Victor Hugo Faria Dias Magalhães

Avaliação de ferramentas de análise estática de código para a linguagem Go

Projeto de monografia requisitada na disciplina de Monografia em Sistemas de Informação 1 do Bacharelado de Sistemas de Informação UFMG

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte, Minas Gerais
2023

Sumário

1 Introdução

A Análise estática de código é uma técnica que se baseia em analisar o código-fonte de um *software* sem executá-lo [Babati et al. 2017]. Apesar dos estudos demonstrando vantagens do uso de ferramentas de análise estática de código durante o processo de desenvolvimento, pessoas desenvolvedoras [Johnson et al. 2013] afirmam que as ferramentas atuais podem não oferecer informações suficientes para que os *problemas* que elas relatam sejam solucionados. Definem-se como *problemas*, erros relacionados a padrões de formatação da linguagem, erros lógico/funcionais, vazamento de recursos, falhas de segurança entre outros [Emanuelsson e Nilsson 2008].

Go é uma linguagem de programação *open-source* multiparadigma desenvolvida pela *Google* e disponiblizada para o público geral em 2009, com os objetivos de ser rápida, eficiente, confiável e simples de desenvolver [McGrath 2020]. *Go* conta com ferramentas de análise estática de código nativas da linguagem, como as ferramentas *go fmt* e *go vet*. Ambas as ferramentas utilizam análise estática do código fonte para validar estilo e heurísticas de qualidade respectivamente [Khatchadourian et al. 2022].

Embora conte com as facilidades das ferramentas de análise estática de código incluídas por padrão na linguagem, vê-se no geral que o uso de tais ferramentas não elimina a existência de *problemas* [Emanuelsson e Nilsson 2008]. Para além disso, algumas propriedades verificadas por essas ferramentas são impossíveis de serem decididas com total certeza, o que torna análises estáticas intrinsicamente imprecisas [Emanuelsson e Nilsson 2008].

Apesar desses pontos, análises estáticas são amplamente difundidas dado o baixo custo de execução e a elevada velocidade em relação a análises totalmente manuais feitas por uma pessoa desenvolvedora [Lavazza, Morasca e Tosi 2021]. Análises estáticas têm a possibilidade de serem poderosas ferramentas para auxiliar durante o desenvolvimento, acelerando as análises manuais e possibilitando que código seja entregue mais rápido e com mais qualidade. Qualidade de código tendo um custo muito importante na indústria de informática: a título de ilustração, em 2020, o Custo da Baixa Qualidade de Software (do inglês, *Cost of Poor Software Quality* -

CPSQ) foi de US$ 2,08 trilhões [Krasner 2021].

Isso posto, ainda que a linguagem *Go* seja relativamente nova, ela já conta com um ecossistema extenso [McGrath 2020], o que permite análises qualitativas e quantitativas de diversos aspectos e propriedades de suas ferramentas de análise estática de código. Logo, surge a necessidade de decidir quais ferramentas são adequadas para diferentes contextos. Desta forma, times de desenvolvimento podem evitar o esforço de decidir entre inúmeros tipos de ferramentas cujas funcionalidades por vezes se sobrepõe e outras vezes se complementam, mas que não foram comparadas.

1.1 Objetivos Gerais

O objetivo deste trabalho de Monografia de Sistemas de Informação I é contribuir para a literatura oferecendo um maior entendimento sobre as diferenças e recursos das ferramentas de análise estática para *Go*. Isso pretende ser efetuado através da comparação de recursos disponíveis em cada uma das ferramentas, utilizando técnicas de comparação para ferramentas de análise estática de código já disponíveis na literatura.

1.2 Objetivos Específicos

Os objetivos específicos desta Monografia em Sistemas de Informação I consistem na realização de uma análise qualitativa e quantitativa de diferentes aspectos das ferramentas de análise estática disponíveis no ecossistema da linguagem *Go*. Reitera-se que o objetivo não é obter um *rank* ou elencar melhores ou piores ferramentas. Esse tipo de análise não está no escopo deste trabalho e também é impossível de ser feita de forma objetiva, dado o caráter subjetivo para escolha das ferramentas que depende das necessidades dos times de desenvolvimento, além das diferenças em funcionalidades que as ferramentas oferecem. Com os resultados desta comparação, espera-se que seja possível obter um critério de comparação sólido e passível de repetição, que possa ser utilizado para comparar ferramentas de análise estática para a linguagem *Go*.

