceber a ordem pela qual as operações (instruções ou chamadas de funções) são executadas. Este tipo de análise de programas é bastante útil, por exemplo, na área de detecção de código inatingível ou, em programas concorrentes, para verificar se dois processos podem executar concorrentemente (análise *May-Happen-in-Parallel*).

A análise *Data-Flow* [KSK09], por outro lado, é uma técnica que permite obter informação sobre o uso das variáveis (ou posições de memória) ao longo da execução do programa. Este tipo de análise começa por percorrer o código, guardando informação sobre as variáveis cujos valores estão a ser usados ou redefinidos. Por exemplo, na afectação $z = x + y$, o valor antigo de z é redefinido, enquanto os valores das variáveis x e y são utilizados.

A análise *Data-Flow* é uma ferramenta poderosa que nos permite fazer vários tipos de optimizações. Algumas destas são referidas de seguida:

- *Available Expression Analysis* – Permite identificar se o valor de uma determinada expressão matemática já foi computado anteriormente possibilitando a optimização de código que executa diversas vezes a mesma expressão.

- *Reaching Definitions Analysis* – Identifica, para um determinado ponto do programa, que afectações foram feitas tais que os valores das variáveis afectadas ainda não foram redefinidos permitindo detectar variáveis que nunca serão usadas no programa.

- *Live Variable Analysis* – Determina se uma variável está viva ou morta, i.e., se o seu valor ainda pode ou não vir a ser usado no programa. Uma das aplicações deste tipo de análise consiste em libertar as posições de memória que guardam os valores das variáveis mortas.

- *Pointer Analysis* – Determina que posição de memória é referenciada por um determinado apontador. Este tipo de análise permite verificar se dois apontadores referenciam a mesma posição em memória.

De forma a facilitar este tipo de análises, é muitas vezes utilizada uma linguagem intermédia SSA (*Static Single Assignment form*) onde, para cada afectação de variável, é criada uma versão distinta da mesma.

Estas técnicas permitem a obtenção de informação diversa sobre um determinado programa, que pode ser usada na optimização de código, detecção de erros de programação e aproximação do tempo de execução de um programa no pior caso, entre outras.

2.4.3 Representação de Programas

A maioria das ferramentas de análise de programas começam por transformar o código fonte ou o código binário do programa numa representação abstracta mais conveniente, trabalhando posteriormente sobre essa mesma representação. Este procedimento facilita a implementação da análise podendo melhorar a sua eficiência, já que a representação do programa é criada tendo em conta a informação relevante e necessária para atingir os objectivos específicos da análise. Nesta secção serão abordadas algumas destas representações.

Control Flow Graph

CFG é uma representação baseada na noção de grafo, e representa todos os cenários de execução possíveis de um programa. Neste grafo, os nós representam blocos básicos, i.e., blocos de código formados por sequências de instruções sem saltos condicionais. Por outras palavras, se uma instrução de um bloco básico é executada, então todas as instruções desse bloco também são executadas. Por outro lado, todos os arcos deste grafo representam as transições de um bloco para outro ou seja, representam os saltos condicionais e chamadas de métodos e de funções do programa.

Abstract Syntax Tree

Uma AST consiste numa árvore que representa a estrutura abstracta do código fonte de um programa. Ao contrário do CFG, este tipo de árvores não utiliza blocos básicos de código. Os nós de uma AST correspondem aos diferentes tipos de operações existindo, por exemplo, nós que representam ciclos, testes condicionais, afectações, entre outros. Este tipo de representação é adequada para fazer pequenas alterações em programas, já que representa directamente as operações do código dos mesmos. No entanto, para poderem representar integralmente as operações dos programas, as AST apresentam diversos tipos de nós semelhantes, como por exemplo os nós *for*, *while*, *dowhile*. Esta redundância faz com que esta seja uma representação mais complexa que os CFG, tornando a análise de programas computacionalmente mais pesada. Apesar disto, as AST são usadas muitas vezes como ponto de partida para a criação doutro tipo de grafos mais adequados a análises mais específicas.

Call Graph

Um *Call Graph* consiste num grafo orientado que representada todas as relações entre subrotinas ou blocos de código de um determinado programa. Os nós de um *Call Graph*

representam as subrotinas ou funções de um programa, sendo que um arco (s_1,s_2) dita que s_1 chama a subrotina s_2. Ciclos neste tipo de grafo representam funções recursivas.

Note-se que, apesar desta representação ser menos completa que as anteriores, não tendo qualquer tipo de informação sobre o código de cada subrotina, a análise do *Call Graph* de um programa pode ser usada para a compreensão do programa por parte do utilizador, ou como base para outras análises mais complexas.

Em linguagens orientadas a objectos, como é o nosso caso, os nós de um *Call Graph* representam métodos e o grafo começa no método main da classe principal do programa [GDDC97].

Apesar de estarmos a apresentar esta representação no âmbito de análise estática, é também possível criar um *Call Graph* dinâmico aquando da execução do programa.

Sumário

As representações referidas permitem uma abstracção do programa que vem facilitar a análise do mesmo. Os algoritmos e análises descritas neste documento e implementadas na ferramenta MoTH, utilizarão algumas destas representações.

2.4.4 Ferramentas

De forma a implementar um protótipo que validasse os algoritmos e técnicas apresentadas neste documento, criámos uma extensão de uma ferramenta de análise estática de programas já existente.

Actualmente existe uma grande variedade de ferramentas de análises de programas. A maioria destas ferramentas partilha um conjunto de técnicas comuns para detecção de erros usuais, tais como referências inválidas a memória. No entanto, é possível distinguilas segundo algumas propriedades tais como a sua complexidade, disponibilidade do código fonte, completude e flexibilidade. Assim, cabe ao utilizador criar uma especificação ou lista de requisitos da sua ferramenta ideal, escolhendo posteriormente uma determinada ferramenta em função dessa especificação.

Propriedades requeridas

Foram definidas algumas propriedades importantes que nos ajudam a escolher uma ferramenta adequada ao nosso projecto.

Actividade Uma das propriedades importantes na escolha de uma ferramenta é o índice de actividade da mesma. Uma comunidade activa permite a um utilizador de uma ferramenta desenvolver um projecto sólido, apoiado por especialistas na área, com grande conhecimento das capacidades, limitações e implementação dessa mesma ferramenta.

Maturidade Uma ferramenta é madura se já foi testada e utilizada em diversos projectos de natureza académica e/ou industrial. Isto funciona como validação da ferramenta e demonstra a robustez e eficácia da mesma.

Disponibilidade de Código Outra propriedade importante na escolha de uma ferramenta está relacionada com o tipo de licenciamento e a disponibilidade do código da mesma. Como foi dito anteriormente, quase todas as ferramentas de análise de programas partilham um conjunto de técnicas comuns associadas a erros usuais em ambientes de programação. No entanto, se por exemplo um utilizador quiser detectar um tipo de erro específico da sua aplicação, terá de adaptar a ferramenta para criar esta nova funcionalidade. Tendo isto em conta, a indisponibilidade do código fonte poderá limitar, ou mesmo inviabilizar, a adopção da ferramenta às necessidades do utilizador.

Linguagem Alvo Ao trabalharmos directamente sobre o código binário podemos analisar programas previamente compilados. Isto permite-nos, por exemplo, analisar programas que importam bibliotecas externas cujo código fonte não se encontra acessível. Para além disto, o Java ByteCode tem a mesma expressividade que as linguagens alto-nível que o geram como, por exemplo, o Java. Assim, no âmbito do nosso projecto, apenas consideraremos a análise de programas Java ByteCode pelo que só serão tidas em conta as ferramentas que trabalham sobre esta linguagem.

Tipos de análises implementadas Finalmente, interessa-nos saber, para cada ferramenta, o tipo de técnicas de análise disponibilizadas ao utilizador. Quanto mais completa for a ferramenta base utilizada, menos implementações adicionais terão que ser implementadas na aplicação. Cada ferramenta será então avaliada na disponibilização de técnicas tais como análises *Control-Flow*, *Data-Flow*, *AST Walker* e a criação de dependências entre diferentes análises.

Exemplos de ferramentas

Vistas as propriedades requeridas na escolha de uma ferramenta, falta então fazer uma comparação das ferramentas actuais que melhor satisfazem essas mesmas propriedades.

Ciera Christopher et al. [Chr06] faz uma comparação de algumas ferramentas de análise de programas, tendo em conta o desenho de uma ferramenta abstracta ideal. Outras comparações deste tipo de ferramentas são feitas baseando-se em testes com programas conhecidos da literatura usados neste tipo de validações [RAF, WJKT05]. Todas estas comparações serviram para nos fornecer informação sobre as diversas ferramentas existentes, bem como para nos ajudar a criar um modelo de uma ferramenta adequada ao nosso projecto.

Foi feita uma pesquisa das ferramentas de análise de programas que trabalham sobre Java ByteCode. Nessa pesquisa, para cada ferramenta encontrada, foram estudadas as

propriedades referidas anteriormente permitindo-nos fazer uma comparação genérica das ferramentas observadas. Estas ferramentas são referidas em seguida:

ASM [ASM] É uma ferramenta de código aberto que faz manipulação e análise de programas Java ByteCode permitindo a geração dinâmica de novas classes bem como a modificação de classes existentes. Apesar de oferecer as mesmas funcionalidades que outras ferramentas, tais como análises *Control-Flow*, *Data-Flow* e *AST Walker*, o ASM foca-se na performance e simplicidade de implementação, tornando-se atractivo para sistemas de análise dinâmica de programas. Esta ferramenta já atingiu um nível maduro, tendo sido utilizada por diversas empresas como a Oracle, e tem uma comunidade activa que disponibiliza ajuda sob a forma de tutoriais e listas de distribuição.

BCEL [BCE] Tal como o ASM, o *Byte Code Engineering Library* (BCEL) também permite a análise, criação e manipulação de ficheiros binários da JVM. É uma ferramenta madura usada em vários projectos tais como compiladores, optimizadores, geradores de código ou mesmo ferramentas de análise estática e dinâmica de programas. Apesar de não ter tido muito desenvolvimento nos últimos anos, e da respectiva comunidade parecer inactiva, o BCEL serve como base de algumas ferramentas de análise de programas. Finalmente, o BCEL não implementa análises *Control-Flow* e *Data-Flow*.

FindBugs [Fin] Esta ferramenta de código aberto foi especialmente desenhada para fazer a detecção de erros em Java ByteCode usando análise estática. Utiliza um sistema de padrões de erros que são detectados na representação abstracta de um programa. O FindBugs utiliza o BCEL para fazer análises sobre a AST, trabalha de uma forma independente do código fonte e implementa diversos tipos de análises, tais como *AST Walker*, *Control-Flow* e *Data-Flow*, permitindo também a criação de dependências entre análises. É uma ferramenta bastante madura que, para além do número imponente de utilizadores em todo o mundo, tem diversos patrocinadores como, por exemplo, a Google e a Sun (Oracle). Tem ainda uma comunidade activa que fornece diversos tutoriais, manuais e exemplos.

Javassist [Jav] Consiste numa biblioteca de classes que editam Java ByteCode. O Javassist oferece dois níveis de interface: *source* e *bytecode*. O primeiro, consiste numa interface que opera em cima do ByteCode criando uma linguagem intermédia entre o código fonte e o código binário, onde o utilizador pode introduzir fragmentos de código Java. O segundo nível permite a um utilizador editar e modificar os ficheiros binários do programa. Apesar de ser referido nesta secção por uma questão de completude, o Javassist é uma ferramenta pouco madura cuja comunidade parece estar relativamente inactiva.

Soot [Soo] O Soot é uma das ferramentas actuais de análise de programas mais completas e maduras. Tal como no Javassist, esta ferramenta também disponibiliza uma linguagem de programação, intitulada *Jimple*, cujo nível de abstracção se encontra entre o código fonte e o Java ByteCode. Assim, o Soot transforma os ficheiros Java ByteCode em ficheiros *Jimple* e executa as análises sobre esta representação intermédia. Apesar de não permitir a criação de dependências entre análises, o Soot implementa análises para *Control-Flow*, *Data-Flow* e *AST Walker*. É uma ferramenta de código aberto que disponibiliza um grande conjunto de informação em tutoriais, exemplos, manuais e listas de distribuição activas.

Tabela 2.1: Comparação de Ferramentas de Análise de Programas

	ASM	**BCEL**	**FindBugs**	**Javassist**	**Soot**
Actividade	Comunidade Activa, tutoriais e API	Comunidade Inactiva, pequeno manual com exemplos	Mailing List activa, manuais, tutoriais, exemplos	Comunidade inactiva, API, tutoriais	Mailing List activa, manuais, tutoriais, exemplos
Maturidade	Maduro	Maduro	Maduro	Pouco maduro	Maduro
Disponibilidade do Código	Open Source	Open Source	Open Source	Open Source	Open Source
Linguagem Alvo	JBC	JBC	JBC	JBC	JBC
Tipos de análises	ASTW, CF, DF	ASTW	ASTW, CF, DEP, DF	DF	ASTW, CF, DF

ASTW – *AST Walker*, CF – *Control-Flow*, DF – *Data-Flow*, DEP – Dependência de análises, JBC - Java ByteCode

2.4.5 Sumário

Abordámos o tema de Análise de Programas referindo algumas das técnicas utilizadas bem como algumas representações intermédias de programas que vêm facilitar o desenho e a implementação deste tipo de análises. Foi apresentado um estudo das principais ferramentas de análise estática de programas que trabalham sobre Java ByteCode, bem como das propriedades destas que são relevantes no contexto do nosso trabalho. Este estudo é esquematizado na Tabela 2.1.

Na ponderação das propriedades e características das ferramentas, o FindBugs e o Soot descaram-se pela sua maturidade e adequação. No entanto, o FindBugs é actualmente utilizado para implementação de pequenos algoritmos que permitem detectar erros extremamente simples. A existência de uma comunidade a desenvolver análises grandes e complexas na ferramenta Soot, incluindo membros do grupo local de trabalho, permite-nos um maior apoio no desenvolvimento de novas análises. Por esta razão, escolhemos o Soot como ferramenta base para desenvolver a nossa extensão de detecção de *dataraces*.

Desta forma, todas as análises desenvolvidas ao longo deste trabalho recebem a informação disponibilizada pela ferramenta Soot como, por exemplo, o *Call Graph* e o *Control Flow Graph* do programa.

3

$\mathcal{M}$oT$_H$ - Detecção de Dataraces

3.1 Introdução

Ao longo deste capítulo apresentaremos $\mathcal{M}$oT$_H$, uma ferramenta que utiliza análise estática na detecção de *dataraces* em programas Java ByteCode com semântica transaccional. Em [PDL+11] é possível encontrar a descrição e avaliação de uma versão anterior deste trabalho. Tal como foi referido anteriormente, por uma questão de simplicidade, ao longo deste documento usaremos o termo *datarace* para referir qualquer tipo de anomalia que possa ocorrer em ambientes de atomicidade forte como, por exemplo, *high-level dataraces* e *stale-value errors*.

A detecção de *dataraces*, tal como é ilustrado na Figura 3.1, é constituída por duas grandes fases. A primeira, permite-nos reunir toda a informação necessária para detectar este tipo de anomalias. Para isso, são identificados todos os fios de execução do programa de entrada bem como as suas *views*, i.e., os conjuntos de acessos feitos dentro de cada transacção executada por este. Para além desta, é ainda recolhida informação relacionada com as dependências entre variáveis do programa. Numa segunda fase, toda a informação recolhida é disponibilizada como *input* para um conjunto de Sensores. Um Sensor consiste num *plugin* que implementa um algoritmo específico que detecta um determinado tipo de *dataraces*.

As Secções 3.2 e 3.3 descrevem mais pormenorizadamente cada uma destas fases.

3.2 Computação da Informação Base

Como foi referido na secção anterior, a detecção de *dataraces* por parte dos Sensores é feita com base num conjunto amplo de informação computada previamente. A recolha

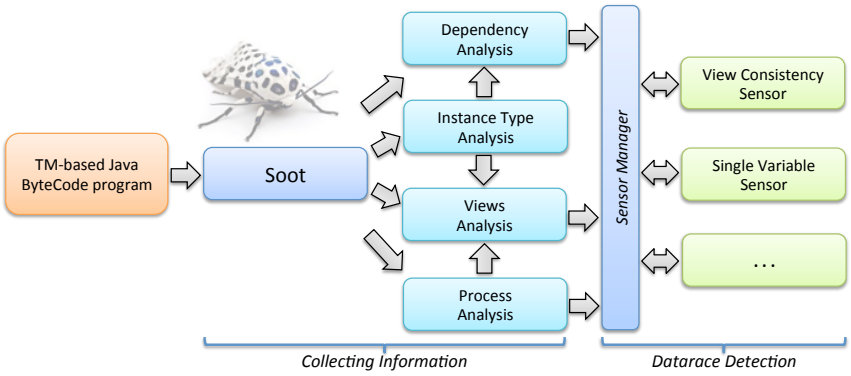


Figura 3.1: Procedimento da Detecção de Dataraces

desta informação é feita através de inúmeras análises que foram implementadas como extensões da ferramenta Soot. Assim, a ferramenta Soot analisa o Java ByteCode do programa gerando a respectiva linguagem Jimple, que consiste numa linguagem cujo nível de abstracção se encontra entre o código binário e o Java. Todos os componentes da ferramenta MoTh trabalham directamente sobre esta linguagem.

A maioria das análises implementadas têm como objectivo a computação da informação necessária para os Sensores. No entanto, foram implementadas análises complementares com o intuito de tornar essa mesma informação mais precisa e detalhada, aumentando consequentemente a eficiência da ferramenta MoTh.

Sendo que a maioria destes algoritmos analisam o *Control-Flow* do programa de entrada, definimos uma linguagem imperativa simples cuja sintaxe é ilustrada na Figura 3.2 e que nos permitirá explicar intuitivamente o funcionamento das análises descritas ao longo desta secção. Nesta linguagem, as variáveis têm como valor números inteiros ou referências a memória. Os valores booleanos são codificados utilizando os números 0 e 1, denotando *false* e *true* respectivamente. Um programa corresponde a um conjunto de procedimentos e funções.

Esta linguagem imperativa simples captura a essência da linguagem Java, representando uma abstracção da mesma. Assim, esta será utilizada na elaboração do nosso modelo teórico, permitindo a definição da execução simbólica de cada uma das análises implementadas, apesar do protótipo desenvolvido trabalhar directamente sobre o código Java compilado.

Para formalizar a nossa abordagem, definiremos ainda alguns conjuntos que serão utilizados ao longo das próximas secções. Estes conjuntos estão representados e descritos na Tabela 3.1. Tal como em [AHB03], generalizaremos o conceito de variáveis partilhadas para atributos de instâncias de objectos.

Seja Classes o conjunto de classes usadas num determinado programa Java e Fields o conjunto de todos os atributos de todas as classes de Classes. Seja também Methods o conjunto de todos os métodos invocados no programa e Atomics $\subseteq$ Methods o conjunto de todos os métodos atómicos, que correspondem às transacções do programa.

e	::=		(*expression*)
		x	(variables)
	|	n	(constant)
	|	$e \oplus e$	(binary op)
	|	null	(null value)
A	::=		(*assignments*)
		$x := e$	(local)
	|	$x := y.f$	(heap read)
	|	$x.f := e$	(heap write)
	|	$x := func(\vec{y})$	(function call)
	|	$x :=$ new $C()$	(allocation)
S	::=		(*statements*)
		$S\ ;\ S$	(sequence)
	|	A	(assignment)
	|	$proc(\vec{x})$	(procedure call)
	|	if e then S else S	(conditional)
	|	while e do S	(loop)
	|	return e	(return)
	|	skip	(skip)

Figura 3.2: Sintaxe da linguagem imperativa

Assim, podemos definir os conjuntos que representam as variáveis do programa. Seja LocalVars o conjunto de todas as variáveis locais do programa. Uma variável local $l \in$ LocalVars corresponde a um par (n,m) onde n é o nome da variável local e $m \in$ Methods é o método onde esta foi declarada. Seja também GloVars $\subseteq$ Classes $\times$ Fields o conjunto de variáveis globais de um programa representadas pela sua classe e pelo nome do atributo respectivo. Finalmente, o conjunto Vars $=$ LocalVars $\cup$ GloVars corresponde ao conjunto de todas as variáveis do programa, sejam estas locais ou globais.

A análise de programas por parte da ferramenta MoTh impõe algumas restrições. Em primeiro lugar, consideraremos que cada transacção é encapsulada num método com a anotação @Atomic, sendo que as variáveis que não são globais à classe são passadas como parâmetro. Apenas consideraremos programas que utilizem o mecanismo de *flat nesting*, i.e., se uma transacção é chamada dentro de o escopo de outra então é tratada como se fosse uma chamada de um método não atómico, e assumiremos que não há chamadas a bibliotecas IO dentro de transacções. Finalmente, como ainda não existe um consenso na semântica associada ao tratamento de excepções em MT, i.e., se uma transacção deve ou não abortar quando é lançada uma excepção, não serão contempladas excepções dentro do escopo transaccional.

Para cada análise, começaremos por justificar a sua necessidade e utilidade numa breve introdução, em seguida faremos uma descrição intuitiva do seu funcionamento e, finalmente, apresentaremos a execução simbólica da mesma.

