Técnicas para Detecção de Código Morto: Uma Revisão Sistemática de Literatura

Alternative Title: Detection Techniques of Dead Code: Systematic Literature Review

Camila Bastos
Federal University of Lavras
Department of Computer Science
Caixa Postal 3037 - CEP 37220-000
Lavras - MG - Brasil
camila.bastos@posgrad.ufla.br

Paulo Afonso Junior
Federal University of Lavras
Department of Computer Science
Caixa Postal 3037 - CEP 37220-000
Lavras - MG - Brasil
pauloajunior@dcc.ufla.br

Heitor Costa
Federal University of Lavras
Department of Computer Science
Caixa Postal 3037 - CEP 37220-000
Lavras - MG - Brasil
heitor@dcc.ufla.br

RESUMO

A evolução é necessária para sistemas de informação não se tornarem inadequados e obsoletos. No entanto, essa evolução tem sido identificada como aspecto crítico em assegurar a manutenibilidade, por causa do aumento da quantidade de código morto nesses sistemas. A identificação e a eliminação de código morto reduzem o tamanho do código, diminuem a complexidade e facilitam a compreensão. Algumas técnicas foram propostas para automatizar a detecção desse código morto e estão disponíveis na literatura. Com base nisso, foi utilizada a técnica Revisão Sistemática de Literatura para encontrar técnicas de detecção de código morto existentes e o domínio em que foram aplicadas. Como resultado, duas principais técnicas foram encontradas: Análise de Acessibilidade e Análise de Fluxo de Dados. Além disso, análise quantitativa e análise qualitativa foram realizadas e são apresentadas para subsidiar pesquisadores sobre qual técnica utilizar.

Palavras-Chave

Código morto, Técnicas de Detecção de Código Morto.

ABSTRACT

The evolution is necessary for information systems do not become inadequate. However, this evolution has been identified as critical aspect in ensuring maintainability, because of the increased amount of dead code in these systems. The identification and elimination of dead code decreases the code size and the complexity, facilitating the understanding. Techniques have been proposed in the literature for automating the detection of dead code. Thus, Systematic Literature Review was performed to find existing dead code detection techniques. As result, two main techniques were found: Accessibility Analysis and Data Flow Analysis. In addition, quantitative and qualitative analysis were performed and they are presented to help researchers on which technique to use.

SBSI 2016, May 17-20, 2016, Florianópolis, Santa Catarina, Brazil.

Categories and Subject Descriptors

D.2 SOFTWARE ENGINEERING (K.6.3). D.2.7 Distribution, Maintenance, and Enhancement

General Terms

Experimentation.

Keywords

Dead code, Detection Techniques of Dead Code.

1. INTRODUÇÃO

Os sistemas de informação tendem a sofrer constantes modificações ao longo de seu tempo de vida. De acordo com a Lei da Mudança Contínua, esses sistemas precisam ser adaptados continuamente para evoluírem e não se tornarem inadequados e obsoletos [9][10]. Apesar de necessária, a evolução tem sido identificada como aspecto crítico em assegurar a manutenibilidade desses sistemas. Isso ocorre por causa do aumento da poluição do código ao longo da evolução, por exemplo, surgimento/aumento de código morto [6][4].

A identificação e a eliminação de código morto podem contribuir para reduzir a poluição do código de sistemas de informação. De modo geral, código morto pode ser definido como trechos de código nunca executados e que pode ser removido sem alterar o comportamento ou a funcionalidade desses sistemas [5]. A presença de código morto afeta a sua manutenibilidade por dificultar a compreensão, aumentar desnecessariamente o tamanho do código e dificultar a execução de testes [12][1][7].

A identificação de código morto é considerada uma tarefa árdua de ser realizada, pois depende da análise e da compreensão de

quase todo o código de um sistema de informação, pois é o principal artefato disponível [11][3]. Com isso, técnicas que automatizam a identificação de código morto podem contribuir com a redução da poluição do código e, consequentemente, facilitar a execução das atividades de manutenção.

Ao longo dos anos, pesquisadores propuseram técnicas para automatização da detecção de código morto em sistemas de informação (apresentadas na Seção 4). Foram propostas técnicas considerando código morto como componentes de código não executados, código que produz resultados não utilizados e código parcialmente morto, cujos resultados produzidos podem não ser utilizados dependendo do fluxo de execução. Além disso, as técnicas propostas foram aplicadas a diferentes domínios.

Tendo conhecimento da importância da automatização da detecção de código morto, neste trabalho, o objetivo é fornecer uma visão geral das técnicas propostas para a detecção de código morto e o domínio em que foram aplicadas. Além disso, apontar a distribuição de estudos relacionados ao longo dos anos, autores que realizaram pesquisa na área e eventos relacionados ao assunto. Com essas informações, os pesquisadores interessados na área podem optar pela técnica mais adequada de acordo com seu objetivo de aplicação. A busca desses trabalhos foi realizada utilizando a técnica Revisão Sistemática de Literatura (RSL). Os resultados obtidos foram sintetizados de forma a oferecer visões quantitativas e qualitativas em relação ao assunto.

O restante do trabalho está organizado da seguinte maneira. A metodologia de condução da RSL está descrita na Seção 2. A análise quantitativa e a análise qualitativa são discutidas na Seção 3 e na Seção 4, respectivamente. Conclusões, contribuições e sugestões de trabalhos futuros são apresentadas na Seção 5.

2. REVISÃO SISTEMÁTICA DE LITERATURA

A RSL foi escolhida para realização deste estudo por ser um processo metodológico cuidadosamente controlado por um protocolo de investigação formal e que fornece consistência e robustez nos seus resultados [2]. A questão de pesquisa definida para guiar a seleção de trabalhos relevantes foi:

```
Quais são as técnicas de detecção de código morto
existentes na literatura?
```

Com base na questão, foi definida uma *string* de busca para auxiliar na padronização da pesquisa em diferentes repositórios de trabalhos científicos. Essa *string* é composta por palavras-chave que especificam o assunto procurado nos trabalhos:

```
("Dead Code" OR "Unnecessary Code" OR "Unused
Code" OR "Dead Source Code" OR "Inaccessible Code"
OR "Unapproachable Code" OR "Unreachable Code")
AND (Detection OR Detecting OR Removing OR
Remotion OR Elimination OR Eliminating) AND
(Technique OR Algorithm OR Strategy)
```

O nome e o endereço eletrônico dos repositórios utilizados são listados na Tabela 1. Para serem selecionados, os artigos deveriam possuir acesso livre ao seu conteúdo e ser publicado a partir do ano de 1990 (critérios de seleção). Não foram definidas restrições quanto ao idioma do artigo. Após a definição do protocolo, a seleção dos artigos relevantes foi realizada por meio da execução dos seguintes passos:

- **Passo 1**: A *string* de busca foi utilizada nos repositórios para coletar trabalhos. Os trabalhos resultantes foram analisados para retirar os que não são artigos;

Tabela 1. Repositórios de artigos científicos utilizados.