2 Referencial Teórico

2.1 Linguagem *Go*

O ecossistema da linguagem *Go* é extenso e conta com múltiplas ferramentas e participação da comunidade que cresce a cada dia [McGrath 2020]. Dado o caráter performático e construído para programação concorrente [Pang 2016] é possível entender o crescimento em popularidade da linguagem, se tornando por exemplo, a ferramenta de programação mais utilizada por profissionais de *DevOps* em 2021 [Dada et al. 2022]. A linguagem *Go* conta com facilidades construídas por padrão na linguagem, como por exemplo: sintaxe simples, suporte à concorrência, sistema de tipos estáticos, coleta de lixo e segurança de memória (ou do inglês, *memory safety*) [McGrath 2020].

2.2 Análise estática de código

Análise estática de código permite que validações e métricas sejam realizadas e obtidas sem a execução do programa. O processo de analisar estaticamente um programa não só cobre falhas de análise e operação dos compiladores como também se mostra uma poderosa ferramenta para detectar irregularidades, problemas de padronização e possíveis defeitos [Stefanović et al. 2020]. Ferramentas de análise estática são em síntese, programas que analisam o código fonte de outros programas e conseguem atuar em problemas visíveis no código [Stefanović et al. 2020].

Figura 1 – Tipos de problemas em código - Fonte: [Stefanović et al. 2020]

Diferentes tipos de análises podem ser realizadas para encontrar diferentes tipos de *problemas*. Abaixo exemplos de categorias de regras para as quais ferramentas comerciais conseguem encontrar problemas [Novak, Krajnc e Zontar 2010]:

- Problemas sintáticos
- Código-fonte inacessível
- Variáveis não declaradas
- Variáveis não inicializadas
- Funções e procedimentos não utilizados
- Variáveis utilizadas antes da inicialização
- Não utilização de valores de funções
- Uso incorreto de ponteiros
- Concorrência

- Más práticas
- Interface do Usuário (do inglês, *User Interface, UI*)
- Corretude
- Design
- Exceções
- Interoperabilidade
- Manutenibilidade
- Nomenclatura
- Desempenho
- Portabilidade
- Segurança
- Serialização

Muitas ferramentas implementam análises que cobrem múltiplas destas classes de problemas, o que exige uma análise sistemática que leve em consideração a taxonomia das ferramentas, analisando não apenas as funcionalidades, mas também seu contexto de manutenção, usabilidade, integração, fornecimento de resultados entre outras características intrínsicas às ferramentas [Novak, Krajnc e Zontar 2010].

2.2.1 Regras de análise estática de código

Regras de análise estática de código são definições objetivas ou heurísticas, que definem um grupo de características de um código fonte como problemático. Ao longo dos anos, essas regras foram construídas por comunidades e empresas, em um processo altamente manual e que consome muito tempo [Garg 2022]. Existem inúmeras regras adotadas por diversas ferramentas, porém, é possível categorizar boa parte das regras em um conjunto de 9 itens [Novak, Krajnc e Zontar 2010]:

1. Estilo - inspeciona a aparência visual do código-fonte.

2. Nomenclatura - revisão para verificar se as variáveis estão corretamente nomeadas (ortografia, padrões de nomenclatura, ...).
3. Geral - regras gerais da análise de código estático, incluindo detecção de *bugs* não relacionados a outros itens.
4. Concorrência - erros com a execução de código concorrente.
5. Exceções - erros ao lançar ou não lançar exceções.
6. Desempenho - erros com o desempenho das aplicações.
7. Interoperabilidade - erros com o comportamento comum entre partes do código.
8. Segurança - erros que podem afetar a segurança da aplicação.
9. Manutenibilidade - regras para melhorar a manutenibilidade da aplicação.

3 Metodologia

Primeiramente, é necessária a seleção de um grupo de ferramentas de análise estática para *Go*. Planeja-se utilizar ferramentas de código aberto (do inglês, *open-source*) disponíveis no mercado que sejam populares entre as pessoas desenvolvedoras, de forma independente de análise prévia de suas funcionalidades. A escolha das ferramentas planeja ser orientada pelas métricas de popularidade de portais de código aberto, como o *Github* ou *Gitlab*. Baseando-se nesses pontos, levantou-se uma lista de ferramentas: *Staticcheck*, *goreporter*, *gokart*, *revive*, *govulncheck*, *prealloc*, e *Chronos*.

