Departamento de Ciências e Tecnologias da Informação

Aplicação da OCL à especificação do problema de elaboração de horários

Cláudia Maria da Rocha Moreira

Dissertação submetida como requisito parcial para obtenção do grau de

Mestre em Gestão de Sistemas de Informação

Orientador:

Professor Doutor Pedro Nogueira Ramos, Professor Associado,

Instituto Superior de Ciências do Trabalho e da Empresa/IUL

Janeiro, 2011

Agradecimentos

Ao meu orientador Professor Doutor Pedro Nogueira Ramos pelo sempre disponível aconselhamento científico e pelo que ensinou durante a elaboração da pressente tese.

Ao meu marido, Nelson Russo que ao longo da tese sempre demonstrou uma enorme carinho, compreensão e estímulo para me incentivar.

Aos meus amigos, pela amizade e por terem sempre acreditado em mim.

Ao meu filho Daniel, nascido durante esta jornada, que com o seu sorriso, sempre presente, deu à mãe uma enorme coragem e força.

Ao Björn Freitag da Universidade de Dresden (Alemanha) pela disponibilidade e conhecimento transmitido que me permitiu testar a interface utilizada.

Ao professor João Pascoal Faria da Faculdade de Engenharia do Porto pela ajuda prestada quando solicitada.

Por fim, e pela sua importância queria agradecer de uma forma muito especial aos meus Pais e Irmão, que sempre estiveram presentes e que me apoiaram de uma forma única que me encheu de carinho, confiança e determinação.

Resumo

Um horário bem elaborado é um requisito importante para a boa administração de qualquer instituição de ensino superior. Porém, o grande número de factores envolvidos, como a quantidade de unidades curriculares, a alocação dos docentes e a distribuição dos recursos, torna o problema da criação dos horários muito complexo.

A OCL (Object Constraint Language) é uma linguagem que pode ser utilizada para especificar, de forma precisa, elementos que as notações gráficas da UML não são capazes de representar, como, por exemplo, restrições, expressões associadas a atributos derivados, expressões de consulta e definições contratuais de operações. Embora a OCL tenha sido definida com o objectivo de ser uma linguagem de uso mais fácil, se comparada às linguagens formais tradicionais, as especificações produzidas com a OCL podem apresentar problemas de legibilidade e manuseamento.

Este trabalho utiliza a OCL para a modelação de uma ferramenta de elaboração e optimização de horários e a conversão delas para SQL através de uma ferramenta de transformação automática baseada em MDA. Com isso, pretende-se testar a capacidade expressiva e robustez da linguagem OCL como forma de validar se um modelo satisfaz ou não um conjunto de restrições, independentemente da forma como ele foi obtido.

Ou seja, pretende-se responder à seguinte questão: A OCL pode ser utilizada para definir, de forma precisa e não ambígua, as restrições e a semântica das operações de um modelo?

Palavras-chave: Elaboração de horários, restrições, OCL

Abstract

A well-designed timetable is an important prerequisite for the proper administration of any institution of higher education. However, the large number of factors involved, such as the amount of course units, the allocation of teachers and distribution of resources makes the issue of creation of very complex timetable.

The OCL (Object Constraint Language) is a language that can be used to specify, precisely, that the elements of the UML graphical notations are not capable of representing, for example, constraints associated with attributes derived expressions, query expressions and definitions of contractual transactions. Although the OCL has been defined in order to be an easier language to use when compared to traditional formal languages, the specifications produced with the OCL can present problems in handling and readability.

This work uses OCL for modeling a tool for development and optimization of timetables and their conversion to SQL via an automated transformation tool based on MDA. With this, we intend to test the expressive power and robustness of language OCL in order to validate whether or not a model fits a set of constraints, regardless of how it was obtained.

That is, we intend to answer the following question: The OCL can be used to define, precisely and unambiguously, the constraints and semantics of the operations of a model?

Keywords: Timetabling, constraints, OCL

Índice

Índice de Tabelas

Índice de Figuras

Lista de abreviaturas

CASE	*Computer Aided Software Engeneering*
CLP	*Constraint Logic Programming*
DOT	*Dresden OCL Toolkit*
e.g.	*Por Exemplo*
ECTS	*European Credit Transfer and Accumulation System*
GRASP	*Greedy Randomized Adaptive Search Procedure*
IBM	*International Business Machines*
L	*Aulas de Laboratório*
MDA	*Model Driven Architectire*
MOF	*Meta-Object Facility*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OO	*Orientação a objectos - Modelo de programação*
P	*Aulas Práticas*
PDF	*Portable Document Format.*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
SPACES	*Specification bAsed Component tESter*
SQL	*Structured Query Language*
T	*Aulas Teóricas*
TP	*Aulas Teóricas/Práticas*
UC	*Unidade Curricular*
UML	*Unified Modeling Language*
XMI ou XML	*XML Metadata Interchange*

1. Introdução

1.1. Motivação

Ao longo dos últimos 40 anos aproximadamente, a comunidade científica investiga uma solução computacional que auxilie num antigo, longo e duro processo: a alocação de recursos com restrições. Um dos problemas típicos desta natureza é o problema de alocação de horários escolares (Radaelli & Terra, 2007).

O problema de Elaboração de Horários consiste em definir o dia, hora e sala em que cada aula irá ser leccionada, tendo em conta número limitado de períodos de tempo e local, e minimizar as violações de um conjunto de restrições. A criação de horários consiste na alocação de um número de eventos a um número finito de espaços de tempo (ou períodos), em que as necessidades de restrições são satisfeitas (Burke & Newall, 1999).

Seria impraticável, as aulas, da mesma unidade curricular, serem dadas todas ao mesmo tempo, ou no mesmo dia, visto que exigiria um enorme esforço de coordenação de docentes, provavelmente aumentaria o número de docentes necessários para concretização das sessões lectivas, e no ponto de vista dos estudantes, pedagogicamente, seria um desastre.

Além das restrições óbvias, várias outras restrições podem ser impostas, nomeadamente a imposição que determinadas horas ou dias do horário de um estudante/docente/sala têm de estar livres ou que determinadas aulas sejam dadas após uma dada hora ou apenas em certos dias. Podemos ainda considerar dois tipos de restrições, as obrigatórias (restrições fortes), isto é, aquelas que não podem ser infringidas, como por exemplo, um docente não pode leccionar duas aulas ao mesmo tempo, ou as restrições que não precisam de ser obrigatoriamente cumpridas (restrições fracas) mas que têm influência na qualidade da solução. Por exemplo, o número de furos[1] num horário de ser minimizado. No entanto, esta restrição pode não ser integralmente cumprida.

Existem instituições que utilizam horários flexíveis, o que permite maior optimização de recursos, ajustando-se às necessidades da escola e dos estudantes. Aos Conselhos Executivos dos Departamentos, é possível uma gestão mais eficiente dos Docentes

[1] Tempo livre entre dois períodos de aulas.

(estes podem leccionar diferentes unidades curriculares e ajustar os recursos a outras actividades académicas). Para mitigar o insucesso escolar a algumas unidades curriculares, é possível a criação de turmas extraordinárias adequadas ao número de alunos, mas torna o problema de elaboração mais complexo.

Apesar de existirem ferramentas disponíveis no mercado, como por exemplo a ferramenta THOR, não existente nenhuma que trate de forma eficiente as restrições reais existentes na instituição em estudo e que seja possível alterar/adicionar de forma rápida as restrições. A inexistência de soluções satisfatórias prontas a usar, foi a principal motivação para a definição do tema desta dissertação.

Os processos de desenvolvimento de *software*, tipicamente, envolvem: a identificação do domínio do problema por meio de modelos de análise; a definição da arquitectura da solução, utilizando modelos de desenho; e a implementação propriamente dita da solução, numa linguagem de programação (modelos de implementação) (Jacobson, Booch, & Rumbaugh, 1999). Neste contexto, a modelação é um dos recursos utilizados para auxiliar no desenvolvimento e manutenção de *software*.

A UML é uma linguagem de modelação largamente utilizada, fornecendo um conjunto de convenções diagramáticas que são usadas para auxiliar a esboçar e a documentar sistemas de *software* (Almeida, 2006).

A versão 1.0 da UML foi bastante criticada por ser definida com muito pouco rigor. A sua especificação era suportada quase exclusivamente por elementos gráficos da própria UML, por meio de um meta-modelo MOF (Meta-Object Facility). Esta abordagem permitia uma grande ambiguidade na interpretação dos modelos. Um primeiro passo em direcção a uma maior precisão da especificação UML foi a criação da linguagem textual OCL (Object Constraint Language) (OMG) (Warmer & Kleppe, 2003), baseada numa linguagem desenvolvida por um grupo de pesquisa da IBM para definição de restrições em modelos. Ela é baseada em teoria de conjuntos e lógica de predicados, sendo que a sua semântica formal encontra-se definida na tese de Richters (Richters, 2002).

OCL é uma linguagem criada especificamente para descrever expressões e regras complexas (Morgan, 2001). O objectivo da criação da OCL foi estender o poder de expressão de restrições da UML, permitindo especificar diversos tipos de regras, tais como, invariantes, valores iniciais de atributos e computação de valores derivados. A princípio, ela foi utilizada unicamente para a definição de regras de boa formação das especificações dos meta-modelos da UML e da própria MOF. No entanto o seu uso foi posteriormente alargado para representar propriedades e restrições cuja complexidade praticamente impossibilita a sua representação através de representações iconográficas. O uso de linguagem natural para descrever essas

restrições não é solução pois a sua imprecisão dificulta a automatização do processamento do modelo. Sendo adoptada como parte integrante da UML 1.1, a OCL foi então disponibilizada para a comunidade de modelação de *software*.

Actualmente, a OCL tem sido usada em diferentes contextos e com diferentes propósitos no desenvolvimento de *software*. Como exemplo de utilização veja-se (Varella, Pereira, Held, Zimbrão, & Silva, 2004) onde expressões em OCL são utilizadas para detectar eventos em base de dados que violem regras comportamentais que uma aplicação deve satisfazer.

A literatura salienta que a especificação de modelos UML, enquadrados numa arquitectura MDA[2], contendo regras comportamentais escritas em OCL atenua e os problemas comuns no desenvolvimento de aplicações. No entanto, é ainda difícil encontrar na literatura descrições de aplicações de OCL a casos reais. Uma forte motivação deste trabalho consiste em averiguar até que ponto a OCL é suficientemente rica para representar o tipo de restrições que caracterizam o problema dos horários escolares.

1.2. Objectivos gerais

O principal objectivo deste trabalho é testar a capacidade expressiva e robustez da linguagem OCL como forma de validar se um modelo satisfaz ou não um conjunto de restrições, independentemente da forma como ele foi obtido.

Ou seja, pretende-se responder à seguinte questão: A OCL pode ser utilizada para definir, de forma precisa e não ambígua, as restrições e a semântica das operações de um modelo?

Em termos operacionais, a satisfação do objectivo enunciado implica que tenhamos que efectuar a conversão das restrições OCL para uma linguagem que possamos manipular computacionalmente. Optámos pela linguagem SQL visto que é relativamente independente de soluções tecnológicas de implementação (um requisito da abordagem MDA) A transformação de restrições OCL em expressões SQL é apenas um objectivo intermédio do trabalho.

Serão utilizadas restrições reais, recolhidas por docentes experientes na elaboração

[2] MDA (Model Driven Architecture), que é um Framework que utiliza UML e outros padrões da OMG (Object Management Group) para desenvolver *software* baseado em modelos abstractos (Kleppe, Warmer, & Bast, 2003)

dos horários. As regras são inicialmente representadas em OCL e posteriormente, de uma forma automática, convertidas para comandos SQL. O procedimento para testar as restrições consiste em desenhar soluções que não satisfaçam essas regras e verificar se os comandos SQL detectam essas situações de violação.

Um segundo objectivo genérico consiste em encontrar uma solução satisfatória para a elaboração dos horários. Este objectivo passa pela escolha da tecnologia mais adequada ao problema, e pela identificação dos requisitos que asseguram uma solução satisfatória. Dada a limitação de tempo inerente a uma dissertação, não era nossa intenção testar se a solução encontrada satisfazia globalmente as necessidades da escola. Pretende-se apenas testar se o protótipo desenvolvido oferece indícios fortes de que a solução testada poderá mostrar-se satisfatória quando totalmente implementada.

A Figura 1 pretende ilustrar os dois objectivos do presente trabalho. Idealmente os dois objectivos deveriam estar interligados. Ou seja, as restrições OCL deveriam servir de suporte à aplicação para elaboração de horários. Se assim fosse, os comandos SQL poderiam servir de teste aos horários gerados pela aplicação. Como será apresentado ao longo do trabalho, o facto de termos optado pela utilização de algoritmos genéticos, aliado ao tempo disponível para terminar o trabalho, impediu a utilização das restrições OCL na avaliação das soluções encontradas.

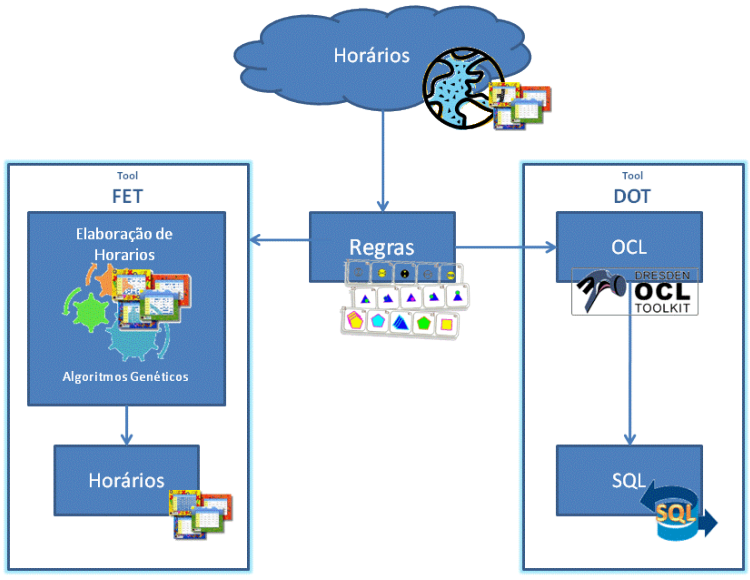


Figura 1 - Representação do caso de estudo

1.3. Objectivos específicos

Os objectivos específicos consistem em:

- Representação do problema e identificação do algoritmo mais adequado;
- Identificação das restrições;
- Representação e validação das restrições em OCL;
- Conversão das restrições representadas em OCL para SQL.
- Avaliar a melhor forma de resolver o problema de elaboração de horários com a utilização de algoritmos.

1.4. Importância do trabalho

Este trabalho contribui para a comunidade científica através da disponibilização de um caso de estudo real sobre a utilização do OCL. Note que são escassos os casos de estudo nesta área.

O facto de termos utilizado a ferramenta DOT (Dresden OCL Toolkit) é também um aspecto relevante deste trabalho. Trata-se de uma ferramenta que ainda reside no mundo universitário e que, por essa razão, ainda não pode ser testada de uma forma robusta. A utilização da ferramenta foi benéfica para o presente trabalho, mas também para a Universidade que a está a desenvolver, devido aos testes efectuados neste trabalho e pela melhoria da ferramenta de acordo com os problemas reportados. Foi necessário desenvolver um esforço considerável para conseguir obter os resultados pretendidos com a ferramenta.

Para uma instituição de ensino é muito importante ter bons horários, para que a gestão do tempo e dos recursos seja feita de forma eficiente. Devido à sua complexidade matemática, pode-se dizer que a criação de horário é um problema difícil de ser modelado, principalmente quanto às características da instituição, tais como cursos, unidades curriculares oferecidas nos semestres/trimestres, número de docentes e salas.

Encontrar e desenvolver uma solução (apesar de parcial) para uma escola em concreto, é outra mais-valia importante deste trabalho.

1.5. Organização do documento

A forma de organização deste trabalho segue uma sequência lógica para familiarizar o leitor com todos os pormenores necessários para o entendimento do modelo proposto.

A estrutura do trabalho é como o descrito a seguir:

O capítulo um consiste de uma apresentação preliminar do trabalho, constando os objectivos e importância.

O capítulo dois faz uma breve descrição da problemática da elaboração de horários e a apresentação dos métodos de solução encontrados na bibliografia, ressaltando os aspectos positivos e negativos de cada abordagem. Também aborda a linguagem OCL e ferramentas utilizadas para a definição da mesma.

O capítulo três faz uma descrição exaustiva do problema de geração de horários.

O capítulo quatro é dedicado à aplicação prática do modelo, a fim de ilustrar o desempenho do método proposto.

As conclusões obtidas em decorrência do desenvolvimento e da aplicação da metodologia proposta são expostas no capítulo cinco.

No final são colocados dois anexos (A e B). No Anexo A é mostrada a instalação da ferramenta Eclipse. O Anexo B contém todas as restrições utilizadas no modelo.

2. Levantamento do estado da arte

Conforme descrito por Qu (Qu, 2002), um Problema de Horário Educacional Geral consiste em fixar vários eventos (exames, cursos, reuniões, aulas etc.) num número limitado de períodos de tempo e local, enquanto se minimiza as violações de um conjunto de restrições. Para Burke e Newall (Burke & Newall, 1999), a essência do problema de horário consiste na alocação de um número de eventos a um número finito de espaços de tempo (ou períodos), em que as necessidades de restrições são satisfeitas.

A solução do problema consiste, então, em gerar horários, que minimize os conflitos, maximize preferências, compacte horários dos docentes e estudantes e utilize equipamentos e salas disponíveis de forma eficiente. De acordo com Souza, Costa e Guimarães (Souza, Costa, & Guimarães, 2002), a essas exigências somam-se as expectativas, imposições e desejos da administração escolar, estudantes e docentes, em relação a um determinado horário de aula a ser adoptado na prática.

Este problema pode admitir uma grande variedade de formulações, em função do tipo de instituição de ensino (escola ou universidade) e do sistema educacional no qual ela está inserida. No entanto, pode-se observar que, em qualquer caso, para a ocorrência de uma determinada aula, é necessário que todas as entidades envolvidas (docentes, estudantes e salas) estejam disponíveis no intervalo de tempo definido para o acontecimento da mesma.

Devido à dificuldade de se obter uma solução manual, à característica combinatória e a difícil generalização (em virtude da quantidade de variáveis que o problema pode assumir), diversos algoritmos heurísticos vêm sendo propostos, com o objectivo de tentar resolver diferentes aspectos do problema.

Segundo Burke e Newall (Burke & Newall, 1999) existem diversas variações no problema de construção de horários. A elaboração de horários de escola superior pode ser dividida em duas categorias principais: aulas e exames. Aulas são os conteúdos ministrados pelos docentes, e exames são as provas às quais os estudantes devem submeter-se para aprovação na unidade curricular.