Repositórios	Endereços Eletrônicos
IEEE Xplore Digital Library	http://ieeexplore.ieee.org
ACM Digital Library	http://www.acm.org
Enginnering Village	http://www.engineeringvillage.com
Springer Link	http://link.springer.com
Science Direct	www.sciencedirect.com
Scopus	http://www.scopus.com

- **Passo 2**: Em seguida, foi realizada a seleção primária com a leitura do título, do resumo e das palavras-chave para seleção de trabalhos relevantes;
- **Passo 3**: Os artigos de cada repositório resultantes do Passo 2 foram reunidos para facilitar no processo de eliminação de trabalhos duplicados;
- **Passo 4**: Foi realizada a seleção secundária. Os artigos resultantes do Passo 3 foram analisados com a releitura do resumo e a leitura das conclusões.

A quantidade de artigos obtidos com a execução desses passos é apresentada na Tabela 2. A segunda coluna contém a Quantidade Inicial (QI) de artigos obtidos após a realização do Passo 1. A terceira coluna contém a quantidade de artigos obtidos após a Seleção Primária (SP), em que foi realizada a leitura de títulos, resumos e palavras-chave desses artigos. Na quarta, na quinta e na sexta colunas são apresentados os resultados obtidos com Seleção Secundária (SS), em que foi realizada a releitura do resumo e a leitura das conclusões. São apresentadas a quantidade de artigos irrelevantes (IR), de artigos repetidos (RP), de artigos incompletos (IN) e de artigos selecionados em cada repositório (R).

Tabela 2. Quantidade de artigos selecionados.

Repositórios	QI	SP	SS			
			IR	RP	IN	R
IEEE Xplore	1.577	44	20	7	1	16
ACM	1.216	16	3	2	1	10
EiCompendex	79	17	4	4	2	7
Springer Link	327	3	2	0	0	1
Science Direct	331	14	4	2	0	8
Scopus	290	62	16	25	10	11
Total	**3.820**	**156**	**49**	**39**	**14**	**53**

Três pesquisadores (Pesquisador A - PA, Pesquisador B - PB e Pesquisador C - PC) foram envolvidos na seleção dos estudos primários e seguiram o seguinte procedimento:

- PA executou a *string* de busca nas fontes selecionadas e documentou os resultados no software JabRef (http://jabref.sourceforge.net/);
- PA verificou e excluiu os trabalhos que não eram artigos e os repetidos (com título, autores e resumo iguais). Na identificação de trabalhos repetidos, os trabalhos mantidos foram aqueles com palavras-chave mais relacionadas com código morto e técnicas de detecção;
- Os trabalhos encontrados foram avaliados por PA e por PB, de maneira individual e separada, quanto ao atendimento aos

critérios de inclusão e de exclusão estabelecidos. Os trabalhos, cujas avaliações causaram dúvidas quanto a sua inclusão/ exclusão por parte dos pesquisadores, foram incluídos;

- Os trabalhos foram documentados em uma lista de trabalhos incluídos/excluídos com justificativa para inclusão/exclusão;
- Realizou-se a interseção entre os trabalhos selecionados por PA e por PB, sendo esses trabalhos documentados (Interseção 1). Na ocorrência de desacordos sobre a inclusão/exclusão de trabalhos, PA e PB discutiram e resolveram. Em casos que não houve consenso, o trabalho foi incluído. Os trabalhos excluídos foram documentados em uma lista de trabalhos excluídos com justificativa para exclusão;
- PC avaliou os trabalhos resultantes da *string* de busca considerando o título, o resumo e as palavras-chave;
- PC realizou a interseção entre os trabalhos selecionados por ele e os trabalhos presentes na Interseção 1. Juntamente com PA e PB, as discordâncias foram resolvidas. O resultado foi o conjunto de trabalhos resultantes da Revisão Sistemática.

Os 53 artigos selecionados foram utilizados para realizar a análise qualitativa e a análise quantitativa descritas nas próximas seções.

3. ANÁLISE QUANTITATIVA

A análise quantitativa pode ser realizada para investigar a precisão da análise qualitativa e investigar aspectos relevantes a respeito do assunto estudado. Neste trabalho, foram realizadas análises quantitativas em relação ao ano em que os artigos foram publicados, aos autores mais influentes e aos locais em que foram publicados.

Na análise de anos de publicação, foi investigada a quantidade de artigos relacionados com detecção de código morto publicados. Nessa análise, o objetivo é verificar se pesquisas nessa área apresentam tendência de crescimento, podendo ser um incentivo a outros pesquisadores ingressarem no assunto. A quantidade de artigos publicados não seguiu uma tendência (Figura 1), pois houve oscilação na quantidade de artigos nos anos. Apesar da quantidade de pesquisas na área não apresentar crescimento, pode-se perceber que esse assunto é estudado continuamente.

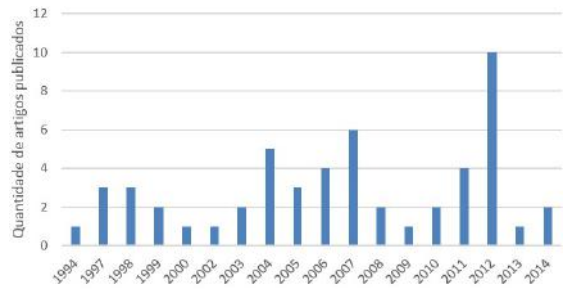

Figura 1. Publicações por ano.

Para identificar possíveis autores influentes na área, foram contabilizados os autores dos artigos obtidos na RSL. Com isso, foram identificados 117 autores distintos, o que mostra razoável interesse dos pesquisadores sobre o assunto. Na Figura 2, são apresentados os autores que publicaram mais de um artigo. Apesar da quantidade significativa de pesquisadores encontrados, poucos publicaram mais de um artigo relacionado ao tema. Praticamente, metade dos estudos encontrados (51%) não tinham como objetivo principal a detecção de código morto, podendo ser uma justificativa para os resultados apresentados. Possivelmente, publicações futuras dos autores desses trabalhos estariam relacionadas com outras otimizações e não especificamente com detecção de código morto.