Nome	Descrição
Classes	Conjunto de todas as classes utilizadas no programa de entrada
Fields	Conjunto de todos os atributos das classes pertencentes ao conjunto Classes
Methods	Conjunto de todos os métodos invocados no programa de entrada
Atomics	Conjunto de todos os métodos atómicos invocados no programa de entrada (Atomics $\subseteq$ Methods)
LocalVars	Conjunto de todas as variáveis locais do programa de entrada, representadas pelo método onde foram declaradas e o seu nome
GloVars	Conjunto de todas as variáveis globais do programa de entrada, representadas pela sua classe e pelo nome do atributo respectivo
Vars	Conjunto de todas as variáveis locais ou globais do programa de entrada (Vars $\equiv$ LocalVars + GloVars)
Accesses	Conjunto de todos os acessos de leitura e escrita feitos a variáveis globais dentro do escopo transaccional
Views	Conjunto de *views*, i.e., de conjuntos de acessos a variáveis globais dentro de um determinado método atómico
VarClass	Conjunto de todas as possíveis classes de implementação de cada variável do programa
Versions	Conjunto de todas as versões de variáveis do programa de entrada
AVersions	Conjunto de todas as versões de variáveis globais acedidas dentro de um determinado bloco atómico
Deps	Conjunto de dependências entre os blocos atómicos do programa de entrada

Tabela 3.1: Conjuntos utilizados ao longo deste documento

3.2.1 Análise de Processos

Introdução

Os *dataraces* abordados neste documento são gerados através de interacções anómalas entre transacções de diferentes fios de execução. Assim, para detectar este tipo de conflitos, é necessário apurar inicialmente que fios de execução podem ser gerados num determinado programa, e que métodos transaccionais são executados por cada um destes.

A Análise de Processos consiste num algoritmo executado sobre o *Call Graph* do programa que computa toda esta informação que será posteriormente usada pelas outras análises e pelos Sensores da ferramenta MoTh. Ao longo deste documento será usado o termo *processo* para referir qualquer tipo de fio de execução gerado num programa de entrada.

Descrição

Um processo Java pode ser criado através de uma classe que estende a classe `Thread` ou que implementa a interface `Runnable`. Na ferramenta MoTh existem dois tipos de processos: o processo gerado pelo método main da classe principal do programa, e os processos criados dentro do escopo desse método através de uma construção de criação de *threads*. Denotaremos o primeiro por P_{main}, e por P_{C} o processo gerado pelo método run da classe C, onde `C implements Runnable` ou `C extends Thread`.

Na ferramenta MoTh, como só é criado um processo para cada classe que permite a geração de *threads*, são sempre verificados conflitos entre duas instâncias do mesmo processo. Por outro lado, devido à falta de uma análise *May-Happens-In-Parallel*, assumiremos que qualquer troca de contexto entre dois processos é possível, i.e., que uma determinada transacção de um processo pode ocorrer entre quaisquer duas transacções de outro processo.

Listagem 3.1: Exemplo de um programa de entrada (parte 1)

```
//Classe Main
public static void main(String[] args){
  MyThread1 t1 = new MyThread1();
  MyThread2 t2 = new MyThread2();
  startTimer();
  t1.start();
  t2.start();
  stopTimer();
}
```

Listagem 3.2: Exemplo de um programa de entrada (parte 2)

```
//Classe MyThread1
public void run(){
  m1();
  m2();
}
//Classe MyThread2
public void run(){
  m3();
  m4();
}
```

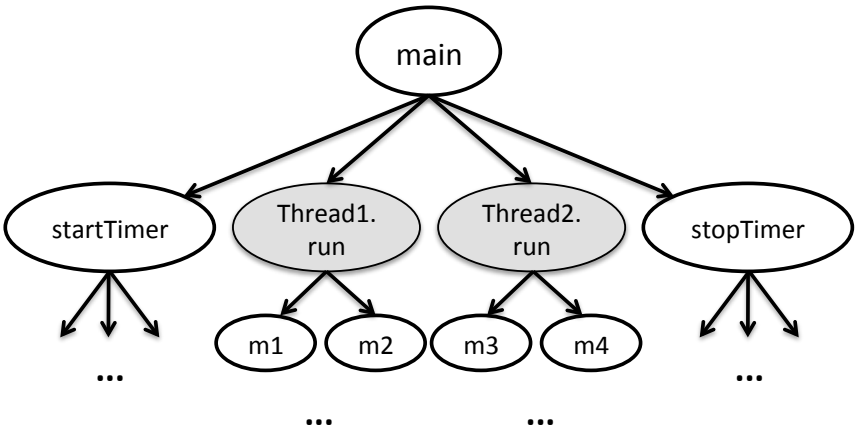


Figura 3.3: *Call Graph* do código das Listagens 3.1 e 3.2

O algoritmo da Análise dos Processos percorre o *Call Graph* do programa de entrada guardando sempre informação sobre o processo corrente na análise de cada nó do grafo. Inicialmente, o processo corrente consiste no nó do método main da classe principal e, cada vez que é atingido um nó que representa o método run das classes `Thread` ou `Runnable`, é criado um novo processo que passa a ser o corrente. Finalmente, sempre que é encontrado um método atómico, é guardada informação que associa o processo corrente ao método executado pelo mesmo.

As Listagens 3.1 e 3.2 ilustram um exemplo de um programa onde são criados e executados dois processos Java, através das classes MyThread1 e MyThread2 que estendem a classe `Thread`. O *Call Graph* gerado neste exemplo é ilustrado na Figura 3.3. Os nós do grafo gerado correspondem aos métodos chamados no código do programa, começando no método main da classe principal do mesmo. Na figura, os nós mais escuros correspondem a métodos de criação de processos (run), sendo que os claros representam os restantes métodos do programa de entrada.

Algoritmo 1: Função analiseNode

```
Input: method, process, visited[]
Result: void
foreach Edge e : outOf(method) do
    if !visited.contains(process, target(e)) then
        add(visited, (process,target(e)));
        if isThreadCreationEdge(e) then
            process = createProcess(target(e));
            analyseNode(target(e), process,visited);
        else
            if isSynchronizedBlock(target(e)) then
                associate(process, target(e));
            else
                analyseNode(target(e), process, visited);
            end if
        end if
    end if
end foreach
```

Algoritmo

O pseudo-código da Análise de Processos, implementada sobre o *Call Graph*, é ilustrada no Algoritmo 1. Este algoritmo percorre todos os métodos chamados no programa, identificando a criação de novos processos. Para além disso, o algoritmo identifica, para cada processo, os métodos atómicos chamados pelo mesmo.

Foi implementada uma função recursiva analiseNode que analisa um determinado nó (método) do *Call Graph* tendo em conta o processo corrente, ambos passados como parâmetro. O último parâmetro consiste num vector que guarda todos os pares (m,p), onde m consiste num método que corresponde a um nó do *Call Graph* que já foi visitado pelo processo p.

A função ilustrada é inicialmente chamada com o método main da classe principal, o processo corrente P_{main} e um vector vazio como parâmetros. Depois, sempre que analisamos um nó do grafo, percorremos todos os sucessores do mesmo no *Call Graph* do programa verificando se já foram visitados por este processo. Para cada sucessor, se este corresponder à criação de um processo, então o processo corrente é alterado. Se, por outro lado, o método sucessor for um método atómico, então associamos o processo corrente ao método invocado. Finalmente, a função é chamada recursivamente para todos os sucessores.

Observações

O resultado desta análise é usado em praticamente todas as demais análises descritas nesta secção. Para além disso, cada Sensor da ferramenta MoTh recebe um conjunto $\mathcal{P}s$ que corresponde ao conjunto dos processos gerados por esta análise. Finalmente, é ainda disponibilizado o conjunto de métodos atómicos (transacções) executados por cada um desses processos.

3.2.2 Análise de *Views*

Introdução

A definição de *view*, descrita por Artho et al. em [AHB03], expressa o conjunto de variáveis que foram acedidas dentro de um determinado bloco sincronizado de código. Com o intuito de aumentar a precisão desta abordagem, este conceito foi estendido de forma a distinguir acessos de leitura e escrita. Desta forma, por exemplo, não teremos de considerar conflitos entre acessos de leitura. Uma *view* contém, portanto, o conjunto dos acessos a memória feitos dentro de um determinado bloco sincronizado, em vez de conter apenas o conjunto de variáveis acedidas no mesmo.

A Análise de *Views* permite-nos computar toda a informação relacionada com acessos a variáveis partilhadas dentro do escopo de transacções, que será utilizada posteriormente para detectar, por exemplo, acessos parciais a conjuntos de variáveis que deveriam ser acedidos atomicamente, i.e., como um todo.

Descrição

O conceito de *view* proposto em [AHB03] expressa todas as variáveis acedidas dentro de um determinado bloco sincronizado, para programas desenvolvidos num paradigma orientado a objectos. Tal como os autores, generalizaremos o conceito de variável partilhada para atributos de instâncias de objectos. Note-se que, no contexto específico de programas MT, um bloco sincronizado corresponde a um bloco transaccional.

Seja Accesses o conjunto de todos os acessos de leitura e escrita feitos a variáveis de GloVars dentro do escopo transaccional. Um acesso $a \in$ Accesses consiste num par (α,v), com $\alpha \in \{r, w\}$ e $v \in$ GloVars, onde α representa o tipo de acesso (r-leitura ou w-escrita) e v representa a variável acedida.

Uma *view* $v \subseteq$ Accesses de um método atómico consiste num subconjunto de Accesses, e inclui todos os acessos a variáveis globais desse método. O conjunto de todas as *views* do programa é representado por Views.

Existe uma correspondência de um para um entre uma *view* e o respectivo método atómico. Assim definimos a função bijectiva que retorna o método atómico de uma determinada *view*:

$$\Gamma : \text{Views} \longrightarrow \text{Atomics}$$

A função inversa desta, representada por Γ^{-1}, retorna a *view* dado o respectivo método atómico.

O conjunto de *generated views* $V(p)$ de um processo p representa o conjunto de todas as *views* dos métodos atómicos executados por p:

$$v \in V(p) \Leftrightarrow a = \Gamma(v) \wedge \text{executes}(p, a)$$

Note-se que o predicado executes(p, a) é definido através da informação computada com a Análise de Processos descrita na Secção 3.2.1, sendo verdadeiro se o processo p executa o método atómico a e falso caso contrário.

Finalmente, como queremos distinguir acessos de leitura e escrita, para cada *view* geramos a sua *view* de leitura (V_r) e a sua *view* de escrita (V_w). Assim, sendo $\alpha \in \{r, w\}$, obtemos:

$$V_\alpha(p) \triangleq \{(\alpha, v) | (\alpha, v) \in V(p)\}$$

O conjunto das *views* de leitura e escrita de cada processo será disponibilizado como *input* para cada sensor, de forma a detectar diversos tipos de conflitos.

Execução Simbólica

Na Figura 3.4 são apresentadas as regras de execução simbólica para cada tipo de declaração da sintaxe da linguagem definida na Figura 3.2, que geram as *views* de cada processo. Por questões de simplicidade das regras, optámos por omitir a substituição e tratamento dos parâmetros dos métodos invocados.

A execução simbólica de cada procedimento começa com uma *view* vazia e, sempre que é feito um acesso de leitura ou escrita numa variável global, adicionamo-lo a essa mesma *view*. Por outro lado, como estamos interessados apenas em variáveis globais, as regras ASSIGN, ALLOCATION e RETURN não adicionam qualquer tipo de acesso à *view* dos métodos atómicos, tendo o mesmo efeito que a regra SKIP.

Sendo as variáveis globais representadas pela sua classe e pelo nome do atributo respectivo, assumiremos uma função typeof que, dado um objecto, retorna a sua classe.

Na regra PROC CALL, dado o identificador do procedimento $proc$, a função spec retorna a *view* computada, $\mathcal{V}_p$. Posteriormente, a *view* resultante $\mathcal{V}_p$ e a *view* corrente $\mathcal{V}$ são unidas. É importante realçar que as duas *views* referidas são computadas independentemente, tornando a nossa execução simbólica composicional.

O funcionamento da regra FUNC CALL é semelhante à PROC CALL. As restantes regras de execução simbólica são auto-explicativas.

No final da execução simbólica obtemos a informação base relacionada com acessos a variáveis globais, que será usada pelos sensores para detectar os diferentes tipos de *dataraces*.

Mecanismo de Anotações

Nesta análise de *Views* é possível encontrar métodos cujo código não se encontra disponível. Se um método for nativo, i.e., estiver implementado numa linguagem diferente da linguagem Java, então o seu código será ilegível para a nossa ferramenta. Para minorar este problema, foi desenvolvido um mecanismo de anotações que nos permite anotar métodos nativos, obtendo assim informação sobre os possíveis acessos feitos dentro do

$$\boxed{\langle \mathcal{V}, S \rangle \Longrightarrow \langle \mathcal{V}' \rangle}$$

$$\frac{\langle \mathcal{V}, S_1 \rangle \Longrightarrow \langle \mathcal{V}' \rangle \qquad \langle \mathcal{V}', S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}{\langle \mathcal{V}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle} (\textsc{Seq})$$

$$\frac{}{\langle \mathcal{V}, x := y \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Assign})$$

$$\frac{c = \mathsf{typeof}(y) \qquad \mathcal{V}' = \mathcal{V} \cup \{(r, (c, f))\}}{\langle \mathcal{V}, x := y.f \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Heap Read})$$

$$\frac{c = \mathsf{typeof}(x) \qquad \mathcal{V}' = \mathcal{V} \cup \{(w, (c, f))\}}{\langle \mathcal{V}, x.f := y \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Heap Write})$$

$$\frac{}{\langle \mathcal{V}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Allocation})$$

$$\frac{\mathsf{spec}(\mathit{func}) = \mathcal{V}_f \qquad \mathcal{V}' = \mathcal{V}_f \cup \mathcal{V}}{\langle \mathcal{V}, x := \mathit{func}(\vec{y}) \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Func Call})$$

$$\frac{\mathsf{spec}(\mathit{proc}) = \mathcal{V}_p \qquad \mathcal{V}' = \mathcal{V}_p \cup \mathcal{V}}{\langle \mathcal{V}, \mathit{proc}(\vec{x}) \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Proc Call})$$

$$\frac{\langle \mathcal{V}, S_1 \rangle \Longrightarrow \langle \mathcal{V}' \rangle \qquad \langle \mathcal{V}, S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}{\langle \mathcal{V}, \mathsf{if}\, e\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{V}' \cup \mathcal{V}'' \rangle} (\textsc{Conditional})$$

$$\frac{\langle \mathcal{V}, S \rangle \Longrightarrow \langle \mathcal{V}' \rangle}{\langle \mathcal{V}, \mathsf{while}\, e\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{V} \cup \mathcal{V}' \rangle} (\textsc{Loop})$$

$$\frac{}{\langle \mathcal{V}, \mathsf{return}\, e \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Return})$$

$$\frac{}{\langle \mathcal{V}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Skip})$$

Figura 3.4: Regras de Execução Simbólica da Análise de *Views*

escopo dos mesmos.

Não sendo possível analisar este tipo de métodos, assumimos sempre o pior cenário possível. Assim, assumimos que os métodos nativos lêem e alteram todos os objectos referenciados pelos seus parâmetros, bem como o objecto no qual o método foi invocado. Para tal, foi criado um mecanismo de anotações baseado num ficheiro XML que, para cada método de cada classe, contém toda a informação relacionada com os acessos feitos dentro do escopo do mesmo. Finalmente, sempre que, por alguma razão, não tivermos acesso ao corpo de um método, assumiremos que estamos perante um método nativo.

Para minimizar o prejuízo resultante da assunção do pior cenário possível, o mecanismo de anotações de métodos permite ao utilizador corrigir de uma forma simples e intuitiva as anotações geradas. Assim, sempre que a ferramenta MoTh assume que um método é nativo, são geradas anotações automáticas desse método através da adição de novos elementos ao ficheiro XML. Quando isto acontece, o utilizador é alertado podendo rever as anotações geradas que serão usadas posteriormente em futuras análises.

Listagem 3.3: Exemplo da declaração de um método nativo

```
1 class MathComputer{
2   /**
3    * This method returns the maximum between x1 and x2
4    * @param x1 first number
5    * @param x2 second number
6    * @return the maximum of both numbers
7    */
8   public native int getMax(int x1, int x2);
9 }
```

Para uma melhor compreensão do funcionamento deste mecanismo, considere-se a Listagem 3.3 onde é ilustrada a declaração de um método nativo com a respectiva descrição. Se este método fosse chamado no código do programa, então a ferramenta MoTh iria assumir o pior cenário, i.e., que este lê e escreve em ambos os parâmetros e no objecto no qual o método foi chamado. Assim, o fragmento XML ilustrado na Listagem 3.4 seria adicionado ao ficheiro das anotações de métodos. Contudo, o utilizador seria alertado sobre a criação desta anotação e, com base na descrição do método, poderia rever as anotações geradas, e corrigi-las. Neste exemplo, segundo a descrição do método, os acessos feitos correspondem à leitura de ambos os parâmetros. Com isto, para respeitar a semântica do método nativo, as anotações geradas poderiam ser corrigidas pelo utilizador para o fragmento XML ilustrado na Listagem 3.5.

Note-se que, quando um método com anotações é chamado, a ferramenta MoTh assume que essas anotações estão correctas, não analisando o método em questão independentemente do seu código estar ou não disponível. Esta característica pode ser usada para fazer *caching* de informação através da anotação de código estável, como é o caso do JDK.

Listagem 3.4: Fragmento XML gerado pela ferramenta MoTh

```
<class id="MathComputer">
  <method id="getMax(int,int)">
    <reads>this</reads>
    <writes>this</writes>
    <reads>0</reads>
    <writes>0</writes>
    <reads>1</reads>
    <writes>1</writes>
  </method>
</class>
```

Listagem 3.5: Fragmento XML corrigido pelo utilizador

```
<class id="MathComputer">
  <method id="getMax(int,int)">
    <reads>0</reads>
    <reads>1</reads>
  </method>
</class>
```

Esta solução permitiu-nos eliminar alguns falsos negativos obtidos nas primeiras versões da ferramenta relacionados com a indisponibilidade do código de métodos nativos do JDK como, por exemplo, o método arraycopy da classe System ou o método socketClose0 da classe PlainSocketImpl.

Observações

A análise das *Views* é uma das análises mais importantes da ferramenta MoTh, constituindo a base da implementação do *plugin* View Consistency Sensor, descrito na Secção 3.3.1. O levantamento de todos os acessos feitos a variáveis globais dentro do escopo transaccional permitir-nos-á não só detectar *high-level dataraces*, i.e., verificar se ocorreram acessos parciais a determinados grupos de variáveis que deviam ser acedidos atomicamente, como também apurar se um determinado processo escreveu numa variável global específica.

Foi adaptado e implementado um mecanismo de anotações de forma a evitar a perda de informação relacionada com a indisponibilidade do código de alguns métodos do programa de entrada.

No entanto, a informação gerada por esta análise é imprecisa, tornando a versão actual da ferramenta MoTh incorrecta segundo os conceitos de correcção e completude, como aplicados por Flanagan et al. [FLL+02] no contexto de detecção de anomalias, podendo ser reportados falsos negativos.

A Listagem 3.6 ilustra este cenário onde a *view* gerada pela análise não corresponde à semântica do método. Segundo a nossa análise, a *view* do método getElement é constituída pela leitura das variáveis $getFirst$, $first$ e $second$. Com isto, estamos a assumir que este método acede atomicamente aos dois elementos do par quando, na verdade, acede apenas a um dos elementos. Idealmente, este método devia gerar duas *views*, $\{(r, getFirst), (r, first)\}$ e $\{(r, getFirst), (r, second)\}$, que tornaria possível detectar o acesso parcial ao par (acedemos a $first$ ou a $second$ mas nunca aos dois simultaneamente). No entanto, esta metodologia provocaria uma explosão de estados possíveis fazendo com que o número de *views* geradas crescesse exponencialmente com o número

Listagem 3.6: Exemplo que ilustra a imprecisão da informação computada pela Análise de *Views*

```
1  class Pair{
2    int first = 0;
3    int second = 0;
4
5    @Atomic
6    public int getElement(boolean getFirst) {
7      if(getFirst)
8        return first;
9      else
10       return second;
11   }
12 }
```

de ramificações condicionais. Com o intuito de tornar a detecção de *dataraces* computacionalmente viável, prescindimos de alguma precisão da ferramenta considerando que cada transacção corresponde apenas a uma *view*. Este compromisso entre a correcção e a performance desta abordagem permitiu-nos obter bons resultados na análise de programas mais complexos e maduros.

3.2.3 Análise dos Tipos de Instância

Introdução

A maioria das análises desenvolvidas na ferramenta MoTh seguem a estrutura do *Control Flow Graph* do programa analisado. Neste processo, sempre que nos deparamos com uma chamada de um determinado método, analisamos o código do mesmo. No entanto, é possível que o código dos métodos não se encontre disponível mesmo que estes não sejam nativos se, por exemplo, a chamada do método for feita num objecto cujo tipo é uma interface. Este problema está relacionado com o conceito de *Dynamic Dispatch*, i.e., o processo de fazer corresponder a chamada de um método a um fragmento de código específico.

Este cenário é ilustrado na Listagem 3.7, onde o objecto list é do tipo List e pode ser implementado por mais do que uma classe, levantando a questão: "Que fragmento de código devemos analisar quando é chamado o método add na linha de código 11?".

Descrição

Numa primeira abordagem a este problema poder-se-ia analisar todas as possíveis implementações de cada interface. No entanto, esta solução ingénua e imprecisa poderia gerar bastantes falsos positivos já que estaríamos a analisar classes que nunca seriam usadas no programa.

Assim, optámos por implementar uma análise estática inter-procedimental que nos permite apurar as possíveis classes de implementação de cada variável do programa. O

Listagem 3.7: Exemplo que ilustra a utilidade da análise dos tipos de instância

```
1  protected static List<Integer> list;
2  public static void main(String[] args){
3    Random r = new Random();
4    int value = r.nextInt();
5
6    if(value % 2 == 0)
7      list = new ArrayList<Integer>();
8    else
9      list = new LinkedList<Integer>();
10
11   list.add(1);
12 }
```

algoritmo percorre o *Control Flow Graph* do programa e, sempre que encontra uma nova instanciação de um objecto, associa o objecto com a classe usada na instanciação.

