MCL: Uma Linguagem de Restrição Baseada em Medidas de Software

Alternative Title: MCL: Metrics-based Constraint Language

Christian Marlon Souza Couto
Universidade Federal de Lavras
Lavras, Minas Gerais
christiancouto@posgrad.ufla.br

Luana Almeida Martins
Universidade Federal de Lavras
Lavras, Minas Gerais
luana.martins1@posgrad.ufla.br

Heitor Costa
Universidade Federal de Lavras
Lavras, Minas Gerais
heitor@dcc.ufla.br

Ricardo Terra
Universidade Federal de Lavras
Lavras, Minas Gerais
terra@dcc.ufla.br

ABSTRACT

Software measures are underused due to the difficulty of interpreting their results and associating them to software quality. Different environments, languages, and development methodologies require specific measures and range of values. Thus, this paper proposes MCL (*Metrics-based Constraint Language*), a language that allows to specify, for different system components, the measures to be used and the expected range of values for each measure. We implemented a tool, called **MCLcheck**, to verify if a system conforms to the specified MCL restrictions and to report the detected violations. We explored different contexts of language usage through the **MyAppointments** system, demonstrating the applicability of MCL and its effectiveness as a language that provides support for preservation of quality factors, maintainability, and performance of information systems.

CCS CONCEPTS

• General and reference → Metrics; **• Software and its engineering → Constraint and logic languages**;

KEYWORDS

Software Measures; Constraint Language; Software Quality.

1 INTRODUÇÃO

Medidas de software proveem base quantitativa para o desenvolvimento e a validação de atributos de sistemas de informação. Entretanto, essas medidas são subutilizadas por causa da dificuldade de interpretar seus resultados e da sua associação à qualidade desses sistemas [14]. Existem diversas ferramentas que calculam essas medidas de forma automática (e.g., *Eclipse Metrics*[1] e *Analizo*[2]). No entanto, ferramentas como o *Analizo* fornecem apenas valores brutos, não permitindo especificar uma faixa de valores esperados para as medidas, o que dificulta a interpretação e a comparação dos valores. Outras como o *Eclipse Metrics* permitem especificar um intervalo de valores, porém a interpretação é dada por cores e/ou termos como "ruim" destacado de vermelho, o que dificulta o entendimento dos valores obtidos visto que não é perceptível o quanto deve-se melhorar para atingir o valor esperado para a métrica.

Dadas tais limitações, este artigo propõe uma linguagem de restrição baseada em medidas de software denominada MCL (*Metrics-based Constraint Language*), que permite especificar valores para um conjunto de medidas a serem aplicadas em sistemas de informação. A linguagem proposta foi implementada como **MCLcheck**, uma ferramenta responsável por verificar se um sistema está em conformidade com as restrições estabelecidas e reportar as violações encontradas.

Como exemplo, considera-se um arquiteto de software que objetiva diminuir os custos relacionados a manutenção de sistemas de informação atentando-se em fatores que podem influenciar negativamente a manutenibilidade, como coesão e acoplamento (Figura 1).

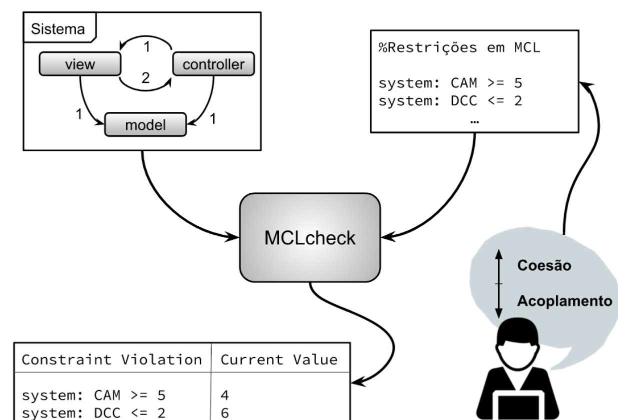


Figura 1: Estrutura das Restrições MCL

[1] Disponível em: https://sourceforge.net/projects/metrics/
[2] Disponível em: http://www.analizo.org/

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SBSI 2018, June 2018, Caxias do Sul, Rio Grande do Sul, Brazil
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Com isso, foram definidos que o acoplamento entre as classes do sistema será medido por meio da métrica DCC (*Design Class Coupling*) restringindo seu valor para menor ou igual a 2, e a coesão será medida por meio da métrica CAM (*Cohesion Among Methods*) restringindo seu valor para maior ou igual a 5. Dadas as restrições em MCL, a ferramenta `MCLcheck` realiza a verificação no sistema e reporta as violações. Nesse exemplo houve violação de ambas as restrições, visto que a métrica DCC obteve valor de 6 e CAM de 4.

O restante deste trabalho está organizado como a seguir. Os conceitos fundamentais ao estudo são introduzidos na Seção 2. A linguagem proposta e a ferramenta desenvolvida são apresentadas na Seção 3. A aplicação da MCL em diversos cenários é discutida na Seção 4. Trabalhos relacionados são abordados na Seção 5. Conclusão e trabalhos futuros são apresentadas na Seção 6.

2 BACKGROUND

2.1 Medidas de Software

Várias medidas[3] para avaliar sistemas de informação foram propostas para diferentes contextos de utilização [13]. Neste trabalho, 42 medidas para sistemas de informação orientados a objetos são analisadas (Tabela 1). Essas medidas podem ser organizadas em quatro conjuntos: i) QMOOD [1]: DSC, NOH, ANA, DAM, DCC, CAM, MOA, MFA, NOPM, CIS e NOM (são utilizadas em proporções diferentes para calcular REU, FLE, ENT, FUN, EXT e EFE); ii) CK [2]: DIT, NOC, LCOM, WMC, RFC e CBO; iii) medidas propostas por Martin [9]: CA, CE, RMA, RMI e RMD; e iv) medidas relacionadas a tamanho e complexidade: LOC, MLOC, NSC, NOI, NOP, NONM, NMI, NORM, NSM, NOF, NSF, PAR, SIX, SIX2, VG e NBD. Dessas medidas, duas não são calculadas pelo *Eclipse Metrics* (RFC e CBO) e, portanto, não são consideradas neste estudo. Porém, essas medidas são similares às medidas de outros conjuntos, tais como CIS que pode ser utilizada no lugar da RFC, e CA e CE que podem ser agrupadas e utilizadas no lugar da CBO.