A construção de horários deve ter em consideração os interesses dos docentes, dos estudantes, e rentabilizar da melhor forma possível os recursos disponíveis, sendo uma questão primordial na administração escolar, em qualquer nível de ensino. É um problema clássico que historicamente vem sendo solucionado, ou de forma

artesanal ou através de soluções empíricas, típicas apenas da própria instituição que as desenvolveram.

O problema da construção de horários escolares despertou o interesse da comunidade académica a partir do início da década de sessenta. Uma das primeiras referências a esse problema é apresentada no artigo de Appleby, Black e Newman (Appleby, Blake, & Newman, 1961). Os autores apresentavam técnicas para a construção da solução para o problema por meio de utilização de computadores e descreveram a comparação entre o problema dos horários e outros problemas de agendamento conhecidos na época.

Em 1962, Gotlieb (Gotlieb, 1962) apresentou a primeira expressão completa para o problema dos horários para cursos (*Timetabling Problem-Course Scheduling*), ao afirmar que o problema consistia em fixar, num determinado período de tempo conhecido, um conjunto de aulas, de forma a atender às restrições estabelecidas por um certo currículo de estudos para um certo grupo de unidades curriculares. Cada aula era apenas para um único grupo de estudantes, leccionada apenas por um único docente. Nesse trabalho, considerava-se que o número de vezes em que ocorreriam aulas entre estudantes e docente podia ser escolhido livremente e, adicionalmente, que o docente estava totalmente disponível para leccionação das aulas que lhe fossem agendadas, independentemente do horário das mesmas.

Em 1964, Broder (Broder, 1964) baseia-se no problema de horários para exames (*timetabling problem-examination scheduling*), tendo como objectivo minimizar os conflitos de horários para a ocorrência de exames finais de uma comunidade de estudantes de uma instituição superior. Na abordagem adoptada utilizou o algoritmo *Largest Degree First* e algoritmo de *Monte Carlo*, em caso de empate, no qual a solução com o menor número de conflitos era a escolhida, dentro de um conjunto de soluções aleatoriamente obtidas. No mesmo ano, Cole (Cole, 1964) apresentou um algoritmo que viabiliza a introdução de restrições como: certos conjuntos de exames podem ser consecutivos; estabelecimento de ordem de precedência para alguns exames; restrições quanto à capacidade, em número de estudantes, das salas de aulas; certos exames podem ser associados a determinados períodos (pela manhã, no caso).

A utilização de uma matriz de horários (de exames, no caso) para representar a existência, ou não, de qualquer tipo de conflito entre pares de exames foi um grande avanço do trabalho de Cole. Considera-se que, se dois exames, denominados i e j, tivessem qualquer tipo de conflito, então, ao elemento [i,j] da matriz de horários deveria ser associado o valor um; caso contrário, deveria ser associado o valor zero. A abordagem torna possível através da matriz de horários a visualização dos

conflitos de uma maneira simples e de fácil manipulação.

Em 1964, Berghuis, Heiden e Bakker (Berghuis, van der Heiden, & Bakker, 2005), em 1965, e Barraclough (Barraclough, 1965), propuseram métodos para a automatização da geração de horários utilizando computadores como ferramenta para a proposição de soluções práticas.

Nos anos imediatamente seguintes, a pesquisa com diversas abordagens do problema dos horários escolares aprofundou-se, diversificou-se e fundamentou-se, com especial atenção à modelação de casos reais.

Welsh e Powell (Welsh & Powell, 1967), estabelecem a relação de similaridade existente entre o problema de geração de horários e o problema da coloração de grafos. O referido problema foi demonstrado como sendo um exemplar da família dos problemas NP-Completos, por Karp (Karp, 1972).

Os resultados teóricos obtidos até o fim da década de 1960 já satisfaziam boa parte das necessidades de aplicações práticas dessa classe de problema existentes naquele momento, desde que os tipos e a quantidade de restrições impostas a uma instância particular do problema não fossem complexos ou numerosos.

No início da década de 1980, Schmidt e Stöhlein (Schmidt & Ströhlein, 1980) apresentam um estudo do estado da arte dos trabalhos já realizados sobre o tema até aquele momento.

As abordagens interactivas também fizeram parte das alternativas de solução da década de 1980, como a apresentada por White e Wong (White & Wong, 1988) em 1988. Na mesma abordagem, porém mais recente e abrangente, existem os trabalhos de Feldman e Golumbic (Feldman & Golumbic, 2005), de Gupta e Akhter (Gupta & Akhter, 2000), entre outros.

Caldeira (Caldeira & Rosa, 1997) discute a implementação de dois algoritmos genéticos apresentados por Holland (Holland J. H., 1975), utilizados para resolver o problema de horários escolares para pequenas escolas, fazendo uma comparação entre os resultados obtidos pelas duas abordagens propostas.

Fang (Fang, 1994) investiga a utilização de algoritmos genéticos para resolver um grupo de problemas de geração de horários. Nesse trabalho é proposta uma *framework* para a utilização de algoritmos genéticos para a resolução do problema no contexto de instituições de ensino. Esta *framework* possui como pontos flexíveis: a declaração das restrições específicas do problema; a utilização de uma função para avaliar a soluções, também específica para o problema; e a utilização de um algoritmo genético que é independente do problema considerado. Mostra que os algoritmos genéticos são bastante efectivos e úteis para a resolução de problemas de

geração de horários.

Fernandes (Fernandes, Caldeira, & Rosa, 2002) classifica as restrições para o problema de geração de horários escolar em restrições fortes e fracas. Violações às restrições fortes (como, por exemplo, a alocação de um docente em duas salas diferentes ao mesmo tempo) resultam numa solução inválida. Violações às restrições fracas resultam em soluções válidas, mas com menos qualidade da solução (por exemplo, a preferência dos docentes por determinados horários). Fernandes descreve um método para a resolução do problema baseado em algoritmos evolucionários.

Abramson (Abela & Abramson, 1991) apresenta um algoritmo genético paralelo para o problema. Nesse trabalho é feita uma comparação com os algoritmos genéticos convencionais, concluído que a abordagem paralela pode ser até 9,3 vezes mais rápida que a abordagem sequencial, para as instâncias do problema consideradas.

2.1. Os métodos de optimização

Quanto aos caminhos propostos para a solução, pode-se indicar os principais tratamentos encontrados na literatura nos seguintes conceitos básicos: Teoria dos Grafos, Programação Lógica com Restrições, Algoritmos Genéticos ou Meméticos, entre outros.

2.1.1. Algoritmos Genéticos

Os Algoritmos Genéticos são algoritmos que se baseiam na Teoria da Evolução através de selecção natural proposta por Charles Darwin no século XIX. A ideia fundamental dos Algoritmos Genéticos é adquirir e tratar possíveis soluções de problemas como "indivíduos" de uma "população" que "evoluirá" a cada iteração ou "geração". É qualquer modelo computacional baseado em população, que utiliza operadores de cruzamento e mutação para gerar novos pontos amostrais num espaço de procura. O maior interesse no algoritmo genético está em usá-lo como ferramenta de optimização, porque se trata de uma poderosa ferramenta para procura de soluções de problemas de alta complexidade.

Baseado na analogia com o processo de evolução biológica das espécies, os algoritmos genéticos mantêm uma determinada informação relevante sobre o ambiente e a acumulam durante o período de adaptação. Posteriormente, utilizam tal informação acumulada para minimizar o espaço de procura e gerar novas e melhores soluções dentro de um domínio.

O funcionamento básico pode ser descrito da seguinte forma: inicialmente é gerada

uma população, aleatoriamente, com indivíduos que podem ser vistos como possíveis soluções do problema. Durante o processo evolutivo, é calculado o grau de adaptação de cada indivíduo, através de uma função de custo, sendo que os mais adaptados, ou seja, os que possuem baixa penalização são seleccionados enquanto os outros são eliminados através do mecanismo de selecção natural. É gerada uma nova população que pode ter sofrido modificações nas suas características fundamentais através de mutações e cruzamentos ou recombinação genética de acordo com as probabilidades definidas no algoritmo. Após a reprodução, temos uma nova população que deverá ser avaliada, repetindo o processo descrito anteriormente. Como critérios de paragem fixamos um limite (solução óptima) e uma tolerância (uma solução viável). Deve-se também especificar um número máximo de gerações. Após esse processo, não havendo violação das condições descritas no algoritmo, este retorna uma solução viável e não necessariamente óptima.

Para Costa e Bruna (Costa & Bruna, 2003) o problema foi resolvido utilizando algoritmos genéticos apresentando resultados satisfatórios em relação ao tempo de processamento e execução.

Peachter, Luchian e Petriuc (Paechter, A., Luchian, & Petriuc, 1994) desenvolveram o sistema Neeps & Tatties, o qual está sendo usado para programar cursos no Departamento de Ciências da Computação da Universidade de Napier. Este algoritmo genético codifica os horários como uma sequência de eventos, que deve ser introduzida num programa especial que usa a ordem para produzir um horário. Para isto é necessário um tipo diferente de operador de recombinação, que toma elementos da sequência de cada pai para produzir uma nova sequência.

2.1.2. Algoritmos Meméticos

Trata-se de uma meta heurística pertencente à classe dos algoritmos populacionais. Um algoritmo populacional usa várias soluções na procura de soluções viáveis no espaço de procura. Os Algoritmos Meméticos têm sido amplamente estudados e aplicados a vários problemas de optimização encontrados na literatura. Uma ampla lista de aplicações pode ser encontrada em trabalhos realizados por Cotta e Moscato (Cotta & Moscato, 2003). O termo "Algoritmo Memético" foi introduzido por Moscato (Moscato, 1989) para descrever um processo evolutivo que tenha a procura local como parte decisiva na evolução.

Esta procura pode ser caracterizada como sendo um refinamento local dentro de um espaço de procura, de modo que um agente (indivíduo autónomo) pode ter o seu

nível de adaptação aumentado após passar por uma etapa de refinamento. Segundo Cotta e Moscato (Cotta & Moscato, 2003), a ideia básica que sustenta os Algoritmos Meméticos é a combinação de conceitos e estratégias de diferentes meta heurísticas, como a procura baseada em populações e as técnicas de procura local, com a intenção de unir as vantagens das mesmas. Conforme Norman e Moscato (Norman & Moscato, 1992), a ideia principal é explorar a vizinhança das soluções obtidas por um algoritmo genético e caminhar em procura do óptimo local (para cada solução) antes de retornar para o algoritmo genético e continuar o processo.

Assim como os Algoritmos Genéticos, os Algoritmos Meméticos baseiam-se em processos naturais, tais como recombinação, selecção, mutação, entre outros.

Enquanto o algoritmo genético é baseado na evolução biológica dos indivíduos, o algoritmo memético é baseado na evolução cultural dos mesmos. Deve-se ressaltar que este processo não se limita simplesmente a transmitir a informação inalterada: ela também é processada e aumentada, pela comunicação com os outros agentes.

Algoritmos Meméticos utilizam ainda o conceito de "evolução cultural", onde a adaptabilidade de um indivíduo pode ser modificada no decorrer da sua existência dentro da população. No caso da geração de horários, esta evolução cultural poderia ser representada por mudanças em características como ausência de aulas aos sábados ou em mais de uma aula.

Um indivíduo pode ser geneticamente pouco favorecido ao nascer, mas devido às condições em que vive, por trocas de informação com outros indivíduos, experiências pessoais entre outros aspectos, pode-se tornar mais adaptado, e mais do que isso, transmitir essa experiência aos seus descendentes (evolução cultural). As etapas básicas deste algoritmo podem ser descritas da seguinte maneira: inicialmente gera-se uma população constituída por um número fixo de soluções possíveis dentro do espaço de soluções do problema. Em seguida, avalia-se a população através de uma função de avaliação que mede a qualidade da solução representada por cada agente, ou seja, quanto melhor for a solução representada maior será o valor da sua função de avaliação. A população é modificada através de recombinação, troca de *memes* (uma unidade análoga ao gene dos Algoritmos Genéticos tradicionais) entre dois agentes, ou de mutação, mudança aleatória no valor dos *memes* para garantir a diversidade de soluções e que é intensificada nas crises de diversidade quando ocorre uma estagnação da população.

2.1.3. Simulated Annealing

Um algoritmo de *Simulated Annealing* pertence a uma classe de algoritmos de

pesquisa local que também são conhecidos sob a designação genérica de "algoritmos de limiar" (do inglês, Threshold Algorithms).

É um método de optimização baseado em conceitos da física, utilizados na termodinâmica, sendo tratado inicialmente em trabalhos realizados por Kirkpatrick, Gellat e Vecchi (Kirkpatrick, Gellat, & Vecchi, 1983). O nome "recozimento" (annealing) é dado ao processo de aquecimento de um sólido até ao seu ponto de fusão, seguido de um arrefecimento lento e gradual, até que se alcance novamente o seu estado sólido. É uma técnica de localização local, que obtém boas soluções sobre problemas com muitas restrições. O processo inicia-se com um membro qualquer do espaço de soluções, normalmente gerado aleatoriamente, e selecciona um dos seus vizinhos (indivíduos) aleatoriamente. Se este vizinho for melhor que o anterior ele é aceite e substitui a solução corrente. Se o novo vizinho for pior, ele pode ser aceite de acordo com uma probabilidade relacionada a um parâmetro de controlo chamado "Temperatura", que é assim chamado por analogia ao fenómeno físico. A temperatura assume, inicialmente, um valor elevado. Após um número de iterações, a temperatura é gradativamente diminuída por uma razão de arrefecimento. Quando a temperatura está elevada, praticamente todos os movimentos são autorizados como no caso da matéria em fusão onde a energia do sistema é muito elevada.

Quanto mais a temperatura baixa, mais as degradações vão sendo penalizadas em função de lucro dos movimentos melhores, fazendo o método convergir para o óptimo local mais próximo. Esse processo é repetido até que a temperatura seja tão pequena que mais nenhum movimento seja aceite. A melhor solução encontrada durante a procura é tomada como uma boa aproximação para a solução óptima.

Johnson (Johnson, 1990) utilizou as heurísticas num problema de horário de exames que atingiu 96050 variáveis e 287240 restrições num modelo linear inteiro.

2.1.4. Tabu Search

Como o *Simulated Anneling*, o *Tabu Search* mantém um rastro de possíveis soluções para o horário corrente. A diferença está no método pelo qual move novos horários para a lista de aceites.

É uma estratégia de procura local desenvolvida para encontrar soluções quase óptimas para problemas de optimização. A técnica utiliza uma estrutura de memória para guiar a procura e continuar a exploração do espaço de soluções mesmo que não haja movimento de melhoria, evitando a formação de ciclos, isto é, o retorno a um óptimo local previamente visitado. A palavra *tabu* sugere algo proibido ou pelo menos inibido.

Utiliza restrições *tabu* para impossibilitar certos movimentos e alguns procedimentos denominados critérios de aspiração são utilizados para decidir quando os movimentos classificados como *tabu* podem ser executados. As restrições *tabus* são controladas por uma lista *tabu*, que define todos os movimentos que têm um certo atributo como sendo *tabu* por um determinado número de iterações, que memoriza os últimos movimentos executados. Tais movimentos são proibidos, a menos que a solução satisfaça um certo critério de aspiração, que é uma técnica que permite aceitar certos movimentos *tabu* julgados interessantes para o prosseguimento da pesquisa. Em geral, espera-se que essa solução seja melhor que a melhor solução encontrada até então. A lista *tabu*, usualmente, é de tamanho fixo e quando um novo movimento é adicionado à lista, ao mesmo tempo, o mais antigo é removido. Quando um movimento atinge um nível de aspiração, pode ser removido da lista *tabu*, sendo aceite como solução viável ou não.

Tabu Search foi aplicado com sucesso por Bouffet e Nègre (Boufflet & Nègre, 2006) para gerar os horários na Universidade de Tecnologia de Compiègne. As suas listas de tabelas contêm os sete movimentos mais recentes. Se a vizinhança corrente não contiver uma solução melhorada, a função de aspiração pode seleccionar uma solução da lista de tabelas.

Hertz (Hertz, 1992) desenvolveu e aplicou o algoritmo *Tabu Search* TATI no agendamento de cursos, o qual adaptou mais tarde para um problema mais complexo e com mais restrições. A duração de uma aula não é fixada, e existem dez diferentes tipos de movimentos (mover uma aula para outro dia, mudar a duração das aulas, etc.) Quando o horário de uma aula num dia particular é modificado, pode ser mudado para outro período (possivelmente noutro dia). Porém, para um determinado número de interacções esta tabela move a aula para um período no dia original.

2.1.5. Teoria de grafos

O problema da geração de horários pode ser caracterizado como um modelo de procura em grafos, considerando como ponto de partida os eventos previamente alocados, para os quais horários e salas apresentam-se a *priori*.

Na Teoria de Grafos, um grafo é uma estrutura formada por vértices e arestas. Pode ser interpretado como uma tripla *(N,A,G)* onde: *N* é um conjunto não vazio de nós ou vértices, *A* é um conjunto de arcos ou arestas e *G* é a função que associa a cada arco

um par não ordenado de nós, denominados terminações de *A*. O processo de coloração de grafos consiste na atribuição de cores aos vértices que satisfazem as seguintes propriedades: vértices adjacentes recebem cores diferentes e a menor quantidade possível de cores deve ser utilizada. Noutras palavras, uma coloração de vértices é uma função *G: V -> C* que associa a cada vértice $v \in V$ do grafo uma cor e de modo que $G(v) \neq G(w)$ sempre que *v* é adjacente a $w \in V$. Uma k-coloração de um grafo é uma coloração que utiliza um total de *k* cores. Denomina-se número cromático de um grafo o menor número de cores *k*, para o qual existe uma k-coloração. Em analogia com o problema de geração de horários, cada unidade curricular é representada por um vértice de um grafo. Um arco conecta dois vértices se as unidades curriculares correspondentes serão escolhidas pelo mesmo estudante ou serão ministradas pelo mesmo docente. O algoritmo de solução procura por uma coloração dos nodos de forma que quaisquer dois vértices unidos por um arco não possuam a mesma cor. Além disso, o algoritmo minimiza o número de cores utilizadas.