Após a obtenção do grupo de ferramentas a serem avaliadas, pretende-se realizar uma análise taxonômica [Novak, Krajnc e Zontar 2010] das ferramentas e uma análise comparativa [Emanuelsson e Nilsson 2008] das funcionalidades das ferramentas.

Planeja-se realizar a análise taxonômica construindo uma versão adaptada para o contexto das ferramentas da linguagem *Go* de uma árvore de taxonomia presente em outros trabalhos da literatura [Novak, Krajnc e Zontar 2010], para cada uma das ferramentas selecionadas. A construção da árvore de taxonomia depende de um processo de definição das categorias a partir das quais as ferramentas serão avaliadas.

Para além disso, a análise comparativa se baseará nas funcionalidades das ferramentas. Planeja-se listar em uma tabela as ferramentas e as funcionalidades presentes em todas, assinalando quais funcionalidades qual ferramenta apresenta. Nesta análise planeja-se incluir informações quantitativas sobre as ferramentas, como número de *downloads* e número de *estrelas* (métrica que informa quantas pessoas favoritaram um repositório) no *Github* ou *Gitlab*.

Por fim, planeja-se executar as ferramentas no projeto *open-source* feito em *Go* chamado *tsuru* utilizando um ambiente controlado, e mensurar tempo de execução, e uso de recursos (utilização de *CPU* e consumo de memória *RAM*).

O *tsuru* é um projeto *open-source* de plataforma como serviço (do inglês *Platform as a Service - PaaS*), que tem como objetivo facilitar agilizar o processo de

disponibilizar uma aplicação, abstraindo o processo de configuração de servidores. O projeto foi escolhido por ser *open-source*, pela sua popularidade, por contar com atualizações frequentes e ser escrito totalmente em *Go*.

O ambiente controlado é composto por uma máquina virtual disponível no ambiente de *cloud* da *Oracle*, considerado um ambiente confiável e com documentação interessante [Safonov 2016]. As especificações da máquina virtual *cloud* são as seguintes:

- Processador *Ampere Altra* 1 *Core* de 3GHz,
- 4GB RAM,
- *Canonical Ubuntu* 20.04 Minimal

O uso da máquina virtual no ambiente *cloud* é interessante devido ao isolamento de ambientes externos e baixo compatilhamento de recursos com outras aplicações. Após a instalação de todas as ferramentas na máquina virtual, o código fonte do *tsuru* foi baixado e cada uma das ferramentas foi executada, de forma a analisá-lo estaticamente por completo.

Para realizar as medições, foi utilizado o software de *shell zsh* e sua extensão, o comando *time*. O comando *time* do *zsh* permite diversas medições e customizações. Entre elas, a possibilidade de medir os tempos de execução, o uso de memória e de *CPU* por algum processo. O *zsh::time* mede as variáveis necessárias, a partir das seguintes definições:

$$u = Tempo\ gasto\ pelo\ processo\ no\ modo\ de\ usuário$$

$$s = Tempo\ gasto\ pelo\ processo\ no\ modo\ de\ sistema$$

$$e = Tempo\ total\ gasto\ pelo\ processo$$

$$m = Memória\ máxima\ gasta\ pelo\ processo\ em\ kB$$

$$p = porcentagem\ de\ CPU\ 100 * (u + s)/e$$

Nota-se que dada a definição do cálculo da porcentagem de *CPU*, o resultado pode exceder 100%, e serve para fins de comparação entre as ferramentas.

4 Resultados

Para a construção das árvores taxonômicas, realizou-se a análise da documentação dos sete projetos. A partir disso foram levantadas as categorias que são representadas pelos galhos das árvores taxonômicas [Novak, Krajnc e Zontar 2010]. A partir do levantamento das documentações, obtiveram-se as figuras 2 a 8:

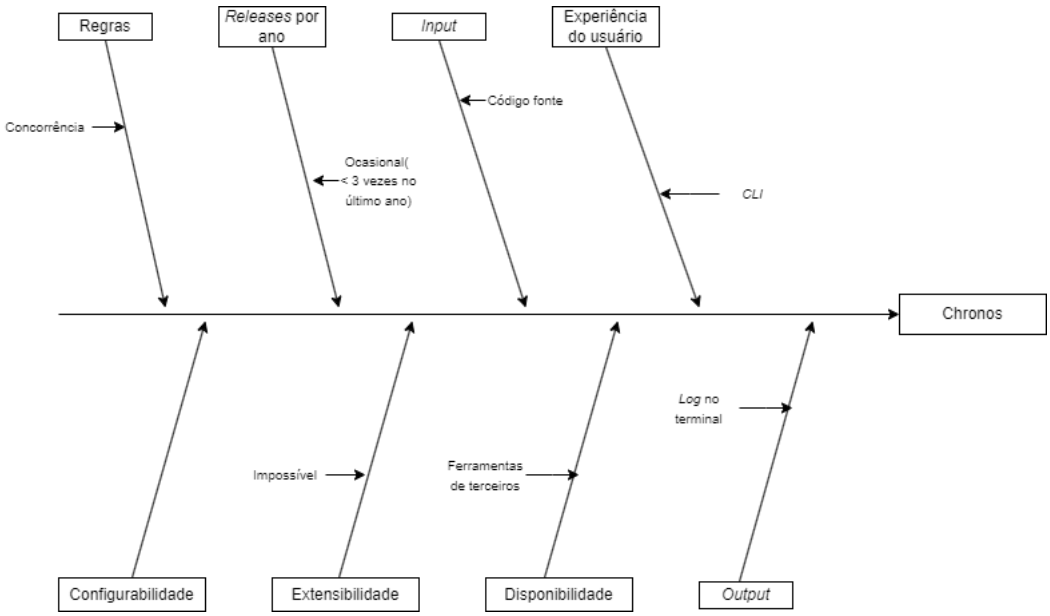

Figura 2 – Árvore taxonômica *Chronos* - Figura autoral

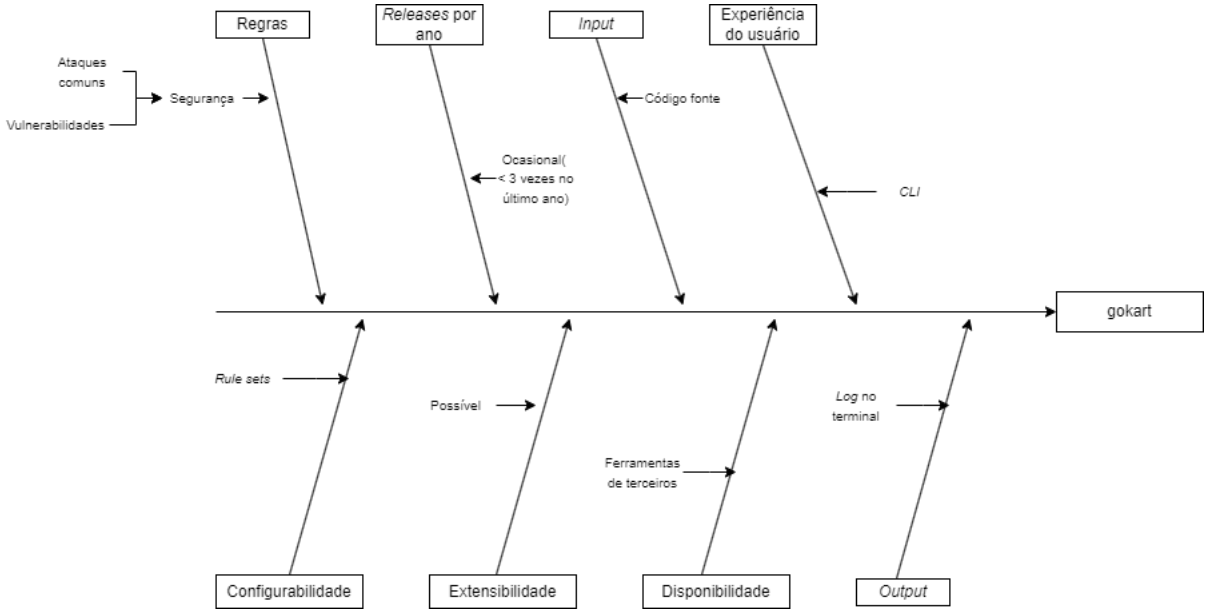

Figura 3 – Árvore taxonômica *gokart* - Figura autoral

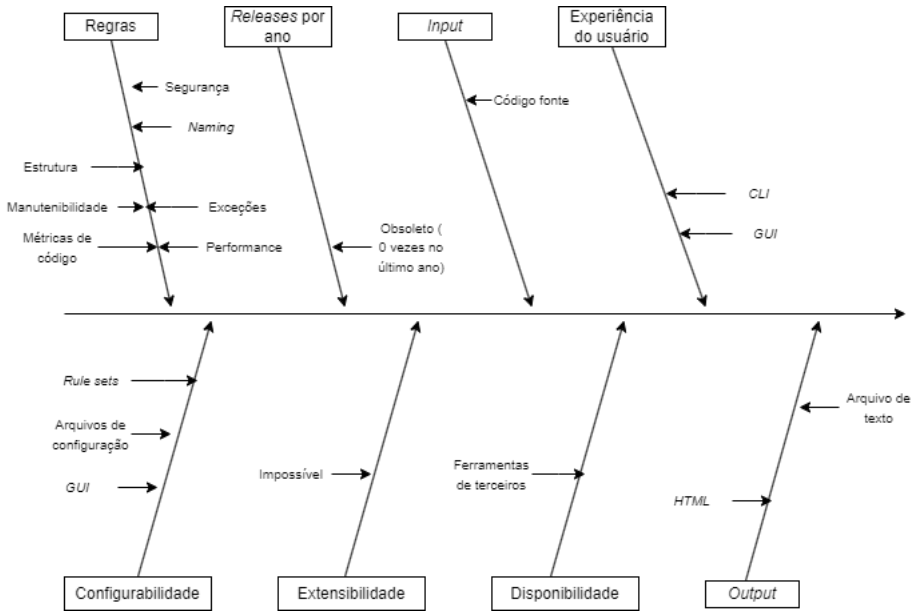

Figura 4 – Árvore taxonômica *goreporter* - Figura autoral

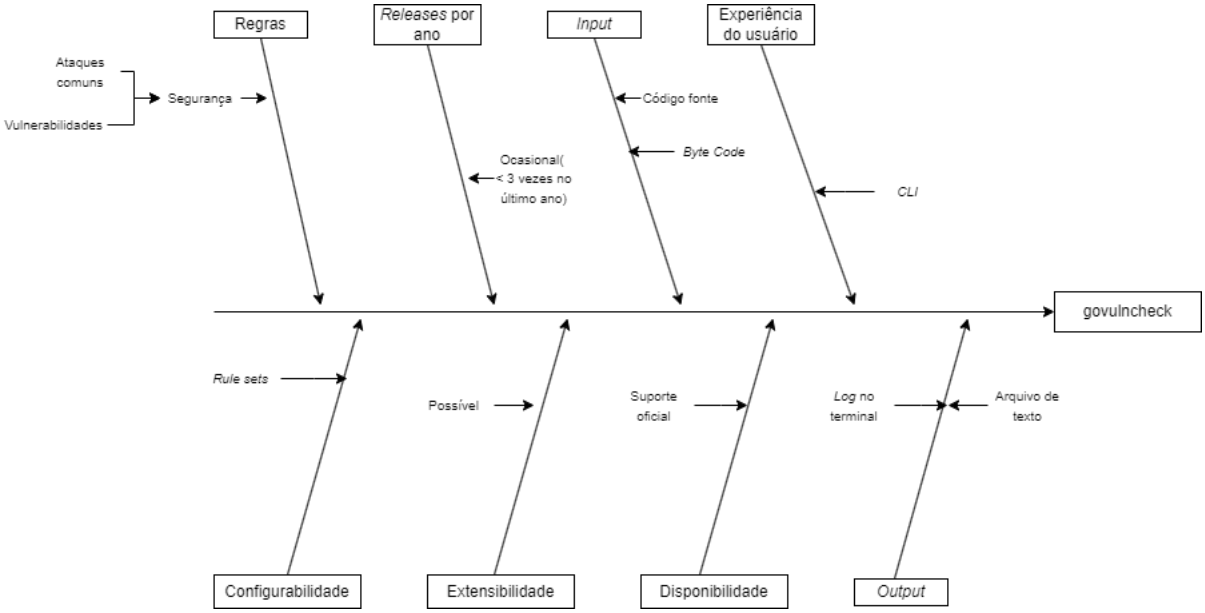

Figura 5 – Árvore taxonômica *govulncheck* - Figura autoral

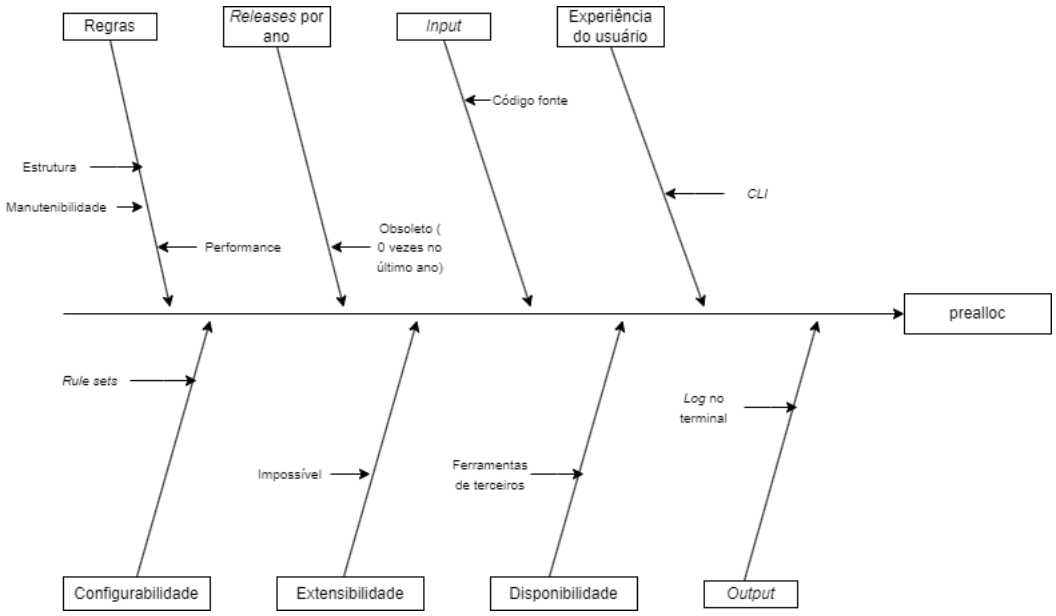

Figura 6 – Árvore taxonômica *prealloc* - Figura autoral

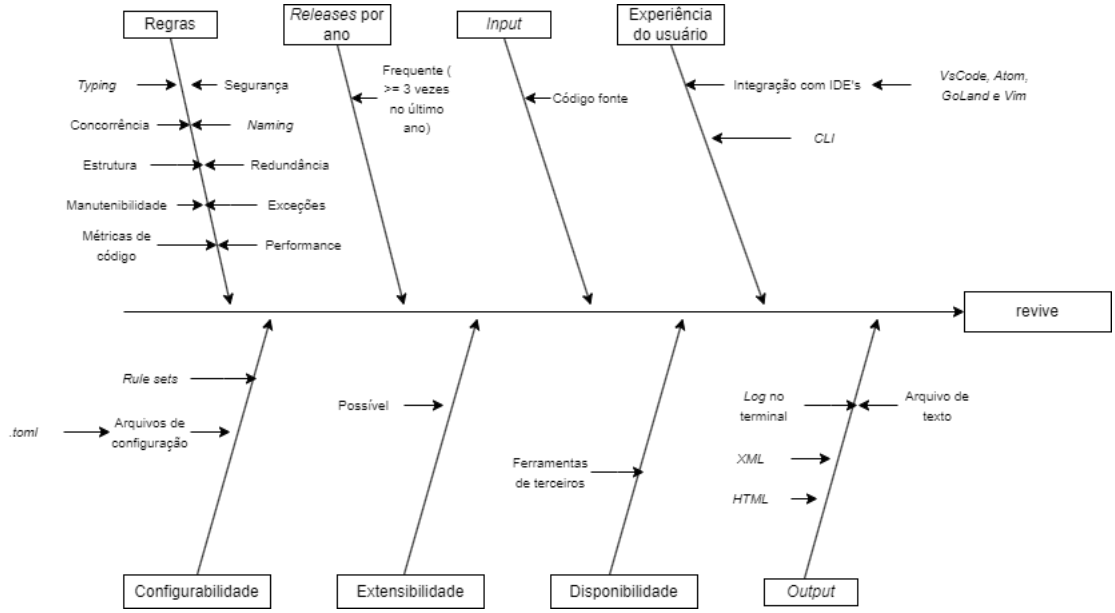

Figura 7 – Árvore taxonômica *revive* - Figura autoral

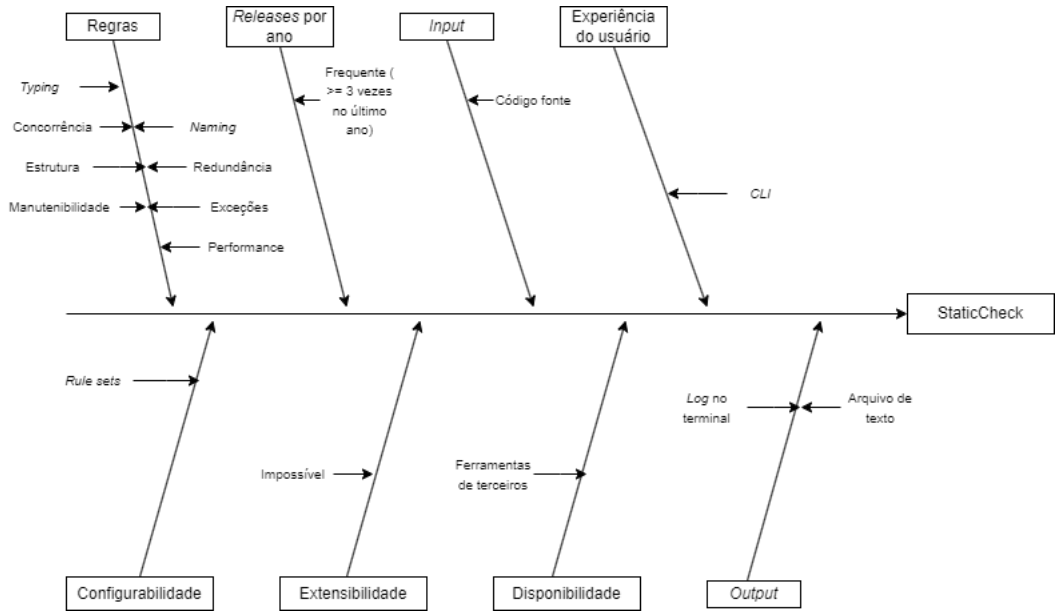

Figura 8 – Árvore taxonômica *Staticcheck* - Figura autoral

É possível perceber que as árvores taxonômicas apresentam uma estrutura visual parecida, mesmo entre ferramentas que compartilham poucas semelhanças. O resultado é evidenciado