[3] O termo *medida* é utilizado em decorrência da sua definição na norma ISO/IEC 25000 [7], na qual é considerada uma variável que recebe o valor resultante da medição.

2.2 Linguagem de Domínio Específico

A linguagem MCL pode ser classificada como uma DSL (*Domain-Specific Languages*) por ser uma linguagem de restrição baseada em medidas que atende a um domínio específico. DSLs são linguagens que proveem notações e construções adaptadas para determinado domínio de aplicação. Dessa forma, as DSLs proporcionam maior

Tabela 1: Medidas de Software

Sigla	Nome	Definição
ANA	*Average Number of Ancestors*	Quantidade média de classes a partir das quais cada classe herda informações.
CA	*Afferent Coupling*	Quantidade de classes fora de um pacote que depende de classes dentro do pacote.
CAM	*Cohesion Among Methods*	Calculada pela soma da interseção de parâmetros de um método com o conjunto máximo de todos os tipos de parâmetros na classe.
CE	*Efferent Coupling*	Quantidade de classes dentro de um pacote que depende de classes fora do pacote.
CIS	*Class Interface Size*	Contagem da quantidade de métodos públicos em uma classe.
DAM	*Data Access Metrics*	Quantidade de atributos privados/protegidos para a quantidade total de atributos na classe.
DCC	*Direct Class Coupling*	Contagem da quantidade diferente de classes a que uma classe está relacionada.
DIT	*Depth of Inheritance Tree*	Distância da classe *Object* na hierarquia de herança.
DSC	*Design Size in Classes*	Quantidade total de classes de origem.
EFE	Eficácia	Medida do nível de eficácia, definido como: 0.2*ANA+0.2*DAM+0.2*MOA+0.2*MFA+0.2*NOP.
ENT	Entendibilidade	Medida do nível de entendibilidade, dado por: -0.33*ANA+0.33*DAM-0.33* DCC+0.33*CAM-0.33*NOP-0.33*NOM-0.33*DSC.
EXT	Extensibilidade	Medida do nível de extensibilidade do código, dada por: 0.5*ANA-0.5*DCC+0.5*MFA+0.5*NOP.
FLE	Flexibilidade	Medida do nível de flexibilidade, definido como: 0.25*DAM-0.25*DCC+0.5*MOA+0.5*NOP.
FUN	Funcionalidade	Medida do nível de funcionalidade do código, dada por: 0.12*CAM+0.22*NOP+0.22*CIS+0.22*DSC+0.22*NOH.
LCOM	*Lack of Cohesion in Methods*	Contagem da quantidade de métodos que acessam um ou mais dos mesmos atributos.
LOC	*Lines of Code*	Linhas totais de código (preenchidas e sem comentários).
MFA	*Measure of Functional Abstraction*	Calcula a proporção da quantidade de métodos herdados para a quantidade de métodos acessíveis pelos métodos membros da classe.
MLOC	*Method Lines of Code*	Linhas de código dentro do corpo do método (preenchidas e sem comentários).
MOA	*Measure of Aggregation*	Quantidade de declarações de dados cujos tipos são classes definidas pelo usuário.
NBD	*Nested Block Depth*	Calcula a profundidade dos blocos de código aninhados.
NMI	*Number of Inherited Methods*	Quantidade total de métodos herdados no escopo selecionado.
NOC	*Number of Children*	Quantidade total de subclasses diretas de uma classe.
NOF	*Number of Attributes*	Quantidade total de campos definidos no escopo selecionado.
NOH	*Number of Hierarquies*	Quantidade de hierarquias de classe no *design*.
NOI	*Number of Interfaces*	Quantidade total de interfaces no escopo selecionado.
NOM	*Number of Methods*	Quantidade total de métodos definidos no escopo selecionado.
NONM	*Number of Normal Methods*	Quantidade de métodos normais no escopo selecionado.
NOP	*Number of Packages*	Quantidade de pacotes do escopo selecionado.
NOPM	*Number of Polimorphic Methods*	Quantidade de métodos polimórficos.
NORM	*Number of Overridden Methods*	Quantidade total de métodos sobrescritos no escopo selecionado.
NSC	*Number of Classes*	Quantidade total de classes no escopo selecionado.
NSF	*Number of Static Attributes*	Quantidade de atributos estáticos no escopo selecionado.
NSM	*Number of Static Methods*	Quantidade de métodos estáticos no escopo selecionado.
PAR	*Number of Parameters*	Quantidade de parâmetros no escopo selecionado.
REU	Reusabilidade	Medida do nível de reusabilidade do código, dada por: -0.25*DCC+0.25*CAM+0.5*CIS+0.5*DSC.
RMA	*Abstractness*	Calcula a quantidade de classes abstratas no pacote para todas as classes.
RMD	*Normalized Distance from Main Sequence*	Calcula o equilíbrio entre abstração e instabilidade do pacote, dado por |RMA + RMI - 1|.
RMI	*Instability*	Calcula o esforço para modificar um pacote, definido como CE / (CA + CE).
SIX	*Specialization Index*	Média do índice de especialização, definido como NORM * DIT / NOM.
SIX2	*Specialization Index 2*	Média do índice de especialização, definido como NORM * DIT / (NOM + NMI).
VG	*McCabe's Cyclomatic Complexity*	Conta a quantidade de fluxos através de um código.
WMC	*Weighted Methods per Class*	Soma da complexidade ciclomática de McCabe para os métodos em uma classe.

expressividade e facilidade de uso em comparação com linguagens de programação de propósito geral [11]. Exemplos de DSLs incluem (i) Lexx e Yacc para análise léxica e *parsing*, (ii) awk e PERL para processamento de texto, (iii) VHDL para descrição de hardware, (iv) TeX e LaTeX para preparação de documentos, (v) HTML e SGML para marcação de documentos, (vi) Tcl/Tk para *script* GUI, (vii) postscript e Open GL para gráficos, (viii) SQL e LDL para banco de dados, (ix) Mathematica e Maple para computação simbólica e (x) AutoCAD para *design* assistido por computador. Algumas linguagens de propósito geral podem ser ditas específicas de domínio, como as linguagens declarativas Prolog e ML [5, 6, 16].