2.1.6. "branch-and-bound"

O método "*branch-and-bound*" é utilizado para optimizar a procura por soluções inteiras para um problema linear, de forma iterativa intercalado com o método *Simplex* para problemas lineares. O método percorre os vértices do conjunto compacto de soluções do problema. Quando o problema só admite soluções inteiras, o conjunto é formado por um número não necessariamente finito de pontos do $\Re_n$. Cada dimensão corresponde a uma variável a ser optimizada. O algoritmo *Simplex* pode ser visto como um processo combinatório que procura encontrar as colunas da matriz de restrições que induzem uma base e, portanto, uma solução básica óptima. A dificuldade advém do facto que tipicamente existe um número exponencial de possíveis combinações de colunas, gerando portando um desempenho de pior caso de ordem exponencial.

O algoritmo "*branch-and-bound*" (Algoritmo de Bifurcação e Limite) pode ser entendido como uma extensão da estratégia de divisão e conquista para problemas de natureza inteira mista, ou seja, divida um problema *P* num conjunto de subproblemas {SPk} de forma que a solução de P possa ser obtida através da solução dos subproblemas, resolva os subproblemas e, no final, obtenha a solução do problema de interesse.

Neste algoritmo as divisões são feitas iterativamente, sempre observando que os subproblemas devem ser mais fáceis de serem resolvidos que o problema original, além de se procurar descartar subproblemas por meio de enumeração implícita; isto

equivale a dizer que, de alguma forma, podemos garantir que a solução óptima não é solução de um certo subproblema e, por conseguinte, podemos descartá-lo.

2.1.7. Pesquisa em Vizinhança Variável

O Método de Pesquisa em Vizinhança Variável, conhecido como VNS (*Variable Neighborhood Search*), proposto por Hansen e Mladenovic (Hansen & Mladenović, 2006), é um método de pesquisa local que consiste em explorar o espaço de soluções através de trocas sistemáticas de estruturas de vizinhança. A ideia do método é a de explorar vizinhanças gradativamente mais "distantes" da solução corrente, focando a procura em torno de uma nova solução se e somente se um movimento de melhoria é realizado. O método VNS também inclui um procedimento de pesquisa local a ser aplicado sobre o vizinho da solução corrente.

Contrariamente às outras meta heurísticas baseadas em métodos de pesquisa local, o método VNS não segue uma trajectória, mas sim explora vizinhanças gradativamente mais "distantes" da solução corrente e focaliza a procura em torno de uma nova solução se e somente se um movimento de melhora é realizado. O método inclui, também, um procedimento de pesquisa local a ser aplicado sobre a solução corrente. Esta rotina de pesquisa local também pode usar diferentes estruturas de vizinhança.

Na sua versão original, o método VNS faz uso do método VND (*Variable Neighborhood Descent*) para fazer a pesquisa local.

Neste algoritmo, parte-se de uma solução inicial qualquer e a cada iteração selecciona-se aleatoriamente um vizinho s' dentro da vizinhança $N(k)(s)$ da solução s corrente. Este vizinho é então submetido a um procedimento de procura local. Se a solução óptima local, s'', for melhor que a solução s corrente, a procura continua de s'' recomeçando da primeira estrutura de vizinhança $N(1)(s)$. Caso contrário, continua-se a procurar a partir da próxima estrutura de vizinhança $N(k+1)(s)$. Este procedimento é encerrado quando uma condição de paragem for atingida, tal como o tempo máximo permitido de CPU, o número máximo de iterações ou número máximo de iterações consecutivas entre dois melhoramentos. A solução s' é gerada aleatoriamente de forma a evitar ciclos, situação que pode ocorrer se alguma regra determinística for usada.

2.1.8. Programação Lógica

Com o surgimento do Prolog, para Yunes (Yunes, 2000), surgiu um novo paradigma de programação que permite ao programador preocupar-se com "o quê", a lógica, sem se preocupar com "o como", o controlo. Os sistemas baseados em lógica usam um conjunto de regras fornecidas pelo utilizador para responder às perguntas solicitadas. Na maioria dos casos, estes sistemas são usados para comprovar uma dada afirmação, ou para determinar qual o domínio das variáveis a uma pergunta onde será atribuído o valor verdade.

O mecanismo usado pela programação lógica para solucionar um problema é uma procura em profundidade em árvore com todas as possíveis derivações lógicas (Colmerauer, 1973), da esquerda para a direita, obtidas a partir de um objectivo, respeitando as regras informadas. Como o sistema desconhece qual ou quais das várias alternativas podem levar a uma solução satisfatória, ele deve percorrer todos os caminhos ou tentar inferir qual o correcto. Neste tipo de procura, uma solução só pode ser determinada no momento em que todos as incógnitas possuírem um valor definido ou não for possível a continuidade do processo.

Yunes (Yunes, 2000) cita que isto gera um problema típico de explosão combinatória e inviabiliza o uso da programação lógica em aplicações de médio e grande porte, porquê os tempos computacionais tornam-se elevados, caso uma procura ocorra em ramos da árvore onde não exista solução viável. Outra restrição ao uso da programação lógica é que os objectivos manipulados são estruturas não interpretadas e, por isso, a igualdade ocorre apenas entre aqueles objectivos que são sintacticamente idênticos.

2.1.9. Programação Lógica com Restrições

A Programação Lógica com Restrições (*Constraint Logic Programming, CLP*) é a tentativa de superar as limitações apresentadas pela programação lógica por meio da adição de mecanismos responsáveis pela resolução de restrições. Neste sentido, podemos ver a CLP como uma extensão de um sistema de programação lógica, no qual se introduzem estruturas de dados mais ricas e complexas, as quais permitem que, objectos semânticos como expressões aritméticas sejam codificadas e manipuladas directamente. A ideia básica consiste em complementar o núcleo computacional do sistema lógico, a unificação, com um manipulador de restrições num determinado domínio (Jaffar & Lassez, 1987).

A CLP é usada para explorar um conjunto de possibilidades para resolução do modelo, esta exploração é feita com o uso de métodos de pesquisa. As restrições são

usadas para minimizar o espaço de soluções com a eliminação de alternativas inexequível na solução do problema, ao contrário da programação lógica que explora toda a árvore de procura.

2.1.10. Greedy Randomized Adaptive Search Procedures

Greedy Randomized Adaptive Search Procedure (GRASP), ou procedimento de procura adaptativa gulosa e aleatória, é uma técnica iterativa proposta em Feo e Resende (Feo & Resende, 1995), que consiste em duas fases: uma fase de construção, na qual uma solução é gerada, elemento a elemento, e outra fase de procura local, na qual um óptimo local na vizinhança da solução construída é pesquisado. A melhor solução encontrada ao longo de todas as iterações GRASP realizadas é retornada como resultado do algoritmo de optimização GRASP. Na Fase de Construção, uma solução é iterativamente construída, elemento por elemento. A cada iteração dessa fase, os próximos elementos candidatos a serem incluídos na solução são colocados numa lista de candidatos, seguindo um critério de ordenação pré determinado.

O processo de selecção é baseado numa heurística adaptativa gulosa que estima o benefício da selecção de cada um dos elementos. A fase de procura local começa com uma solução obtida pela fase de construção GRASP, navega pelo espaço de pesquisa passando de uma solução para outra, que seja sua vizinha, em procura de um óptimo. A eficiência da procura local depende da qualidade da solução construída na fase de construção. O algoritmo GRASP procura, portanto, conjugar bons aspectos dos algoritmos puramente gulosos, com aqueles dos algoritmos aleatórios de construção de soluções.

Segundo Feo & Resende (Feo & Resende, 1995), o procedimento GRASP procura conjugar bons aspectos dos algoritmos puramente gulosos, com aqueles dos procedimentos aleatórios de construção de soluções.

2.1.11. Abordagem pela programação Linear

A programação linear é uma técnica matemática que tem por objectivo "*encontrar a melhor solução para problemas que tenham os seus modelos representados por expressões lineares*" (Bregalda, Oliveira, & Bornstein, 1988)

Os problemas de programação linear são um tipo de problema de optimização e referem-se à distribuição eficiente de recursos limitados entre actividades competitivas com a finalidade de atender a um determinado objectivo. Em tratando-se de programação linear, esse objectivo será expresso por uma função linear, à qual

dá-se o nome de função objectivo. As informações de proporção de consumo e designação de recursos consumidos são fornecidas por equações ou inadequações lineares que são chamadas restrições (Puccini, 1981).

Akkoyulu (Akkoyunlu, 1973) sugere um algoritmo de programação linear para solucionar o problema do horário de aulas, considerando um custo associado à escolha de um horário para uma aula.

A função objectivo associada ao problema é uma função de minimização, com os valores de custo positivos representando as escolhas desejáveis e os valores negativos as escolhas indesejáveis.

Akkoyulu (Akkoyunlu, 1973) estabelece uma constante na função objectivo, que recebe um valor arbitrariamente elevado para assegurar a associação de uma aula a cada horário disponível, pois em razão do seu valor, toda a solução procura alocar mais de um horário a uma aula. Demonstra que o problema assim formulado, apresenta uma solução inteira se substituirmos as restrições de 0 ou 1 por >= 0. Como consequência, é possível aplicar-se os métodos comuns de programação linear em substituição dos métodos de programação inteira.

Numa segunda fase porém, Trevelim (Trevelin, 1983) faz uso do Método *Simplex* com uma nova matriz de restrições, incluindo a disponibilidade de salas de aula, interligando os cursos, porque cada um concorre às salas disponíveis em cada horário de aula da semana. Nesta fase surgem ainda restrições suplementares, relativo à indisponibilidade dos docentes, que são levadas em consideração. É nesta fase que as maiores dificuldades aparecem, principalmente em razão do grande número de restrições.

2.1.12. Comparação de Algoritmos

Os algoritmos estudados encontram-se sumarizados na Tabela 2. A descrição de cada coluna segue abaixo:

Algoritmo – Nome do algoritmo;

Baseado – "Em quê?" que se baseia o algoritmo;

Síntese – Pequeno resumo de como o algoritmo funciona.

Algoritmo	Baseado	Adequado a que tipo de problemas	Vantagens	Desvantagens
Algoritmos Genéticos	Teoria da evolução biológica	Problemas de alta complexidade	• São fáceis de combinar com outros métodos.	• Pode percorrer soluções descartadas anteriormente; • Retorna uma solução viável e não necessariamente óptima. • Pode ser bem mais lentos que outros métodos por trabalharem com uma população de soluções.
Algoritmos Meméticos	Teoria da evolução cultural	n/a	n/a	• Pode percorrer soluções descartadas anteriormente.
Simulated Annealing	Algoritmos de pesquisa local	Problemas com muitas restrições	• Aproxima-se da solução óptima.	• Reduz muito a lista de soluções possíveis.
Tabu Search	Algoritmos de pesquisa local	Problemas de pequena dimensão (lista com tamanho definido)	• Evita a formação de ciclos.	• Reduz muito a lista de soluções possíveis (lista de *tabu* é de tamanho fixo).
Teoria de grafos	Modelo de procura em grafos	Problemas de pequena dimensão	n/a	• Apresenta um crescimento exponencial.
branch-and-bound	Problemas lineares	Problemas de pequena dimensão	• Pode ser aplicado a problemas de programação inteira mista.	• Como particiona o problema principal em subproblemas, a enumeração completa é impraticável quando o número de variáveis de um problema é grande.
Pesquisa em Vizinhança Variável	Algoritmos de pesquisa local	Problemas de pequena dimensão	n/a	n/a

Programação Lógica	Sistemas baseados em lógica	Problemas de pequena dimensão	n/a	• Gera um problema típico de explosão combinatória.
Programação Lógica com Restrições	Sistemas baseados em lógica	Problemas com muitas restrições	• Reduz o esforço de programação e torna mais natural a programação; • Encontra as soluções onde as restrições sejam satisfeitas; • Facilidade de manutenção.	• n/a
GRASP	Procura adaptativa gulosa e aleatória	n/a	• Rápido; • Geralmente produz boas soluções.	• Acha uma solução inicial de forma gulosa, e melhora a solução através de uma busca local, isto produz pouca variedade na solução, podendo assim, ficar preso a um mínimo local.
Abordagem p/ prog. Linear	Técnicas matemáticas	Problemas de pequena dimensão	n/a	• Apresenta um crescimento exponencial.

Tabela 1 – Síntese dos Algoritmos

2.2. OCL

A UML é uma linguagem de modelação bastante utilizada, fornecendo um conjunto de convenções de diagramas que são usadas para auxiliar a delineação e documentação dos sistemas de *software*.

A versão 1.0 da UML foi bastante criticada por ser definida com muito pouco rigor. A definição foi criada utilizando apenas os elementos gráficos da própria UML, por meio de um meta-modelo MOF (*Meta-Object Facility*). Esta abordagem permitia uma grande ambiguidade na interpretação dos modelos. Um primeiro passo em direcção a uma maior precisão da especificação UML foi a criação da linguagem textual OCL

(*Object Constraint Language*) (OMG) (Warmer & Kleppe, 2003), baseada numa linguagem desenvolvida por um grupo de pesquisa da IBM para definição de restrições em modelos.

OCL é uma linguagem criada especificamente para descrever expressões e regras complexas (Morgan, 2001). O objectivo da criação desta linguagem foi alargar o poder de expressão de restrições da UML, permitindo especificar diversos tipos de regras, como *invariantes*, valores iniciais de atributos, e computação de valores derivados, desempenhando um papel importante na fase de análise do ciclo de vida do *software*. Em outras palavras, OCL é utilizado para enriquecer semanticamente a definição da UML.

Com a especificação de modelos UML contendo regras de negócio escritas em OCL e a utilização das técnicas de MDA[3], a portabilidade da aplicação será facilitada, visto que as referidas técnicas permitem que um mesmo modelo fonte possa ser transformado em modelos para várias plataformas (bastando alterar a definição de transformação correspondente à linguagem alvo).

Na abordagem tradicional, os modelos conceptuais das aplicações são transformados manualmente para as linguagens computacionais específicas, de modo que, caso seja necessário alterar a plataforma de implementação, todo o trabalho de transformação deve ser refeito. Para resolver este problema, foi desenvolvida uma *Framework* que utiliza UML e outros padrões da OMG para desenvolver *software* baseado em modelos abstractos (Kleppe, Warmer, & Bast, 2003).

Este desenvolvimento orientado por modelos permite especificar os conceitos relevantes do domínio da aplicação, as regras de negócio, os seus relacionamentos e semântica usando várias linguagens de modelação conceitual. Como mostra a Figura 2, a estratégia de transformação de modelos facilita o mapeamento automático de modelos independentes de plataforma (PIM - Platform Independent Model) para modelos voltados para plataformas específicas (PSM - Platform Specific Model), podendo ser utilizada para gerar diagramas de base de dados e restrições de integridade sobre eles.

[3] MDA (Model Driven Architecture), que é uma *Framework* que utiliza UML e outros padrões da OMG (Object Management Group) para desenvolver *software* baseado em modelos abstractos.

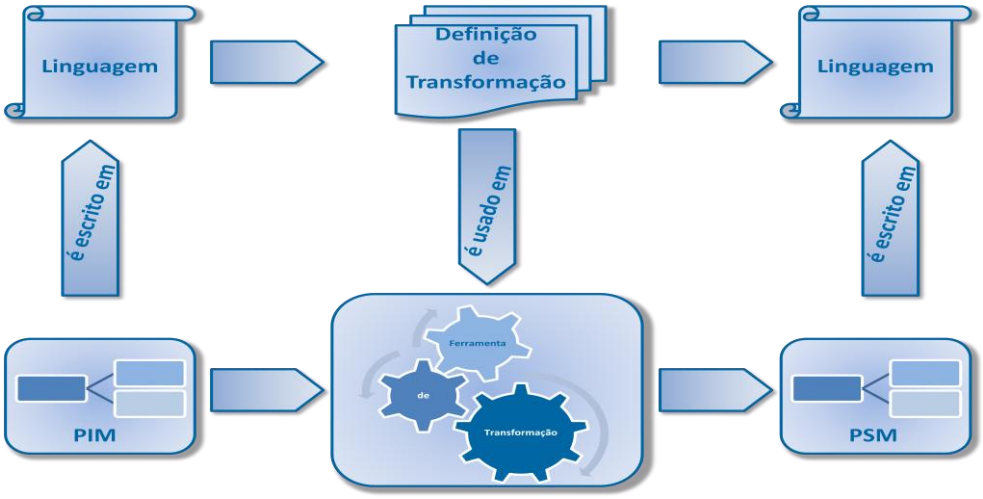


Figura 2 – Visão Geral da Abordagem MDA (Kleppe, Warmer, & Bast, 2003)

A transformação entre modelos deve ser feita por uma ferramenta de transformação que irá utilizar uma definição de transformação (mapeamento) entre os elementos da linguagem de origem para a linguagem alvo. Esta definição de transformação representa a vantagem da MDA em relação à abordagem tradicional de desenvolvimento: em caso de haver a necessidade de alterar a plataforma de implementação do modelo (por exemplo, trocar o SGBD que está a ser utilizado), basta que a definição de transformação seja alterada para incorporar elementos da nova plataforma.

Embora exista uma série de propostas para formalizar a especificação (ou parte dela) de Sistemas de Informação, a maior parte da especificação, ainda é feita em Linguagem Natural. Recentemente, tem sido difundido o uso da linguagem OCL para a representação de restrições nos modelos de Classes, e inclusive, tem sido proposto, por alguns autores, o uso da OCL para a representação das Regras de Negócio.

Com a OCL é possível testar a construção dos modelos, retirar métricas ao nível do desenho e ainda especificar vários tipos de restrições, que poderão reflectir um desenho por contracto e/ou regras de negócio, através de pré/pós condições e *invariantes*.

Como é uma linguagem formal, semelhante a uma linguagem de programação, torna-se, também, possível a criação de ferramentas para conceber código tendo como entrada os modelos e as especificações/restrições desta linguagem e como saída programas fontes em linguagens de programação ou, por exemplo, "*triggers*" para bases de dados. Além disso, existe uma série de abordagens propondo o uso de OCL para a documentação de Regras de Negócio.

Tem sido usada em diferentes contextos e com diferentes propósitos no desenvolvimento de *software*. Devos e Steegmans (Devos & Steegmans, 2005), além de Penker e Eriksson (Penker & Eriksson, 2000), utilizam OCL na modelação de regras de negócio. Um analista observa e elabora um modelo de negócio, de modo preciso: as regras da legislação vigente, da organização ou impostas pelo negócio, entre outras. Varella e outros (Varella, Pereira, Held, Zimbrão, & Silva, 2004) utilizam as regras de negócio expressas em OCL para detectar acontecimentos na base de dados implementadas de forma incorrecta.

Na especificação da UML 2.0 (OMG), as expressões OCL são utilizadas para dar maior precisão às regras de boa formação dos conceitos apresentados. Com isso, Bauerdick e outros (Bauerdick, Gogolla, & Gutsche, 2004) identificaram automaticamente inconsistências entre as expressões OCL e a estrutura de classes da especificação, por meio da ferramenta apresentada pelos autores. A partir de diagramas de classe de um modelo UML, anotados com restrições OCL, Richters e Gogolla (Gogolla & Richters, 2003) verificaram a ocorrência de violações dessas restrições na implementação respectiva em linguagem Java.