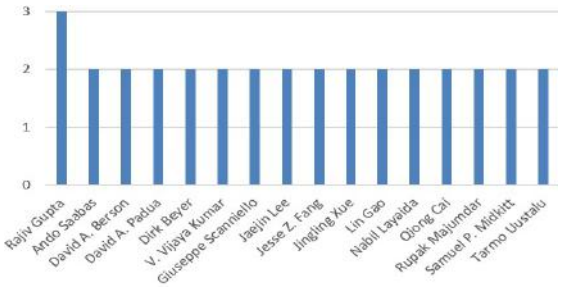

Figura 2. Autores dos artigos selecionados.

Foram identificados 43 locais em que os artigos selecionados foram publicados (Figura 3). O local com mais publicações foi *ACM SIGPLAN Notices*, voltado para assuntos relacionados com compiladores e linguagens de programação. Isso ocorre porque parte dos trabalhos encontrados propõem a detecção de código morto com o objetivo de otimização de compiladores e, em menor quantidade, com propósito de facilitar as atividades de manutenção. Em seguida, *International Conference Software Engineering* (ICSE) tem três publicações. Depois, *Electronic Notes in Theorical Computer Science* (ENTCS), *Information and Software Technology* (IST), *International Symposium on Code Generation and Optimization* (ISCGO), *Science of Computer Programming* (SCP) e *Conference on Software Maintenance and Reengineering* (CSMR) têm duas publicações.

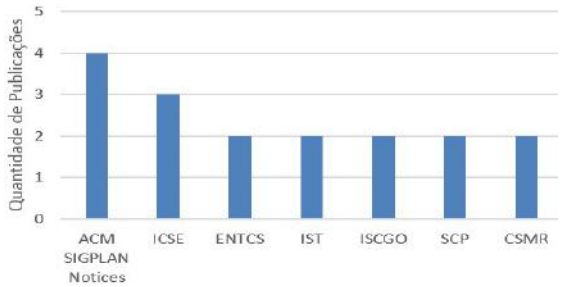

Figura 3. Locais de publicação dos artigos.

4. ANÁLISE QUALITATIVA

Para responder à questão de pesquisa, os artigos selecionados foram analisados qualitativamente. Esses artigos foram lidos na íntegra para encontrar quais técnicas de detecção de código morto foram utilizadas. A questão de pesquisa a ser respondida foi:

```
Quais são as técnicas de detecção de código morto
existentes na literatura?
```

Foram encontrados artigos relacionados com detecção de código morto aplicados em diferentes contextos. A análise de fluxo de

dados e a análise de acessibilidade foram as técnicas mais utilizadas, mas aplicadas de maneiras distintas.

4.1 Análise de Fluxo de Dados

A análise de fluxo de dados (AFD) é indicada para detecção de atribuições parcialmente mortas e de componentes de código que produzem resultados não utilizados. Uma instrução é parcialmente morta se o valor produzido por ela não é utilizado em um caminho de execução, porém utilizado em outros caminhos [8]. Na Figura 4, o fluxo de execução dado pela sequência `1`, `2`, `3`, `5` e `6` gera uma atribuição parcialmente morta, pois o valor atribuído a variável `y` em `1` não é utilizado, sendo substituído pela atribuição efetuada em `3`. Entretanto, para o fluxo `1`, `2`, `4`, `5` e `6` não acontece a substituição do valor de `y`.

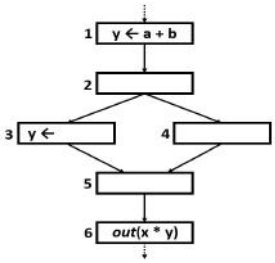

Figura 4. Exemplo de código parcialmente morto.

Foram encontrados artigos que utilizam a técnica AFD para solucionar problemas de código morto. Em [A31], *bytecodes* Java foram convertidos para *Static-Single Assignment* (SSA) e representados por uma árvore. SSA é uma representação intermediária que fornece informações relacionadas ao fluxo de dados, simplificando a implementação de otimizações. Antes de iniciar a análise, os componentes de código representados pelos nós da árvore são definidos como mortos. Esses componentes devem estar de acordo com uma das três condições de vivacidade para serem classificados como vivos: i) declarações que afetam a saída do sistema de informação; ii) instruções que possuam resultados utilizados por instruções classificadas como vivas; e iii) declarações condicionais que possuam outras instruções vivas dependentes. Após a análise dos nós da árvore, os nós que continuam marcados como mortos são removidos.

Além da representação em árvore, é comum ser utilizada a representação em grafos para realizar a análise dinâmica de fluxo de dados, como realizado em [A53]. Essa análise dinâmica é efetivada utilizando a representação binária do código por meio de um grafo. Em [A37], o código parcialmente morto é eliminado, utilizando análise de caminhos frequentemente executados. Basicamente, o primeiro passo do algoritmo é verificar quais instruções estão presentes nos caminhos mais executados. Para cada caminho, os predicados dessas instruções são analisados; se não for utilizado ao longo desse caminho, a instrução é definida como morta. Em [A19], a AFD é feita após a identificação de quais partes do código devem ou podem ser analisadas. Além disso, em [A33], a AFD é realizada sob demanda e na ordem inversa do grafo.

Em [A47] e [A41], o sistema de informação é dividido em diversos blocos. A detecção de código morto é realizada com a AFD em cada bloco. Em [A3], foi utilizada a representação SDDA (*Speculative Data Dependence Analysis*), desconsiderando as dependências de dados com baixa probabilidade de serem otimizadas. Em seguida, a AFD foi feita e as atribuições consideradas parcialmente mortas foram movidas para ramos do grafo em que se tornam vivas. Outra abordagem baseada na análise de divisões do sistema de informação foi proposta em [A23]. O sistema é dividido em fatias de acordo com sua semântica. Cada fatia é representada por um subgrafo para análise do fluxo de dados. Além disso, em [A36], são apresentados algoritmos para eliminação de código parcialmente morto. Essa eliminação é realizada por meio do fatiamento do código para reestruturação do fluxo de execução.