3 UMA LINGUAGEM DE RESTRIÇÃO BASEADA EM MEDIDAS DE SOFTWARE

A sintaxe da linguagem MCL, suas utilidades e exemplos de restrições, que podem ser elaboradas para verificar violações no código-fonte de sistemas de informação orientados a objetos, são apresentados nesta seção. A ferramenta **`MCLcheck`** foi implementada para dar apoio automatizado à linguagem MCL. Essa ferramenta é um *pulg-in* para Eclipse IDE que permite a escrita de restrições MCL e verifica se cada restrição é violada. Uma interpretação em alto nível da sua arquitetura é ilustrada pela Figura 2. **`MCLcheck`** recebe como entrada as restrições elaboradas pelo usuário e o valor das 42 medidas dos componentes do sistema, fornecido pelo *Eclipse Metrics*. As restrições são utilizadas para verificar violações, retornando como saída as violações encontradas.

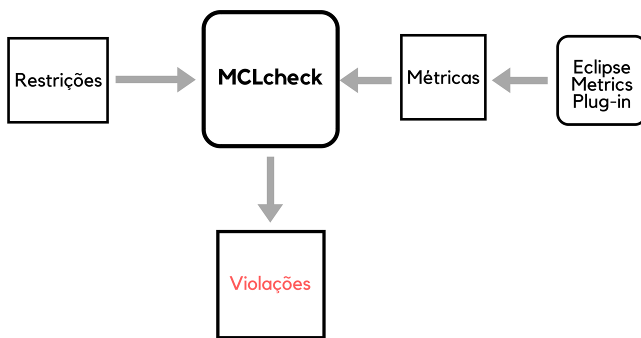


Figura 2: MCLcheck

3.1 Linguagem MCL

A linguagem MCL inclui em sistemas de informação restrições baseadas em medidas. Essas restrições possuem sintaxe intuitiva e simplificada, visando à uma linguagem fácil de ser aprendida e utilizada. Uma restrição pode ser definida por meio da escolha do(s) componente(s) do sistema que se deseja restringir e o valor da medida que determinará como será a restrição, utilizando operadores de comparação para verificar estaticamente no código-fonte se a regra é violada. A sintaxe de uma restrição possui a seguinte estrutura:

```
componente: métrica operador valor
```

Para determinar uma restrição, é definido o componente que se pretende restringir que pode ser todo o sistema, um conjunto de pacotes, um único pacote, uma classe ou um método. Assim, uma medida é escolhida, sendo que ela possui um escopo específico, ou seja, uma mesma medida pode ser utilizada para todos, um subconjunto ou um único componente do sistema. Em seguida, é determinado qual operador a ser utilizado para indicar o tipo de restrição pretendido. Por fim, indica-se o valor para delimitar a faixa de restrição que o componente deve respeitar. Os tipos de componentes, as medidas, os operadores e a faixa de valores a serem utilizados para elaborar regras são descritos na Figura 3.

3.1.1 Componente

A linguagem MCL fornece suporte para diferentes componentes de um sistema de informação orientado a objetos. Basicamente, a MCL suporta a elaboração de restrições considerando todo o sistema, um conjunto de pacotes, um pacote, uma classe ou um método. A seguir, são apresentados detalhes como cada componente deve ser escrito considerando a sintaxe da MCL.

Sistema: é utilizada a palavra reservada *system* ou o nome do sistema. Por exemplo, considerando um sistema **`S`** em que se deseja restringir o valor da medida FUN, relacionada à funcionalidade, de ser sempre superior a 10, pode ser criada uma restrição das seguintes formas:

```
system: FUN > 10
      ou
   S: FUN > 10
```

Conjunto de pacotes: é utilizado um prefixo que seria o nome comum aos pacotes a serem restringidos seguido do caractere *. Por exemplo, considera-se um sistema que tudo relacionado ao armazenamento e à recuperação de dados esteja em pacotes com o nome iniciado em **`persistence`**. Para criar uma restrição para restringir o valor da medida RMI, relacionada ao esforço de se modificar um pacote, sendo inferior ou igual a 0.5 para esses pacotes, pode ser especificada a seguinte restrição:

```
persistence.*: RMI <= 0.5
```

Nesse exemplo, se qualquer pacote violar a restrição, então a violação será detectada. Assim, se algum pacote de **`persistence`** possuir RMI > 0.5, uma violação é detectada, mesmo que o restante dos pacotes possuírem RMI <= 0.5.

Pacote: é utilizado o nome completo do pacote. Por exemplo, para restringir o número de classes abstratas dentro de um pacote para no máximo duas, utiliza-se a medida RMA sendo inferior a 2 para o pacote **`com.foo.bar`** de um sistema **`S`**. Desse modo, pode ser elaborada a seguinte restrição:

```
com.foo.bar: RMA <= 2
```

Classe: é utilizado o nome qualificado, i.e., a junção do nome do pacote em que a classe está e do nome da classe. Por exemplo, considerando o mesmo pacote do exemplo anterior, **`com.foo.bar`**, deseja-se restringir que a classe **`Main`** possua no máximo 5 métodos.