Segundo Demuth and Hussmann (Demuth & Hussmann, 1999) e Zimbrão da Silva (Silva, Almeida, Souza, Sulaiman, & Neto, 2001) as expressões em OCL podem ser traduzidas para consultas em SQL, bem como convertidas em *triggers* na base de dados.

Lavazza e Barresi (Lavazza & Barresi, 2005) propõem uma ferramenta para suportar a criação de planos de medição de processos que utilizem a metodologia GQM (Goal/Question/Metrics). Na sua abordagem, métricas complexas, que não possam ser representadas directamente por um atributo do modelo do processo, são representadas como operações, cuja semântica é descrita em OCL.

Mak, Choy e Lun (Mak, Choy, & Lun, 2004), além de Zdun e Avgeriou (Zdun & Avgeriou, 2005), utilizaram expressões OCL para modelar precisamente padrões de desenho (*design patterns*) (Gamma, Helm, Johnson, & Vlissides, 1994). Segundo os autores, com estas expressões, foi possível identificar automaticamente padrões de desenho presentes num dado diagrama de classe UML, entre outros benefícios.

Segundo Ol'khovich e Koznov (Ol'khovich & Koznov, 2003), OCL pode ser usada na verificação automática da integridade e consistência de modelos UML. Por exemplo, pode-se verificar a correspondência dos valores dos atributos de objectos nos diagramas de colaboração em relação às restrições das classes correspondentes, presentes nos diagramas de classes. Além disso, sugerem que o uso de especificações OCL possa representar uma fonte de informação adicional para a realização de engenharia directa (*forward engineering*). Diferentes ferramentas foram criadas por

diversos autores para a utilização de OCL em diagramas de classe UML (Akehurst & Patrascoiu, 2004) , (Baar, 2005) (Briand, Dzidek, & Labiche, 2004), (Hussmann, Demuth, & Finger, 2002).

Um objectivo comum entre estes autores é posteriormente gerar código que verifique se as restrições modeladas em OCL são respeitadas pela implementação respectiva.

2.2.1. Descrição do OCL

OCL é uma linguagem de expressão que descreve as restrições em linguagens orientadas a objectos e outros artefactos de modelação. Uma restrição pode ser vista como uma limitação a um modelo ou a um sistema.

Há uma necessidade de descrever as restrições sobre os objectos no modelo. Tais restrições são frequentemente descritas em linguagem natural, pelo que a prática tem demonstrado que este tipo de descrição produzirá sempre ambiguidades.

Apesar do poder de expressão de linguagens gráficas e visuais de modelação como a da linguagem UML, há propriedades e restrições de *software* que são muito complexas ou impossíveis de serem expressadas num diagrama apropriadamente por meio de uma representação exclusivamente gráfica. Mesmo os mecanismos de extensão da UML – estereótipos (*stereotypes*), valores com etiquetas (*tagged values*) e restrições (*constraints*) pré-definidas – podem se demonstrar insuficientes.

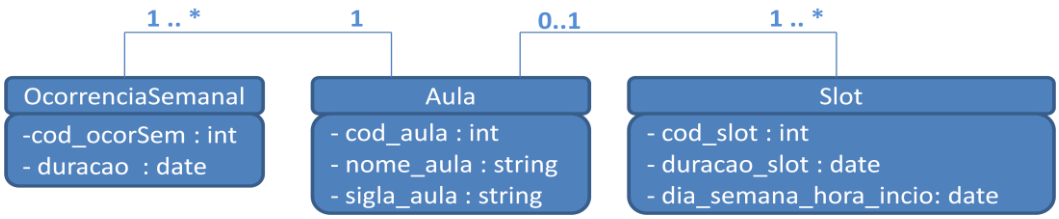


Figura 3 – Diagrama – exemplo do uso de OCL

Como é que se pode definir, no diagrama da Figura 2, que a soma dos *Slots* têm a mesma duração que a duração das ocorrências semanais de uma dada unidade curricular?

Multiplicidades podem ser representadas utilizando puramente UML, mas neste caso isso não é suficiente. A multiplicidade '*1..**' da associação entre as classes *Slot* e *Aula*, não é capaz de reflectir essa restrição, dado que a duração das aulas variam conforme a unidade curricular e o tipo de aula.

Características, como esta, não podem ser expressas utilizando apenas multiplicidades, estereótipos, devendo ser registadas nos modelos utilizando restrições em linguagem natural.

Desde a versão 1.1, a UML contempla OCL, uma linguagem puramente textual para descrição de restrições. Ela é capaz de especificar informações adicionais em diagramas UML as quais não podem ser descritas através de gráficos. Baseada na lógica de predicados e teoria de conjuntos, suportando a expressão de restrições que reflectem o paradigma de Projecto por Contrato (*Design by Contract*) (Meyer, 1997): *invariantes* (propriedades que caracterizam uma classe como um todo, sendo verdadeiras em todas as suas instâncias), *pré-condições* (condições que devem ser satisfeitas para que uma dada operação seja executada com sucesso) e *pós-condições* (condições garantidas por uma operação após o término da sua execução bem-sucedida). *Pré-condições* e *pós-condições* representam as duas partes de um contrato: se a instância cliente de uma dada operação garante que as pré-condições dessa operação sejam satisfeitas, a instância fornecedora da operação garante que a pós-condição será verdadeira, terminada a execução da operação.

OCL pode ser utilizada para diferentes finalidades (Object Constraint Language - OMG-UML V1.1, 1998):

- para especificar invariantes - Tipicamente, expressões em OCL especificam condições invariantes: uma ou mais expressões booleanas precisam ser verdadeiras para todas as instâncias de uma determinada classe de um modelo do sistema;
- para descrever as pré-condições e pós-condições em operações e métodos – Uma pré-condição precisa ser verdadeira para permitir a execução da operação; já uma pós-condição representa as modificações que devem acontecer após a execução da operação.
- pré-condições e pós-condições são restrições que, através de expressões booleanas, especificam a aplicabilidade e o efeito de uma operação sem declarar o algoritmo que implementa a respectiva operação.
- para descrever condições de Guarda (*Guards*) - Condições de guarda em diagramas de estado da UML podem ser especificadas através de uma expressão OCL. Uma condição de guarda informa a condição que deve ser atendida para que uma mudança de estado aconteça.
- especificação de regras de derivação e valores iniciais para atributos - É importante realçar a diferença entre uma regra de derivação e um valor inicial: o conteúdo derivado de um atributo deve sempre conter um valor que expressa a regra de derivação, enquanto um valor inicial (*default*) de um

atributo precisa ser garantido somente no momento de criação da instância do objecto em questão; após esse momento, o valor pode assumir qualquer conteúdo.

- especificação de expressões que podem complementar o entendimento de diagramas da UML - O diagrama de classes é beneficiado com expressões OCL, através de valores iniciais, invariantes, e pré-condições e pós-condições sobre métodos, por exemplo. Outros diagramas também podem usar, tais como: diagramas de actividades podem especificar condições e valores de parâmetros; diagramas de componentes podem especificar explicitamente as suas interfaces; pré-condições e pós-condições dos casos de uso podem ser escritos usando expressões OCL.

OCL é uma linguagem com diversos tipos de objectos. Encontrando-se organizados hierarquicamente, conforme representado na Figura 4.

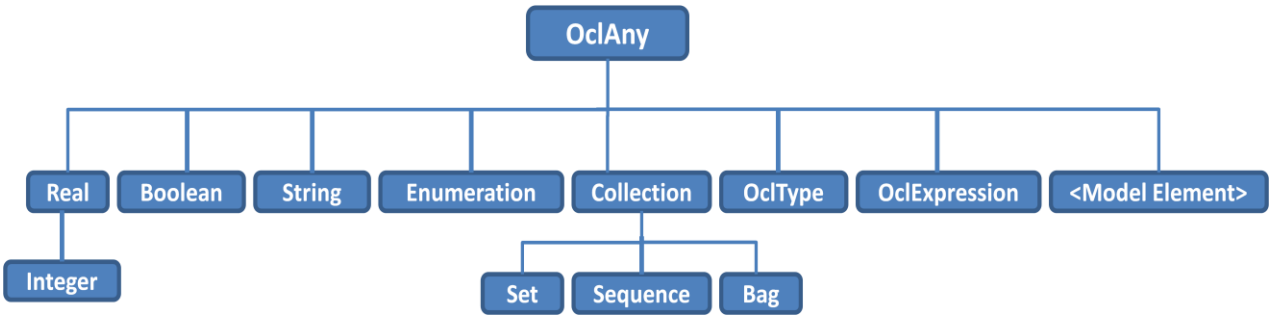


Figura 4 - Classificação hierárquica para os tipos de dados em OCL

Os principais tipos de OCL estão divididos em tipos primitivos e tipos colecção. Os tipos primitivos são: *Boolean*, *Integer*, *Real* e *String*. Os tipos colecção são: *Collection* (agrupado qualquer elemento do mesmo tipo, duplicados ou não, ordenados ou não; colecção), *Set* (conjunto de elementos do mesmo tipo, não ordenados, sem valores duplicados; conjunto matemático), *Bag* (como *Set*, mas permite valores duplicados; multi-conjunto) e *Sequence* (como *Bag*, mas os seus elementos possuem índice interno de ordenação).

As Expressões em OCL não possuem efeitos colaterais: apenas retornam um valor, não alterando nenhuma informação do modelo. A sua avaliação é instantânea – os estados dos objectos não podem ser alterados durante essa avaliação. Além disso, utilizando expressões OCL, é possível a navegação entre as classes de um modelo UML, a partir de uma classe base, que fornece o contexto da expressão.

Para "*aceder*" a um atributo ou operação, utiliza-se o operador "*.*"; para "*aceder*" a propriedades de uma colecção que pode ser definida pelas instâncias de uma classe,

ou pelas instâncias ligadas a uma dada instância de uma classe, utiliza-se o operador "->".

Para demonstrar o descrito, uma solução para a restrição do exemplo da Figura 3, pode ser formulada em OCL da seguinte forma:

context Aula inv duracaoDeUmaAula :

self.Slot ->duracao() = self.OcorrenciaSemanal->duracaoOcorrencia()

O contexto de onde parte a navegação é fornecido (*classe Aula*); a restrição é um *invariante* (palavra-chave *inv*) com o nome *duracaoDeUmaAula*.

Ao avaliar o lado esquerdo da equação, percorre-se do contexto (*classe Aula*) até o alvo da associação (*classe Slot*); como a sua multiplicidade é '*1..**', neste sentido da associação temos uma colecção de elementos, para ter acesso às propriedades desta colecção utiliza-se o operador "->". A operação executada (*duracao()*) retorna a duração de uma determinada aula. Para a avaliação do lado direito (do sinal de igual), percorre-se do contexto (*classe Aula*) em direcção alvo da outra associação (*classe OcorrenciaSemanal*), retorna a duração da ocorrência. Por fim, efectua-se a comparação entre os valores obtidos em ambos os lados da equação (OCL não especifica uma ordem para avaliação; apenas para fins de explicação do exemplo, avaliou-se o lado esquerdo primeiro).

Algumas expressões, quando avaliadas, podem resultar num valor indefinido. A divisão de um número por zero e o resultado da operação *first* aplicada a uma colecção sem elementos são alguns exemplos de expressões que resultam num valor indefinido. Além disso, como um atributo de uma classe pode ser definido com multiplicidade [0..1], indicando que nem toda as instâncias dessa classe precisam ter um valor definido para esse atributo, expressões que envolvam um atributo com essa multiplicidade também podem resultar num valor indefinido.

Para lidar com esse facto, as operações lógicas envolvendo expressões OCL são definidas considerando três valores: verdadeiro, falso e indefinido ($\perp$).

Em geral, o resultado de expressões envolvendo um valor indefinido é também indefinido, com excepção de algumas expressões envolvendo operações booleanas, ou de algumas operações de colecção como, por exemplo, select, *forAll*, *exists*. A operação booleana *oclIsUndefined*() permite verificar se o valor de uma expressão é indefinido.

2.2.2. Ferramentas utilizadas

O processo de engenharia directa envolve a geração de um ou mais artefactos de

software que sejam mais parecidos com o programa final a ser implantado (por exemplo, código-fonte), na forma e em nível de detalhe, do que os artefactos utilizados neste processo (por exemplo, diagramas de classe de um modelo UML) (Sendall & Küster, 2004). Mais especificamente, a desenvolvimento de código é a tarefa de criação do código-fonte, numa linguagem de programação, a partir de modelos existentes. Nos últimos anos e de forma crescente, algumas organizações têm usado modelos UML para estimular o desenvolvimento de código e outros artefactos de *software*. Geralmente, isso é feito a partir de diagramas de classe, que são usados para gerar interfaces entre os vários componentes de uma aplicação distribuída (Henderson-Sellers, 2005). A geração automática de código a partir de um modelo simplifica a implementação, agilizando o desenvolvimento de *software*.

Foram estudadas diferentes ferramentas que utilizam a linguagem OCL, presentes em diagramas de classe UML, para realizar a transformação em código a partir de diagramas.

2.2.2.1. USE

USE (UML-based Specification Environment) é um sistema para a especificação de sistemas de informação. A especificação da utilização contém uma descrição textual de um modelo usando os recursos encontrados em diagramas de classe UML (classes, associações, etc.). Expressões escritas na OCL são usadas para especificar restrições de integridade adicionais sobre o modelo. Expressões OCL podem ser inseridos e avaliados para obter informações detalhadas sobre a consulta de um estado do sistema.

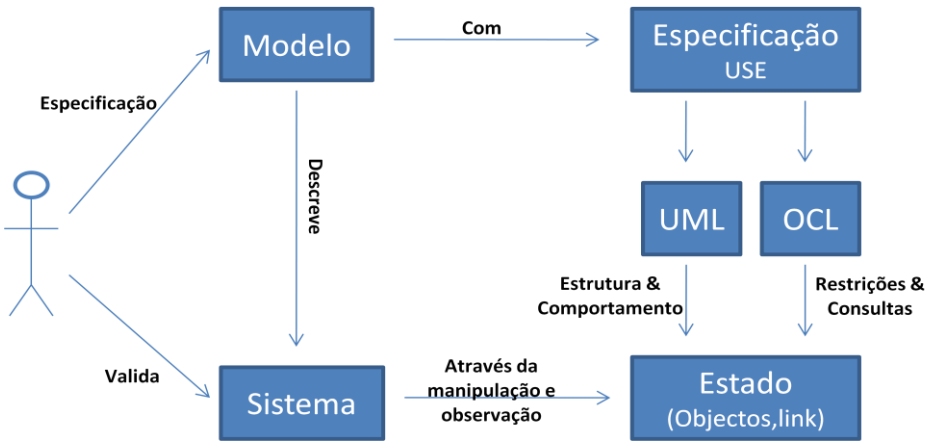


Figura 5 - Visão geral da abordagem USO (USE, 2009)

USE foi desenvolvida na Universidade de Bremen, Alemanha (Bohling, Gogolla, & Richters, 2003), (Bauerdick, Gogolla, & Gutsche, 2004), (Bohling, Richters, & Gogolla, 2005)) e é utilizada para verificar inconsistências em diagramas de classe UML.

A USE não gera código a partir do diagrama e das suas restrições, mas é capaz de interpretar expressões OCL, utilizando-a como uma linguagem de consulta sobre um dado diagrama de classe UML, exibindo o resultado das expressões fornecidas. Essencialmente, é utilizada para a verificação de modelos. Isso é feito por meio da animação, realização de testes e a validação de diagramas de classe UML e as suas respectivas restrições OCL, focando-se nos estágios iniciais do processo de desenvolvimento. Desta forma, podem ser identificados, antecipadamente, defeitos presentes em diagramas de classe (por exemplo, se uma classe não pode possuir instâncias). Entretanto, USE também pode ser usada para verificar se uma dada implementação em Java não viola as restrições presentes no diagrama de classe do modelo respectivo.

USE possibilita a validação de diagramas de classe UML anotados com OCL. No seu trabalho, Bauerdick e outros (Bauerdick, Gogolla, & Gutsche, 2004) utilizaram essa ferramenta para identificar, na super-estrutura da especificação da UML 2.0 (OMG), erros de sintaxe e de verificação de tipos em algumas das suas regras de boa formação e definições de operações. Por exemplo, seja a seguinte definição da operação *Kernel::Package::visibleMembers()*.:

visibleMembers():Set(Kernel_PackageableElement)=
member->select(m | self.makesVisible(m))

Esta operação deve retornar quais os membros de um pacote que podem ser acedidos fora dele. Há um problema com a definição desta operação. Seja o diagrama de classe presente na Figura 6 (extraída do trabalho dos autores). O identificador *member* é o nome do papel que a super-classe *NamedElement* assume em sua associação com uma das suas *sub-classes*, *Namespace*. A operação *select* retornará um conjunto de elementos do tipo *NamedElement* que atendam à restrição *self.makesVisible(m).*

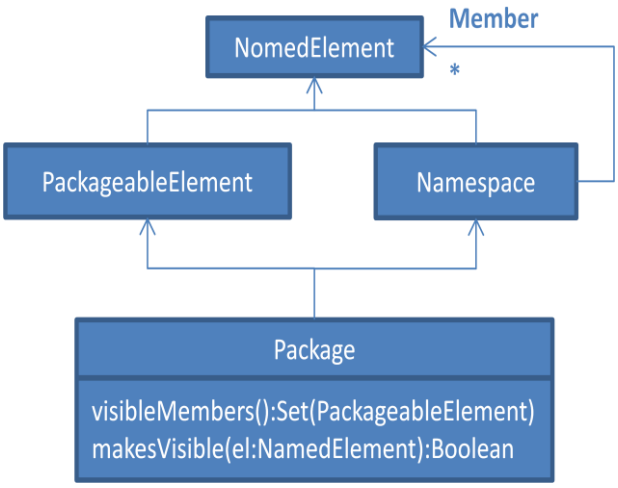


Figura 6 - Diagrama de classe para contexto da operação visibleMembers()

Entretanto, segundo o diagrama da Figura 6, a operação *visibleMembers()* deveria produzir como resultado um conjunto de elementos do tipo *PackageableElement*, sub-tipo de *NamedElement*, evidenciando a inconsistência entre a regra de boa-formação do modelo e um dos seus diagramas de classe.