Finalmente, esta informação pode ser posteriormente aplicada em qualquer análise que segue o *Control Flow Graph* do programa como, por exemplo, a Análise das *Views*. Nestas análises, o resultado da chamada de um método num objecto com várias classes de implementação possíveis consiste agora na união dos resultados das chamada desse mesmo método em cada uma dessas possíveis classes.

Se pretendêssemos computar as *views* do exemplo da Listagem 3.7, o algoritmo detectaria que neste programa as possíveis classes de implementação do objecto list correspondem às classes ArrayList e LinkedList e, assim, quando o método add fosse chamado nesse objecto, obteríamos uma *view* constituída pela união das *views* resultantes da análise do código desse mesmo método em ambas as classes.

Execução Simbólica

Na Figura 3.5 são apresentadas as regras de execução simbólica desta análise para cada tipo de declaração da sintaxe da linguagem, que computam as possíveis classes de implementação de cada variável.

Seja VarClass $\subseteq$ Vars $\times$ Classes o conjunto de todas as possíveis classes de implementação de cada variável. A execução simbólica de cada procedimento começa com um conjunto vazio ($\mathcal{CS} = \emptyset$), onde vão sendo adicionados pares (v,c) $\in$ VarClass, onde $v \in$ Vars é uma variável e $c \in$ Classes é uma possível classe de implementação de v. Por uma questão de legibilidade, representaremos por v toda a variável local (v,m) $\in$ LocalVars declarada no método m que está a ser analisado.

Sempre que nos deparamos com uma afectação $var_1 := var_2$ (sendo var_1 e var_2 variáveis locais ou globais), então var_1 herda as possíveis classes de implementação de var_2. Para isso, foi criada a função impl de forma a obter todas as possíveis classes de implementação de uma determinada variável. Assim, a função impl é definida da seguinte forma:

Definição 1 (Função Impl) *Seja* $\mathcal{CS} \subseteq \mathsf{VarClass}$ *o conjunto actual de possíveis classes de implementação das variáveis do programa e* $var \in \mathsf{Vars}$ *uma dessas variáveis. A função* impl *retorna o conjunto de possíveis classes de implementação de* var *pertencentes a* $\mathcal{CS}$*.*

$$\mathsf{impl} : \mathsf{VarClass} \times \mathsf{Vars} \rightarrow \mathsf{Classes}$$

$$\mathsf{impl}(\mathcal{CS}, var) \triangleq \{c \mid (var, c) \in \mathcal{CS}\}$$

Finalmente, sempre que é retornada uma variável numa função, adicionamos a informação das possíveis classes de implementação do valor retornado. Desta forma, na regra FUNC CALL, quando temos uma afectação de uma variável com o resultado de uma função, a variável afectada herda todas as classes de implementação do retorno dessa mesma função. Representámos por returnVar a variável especial que representa o retorno da função em questão.

Observações

Com esta Análise dos Tipos de Instâncias, conseguimos analisar métodos chamados em objectos com um tipo interface, aumentando a precisão da nossa abordagem. O resultado desta análise foi usado tanto na Análise de *Views* como nas Análises de Dependências de Dados e Controlo.

3.2.4 Análise de Dependências de Dados

Introdução

Um *stale-value error* ocorre quando o valor de uma variável partilhada sai do escopo de uma transacção t_1 e é usado numa transacção t_2 já que, entre estas duas transacções, outro processo poderia ter alterado o valor original dessa variável, resultando no uso de um valor obsoleto por parte do primeiro processo. Como é que podemos apurar se o valor de uma variável saiu do escopo de uma transacção para outra?

A análise *Data-Flow* apresentada em seguida permite-nos percorrer o código de um programa apurando o fluxo de dados do mesmo, i.e., identificando o percurso do valor de cada variável no programa.

Descrição

Um dos desafios na implementação de uma análise de dependências está relacionado com as múltiplas versões de cada variável. A título de exemplo considere-se o fragmento de código ilustrado na Listagem 3.8. A variável x é afectada primeiro com o valor de z e depois com o valor dois. Depois destas afectações, y recebe o valor de x, i.e., dois. Se considerarmos simplesmente as dependências entre variáveis obteríamos o grafo de dependências disponibilizado na Figura 3.6(a). No entanto, por transitividade, obteríamos

$$\boxed{\langle \mathcal{CS}, S \rangle \Longrightarrow \langle \mathcal{CS}' \rangle}$$

$$\frac{\langle \mathcal{CS}, S_1 \rangle \Longrightarrow \langle \mathcal{CS}' \rangle \qquad \langle \mathcal{CS}', S_2 \rangle \Longrightarrow \langle \mathcal{CS}'' \rangle}{\langle \mathcal{CS}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{CS}'' \rangle} (\textsc{Seq})$$

$$\frac{\mathcal{CS}' = \mathcal{CS} \cup \{(x, c) \mid c \in \mathsf{impl}(y)\}}{\langle \mathcal{CS}, x := y \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Assign})$$

$$\frac{c1 = \mathsf{typeof}(y) \qquad \mathcal{CS}' = \mathcal{CS} \cup \{(x, c) \mid c \in \mathsf{impl}(\mathcal{CS}, (c1, f))\}}{\langle \mathcal{CS}, x := y.f \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Heap Read})$$

$$\frac{c1 = \mathsf{typeof}(x) \qquad \mathcal{CS}' = \mathcal{CS} \cup \{((c1, f), c) \mid c \in \mathsf{impl}(\mathcal{CS}, y)\}}{\langle \mathcal{CS}, x.f := y \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Heap Write})$$

$$\frac{\mathcal{CS}' = \mathcal{CS} \cup \{(x, C)\}}{\langle \mathcal{CS}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Allocation})$$

$$\frac{\mathsf{spec}(func) = \mathcal{CS}_f \qquad \mathcal{CS}' = \mathcal{CS}_f \cup \mathcal{CS} \cup \{(x, c) \mid c \in \mathsf{impl}(\mathcal{CS}, \mathsf{returnVar})\}}{\langle \mathcal{CS}, x := func(\vec{y}) \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Func Call})$$

$$\frac{\mathsf{spec}(proc) = \mathcal{CS}_p \qquad \mathcal{CS}' = \mathcal{CS}_p \cup \mathcal{CS}}{\langle \mathcal{CS}, proc(\vec{x}) \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Proc Call})$$

$$\frac{\langle \mathcal{CS}, S_1 \rangle \Longrightarrow \langle \mathcal{CS}' \rangle \qquad \langle \mathcal{CS}, S_2 \rangle \Longrightarrow \langle \mathcal{CS}'' \rangle}{\langle \mathcal{CS}, \mathsf{if}\, e\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{CS}' \cup \mathcal{CS}'' \rangle} (\textsc{Conditional})$$

$$\frac{\langle \mathcal{CS}, S \rangle \Longrightarrow \langle \mathcal{CS}' \rangle}{\langle \mathcal{CS}, \mathsf{while}\, e\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{CS} \cup \mathcal{CS}' \rangle} (\textsc{Loop})$$

$$\frac{\mathcal{CS}' = \mathcal{CS} \cup \{(\mathsf{returnVar}, c) \mid c \in \mathsf{impl}(\mathcal{CS}, e)\}}{\langle \mathcal{CS}, \mathsf{return}\, e \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Return})$$

$$\frac{}{\langle \mathcal{CS}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{CS} \rangle} (\textsc{Skip})$$

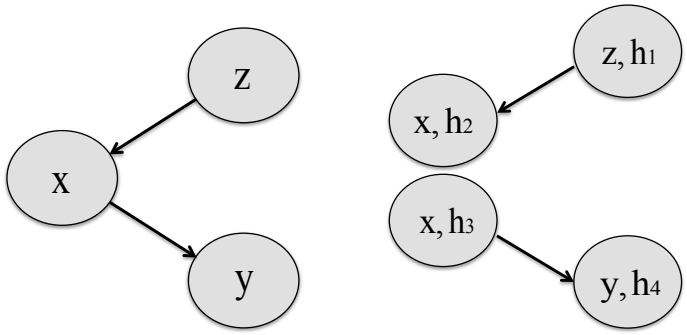
Figura 3.5: Regras de Execução Simbólica da Análise dos Tipos de Instância

uma dependência entre as variáveis z e y que não existe, já que o conteúdo de x (que continha o valor de z) foi redefinido com o valor dois.

Listagem 3.8: Fragmento de código que ilustra a utilidade da representação multi-versões

```
1 int x,y,z = 0;
2 x = z;
3 x = 2;
4 y = x;
```

Assim, para resolver este problema, estendemos o conceito de dependências entre variáveis para dependências entre versões de variáveis. Esta abordagem assemelha-se ao conceito de *Static Single Assignment Form* (SSA), que consiste na propriedade de uma representação intermédia que dita que cada variável dessa representação tem exactamente uma afectação.



(a) Grafo de dependências entre variáveis

(b) Grafo de dependências entre versões de variáveis

Figura 3.6: Grafos de dependências do fragmento de código ilustrado na Listagem 3.8

Na nossa representação multi-versões, é criada uma nova versão de uma determinada variável cada vez que esta é afectada. O conjunto Versions contém todas as versões de variáveis do programa de entrada. Cada elemento deste conjunto consiste num triplo (v,h,m), onde $v \in$ Vars é a variável em questão e h é um identificador único da linha de código onde v foi afectada pela última vez. Por questões de modularidade e escalabilidade desta análise, sempre que uma variável entra no escopo de um método atómico, é criada uma nova versão dessa variável. Não existindo aninhamento de transacções, $m \in \text{Atomics} \cup \{\bot\}$ corresponde ao método atómico onde a versão está a ser utilizada, ou $\bot$ caso esta seja utilizada fora do escopo transaccional.

Se uma determinada variável v ainda não foi afectada no fragmento de código que estamos a analisar, então criamos uma versão (v,$h_?$,m), onde $h_?$ consiste num identificador especial usado para o efeito. Por questões de legibilidade, omitiremos o terceiro elemento do triplo e representaremos as versões (v,$h_?$) por apenas v.

Com isto, no exemplo anterior, são usadas duas versões de x, (x,h_2) e (x,h_3), uma

```
1  public void incX(int val) {
2    int tmp = readX();
3    tmp = tmp + val;
4    setX(tmp);
5  }
6
7  @Atomic
8  private void setX(int p0) {...}
9
10 @Atomic
11 private int readX() {...}
```

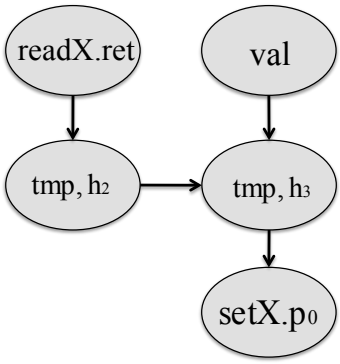


Figura 3.7: Exemplo que ilustra as dependências de dados

versão de y, (y,h_4), e uma de z, (z,h_1). Assim, com este sistema de multi-versões de variáveis, obteríamos o grafo de dependências ilustrado na Figura 3.6(b). Note-se que conseguimos eliminar a dependência entre z e y, obtendo apenas as dependências reais do programa.

Na Figura 3.7 é apresentado um fragmento de código mais complexo, com o respectivo grafo de dependências de dados. O método incX lê o valor da variável x num método atómico guardando-o numa variável temporária, incrementa o valor dessa variável com um valor passado como parâmetro, e escreve-o de novo em x num segundo método atómico.

Neste exemplo, começamos por criar uma dependência entre o retorno da chamada do método readX e a versão (tmp,h_2) da variável tmp. Depois, no incremento desta variável, criamos as dependências incX.val$\rightarrow$(tmp,h_3) e (tmp,h_2)$\rightarrow$(tmp,h_3), já que as variáveis inc e tmp são usadas para afectar novamente a variável tmp. Finalmente, a última versão da variável tmp é usada como parâmetro do método setX, criando a dependência (tmp,h_3) $\rightarrow$ setX.p_0.

Tal como na Análise de *Views*, também aqui podemos encontrar métodos nativos cujo código fonte se encontra indisponível. Assim, nestes casos, assumimos a existência de uma variável especial que depende de todos os parâmetros do método e cujo valor é usado no retorno desse método. Este tipo de assunções podia ser usado para estender o Mecanismo de Anotações, descrito na Secção 3.2.2, de forma a suportar anotações de dependências. Assim, tal como foi feito com os acessos do método, o utilizador teria a possibilidade de corrigir as anotações criadas relacionadas com as dependências de cada método.

Execução Simbólica

Na Figura 3.8 são representadas as regras de execução simbólica desta análise para todas as declarações da linguagem imperativa. A informação passada entre as declarações consiste num grafo de dependências $\mathcal{D}$, implementado como um conjunto de arcos entre versões de variáveis, e num conjunto $\mathcal{H} \subseteq$ Versions que contém todas as versões actuais de cada variável. Num determinado momento, as versões actuais de uma variável são

todas aquelas que foram criadas na última afectação dessa variável num determinado caminho do *Control Flow Graph*. Note-se que, devido à existência de código condicional, em determinados momentos do código podem existir mais do que uma versão actual possível da mesma variável.

Para apurar em cada momento as versões actuais de uma variável, foi definida a função $\mathsf{Ver}_{\mathcal{H}}$ aplicada ao conjunto $\mathcal{H}$, que retorna uma lista com as versões actuais da variável passada por parâmetro (local ou global). Assim esta função pode ser definida da seguinte forma:

Definição 2 (Função $\mathsf{Ver}_{\mathcal{H}}$) *Seja $\mathcal{H} \subseteq \mathsf{Versions}$ o conjunto de todas as versões actuais das variáveis de um programa e $v \in \mathsf{Vars}$ uma dessas variáveis. A função $\mathsf{Ver}_{\mathcal{H}}$ retorna todas as versões actuais da variável v:*

$$\mathsf{Ver}_{\mathcal{H}} : \mathcal{P}(\mathsf{Versions}) \times \mathsf{Vars} \rightarrow \mathcal{P}(\mathsf{Versions})$$

$$\mathsf{Ver}_{\mathcal{H}}(v) \triangleq \begin{cases} \{(v,h)|(v,h) \in \mathcal{H}\} & \text{se } \exists (v,h) \in \mathcal{H} \\ \{(v,h_?)\} & \text{caso contrário} \end{cases}$$

Cada vez que é feita uma afectação de uma variável é criada uma nova versão dessa variável que substitui as existentes. Para isso, definimos uma função $\mathsf{Subs}_{\mathcal{H}}$ que actualiza o conjunto $\mathcal{H}$ substituindo todas as versões actuais de uma determinada variável pela versão passada por parâmetro:

Definição 3 (Função $\mathsf{Subs}_{\mathcal{H}}$) *Seja $\mathcal{H} \subseteq \mathsf{Versions}$ o conjunto de todas as versões actuais das variáveis de um programa e $(v,h) \in \mathsf{Versions}$ a nova versão da variável v. A função Subs substitui todas as versões da variável v por (v,h), retornando o novo conjunto de versões actuais:*

$$\mathsf{Subs}_{\mathcal{H}} : \mathcal{P}(\mathsf{Versions}) \times \mathsf{Versions} \rightarrow \mathcal{P}(\mathsf{Versions})$$

$$\mathsf{Subs}_{\mathcal{H}}((v,h)) \triangleq (\mathcal{H} \setminus \{(v,h')|(v,h') \in \mathcal{H}\}) \cup \{(v,h)\}$$

Finalmente, a função getHash, baseada no número da linha de código de uma determinada declaração da linguagem imperativa, retorna um identificador único no sistema dessa mesma declaração. Esta função de *hash* permitir-nos-á criar as diversas versões de cada variável.

A análise de um procedimento começa com um conjunto de versões vazio ($\mathcal{H} = \emptyset$) e com um grafo de dependências vazio ($\mathcal{D} = \emptyset$) onde vão sendo adicionados arcos (dependências) entre versões de variáveis. Mais uma vez, por uma questão de legibilidade, representaremos por v toda a variável local (v,m) $\in$ LocalVars declarada no método m que está a ser analisado. Representaremos por $m.p_i$ a única versão da variável especial que corresponde ao i-ésimo parâmetro do método m e por $returnVar$ a única versão da variável especial que representa o retorno do procedimento que está a ser analisado.

Sempre que temos uma afectação, através das regras ASSIGN, HEAP READ ou HEAP WRITE, são criadas dependências entre todas as versões da variável usada na afectação e

$$\boxed{\langle \mathcal{D}, \mathcal{H}, S\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle}$$

$$\frac{\langle \mathcal{D}, \mathcal{H}, S_1\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle \qquad \langle \mathcal{D}', \mathcal{H}', S_2\rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}''\rangle}{\langle \mathcal{D}, \mathcal{H}, S_1; S_2\rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}''\rangle}(\textsc{Seq})$$

$$\frac{h = \mathsf{getHash}(x := y) \qquad \mathcal{H}' = \mathsf{Subs}_{\mathcal{H}}((x, h)) \qquad \mathcal{D}' = \mathcal{D} \cup \{v_i \to (x, h) | v_i \in \mathsf{Ver}_{\mathcal{H}}(y)\}}{\langle \mathcal{D}, \mathcal{H}, x := y\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle}(\textsc{Assign})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(y) \qquad h = \mathsf{getHash}(x := y.f) \\ \mathcal{H}' = \mathsf{Subs}_{\mathcal{H}}((x, h)) \qquad \mathcal{D}' = \mathcal{D} \cup \{v_i \to (x, h) | v_i \in \mathsf{Ver}_{\mathcal{H}}((c, f))\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := y.f\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle}(\textsc{Heap Read})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(x) \qquad h = \mathsf{getHash}(x.f := y) \\ \mathcal{H}' = \mathsf{Subs}_{\mathcal{H}}(((c, f), h)) \qquad \mathcal{D}' = \mathcal{D} \cup \{v_i \to ((c, f), h) | v_i \in \mathsf{Ver}_{\mathcal{H}}(y)\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x.f := y\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle}(\textsc{Heap Write})$$

$$\frac{h = \mathsf{getHash}(x := \mathsf{new}\, C()) \qquad \mathcal{H}' = \mathsf{Subs}_{\mathcal{H}}((x, h))}{\langle \mathcal{D}, \mathcal{H}, x := \mathsf{new}\, C()\rangle \Longrightarrow \langle \mathcal{D}, \mathcal{H}'\rangle}(\textsc{Allocation})$$

$$\frac{\begin{array}{c} h = \mathsf{getHash}(x := \mathit{func}(\vec{y})) \qquad \mathsf{spec}(\mathit{func}) = \langle \mathcal{D}_f, \mathcal{H}_f\rangle \qquad \mathcal{D}' = \mathcal{D}_f \cup \mathcal{D} \\ \mathcal{D}'' = \mathcal{D}' \cup \{v_i \to (\mathit{func}.p_i) | y_i \in \vec{y}, v_i \in \mathsf{Ver}_{\mathcal{H}}(y_i)\} \cup \{(\mathsf{returnVar}) \to (x, h)\} \\ \mathcal{H}' = \mathsf{Subs}_{\mathcal{H}}((x, h)) \qquad \mathcal{H}'' = \{(v, h_v) | (v, h_v) \in \mathcal{H}' \wedge ((v, h_?) \in \mathcal{H}_f \vee (v, h') \notin \mathcal{H}_f)\} \\ \mathcal{H}''' = \{(v, h_v) | (v, h_v) \in \mathcal{H}_f \wedge h_v \neq h_?\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := \mathit{func}(\vec{y})\rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}'' \cup \mathcal{H}'''\rangle}(\textsc{Func Call})$$

$$\frac{\begin{array}{c} \mathsf{spec}(\mathit{proc}) = \langle \mathcal{D}_p, \mathcal{H}_p\rangle \qquad \mathcal{D}' = \mathcal{D}_p \cup \mathcal{D} \qquad \mathcal{D}'' = \mathcal{D}' \cup \{v_i \to (\mathit{proc}.p_i) | x_i \in \vec{x}, v_i \in \mathsf{Ver}_{\mathcal{H}}(x_i)\} \\ \mathcal{H}' = \{(v, h) | (v, h) \in \mathcal{H} \wedge ((v, h_?) \in \mathcal{H}_p \vee (v, h') \notin \mathcal{H}_p)\} \qquad \mathcal{H}'' = \{(v, h) | (v, h) \in \mathcal{H}_p \wedge h \neq h_?\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathit{proc}(\vec{x})\rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}' \cup \mathcal{H}''\rangle}(\textsc{Proc Call})$$

$$\frac{\begin{array}{c} \langle \mathcal{D}, \mathcal{H}, S_1\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle \qquad \langle \mathcal{D}, \mathcal{H}, S_2\rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}''\rangle \\ \mathcal{H}''' \triangleq \mathcal{H}' \cup \mathcal{H}'' \cup \{(v, h_?) | (v, h_1) \in \mathcal{H}' \wedge (v, h_2) \notin \mathcal{H}''\} \cup \quad \{(v, h_?) | (v, h_1) \in \mathcal{H}'' \wedge (v, h_2) \notin \mathcal{H}'\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{if}\, e\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2\rangle \Longrightarrow \langle \mathcal{D}' \cup \mathcal{D}'', \mathcal{H}'''\rangle}(\textsc{Conditional})$$

$$\frac{\begin{array}{c} \langle \mathcal{D}, \mathcal{H}, S\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle \\ \mathcal{H}'' \triangleq \mathcal{H} \cup \mathcal{H}' \cup \{(v, h_?) | (v, h_1) \in \mathcal{H} \wedge (v, h_2) \notin \mathcal{H}'\} \cup \{(v, h_?) | (v, h_1) \in \mathcal{H}' \wedge (v, h_2) \notin \mathcal{H}\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{while}\, e\, \mathsf{do}\, S\rangle \Longrightarrow \langle \mathcal{D} \cup \mathcal{D}', \mathcal{H}''\rangle}(\textsc{Loop})$$

$$\frac{\mathcal{D}' = \mathcal{D} \cup \{v_i \to (\mathsf{returnVar}) | v_i \in \mathsf{Ver}_{\mathcal{H}}(e)\}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{return}\, e\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}\rangle}(\textsc{Return})$$

$$\frac{}{\langle \mathcal{D}, \mathcal{H}, \mathsf{skip}\rangle \Longrightarrow \langle \mathcal{D}, \mathcal{H}\rangle}(\textsc{Skip})$$

Figura 3.8: Regras de Execução Simbólica da Análise de Dependências de Dados

a nova versão da variável afectada. Por outro lado, tendo sido criada uma nova versão de variável, o conjunto das versões actuais é actualizado usando o identificador (*hash*) único da declaração da linguagem.