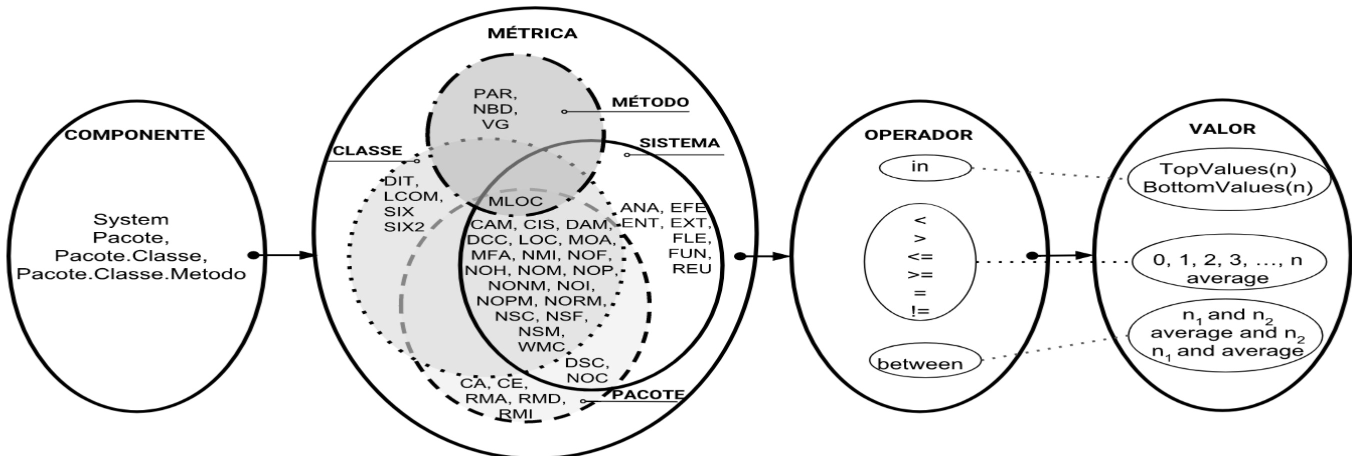


Figura 3: Estrutura das Restrições MCL

Portanto, especifica-se a seguinte restrição:

```
com.foo.bar.Main: NOM <= 5
```

Método: é utilizada a junção do nome qualificado da classe, do nome do método e dos tipos de parâmetros entre parênteses. Por exemplo, considera-se o método **`main`**, que possui o parâmetro *array* de *strings*. Para restringir o número de linhas do método para até 20, utilizando a medida MLOC, pode ser elaborada a restrição:

```
com.foo.bar.Main.main(String[]): MLOC <= 20
```

3.1.2 Valores Esperados

A linguagem MCL suporta 42 medidas e 3 tipos de valores esperados que delimitam as medidas (Figura 3).

Números Inteiros: o valor esperado é qualquer número inteiro, desde que esteja na faixa de valores delimitada para a medida e que se saiba se um número alto ou baixo do valor dessa medida é ou não o ideal. Por exemplo, as medidas CAM e LCOM podem ser utilizadas para medir a coesão de uma classe **`C`** de um pacote **`p`**. No entanto, cada uma se comporta de maneira diferente, sendo que um valor alto para CAM é o ideal e um valor baixo para LCOM é o ideal. Portanto, ao elaborar uma restrição que mantém alto valor de coesão para a classe **`C`**, pode-se fazê-la de duas formas:

```
p.C: CAM > 0.8
      ou
p.C: LCOM < 0.2
```

Média: pode-se determinar o valor esperado por meio da média calculada dos valores da medida especificada na restrição, verificados para todos os componentes do mesmo tipo de componente incluído na restrição. Para isso, usa-se a palavra reservada *average*. Por exemplo, ao restringir a quantidade de classes de um pacote **`p`** com a média da quantidade de classes por pacote do sistema, pode ser definida a restrição utilizando a medida NOC da seguinte forma:

```
p: NOC < average
```

Intervalo: o valor esperado pode ser determinado ao estabelecer uma faixa de valores na qual o valor da medida deve estar contido. Para delimitar a faixa de valores, são utilizadas as palavras reservadas *between* e *and*. Por exemplo, para determinar que (i) o valor da medida ENT, relacionada à compreensibilidade, para um sistema esteja entre uma faixa de valores e (ii) o valor da medida DCC, relacionada ao acoplamento, para uma classe **`C`** de um pacote **`p`** esteja entre um valor mínimo e o valor máximo igual a média de DCC para as classes do sistema, podem ser definidas as seguintes restrições:

```
(i) system: ENT between 4 and 8
(ii) p.C: DCC between 2 and average
```

Em (ii), se o valor de *average* for menor que 2, troca-se automaticamente os valores de lugar, ou seja, a restrição passa a ser: `p.C: DCC between average and 2`.

Melhores Valores: o valor esperado de uma medida para um componente do sistema pode ser determinado para ele estar contido entre os melhores valores da medida calculada para os tipos de componentes especificados na restrição. Para isso, são utilizadas as regras $TopValues(n)$ ou $BottomValues(n)$, sendo $n \in \mathbb{N}$ e $n > 0$. Por exemplo, para verificar se uma classe está entre as 10 classes mais encapsuladas do sistema, utiliza-se o valor da medida DAM para uma classe **`C`** do pacote **`p`** e, assim, define-se a restrição que irá calcular o valor de DAM para todas as classes do sistema, ordenar as classes do maior ao menor valor de DAM e verificar se **`C`** está entre as 10 classes com os maiores valores calculados de DAM:

```
p.C: DAM in TopValues(10)
```

3.1.3 Operadores

Além da escolha da medida e do valor esperado, deve-se escolher o tipo de operador para delimitar o valor desejado para a medida, que pode ser de três tipos descritos a seguir.

Operadores de Comparação: utilizados para comparar o valor atual da medida com o seu valor esperado. Desse modo, esses valores são comparados verificando se são iguais (=), diferentes (! =) menor (<), maior (>), menor ou igual (<=) ou maior ou igual (>=).