Noutra utilização da ferramenta, Richters e Gogolla (Gogolla & Richters, 2003) fizeram a validação de uma implementação de *software*, em relação às restrições especificadas nos diagramas de UML respectivo. Utilizando um modelo e as suas restrições, a ferramenta gera um aspecto (Laddad, 2003). Quando compilado com a implementação Java do modelo respectivo e executado, esse aspecto é responsável por enviar à USE as mudanças de estado ocorridas durante a execução do programa, para que a ferramenta faça a verificação das restrições. No caso de se verificar a violação de restrição, a USE notifica o utilizador, indicando a restrição violada. Invariantes são verificados antes e depois da execução de cada método público Java, bem como depois da invocação dos construtores. *Pré-condições* são verificadas antes da chamada dos métodos respectivos. *Pós-condições*, após a chamada.

Um inconveniente do uso da ferramenta é a necessidade que as informações de entrada – diagrama de classe e as suas restrições – estejam num formato textual próprio da USE, não suportando a leitura de um modelo já construído noutro formato, como XMI[4] (ou XML Metadata Interchange), por exemplo.

[4] XMI (ou XML Metadata Interchange) é um padrão da OMG (grupo de gestão de objectos) para troca de informações baseado em XML. O uso mais comum é na troca facilitada de meta dados entre as ferramentas de modelação (baseadas no UML da OMG) e os repositórios (OMG-MOF).

É implementado em Java (tm). Portanto, deve ser executado em qualquer plataforma na qual utilize um sistema de execução Java (por exemplo, o *Sun JDK*). Foi testado principalmente em plataformas *Unix-like* (*Solaris e Linux*), mas também funciona no *Windows*. Pode ser utilizado de duas formas, através de um interface gráfico ou através de uma linha de comandos.

2.2.2.2. DOT(Dresden OCL Toolkit)

Na Universidade de Tecnologia de Dresden, Alemanha, foi desenvolvida a ferramenta DOT (*Dresden OCL Toolkit*) (Hussmann, Demuth, & Finger, 2002), (Loecher & Ocke, 2003). É melhorada e mantida principalmente por estudantes e pessoal científico de Tecnologia de Software *Group at Technische Universität Dresden*, onde o projecto é também coordenado.

A primeira versão, o *Dresden OCL Toolkit (1999)* neste momento encontrasse obsoleta. Em 2005, foi lançada a segunda versão, chamada de *Dresden OCL2 Toolkit que* é baseada num dicionário de dados do Netbeans. A terceira e última versão é a *Dresden OCL2 Toolkit* para Eclipse, e é baseado num modelo de dinâmica (*Pivot Model*) e, portanto, pode ser utilizado por dicionário de dados. (Dresden OCL Toolkit).

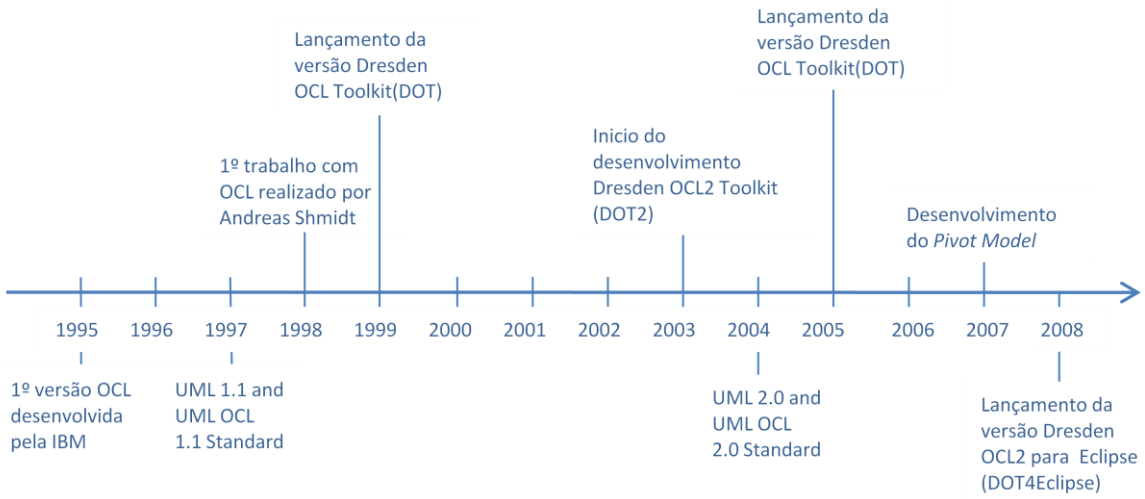


Figura 7 – Percurso de desenvolvimento do OCL pela Dresden OCL Toolkit (Dresden OCL Toolkit)

Este compilador é capaz de realizar a análise sintáctica (*parsing*) de expressões OCL, verificação de tipos e avaliação de restrições presentes em modelos comuns. Também é capaz de gerar, automaticamente, o código Java e SQL[5] a partir das expressões OCL e do modelo UML fornecidos.

[5] Existe uma versão desactualizada.

A DOT gera o código e avalia/controla o código-fonte da aplicação, para que sejam verificadas as restrições em tempo de execução. Ocorre em quatro fases: 1) a partir de um arquivo XMI ou de comentários no código-fonte, é realiza a análise sintáctica, resultando numa árvore sintáctica abstracta (ou *Abstract Syntax Tree* - AST); 2) a AST das expressões, em OCL, é transformada, convertendo-se algumas das operações OCL para um sub-conjunto de operações dessa linguagem, a fim de simplificar a posterior geração de código (por exemplo, *select* é transformado em *iterate*); 3) navegando pela AST, o criador de código produz as expressões correspondentes em Java; 4) essas expressões são então inseridas na aplicação, para que sejam testadas em tempo de execução.

A inserção do código para verificação das expressões na aplicação é feita conforme cada tipo de restrição – *invariantes*, *pré-condições* ou *pós-condições*. Para os *invariantes*, em vez de serem verificados antes e depois da execução de cada método público e após cada construtor, a sua execução ocorre sempre no fim dos métodos que modifiquem o valor de campos utilizados pelos *invariantes*.

Para cada campo presente no código, são adicionados *observadores* (mecanismo de desenho *Observer* (Gamma, Helm, Johnson, & Vlissides, 1994) que são notificados se o valor do campo respectivo mudar. Isso determina quais *invariantes* devem ser verificados devido às mudanças ocorridas, disparando por sua vez a execução dos *invariantes* relacionados.

A DOT tem sido utilizada na arquitectura de diferentes ferramentas (por exemplo, protótipo da abordagem de *Verheecke*, *KeY Tool*), nas quais é utilizado como um "intermediário" entre as funcionalidades de modelação e as de inserção de código dessas ferramentas.

2.2.2.3. SPACES

SPACES (*SPecification bAsed Component tESter*) é uma ferramenta desenvolvida por Barbosa e outros (Barbosa, Andrade, Machado, & Figueiredo, 2004), na Universidade Federal de Campina Grande. Ela faz uso de especificações em UML (diagramas de classe, de casos de uso e de sequência), comentada com OCL, para gerar casos de teste automaticamente (Figura 8).

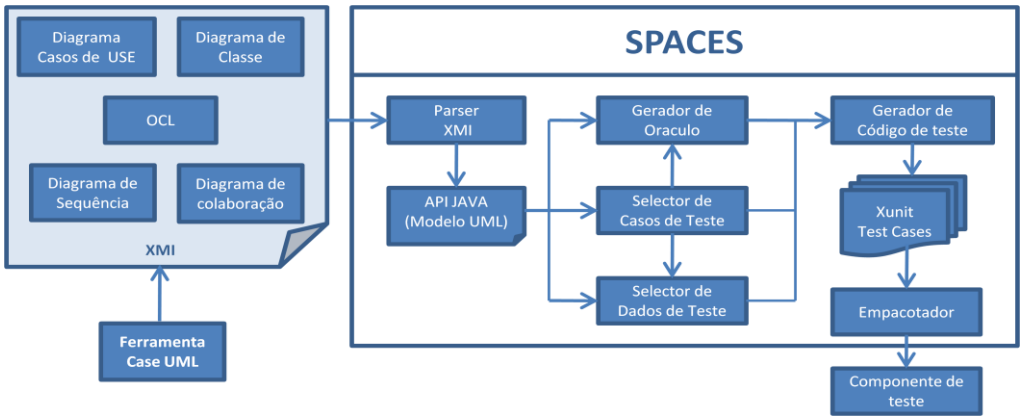


Figura 8 - Arquitectura da Ferramenta SPACES (Andrade, Barbosa, Machado, & Figueiredo, 2004)

O módulo "*Parser XMI*" possibilita o acesso da ferramenta à especificação do componente sob a forma de artefactos UML e restrições OCL através da criação de uma instância de uma API que disponibiliza essas informações para os restantes módulos. Nos módulos "*Selector de Casos de Teste*", "*Gerador de Oráculos*" e "*Selector de Dados de Teste*" são produzidos os artefactos de teste: casos de teste, oráculos e dados de teste, respectivamente. Estes artefactos são utilizados pelo módulo "*Gerador de Código de Teste*" para realizar a construção do código responsável por testar a implementação do componente. Uma vez geradas as classes de teste, o módulo "*Empacotador*" tem a responsabilidade de construir um novo componente a partir do qual os casos de teste poderão ser executados e os resultados analisados. Este componente de teste é distribuído com a implementação do componente desenvolvido de forma a possibilitar a re-execução do teste do mesmo por parte dos clientes. (Andrade, Barbosa, Machado, & Figueiredo, 2004)

A arquitectura permite ainda a expansão da ferramenta para dar suporte à utilização de outras linguagens de especificação de restrições além de OCL, por exemplo *Object Z*, e à geração de código de teste em diferentes linguagens de programação além de Java, por exemplo *C++*. (Andrade, Barbosa, Machado, & Figueiredo, 2004)

2.2.2.4. OCL2J

Foi desenvolvida por Briand e outros (Briand, Dzidek, & Labiche, 2004), (Dzidek, Briand, & Labiche), na Universidade de Carleton, Canadá. Visa converter restrições OCL, presentes em diagramas de classe de um modelo UML, adicionando-os a programas Java. Para isso, ela utiliza *AspectJ*, uma implementação Java de POO (Programação Orientada por Objectos) (Laddad, 2003). Desta forma, ela faz a verificação automatizada de restrições OCL durante a execução de uma aplicação

Java a ser testada.

Em linhas gerais, funciona da seguinte maneira: as informações necessárias à ferramenta são retiradas do diagrama de classe UML e do código-fonte da aplicação Java a ser testada/controlada; todas as expressões OCL encontradas são analisadas, gerando ASTs que serão usadas para criar o código correspondente em Java; a aplicação-alvo é então testada/controlada com o código gerado das restrições, utilizando aspectos feitos em *AspectJ*.

No seu trabalho, Verheecke (Verheecke, 2001) aponta algumas desvantagens no uso de POO para controlar as restrições OCL, como o que foi realizado na ocl2j:

- com o acréscimo de novos comportamentos e funcionalidades a uma aplicação, utilizando-se aspectos, o programador perde o controlo sobre o código, porque essas mudanças ocorrem à sua revelia ("caixa-preta");
- a gestão de configurações do projecto deve gerir duas árvores de código-fonte;
- o programador deve estar atento para editar apenas a versão original do código-fonte, não controladas;
- é necessário que se execute a ferramenta de controlo após cada mudança do código-fonte, e não apenas quando as restrições mudarem;
- conflitos entre o código-fonte original e o código-fonte testado são difíceis de se localizar e resolver;
- traços dinâmicos de excepções (*stack traces of runtime exceptions*) apontam para o código-fonte modificado, ficando a cargo do programador localizar, no código original, o trecho de código correspondente;
- a maioria dos programadores Java provavelmente têm uma profunda aversão a usar um ambiente Java dedicado apenas para verificar restrições OCL, exigindo-se, ainda, o conhecimento de conceitos de POO.

Acrescenta-se às desvantagens acima a perda do raciocínio por módulos. Segundo Wand (Wand, 2003), as linguagens de POO violam o raciocínio modular, o seu uso impede a dedução do comportamento de um módulo composto a partir das suas partes constituintes. Os aspectos podem mudar radicalmente o funcionamento do módulo, tornando-o dependente do contexto onde é aplicado. Para resolver estes problemas, Verheecke (Straeten & Verheecke, 2002) propõe armazenar as restrições OCL, já convertidas para Java, em classes separadas, onde cada classe representa uma restrição.

2.2.2.5. KeY Tool

KeY Tool (Ahrendt, et al., 2005), (Giese & Heldal, 2004), (Hahnle, Giese, & Daniel,

2004), (Larsson & Giese, 2005), (Hähnle, Johannisson, & Ranta, 2002) é uma ferramenta de especificação formal e verificação de programas no desenvolvimento de *software* UML. Foi desenvolvida na Universidade de Tecnologia de Chalmers, Suécia, como um *plug-in* para uma ferramenta CASE (*Computer Aided Software Engineering*) comercial[6].

Um dos objectivos do projecto *KeY* é a divulgação de métodos formais na indústria de desenvolvimento de *software*, por meio da utilização de restrições OCL, focando a sua aplicação em diagramas de classe de um modelo UML. A *KeY Tool* suporta um sub-conjunto da linguagem Java, chamado *Java Card*. Ela exclui certas características (por exemplo, cópias e carregamento dinâmico de classes), com uma API reduzida, sendo voltado para execução em *smart cards* e outros dispositivos com memória limitada, tais como sistemas embutidos.

É feita uma extensão (Figura 9) do UML/baseado (*UML/Java-based*) em "*CASE tool TogetherCC*" adicionando recursos para apoiar a anotação de diagramas UML com restrições OCL. Um "*Componente de Verificação*" é responsável pela geração de fórmulas em lógica dinâmica que expressam a relação entre especificação e implementação. Finalmente, o componente de dedução pode ser usado para validar as referidas fórmulas. (Beckert, et al.)

Figura 9 - Estrutura geral do KeY Tool (Beckert, et al.)

Os autores advogam a utilidade da ferramenta em três cenários de aplicação:

- modelação precisa. Se por um lado, as alterações na implementação ocorrem frequentemente, por outro lado, os modelos (e os seus diagramas de classe

[6] *Borland Together Control Center*. Para maiores informações, http://www.borland.com/together/index.html

comentados com restrições OCL) são menos sujeitos a mudanças. O uso de restrições OCL em diagramas de classe de um modelo de desenho, removendo as suas ambiguidades, pode resultar em ganhos significativos na identificação e remoção de defeitos durante o processo de desenvolvimento. Nesse sentido, a KeY Tool fornece *templates* (KeY Idioms e KeY Patterns) para auxiliar os utilizadores menos experientes com a linguagem. Além disso, é disponibilizado o recurso da tradução de restrições em OCL para descrições em linguagem natural, em inglês (Hähnle, Johannisson, & Ranta, 2002).

- desenvolvimento de *softwares* críticos. A extensão dos prejuízos (humanos e/ou materiais) que podem ser causados no caso de implementações defeituosas justifica o investimento de esforço adicional em verificação formal.
- ferramenta de ensino. A ferramenta pode ser usada como um recurso auxiliar no ensino de provas de teoremas, utilizando lógica de predicados. Além disso, pode auxiliar na aprendizagem da própria OCL.

A *KeY Tool* faz a verificação de restrições presentes num diagrama de classe de um dado modelo, partindo da premissa que as relações entre classes implicam relações entre as restrições correspondentes.

Desta forma, as restrições podem ser analisadas independentemente de como forem implementadas. Além disso, a ferramenta faz a verificação de consistência entre implementações Java e restrições OCL comentadas no diagrama de classe, utilizando um processador de teoremas. Para tanto, o diagrama de classes, as restrições comentadas, bem como a implementação Java são utilizadas na construção automatizada de provas formais da sua consistência. Para a conversão das expressões OCL para Java, utilizam-se as funcionalidades da ferramenta DOT.

2.2.2.6. Ocl4Java

Foi desenvolvida pela Universidade de Kent, Reino Unido (Akehurst & Patrascoiu, 2004), (Linington & Akehurst, 2003). Parte do projecto KMF (*Kent Modeling Framework*), ela faz a geração de código Java a partir de expressões OCL presentes em diagramas de classes. Suporta a UML 1.4 e OCL 2.0.

O processamento feito pela ferramenta é dividido em duas etapas: análise (léxico, sintáctica e semântica) e síntese (geração de código ou interpretação). No fim da análise sintáctica, é gerada uma AST das expressões OCL. O analisador semântico consome essa AST, juntamente com o diagrama de classes de um dado modelo UML (referido pelas expressões OCL), produzindo um modelo semântico, que pode ser usado para a geração de código Java correspondente às expressões fornecidas. Os autores identificaram que a gramática fornecida na especificação da OCL (OCL03) é

ambígua.

Criando regras de remoção de ambiguidades (*disambiguating rules*), desenvolveram uma gramática LALR para a sua implementação (Akehurst & Patrascoiu, 2004),. Para a análise léxica, essa gramática foi particionada em dois níveis: o primeiro contém os termos básicos da linguagem; o segundo contém as sub-gramáticas associadas a cada um dos termos definidos no primeiro nível. Isso foi feito porque, segundo os autores, permite que a gramática da linguagem seja descrita utilizando apenas símbolos básicos. Para a construção do analisador léxico, foi utilizado o *JFlex*, um gerador de analisadores léxicos. Como representação intermediária da conversão das expressões OCL, foi escolhida a AST, dada "a *sua habilidade em capturar toda a informação necessária à execução de quaisquer operações relacionadas ao fluxo de controlo e de dados do código-fonte*".

2.2.2.7. Atenas

A ideia básica da ferramenta é servir como uma base formal para a documentação das regras de negócio de um sistema de informação. No entanto, a ferramenta está a ser desenvolvida de forma a poder ser introduzida gradualmente num sistema em desenvolvimento ou mesmo já em produção. Para tal, a especificação de uma regra de negócio poderá ser feita utilizando-se uma linguagem formal (OCL, Pascal, PL/SQL), ou de maneira informal, através de um texto.

As regras de negócio estão divididas em três grupos: regras extraídas automaticamente do diagrama da base de dados, regras registadas manualmente pelo utilizador e regras inferidas automaticamente a partir das outras duas. A ferramenta possui funcionalidades para lidar com estes três tipos de regras.

Validação do Sistema – uma vez que todas as regras de negócio estejam estabelecidas numa linguagem formal, é possível gerar a lista de eventos de sistema (inserções, alterações, etc.) onde as mesmas devem ser avaliadas. É possível examinar o código para verificar se as regras de negócio estão realmente sendo avaliadas, e se o estão da forma correcta.