Com o fluxo de dados, pode-se realizar análise estática. Em [A22], o algoritmo proposto divide o sistema de informação em regiões analisadas separadamente e representadas por nós do grafo. Em [A34], além do grafo ser analisado de maneira *top-down*, como nas outras abordagens, é realizada a análise *bottom-up*. Em [A46], a detecção é realizada de maneira estática, mas o objetivo é a análise em aplicativos para Android. Em [A20], a detecção de código morto é realizada em aplicações JavaScript. Os eventos do sistema são modelados com grafos que representam o fluxo de dados. Os nós do grafo não executados são considerados mortos.

Também, foram propostos estudos relacionados com detecção de código morto em sistemas de informação concorrentes. Em [A29], a AFD é utilizada para detecção de variáveis vivas do tipo ponteiros, utilizando *threads*. Em [A12], foi proposta uma adaptação do processo de AFD tradicional. Quando uma atribuição parcialmente morta é encontrada, ela é movimentada para ambos os ramos do grafo. Assim, a atribuição torna-se morta em um dos ramos e pode ser eliminada. O procedimento é realizado em dois passos: i) *code sinking* (movimentação do código para nós subjacentes no grafo de análise); e ii) eliminação de código morto. Considerando que otimizações realizadas em sistemas de informação em paralelos não podem prejudicar o desempenho, foi definida uma restrição cuja otimização em um componente paralelo é realizada quando for propagada para os outros componentes.

Outra abordagem para otimização de sistemas de informação em paralelos foi dada em [A27]. O algoritmo proposto inicia com a marcação das declarações desses sistemas como mortas. Esse conjunto de declarações é armazenado em uma lista utilizada para acompanhamento ao longo da sua execução. A cada instrução marcada como viva, a lista é atualizada. As regras para uma declaração ser considerada viva são equivalentes às apresentadas em [A31]. A mesma abordagem proposta em [A29] foi realizada em [A21], mas para sistemas de informação não paralelos, fazendo com que a definição da restrição seja desnecessária. No trabalho [A42], antes do *code sinking*, é feita uma análise de disponibilidade para verificar em quais trechos de código essa operação pode ser feita e os impactos que podem acarretar. Para detecção de parâmetros não utilizados em procedimentos, em [A28], foi realizada análise de dependência. Para uma variável `l`, é feito um cálculo para verificar as variáveis que dependem de `l` no procedimento. Caso não seja encontrada variável dependente, `l` é considerada uma variável morta.

Dois estudos propuseram a integração de métodos matemáticos com AFD. Em [A18], o grafo de controle de fluxo, determinado SPTerm, é construído utilizando expressões algébricas. A detecção de código morto é realizada com análise *top-down* e *bottom-up* no grafo. Em [A39], a AFD é realizada de maneira

tradicional, mas foram utilizadas demonstrações matemáticas para provar a eficiência da técnica.

Outra variação da AFD foi apresentada em [A4], cuja detecção de código morto é realizada com auxílio de tabelas *Hash*. Esse estudo considera o conceito de propagação de cópias, ou seja, dada uma atribuição `x := y`, o uso de `y` será posteriormente substituído pelo uso de `x` sem que instruções intermediárias alterem seu valor, fazendo com que o valor de `x` e `y` seja igual, tornando `y` inútil e código morto. O algoritmo proposto consiste basicamente em dois passos: i) é feita a análise para identificar cópias e as informações obtidas são armazenadas em tabelas *Hash*; e ii) as informações contidas na tabela *Hash* são analisadas para detecção do código morto.

Em [A52], foi realizado um estudo para detectar atribuições mortas utilizando análise de pilhas e de *bytecodes* Java. A análise é realizada para detectar instruções de atribuição mortas. A análise de código morto baseada em pilha não é simples, pois as instruções empilhadas podem não estar de maneira sequencial na pilha, dificultando a visualização das dependências entre as instruções. A abordagem proposta para eliminar código morto consiste em duas etapas: i) instruções de armazenamento e de adição e saltos condicionais são selecionados; e ii) essas instruções são analisadas e eliminadas, se consideradas mortas.

4.2 Análise de Acessibilidade

A análise de acessibilidade (AA) é uma técnica que pode ser utilizada para verificar os componentes do sistema de informação que não podem ser acessados e, consequentemente, nunca são executados. De modo geral, a AA é baseada na definição de dois conceitos apresentados em [A1] e [A35]: i) conjunto de entidades alcançáveis (conjunto de entidades `R` alcançáveis a partir de um conjunto de entidades); e ii) conjunto de entidades fonte (conjunto `S` das entidades presentes no sistema). O conjunto de entidades mortas é dado pela diferença entre `S` e `R`, ou seja, as entidades não alcançáveis não são necessárias para a execução do sistema de informação e são consideradas como código morto.

Foram encontrados estudos que aplicam a AA para detectar código morto, mas com adaptações de acordo com o domínio na qual é aplicada. Em [A2], foi utilizada a AA dirigida por propósitos, ou seja, trechos de código considerados vulneráveis quanto à ocorrência de problemas são analisados. Esse tipo de análise implica em um custo mais baixo e consumo de menos recursos do computador. No estudo realizado em [A11], essa AA foi utilizada em sistemas de informação desenvolvidos com a linguagem de programação Esterel.

Em [A5], a análise para detecção de código morto é realizada no processo de engenharia reversa, em que o código Java é transformado em redes de Petri. Em seguida, é realizada a busca por componentes de código inacessíveis utilizando a análise da rede. Essa técnica depende do nível de abstração utilizado no modelo, pois, quanto maior a quantidade de detalhes colocado no diagrama, maior o seu tamanho e a sua complexidade. A AA proposta em [A6] efetua o rastreamento para cada componente `l` do sistema de informação. Se `l` for um componente acessível, a consulta retorna o caminho até `l`. Caso contrário, é retornado uma prova que `l` é inacessível e, consequentemente, código morto.

Nos estudos realizados em [A26] e em [A48], a AA é realizada sob código representado em SSA. Em [A48], o grafo gerado é analisado e métodos não alcançáveis são definidos como mortos. Em [A26], o procedimento de detecção de código morto é equivalente, mas a representação em SSA é gerada de maneira diferente, em que as relações do grafo são calculadas em termos de equações matriciais simples.