Operador de Delimitação: utilizado para restringir o valor esperado de uma medida em uma faixa de valores. Nesse conjunto de operadores, está presente o operador *between*, sendo o valor esperado um intervalo.

Operador de Inclusão: constituído pelo operador *in*, sendo o valor esperado $TopValues(n)$ ou $BottomValues(n)$.

3.1.4 Exemplos Ilustrativos

A seguir, são apresentados exemplos de regras que podem ser elaboradas seguindo a estrutura apresentada na Figura 3:

```
1: system: REU between 2 and 5
2: p.*: CAM in TopValues(10)
3: p: DAM = 1
4: p.C: DCC < 4
5: p.C.m: MLOC <= 15
6: p.C: LCOM in BottomValues(10)
7: p.C.m: NBD <= 6
8: p.*: CA > average
```

1: Para manter o código de um sistema com um alto nível de reutilização, podendo assim ser utilizado em outros projetos de sistemas de informação, pode ser criada uma restrição que delimita a medida REU, relacionada à reusabilidade, a uma faixa de valores entre 2 e 5, visando garantir que essa propriedade não seja infrigida;

2: Quando um conjunto de pacotes precisa estar entre os pacotes mais coesos do sistema, é criada uma restrição utilizando a medida CAM para verificar se o valor calculado para cada pacote está entre os 10 melhores valores da medida CAM calculados para todos os pacotes do sistema;

3: Para as classes do pacote **p**, é utilizada a medida DAM para verificar se a quantidade de atributos privados/ protegidos do pacote dividido pela quantidade total de atributos seja igual a 1, garantindo dessa forma que essas classes estejam totalmente encapsuladas;

4: Se uma classe **C** de um pacote **p** não pode ter alto acoplamento, é definida uma restrição com a medida DCC para restringir o valor de acoplamento da classe **C** sendo inferior a 4;

5: Para evitar que um método **m** tenha quantidade elevada de linhas, é criada uma restrição com a medida MLOC para limitar o seu tamanho em no máximo 15 linhas;

6: É verificada se a falta de coesão de uma classe **C** de um pacote **p** é uma das mais baixas do sistema, considerando as classes d sistema. Desse modo, é elaborada uma restrição que verifica se o valor da medida LCOM da classe **C** está entre os 10 menores valores da medida LCOM calculados para as classes do sistema;

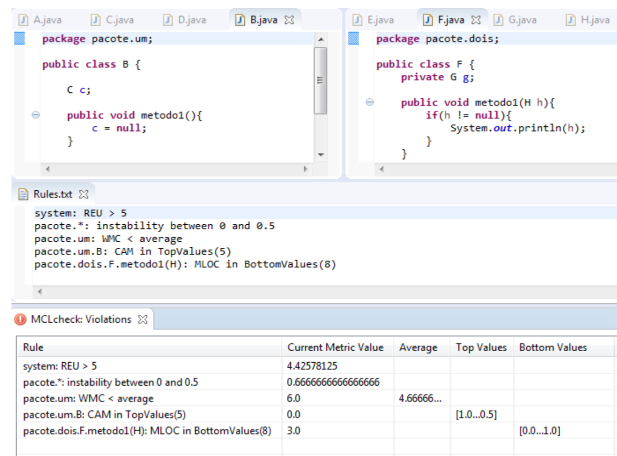
7: Se um método **m** não pode ter muitos blocos aninhados, que o deixaria mais complexo e menos legível, é determinada uma restrição para o método não ter o valor da medida NBD superior a 6;

8: Para verificar se as classes de um conjunto de pacotes são mais utilizadas do que todas as outras classes do sistema, é utilizada a medida CA, que fornece a quantidade de classes que dependem de determinadas classes dentro de um pacote, para verificar se seu valor para cada pacote do conjunto é maior que a média dos valores da medida CA calculados, considerando todos os pacotes do sistema.

3.2 Ferramenta

Para ter um apoio automatizado para utilizar a linguagem MCL, foi implementado um *plug-in* para o Eclipse IDE (**MCLcheck**). Utilizando como base o código-fonte do *plug-in Eclipse Metrics*, foram acrescentadas as funções para inclusão de restrições MCL e para detecção de violações a essas restrições. Ao utilizar o **MCLcheck**, são calculadas 42 medidas para os componentes do sistema e é disponibilizado um arquivo para o usuário informar as restrições, as quais o **MCLcheck** utiliza para verificar se há violações. Caso sejam encontradas, as violações são reportadas ao usuário.

O cálculo do valor das medidas é feito pelo *Eclipse Metrics*. Esses valores são utilizados pelo **MCLcheck** para comparar com os valores (ou intervalo de valores) das medidas definidas nas restrições. O cálculo e a comparação são feitos estaticamente no código-fonte do sistema. A Figura 4 fornece uma visualização do funcionamento do **MCLcheck** ao ser utilizado no Eclipse IDE, na qual pode-se observar as restrições incluídas propositalmente para gerar violações e mostrá-las ao usuário.



Rule	Current Metric Value	Average	Top Values	Bottom Values
system: REU > 5	4.42578125			
pacote.*: instability between 0 and 0.5	0.6666666666666666			
pacote.um: WMC < average	6.0	4.66666...		
pacote.um.B: CAM in TopValues(5)	0.0		[1.0...0.5]	
pacote.dois.F.metodo1(H): MLOC in BottomValues(8)	3.0			[0.0...1.0]

Figura 4: Ferramenta MCLcheck

4 APLICABILIDADE EM SI

Nesta seção, são apresentadas as possíveis aplicações da linguagem MCL em sistemas de informação, revelando a utilidade da MCL como ferramenta de suporte eficaz para o desenvolvimento e a manutenção desses sistemas. Desse modo, para mostrar a aplicabilidade prática da linguagem MCL, foi utilizado o sistema **MyAppointments** [15], apresentando restrições que agregariam manutenibilidade ao sistema. Essas restrições são inclusive um guia de possíveis utilizações a serem aplicadas a outros sistemas, com leves adaptações.