Validação de dados antigos – uma tarefa árdua quando um sistema é implementado em substituição de outro é a importação dos dados do sistema antigo. Mais do que colocar a base sob um novo formato, é imprescindível também assegurar a integridade dos dados. Porém, quando uma base de dados está para ser migrada nem sempre é viável inserir os dados simulando o uso normal do novo sistema. Em geral, será feita uma carga massiva de dados, e somente então as regras de negócio serão avaliadas. Se as regras de negócio não estiverem formalizadas, esta tarefa irá

exigir a codificação de procedimentos especiais para verificar a validade das regras de negócio, e que serão utilizados apenas para a importação dos dados e descartados após isto. Com a abordagem proposta pela ferramenta, será possível gerar estes procedimentos automaticamente para todas as regras que estejam formalmente estabelecidas. Certamente esta abordagem não irá revelar todas as regras de negócio, pois nem todas as regras geram mensagens de erro. No entanto, boa parte das regras de restrição podem ser obtidas desta forma – algumas até automaticamente, como é o caso das restrições de integridade codificadas da base de dados. Desta forma, o diagrama da base de dados é uma fonte importante que pode ser utilizada para a extracção automática de regras de negócio já formalizadas.

Manutenção do Sistema – a ferramenta permite analisar o impacto das mudanças em regras de negócio, através do relatório de impacto de uma mudança em alguma regra de negócio. Cada alteração proposta de uma regra (introdução, remoção ou alteração) irá gerar uma lista de eventos, mapeados para trechos de código, que devem ser inspeccionados de forma a garantir que as alterações necessárias sejam realizadas. Isto pode ser visto no fluxograma da Figura 10. Além disso, para cada regra será mantida uma informação de grande utilidade para a manutenção do sistema: os trechos do código fonte onde a regra está sendo avaliada.

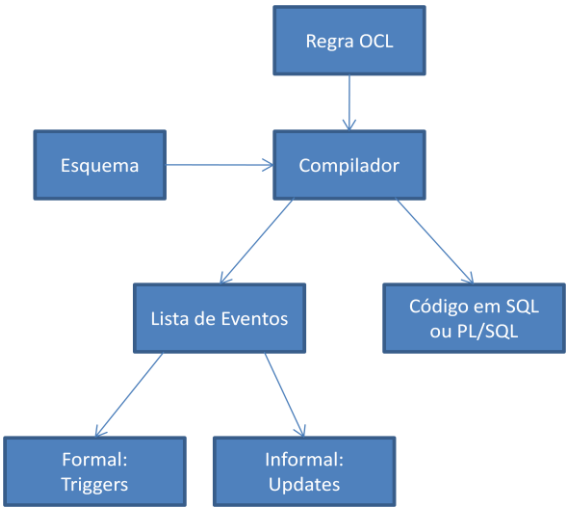


Figura 10 - Esquema de avaliação do impacto de alterações na base de dados (Silva, Almeida, Souza, Sulaiman, & Neto, 2001).

Um compilador de OCL irá permitir que as regras de negócio formuladas em OCL sejam automaticamente convertidas para um código SQL equivalente que teste a validade da regra ou produza o seu efeito.

A ferramenta foi implementada para ser usada em ambiente Windows, implementada em Delphi com suporte a uma base de dados ORACLE.

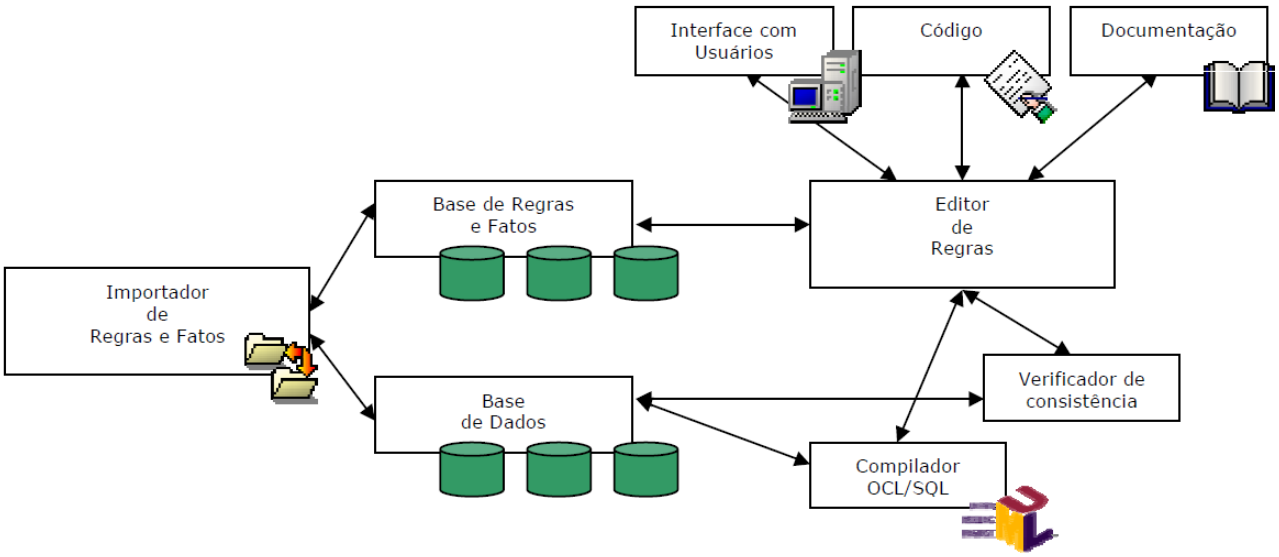


Figura 11 - Arquitectura do Sistema Atenas (Silva, Almeida, Souza, Sulaiman, & Neto, 2001).

2.2.2.8. Comparação de ferramentas

As abordagens estudadas encontram-se sumarizadas na Tabela 2. Para os casos em que uma dada informação sobre a ferramenta em questão não esteve clara, foi utilizado o valor "n/i" (não informado). A descrição de cada coluna segue abaixo:

Suporte: Tipo de suporte oferecido pela ferramenta, adaptado da caracterização feita por Hussmann e outros (Hussmann, Demuth, & Finger, 2002): Análise sintáctica, Verificação de tipos, Consistência lógica (identificação de contradições num dado conjunto de restrições), Validação dinâmica (avaliação de invariantes, pré-condições e pós-condições em tempo de execução), Automação de testes (verificação automática de resultados de testes, usando pré-condições e pós-condições OCL para derivar casos de teste), Verificação de código (verificação de conformidade do código-fonte às restrições presentes no diagrama de classe respectivo).

Automação: Mostra se a operação da ferramenta não requer nem possibilita a interacção do utilizador. Nesse caso, ela é automática; caso contrário, semi-automática.

Linguagem – Em que linguagem é possível converter o OCL.

Open Source – Se o código da aplicação é código aberto, isto é, se qualquer pessoa com conhecimento pode modificá-lo.

Ferramenta	Grau de Automação	Suporte	Linguagem	Open Source
USE	Semi-automática	• É utilizada para a verificação de modelos; • Validar se as restrições estão bem implementadas.	-------	Sim
DOT (Dresden OCL Toolkit)	Automática	• Realiza a análise sintáctica; • Gera código Java a partir das expressões OCL.	Java/SQL[7]	Sim
SPACES	Semi-automática	• Análise sintáctica; • Automação de testes.	Java, C++	n/i
OCL2J	Automática	• Análise sintáctica; • Verificação de Tipos; • Validação dinâmica.	Java	n/i
KeY Tool	Semi-automática	• Análise sintáctica; • Verificação de Tipos; • Consistência lógica; • Validação dinâmica; • Verificação de código.	Java	n/i
Ocl4Java	Automática	• Análise sintáctica; • Verificação de Tipos; • Verificação de código.	Java	Sim
Atenas	Automática	• Validação do Sistema; • Automação de testes; • Verificação de código.	SQL	n/i

Tabela 2 - Sumário das ferramentas

[7] Apenas existe numa versão muito antiga

3. Descrição do caso de estudo

3.1. Elaboração de horários

Numa instituição de ensino superior, a elaboração de horários deve levar em conta diversos factores. Entre eles, a disponibilidade de salas, horários de trabalho dos docentes, e as unidades curriculares oferecidas no período são as mais importantes.

Para cada curso oferecido pela instituição, um conjunto de unidades curriculares (UC) é leccionado em cada período lectivo, geralmente um semestre. Estas unidades curriculares podem ser divididas em grupos (por departamento, posteriormente, por áreas cientificas), com conteúdos inter-relacionados chamados de pré-requisitos. Este agrupamento permite especificar que unidades curriculares podem ser ministradas pelos docentes, de acordo com a sua habilitação, bem como evitar conflitos de horários.

Neste trabalho, considera-se que cada curso é formado por um conjunto de unidades curriculares, com a duração de um semestre ou um trimestre. Para concluir o curso, o estudante deverá matricular-se e conseguir aprovação em todas as unidades curriculares do curso.

Num determinado semestre/trimestre, as unidades curriculares são leccionadas em determinados dias da semana, sendo que o estudante poderá matricular-se em qualquer unidade curricular do seu curso. Porém, a matrícula nas unidades curriculares está sujeita a um limite de créditos (ECTS) por semestre/trimestre, sendo obrigatório a inscrição em primeiro lugar das unidades curriculares em atraso.

Para orientação do estudante, os cursos oferecem um plano curricular, com uma distribuição das unidades curriculares por semestre/trimestre (Figura 12), com a indicação do número de ECTS por unidade curricular.

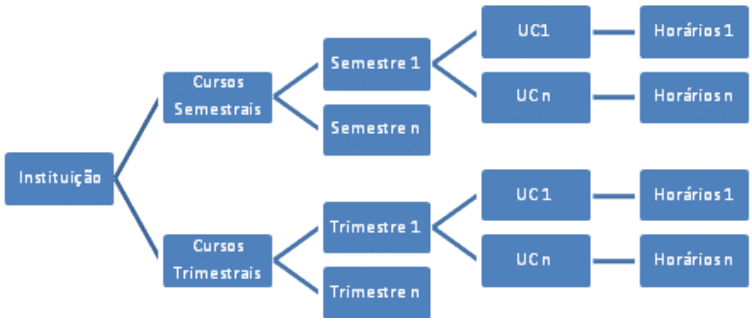


Figura 12 – Distribuição das UCs

Como uma mesma unidade curricular pode ser ministrada em diversos cursos, estas estão agrupadas por departamento, levando em conta as áreas de actuação.

Dentro de cada departamento, as unidades curriculares são agrupadas de acordo

com o seu conteúdo, por áreas científicas.

Cada unidade curricular necessita de um conjunto de recursos para ser leccionada. Uma grande parte necessita apenas de um sala de aula com capacidade que suporte as vagas oferecidas. Outras unidades curriculares requerem recursos adicionais, tais como laboratórios de informática, salas de desenho, laboratórios, etc.

As unidades curriculares dos cursos são leccionadas pelos docentes da instituição ou por docentes convidados, caso os recursos internos sejam insuficientes. Os docentes geralmente têm uma formação específica dentro de uma área, capacitando-se assim a leccionar um conjunto de unidades curriculares. Como as unidades curriculares estão agrupadas por departamentos, os docentes ficam ligados aos departamentos que as agrupam.

Numa Instituição existem diversos cursos, os quais podem ter aulas em diferentes períodos do dia (manhã, tarde ou pós laboral).

Cada curso apresenta um determinado número de semestres/trimestre proposto pelo plano curricular. Para cada unidade curricular dentro do semestre/trimestre há um horário associado, fazendo com que cada estudante tenha, durante a semana, uma ou mais unidades curriculares no seu horário.

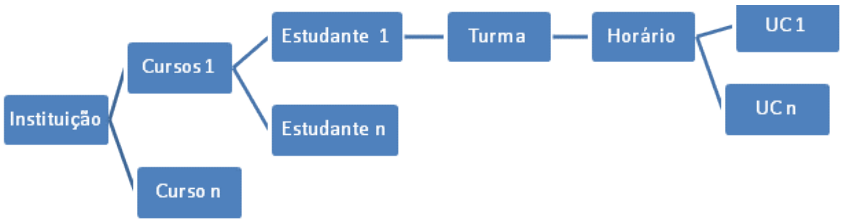


Figura 13 - Distribuição dos estudantes

As turmas são determinadas pelo agrupamento das unidades curriculares de um determinado semestre de um curso. Uma unidade curricular pode ter mais do que uma turma e podem existir unidades curriculares de cursos distintos a partilhar as mesmas aulas. Um estudante não é obrigado a inscrever-se à mesma turma a todas as unidades curriculares, mas caso não o faça não é garantida a não sobreposição de aulas.

As unidades curriculares e os docentes da instituição estão agrupados por departamento e posteriormente por áreas científicas, em função da sua área de conhecimento. Dentro dos departamentos as unidades curriculares estão agrupadas com base no seu conteúdo por áreas científicas. Desta forma, é possível localizar os docentes que podem leccionar as unidades curriculares, de acordo com as características da sua formação.

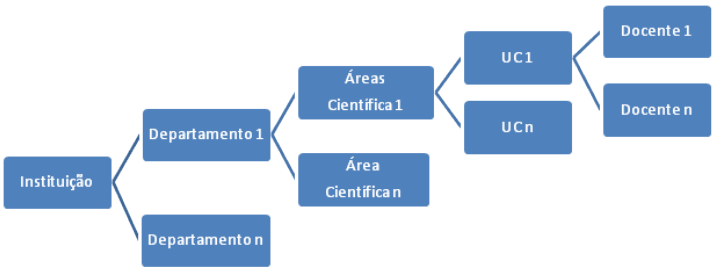


Figura 14 - Distribuição dos docentes e das unidades curriculares

Uma unidade curricular pode ser oferecida num ou mais cursos, em diferentes períodos do dia dependendo do curso e do semestre/trimestre.

Os períodos de aulas dividem-se em três períodos distintos, manhã, tarde e pós laboral, a fim de atender aos horários das turmas.

No processo de ensino de uma unidade curricular, são utilizados diferentes bens físicos, denominados de recursos. Estes recursos só podem ser alocados para uma turma num determinado horário, sendo que uma turma pode ter de um a mais recursos diferentes alocados durante um semestre/trimestre.

3.1.1. Processo de elaboração de horários

A elaboração de horários consiste num procedimento sistemático e automático que procura compatibilizar os diversos horários disponíveis na semana, tendo em conta todas as unidades curriculares que são oferecidas num semestre/trimestre de um curso, e satisfazendo as disponibilidades de recursos.

É importante ressaltar que a dimensão do problema e as características específicas da instituição constituem um aspecto extremamente importante no que diz respeito à elaboração de horários.

Quando se elabora um horário devesse procura alocar as unidades curriculares dos cursos de forma que uma unidade curricular que utilize um recurso específico (ex. sala comum, laboratório) não seja alocada com outra que necessite do mesmo recurso, maximizando a sua utilização. Na alocação é levada em consideração a área científica da unidade curricular, de modo que unidades curriculares de uma mesma área científica fiquem em horários diferentes. Com isto pretende-se que um mesmo docente possa ser alocado para o máximo de unidades curriculares dentro da sua área de habilitação.

Um horário é uma relação entre diversos elementos, tais como tempo, unidades curriculares, docentes e estudantes. Os elementos são também chamados de recursos. As características destes elementos são especificados pelo problema, e a melhor

relação entre eles deve ser definida como parte da solução. Por exemplo, é possível especificar as unidades curriculares e o horário como parte do problema. A solução deverá encontrar um docente capacitado a ministrá-la, com base nas informações sobre os docentes disponíveis e seus horários.

O problema da construção de horários tem sido à muito tempo conhecido como um problema pertence aos chamados NP-completos, e nenhum método de resolução conhecido o resolveu num tempo razoável.

Segundo Burke e outros (Elliman, Burke, & Weare, 1995), existem diversas variações no problema de horários. Os horários universitários podem ser divididos em duas categorias principais: aulas e exames. Aulas são os conteúdos ministrados pelos docentes, e exames são as provas às quais os estudantes devem submeter-se para aprovação na unidade curricular. A maior diferença entre horários de aulas e horários de exame são:

- exames devem ser marcados de forma que nenhum estudante tenha mais de um exame de cada vez; mas os horários das aulas devem ser construídas antes do estudante se inscrever no mesmo.
- como o espaço físico é frequentemente uma restrição, exames podem compartilhar salas, mas apenas uma aula pode ser alocada numa sala no mesmo instante.

O processo de horário é muito difícil de ser feito, pelo facto de que muitas pessoas são afectadas pelos seus resultados. Romero identifica três principais pontos neste processo, cada um com as suas próprias características e necessidades (Romero, 1982).

A administração fixa os padrões mínimos para os quais o horário deve ajustar-se. Por exemplo, não permitir duas aulas teóricas seguidas.

As preocupações dos departamentos são maiores nos horários dos cursos. Querem o horário em harmonia com o conteúdo programático leccionado, e sala ou laboratórios mais específicos.

O terceiro ponto é os estudantes, onde cada um possui o seu próprio horário, e que afecta somente a ele. Dado o número de estudantes envolvido é difícil de obter um critério específico sobre o qual é o melhor horário para os estudantes. Muitos estudantes preferem não ter aulas à tarde numa sexta-feira, e ter uma pausa entre exames consecutivos. Se forem levadas em conta as preferências dos estudantes, há um aumento considerável da dificuldade do problema.

Segundo Burke, as restrições de horários são muitas e variadas (Elliman, Burke, & Weare, 1995), sendo que alguns dos tipos mais comuns são relacionadas a seguir.

Alocação de Recursos. Um recurso deve ser associado para um outro recurso de tipo diferente ou para uma turma. Por exemplo um docente pode preferir leccionar algumas aulas de uma unidade curricular numa sala normal, e outras num laboratório.

Tempo alocado. Uma aula ou um recurso pode ser associado a um docente, implicando que neste período o docente está indisponível para outras aulas.

Distribuição de aulas. As aulas devem ser distribuídas uniformemente durante o período. Por exemplo, aulas teóricas devem ser intercaladas com aulas práticas.

Coerência das aulas. Por exemplo, uma unidade curricular que poderia ter as suas aulas todas num único dia, deve ser alocada em dois dias, por motivos pedagógicos.

Capacidades das salas. O número de estudantes numa sala não pode exceder a sua capacidade.

Ainda em relação às restrições, Burke divide-as em duas categorias: Rígidas (*hard*) e Flexíveis (soft) (Elliman, Burke, & Weare, 1995):

- Restrições Rígidas. Um horário que quebra uma restrição rígida não pode ser considerado parte da solução, e deve ser reparado ou rejeitado pelo algoritmo do horário. Por exemplo, nenhuma docente pode dar duas aulas ao mesmo tempo em salas distintas.
- Restrições Flexíveis. Restrições flexíveis nem sempre são menos importantes que restrições rígidas, e dificilmente levam um horário a ser rejeitado. São aplicadas a qualquer método de horário, geralmente avaliadas por uma função que penaliza o horário, calculando até que ponto este quebrou a sua restrição. Algumas restrições flexíveis são mais importantes que outras, e têm uma maior prioridade.