mais ainda entre ferramentas que possuem um número maior de características semelhantes, como é evidenciado entre a figura 3 e a figura 5. Cabe ressaltar que a semelhança visual oferece um *framework* de comparação intuitivo, que independe de acessar as diferentes fontes de documentação entre as ferramentas.

Para obter os resultados da análise comparativa presentes na tabela 2, foram utilizados os dados presentes nas documentações oficiais de cada uma das ferramentas, além de dados obtidos através de medições realizadas em um ambiente controlado. Utilizando o comando *time* do software de shell *zsh* foi possível obter os dados utilizados para preencher a tabela 2, o *output* das medições encontra-se na tabela 1.

Ferramenta	Comando	Resultado
Chronos	chronos –mod ./	user time: 4.22s system time: 7.18s real time: 8.24s total cpu: 259% max memory: 1568 kB
Gokart	gokart scan	user time: 8.58s system time: 10.28s real time: 20.37s total cpu: 92% max memory: 338 kB
Goreporter	goreporter -p .	user time: 2.04s system time: 4.40s real time: 15.73s total cpu: 40% max memory: 99 kB
Govulncheck	govulncheck ./...	user time: 1.14s system time: 0.94s real time: 3.82s total cpu: 263% max memory: 711 kB
Prealloc	prealloc ./...	user time: 0.03s system time: 0.00s real time: 0.10s total cpu: 34% max memory: 10 kB
Revive	revive	user time: 0.08s system time: 0.12s real time: 0.27s total cpu: 76% max memory: 36 kB
Staticcheck	staticcheck ./...	user time: 1.21s system time: 1.42s real time: 5.41s total cpu: 251% max memory: 192 kB

Tabela 1 – Tabela de resultados

-	*Chronos*	*gokart*	*goreporter*	*go vuln-check*	*prealloc*	*revive*	*Staticcheck*
Race conditions	Sim	Não	Não	Não	Não	Sim	Sim
Estilo de codificação	Não	Não	Sim	Não	Não	Sim	Sim