4.1 Sistema MyAppointments

O **MyAppointments** é um sistema de informação para gerenciar contatos pessoais, fornecendo funções para adicionar, buscar, modificar e excluir contatos. Desenvolvido em Java, sua arquitetura segue o modelo MVC (*Model-View-Controller*), sendo seus componentes alocados de acordo com sua funcionalidade. Possui 22 classes (6 classes em *model*, 7 classes em *view*, 7 classes em *controller* e 2 classes utilitárias), 99 métodos e 1.213 linhas de código. Apesar de sua simplicidade, seu desenvolvimento baseou-se em padrões de qualidade comumente utilizados em projetos reais.

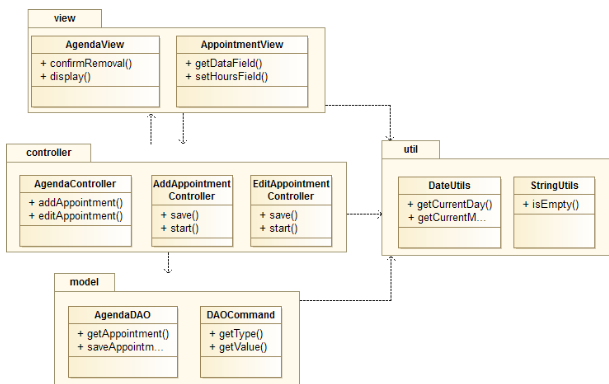


Figura 5: Arquitetura do MyAppointments

Na Figura 5, é apresentado o diagrama de classes simplificado do **MyAppointments**, informando seus componentes principais e algumas de suas classes e métodos utilizados para elaborar as restrições MCL do sistema.

4.2 Restrições no MyAppointments

Por oferecer amplo conjunto de opções na escolha de medidas, componentes e operadores para a elaboração das restrições, a linguagem MCL provê suporte a muitos cenários em que a sua utilização se torna relevante para assegurar diferentes características relacionadas a manutenibilidade de um sistema. Nesta seção, são apresentados exemplos de cenários utilizando MCL no **MyAppointments**.

4.2.1 Conformidade Arquitetural

Em um sistema de informação que segue a arquitetura MVC, a camada *model* não deve depender das camadas *view* e *controller*. Com o propósito de verificar no **MyAppointments** se classes pertencente à camada *model* não dependem de classes pertencentes a outras camadas, a linguagem MCL permite a elaboração de uma restrição que especifica o valor 0 para a medida CE:

```
model.*: CE = 0
```

Assim, serão verificadas as classes pertencentes a *model* se o valor da medida CE é igual a zero (se há dependências de saída dessas classes com as classes das outras camadas do sistema). Caso não haja violação dessa restrição para as classes de *model*, então esse aspecto da arquitetura MVC está preservado.

4.2.2 Componentes Largamente Utilizados

Classes utilitárias contém recursos utilizados por grande parte do sistema e não devem depender de outras classes do sistema, somente serem utilizadas por elas. Para garantir a preservação desse conceito, podem ser criadas duas restrições para as classes utilitárias do **MyAppointments**:

```
util: CE = 0
util: CA in TopValues(1)
```

A primeira restrição utiliza a medida CE para verificar se as classes em *util* depende de outras classes do sistema. A segunda restrição utiliza a medida CA para, além de verificar se as classes de *util* estão sendo realmente utilizadas por outras classes do sistema, verificar se essas classes são as mais utilizadas do sistema ao restringir que o valor de CA seja o maior dentre todos os pacotes. Caso não haja violações às duas restrições, então o conceito de classe utilitária está preservado.

4.2.3 Componentes Hierárquicos

Uma das estratégias para prevenir que um sistema de informação orientado a objetos tenha alto acoplamento é evitar grande quantidade de hierarquias. Para isso, pode ser utilizada a medida NOH para restringir a quantidade de hierarquias do **MyAppointments**:

```
system: NOH between 1 and 3
```

Assim, pode ser controlada a quantidade de classes hierárquicas do sistema em um nível desejado. Nesse caso, restringe a no mínimo uma classe hierárquica (devido a existência de classes abstratas no sistema que são estendidas por classes concretas) e no máximo 3 classes. Uma violação dessa restrição indica que a restrição foi infrigida e o acoplamento entre as classes do sistema está alto (se o valor de NOH ultrapassar 3) ou que não há classes hierárquicas (se o valor de NOH for igual a 0).

4.2.4 Coesão e Acoplamento

Fatores como alto acoplamento e baixa coesão podem interferir negativamente na manutenibilidade de sistemas de informação. Por isso, é desejável manter baixo o acoplamento e alto a coesão entre classes de um sistema [3]. A linguagem MCL permite criar restrições utilizando medidas de coesão e de acoplamento. Um modo de utilizar essas restrições no **MyAppointments** é:

```
model: DCC in BottomValues(2)
controller.AddAppointmentController: LCOM = 0
```

Como a camada *model* não pode depender de classes de outras camadas, com exceção de classes utilitárias, a primeira regra restringe que a quantidade de dependências de *model*, utilizando-se

da medida DCC, esteja entre os dois menores valores do sistema. O fato da restrição ser entre os dois menores valores de DCC se deve pelo motivo do pacote *util* não poder possuir nenhuma dependência com outras classes e, portanto, possuir o menor valor possível de DCC, ou seja, 0. Nesse caso, *model* deverá possuir o segundo menor valor de DCC considerando todos os pacotes do sistema. A segunda restrição garante que a classe **AddAppointmentController** tem a mais alta coesão possível ao assegurar que o valor da medida LCOM é igual a 0.