3.1.1.1. Horários

A elaboração de um horário consiste em definir o dia, hora e sala em que cada aula de uma turma irá ser leccionada. Cada ano lectivo de um curso possui diversas unidades curriculares e várias turmas.

Uma unidade curricular tem uma duração total, que pode ser dividida em aulas Teórica (T), aulas Práticas (P), aulas Teórica/Prática (TP) e aulas de Laboratório (L). Cada tipo de aula tem uma divisão específica, por exemplo, Introdução à Programação tem um total de cinco horas semanais, três horas teórica/práticas e duas de laboratório, mas as horas teórica/práticas estão divididas em duas aulas (que ocorrem em dias diferentes) e as horas de laboratório é apenas uma aula.

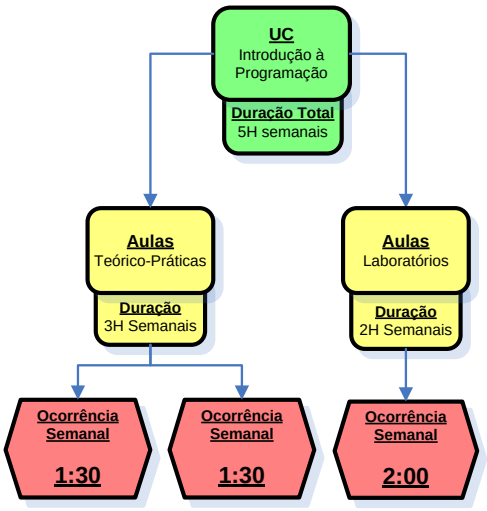


Figura 15 - Divisão das horas semanais de uma unidade curricular

Cada docente tem atribuído um conjunto de aulas de um dado tipo de aula (T; TP; P;L) que são leccionadas a turmas ou conjunto de turmas de um curso.

Cada docente pode definir um conjunto de restrições obrigatórias (restrições fortes) e um conjunto de preferências (restrições fracas).

Cada sala tem uma determinada configurações (Normal, Laboratório, Informática, etc.) e uma determinada capacidade. Cada tipo de aula de uma unidade curricular necessita de um determinado tipo de sala específico. Cada aula tem um número de estudantes, que não pode ultrapassar a capacidade da sala.

São definidos três períodos: manhã, tarde e noite. Sendo o turno das 8H00 às 13H00, turno da tarde das 13H00 às 18H00, e o turno da noite das 18H00 às 00H00.

Nos horários podemos ainda considerar dois tipos de restrições, as obrigatórias (restrições fortes), isto é, aquelas que não podem ser infringidas, e as restrições que não precisam de ser obrigatoriamente cumpridas (restrições fracas) mas que têm influência na qualidade da solução.

Os horários para serem considerados admissíveis devem respeitar as seguintes restrições (restrições fortes):

- Uma turma não pode ter duas aulas ao mesmo tempo;
- Uma ocorrência semanal tem apenas um tipo de aulas de um determinada unidade curricular;
- Um estudante não pode ter duas aulas teóricas seguidas;
- Num tipo de aula, a sala tem que ter pelo menos um estudante sendo o número máximo a capacidade da sala;
- Uma aula, de um determinado tipo, não pode ultrapassar o número de estudantes permitido por tipo de aula;

- Duas aulas não podem ocorrer ao mesmo tempo na mesma sala;
- Uma aula tem um docente;
- Um docente não pode leccionar duas aulas ao mesmo tempo;
- Um docente não pode leccionar mais de dezasseis horas semanais, com uma média anual de doze horas.

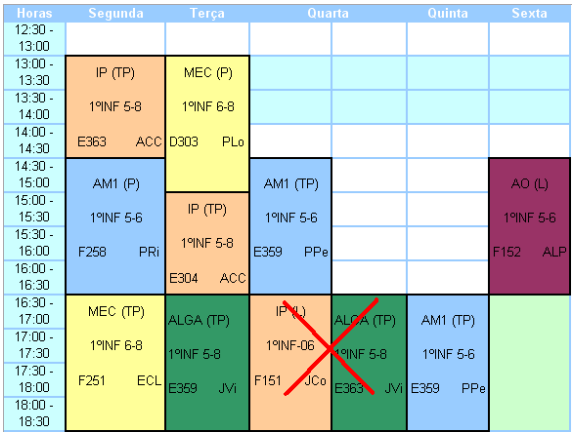


Figura 16 - Restrições fortes - Horário inadmissível

As restrições fracas não precisam de ser obrigatoriamente cumpridas, mas se houver um número elevado deste tipo de restrições o horário é considerado inadmissível. Para podermos avaliar a qualidade do horário, vamos estabelecer uma pontuação para cada restrição. Quando um horário tiver um determinada pontuação (muitas restrições fracas) é considerado inadmissível.

Restrições fracas identificadas:

- O horário das turmas não deve ter interrupções;
- Minimizar o número de aulas fora do período definido para a turma;
- Os estudantes não devem ter mais de três horas de aulas da mesma unidade curricular seguidas;
- Os docentes não devem de ter mais de quatro horas de aulas consecutivas;
- O horário dos docentes não deve ter interrupções. Se existirem interrupções, deverá ser no máximo de uma hora.
- Um docente não deve leccionar mais que duas aulas por dia a uma mesma turma;
- O número de dias que o docente lecciona deve ser minimizado.

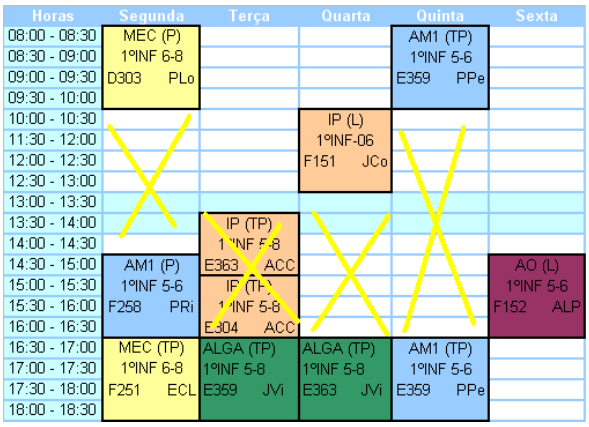


Figura 17 - Restrições fracas - Horário inadmissível

4. Modelo proposto

4.1. Introdução

Depois da análise dos métodos de optimização existentes, levando em consideração as características do problema de construção de horários, optou-se por desenvolver um modelo com base em programação lógica com restrições com auxílio do algoritmo genético. O problema é reduzido à representação de restrições e à evolução de cromossomas, recorrendo-se a técnicas heurísticas para encontrar a solução óptima.

A escolha da programação lógica com restrições deve-se ao facto do OCL trabalhar com restrições, logo é o algoritmo mais adequado para trabalhar com a referida ferramenta.

Como a programação lógica com restrições necessita de ter um algoritmo para a geração dos vários sucessores, é necessário escolher um segundo algoritmo. A escolha do algoritmo genético deve-se ao facto de que a abordagem baseada em grafos torna-se inviável pelo grande número de vértices e arestas necessário para representar o problema por completo.

A abordagem de algoritmos meméticos pode percorrer soluções descartadas anteriormente.

A abordagem de *simulated annealing* apresenta a desvantagem de reduzir muito a lista de soluções possíveis, o que torna inviável a sua aplicação no modelo estudado, pela quantidade de possíveis soluções. A técnica de *tabu search* é descartada pelo mesmo motivo.

A abordagem de programação linear, *branch-and-bound* e teoria dos grafos apresentam um crescimento exponencial em função do número das unidades curriculares. Como o modelo leva em conta muitas unidades curriculares, esta abordagem também se torna inviável.

Programação lógica gera um problema típico de explosão combinatória e inviabiliza o uso em aplicações de médio e grande porte, porquê os tempos computacionais tornam-se elevados, caso uma procura ocorra em ramos da árvore onde não exista solução viável.

GRASP acha uma solução inicial de forma gulosa, e melhora a solução através de uma busca local, isto produz pouca variedade na solução, podendo assim, ficar preso a um mínimo local.

Algumas das vantagens de se utilizar algoritmos genéticos são (Goldberg, 1989):

- podem resolver problemas complexos rápida e de forma confiável;
- a construção de algoritmos genéticos e modelos existentes é geralmente simples;
- são extensíveis;
- são fáceis de combinar com outros métodos.

Algoritmos genéticos são versáteis e requerem pouco conhecimento sobre a função a ser optimizada. Em geral, algoritmos genéticos rapidamente descobrem sub-regiões de alta qualidade em vastos espaços de procura, mas depois demoram a convergir. Para funções desconhecidas, descontínuas e não diferenciáveis, algoritmos genéticos estão entre os mais indicados.

4.2. Algoritmo genético

Algoritmos genéticos são um conjunto de modelos computacionais inspirados na genética. Estes algoritmos modelam uma solução para um problema específico numa estrutura de dados como a de um cromossoma e aplicam operadores que recombinam estas estruturas, preservando informações críticas.

Um algoritmo genético é um procedimento que mantém uma população de estruturas (chamadas indivíduos), representando possíveis soluções de um determinado problema. Estas estruturas são então avaliadas para gerar oportunidades reprodutivas, para que cromossomas que representam uma solução "melhor" tenham mais hipóteses de se reproduzirem do que os que representam uma solução "pior". A definição do que seja uma solução "melhor" ou uma solução "pior" é tipicamente relacionada à população actual.

Um algoritmo genético é, assim, qualquer modelo computacional baseado em população que utiliza operadores de cruzamento e mutação para gerar novos pontos de amostra num espaço de procura. O maior interesse no algoritmo genético está em usá-lo como ferramenta de optimização, pois trata-se de uma poderosa ferramenta para procura de soluções de problemas de alta complexidade.

Baseado na analogia com o processo de evolução biológica das espécies, os algoritmos genéticos mantêm uma determinada informação relevante sobre o ambiente e acumulam-na durante o período de adaptação. Posteriormente, utilizam a informação acumulada para minimizar o espaço de procura e gerar novas e melhores soluções dentro de um domínio.

Deve ser observado que cada cromossoma, chamado de indivíduo no algoritmo genético, corresponde a um ponto no espaço de soluções do problema de optimização. O processo de solução adoptado nos algoritmos genéticos consiste em gerar, através de regras específicas, um grande número de indivíduos (população).

4.2.1. Principais definições

A seguir, são apresentadas as principais definições relacionadas com algoritmos genéticos:

- cromossoma ou genótipo: cadeia de caractéres, representando alguma informação relativa às variáveis do problema. Cada cromossoma representa, deste modo, uma solução do problema;
- gen ou gene: é a unidade básica do cromossoma. Cada cromossoma tem um certo número de genes, cada um descrevendo certa variável do problema. Podem ser do tipo binário, inteiro ou real;
- população: conjunto de cromossomas ou soluções;
- fenótipo: cromossoma descodificado;
- geração: o número da iteração que o algoritmo genético executa para gerar uma nova população;
- operações genéticas: operações que o algoritmo genético realiza sobre cada um dos cromossomas;
- espaço de procura ou região viável: o conjunto, espaço ou região que compreende as soluções possíveis ou viáveis do problema a ser optimizado. Deve ser caracterizado pelas funções de restrição, que definem as soluções viáveis do problema a ser resolvido;
- função objectivo ou de aptidão: construída a partir dos parâmetros envolvidos no problema. Fornece uma medida da proximidade da solução em relação a um conjunto de parâmetros. A função de aptidão permite o cálculo da aptidão de cada indivíduo e fornecerá o valor a ser usado para o cálculo da sua probabilidade de ser seleccionado para reprodução;
- aptidão bruta: saída gerada pela função de aptidão para um indivíduo da população;
- aptidão máxima: melhor indivíduo da população.

4.2.2. Aspectos Principais dos Algoritmos Genéticos

Como primeiro aspecto considerado, tem-se a representação do problema, de maneira que os algoritmos genéticos possam trabalhar adequadamente sobre eles (Carvalho, Braga, & Ludermir, 2002).

Pode-se representar as possíveis soluções de um problema no formato de um código genético, que irá definir a estrutura do cromossoma a ser manipulado pelo algoritmo.

Esta representação do cromossoma depende do tipo de problema e do que,

essencialmente, se deseja manipular geneticamente. Os principais tipos de representação e os problemas aos quais são tipicamente aplicados são apresentados na Tabela 2.1 (Holland J. H.):

Representação	Problema
Binário	Numéricos, Inteiros
Números Reais	Numéricos
Permutações de Símbolos	Baseados em Ordem
Símbolos repetidos	Agrupamento

Tabela 3 – Tipos de Representação de Cromossomas

Tradicionalmente, os indivíduos são representados tipicamente por vectores binários, nos quais cada elemento de um vector denota a presença de (1) ou ausência (0) de uma determinada característica, ou seja, o seu genótipo (Carvalho, Braga, & Ludermir, 2002) (Algoritmos Genéticos).

Definida a representação do problema, a execução do algoritmo pode ser resumida nos seguintes passos:

- Escolhe-se uma população inicial, normalmente formada por indivíduos criados aleatoriamente;
- Avalia-se toda a população de indivíduos segundo algum critério, determinado por uma função, que avalia a qualidade do indivíduo (função de aptidão ou "*fitness*");
- Em seguida, através do operador de selecção, são escolhidos os indivíduos de melhor valor (dado pela função de aptidão) como base para a criação de um novo conjunto de possíveis soluções, chamado de nova geração.

Esta nova geração é obtida pela aplicação, sobre os indivíduos seleccionados, de operações que misturem as suas características (*genes*), através dos operadores de cruzamento (*crossover*) e mutação.

Estes passos são repetidos até que uma solução aceitável seja encontrada ou até que o número predeterminado de passos seja atingido ou, então, até que o algoritmo não consiga mais melhorar a solução já encontrada.

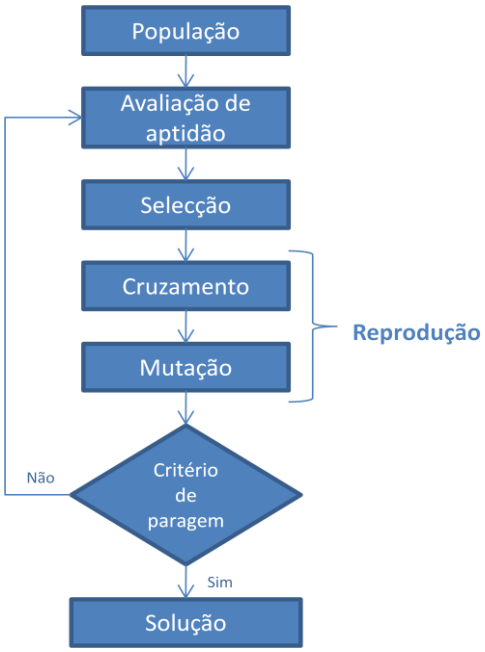


Figura 18 - Estrutura básica de um Algoritmo Genético

População - conjunto de indivíduos que estão a ser cogitados como solução e que serão usados para criar o novo conjunto de indivíduos para análise.

Indivíduos - Uma das principais formas de representação de problemas é fazer com que cada atributo seja uma sequência de *bits* e o indivíduo seja a concatenação das sequências de *bits* de todos os seus atributos.

A codificação para representar um indivíduo, usando o próprio alfabeto do atributo que se quer representar (letras, códigos, números reais, etc.), também é muito utilizada.

Avaliação de Aptidão (*Fitness*) - Neste componente é calculada, por intermédio de uma determinada função, o valor de aptidão de cada indivíduo da população. Cada indivíduo é uma entrada para uma ferramenta de análise de desempenho, cuja saída fornece medidas que permitem ao algoritmo genético o cálculo da aptidão desse indivíduo. Ainda nesta fase, os indivíduos são ordenados conforme a sua aptidão (Haralick & Elliott, 1980). Este é o componente mais importante de qualquer algoritmo genético. É através desta função que se mede a proximidade a que um indivíduo está da solução desejada ou de uma boa solução.

É essencial que a função de aptidão seja muito representativa e diferencie, na proporção correcta, as "más" soluções das "boas" soluções. Se houver pouca precisão na avaliação, uma óptima solução pode ser posta de lado durante a execução do algoritmo, além de despender recursos computacionais num espaço de busca pouco promissor.

Selecção - Dada uma população em que a cada indivíduo foi atribuído um valor de

aptidão, o processo de selecção escolhe, então, um subconjunto de indivíduos da população actual, gerando uma população intermediária. Existem vários métodos para seleccionar os indivíduos sobre os quais serão aplicados os operadores genéticos, com por exemplo o método de selecção por Roleta; o método de selecção por Torneio; o método da Amostragem Universal Estocástica; e o método da selecção Elitista (Carvalho, Braga, & Ludermir, 2002).

4.2.3. Operadores Genéticos/ reprodução

O princípio básico dos operadores genéticos é transformar a população através de sucessivas gerações, estendendo a procura até chegar a um resultado satisfatório. Os operadores genéticos são necessários para que a população se diversifique e mantenha as características de adaptação adquiridas pelas gerações anteriores.

Durante a fase de reprodução de um algoritmo genético, seleccionam-se indivíduos da população que serão cruzados para formar descendentes, que, por sua vez, constituirão a geração seguinte. Os pares são seleccionados aleatoriamente, usando-se um método que favoreça os indivíduos melhor adaptados. Logo que forem escolhidos os pares, os seus cromossomas misturam-se e combinam-se, usando os operadores de cruzamento (*crossover*) e mutação. Eles são utilizados para assegurar que a nova geração seja totalmente nova, no entanto, mantendo características de adaptação adquiridas pelas populações anteriores.

Durante este processo, os melhores indivíduos podem ser colectados e armazenados para avaliação (Algoritmos Genéticos). Neste algoritmo, as seguintes variáveis são utilizadas:

- t: tempo actual;
- d: tempo determinado para finalizar o algoritmo;
- P: população.

```
Prodimento AG
{
  t=0;
  iniciar_população(P,t);
  avaliação (P,t);
  repetir até (t=d)
  {
    t=t+1;
    Selecção_dos_pais(P,t)
   Cruzamento (P,t);
   Mutação (P,t);
  Avaliação (P,t);
  Sobreviem (P,t);
  }
 }
```

Figura 19 - Algoritmo básica do uso dos operadores

4.2.3.1. Operador de cruzamento

Este operador é utilizado após a selecção do indivíduo. Esta fase é marcada pela troca de segmentos entre "casais" de cromossomas, seleccionados para dar origem a novos indivíduos, que formarão a população da próxima geração. Esta mistura é feita tentando imitar a reprodução de genes em células. Parte das características de um indivíduo é trocada pela parte equivalente do outro. O resultado desta operação é um indivíduo que, potencialmente, combine as melhores características dos indivíduos usados como base.