Em [A38], é realizada a detecção de código morto em sistemas de informação embarcados utilizando AA. Primeiramente, os métodos desses sistemas são definidos como acessíveis. Em seguida, o conteúdo de cada método é analisado e as chamadas para outros métodos encontradas são armazenadas. Métodos não chamados são considerados inacessíveis e definidos como código morto. A solução proposta para encontrar código inacessível em [A14] é baseada no conceito de Autômato Invariante de Erro (AIE). Esse conceito pode ser considerado uma abstração de um fragmento de código de entrada que contém somente declarações e fatos relevantes para a compreensão das causas de inconsistências. Com AIE, pode-se descrever os estados alcançáveis a partir de determinado local do sistema de informação. O trabalho realizado em [A15] utiliza AA simbólica, uma técnica formal que verifica a inacessibilidade de instruções de código de maneira exaustiva.

4.3 Demais Técnicas

Além de AFD e de AA, foram encontradas outras maneiras para detectar código morto. No estudo mostrado em [A40], foi realizada a detecção de código morto utilizando anotações de código Java (comentários de código). No exemplo apresentado na Figura 5, o `return 2` é considerado código morto, pois, de acordo com a anotação referente ao método `withPre (int x)`, o valor da variável `x` não poderá ser menor que `10`. Assim, o código dentro da instrução `if` pode ser inútil.

```
/*@ requires x > 10;
  @ ensures
  @  \result == 1; */
int withPre(int x) {
   if (x < 10) {
     //not checked
     return 2;
   }
   return 1;
}
```

Figura 5. Exemplo de anotação de código.

Alguns estudos aplicaram métodos matemáticos para detectar código morto. No trabalho realizado em [A45], é utilizado o estimador de Kaplan Meier para estimar a probabilidade de um método ser "sobrevivente" nas futuras versões do sistema de informação em análise. Em [A43], é realizada uma análise do sistema utilizando relacionamento de seus componentes. Esses relacionamentos foram representados por formulações matemáticas e utilizados para detecção de código morto. Em [A24], foi apresentada uma abordagem cuja eliminação de código morto é realizada em sistemas funcionais por meio da combinação de inferências.

Dois estudos propuseram a detecção de código morto utilizando análise de esquemas XML (*eXtensible Markup Language*). Um esquema é um documento que fornece meios para definição da estrutura, do conteúdo e da semântica de documentos XML. Em [A51], foi realizada a análise de documentos XQuery juntamente com o esquema que descreve seu conjunto de restrições. Para cada expressão XQuery, é efetuada uma análise em seu esquema para validar o significado da expressão de acordo com as restrições definidas. É comum que expressões XPath não estejam de acordo com a especificação no esquema. Nesse caso, as instruções XQuery dependentes dessas expressões são consideradas código

morto. Em [A13], é apresentada uma ferramenta que detecta código morto por meio da verificação das expressões XPath com o esquema que as define. Para detectar inconsistências, o algoritmo converte as expressões XPath em um esquema e realiza a comparação com o esquema original. Expressões em desacordo são consideradas código morto.

Em [A8], foi apresentado um algoritmo para definição de variáveis mortas em laços de repetição. Basicamente, o algoritmo proposto faz uma varredura do código para encontrar as instruções de atribuição definidas. As variáveis, que recebem o conteúdo da atribuição, são adicionadas ao conjunto das variáveis declaradas. Em seguida, é realizada outra análise para encontrar as variáveis atribuídas. As variáveis encontradas são adicionadas ao conjunto das variáveis utilizadas. Se uma variável está no conjunto de variáveis declaradas e não está no conjunto de variáveis utilizadas, ela é considerada código morto.

Uma abordagem diferente foi proposta em [A44] para detecção de código morto em aplicações de dispositivos móveis. Como sistemas embarcados possuem recursos de memória limitados, os arquivos de código da aplicação ficam hospedados em um servidor. De acordo com as requisições de uso, os arquivos necessários são transferidos para o dispositivo. Com isso, podem ser analisados quais blocos nunca foram transferidos para o dispositivo, sendo desnecessário para aplicação e classificados como código morto.

Há estudos que optaram por não analisar diretamente o código para detectar código morto, mas suas representações. Em [A10], para suprir a necessidade de análise de código em diversas linguagens, foi proposto efetuar análise sob código *assembler*. A abordagem utilizada em [A25] é uma análise do código representado em *Concurrent Static Single Assignment* (CSSA) em sistemas de informação em paralelos, utilizando um algoritmo para detectar propagação de cópias responsável pela geração de código morto.

Em [A30], uma técnica foi proposta para detectar código morto em código PHP (*Hypertext Preprocessor*). Essa técnica realiza análise sobre os arquivos do sistema de informação e verifica a quantidade de vezes que esse arquivo foi utilizado em tempo de execução. Quanto menor for a quantidade obtida para um arquivo, maior será a chance de ser classificado como morto. Esse tipo de análise pode produzir resultados não exatos, pois arquivos que ainda estão em desenvolvimento não serão executados e serão rotulados como morto. Para resolver esse problema, a técnica proposta realiza uma verificação do tempo de acesso, ou seja, arquivos não executados, mas que possuam tempo de acesso recente, não serão considerados código morto.

Utilizando algumas medidas, é possível prever a existência de código morto em sistemas de informação Java. No estudo desenvolvido em [A9], foi investigada a relação existente entre medidas e código morto. As medidas analisadas foram propostas por Chidamber e Kemerer (CK) e algumas de tamanho de código tradicionais. Com essa técnica, podem ser previstos métodos mortos. Como resultado, obteve-se que as medidas *Weighted Methods per Class* (WMC) e *Response For a Class* (RFC) propostas por CK e as medidas de tamanho *Number of Message Sends* (NMS), *Number of Methods* (NM) e *Lines of Code* (LOC) são importantes para previsão de código morto.

Na eliminação de código morto realizada em [A50], foram utilizadas *liveness patterns* baseados em gramáticas e em árvores regulares para análise de sistemas de informação recursivos. *Liveness patterns* são construtores responsáveis por indicar quais partes do código devem ser mortos e quais devem ser vivos. Foi realizada análise em pontos desses sistemas com base nas restrições construídas a partir da semântica e da linguagem de programação.

4.4 Trabalhos sem Sugestões de Técnicas

O objetivo das seleções primária e secundária executadas na RSL foi selecionar artigos que contém resposta para a questão de pesquisa. Como essas seleções são baseadas apenas na leitura do resumo e da conclusão, não é possível garantir que o artigo possua uma resposta para a questão de pesquisa. Com isso, alguns artigos selecionados foram relacionados com código morto, entretanto não propuseram técnicas para detecção ([A7], [A17], [A32], [A49] e [A16]).