4.2.5 Comunicação entre Componentes

Objetos comunicam entre si por meio de mensagens e isso é essencial na programação orientada a objetos, principalmente quando o objetivo é simular os objetos e seus comportamentos de acordo com o que é percebido no mundo real. Porém, quando ocorre de um objeto comunicar-se com vários outros objetos, ocasiona acúmulo de responsabilidades, afetando negativamente características como coesão e acoplamento de uma classe. A linguagem MCL dispõe da medida CIS (quantidade de métodos públicos em uma classe) para auxiliar a restringir a troca de mensagens com outros objetos. Como exemplo, pode ser criada a restrição no **MyAppointments**:

```
model.DAOCommand: CIS <= 3
```

Como a classe **DAOCommand** possui somente dois métodos *getters*, além de seu construtor, ao restringir o valor da medida CIS, essa classe não trocará mensagens com outros objetos além do necessário. Fica a cargo do programador verificar estratégias para contornar essa violação caso detectada.

4.2.6 Componentes de Qualidade

Considerar qualidade no processo de implementação de sistemas de informação orientados a objetos é primordial principalmente quando se pretende atender a requisitos não-funcionais (reusabilidade, funcionalidade, eficácia, etc.). A linguagem MCL fornece suporte robusto para qualidade ao considerar as medidas QMOOD, focadas exclusivamente na qualidade desses sistemas. Visto que o **MyAppointments** foi implementado seguindo altos padrões de qualidade, pode-se criar restrições para assegurar que futuras modificações não impactem negativamente na qualidade:

```
system: REU <= 20
system: FLE <= 1
system: EFE <= 1
system: EXT <= 1.5
system: FUN <= 0
system: ENT >= -9
```

Baseado nos valores correntes de cada medida QMOOD calculada para o **MyAppointments**, essas seis regras garantem que, ao realizar modificações no código, a qualidade atual do sistema não diminua.

4.2.7 Tamanho dos Componentes

A preocupação com o tamanho de um sistema de informação é relevante quando são considerados fatores como complexidade e compreensibilidade do código-fonte. Manter o tamanho dos componentes o mínimo possível agrega em atividades de manutenção, poupando tempo e custo. A linguagem MCL trabalha com várias medidas orientadas a tamanho no contexto de sistemas orientados a objetos, ficando a cargo do programador delimitar as restrições de acordo com o componente que deseja restringir o tamanho. No **MyAppointments**, podem ser elaboradas as restrições:

```
system: LOC <= 2000
system: NOP <= 6
util: NSC <= 2
controller.AgendaController: NOM <= 8
view.AppointmentView: NOF <= average
model.AgendaDAO.getAppointment: MLOC <= 20
```

Essas restrições limitam a quantidade de linhas de código do sistema (primeira restrição - utilizando a medida LOC), a quantidade de pacotes do sistema (segunda restrição - utilizando a medida NOP), a quantidade de classes do pacote **util** (terceira restrição - utilizando a medida NSC), a quantidade de métodos da classe **AgendaController** (quarta restrição - utilizando a medida NOM), a quantidade de atributos da classe **AppointmentView** (quinta restrição - utilizando a medida NOF e *average* para determinar que o valor de NOF seja menor que a média de NOF calculada a partir de todas as classes do sistema) e a quantidade de linhas do método **getAppointment** (sexta restrição - utilizando a medida MLOC). Restrições referentes a tamanho devem ser feitas quando tem conhecimento prévio dos tipos de modificações a serem feitas no sistema e, preservando o tamanho dos componentes desejados, preserva-se a entendibilidade e a complexidade do sistema de informação.

4.2.8 Manutenibilidade dos Componentes

Grande parte dos esforços de construção de um sistema de informação são realizados em atividades de manutenção. Quanto mais dificuldade se tem para manter o sistema, maior será o seu custo. Garantir que componentes do sistema tenha alto grau de manutenibilidade auxilia no processo de manutenção quanto ao esforço e ao tempo gastos em modificações. A linguagem MCL provê medidas de complexidade de componentes:

```
view: RMI <= 0.4
view.Agendaview.display(): VG < 4
view.AppointmentView: WMC < average
```

A primeira restrição delimita o esforço para modificar um pacote por utilizar a medida RMI. A segunda restrição delimita o fluxo de execução do código por utilizar a medida VG para o método **display**. A terceira restrição delimita a complexidade dos métodos do pacote **AppointmentView** por utilizar a medida WMC, sendo que o valor de WMC para a classe **AppointmentView** tem que ser menor que o valor da média de WMC calculada a partir de todas as classes do sistema. Com essas restrições, é mantido o nível de complexidade do código aceitável e, como consequência, alto nível de manutenibilidade.

5 TRABALHOS RELACIONADOS

As medidas de sistemas de informação são utilizadas como recursos essenciais para a melhoria da qualidade e controle de custos referentes ao desenvolvimento de software [4]. Para automatizar o processo de medição desses sistemas, os seguintes estudos se destacam.

Morais et al. [12] propuseram um serviço *web* para monitoramento e interpretação de medidas de código-fonte denominado *Kalibro Metrics*. Esse serviço suporta a conexão com ferramentas de coleta de medidas de código-fonte, estendendo-a para fornecer uma avaliação dos resultados obtidos. Esse serviço viabilizou o desenvolvimento da ferramenta *Mezuro* [10], cuja ideia é fornecer um intervalo de valores para as medidas juntamente com as possíveis interpretações desses intervalos. Essa ferramenta suporta a análise de código-fonte em C, C++ e Java. Além disso, permite a atribuição de pesos para as medidas e a criação de medidas a partir das existentes. Contudo, a medição ocorre com granularidade grossa, por exemplo, ao especificar que a medida CAM deve possuir valor maior que 0.8, as classes do sistema são avaliadas com base nessa especificação. Diferentemente, a abordagem proposta neste artigo considera granularidade fina. Assim, no exemplo anterior, pode ser definido tal valor para apenas um conjunto de classes.