Na regra PROC CALL, o procedimento invocado é analisado, retornando dois conjuntos que representam as dependências e as versões actuais das variáveis desse procedimento. As dependências retornadas são unidas às anteriores. Para além destas, para cada parâmetro do procedimento invocado, acrescentamos uma dependência entre a variável especial que o representa e a variável passada como argumento. A regra FUNC CALL é semelhante acrescentando apenas mais uma dependência entre o retorno do método invocado e a variável afectada.

O tratamento das versões actuais das variáveis nestas declarações da linguagem é um pouco mais complexo. Primeiro, como as variáveis podem ter sido afectadas no procedimento invocado, sobrepondo as suas versões anteriores, retiramos as versões anteriores que se tornaram obsoletas (que já não são actuais). Assim, sempre que no conjunto retornado, $\mathcal{H}_p$, existir uma versão (v,h) e não existir a versão (v,$h_?$) dessa variável, i.e., a variável foi redefinida, retiramos todas as versões de v do conjunto $\mathcal{H}$. Depois, retiramos todas as versões (v,$h_?$) do conjunto $\mathcal{H}_p$ já que estas versões correspondem às versões antigas anteriores à chamada do procedimento. No final, o conjunto das versões actuais, depois da invocação do procedimento, consiste na união destes dois novos conjuntos.

Na regra CONDITIONAL, as dependências geradas em ambos os ramos são adicionadas ao conjunto $\mathcal{D}$ inicial. O mesmo é feito no que diz respeito às versões actuais, com uma pequena subtileza: sempre que uma variável v é afectada apenas num dos ramos, temos de guardar informação sobre as duas versões possíveis dessa variável. Para tal, acrescentamos a versão (v,$h_?$) ao conjunto das versões actuais. Assim, se a execução do programa seguir o ramo onde a variável é redefinida, obtemos a nova versão gerada pela afectação da mesma. No entanto, se a execução do programa seguir o ramo onde a variável não é redefinida, continuamos a usar a versão anterior, i.e., (v,$h_?$).

A regra Loop é semelhante à anterior. As restantes regras são auto-explicativas.

Observações

A análise descrita anteriormente cria um grafo com todas as dependências de dados, geradas através das afectações do programa. Isto permite-nos criar o fluxo de dados de cada variável sabendo, por exemplo, se uma variável temporária reflecte o valor de uma determinada variável global, ou se valor de uma variável saiu do escopo de uma transacção específica. Esta informação será utilizada na implementação de ambos os sensores da ferramenta MoTh.

Listagem 3.9: Exemplo de código que ilustra uma dependência de controlo

```
1 int x,y = 0;
2 if(y > 0){
3   x = 2;
4 }
```

3.2.5 Análise de Dependências de Controlo

Introdução

Apesar de nos dar uma ideia do fluxo de dados do programa, a análise de dependência de dados é insuficiente para apurar todas as dependências entre variáveis. Na Listagem 3.9 é ilustrado um fragmento de código onde as dependências entre variáveis não são detectáveis através da análise de dependências de dados. Apesar de não existir nenhuma dependência de dados directa entre as variáveis x e y, o valor da primeira depende indubitavelmente da segunda, já que a afectação $x = 2$ só é executada se o valor de y for superior a 0. Referir-nos-emos a este tipo de dependências como dependências de controlo.

A análise descrita em seguida detecta as dependências de controlo do programa, completando o grafo de dependências criado pela análise anterior.

Descrição

Sempre que uma afectação/retorno de uma variável depende de uma determinada condição, i.e., só é executada se essa condição for verdadeira, a variável afectada/retornada depende dessa condição. Este cenário pode ocorrer com qualquer construção condicional como é o caso das construções if-then-else, for, while, entre outras.

A análise de dependências de controlo começa por percorrer o *Control Flow Graph* do programa apurando, em cada momento, o conjunto de variáveis condição das quais as afectações podem depender. No contexto deste trabalho, uma variável condição é uma variável booleana cujo valor influencia a execução de pelo menos uma declaração da linguagem. A Listagem 3.10 ilustra este conceito, apontando o conjunto de variáveis condição das quais as afectações/retornos de uma determinada região de código dependeriam.

Note-se que a linguagem Jimple da ferramenta Soot, transforma qualquer instrução do tipo $if(condition)$ em duas operações: $b = condition$ e $if(b)$. Assim, para cada instrução condicional teremos uma única variável condição. Por exemplo, se tivermos a instrução $if(x > y)$, é criada uma variável condição que depende das variáveis x e y e que será posteriormente utilizada neste algoritmo.

Finalmente, sempre que encontramos uma afectação de uma variável v, criamos uma

Listagem 3.10: Fragmento de código que ilustra as variáveis condição de cada região

```
boolean b1,b2,b3,b4;
if(b1){
  //depends on "b1"
  if(b2){
    //depends on "b1" and "b2"
  }
  else if(b3){
    //depends on "b1", "b2" and "b3"
  }
  else{
    //depends on "b1", "b2" and "b3"
  }
  if(b4){
    //depends on "b1" and "b4"
  }
  //depends on "b1"
}
```

dependência de controlo entre as versões únicas de cada variável condição e a nova versão de v.

A Figura 3.9 ilustra um exemplo mais complexo de uma função que avalia se um número é superior a outro, com o respectivo grafo de dependências (de dados e de controlo). No grafo, a variável criada pelo Jimple que recebe o valor resultante da comparação $if(x > y)$ foi representada por bool(if). Note-se que ambos os tipos de dependências são fulcrais para detectar a dependência entre os parâmetros do método gT e o seu retorno.

```
boolean gT(int x1,int x2) {
  if(x1>x2){
    return true;
  }
  return false;
}
```

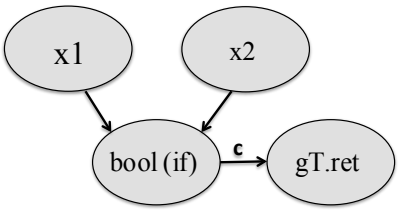


Figura 3.9: Exemplo de um método com dependências de dados e de controlo

Execução Simbólica

Na Figura 3.10 são apresentadas as regras de execução simbólica desta análise para todas as declarações da linguagem imperativa. A informação passada entre as declarações da linguagem é constituída por dois conjuntos: $\mathcal{IS}$ e $\mathcal{D}$. O primeiro contém todas as versões das variáveis condição que afectam a declaração corrente a ser analisada, i.e., correspondem às variáveis condição de cada região do programa, tal como foi ilustrado na Listagem 3.10. O segundo, $\mathcal{D}$, consiste no conjunto de todas as dependências de controlo computadas até ao momento.

No início da análise de cada procedimento, o conjunto $\mathcal{D}$ encontra-se vazio e $\mathcal{IS}$ contém a união de todas as versões de variáveis condição que afectam cada chamada desse procedimento. Assim, se p_1, p_2 e p_3 são procedimentos do programa, sendo que p_1 chama p_2 com $\mathcal{IS} = \{b_1, b_2\}$ e p_3 chama p_2 com $\mathcal{IS} = \{b_3, b_4\}$ então, no início da análise de p_2, o conjunto $\mathcal{IS}$ corresponde ao conjunto $\{b_1, b_2, b_3, b_4\}$.

No final da análise, o conjunto $\mathcal{D}$ é unido com o conjunto obtido na Análise de Dependências de Dados (com o mesmo nome), criando um único grafo com todas as dependências entre variáveis do programa (de dados e controlo). Como cada variável condição tem apenas uma única versão, ao longo da formalização desta análise representaremos apenas por v a única versão da variável condição v.

Sempre que temos uma afectação de uma variável, ligamos através de uma dependência cada uma das variáveis condição à nova versão da variável afectada. Este cenário ocorre nas regras ASSIGN, HEAP READ, HEAP WRITE, ALLOCATION e FUNC CALL. Quando uma variável é retornada, i.e., na regra RETURN, o tratamento é semelhante.

Nas regras CONDITIONAL e LOOP, devido à existência de uma nova região condicional, é adicionada a versão única da variável condição utilizada ao conjunto $\mathcal{IS}$. Quando saímos do escopo dessa região condicional, a versão da variável utilizada é de novo removida do conjunto. Por questões de legibilidade, a versão única da variável condição c foi representada apenas por c.

As restantes regras de execução simbólica são auto-explicativas.

Observações

Esta análise permite-nos completar o grafo de dependências, criando dependências que não seriam detectadas pela análise de dependências de dados. Esta informação é valiosa para criar o fluxo de dados de cada variável do programa.

3.2.6 Sumário

Nesta secção foram descritas algumas análises que permitem computar toda a informação necessária que será usada pelos Sensores na detecção de *dataraces*. Para além destas, foram ainda descritas outras análises complementares que permitem aumentar a precisão das primeiras e, consequentemente, da ferramenta.

Assim, podemos resumir a informação disponibilizada aos Sensores da ferramenta:

- Conjunto $\mathcal{P}s$ com todos os processos gerados pelo programa.
- Para cada processo $p \in \mathcal{P}s$, os conjuntos das *views* de leitura e escrita executadas por este representados, respectivamente, por $V_r(p)$ e $V_w(p)$.
- Um grafo DepGraph com todas as dependências de dados e controlo entre as versões das variáveis do programa.

Note-se que a estrutura das análises implementadas é completamente modelar, sendo possível criar e integrar novas análises no sistema.

$$\boxed{\langle \mathcal{IS}, \mathcal{D}, S\rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}'\rangle}$$

$$\frac{\langle \mathcal{IS}, \mathcal{D}, S_1\rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}'\rangle \qquad \langle \mathcal{IS}', \mathcal{D}', S_2\rangle \Longrightarrow \langle \mathcal{IS}'', \mathcal{D}''\rangle}{\langle \mathcal{IS}, \mathcal{D}, S_1; S_2\rangle \Longrightarrow \langle \mathcal{IS}'', \mathcal{D}''\rangle}(\textsc{Seq})$$

$$\frac{h = \mathsf{getHash}(x := y) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \to (x, h) |\ v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := y\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}'\rangle}(\textsc{Assign})$$

$$\frac{h = \mathsf{getHash}(x := y.f) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \to (x, h) |\ v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := y.f\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}'\rangle}(\textsc{Heap Read})$$

$$\frac{c = \mathsf{typeof}(x) \qquad h = \mathsf{getHash}(x.f := y) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \to ((c, f), h) |\ v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x.f := y\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}'\rangle}(\textsc{Heap Write})$$

$$\frac{h = \mathsf{getHash}(x := \mathsf{new}\, C()) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \to (x, h) |\ v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := \mathsf{new}\, C()\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}'\rangle}(\textsc{Allocation})$$

$$\frac{\begin{array}{c} h = \mathsf{getHash}(x := \mathit{func}(\vec{y})) \qquad \mathsf{spec}(\mathit{func}) = \langle \mathcal{IS}_f, \mathcal{D}_f\rangle \\ \mathcal{D}' = \mathcal{D} \cup \mathcal{D}_f \cup \{v \to (x, h) |\ v \in \mathcal{IS}\} \end{array}}{\langle \mathcal{IS}, \mathcal{D}, x := \mathit{func}(\vec{y})\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}'\rangle}(\textsc{Func Call})$$

$$\frac{\mathsf{spec}(\mathit{proc}) = \langle \mathcal{IS}_p, \mathcal{D}_p\rangle}{\langle \mathcal{IS}, \mathcal{D}, \mathit{proc}(\vec{x})\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D} \cup \mathcal{D}_p\rangle}(\textsc{Proc Call})$$

$$\frac{\mathcal{IS}' = \mathcal{IS} \cup \{c\} \qquad \langle \mathcal{IS}', \mathcal{D}, S_1\rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}'\rangle \qquad \langle \mathcal{IS}', \mathcal{D}, S_2\rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}''\rangle}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{if}\, c\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \cup \mathcal{D}''\rangle}(\textsc{Conditional})$$

$$\frac{\mathcal{IS}' = \mathcal{IS} \cup \{c\} \qquad \langle \mathcal{IS}', \mathcal{D}, S\rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}'\rangle}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{while}\, c\, \mathsf{do}\, S\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D} \cup \mathcal{D}'\rangle}(\textsc{Loop})$$

$$\frac{\mathcal{D}' = \mathcal{D} \cup \{v \to \mathit{returnVar} |\ v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{return}\, e\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}'\rangle}(\textsc{Return})$$

$$\frac{}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{skip}\rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}\rangle}(\textsc{Skip})$$

Figura 3.10: Regras de Execução Simbólica da Análise de Dependências de Controlo

3.3 Sensores

A detecção de *dataraces* na ferramenta MoTh é conseguida através da adição de novos Sensores (*plugins*) ao sistema. No entanto, os conjuntos de conflitos detectados por cada Sensor não têm que ser necessariamente disjuntos, já que estes são unidos antes de serem apresentados ao utilizador.

Cada Sensor implementa um algoritmo desenhado especificamente para detectar um determinado tipo de *datarace*. Estes algoritmos são completamente independentes uns dos outros e podem ser executados em paralelo de forma a melhorar consideravelmente a eficiência da ferramenta.

Até ao momento, foram definidos dois tipos de Sensores que detectam a maioria dos *dataraces* reportados na literatura. No entanto, a estrutura da ferramenta MoTh faz com que esta seja completamente extensível: se um determinado *datarace* não for detectado pelos nossos Sensores, um novo Sensor pode ser desenhado, implementado e integrado na ferramenta com o intuito de detectar esse tipo de conflitos.

As próximas secções descrevem cada um dos Sensores implementados na ferramenta MoTh. Para cada um destes Sensores, será apresentado um exemplo que ilustre o seu funcionamento.

3.3.1 ViewConsistency Sensor

O primeiro Sensor foi desenhado para detectar todos os *dataraces* relacionados com acessos parciais a conjuntos atómicos de variáveis que devem ser acedidos como um todo. O algoritmo descrito em seguida é uma extensão do conceito de *view consistency* apresentado em [AHB03], incorporando a distinção entre acessos de leitura e de escrita.

As *maximal views* de um processo são todas aquelas que não são subconjuntos de outras *views* desse processo, e representam os conjuntos de variáveis que devem ser acedidas atomicamente. Assim, acessos parciais a estes conjuntos geram *dataraces*. Tal como na geração das *views*, também aqui distinguimos as *maximal views* de leitura (M_r) e as *maximal views* de escrita (M_w). Assim, sendo $\alpha \in \{r, w\}$, obtemos:

$$v_m \in M_\alpha(p) \Leftrightarrow v_m \in V_\alpha(p) \wedge (\forall v \in V_\alpha(p) \ : v_m \subseteq v \Rightarrow v = v_m)$$

Dado o conjunto de *views* de um processo p e a *maximal view* de outro processo v_m, as *overlapping views* de leitura/escrita de p com v_m são todas as intersecções não vazias das *views* de leitura/escrita de p com v_m. Assim, sendo $\alpha \in \{r, w\}$, obtemos:

$$\mathsf{overlap}_\alpha(p, v_m) \triangleq \{v_m \cap v \,|\, (v \in V_\alpha(p)) \wedge (v_m \cap v \neq \emptyset)\}$$

Segundo [AHB03], a compatibilidade entre um processo p e a *maximal view* v_m de outro processo é verificada se e só se todas as *overlapping views* de p com v_m formarem uma cadeia. No entanto, para garantir que as leituras parciais de um conjunto atómico foram

Algoritmo 2: Função reachedVars

```
Input: view
Result: Set<Version>
workSet = new Set<Version>();
visited = new Set<Version>();
foreach Access a : view do
    foreach (a.var, h, Γ(view)) : DepGraph.vertices() do
        workSet.add((a.var, h, Γ(view)));
    end foreach
end foreach
while !workSet.isEmpty() do
    Version v = workSet.removeFirst();
    if !visited.contains(v) then
        visited.add(v);
        foreach Version s : getSuccs(v, DepGraph) do
            workSet.add(s);
        end foreach
    end if
end while
return visited;
```

feitas com um objectivo comum, devendo ser encapsuladas numa leitura única, estendemos este conceito usufruindo da informação disponibilizada pelo grafo de dependências. Assim, mesmo que duas *overlapping views* de leitura não formem uma cadeia, só consideraremos a ocorrência de um *datarace* se as variáveis que dependem de cada uma não forem disjuntas. Com isto, sendo $\alpha \in \{r, w\}$, temos que:

$$\mathsf{comp}_w(p, v_m) \Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_w(p, v_m) \ : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$$
$$\mathsf{comp}_r(p, v_m) \Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_r(p, v_m) \ : v_1 \subseteq v_2 \vee v_2 \subseteq v_1 \vee$$
$$\mathsf{reachedVars}(v_1) \cap \mathsf{reachedVars}(v_2) = \emptyset$$

A função auxiliar reachedVars é implementada através do Algoritmo 2 que consiste num algoritmo de inundação no grafo DepGraph. O algoritmo começa por computar o conjunto de variáveis da *view* passada como parâmetro, guardando-as numa estrutura de dados workSet. Depois, iterativamente, os elementos desse conjunto vão sendo substituídos pelos seus sucessores no grafo de dependências, sendo sempre guardada a informação sobre os nós visitados numa segunda estrutura de dados visited. No final, o conjunto workSet encontra-se vazio, enquanto o conjunto visited contém todos os nós atingíveis a partir das variáveis da *view* passada como parâmetro.

Note-se que, intencionalmente, nas definições de $\mathsf{overlap}_\alpha$ e comp_α, nada é dito sobre o tipo da *maximal view* v_m, já que estas são aplicáveis independentemente da maximal *view* ser de leitura ou de escrita.

Finalmente, o conceito de *view consistency* é definido através da mútua compatibilidade entre todos os processos. Um processo só pode ter *views* de leitura/escrita que são compatíveis com todas as *maximal views* de escrita/leitura de todos os outros processos.

Listagem 3.11: Exemplo que ilustra o funcionamento do ViewConsistency Sensor (parte 1)

```
// Classe Pair

@Atomic
public void setPair(int x, int y){
  this.first = x;
  this.second = y;
}

@Atomic
public void setFirst(int x){...}

@Atomic
public void setSecond(int y){...}

@Atomic
public int getFirst(){...}

@Atomic
public int getSecond(){...}

public int getSum(){
  int x = getFirst();
  // x pode ter sido alterado
  int y = getSecond();
  return x + y;
}
```

Listagem 3.12: Exemplo que ilustra o funcionamento do ViewConsistency Sensor (parte 2)

```
// Processo p1

public void run() {
  Random r = new Random();
  while(true){
    sleep(50);
    int n1 = r.nextInt();
    int n2 = r.nextInt();
    pair.setFirst(n1);
    pair.setSecond(n2);
    pair.setPair(n1,n2);
  }
}


// Processo p2

public void run() {
  while(true){
    sleep(60);
    int sum  = pair.getSum();
    System.out.println(sum);
  }
}
```

Assim, definimos a seguinte propriedade que tem de ser verificada de forma a garantir a ausência deste tipo de *dataraces*:

Propriedade 1 (Compatibilidade de *Views*)

$$\forall p_1, p_2 \in \mathcal{P}s, m_r \in M_r(p_1), m_w \in M_w(p_1) \ : \mathsf{comp_w}(\mathsf{p_2}, \mathsf{m_r}) \wedge \mathsf{comp_r}(\mathsf{p_2}, \mathsf{m_w})$$

Exemplo

Para exemplificar o funcionamento do algoritmo descrito, considerem-se as Listagens 3.11 e 3.12 que ilustram uma implementação propensa a erros de um objecto que representa um par de inteiros e que é partilhado por dois processos. A classe Pair disponibiliza métodos atómicos que actualizam o par (parcial e globalmente), os métodos getFirst e getSecond que permitem obter cada um dos elementos do mesmo e um método não atómico que retorna a soma dos dois utilizando os dois métodos anteriores.

O processo p_1 limita-se a fazer acessos de escrita actualizando o estado parcial e global do par, enquanto o processo p_2 executa dois métodos atómicos de forma a imprimir constantemente a soma dos elementos do par. Para cada processo, podemos calcular o respectivo conjunto de *views* de leitura e escrita, i.e., os conjuntos de acessos feitos dentro do escopo de métodos atómicos executados pelo mesmo. As *views* geradas pelo programa

são apresentadas na Tabela 3.2.

Processo	*Views*	Acessos
p_1	V_r	$\emptyset$
	V_w	$\{(w, Pair.first)\}, \{(w, Pair.second)\}, \{(w, Pair.first), (w, Pair.second)\}$
p_2	V_r	$\{(r, Pair.first)\}, \{(r, Pair.second)\}$
	V_w	$\emptyset$

Tabela 3.2: *Views* dos processos ilustrados na Listagem 3.12

Com esta informação, podemos calcular as *maximal views* do programa. Intuitivamente, o conjunto de *maximal views* é obtido retirando, do conjunto das *views*, aquelas que estão contidas dentro de outras. A Tabela 3.3 contém as *maximal views* de leitura e escrita dos processos p_1 e p_2.