Tipagem	Não	Não	Sim	Não	Não	Sim	Sim
Má ordem de retorno	Não	Não	Sim	Não	Sim	Sim	Não
Strings	Não	Sim	Sim	Não	Não	Sim	Sim
Memory leaks	Não	Sim	Sim	Não	Não	Sim	Sim
Variáveis não inicializadas	Não	Não	Sim	Não	Sim	Sim	Sim
Não terminação	Sim	Não	Não	Não	Não	Sim	Sim
Extensível	Não	Sim	Não	Sim	Não	Sim	Não
Métricas	Não	Não	Sim	Não	Não	Sim	Não
Vulnerabilidades	Não	Sim	Não	Sim	Não	Não	Não
Riscos de segurança	Não	Sim	Sim	Sim	Sim	Sim	Não
Nomenclatura	Não	Não	Sim	Não	Não	Sim	Sim
Desempenho	Não	Não	Sim	Não	Sim	Sim	Sim
Concorrência	Sim	Não	Não	Não	Não	Sim	Sim
Versão do Golang	>= 1.15	>= 1.19	>= 1.6	>= 1.18	>= 1.15	>= 1.19	>= 1.19
Estrelas no Github	404	2100	3100	280	530	4200	5400
Forks no Github	13	108	275	43	23	251	340
Tempo de execução	8.24s	20.37s	15.73s	3.82s	0.1s	0.27s	5.41s
RAM (máx)	1568kB	338kB	99kB	711kB	10kB	36kB	192kB
CPU utilizada (%)	259%	92%	40%	263%	34%	76%	251%