4.5 Discussão dos Resultados

As principais técnicas de detecção de código morto encontradas foram AFD e AA. Além dessas, foram encontradas algumas técnicas referenciadas em menor quantidade, como a análise de anotações de código, estimador de Kaplan Meier, análise de arquivos XPath, usabilidade de arquivos e medidas. Conforme apresentado na Figura 6, a AFD foi a técnica mais utilizada, sendo referenciada em quase metade dos artigos selecionados (45%). As técnicas menos referenciadas foram contabilizadas em "Demais Técnicas" (25%). Alguns artigos não propuseram técnicas de detecção de código morto, representando 9% do total.

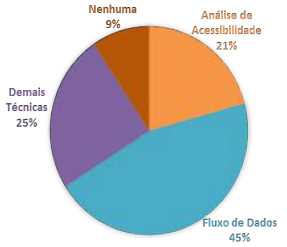

Figura 6. Quantidade de artigos por técnica.

Durante a análise dos artigos, foi identificado que o termo "código morto" está associado a diferentes definições. Há trabalhos que o definem como componentes de código que nunca são executados. Outros trabalhos relacionam esse termo a componentes de código que produzem resultados não utilizados. Nesse contexto, ainda existe o conceito de código parcialmente morto, cujos componentes do sistema de informação produzem resultados não utilizados dependendo do caminho da execução. Conforme apresentado na Tabela 3, algumas técnicas foram mais usadas de acordo com a definição de código morto considerada.

Dos 20 estudos que definiram código morto como Componentes de Código Não Executados (CNE), 11 estudos (55%) utilizaram a AA como técnica de detecção. A AFD foi utilizada por 3 estudos (15%) e algumas técnicas variadas (30%) foram propostas para detecção desse problema. Foram encontrados 18 estudos que definiram código morto como componentes do sistema de informação que produzem Resultados Não Utilizados (RNU). Dentre esses estudos, 12 (66,67%) utilizaram a AFD para detecção. Técnicas Diversas (TD) foram propostas em 6 estudos (33,33%). Não foram encontrados estudos que consideraram essa definição e utilizaram a AA para detecção.

Tabela 3. Classificação dos estudos.

	CNE	RNU	PM
AA	[A1] [A2] [A5] [A6] [A11] [A14] [A15] [A26] [A35] [A38] [A48] - Total: 11 estudos	-	-
AFD	[A20] [A23] [A46] - Total: 3 estudos	[A3] [A4] [A18] [A27] [A28] [A29] [A31] [A34] [A39] [A47] [A52] [A53] - Total: 12 estudos	[A12] [A21] [A22] [A33] [A36] [A37] [A41] [A42] - Total: 8 estudos
TD	[A9] [A10] [A30] [A40] [A45] [A44] - Total: 6 estudos	[A8] [A13] [A24] [A25] [A50] [A51] - Total: 6 estudos	-

A única técnica de detecção aplicada nos 8 estudos que utilizaram o conceito de código Parcialmente Morto (PM) foi a AFD. Dois estudos não foram classificados: i) em [A43], não foi apresentada a definição de código morto utilizada, e ii) em [A19], foi utilizada a AFD considerando duas definições de código morto. De acordo com os resultados, para detecção de código não executado, a técnica mais adequada é a AA. Para detecção de código parcialmente morto e de componentes de código que produzem resultados não utilizados, a AFD pode ser a mais indicada. Além disso, foi possível identificar que alguns artigos propuseram ferramentas para análise de código morto. Esse fato levou a elaboração de uma nova pergunta para identificar nesses artigos:

`Quais artigos desenvolveram ferramentas para aplicação das técnicas de detecção de código morto propostas?`

Na Tabela 4, são apresentados o código dos artigos que propuseram/utilizaram ferramentas, o repositório em que foram encontrados e a técnica de detecção de código morto utilizada. Apenas 8 artigos (15%) propuseram/utilizaram ferramentas. Pode-se observar que metade dos artigos que propuseram/utilizaram ferramentas não definiram a técnica utilizada.

Tabela 4. Estudos que propuseram ferramentas.

Código	Repositório	Técnica
[A6]	IEEE	Análise de Acessibilidade
[A13]	Scopus	Outras técnicas
[A16]	IEEE	Não sugeriu técnicas
[A17]	ACM	Não sugeriu técnicas
[A19]	ScienceDirect	Análise de Fluxo de Dados
[A30]	IEEE	Outras técnicas
[A32]	Scopus	Não sugeriu técnicas
[A49]	IEEE	Não sugeriu técnicas

5. CONSIDERAÇÕES FINAIS

Neste artigo, foram apresentados os resultados obtidos com a execução de uma RSL para encontrar na literatura trabalhos que propuseram técnicas de detecção de código morto. Após a seleção secundária, foram obtidos 53 artigos considerados relevantes para o trabalho. Análises qualitativas foram realizadas para responder a questão de pesquisa.

Apesar de não existir uma tendência de aumento das pesquisas em relação a código morto, foi possível notar que esse tema foi constantemente estudado nos últimos anos. Isso mostra que há lacunas na área que motivam os pesquisadores a estudarem código morto em sistemas de informação. Além disso, foi possível notar que a maioria dos artigos encontrados foram publicados em *ACM SIGPLAN Notices*, podendo ser a primeira fonte de busca para pesquisadores interessados em pesquisar o assunto.

A AFD e a AA são as técnicas de detecção de código morto mais referenciadas, sendo mais adequadas para detecção de código que produz resultados não utilizados e código não executado, respectivamente. Além disso, foram encontradas outras técnicas menos utilizadas, tais como, análise de anotações de código e formulações matemáticas.

Existem poucos estudos cujo objetivo principal é analisar código morto. Apesar da quantidade de artigos selecionados na RSL, aproximadamente metade dos estudos tratavam somente desse assunto. Os outros estudos propuseram técnicas de análise que poderiam ser utilizadas em otimizações diversas, inclusive detecção de código morto, indicando possível carência de pesquisas realizadas de maneira exclusiva nessa área. Além disso, com a análise quantitativa, foi possível analisar os autores mais importantes e conferências e *journals* que possuem trabalhos relacionados ao assunto.