A computação das *overlapping views* é feita para todas as possíveis combinações entre as *views* de cada processo e as *maximal views* dos outros processos do programa. Por exemplo, as *overlapping views* de leitura entre o processo p_2 e a única *maximal view* de escrita do processo $p1$ correspondem às próprias *views* de leitura de p_2, já que estas estão contidas na *maximal view* referida:

$$\mathsf{overlap}_r(p_2, \{(w, Pair.first), (w, Pair.second)\}) = \{\{(r, Pair.first)\}, \{(r, Pair.second)\}\}$$

Finalmente, podemos constatar que a propriedade de compatibilidade de leitura não é verificada entre o processo e a *maximal view* referidos, já que as suas *overlapping views* de leitura não formam uma cadeia ($\{Pair.first\} \not\subseteq \{Pair.second\} \wedge \{Pair.second\} \not\subseteq \{Pair.first\}$), e existe uma variável que depende de variáveis de ambas ($first \rightarrow sum$ e $second \rightarrow sum$).

Assim, seria reportado um *datarace* já que a propriedade de Compatibilidade de *Views* não foi verificada. De facto, este programa pode ter um comportamento inconsistente se, entre as chamadas dos dois métodos atómicos do método getSum, o par for actualizado com o método setPair. Nesse caso, obteríamos a soma do antigo valor de *first* com o novo valor de *second*.

3.3.2 Dependency Sensor

Mesmo com a extensão das *views* de forma a suportarem distinção entre leituras e escritas, o conceito de *view consistency* é insuficiente, deixando por detectar outros tipos de anomalias, tais como os *stale-value errors*. Assim, para obter uma ferramenta mais completa e eficiente, adicionámos um novo Sensor desenhado para detectar *dataraces* gerados por uma única variável como, por exemplo, os *stale-value errors*.

Como exemplo deste tipo de conflitos, considere-se um processo que lê o valor de uma variável partilhada num bloco atómico e, com base no valor observado, actualiza o valor dessa variável num segundo bloco atómico. O valor lido inicialmente pode ter sido

Processo	*Views*	Acessos
p_1	M_r	$\emptyset$
	M_w	$\{(w, Pair.first), (w, Pair.second)\}$
p_2	M_r	$\{(r, Pair.first)\}, \{(r, Pair.second)\}$
	M_w	$\emptyset$

Tabela 3.3: *Maximal views* dos processos ilustrados na Listagem 3.12

alterado entretanto por outro processo, resultando numa actualização perdida por parte do outro processo.

O nosso algoritmo utiliza o grafo de dependências para detectar quando o valor de uma determinada variável sai do escopo de um bloco atómico para outro detectando, portanto, possíveis *Stale-Values*.

Em primeiro lugar, precisamos de apurar que tipos de variáveis podem ter o comportamento referido. Assim, definimos o conjunto AVersions $\subseteq$ Versions que representa o conjunto de variáveis que podem sair do escopo de um determinada bloco atómico. Tal como foi referido na Análise de Dependências de Dados, cada versão de uma variável tem associada a informação do método atómico onde foi utilizada, ou $\perp$ caso tenha sido utilizada fora do contexto transaccional. Assim, o conjunto AVersions contém todas as versões de variáveis que são globais e que foram utilizadas dentro do escopo de uma transacção:

$$(v, h, m) \in \mathsf{AVersions} \Leftrightarrow (v, h, m) \in \mathsf{Versions} \ \wedge v \in \mathsf{GloVars} \ \wedge m \neq \perp$$

Seja Deps o conjunto de dependências entre métodos atómicos de um determinado programa. Um elemento deste conjunto consiste num triplo (a_1,var,a_2), com $a_1, a_2 \in$ Atomics, sendo que a_2 depende de a_1 através da variável var. Por outras palavras, o valor da variável var saiu do escopo do método atómico a_1 e entrou em a_2 sendo que, entre estes dois, o valor original de var poderia ter sido modificado por outro processo.

Em súmula, uma dependência entre métodos atómicos (a_1,v_1,a_2) é criada quando todas as seguintes condições são verificadas:

- $i_1, i_2 \in \mathsf{AVersions} : i_1 = (v_1, h_1, a_1)\ ,\ i_2 = (v_2, h_2, a_2)$;
- $\mathsf{hasPath}(\mathsf{DepGraph}, i_1, i_2)$;
- $a_1 \neq a_2 \vee (\mathsf{hasPath}(\mathsf{DepGraph}, i_1, a_1.return) \wedge \ \mathsf{hasPath}(\mathsf{DepGraph}, a_1.return, a_2.p_i) \wedge \ \mathsf{hasPath}(\mathsf{DepGraph}, a_2.p_i, i_2))$

Intuitivamente, as primeira duas condições ditam que existem duas versões de variáveis i_1 e i_2, criadas nos métodos atómicos da dependência respectivamente, e que existe um caminho no grafo entre a primeira e a segunda.

Na última condição, se os métodos atómicos das duas versões são distintos, significa que o valor de i_1 saiu do escopo do primeiro para o segundo bloco atómico sendo, por

isso, criada uma dependência entre os mesmos. Por outro lado, se ambas as variáveis tiverem o mesmo método atómico ($a_1 = a_2$), temos que garantir que o valor da variável i_1 saiu e voltou a entrar dentro escopo desse método atómico (caso contrário, o valor manteve-se sempre dentro do escopo transaccional). Assim, só criamos uma dependência se o valor da primeira variável i_1 sai do escopo do método atómico através do seu retorno, volta a entrar num dos parâmetros do método, e é usada para afectar a segunda variável i_2.

Um processo p escreve numa variável $var \in \mathsf{Vars}$ se existe um acesso de escrita daquela variável numa das *views* de escrita desse processo:

$$writes(var, p) \Leftrightarrow \exists v \in V_w(p) : (w, var) \in v$$

Finalmente, tendo definido o conjunto de dependências entre métodos atómicos (ou transacções), a detecção de *dataraces* é feita através do conceito de compatibilidade entre processos e dependências. Nenhum processo pode escrever numa variável que sai do escopo de um método atómico para outro, i.e., que está presente numa dependência entre métodos atómicos. Assim, definimos a seguinte propriedade que tem de ser verificada de forma a garantir a ausência deste tipo de *dataraces*:

Propriedade 2 (Compatibilidade de Dependências)

$$\forall p \in \mathcal{P}s, (a_1, var, a_2) \in \mathsf{Deps} : \neg writes(var, p)$$

Se esta propriedade não for verificada, então é reportado um *datarace* ao utilizador sendo referidas as três transacções envolvidas no conflito (as transacções dos métodos atómicos a_1 e a_2, e a transacção do processo p que escreve na variável var).

Exemplo

Para exemplificar o funcionamento deste Sensor, considere-se a Listagem 3.13 que descreve um exemplo de um programa com um *stale-value error*. Este programa foi retirado da literatura [vG03] e representa um dos testes utilizados na validação da ferramenta.

O grafo de dependências do programa é representado na Figura 3.11. O retorno do método inc é guardado na variável temporária i do método run. Depois, essa variável é usada como parâmetro de uma nova chamada do método inc. No método inc, o valor antigo de i e a variável local a (ou $inc.p_0$) são usados para afectar novamente i. O novo valor de i é finalmente retornado.

Em primeiro lugar, de todas as versões de variáveis existentes, calculamos aquelas que podem ter saído do escopo de uma transacção, i.e., as variáveis globais utilizadas

Listagem 3.13: Programa adaptado de [vG03] que ilustra o funcionamento do Dependency Sensor

```
1  //Classe Counter
2  int i;
3
4  @Atomic
5  int inc(int a) {
6    i += a;
7    return i;
8  }
9
10 //Processo p1
11 static Counter c;
12
13 public void run() {
14   int i = c.inc(0);
15   c.inc(i); // Valor do contador pode ter sido alterado
16 }
```

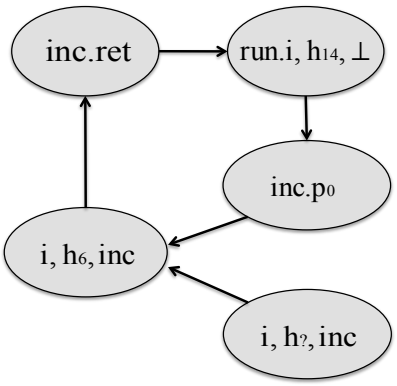


Figura 3.11: Grafo de dependências do fragmento de código da Listagem 3.13

dentro de um método atómico:

$$\text{Versions} = \{(inc.ret), (run.i, h_{14}, \bot), (inc.p_0), (Counter.i, h_?, inc), (Counter.i, h_6, inc)\}$$
$$\text{AVersions} = \{(Counter.i, h_?, inc), (Counter.i, h_6, inc)\}$$

Tal como se pode observar no grafo de dependências, existe um caminho da versão $(Counter.i, h_6, inc)$ para ela própria. Assim, as primeiras duas condições para a criação de uma dependência de métodos atómicos foram verificadas. Em relação à terceira condição, apesar de a_1 e a_2 corresponderem ao método inc, existe de facto um caminho entre qualquer versão de i e o retorno desse método, um caminho entre o retorno e o primeiro parâmetro desse método e, finalmente, um caminho do parâmetro de novo para essa versão da variável i. Assim, podemos concluir que a variável i saiu do escopo do método atómico voltando a entrar e, por isso, é criada a dependência entre métodos atómicos (inc, *Counter.i*, inc).

O processo p_1 escreve na variável *Counter.i* já que existe um acesso de escrita a essa mesma variável na *view* gerada pelo método atómico inc executado por p_1. Finalmente, podemos constatar que a propriedade de Compatibilidade de Dependências não é verificada, já que existe uma dependência entre métodos atómicos com uma variável que é escrita num determinado processo.

De facto, os autores deste teste referem que existe um *datarace* que ocorre se, entre as duas chamadas do método inc feitas no método run, outro processo escrever na variável i executando, por exemplo, o método inc.

3.3.3 Sumário

Foram criados dois Sensores que, até o momento, detectam a maioria dos *dataraces* apresentados na literatura. Intuitivamente, o primeiro Sensor detecta conflitos gerados por acessos parciais a conjuntos de variáveis que deviam ser acedidos atomicamente, i.e., *high-level dataraces*, e o segundo detecta cenários onde uma única variável sai do escopo de uma transacção para outra, que correspondem a *stale-value errors*.

Apesar disto, e tal como foi referido no início desta secção, outros Sensores podem ser adicionados como *plugins* à ferramenta MoTh de forma a detectar outro tipo de conflitos.

4

Validação e Resultados

4.1 Introdução

Neste capítulo será discutida a validação do nosso trabalho. Para tal, analisaremos tanto a precisão da detecção das anomalias como a robustez, viabilidade e escalabilidade do protótipo desenvolvido.

Começámos por validar os algoritmos e a ferramenta com um conjunto de testes simples que simulam cenários reais, onde existe a possibilidade de ocorrência de *dataraces*, e que a nossa ferramenta deveria detectar. Os testes apresentados foram retirados da literatura e usados para validar trabalhos relacionados [BBA08, AHB03, AHB04, FF04, vG03, IBM, TLS10], ou foram desenvolvidos por nós para testar alguns cenários específicos. A implementação de alguns dos testes descritos baseava-se no mecanismo de controlo de concorrência de *locks*, pelo que foi feita uma tradução para a semântica transaccional.

Foi implementada uma versão do algoritmo de *view consistency* [AHB03] na ferramenta MoTh, ainda que usando análise estática ao contrário da abordagem dinâmica dos autores, devido ao facto desta ter sido o ponto de partida do nosso trabalho. Para além desta, tivemos acesso à ferramenta de Teixeira [TLS10], que também se propõe a detectar estaticamente as anomalias abordadas neste trabalho. Assim, para cada teste, apresentaremos igualmente os resultados reportados por cada uma destas abordagens, permitindo uma percepção mais intuitiva da relevância e da precisão introduzida pela nossa abordagem.

Estes testes, apesar de permitirem avaliar a precisão da ferramenta, não nos permitem apurar se a ferramenta escala para testes maiores e mais complexos, i.e., se os consegue analisar em tempo útil. Assim, foram utilizados igualmente *benchmarks* também retirados da literatura ou desenvolvidos por nós, que nos permitiram analisar a performance da

ferramenta com testes cujo número de transacções e de variáveis globais acedidas dentro destas é substancialmente maior.

Computacionalmente, a parte mais exigente do nosso algoritmo é computada pelo *plugin* Dependency Sensor e consiste em fazer pesquisas de caminhos no grafo de dependências, para testar se existe caminho entre quaisquer duas variáveis globais acedidas dentro do escopo transaccional. Assim, com os *benchmarks* referidos, tentaremos analisar o comportamento da nossa ferramenta quando aumentamos o número de variáveis globais acedidas dentro de transacções do programa de entrada.

Na Secção 4.2 analisaremos pormenorizadamente um exemplo utilizado na validação do nosso trabalho, que permitirá seguir e compreender em detalhe o funcionamento da nossa abordagem. Na Secção 4.3, é feito um resumo dos resultados obtidos, disponibilizando uma comparação entre os resultados obtidos na ferramenta MoTH e nas abordagens de [AHB03] e [TLS10].

Todos os testes apresentados neste capítulo foram analisados num sistema Debian Linux 6.0, com um processador dual-core i5 650 a 3.20 GHz, 4GB de RAM e um disco rígido SATA Samsung P80 SD.

No Anexo 6 são descritos em detalhe todos os programas e *benchmarks* utilizados nesta validação. Para cada teste, começamos por fazer uma pequena descrição do mesmo, disponibilizando quando necessário uma parte do código de forma a ilustrar a semântica do programa. Em seguida são listados os *dataraces* presentes no teste que deveriam ser detectados pela nossa ferramenta. Finalmente, são apresentados os conflitos reportados pela nossa ferramenta, com uma breve discussão e análise dos resultados obtidos.

4.2 Exemplo

De forma a exemplificar o funcionamento completo da nossa abordagem apresentamos, nesta secção, a análise do teste *Connection*, adaptado de [BBA08]. Uma descrição menos exaustiva deste teste é também apresentada, em anexo, na Secção 6.1.

Este teste simula uma aplicação de *chat* em rede. Para tal, uma classe GUI (Graphical User Interface) utiliza os serviços da ligação de rede, representada aqui por um objecto Java, enviando mensagens e desconectando a ligação aleatoriamente. Para além deste objecto, outro representa um contador que contabiliza o número de mensagens enviadas para a rede. A maior parte do código é disponibilizado na Listagem 4.1.

Anomalias

Este teste contém dois *dataraces*, de natureza distinta:

1. O método disconnect reinicia atomicamente o contador e fecha a ligação de rede. No entanto, como estas operações são feitas em dois métodos atómicos distintos, outro processo pode ver um estado inconsistente entre as duas operações, onde a

Listagem 4.1: Teste: Connection (fragmento de código adaptado de [BBA08])

```
1  //Classe Connection
2  final Counter counter;
3  java.net.Socket socket;
4  @Atomic
5  boolean isConnected(){
6    return !socket.isClosed();
7  }
8  @Atomic
9  void send(String msg){
10   socket.write(msg);
11   counter.increment();
12 }
13 @Atomic
14 void closeSocket(){
15   socket.close();
16 }
17 void disconnect(){
18   closeSocket();
19   counter.reset(); //Atomic
20 }
21
22 //Classe GUI
23 void trySendMsg(String msg){
24   if(connection.isConnected()){
25     connection.send(msg);
26   }
27 }
```

ligação estará fechada mas onde o contador expressa um número positivo de mensagens enviadas. Esta anomalia está relacionada com acessos parciais ao objecto que representa a ligação de rede (*Connection*).

2. O método trySendMessage testa se a ligação está fechada no método atómico isConnected e, se não estiver, envia uma mensagem para a mesma no método send. No entanto, outro processo pode ter fechado a ligação antes da mensagem ser enviada. Este *datarace* deve-se ao facto da execução do envio atómico de uma mensagem (método send) depender do resultado do método atómico isConnected.

Computação da Informação Base

Neste exemplo podemos distinguir dois tipos de processos ou fios de execução: o processo principal (P_{main}) e o processo gerado pela classe GUI (P_{GUI}). Enquanto o primeiro não executa métodos atómicos, o segundo executa os métodos closeSocket, isConnected e send da classe Connection, e ainda o método reset da classe Counter.

Para cada um destes métodos atómicos, podemos computar as respectivas *views* de leitura e escrita, i.e., os acessos de leitura e escrita feitos a variáveis globais dentro do escopo de cada um destes métodos. Na Tabela 4.1 são apresentadas as *views* de leitura e

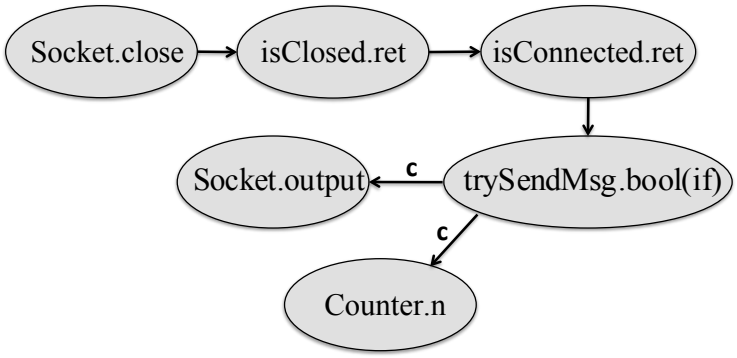


Figura 4.1: Parte do grafo de dependências gerado no Teste Connection

escrita geradas na análise deste programa. De forma a facilitar a compreensão do funcionamento da nossa abordagem, em cada *view* são apenas apresentados os acessos relevantes na detecção dos *dataraces* reportados.

Métodos	***Views***	**Acessos**
closeSocket	V_r V_w	$\emptyset$ $\{(w, java.net.Socket.close)\}$
reset	V_r V_w	$\emptyset$ $\{(w, testes.ConnectionTest.Counter.n)\}$
isConnected	V_r V_w	$\{(r, java.net.Socket.close)\}$ $\emptyset$
send	V_r V_w	$\{(r, testes.ConnectionTest.Counter.n), (r, java.net.Socket.close)\}$ $\{(w, testes.ConnectionTest.Counter.n), (w, java.net.Socket.outputStream)\}$

Tabela 4.1: *Views* de leitura e escrita geradas na análise do programa

Para além da informação relacionada com os acessos a variáveis globais, os Sensores recebem ainda informação relacionada com as dependências entre as variáveis do programa. Para este teste específico, foi gerado um grafo de dependências (de dados e controlo) com cerca de 10000 nós e 38000 arcos. Na Figura 4.1 é ilustrado parte do grafo de dependências gerado, onde o valor da variável Socket.close sai do escopo do método isConnected e é usado no teste de uma condição (representada pela variável *bool*). Por sua vez, o resultado dessa condição influencia a execução do método atómico send, i.e., a afectação do contador e do *socket*. Note-se que, de forma a simplificar o grafo disponibilizado, consideramos que o método send da classe Socket se limita a escrever num objecto, *output*, interno à classe.

Toda esta informação é disponibilizada como input para os Sensores e será útil na detecção dos conflitos.

View Consistency Sensor

Um dos *dataraces* documentados neste teste foi detectado pelo View Consistency Sensor. A *view* de leitura do processo P_{GUI} gerada pelo método send ($V_r(send)$) representa uma

das *maximal view* de leitura desse processo. Assim, como as *views* de escrita dos métodos reset ($V_w(reset)$) e closeSocket ($V_w(closeSocket)$) estão contidas nesta e não formam uma cadeia, foi reportado um *datarace* envolvendo estes três métodos atómicos. As condições que permitem a detecção deste conflito são, portanto, as seguintes:

$$V_w(\mathsf{closeSocket}) \subseteq M_r(\mathsf{send})$$
$$V_w(\mathsf{reset}) \subseteq M_r(\mathsf{send})$$
$$V_w(\mathsf{closeSocket}) \not\subseteq V_w(\mathsf{reset}) \cap V_w(\mathsf{reset}) \not\subseteq V_w(\mathsf{closeSocket})$$

Dependency Sensor

A segunda anomalia foi correctamente detectada pelo Dependency Sensor. Foi criada a dependência entre métodos atómicos (isConnected, Socket.close, send), já que as variáveis Socket.output e Counter.n, afectadas no método atómico send, dependem do valor da variável Socket.close lido anteriormente no método isConnected. Como, entre a execução destes dois métodos atómicos, outro processo pode alterar o valor da variável Socket.close, estamos na presença de um *datarace*. As condições que permitem a detecção deste conflito são, portanto, as seguintes:

$$i_1 = (Socket.close, h_1, isConnected)\ ,\ i_2 = (Counter.n, h_2, send)$$
$$i_1, i_2 \in \mathsf{AVersions}$$
$$\mathsf{hasPath}(\mathsf{DepGraph}, i_1, i_2)$$
$$isConnected \neq send$$
$$\mathsf{writes}(Socket.close, P_{\mathrm{GUI}})$$

Resumo

As duas anomalias foram correctamente detectadas. Não foi reportado nenhum *datarace* adicional. Zero falsos negativos e zero falsos positivos.