Tabela 2 – Comparação entre funcionalidades

5 Conclusão

Ferramentas de análise estática são parte extensa do ecossistema de diversas linguagens, da lingugagem *Go* inclusa, e oferecem diversos recursos durante o processo de desenvolvimento de *software*, sendo importante a comparação entre os requisitos e funcionalidades das diversas ferramentas para os diferentes contextos de desenvolvimento. Para isso, construiu-se neste trabalho uma avaliação sistemática das ferramentas utilizando e adaptando técnicas de comparação existentes na literatura. A análise taxonômica proporcionou uma visão abrangente e estruturada das ferramentas, facilitando uma comparação visual e intuitiva entre suas funcionalidades. Além disso, a adaptação das árvores taxonômicas às características específicas da linguagem *Go* contribuiu para a contextualização das ferramentas dentro desse ecossistema, considerando suas peculiaridades e requisitos. Ademais, a construção da tabela de comparação se provou complementar às árvores taxonômicas, permitindo a adição de informações quantitativas, algumas extraídas da execução real das ferramentas em um ambiente controlado, garantindo a imparcialidade e qualidade das medições. Com os resultados obtidos, conclui-se que apesar da falta de verificação empírica com times de desenvolvimento, as abordagens existentes na literatura de análise comparativa e análise taxonômica são válidas e úteis para a comparação das ferramentas de análise estática para o ecossistema da linguagem *Go*.

Referências

[Babati et al. 2017]BABATI, B. et al. Static analysis toolset with clang. In: *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*. [S.l.: s.n.], 2017. p. 23–29.

[Dada et al. 2022]DADA, O. A. et al. Hidden gold for it professionals, educators, and students: Insights from stack overflow survey. *IEEE Transactions on Computational Social Systems*, IEEE, 2022.

[Emanuelsson e Nilsson 2008]EMANUELSSON, P.; NILSSON, U. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, Elsevier, v. 217, p. 5–21, 2008.

[Garg 2022]GARG, P. Example-based synthesis of static analysis rules. *arXiv preprint arXiv:2204.08643*, 2022.

[Johnson et al. 2013]JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: IEEE. *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.], 2013. p. 672–681.

[Khatchadourian et al. 2022]KHATCHADOURIAN, R. et al. How many mutex bugs can a simple analysis find in go programs? 2022.

[Krasner 2021]KRASNER, H. The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021.

[Lavazza, Morasca e Tosi 2021]LAVAZZA, L.; MORASCA, S.; TOSI, D. Comparing static analysis and code smells as defect predictors: an empirical study. In: SPRINGER. *Open Source Systems: 17th IFIP WG 2.13 International Conference, OSS 2021, Virtual Event, May 12–13, 2021, Proceedings 17*. [S.l.], 2021. p. 1–15.

[McGrath 2020]MCGRATH, M. *GO Programming in easy steps: Discover Google's Go language (golang)*. [S.l.]: In Easy Steps Limited, 2020.

[Novak, Krajnc e Zontar 2010]NOVAK, J.; KRAJNC, A.; ZONTAR, R. Taxonomy of static code analysis tools. 01 2010.

[Pang 2016]PANG, J. Golang-planetary programming language. 2016.

[Safonov 2016]SAFONOV, V. O. *Trustworthy cloud computing*. [S.l.]: John Wiley & Sons, 2016.

[Stefanović et al. 2020]STEFANOVIĆ, D. et al. Static code analysis tools: A systematic literature review. *Annals of DAAAM & Proceedings*, v. 7, n. 1, 2020.