Como sugestões de trabalhos futuros, podem ser desenvolvidas abordagens que utilizam técnicas de detecção de código morto combinadas com técnicas de visualização de software. As técnicas de visualização podem ser úteis para facilitar localização do código morto nos sistemas de informação, assim como auxiliar na compreensão da quantidade de poluição desses sistemas.

6. REFERÊNCIAS

[1] Beyer, D.; Chlipala, A. J.; Henzinger, T. A.; Jhala, R.; Majumdar, R. Generating Tests from Counterexamples. In: International Conference on Software Engineering. pp. 326-335. 2004.

[2] Biolchini, J. C. A.; Mian, P. G.; Natali, A. C.; Conte, T. U.; Travassos, G. H. Scientific Research Ontology to Support Systematic Review in Software Engineering. In: Advanced Engineering Informatics. pp. 133-151. 2007.

[3] Boomsma, H.; Gross, H.G. Dead Code Elimination for Web Systems Written in PHP: Lessons Learned from an Industry Case. In: International Conference of Software Maintenance. pp. 511-515. 2012.

[4] Burd, L.; Rank, S. Using Automated Source Code Analysis for Software Evolution. In: IEEE International Workshop on Source Code Analysis and Manipulation. pp. 204-210. 2001.

[5] Eder, S.; Junker, M.; Jürgens, E.; Hauptmann, B.; Vaas, R.; Prommer, K. How Much Does Unused Code Matter for Maintenance?. In: International Conference on Software Engineering. pp. 1102-1111. 2012.

[6] Gold, N.; Mohan, A. A Framework for Understanding Conceptual Changes in Evolving Source Code. In: International Conference on Software Maintenance. pp. 431-439. 2003.

[7] Guerrouat, A.; Richter, H. A Combined Approach for Reachability Analysis. In: International Conference on Software Engineering Advances. pp. 23-23. 2006.

[8] Knoop, J.; Rüthing, O.; Steffen B. Partial Dead Code Elimination. In: Conference on Programming Language Design and Implementation v. 29, pp. 147-158 1994.

[9] Lehman, M. M.; Belady, L. Program Evolution: Processes of Software Change. Academic Press. 538p. 1985.

[10] Pressman, R.; Maxim, B. Software Engineering: A Practitioner's Approach. McGraw-Hill. 976p. 2014.

[11] Scanniello, G. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In: Conference on Software Engineering and Advanced Applications. pp. 392-397. 2014.

[12] Scanniello, G. Source Code Survival with the Kaplan Meier. In: International Conference on Software Maintenance. pp. 524-527. 2011.