4.3 Discussão dos Resultados

Na validação da ferramenta MoTH, foi usado um conjunto de 20 testes retirados da literatura, utilizados anteriormente para validar outros trabalhos relacionados. Para além destes, foram ainda concebidos os testes Store e Vector Fail, também com o intuito de validar a nossa ferramenta. Alguns destes testes contêm pequenos exemplos de *dataraces* que deveriam ser detectados pela nossa ferramenta e que, por isso, foram utilizados para validar a precisão da mesma. Outros, apesar dos resultados reportados não serem tão interessantes já que, por exemplo, a tradução do programa de *locks* para transacções não mantém a semântica original do mesmo, foram utilizados com o intuito de avaliar o comportamento da nossa ferramenta quando aumentamos o tamanho dos programas de

Testes	Nº de Anomalias	Falsos Negativos MoTH	Falsos Negativos Artho	Falsos Negativos Teix.	Falsos Positivos MoTH	Falsos Positivos Artho	Falsos Positivos Teix.	#AVersions	Linhas de Código	Tempo (seg.)
Connection [BBA08]	2	0	1	1	0	0	1	34	112	49
Coord03 [AHB03]	1	0	0	0	0	0	3	13	170	48
Local [AHB03]	1	0	1	0	0	0	1	3	33	43
NASA [AHB03]	1	0	0	0	0	0	0	7	121	43
Coord04 [AHB04]	1	0	0	0	0	0	3	7	47	45
Buffer [AHB04]	0	0	0	0	1	0	7	8	64	46
DoubleCheck [AHB04]	0	0	0	0	1	0	2	7	51	44
StringBuffer [FF04]	1	0	1	1	0	0	0	12	52	49
Account [vG03]	1	0	1	0	0	0	0	3	65	46
Jigsaw [vG03]	1	0	0	0	0	0	1	33	145	49
OverReporting [vG03]	0	0	0	0	0	0	2	6	52	48
UnderReporting [vG03]	1	0	1	0	0	0	0	3	31	44
Allocate Vector [IBM]	1	0	1	0	0	0	1	24	304	48
Knight [TLS10]	1	0	1	0	0	0	2	10	223	46
Arithmetic Database [TLS10]	3	0	3	1	1	0	0	24	416	57
Elevator [vG03]	16	0	16	-	6	4	-	39	558	50
Philo [vG03]	0	0	0	-	2	0	-	9/594	96	54/560
Tsp [vG03]	0	0	0	-	2	0	-	635	795	786
Store	2	0	1	-	0	1	-	44/608	901	168/1592
Vector Fail	1	1	1	-	0	0	-	10	108	42
Total	**15**	**0**	**10**	**3**	**3**	**0**	**23**	-	-	-

Tabela 4.2: Resultados dos testes

entrada.

Tivemos acesso aos resultados obtidos pela ferramenta de Teixeira [TLS10] na análise de alguns destes testes, e implementámos o algoritmo de *view consistency* [AHB03] na ferramenta MoTH, usando análise estática ao contrário da abordagem dinâmica dos autores. Estes trabalhos estão indubitavelmente relacionados com o nosso e, por isso, em cada teste, apresentamos igualmente os resultados reportados por cada uma delas, permitindo uma percepção mais intuitiva da relevância e da precisão introduzida pela nossa abordagem.

Os resultados da validação descrita estão sistematizados na Tabela 4.2. A primeira coluna, Nº de Anomalias, contém o número de *dataraces* documentados em cada um dos testes. As colunas Falsos Negativos e Falsos Positivos identificam respectivamente o número de falsos negativos e positivos reportados na ferramenta MoTH, na implementação do conceito de *view consistency* de Artho, e na ferramenta de Teixeira [TLS10]. Depois, na coluna #AVersions, é apresentado o número de elementos do conjunto AVersions, i.e., o número de variáveis globais acedidas dentro do escopo transaccional. Este número será importante para comparar o tamanho dos testes apresentados permitindo-nos avaliar a escalabilidade da nossa abordagem. Finalmente, é apresentado o número de linhas de código de cada teste bem como o tempo que a ferramenta MoTH demorou na sua análise, medido em segundos.

Foi inviável obter os resultados da ferramenta de Teixeira em alguns dos testes utilizados nesta validação, pelo que estes foram separados dos restantes e realçados na tabela

através de um sombreado. De forma a poder comparar as três ferramentas, os somatórios apresentados na última linha da tabela dizem respeito apenas aos testes não sombreados.

Por outro lado, nos testes Philo e Store, podemos encontrar acessos às bibliotecas IO do Java dentro do escopo transaccional, que permitem disponibilizar informação ao utilizador através da chamada do método println da classe System. Como não nos propomos a tratar de acessos a este tipo de bibliotecas, mas o tamanho de ambos os testes depende significativamente desses acessos, na tabela foi apresentado o tempo e o tamanho dos programas, com e sem o uso destas chamadas.

Apesar de, segundo a classificação de *soundness* e *completeness* de Flanagat et al. descrita em [FLL+02], a nossa abordagem ser *unsound* e *incomplete*, os 15 *dataraces* encontrados na literatura foram detectados pela ferramenta MoTH. Para além disso, apenas 3 falsos positivos foram assinalados nestes testes, todos com origem no Dependency Sensor, tendo sido gerados mais 10 falsos positivos nos restantes testes.

As razões que levaram à geração dos primeiros três falsos positivos são distintas. Para apurar a razão do primeiro falso positivo gerado no teste Buffer, descrito na Secção 6.6, seria precisa uma análise mais cuidadosa, já que os autores justificam a ausência de *dataraces* através de uma assunção que não foi implementada. O segundo falso positivo foi gerado no teste Double Check (Secção 6.7) e deve-se à imprecisão do algoritmo do Dependency Sensor. O algoritmo teria de ser refinado de forma a excluir este tipo de cenários. Finalmente, o falso positivo reportado no teste Arithmetic Database (Secção 6.15) deveu-se à criação de uma falsa dependência. Esta dependência não teria sido criada se a ferramenta MoTH conseguisse distinguir duas instâncias da mesma classe. A implementação de uma análise complementar *points-to* permitir-nos-ia excluir este tipo de falsos positivos.

Os restantes falsos positivos deveram-se à criação de falsas dependências, a maioria delas relacionadas com a impossibilidade de distinguir duas instâncias da mesma classe.

Note-se que, a custo de um ligeiro acréscimo no número de falsos positivos, a ferramenta MoTH detecta vários *dataraces* gerados por acessos a uma única variável que escaparam à abordagem de Artho e outros que escaparam à ferramenta de Teixeira. Para além disso, conseguimos reduzir significativamente o número de falsos positivos de Teixeira, obtendo apenas cerca de 13% dos falsos positivos gerados pela sua abordagem.

Foram também usados programas maiores e mais complexos que acedem atomicamente a um número de variáveis globais substancialmente maior, tornando a sua análise mais demorada. No entanto, os tempos obtidos são razoáveis tendo em conta que, ao fazer a análise estática destes programas, estamos a analisar todos os cenários possíveis em vez de analisarmos uma única execução específica.

Contudo, a ferramenta pode ainda ser melhorada de forma a reduzir consideravelmente o seu tempo de execução. Em primeiro lugar, a ferramenta MoTH analisa constantemente o código JRE utilizado pelo programa computando os acessos e as dependências entre variáveis do mesmo. Como se pode verificar na diferença dos tamanhos dos testes Philo e Store, as chamadas das funções do JRE podem ser bastante pesadas quando

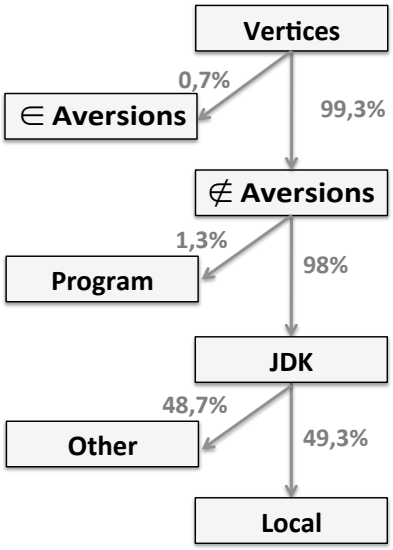


Figura 4.2: Distribuição média dos tipos de versões de variáveis do grafo de dependências

comparadas com o tamanho do resto do programa. Sendo o código do JRE estável, o resultado da sua análise poderia ser guardado em disco de modo a ser utilizado em futuras análises.

Para além desta simples cache, e sendo a parte computacionalmente mais exigente do nosso algoritmo as pesquisas de caminhos no grafo de dependências do programa, poderíamos simplificar o grafo de dependências do JRE, melhorando consequentemente a eficiência da análise. Na Figura 4.2 é apresentada a distribuição média do tipo das variáveis dos grafos de dependências obtidos.

Nas pesquisas de grafos referidas, estamos apenas interessados em caminhos entre as versões de variáveis especiais (que representam os parâmetros ou o retorno dos métodos) ou de variáveis do conjunto AVersions. Assim, podemos excluir do grafo guardado em cache todos os nós que representam versões de variáveis locais do JRE, i.e., em média 49,3% das variáveis totais, ligando os seus predecessores e sucessores no grafo de dependências. Estudos preliminares baseados na contagem dos arcos que envolvem variáveis locais do JRE permitem-nos estimar que a remoção deste tipo de variáveis reduziria para cerca de 51% o número de arcos do grafo de dependências.

Finalmente, os algoritmos baseados na informação computada na primeira fase da análise podem ser paralelizados. Por um lado, os dois sensores são completamente independentes e, por isso, podem executar os respectivos algoritmos concorrentemente. Por outro lado, no *plugin* Dependency Sensor, todas as pesquisas de caminhos no grafo de dependências são independentes entre si, não modificando o grafo lido. Assim, esta estrutura de dados podia ser dividida por diversos processos que computariam um subconjunto das pesquisas necessárias. Esta paralelização permitir-nos-ia reduzir consideravelmente o tempo de execução da ferramenta MoTH.

5

Conclusão

5.1 Conclusões

A Memória Transaccional é uma abordagem recente à programação concorrente. Apesar de ser menos propensa a erros, é possível encontrar erros de concorrência (*dataraces*) em programas com a semântica transaccional, mesmo em ambientes de atomicidade forte.

Não obstante de ser uma abordagem promissora tentando conjugar a performance dos sistemas que utilizam *fine-grained locking* com a simplicidade da programação *coarse-grained locking*, a Memória Transaccional ainda não atingiu a maturidade desejada.

Existe ainda uma grande carência de ferramentas de detecção de *dataraces* em programas MT. Por outro lado, as ferramentas descritas na literatura aplicadas a programas com *locks* não abrangem todo o tipo de anomalias ou reportam um número muito elevado de falsos positivos tornando inviável a sua utilização prática.

Tal como foi referido na Secção 1.3, o objectivo deste trabalho é demonstrar que a análise estática é uma abordagem viável na detecção de *dataraces* em programas de Memória Transaccional.

Com esse objectivo, foi desenvolvida uma ferramenta de análise estática de programas MT escritos em Java ByteCode. A estrutura desta ferramenta é genérica e extensível, através da adição de *plugins*, chamados Sensores, desenhados para detectar *dataraces* específicos.

Foram criadas diversas análises que permitem extrair do programa toda a informação necessária na detecção de *dataraces*.

Como os *dataraces* correspondem a interacções anómalas entre diferentes fios de execução, foi criada a Análise de Processos que apura os diferentes fios de execução (processos) que podem executar num determinado programa.

Devido à impossibilidade em analisar determinados métodos como, por exemplo, aqueles que são chamados em objectos de tipo interface, foi implementada a Análise de Tipos de Instâncias que permite apurar as possíveis classes de implementação deste tipo de objectos.

Foi também desenhada e implementada a Análise de *Views*, através da extensão do algoritmo de *view consistency* [AHB03], que permite apurar os acessos a variáveis globais feitos dentro do escopo transaccional. Apesar dos autores terem uma abordagem interessante, analisando as relações entre variáveis em vez de interacções entre transacções, o facto de não distinguirem acessos de leitura e escrita faz com que sejam gerados muitos falsos positivos.

Devido à necessidade de apurar as dependências entre as variáveis globais do programa, foi criada uma Análise de Dependências que cria um grafo com as dependências de dados e de controlo do programa de entrada.

Finalmente, foram criados dois Sensores que detectam a maioria dos *dataraces* presentes na literatura. O primeiro, ViewConsistency Sensor, faz a detecção de *high-level dataraces*, i.e., anomalias relacionadas com acessos parciais a conjuntos de variáveis que deveriam ser acedidos atomicamente. Para tal, foi estendido o conceito de *view consistency* [AHB03] distinguindo acessos de leitura e de escrita. O segundo Sensor, Dependency Sensor, faz a detecção de *stale-value errors* através da análise das dependências entre as variáveis do programa, apurando se uma determinada variável saiu do escopo de uma transacção para outra.

A ferramenta foi validada com um conjunto de testes retirados da literatura, que contêm *dataraces* documentados e que foram já utilizados para validar trabalhos relacionados, ou desenhados e desenvolvidos por nós com este propósito. Os resultados obtidos mostram a precisão da nossa abordagem, corroborando com a nossa hipótese.

Assim, podemos afirmar que, de facto, a análise estática é uma abordagem viável e apropriada na detecção de *dataraces* em programas de Memória Transaccional.

Além disto, o facto de termos analisado directamente o código binário permitiu-nos obter informação à qual não teríamos acesso analisando o código fonte do programa. Ao comparar os resultados da nossa abordagem com a de Teixeira [TLS10], constatamos que muitos dos falsos positivos gerados pela sua ferramenta devem-se à impossibilidade de analisar o código (binário) do JRE ou a leituras redundantes, onde é feita uma primeira leitura do objecto e uma segunda do atributo desse objecto. Ao analisarmos o código binário estes cenários não se aplicam e, por isso, não foram reportados os referidos falsos positivos. Por outro lado, sendo uma linguagem com menos casos e ambiguidades que a linguagem Java, acreditamos que a análise do Java ByteCode é significativamente mais simples.

5.2 Trabalho Futuro

O trabalho apresentado ao longo deste documento pode ser melhorado através quer do refinamento da informação extraída na análise do programa de entrada, quer da redução do tempo de computação dos Sensores implementados. Estas melhorias foram referidas ao longo do documento e são sintetizadas nesta secção.

Anotações de Dependências

Nas análises de dependências (de dados e de controlo), sempre que nos deparamos com um método nativo, criamos uma variável especial que depende do valor de todos os parâmetros do método e que é usada no retorno do mesmo. Tal como na Análise das *Views*, poderia ser criado um mecanismo de anotações para as dependências, que permitisse ao utilizador expressar intuitivamente as dependências de um determinado método, refinando a informação gerada na análise do programa de entrada.

Paralelização de tarefas

Com o intuito de reduzir o tempo de execução da nossa análise, algumas tarefas independentes podiam ser paralelizadas aproveitando os recursos disponíveis. Em primeiro lugar, tendo a estrutura da ferramenta uma primeira fase de computação de informação e uma segunda onde cada sensor recebe essa informação e executa um algoritmo independente, os algoritmos dos Sensores podiam ser executados concorrentemente. Por outro lado, no *plugin* Dependency Sensor, as pesquisas de caminhos no grafo de dependências são completamente independentes entre si e não alteram o estado do mesmo. Assim, estas pesquisas, que consistem na tarefa computacionalmente mais exigente, podiam ser executadas concorrentemente de forma a reduzir substancialmente o tempo de execução da nossa ferramenta.

Por outro lado, este tipo de algoritmos de pesquisas em grafos são usados como exemplos de problemas onde o uso de GPUs (Graphics Processing Unit) pode reduzir significativamente o seu tempo de execução. Por esta razão, o uso de GPUs poderia aumentar consideravelmente a eficiência da ferramenta MoTH.

Mecanismo de Cache

Actualmente, cada vez que um programa é analisado, a ferramenta MoTH percorre todos os métodos invocados no mesmo de forma a computar a informação relacionada com os acessos e dependências entre variáveis. No entanto, para zonas de código estável, como por exemplo no código do JRE, poderia ser implementado um mecanismo de cache da informação computada fazendo com que, numa segunda análise, essa informação fosse obtida directamente de disco em vez de ser computada novamente, reduzindo consideravelmente o tempo de análise.

Prunning de Dependências

Tal como referido anteriormente, o algoritmo computacionalmente mais exigente da análise descrita neste documento consiste em pesquisar os caminhos entre as variáveis do conjunto AVersions, no grafo de dependências do programa. Nesse algoritmo, para cada variável do conjunto, é executado um algoritmo de inundação no grafo de forma a obter todos os vértices do mesmo que são atingíveis por essa variável. Assim, a complexidade deste algoritmo depende fortemente no tamanho do grafo, mais especificamente no número de arcos do mesmo.

Se fosse implementado um mecanismo que permitisse fazer cache do grafo de dependências do JRE, esse grafo podia ser simplificado (*prunned*) removendo, por exemplo, as variáveis locais. No algoritmo do *plugin* Dependency Sensor apenas as variáveis do conjunto AVersions e as variáveis que representam os parâmetros e retorno dos métodos são importantes. Assim, as variáveis locais podiam ser removidas ligando-se os seus predecessores no grafo com os seus sucessores de forma a manter a transitividade das dependências. Os algoritmos de *prunning* de grafos são significativamente pesados computacionalmente, mas só teriam de ser executados uma vez reduzindo o tamanho do grafo de dependências das variáveis do JRE guardado em disco. Esta medida permitia diminuir consideravelmente o tempo de execução das pesquisas em grafos.

Finalmente, estudos preliminares baseados na contagem dos arcos que envolvem variáveis locais do JRE permitem-nos estimar que a remoção deste tipo de variáveis resultaria na redução de cerca de 52% do número de arcos do grafo de dependências.

Implementação de uma Análise *points-to*

Tal como foi referido ao longo do documento, assumimos que quaisquer dois acessos a objectos da mesma classe são feitos sobre a mesma instância. Esta assunção pode gerar alguns falsos positivos, tal como aconteceu no teste Arithmetic Database. A implementação de uma análise *points-to* complementar permitir-nos-ia distinguir duas instâncias da mesma classe excluindo este tipo de cenários e aumentando, consequentemente, a precisão da nossa ferramenta.

Implementação de uma Análise *May-Happens-In-Parallel*

A implementação de uma análise *May-Happens-In-Parallel* complementar também poderia aumentar a precisão da ferramenta MoTH. Actualmente, estamos a assumir que qualquer transacção de um processo pode ocorrer entre duas transacções de outro processo. Assim, podem ser reportados conflitos entre processos que são executados sequencialmente e que, por isso, nunca poderiam influenciar a execução um do outro.

Combinação com Análise Dinâmica

A análise estática deste trabalho poderia ser complementada com análise dinâmica, permitindo verificar se os conflitos apontados pela ferramenta correspondem, de facto, a anomalias reais em tempo de execução. Esta combinação poderia reduzir o número de falsos positivos reportados pela ferramenta.

Extensão do modelo para suportar múltiplos *locks*

Os algoritmos e o protótipo apresentados nesta dissertação permitem a detecção de *dataraces* em programas com a semântica transaccional sendo que, conceptualmente, uma transacção pode ser vista como um bloco sincronizado num *lock* global. Este modelo podia ser estendido de forma a suportar múltiplos *locks*, tornando a ferramenta mais abrangente.

Bibliografia

[AHB03] Cyrille Artho, Klaus Havelund, e Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, Dezembro 2003.

[AHB04] Cyrille Artho, Klaus Havelund, e Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. *Automated Technology for Verification and Analysis*, pág. 150–164, 2004.

[All70] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, Julho 1970.

[ASM] http://asm.ow2.org/.

[BBA08] Nels E. Beckman, Kevin Bierhoff, e Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, 2008.

[BCE] http://jakarta.apache.org/bcel/.

[BL04] Michael Burrows e K. Rustan M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, Outubro 2004.

[BLM05] Colin Blundell, E Christopher Lewis, e Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[BLM06] Colin Blundell, E Christopher Lewis, e Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5:17–, July 2006.

[Chr06] Ciera N. Christopher. Evaluating Static Analysis Frameworks. *Analysis*, pág. 1–17, 2006.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, e Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *ACM SIGPLAN Notices*, 37(5):258, Maio 2002.

[Dij68] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pág. 43–112. Academic Press, 1968.

[DS91] Evelyn Duesterwald e Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pág. 36–48, New York, NY, USA, 1991. ACM.

[EQ07] Tayfun Elmas e S Qadeer. Goldilocks: a race and transaction-aware java runtime. *ACM SIGPLAN Notices*, pág. 245–255, 2007.

[FF04] Cormac Flanagan e Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pág. 256–267, New York, NY, USA, 2004. ACM.

[FFY08] Cormac Flanagan, Stephen N. Freund, e Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. 43(6):293–303, Junho 2008.

[Fin] http://findbugs.sourceforge.net/.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, e Raymie Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234, Maio 2002.

[FQ03] Cormac Flanagan e Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pág. 1–12, New York, NY, USA, 2003. ACM.

[GDDC97] D. Grove, G. DeFouw, J. Dean, e G. Chambers. Call graph construction in object-oriented languages. In *Proc. OOPSLA'97 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 32, number 10. ACM Press, Sigplan Notices, Oct 1997.

[Gue10] Rachid Guerraoui. *Principles of Transactional Memory*. 2010.

[HLR10] Tim Harris, James R. Larus, e Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[IBM] IBM's Concurrency Testing Repository.

[Jav] http://www.csg.is.titech.ac.jp/ chiba/javassist/.

[KSK09] Uday Khedker, Amitabha Sanyal, e Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 2009.

[LY99] Tim Lindholm e Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, April 1999.

[OC03] Robert O'Callahan e Jong-Deok Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167, Outubro 2003.

[PDL+11] Vasco Pessanha, Ricardo Dias, João Lourenço, Eitan Farchi, e Diogo Sousa. Practical verification of high-level dataraces in transactional memory programs. In *Proceedings of the 9th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '11, pág. 26–34, New York, NY, USA, 2011. ACM.

[RAF] N. Rutar, C.B. Almazan, e J.S. Foster. A Comparison of Bug Finding Tools for Java. *15th International Symposium on Software Reliability Engineering*, pág. 245–256.

[Rep] https://projectos.fct.unl.pt/projects/di-moth/.

[RHW10] Christopher J. Rossbach, Owen S. Hofmann, e Emmett Witchel. Is transactional programming actually easier? *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, pág. 47, 2010.