De modo similar, Kocaguneli et al. [8] propuseram a ferramenta *Prest* para prever configurações de medidas utilizando aprendizado de máquina. Essa ferramenta suporta a extração de medidas nas linguagens C, C++, Java, JSP e PL/SQL. Ainda, a ferramenta permite a aplicação de cores e termos linguísticos (e.g., cor vermelha associada a "ruim") para as condições especificadas para cada medida e a criação de medidas por meio da combinação das medidas existentes e operadores matemáticos. Pelo fato de a interpretação dos resultados ocorrer de forma qualitativa, há uma dificuldade em saber o quanto deve ser melhorado para alcançar o valor definido para a medida. Em contraste, a abordagem proposta neste artigo verifica se um sistema está em conformidade com as restrições estabelecidas e reporta as violações, para as quais é fornecido o valor obtido pela medida e o valor esperado, permitindo saber o quanto deve ser melhorado para alcançar o resultado esperado.

Voltado para as interpretações qualitativas de medidas de sistemas de informação, Oliveira [14] apresenta uma ferramenta que utiliza o conceito de lógica *fuzzy* para definir o grau de qualidade de um código-fonte orientado a objetos. Essa ferramenta permite criar uma relação entre requisitos de qualidade e medidas de software, avaliada conforme as funções de pertinência estabelecidas. No entanto, apresenta os mesmos problemas relacionados a granularidade de aplicação das medidas e interpretação qualitativa dos resultados.

6 CONCLUSÃO

As medidas de software são essenciais para a melhoria da qualidade de sistemas de informação. Contudo, os valores dessas medidas é de difícil interpretação e associação à qualidade desses sistemas. Dessa forma, neste artigo, é proposta a linguagem de restrição baseada em métricas, denominada MCL, que proporciona uma forma de definir um conjunto de medidas e seus respectivos valores de modo a facilitar a interpretação dos resultados e sua associação à qualidade de sistemas de informação. Ainda, foi implementada a ferramenta **MCLcheck** que recebe como entrada um conjunto de restrições especificadas por meio da linguagem MCL e os valores brutos de 42 medidas de sistemas de informação orientados a objetos. A partir desses dados, a ferramenta verifica os valores brutos e a faixa de valores especificados para as medidas, sendo as inconformidades encontradas relatadas pela ferramenta.

Essa ferramenta apresenta como diferencial a configuração de medidas em granularidade "fina". Por exemplo, diferentes classes do projeto podem assumir valores diferentes para uma mesma medida, diferentemente da proposta de outras ferramentas existentes. Ainda, a interpretação dos valores não ocorre de forma qualitativa, o que a facilita a percepção do quanto se deve melhorar para obter o resultado esperado.

Como trabalhos futuros, pretende-se (i) avaliar a facilidade de entendimento e uso da linguagem, (ii) aprimorar a sintaxe de modo a permitir a utilização de operadores lógicos como AND, OR e NOT, (iii) aprimorar a sintaxe para permitir a utilização de expressões regulares para criar conjuntos de classes, métodos ou pacotes, (iv) incluir novas medidas e (v) avaliar a linguagem em sistemas de informação orientados a objetos de código aberto e em projetos reais de empresas.

AGRADECIMENTOS

Este trabalho foi apoiado pela CNPq, CAPES e FAPEMIG.

REFERÊNCIAS

[1] J. Bansiya and C. G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17. https://doi.org/10.1109/32.979986
[2] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895
[3] B. R. de O. Rodrigues, D. E. de Souza, and E. M. L. Figueiredo. 2014. Medindo Acoplamento em Software Orientado a Objeto: Uma Perspectiva do Desenvolvedor. *Abakós* 3, 1 (2014), 3–17. https://doi.org/10.5752/P.2316-9451.2014v3n1p3
[4] A. Gopal, M. S. Krishnan, T. Mukhopadhyay, and D. R. Goldenson. 2002. Measurement programs in software development: determinants of success. *IEEE Transactions on Software Engineering* 28, 9 (2002), 863–875. https://doi.org/10.1109/TSE.2002.1033226
[5] P. Hudak. 1997. Domain-specific languages. *Handbook of programming languages* 3, 39-60 (1997), 21.
[6] P. Hudak. 1998. Modular domain specific languages and tools. In *5th International Conference on Software Reuse (ICSR)*. IEEE, Victoria, BC, Canada, 134–142. https://doi.org/10.1109/ICSR.1998.685738
[7] ISO IEC. 2014. ISO/IEC 25000 Software Engineering Software Product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. *Systems Engineering* 41 (2014), 27.
[8] E. Kocaguneli, A. Tosun, A. B. Bener, B. Turhan, and B. Caglayan. 2009. Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool. In *21th Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute, Boston, MA, USA, 637–642.
[9] R. Martin. 1994. OO Design Quality Metrics – An Analysis of Dependencies. In *9th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Portland, OR, USA, 151–170.
[10] P. Meirelles, F. Kon, and C. Santos. 2011. Semi-Automatic Evaluation of Free Software Projects: A Source Code Perspective. In *7th International Conference on Open Source Systems (OSS)*. Springer, Salvador, BA, Brazil, 42–53.
[11] M. Mernik, J. Heering, and A. M. Sloane. 2005. When and How to Develop Domain-specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892
[12] C. Morais, P. Meirelles, and E. Morais. 2012. Kalibro Metrics: um serviço para monitoramento e interpretação de métricas de código-fonte. In *13th Workshop Internacional de Software Livre (WSL)*. Porto Alegre, RS, Brazil, 1–11.
[13] C. Neelamegam and M. Punithavalli. 2009. A survey-object oriented quality metrics. *Global Journal of Computer Science and Technology* 9, 4 (2009), 183–186.
[14] J. A. C. M. Oliveira. 2015. *Avaliação de código-fonte orientado a objetos usando requisitos não-funcionais, métricas e lógica fuzzy*. B.Sc. thesis. Universidade Tecnológica Federal do Paraná.
[15] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das C. Mendonca. 2010. Static architecture-conformance checking: An illustrative overview. *IEEE Software* 27, 5 (2010), 132–151. https://doi.org/10.1109/MS.2009.117
[16] D. Spinellis. 2001. Notable design patterns for domain-specific languages. *Journal of Systems and Software* 56, 1 (2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3