APÊNDICE - ARTIGOS SELECIONADOS

[A1]	Chen, Y.; Emden, R. G.; Eleftherios, K. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In: IEEE Transactions on Software Engineering. v. 24, n. 9, p. 682-694. 1998.
[A2]	Guerrouat, A.; Harald R. A Combined Approach for Reachability Analysis. In: International Conference on Software Engineering Advances. p.23. 2006.
[A3]	Dai, X., Zhai, A., Hsu, W. C., Yew, P. C. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In: International Symposium on Code Generation and Optimization. pp. 280-290. 2005.
[A4]	Sunitha, K. V. N.; Kumar. V. V. A New Technique for Copy Propagation and Dead Code Elimination Using Hash Based Value Numbering. In: Advanced Computing and Communications. pp. 601-604. 2006.
[A5]	Fuhs, J.; James Cannady. An Automated Approach in Reverse Engineering Java Applications Using Petri Nets. In: SoutheastCon. pp. 90-96. 2004.
[A6]	Beyer, D., Henzinger, T., Jhala, R., Majumdar, R. An Eclipse Plug-in for Model Checking. In: IEEE International Workshop on Program Comprehension. pp. 251-255. 2004.
[A7]	Tempero, E. An Empirical Study of Unused Design Decisions in Open Source Java Software. In: Software Engineering Conference. pp. 33-40. 2008.
[A8]	Behera, C. K.; Pawan, K. An Improved Algorithm for Loop Dead Optimization. In: ACM SIGPLAN Notices. v. 41, n. 5, pp. 11-20. 2006.
[A9]	Scanniello, G. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In: Conference on Software Engineering and Advanced Applications. pp. 392-397. 2014.
[A10]	Davis, I. J., Godfrey, M. W., Holt, R. C., Mankovskii, S., Minchenko, N. Analyzing Assembler To Eliminate Dead Functions: An Industrial Experience. In: European Conference on Software Maintenance and Reengineering. pp. 467-470. 2012.
[A11]	Tardieu, O.; Stephen A. E. Approximate Reachability for Dead Code Elimination in Esterel. In: Automated Technology for Verification and Analysis. pp. 323-337. 2005.
[A12]	Knoop, J. Eliminating Partially Dead Code in Explicitly Parallel Programs. In: Theoretical Computer Science. v. 196, n. 1, p. 365-393. 1998.
[A13]	Geneves, P.; Layaïda, N. Equipping IDEs with XML-Path Reasoning Capabilities. In: ACM Transactions on Internet Technology. v. 13, n. 4, p. 13. 2013.
[A14]	Schäf, M.; Narbonne, D. S; Wies, T. Explaining Inconsistent Code. In: Joint Meeting on Foundations of Software Engineering. pp. 521-531. 2013.
[A15]	Chou, H; Chang, K.; Kuo, S. Facilitating Unreachable Code Diagnosis and Debugging. In: Design Automation Conference. pp. 485-490 .2011.
[A16]	Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., Majumdar, R. Generating Tests from Counterexamples. In: International Conference on Software Engineering. 2004.
[A17]	Eder, S., Junker, M., Jürgens, E., Hauptmann, B., Vaas, R., Prommer, K. H. How Much Does Unused Code Matter for Maintenance?. In: International Conference on Software Engineering. pp. 1102-1111. 2012.
[A18]	Ogawa, M.; Zhenjiang H.; Sasano, I. Iterative-Free Program Analysis. In: International Conference on Functional Programming. p. 111-123. 2003.
[A19]	Stone, A.; Strout, M.; Behere, S. May/Must Analysis and the DFAGen Data-Flow Analysis Generator. In: Information and Software Technology. v. 51, n. 10, p. 1440-1453. 2009.
[A20]	Jensen, S. H., Madsen, M.; Moller, A. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In: European Conference on Foundations of Software Engineering. pp. 59-69. 2011.
[A21]	Knoop, J.; Rüthing, O.; Steffen, B. Partial Dead Code Elimination. In: Conference on Programming Language Design and Implementation. v. 29, n. 6, pp. 147-158. 1994.
[A22]	Xue, J.; Cai, Q.; Gao, L. Partial Dead Code Elimination on Predicated Code Regions. In: Software: Practice and Experience. v. 36, n. 15, p. 1655-1685. 2006.
[A23]	Barros, J. B., Da Cruz, D., Henriques, P. R., Pinto, J. S. Assertion-Based Slicing and Slice Graphs. In: Formal Aspects of Computing. v. 24, n. 2, p. 217-248. 2012.
[A24]	Damiani, F.; Giannini, P. Automatic Useless-Code Elimination for HOT Functional Programs. In: Journal of Functional programming. v. 10, n. 06, p. 509-559. 2000.
[A25]	Lee, J.; David A. P.; Samuel P. M. Basic Compiler Algorithms for Parallel Programs. In: ACM SIGPLAN Notices. pp.1-12.1999.
[A26]	Nguyen, Q. H.; Scholz, B. Computing SSA Form with Matrices. In: Electronic Notes in Theoretical Computer Science. v. 190, n. 1, p. 121-132. 2007.
[A27]	Novillo, D. R. U.; Schaeffer, J. Concurrent SSA Form in the Presence of Mutual Exclusion. In: IEEE Parallel Processing. pp. 356-364. 1998.
[A28]	Wand, M.; Siveroni, I. Constraint Systems for Useless Variable Elimination. In: Symposium on Principles of Programming Languages. pp. 291-302. 1999.
[A29]	El-Zawawy, M. A. Dead Code Elimination Based Pointer Analysis for Multithreaded Programs. In: Journal of the Egyptian Mathematical Society. v. 20, n. 1, pp. 28-37. 2012.
[A30]	Boomsma, H.; Gross, H. G. Dead Code Elimination for Web Systems Written in PHP: Lessons Learned from an Industry Case. In: International Conference of Software Maintenance. pp. 511-515. 2012.
[A31]	Kim, K.; Kim, J.; Yoo, W. Dead Code Elimination in CTOC. In: International Conference on Software Engineering Research, Management & Applications. pp. 584-588. 2007.
[A32]	Chabbi, M.; Mellor-Crummey, J. Deadspy: A Tool to Pinpoint Program Inefficiencies. In: International Symposium on Code Generation and Optimization. pp. 124-134. 2012.
[A33]	Takimoto, M. Demand-driven Partial Dead Code Elimination. In: Information and Media Technologies. v. 5, n. 0, pp. 79-86. 2012.
[A34]	Fernandes, T, Desharnais, J. Describing Data Flow Analysis Techniques with Kleene Algebra. In: Science of Computer Programming. v. 65, n. 2, pp. 173-194. 2007.
[A35]	Tomb, A, Flanagan, C. Detecting Inconsistencies Via Universal Reachability Analysis. In: International Symposium on Software Testing and Analysis. pp. 287-297. 2012.
[A36]	Bodik, R.; Gupta, R. Partial Dead Code Elimination Using Slicing Transformations. In: ACM SIGPLAN Notices. pp. 159-170. 1997.
[A37]	Gupta, R.; Benson, D. A.; Fang, J. Z. Path Profile Guided Partial Dead Code Elimination Using Predication. In: International Conference on Parallel Architectures and Compilation Techniques. pp. 102-113. 1997.
[A38]	Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., Streeter, D. Practical Extraction Techniques for Java. In: ACM Transactions on Programming Languages and Systems. v. 24, n. 6, p. 625-666. 2002.
[A39]	Saabas, A.; Uustalu, T. Program and Proof Optimizations with Type Systems. In: The Journal of Logic and Algebraic Programming. v. 77, n. 1, p. 131-154. 2008.
[A40]	Janota, M.; Grigore, R.; Moskal, M. Reachability Analysis for Annotated Code. In: Specification and Verification of Component-Based Systems. pp. 23-30. 2007.
[A41]	Cai, Q., Gao, L. Xue, J. Region-Based Partial Dead Code Elimination on Predicated Code. In: Compiler Construction. Springer Berlin Heidelberg. pp. 150-166. 2004.
[A42]	Gupta, R.; Berson, D. A.; Fang, J. Z. Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization. In: International Symposium on Microarchitecture. pp. 358-368. 1997.
[A43]	Benton, N. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In: ACM SIGPLAN Notices. v. 39, n.1, pp. 14-25. 2004.
[A44]	Wagner, G.; Gal, A.; Franz, M. "Slimming" a Java Virtual Machine by Way of Cold Code Removal and Optimistic Partial Program Loading. In: Science of Computer Programming. v. 76, n. 11, p. 1037-1053. 2011.
[A45]	Scanniello, G. Source Code Survival with the Kaplan Meier. In: International Conference of Software Maintenance. pp. 524-527. 2011.
[A46]	Payet, É; Spoto, F. Static Analysis of Android Programs. In: Information and Software Technology. v. 54, n. 11, p. 1192-120. 2012.
[A47]	Lokuciejewski, P.; Kelter, T.; Marwedel, P. Superblock-Based Source Code Optimizations for WCET Reduction. In: International Conference on Computer and Information Technology. pp. 1918-1925. 2010.
[A48]	Gutzmann, T.; Lundberg, J.; Lowe, W. Towards Path-Sensitive Points-to Analysis. In: International Conference on Source Code Analysis and Manipulation. p. 59-68. 2007.
[A49]	Su, L.; Lipasti, M. Dynamic Class Hierarchy Mutation. In: International Symposium on Code Generation and Optimization. pp. 98-110. 2006.
[A50]	Liu, Y. A.; Stoller, S. Eliminating Dead Code on Recursive Data. In: Static Analysis. Springer Berlin Heidelberg. pp. 211-231. 1999
[A51]	Genevès, P.; Layaïda, N. Eliminating Dead-Code from XQuery Programs. In: International Conference on Software Engineering. pp. 305-306. 2010.
[A52]	Saabas, A.; Uustalu, T. Type Systems for Optimizing Stack-Based Code. In: Electronic Notes in Theoretical Computer Science. v.190,n.1,pp.103-119. 2007.
[A53]	Jantz, M. J.; Kulkarni, P. A. Understand and Categorize Dynamically Dead Instructions for Contemporary Architectures. In: Workshop on Interaction between Compilers and Computer Architectures. pp. 25-32. 2012.