[SATH+06] Bratin Saha, A.R. Adl-Tabatabai, R.L. Hudson, C.C. Minh, e Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, March*, pág. 29–31, 2006.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, e Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Novembro 1997.

[Soo] http://www.sable.mcgill.ca/soot/.

[TLS10] Bruno Teixeira, João Lourenço, e Diogo Sousa. A Static Approach for Detecting Concurrency Anomalies in Transactional Memory. *Memory*, 2010.

[vG03] Christoph von Praun e Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology*, pág. 2004, 2003.

[VTD06] Mandana Vaziri, Frank Tip, e Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *ACM SIGPLAN Notices*, 41(1):334–345, Janeiro 2006.

[WJKT05] Stefan Wagner, J. Jürjens, Claudia Koller, e Peter Trischberger. Comparing bug finding tools with reviews and tests. *Testing of Communicating Systems*, pág. 40–55, 2005.

[WS03] L. Wang e S. Stoller. Run-Time Analysis for Atomicity. *Electronic Notes in Theoretical Computer Science*, 89(2):191–209, Outubro 2003.

6

Apêndice

Neste capítulo será feita uma descrição de todos os testes utilizados na validação do nosso trabalho. A tabela 6.1 contém uma caracterização dos programas de entrada sendo apresentado, para cada um destes, o número de linhas de código, o número de blocos atómicos, o número de fios de execução distintos (gerados por diferentes classes) e, finalmente, a cardinalidade do conjunto AVersions.

Nas próximas secções, para cada teste, começamos por fazer uma pequena descrição do programa respectivo, sendo disponibilizado parte do código de forma a ilustrar a sua semântica. Em seguida são listados os *dataraces* presentes no teste, que deveriam ser detectados pela nossa ferramenta. Finalmente, são apresentados os conflitos reportados pela nossa ferramenta, com uma breve discussão e análise dos resultados obtidos.

6.1 Teste: Connection

Descrição

Este teste foi adaptado de [BBA08] e simula uma aplicação de *chat* em rede. Nesta implementação, uma classe GUI (Graphical User Interface) utiliza os serviços da ligação de rede, representada aqui por um objecto Java, enviando mensagens e desconectando a ligação aleatoriamente. Para além deste objecto, outro objecto representa um contador que contabiliza o número de mensagens enviadas para a rede. A maior parte do código é disponibilizado na Listagem 6.1.

Nome	LOC	#Atomics	Nº fios de execução	#AVersions
Connection [BBA08]	112	4	2	34
Coord03 [AHB03]	170	6	5	13
Local [AHB03]	33	2	2	3
NASA [AHB03]	121	3	3	7
Coord04 [AHB04]	47	3	2	7
Buffer [AHB04]	64	2	2	8
DoubleCheck [AHB04]	51	2	2	7
StringBuffer [FF04]	52	3	2	12
Account [vG03]	65	2	2	3
Jigsaw [vG03]	145	3	2	33
OverReporting [vG03]	52	2	2	6
UnderReporting [vG03]	31	1	2	3
Allocate Vector [IBM]	304	3	2	24
Knight [TLS10]	223	2	2	10
Arithmetic Database [TLS10]	416	5	2	24
Elevator [vG03]	558	6	2	39
Philo [vG03]	96	2	2	9/594
Tsp [vG03]	795	2	2	635
Store	901	5	4	44/608
Vector Fail	108	3	2	10

Tabela 6.1: Caracterização dos programas de entrada

Anomalias

1. O método disconnect reinicia o contador e fecha a ligação de rede. No entanto, como estas operações são feitas em dois métodos atómicos distintos, outro processo pode ver um estado inconsistente entre as duas operações, onde a ligação se encontra fechada mas o contador expressa um número positivo de mensagens enviadas.

2. O método trySendMessage testa se a ligação está fechada no método atómico isConnected e, se não estiver, envia uma mensagem para a mesma no método send. No entanto, outro processo pode ter fechado a ligação antes da mensagem ser enviada.

Resultados

Tal como esperado, os dois *dataraces* foram correctamente detectados. Enquanto o primeiro *datarace* foi detectado pelo View Consistency Sensor, no segundo foi criada uma dependência entre os métodos isConnected e send, sendo posteriormente detectado o conflito pelo Dependency Sensor.

Resumo

Zero falsos negativos e zero falsos positivos.

Listagem 6.1: Teste: Connection (Réplica da Listagem 4.1)

```
1  //Classe Connection
2  final Counter counter;
3  java.net.Socket socket;
4  @Atomic
5  boolean isConnected(){
6    return !socket.isClosed();
7  }
8  @Atomic
9  void send(String msg){
10   socket.write(msg);
11   counter.increment();
12 }
13 @Atomic
14 void closeSocket(){
15   socket.close();
16 }
17 void disconnect(){
18   closeSocket();
19   counter.reset(); //Atomic
20 }
21
22 //Classe GUI
23 void trySendMsg(String msg){
24   if(connection.isConnected()){
25     connection.send(msg);
26   }
27 }
```

6.2 Teste: Coordinates'03

Descrição

Este exemplo foi adaptado de [AHB03] e ilustra os acessos concorrentes de um conjunto de processos a um par de coordenadas. Cada acesso pode ser feito ao objecto como um todo, ou ser uma leitura/escrita parcial de uma das variáveis do par. A Figura 6.1 ilustra o código do programa.

```
// Thread t1
d1 =  new Coord(1,2)
//Atomic
c.setXY(d1);
```

```
// Thread t2
x2 = c.getX();
use(x2);
```

```
// Thread t3
//Atomic
x3 = c.getX();
//Atomic
y3 = c.getY();
use(x3,y3);
```

```
// Thread t4
//Atomic
x4 = c.getX();
use(x4);
//Atomic
d4 = c.getXY();
x4 = d4.getX();
y4 = d4.getY();
use(x4,y4);
```

Figura 6.1: Teste: Coordinates'03 (fragmento de código adaptado de [AHB03])

Anomalias

Segundo os autores, apesar do Thread 4 também executar mais do que uma transacção, apenas o Thread 3 é potencialmente anómalo. Isto deve-se ao facto deste processo ler o valor das duas variáveis do par em métodos atómicos separados sendo que, entre as duas leituras, outro processo pode ter alterado o estado do par resultando numa leitura inconsistente por parte do Thread 3.

Resultados

O *datarace* descrito foi correctamente detectado pelo View Consistency Sensor, visto que as *views* de leitura do Thread 3, que estão contidas na *maximal view* de escrita do método setXY, não formam uma cadeia, e o método use(x3,y3) recebe ambos os valores lidos.

Resumo

Zero falsos negativos e zero falsos positivos.

6.3 Teste: Local

Descrição

Tal como o anterior, este teste foi retirado de [AHB03] onde é usado como um exemplo de um *datarace* que não é detectado pela abordagem dos autores. Este exemplo consiste num programa simples que contém diversos processos que incrementam uma variável partilhada. No entanto, para o fazer, os processos criam uma cópia dessa variável, incrementam o valor copiado de uma forma não atómica e, finalmente, actualizam finalmente o seu valor. Na Listagem 6.2 é apresentada a maior parte do código deste programa.

Listagem 6.2: Teste: Local (fragmento de código adaptado de [AHB03])

```
1  //Classe Main
2  public void run(){
3    int tmp;
4    tmp = cell.getValue(); // Atomic
5    tmp++;
6    cell.setValue(tmp); // Atomic
7  }
8
9  //Classe Cell
10 @Atomic
11 int getValue() {
12   return n;
13 }
14 @Atomic
15 void setValue(int tmp) {
16   n = tmp;
17 }
```

Anomalias

Este programa pode gerar um possível *stale-value error* já que, entre a leitura e a escrita do objecto da classe Cell, outro processo pode ter alterado o valor da variável global fazendo com que essa actualização fosse perdida na execução do método setValue.

Resultados

Tal como esperado, o *datarace* foi correctamente detectado pelo Dependency Sensor, através da dependência criada entre os métodos atómicos getValue e setValue.

Resumo

Zero falsos negativos e zero falsos positivos.

6.4 Teste: NASA

Descrição

Este teste foi retirado de [AHB03] e consiste num problema detectado num agente remoto de um controlador de naves espaciais desenvolvido pela NASA. Neste programa, um conjunto de processos (Tasks) tentam executar adquirindo primeiro um conjunto de propriedades guardadas numa tabela, que têm de ser mantidas ao longo da sua execução. Para adquirir uma propriedade, uma Task começa por alterar o seu valor para o valor observado e, finalmente, coloca o valor da flag *achieved* a verdadeiro afirmando que a propriedade se verifica. Por outro lado, um objecto *Deamon* monitoriza o estado do sistema verificando se este é consistente com as propriedades adquiridas na tabela. Na Listagem 6.3 são ilustradas as regiões problemáticas do programa.

Anomalias

O *datarace* presente neste teste está relacionado com o facto de cada propriedade ser adquirida em mais do que um método atómico.

Considere-se que uma Task específica adquire uma propriedade e está prestes a executar o segundo método atómico setAchieved. Considere-se ainda que, entretanto, essa propriedade é destruída devido a um evento exterior. Agora, o objecto Deamon irá constatar que a propriedade não se verifica e que a variável *achieved* tem o valor **false**, i.e., não existe qualquer tipo de anomalia. No entanto, quando a Task voltar a executar colocando o valor da variável *achieved* a **true**, a propriedade foi marcada como adquirida ainda que não se verifique no sistema real. O Deamon não detectou uma potencial anomalia.

Listagem 6.3: Teste: NASA (fragmento de código adaptado de [AHB03])

```
1  //Classe Task
2  @Atomic
3  private void setValue(Object v,int N){
4    table[N].value = v;
5  }
6  @Atomic
7  private void setAchieved(Object v, int N){
8    table[N].achieved = true;
9  }
10
11 //Classe Deamon
12 public void run() {
13   ...
14   while (true) {
15     tryIssueWarning(N);
16   }
17   @Atomic
18   private void tryIssueWarning(int N){
19     if (table[N].achieved && system_state[N] != table[N].value)
20       issueWarning();
21   }
22 }
```

Resultados

A anomalia presente neste programa consiste num *high-level datarace* comum onde um conjunto de variáveis partilhadas, formado pelas variáveis *value* e *achieved* da tabela das propriedades, é acedido através de duas ou mais transacções. Assim, o *datarace* foi detectado pelo View Consistency Sensor, já que as *views* de escrita da classe Task, geradas pelos métodos setValue e setAchieved, estão contidas na *maximal view* de leitura da classe Deamon e não formam uma cadeia.

Resumo

Zero falsos negativos e zero falsos positivos.

6.5 Teste: Coordinates'04

Descrição

Este teste, discutido já no escopo deste documento, foi apresentado em [AHB04] como um exemplo simples de um *high-level datarace*. O programa descreve um par de coordenadas cujos valores podem ser reiniciados através do método reset ou trocados com o método atómico swap. A implementação destes dois métodos é ilustrada na Listagem 6.4.

Listagem 6.4: Teste: Cooordinates'04 (fragmento de código adaptado de [AHB04])

```
1  @Atomic
2  public void swap(){
3    int oldX = coord.x;
4    coord.x = coord.y;
5    coord.y = oldX;
6  }
7  public void reset(){
8    resetX();
9  // inconsistent state (0, y)
10   resetY();
11 }
12 @Atomic
13 public void resetX(){
14   coord.x = 0;
15 }
16 @Atomic
17 public void resetY(){
18   coord.y = 0;
19 }
```

Anomalias

Uma vez mais, a ocorrência do *datarace* do programa está relacionada com o facto de operações que deveriam ser executadas atomicamente poderem ser intercaladas com operações conflituosas.

Sendo a operação reset constituída por dois blocos atómicos, é possível que outros processos observem o estado intermédio $\langle 0, y \rangle$ entre estes que, por não corresponder a nenhuma versão completa do par, é inconsistente. Por exemplo, se a operação swap fosse executada por outro processo entre estes dois métodos, teríamos um estado final $\langle y, 0 \rangle$ que corresponde a um estado que não é atingível com uma execução sequencial das operações reset e swap.

Resultados

O *high-level datarace* descrito anteriormente foi correctamente detectado pelo View Consistency Sensor, já que as *views* de escrita geradas pelos métodos resetX e resetY estão contidas na *maximal view* de leitura gerada pelo método swap, e não formam uma cadeia.

Resumo

Zero falsos negativos e zero falsos positivos.

6.6 Teste: Buffer

Descrição

Este teste foi igualmente retirado de [AHB04] e consiste num buffer partilhado por diversos processos que removem e adicionam elementos ao mesmo. Na Listagem 6.5 é ilustrado um fragmento do código deste teste.

Listagem 6.5: Teste: Buffer (fragmento de código adaptado de [AHB04])

```
1  public void run(){
2    int value, fdata;
3    while(true){
4      value = next();
5      fdata = f(value); //long computation
6      add(fdata);
7    }
8  }
9  @Atomic
10 private int next(){
11   return buffer.next();
12 }
13 @Atomic
14 private void add(int fdata){
15   buffer.add(fdata);
16 }
```

Anomalias

Segundo os autores, este teste não contém qualquer tipo de anomalias. Sendo que ambos os métodos atómicos acedem ao buffer, parece haver um *datarace*. Contudo, os autores afirmam que, assumindo que nenhum elemento é inserido mais do que uma vez, a semântica do programa mantém os dados localmente ao processo em questão. A implementação desta assunção não parece ser trivial, não sendo também claro se essa mesma implementação removeria a anomalia. Teria de ser feito um estudo mais profundo da semântica do programa, complementando-o com um cenário real que demonstrasse o funcionamento do programa.

Resultados

Foi gerado um conflito por parte do Dependency Sensor devido a uma dependência gerada entre os métodos next e add, já que o valor usado para introduzir um elemento parece depender do valor retornado pelo método next. Apesar da ausência de *dataraces* ter sido justificada com uma assunção que não foi implementada e que não parece clara, consideraremos este conflito como um falso positivo.

Resumo

Um falso positivo.

6.7 Teste: Double Check

Descrição

Este teste foi apresentado em [AHB04] como mais um programa isento de *dataraces* que pode gerar falsos positivos. O código deste exemplo é disponibilizado na Listagem 6.6.

De forma a diminuir o tamanho das transacções, diminuindo assim a probabilidade de conflito entre as mesmas, o incremento de uma variável é feito em duas operações atómicas. Primeiro, é feita uma cópia do valor da variável, em seguida essa cópia é incrementada e, finalmente, o valor da variável é actualizado com o valor da cópia. No entanto, antes de actualizar o valor da variável, o processo testa atomicamente se o valor lido inicialmente ainda se mantém, descartando a actualização caso o valor tenha sido alterado.

Listagem 6.6: Teste: Double Check (fragmento de código adaptado de [AHB04])

```
1  public void run() {
2    int value, fdata;
3    boolean done = false;
4    while (!done) {
5      value = getSharedField();
6      fdata = f(value); // long computation
7      done = updateSharedField(fdata);
8    }
9  }
10 @Atomic
11 private int getSharedField(){
12   return shared.field;
13 }
14 @Atomic
15 private boolean updateSharedField(int value, int fdata){
16   if (value == shared.field) { //Double check...
17     shared.field = fdata;
18     return true;
19   }
20   return false;
21 }
```

Anomalias

Apesar de parecer existir um *stale-value error* gerado pela leitura e escrita da variável global, como a actualização verifica se o valor foi alterado, este programa não tem qualquer tipo de *datarace*.

Resultados

Foi detectada uma dependência entre os métodos atómicos getSharedField e updateSharedField tendo sido, por isso, reportado um conflito (falso positivo). A definição de *data-race* do Dependency Sensor, que utiliza o conceito de dependências entre métodos atómicos, teria de ser melhorada de forma a excluir este cenário.

Resumo

Um falso positivo.

6.8 Teste: String Buffer

Descrição

O exemplo descrito nesta secção foi retirado de [FF04] e simula o funcionamento da classe java.lang.StringBuffer. Este teste pretende apontar uma anomalia detectada pelos autores nesta classe do JRE. A Listagem 6.7 ilustra o fragmento de código onde ocorre o conflito. Neste programa, o método append foi redefinido sendo que os métodos length e getChars são delegados no objecto da classe do JRE.

Listagem 6.7: Teste: String Buffer (fragmento de código adaptado de [FF04])

```
1  public MystringBuffer append(MyStringBuffer other){
2
3    int len = other.length(); // Atomic
4
5    // ...other threads may change other.length(),
6    // ...so len does not reflect the length of "other" anymore
7
8    char[] value = new char[len];
9    other.getChars(0, len, value, 0); // Atomic
10
11   //...
12 }
```

Anomalias

Existe uma ocorrência de um *stale-value error* neste programa. O tamanho do buffer *other* é copiado para a variável *len* na linha 3. Quando esta cópia é usada, na linha 8, o tamanho do buffer *other* poderia ter sido alterado resultando na criação de um buffer com um tamanho errado.

Resultados

Foi correctamente detectada uma dependência entre os métodos atómicos length e getChars pelo Dependency Sensor, tendo sido reportada a anomalia. Note-se que, apesar

do método getChars ser implementado através da chamada do método nativo arraycopy da classe java.lang.System do JRE, a nossa ferramenta conseguiu detectar a dependência entre os métodos atómicos.

Resumo

Zero falsos negativos e zero falsos positivos.

6.9 Teste: Account

Descrição

Este exemplo consiste num programa simples que simula diversos depósitos concorrentes numa conta bancária, e foi adaptado de [vG03]. Na listagem 6.8 é ilustrada a classe Account, onde um depósito é constituído por dois métodos atómicos. Tal como no teste Local descrito na Secção 6.3, aqui o saldo da conta bancária é lido num método atómico, incrementado fora do escopo transaccional, e actualizado novamente.

Listagem 6.8: Teste: Account (fragmento de código adaptado de [vG03])

```
1  //Classe Account
2
3  int balance;
4
5  @Atomic
6  int getBalance() {
7    return balance;
8  }
9
10 @Atomic
11 private void setBalance(int value){
12   balance = value;
13 }
14
15 void update (int a) {
16   int tmp = getBalance();
17   tmp = tmp + a;
18   setBalance(tmp);
19 }
```

Anomalias

Existe um *stale-value error* relacionado com o facto da actualização do saldo da conta bancária ser feita através de dois métodos atómicos distintos. Entre estes dois, outro processo a executar o mesmo código poderia ter alterado o valor do saldo fazendo com que o valor da variável *tmp* ficasse obsoleto. Esta situação podia gerar uma actualização perdida.

Resultados

O *datarace* foi correctamente detectado pelo Dependency Sensor, através da dependência criada entre os dois métodos atómicos getBalance e setBalance.

Resumo

Zero falsos negativos e zero falsos positivos.

6.10 Teste: Jigsaw

Descrição

Este teste consiste num exemplo real do Jigsaw CMS que foi adaptado de [vG03], e é ilustrado na Listagem 6.9. Não existe ainda um consenso na semântica associada ao tratamento de excepções dentro do escopo transaccional. Assim, o código deste exemplo foi reescrito de forma a evitar o uso de excepções mantendo, no entanto, a semântica do programa.

O programa é constituído por um conjunto de objectos guardados na estrutura *entries*, e uma variável booleana *closed* que representa a disponibilidade desse mesmo conjunto. O método loadResourceStore retorna um elemento do conjunto se este estiver disponível, ou **null** caso contrário. Para isso, é testada a condição de disponibilidade através do método atómico checkClosed e, caso esta seja verificada, o conjunto *entries* é acedido num segundo método atómico lookupEntry. Finalmente, o método atómico shutdown permite reiniciar o conjunto, colocando o valor da variável *closed* a **true**.

Anomalias

O programa descrito contém um *high-level datarace* já que o método loadResourceStore acede à variável condição e ao conjunto *entries* em métodos atómicos distintos, fazendo com que outro processo pudesse alterar o estado do sistema entre os dois. Considere-se que um processo p_1 executa o método loadResourceStore. Se outro processo p_2 executar o método shutdown exactamente antes do primeiro obter um elemento do conjunto, então p_1 irá tentar aceder a um conjunto vazio gerando, possivelmente, uma excepção NullPointerException.

Resultados

O *datarace* foi correctamente detectado pelo View Consistency Sensor. As *views* de leitura geradas pelos métodos checkClosed e lookupEntry estão contidas na *maximal view* de escrita do método shutdown, não formando uma cadeia. Por outro lado, como o retorno do método loadResourceStore depende de ambas as variáveis lidas nas *views* de leitura, estão verificadas todas as condições necessárias para a detecção do *datarace*.

Listagem 6.9: Teste: Jigsaw (fragmento de código adaptado de [vG03])

```
1  boolean closed = false;
2  Map entries = new HashMap();
3  @Atomic
4  boolean checkClosed() {
5    return closed;
6  }
7  @Atomic
8  Entry lookupEntry(Object key) {
9    return (Entry) entries.get(key);
10 }
11
12 @Atomic
13 void shutdown() {
14   entries.clear();
15   closed = true;
16 }
17
18 ResourceStore loadResourceStore() {
19   if(checkClosed()){
20     return null;
21   }
22   Entry e = lookupEntry(new Object());
23   return e.getStore();
24 }
```

Resumo

Zero falsos negativos e zero falsos positivos.

6.11 Teste: Over-Reporting

Descrição

Este teste foi apresentado em [vG03] como um programa isento de *dataraces* que gera falsos positivos na abordagem dos autores. Um mapa, representado pelas estruturas de dados *keys* e *values*, é partilhado por um conjunto de processos, sendo inicializado uma única vez. Passada a fase de inicialização, os processos fazem diversas leituras obtendo, assim, os elementos do mapa. O código deste programa é ilustrado nas Listagens 6.10 e 6.11.

Anomalias

Não existe qualquer tipo de *datarace* neste teste. Este programa foi disponibilizado em [vG03] como um exemplo de um programa correcto que pode levar à geração de falsos positivos.

Resultados

Não foi reportado nenhum conflito.

Listagem 6.10: Teste: Over-Reporting (fragmento de código adaptado de [vG03]) (parte 1)

```
//Classe Map
Object[] keys, values;
boolean init_done = false;
@Atomic
void init() {
  if (!init_done){
    // ... initialize keys and values
  }
}
@Atomic
Object get(Object key) {
  Object res = null;
  for (int i = 0; i < keys.length; i++) {
    if (key.equals(keys[i])) {
      res = values[i];
      break;
    }
  }
  return res;
}
```

Listagem 6.11: Teste: Over-Reporting (fragmento de código adaptado de [vG03]) (parte 2)

```
//Classe Client
static Map m;

public static void main(String[] args) {
  m = new Map();
  new MapClient().start();
  new MapClient().start();
}

public void run() {
  // lazy initialization
  m.init();
  m.get(new Object());
  // ...
}
```

Resumo

Zero falsos negativos e zero falsos positivos.

6.12 Teste: Under-Reporting

Descrição

Este teste foi retirado de [vG03] e, ao contrário do anterior, consiste num programa cuja anomalia não foi detectada pela abordagem dos autores.

Um contador é partilhado por diversos processos que têm a possibilidade de o incrementar através do método atómico inc. Este método, para além de incrementar o valor do contador, retorna o seu valor antigo. A Listagem 6.12 ilustra a classe que representa o contador, bem como o código de um processo que utiliza este método para duplicar o valor do mesmo.

Anomalias

O antigo valor da variável *i*, retornado na linha 14, é usado como parâmetro da segunda chamada do mesmo método. No entanto, entre as duas chamadas do método, outro processo podia ter alterado o valor do contador fazendo com que a variável temporária não reflectisse o verdadeiro valor de *i*.

Listagem 6.12: Teste: Under-Reporting (fragmento de código adaptado de [vG03])

```
1  //Classe Counter
2  int i;
3
4  @Atomic
5  int inc(int a) {
6    i += a;
7    return i;
8  }
9
10 //Classe Thread
11 static Counter c;
12
13 public void run() {
14   int i = c.inc(0);
15   c.inc(i); // Valor do contador pode ter sido alterado
16 }
```

Resultados

Foi detectada uma dependência entre as duas chamadas do método inc, já que a variável *i* sai do escopo do primeiro para o segundo. Assim, o *datarace* foi correctamente detectado pelo Dependency Sensor.

Resumo

Zero falsos negativos e zero falsos positivos.

6.13 Teste: Allocate Vector

Descrição

O teste descrito nesta secção foi adaptado de [IBM]. Um vector é partilhado por diversos processos, sendo disponibilizados métodos que permitem verificar os índices que se encontram livres e marcar uma determinada posição do vector como livre ou ocupada. Na Listagem 6.13 são ilustrados os métodos principais do programa.

Anomalias

Este teste contem um *stale-value error* no método alloc_block. Este método é constituído por dois métodos atómicos sendo que o primeiro procura o índice de um bloco livre, e o segundo marca-o como ocupado. No entanto, entre estes dois, outro processo podia ter alocado o mesmo bloco, criando uma situação indesejada onde o mesmo bloco foi alocado por dois processos.

Listagem 6.13: Teste: Allocate Vector (fragmento de código adaptado de [IBM])

```
1  private void alloc_block(int i) throws Exception {
2    resultBuf[i] = vector.getFreeBlockIndex();  // Atomic
3    if (resultBuf[i] != -1)
4      vector.markAsAllocatedBlock(resultBuf[i]);  // Atomic
5  }
6
7  public void run() {
8
9    // ...
10   for (int i = 0; i < resultBuf.length; i++)
11     alloc_block(i);
12
13   for (int i = 0; i < resultBuf.length; i++) {
14     if (resultBuf[i] != -1)
15       vector.markAsFreeBlock(resultBuf[i]); // Atomic
16   }
17
18   // ...
19 }
```

Resultados

O *datarace* foi correctamente detectado pelo Dependency Sensor, através da criação de uma dependência entre os blocos atómicos getFreeBlockIndex e markAsAllocatedBlock.

Resumo

Zero falsos negativos e zero falsos positivos.

6.14 Teste: Knight

Descrição

O teste seguinte foi retirado de [TLS10] e pretende apurar o número mínimo (óptimo) de movimentos que um cavalo tem que fazer para capturar outra peça num tabuleiro de xadrez. Isto é conseguido através da exploração de todos os movimentos possíveis por parte dos diferentes processos, sendo que cada solução é comparada com a melhor solução (partilhada) encontrada até ao momento. O código deste exemplo é ilustrado na Listagem 6.14.

Anomalias

Este teste também contém um *stale-value error*. Cada processo testa atomicamente se o número de passos da sua solução é inferior ao da melhor solução partilhada por todos os processos e, nesse caso, actualiza a segunda noutro método atómico. No entanto, entre os dois métodos atómicos, outro processo a executar o mesmo código podia ter alterado a melhor solução resultando numa actualização perdida.

Listagem 6.14: Teste: Knight (fragmento de código adaptado de [TLS10])

```
@Atomic
public int get_solution(Point p){
  return solution[p.x][p.y];
}

@Atomic
public void set_solution(Point p, int m){
  solution[p.x][p.y]=m;
}

private void check_and_set_solution() {
  if (moves <= km.get_solution(me)){  // Atomic
    // ... solution could have changed
    km.set_solution(me, moves);  // Atomic
  }
  // ...
}

// ...

public void run() {
  // ...
  if (check_and_set_solution() < 0)
    return;
  // ...
  new Solver(...).start();
  // ...
}
```

Resultados

O *datarace* foi correctamente detectado pelo Dependency Sensor. Foi criada a dependência entre métodos atómicos (get_solution, *solution*, set_solution) e, como outros processos podiam ter escrito nesta variável, foram obtidas todas as condições necessárias para a detecção do conflito.

Resumo

Zero falsos negativos e zero falsos positivos.

6.15 Teste: Arithmetic Database

Descrição

O exemplo descrito nesta secção também foi retirado de [TLS10]. O programa pretende simular uma base de dados com acessos concorrentes a duas tabelas que guardam informação sobre o valor de expressões regulares. A primeira tabela, exp_table, guarda o conjunto das expressões regulares e a segunda, res_table, faz a correspondência entre cada expressão e o seu resultado. Existindo uma chave externa entre as duas tabelas,

cada inserção na base de dados é dividida em duas, uma em cada tabela.

Na Listagem 6.15 são ilustrados os métodos principais do programa. O método get_key_by_result recebe uma chave e acede à tabela res_table retornando o valor da respectiva expressão. O método não atómico insert_new_expression permite a inserção de uma nova expressão regular. Para tal, testa se esta já existe, depois utiliza o valor da chave mais elevada para a introduzir na tabela de resultados, deixando a inserção na tabela exp_table para o final. Contudo, todas as operações referidas estão encapsuladas em métodos atómicos distintos permitindo que estas sejam intercaladas pela execução de outros processos.

Listagem 6.15: Teste: Arithmetic Database (fragmento de código adaptado de [TLS10])

```
1  @Atomic
2  private int get_key_by_result(int result){
3    for (Pair<Integer,Integer> t: res_table)
4      if (t.v == result)
5        return t.k;
6    return -1;
7  }
8
9  private void insert_new_expression(RPN_Expression exp){
10   Integer foreign_key=null;
11   if ((foreign_key = get_key_by_result(exp.evaluate())) < 0){
12     foreign_key=res_table.get_max_key(); // Atomic
13     foreign_key=(foreign_key == null) ? 0 : foreign_key+1;
14     res_table.insert(foreign_key,exp.evaluate()); // Atomic
15   }
16   exp_table.insert(exp,foreign_key);
17 }
```

Anomalias

Todos os conflitos encontrados ocorrem no método insert_new_expression:

1. O método começa por testar atomicamente se a expressão já existe na base de dados, verificando se o seu valor é menor que zero. Se este não existir, o processo insere a expressão noutro método atómico sendo que, entre estes dois, outro processo poderia ter inserido a mesma expressão regular resultando na dupla inserção do mesmo elemento.

2. A situação anterior pode ser igualmente aplicada no método atómico get_max_key. Este método só deveria ser chamado se a expressão regular a inserir não existisse. No entanto, entre os dois métodos atómicos, outro processo podia ter inserido essa mesma expressão.

3. O valor da chave mais elevada é obtido atomicamente e é usado na inserção de um novo elemento no método atómico insert. Entre estes dois métodos atómicos, outro

processo podia ter introduzido ou removido um elemento da base de dados, fazendo com que a variável local que guardava o valor da chave mais elevada ficasse obsoleto.

Resultados

Os três *dataraces* referidos foram correctamente detectados pelo Dependency Sensor, através das dependências get_key_by_result → insert, get_key_by_result → get_max_key e get_max_key → insert, respectivamente. Para além destes, foi reportado mais um conflito (falso positivo) relacionado com uma falsa dependência entre duas iterações do método get_key_by_result.

Resumo

Um falso positivo.

6.16 Teste: Elevador

Descrição

O primeiro *benchmark*, tal como nome indica, simula o funcionamento de um elevador onde cada andar é representado por um processo concorrente. Cada andar tem, em cada momento, um determinado número de pessoas que desejam utilizar o elevador para subir ou descer no edifício. Finalmente, cada processo guarda ainda informação sobre o estado do botão desse andar (se está ou não pressionado).

A Listagem 6.16 mostra um fragmento de código deste teste, onde é decidido para que andar o elevador se deve dirigir. Como se pode verificar, sempre que o elevador é chamado por mais do que um andar, é dada prioridade ao andar mais baixo.

Listagem 6.16: Teste Elevator: fragmento de código adaptado de [vG03]

```
1  // ...
2
3  for (int floor = firstFloor; !foundFloor && floor <= lastFloor; floor++) {
4    if (controls.claimUp(getName(), floor)) { // 2 Atomics
5      foundFloor = true;
6      targetFloor = floor;
7    }
8    else if (controls.claimDown(getName(), floor)) { // 2 Atomics
9      foundFloor = true;
10     targetFloor = floor;
11   }
12 }
13
14 // ...
```

Anomalias

Tal como é ilustrado na Listagem 6.16, o elevador percorre todos os andares até achar o primeiro cujo botão que permite chamar o elevador foi pressionado. Cada andar só é analisado se, em todos os anteriores, o elevador não tiver sido chamado (`foundFloor = false`). No entanto, entre duas chamadas dos métodos atómicos que avaliam se o elevador foi chamado em dois andares distintos, alguém podia ter chamado o elevador no primeiro fazendo com que o elevador não fosse para o andar mais baixo (como seria de esperar pela semântica do programa). Como ambos os métodos claimUp e claimDown são implementados através do teste de duas condições em dois métodos atómicos distintos cada um, então este conflito pode acontecer entre qualquer par de métodos dos quatro referidos (um de cada iteração do ciclo). Assim, este teste contém $4 \times 4 = 16$ *dataraces*.

Resultados

Os dezasseis *dataraces* foram correctamente detectados pelo *plugin* Dependency Sensor. Para além destes, foram ainda reportados 2 conflitos pelo *plugin* ViewConsistency Sensor e 4 pelo Dependency Sensor, num total de 6 falsos positivos.

Os primeiros dois falsos positivos foram gerados devido ao facto da ferramenta MoTH não conseguir distinguir duas instâncias do mesmo objecto. A implementação de uma análise *points-to* poderia ajudar a excluir este tipo de falsos alarmes.

Por outro lado, sempre que o elevador se encontra num andar, testa atomicamente se o botão desse andar foi pressionado e, caso tenha sido, é utilizado outro método atómico que corresponde à entrada das pessoas no elevador. Assim, foram criadas dependências entre os quatro métodos atómicos referidos e o método atómico que introduz as pessoas no elevador gerando consequentemente quatro conflitos. Intuitivamente, se entre o teste atómico que verifica se alguém carregou no botão e a abertura das portas para as pessoas entrarem o botão fosse "cancelado", então estaríamos a abrir as portas sem que ninguém estivesse no andar corrente. No entanto, apesar de outros processos poderem escrever nas variáveis que representam o botão desse andar, essa escrita corresponde à acção de carregar no botão, não influenciando em nada a abertura das portas. Para excluir este conflito seria necessário apurar o valor usado na afectação da variável que representa o botão, o que seria extremamente difícil (ou mesmo impossível) utilizando análise estática.

Resumo

Seis falsos positivos.

6.17 Teste: Philo

Descrição

Este *benchmark* simula o problema do Jantar dos Filósofos, ilustrado na Figura 6.2. Em 1965, Dijkstra introduz este problema de sincronização onde cinco computadores tentam aceder a cinco dispositivos periféricos [Dij68]. Este problema foi mais tarde descrito como o problema do Jantar dos Filósofos por Hoare [Hoa78].

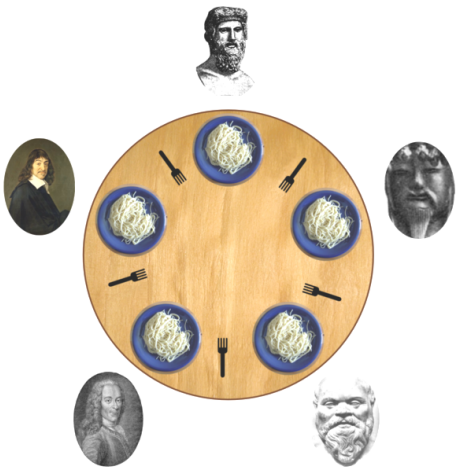


Figura 6.2: Ilustração do Problema do Jantar dos Filósofos

Este problema de programação concorrente consiste num conjunto de filósofos sentados numa mesa circular, cada um deles com um prato à frente e um garfo de cada lado. Para comer, cada filósofo precisa dos dois garfos mais próximos, não podendo utilizar os restantes da mesa. Neste cenário, podem surgir *deadlocks* se cada filósofo apanhar o garfo à sua esquerda e ficar eternamente à espera do segundo (ou vice-versa). Se, para tentar evitar este tipo de conflitos, cada filósofo pousar o garfo sempre que estiver à espera mais do que x segundos pelo segundo, então podemos ter um problema de *starvation* de recursos.

Nesta implementação específica do problema, cada filósofo é representado por um processo que tem dois blocos sincronizados que permitem adquirir e libertar ambos os garfos. O primeiro utiliza espera activa até verificar que ambos os garfos estão disponíveis enquanto o segundo método liberta os garfos e avisa os outros processos.

Anomalias

Tal como os autores referem, não existe qualquer tipo de *datarace* nesta implementação do programa já que os garfos são adquiridos e libertados simultaneamente dentro do escopo transaccional.

Resultados

Foram reportados dois *dataraces* pela ferramenta MoTH que serão considerados como falsos positivos. Estes *dataraces* resultam da criação de falsas dependências entre os dois métodos atómicos. Mais uma vez, as dependências geradas e, consequentemente, os respectivos *dataraces* poderiam ser excluídos através da implementação de uma análise *points-to*, permitindo a distinção entre diferentes instâncias da mesma classe.

Resumo

Dois falsos positivos.

6.18 Teste: Tsp

Descrição

O *benchmark* Tsp (*Travelling Salesman Problem*) consiste numa implementação do problema do Caixeiro Viajante. Dado um conjunto de cidades e as distâncias entre as mesmas, este problema NP-Completo consiste em apurar o caminho mais curto começando e acabando numa determinada cidade passando uma, e uma só, vez por cada cidade. Este problema foi formalizado inicialmente como um problema matemático e tem sido alvo de estudo em diversas áreas, nomeadamente em optimização de algoritmos.

Nesta implementação concorrente específica, são criados diversos processos que vão dividindo o problema em subproblemas computando concorrentemente todas as possibilidades. Originalmente o problema encontrava-se escrito com o mecanismo de *locks* e foi traduzido para a semântica transaccional, ainda que esta tradução contenha aninhamento de transacções que não é contemplado pelo nosso trabalho. Ainda assim, este teste permitir-nos-á ajudar a avaliar a escalabilidade da nossa ferramenta.

Anomalias

Os autores apresentam este teste como um programa isento de *dataraces*.

Resultados

Foram reportados dois conflitos pelo *plugin* Dependency Sensor que são considerados falsos positivos. Não é claro que o programa transaccional mantenha exactamente a semântica do programa original, devido à existência de blocos sincronizados com *locks* distintos. Para além disto, o programa traduzido contém aninhamento de transacções, que não é contemplado pela ferramenta MoTH. Teria de ser feito um estudo mais aprofundado para verificar se existe uma tradução correcta sem aninhamento de transacções e se os falsos positivos reportados continuariam a ser gerados. Contudo, e apesar dos resultados serem pouco conclusivos, este teste permitiu-nos avaliar o comportamento da nossa ferramenta na presença de testes maiores e mais complexos.

Resumo

Dois falsos positivos.

6.19 Teste: Store

Descrição

Este teste foi especialmente concebido para validar a nossa ferramenta e consiste num programa que simula o funcionamento de uma loja. Neste programa, são criados processos Client que, com base nos produtos da loja, fazem pedidos (carrinhos) com diversos produtos, processos Worker que estão constantemente à espera de novos pedidos de clientes e, finalmente, processos Supplier que aleatoriamente vão fornecendo produtos à loja aumentando o seu stock.

Na loja, cada produto disponível, para além da sua informação descritiva, está associado a um número n que representa o número de unidades desse mesmo produto na loja, e a um booleano *soldOut* que dita se este está esgotado ou não. Intuitivamente, em qualquer instante do programa este booleano deve conter o valor da expressão `(n == 0)`. Cada vez que é feita uma venda, estes atributos são actualizados em métodos atómicos distintos podendo gerar estados inconsistentes. É disponibilizado ainda um método que permite verificar a consistência do produto, i.e., se o booleano *soldOut* é coerente com o valor de n.

Na Listagem 6.17 são apresentadas as partes mais problemáticas do código deste teste.

Anomalias

1. O processo Worker testa se existe pelo menos um pedido pendente num método atómico e, se houver, vai buscar o primeiro pedido e trata-o noutro método atómico. Entre estes, outro processo a executar o mesmo código poderia ter consumido todos os pedidos existentes fazendo com que o primeiro tentasse tratar um pedido que não existe gerando, potencialmente, um NullPointerException.

2. Entre os dois métodos atómicos do método sellProduct outro processo poderia testar a consistência desse produto podendo observar que o produto não está esgotado apesar de não ter qualquer unidade (n = 0).

Resultados

Os dois *dataraces* foram correctamente detectados pelos *plugins* Dependency Sensor e ViewConsistency Sensor, respectivamente. No primeiro, foi criada uma dependência entre os métodos hasOrders e treateOrder sendo posteriormente detectado o *datarace*. No segundo, as *views* de escrita geradas pelos métodos atómicos decNumber e setSoldOut estão contidas na *maximal view* de leitura do método isConsistent e não formam uma cadeia.

Listagem 6.17: Teste Store: teste especialmente concebido para validar a ferramenta

```
// Classe Worker

public void run(){
  while(true){
    if(Store.hasOrders()){  // Atomic
      String log = treateOrder(); // Atomic
      Store.addLog(log);
    }
  }
  waitForClients();
}

// Classe StoreProduct

public void sellProduct(int units){
  decNumber(units); // Atomic
  // Someone can read this inconsistent state
  // (We have product but it is sold out)
  setSoldOut(n == 0); // Atomic
}

@Atomic
public boolean isConsistent(){
  return (soldOut && n == 0) || (n > 0 && !soldOut);
}
```

Resumo

Zero falsos negativos e zero falsos positivos.

6.20 Teste: Vector Fail

Descrição

Este teste foi desenvolvido por nós como um exemplo de um programa que contém uma anomalia que não é detectada pela nossa abordagem. O programa é constituído por diversos processos que partilham um vector de números, onde é verificada a propriedade que dita que o valor máximo do vector é sempre duas vezes superior ao valor mínimo do mesmo. Cada processo actualiza periodicamente os elementos do vector partilhado, mantendo sempre a propriedade referida e testando-a no final de cada actualização. Na Listagem 6.18 é apresentada a parte relevante deste teste. Nesta implementação simples, o vector tem apenas dois elementos guardando ainda informação sobre qual dos dois representa o máximo do vector.

Listagem 6.18: Teste: Vector Fail

```
1  // Classe Thread
2  public void run() {
3    while(true){
4      Random r = new Random();
5      int val = r.nextInt(10);
6      vector.setElements(val, val*2);
7
8      int max = vector.getMax();
9      int min = vector.getMin();
10
11     assert(max == 2*min);
12   }
13 }
14
15 // Classe Vector
16 @Atomic
17 public int getMax() {
18   if(firstIsGreater)
19     return first;
20   return second;
21 }
22
23 @Atomic
24 public int getMin() {
25   if(!firstIsGreater)
26     return first;
27   return second;
28 }
```

Anomalias

O *datarace* presente neste teste está relacionado com o facto das leituras dos valores máximo e mínimo serem feitas em blocos distintos. Entre as chamadas dos métodos atómicos getMax e getMin, outro processo a executar o mesmo código pode actualizar os elementos do vector. Se isto acontecer, então a verificação da propriedade, na linha 11, será feita com dois valores que pertencem a duas versões distintas do vector.

Resultados

O *datarace* descrito não foi detectado pela nossa abordagem. Esta anomalia está relacionada com acessos parciais ao vector, mais precisamente aos valores máximo e mínimo. No entanto, segundo a nossa abordagem, ambos os elementos do vector (*first* e *second*) são acedidos nos métodos getMax e getMin. Devido à existência de código condicional, estes métodos deveriam gerar duas *views* cada, permitindo assim detectar o conflito. Nenhum outro conflito foi reportado.

Resumo

Um falso negativo.