A combinação dos genes responsáveis pelas características do pai e da mãe possibilita o surgimento de infinitas possibilidades de tipos diferentes, fornecendo um vasto campo de acção para a selecção e aumentando a velocidade do processo evolutivo.

Consiste em dividir aleatoriamente os cromossomas, produzindo segmentos anteriores e posteriores que realizam um intercâmbio para obter novos cromossomas (descendentes).

As três formas mais comuns de cruzamento em algoritmos genéticos são o cruzamento num ponto (Holland J. H., 1975), o cruzamento em dois pontos (Davis, 1991) e o cruzamento uniforme (Sywerda, 1989), que serão detalhados a seguir.

- Cruzamento num ponto

O operador de cruzamento num ponto selecciona aleatoriamente um ponto de corte na estrutura dos cromossomas progenitores.

Os cromossomas descendentes são constituídos pelos genes resultantes da justaposição das sequências complementares dos seus progenitores, definidas pelo ponto de cruzamento (Figura 20). O ponto seleccionado para corte foi o ponto três [8].

Cromossomas progenitores	Cromossomas descendentes
6 3 2 ↕ 7 1 1 8 7 4 2 1 ↕ 5 1 6 3 8 ↑ Ponto de corte	6 3 2 5 1 6 3 8 4 2 1 7 1 1 8 7

Figura 20 – Cruzamento num ponto

- Cruzamento em dois pontos

O operador de cruzamento em dois pontos selecciona aleatoriamente dois pontos de corte na estrutura dos cromossomas progenitores.

A constituição de cada um dos cromossomas descendentes é idêntica à constituição de um dos progenitores, à excepção da sequência de genes contida entre os pontos de cruzamento, que é idêntica à do outro progenitor (Figura 21).

Cromossomas progenitores	Cromossomas descendentes
6 3 ↕ 2 7 1 1 ↕ 8 7 4 2 ↕ 1 5 1 6 ↕ 3 8 ↑ ↑ Ponto de corte	6 3 1 5 1 6 8 7 4 2 2 7 1 1 3 8

Figura 21 – Cruzamento em dois pontos

[8] O ponto de corte pode-se situar entre 1 e l -1, correspondendo l ao número de genes dos cromossomas, ou seja, ao seu comprimento (*length*).

- Cruzamento uniforme

O operador de cruzamento uniforme selecciona aleatoriamente um determinado número de genes - e respectivos *locus* - na estrutura dos cromossomas progenitores.

Os genes cujos *locus* forem seleccionados são cruzados. Os restantes mantêm-se inalterados.

O exemplo da Figura 22 permite visualizar um cruzamento uniforme. Os *locus* seleccionados para cruzamento foram os *locus* dois, cinco, seis e oito.

Cromossomas progenitores	Cromossomas descendentes
6 3 2 7 1 1 8 7 4 2 1 5 1 6 3 8 ↑ ↑ ↑ ↑ Ponto de corte	6 2 2 5 1 6 3 8 4 3 1 7 1 1 8 7

Figura 22 – Cruzamento uniforme

4.2.3.2. 2 Operador de mutação

Após o cruzamento, cada descendente gerado pelo processo de cruzamento possui uma estrutura ordenada de genes da mesma espécie da dos seus ascendentes.

A mutação, processada ao nível dos genes com probabilidade de ocorrência normalmente baixa[9], consiste na substituição de determinados genes dos cromossomas por outros genes - alelos - existentes no património genético da população.

Admitamos que o gene representado pelo valor sete no quarto *locus* do segundo descendente da Figura 23 foi seleccionado para mutação, e que o valor dois – escolhido aleatoriamente a partir do conjunto H = {1, 2, 3, 4, 5, 6, 7, 8} - o irá substituir.

A constituição do segundo cromossoma será então a seguinte:

[9] Probabilidades de mutação elevadas transformariam o algoritmo genético num método de procura aleatório.

Cromossomas (antes da mutação)	Cromossomas (após a mutação)
4 3 1 **7** 1 1 8 8 ↑ *Locus* Seleccionado	4 3 1 **2** 1 1 8 8

Figura 23 – Mutação

4.2.4. Critérios de Paragem

Diferentes critérios podem ser utilizados para terminar a execução de um algoritmo genético. Como exemplo, os seguintes:

- após um dado número de gerações (avaliações), ou seja, um total de ciclos de evolução de um algoritmo genético;
- quando a aptidão média ou do melhor indivíduo não melhorar mais;
- quando as aptidões dos indivíduos de uma população se tornarem parecidas;
- ao conhecer a resposta máxima da função-objectivo;
- no caso de perda de diversidade da população;
- foi atingida uma solução que satisfaz todas as restrições impostas pelo problema.

4.3. Descrição do modelo

O problema da elaboração de horários escolares é caracterizado pela adjudicação de períodos de tempo às turmas da escola, de modo a que se realizem as respectivas aulas.

A adjudicação dos períodos de tempo é condicionada pela satisfação de algumas normas e restrições de ordem física e pedagógica, enumeradas na Tabela 4.

Tipo	Cód	Restrição
Forte	rh1	Uma turma não pode ter duas aulas ao mesmo tempo;
Forte	rh3	Uma turma não pode ter duas aulas teóricas seguidas;
Forte	rh6	Duas aulas não podem ocorrer ao mesmo tempo na mesma sala;
Forte	rh8	Um docente não pode leccionar duas aulas ao mesmo tempo;
Fraca	rh10	Minimizar o número aulas fora do período definido para a turma;

Tabela 4 - Restrições propostas para o escalonamento automático de reuniões escolares

4.3.1. Modelo de representação

No modelo proposto, um cromossoma é composto por uma lista de horários, sendo que o gene é um par de estrutura primária e secundária (Figura 24). A estrutura primária contém as informações relativas ao curso, semestre e turno. A lista de horários para as unidades curriculares forma a estrutura secundária do gene. Mais detalhadamente, a estrutura secundária do gene será formada por horários específicos e um par que identifica a unidade curricular e o seu grupo.

Na Figura 24, a primeira linha representa os cursos por semestre e turno. A segunda linha representa o código do horário da semana, a terceira representa a unidade curricular para aquele horário e a quarta o grupo da unidade curricular (área cientifica da unidade curricular).

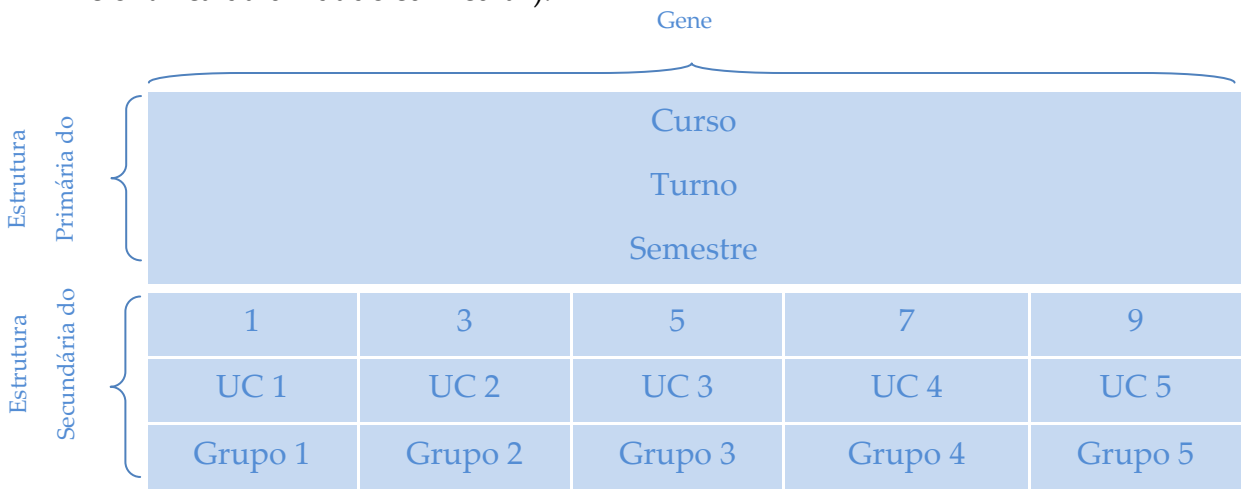


Figura 24 – Representação do Cromossoma

O modelo de representação é codificado por sequências de números naturais representativos dos períodos de tempo disponíveis para realização das aulas.

Os horários foram codificados e divididos por turnos, e podem ser agrupados de acordo com a necessidade do curso, que pode ser, por exemplo, manhã, tarde e pós-laboral. Foi considerado que todas as unidades curriculares tem uma duração de quatro horas mas não são leccionadas de forma consecutiva, por exemplo uma unidade curricular pode ter dois dias de aulas num dia da semana e as restantes noutro dia.

Assumindo a existência de seis dias disponíveis para realização de aulas e que em cada dia podem ser efectuadas nove sessões de aulas de duas horas cada, três de manhã, três de tarde e três pós-laboral, é proposto o seguinte alfabeto

H = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, …., 48, 49, 50, 51}

cujo significado se pode observar na Tabela 5.

			Seg	Ter	Qua	Qui	Sex	Sáb
Manhã	8H – 9H	1ºTempo	1	10	19	28	37	46
	9H -10H							
	10H – 11H	2ºTempo	2	11	20	29	38	47
	11H – 12H							
	12H – 13H	3ºTempo	3	12	21	30	39	48
M/T	13H – 14H							
		1ºTempo	4	13	22	31	40	49
Tarde	14H – 15H							
	15H – 16H	2ºTempo	5	14	23	32	41	50
	16H – 17H							
	17H – 18H	3ºTempo	6	15	24	33	42	51
T/PL	18H – 19H							
		1ºTempo	7	16	25	34	43	X
Pós-laboral	19H – 20H							
	20H – 21H	2ºTempo	8	17	26	35	44	X
	21H – 22H							
	22H – 23H	3ºTempo	9	18	27	36	45	X
	23H - 24H							

Tabela 5 - Períodos de tempo

4.3.1.1. Inicialização

Para a geração da população inicial, a geração é aleatória e a população inicial poderá ser uma solução.

Na criação de uma população, vamos considerar que a instituição oferece três cursos, sendo que cada um possui seis semestres em três turnos diferentes, criar-se-á um gene com cinquenta e quatro (três cursos X seis semestres X três turnos) alelos na estrutura primária. Se cada curso oferecer cinco unidades curriculares por semestre, são necessários cinco alelos na estrutura secundária do gene, para cada alelo na estrutura primária. Assim, tem-se um total de duzentos e setenta (cinquenta e quatro X cinco) alelos secundários para este exemplo.

No processo de geração da população, cada alelo do cromossoma representa um semestre de um curso num determinado turno. Na estrutura secundária do alelo são representadas as unidades curriculares oferecidas para o semestre deste curso, independente do horário.

4.3.1.2. Avaliação

A forma de avaliação do cromossoma usa seis critérios de avaliação, no entanto outros critérios podem ser adicionados. Os critérios utilizados foram:

Choque de Unidades Curriculares (Um docente não pode leccionar duas aulas ao mesmo tempo)

Uma unidade curricular pode ser oferecida em vários cursos diferentes, e a instituição deseja minimizar a quantidade de docentes para esta unidade curricular (Figura 25). Então é interessante que a unidade curricular seja distribuída em horários diferentes, de modo a ser leccionada pelo mesmo docente.

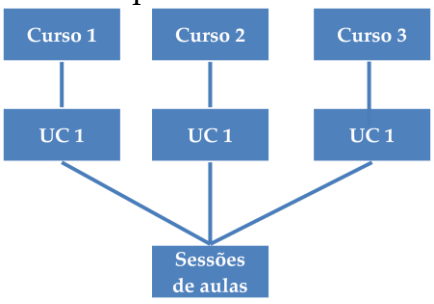


Figura 25 - Choque de UC

Para avaliar o cromossoma, é levado em consideração o seguinte: o melhor indivíduo é aquele que tem a menor repetição de uma unidade curricular para um determinado horário (numa determinada data/hora). Para isto é criada uma lista de ocorrências,

onde são colocados os períodos de tempo[10] (horário / sessões de aulas (SA)), a unidade curricular e a quantidade de ocorrências desta unidade curricular neste período de tempo.
Para calcular a nota, deve-se calcular a diferença entre a quantidade de unidades curriculares do cromossoma e o tamanho da lista de ocorrências. Quanto menor for esta diferença, menor será o número de choques de períodos de tempo existentes.
Também se verifica quantos docentes se encontram disponíveis para leccionar a unidade curricular, se o número de docentes for inferior às ocorrências num determinada período de tempo, isso quer dizer que vai haver um docente a leccionar duas turmas em simultâneo, o que é fisicamente impossível. Os cromossomas que verifiquem esta condição são automaticamente excluídos.

Exemplo:
Partindo do seguinte cromossoma:

C3_UC1	C3_UC1	C4_UC1	C2_UC1	C3_UC1	C1_UC1	C5_UC1	C2_UC1	C4_UC1	C3_UC1
SA3	SA3	SA4	SA2	SA3	SA1	SA5	SA2	SA4	SA3

Tabela 6 - Cromossoma (Choque de UC)

É criada a seguinte lista de ocorrências:

Sessões de aulas	SA1	SA2	SA3	SA4	SA5
UC	C1_UC1	C2_UC1	C3_UC1	C4_UC1	C5_UC1
Qt Ocorrências	1	2	4	1	1

Tabela 7 - Lista de ocorrências (Choque de UC)

A função de avaliação para este cromossoma retorna o valor quatro, número máximo de ocorrências que ocorrem num determinado período de tempo. Caso o número de docentes seja inferior ao número de ocorrências num determinado período de tempo, retorna -1, significa que o cromossoma é inválido, isto é, não respeita uma restrição

[10] SA – Sessões de aulas (SA) / Períodos de tempo – é composta por uma duração (duas horas), com uma hora de inicio e o dia de semana que se realiza.

forte.

Choque de Recursos (Duas aulas não podem ocorrer ao mesmo tempo na mesma sala)

As unidades curriculares estão distribuídas por grupos que definem o tipo de unidades curriculares, e as suas características em relação a necessidade ou não de um tipo recurso (Figura 26). No momento da alocação de uma unidade curricular para um período de tempo, é necessário verificar se já não existe outra do mesmo grupo neste horário, para evitar choques de recursos.

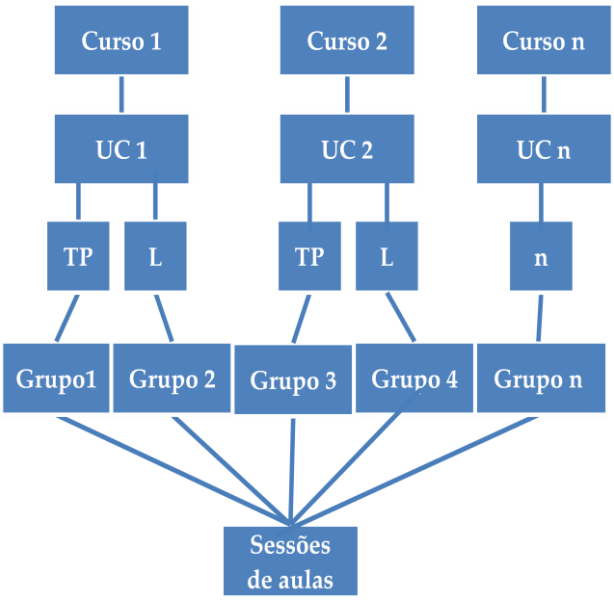


Figura 26 - Choque de recursos

Para avaliar o cromossoma, é levado em consideração o seguinte: as unidades curriculares de um mesmo grupo devem estar em períodos de tempo diferentes para maximizar a utilização dos recursos. No processo de avaliação, é criada uma lista onde consta períodos de tempo, o grupo da unidade curricular e a quantidade de ocorrências desta combinação no cromossoma. A nota é dada somando as ocorrências menos a quantidade de combinações. Quanto menor for este valor, menor será a quantidade de choques de recursos.

Também será analisado quantos recursos existem disponíveis na totalidade para leccionar a unidade curricular, caso o número de ocorrências num determinado período de tempo seja superior aos recursos existentes, o cromossoma é considerado inválido

Exemplo:

Partindo do seguinte cromossoma:

UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10
Grupo 4	Grupo 3	Grupo 4	Grupo 4	Grupo 5	Grupo 1	Grupo 4	Grupo 3	Grupo 4	Grupo 2
SA4	SA3	SA4	SA4	SA5	SA1	SA4	SA3	SA4	SA2

Tabela 8 - Cromossoma (Choque de recursos)

São calculadas as repetições:

Sessões de aulas	SA1	SA2	SA3	SA4	SA5
Grupo	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5
Qt Ocorrências	1	1	2	4	1

Tabela 9 - Lista de ocorrências (Choque de recursos)

A função de avaliação para este cromossoma retorna o valor quatro, número máximo de ocorrências que ocorrem num determinado período de tempo. Caso o número de recursos seja inferior ao número de ocorrências num determinado período de tempo, retorna -1, significa que o cromossoma é inválido, isto é, não respeita uma restrição forte.

Choque de aulas (Uma turma não pode ter duas aulas ao mesmo tempo)

Um estudante pertence a um curso e está inscrito a uma turma que tem um horário escolar (Figura 27), neste horário não pode ter, num período de tempo, duas unidades curriculares em simultâneo. No momento da alocação de uma unidade curricular para um período de tempo, é necessário verificar se já não existe outra alocada no mesmo período de tempo.

Aplicação da OCL à especificação do problema de elaboração de horários

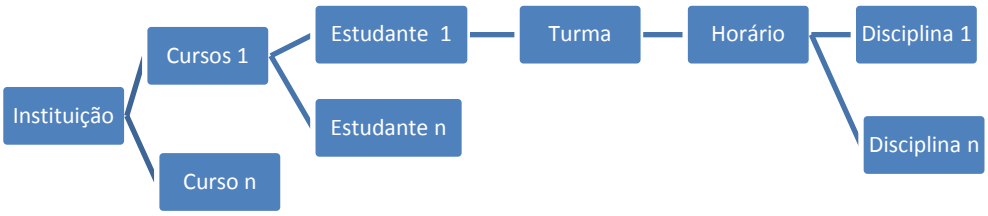


Figura 27 - Choque de aulas

Para avaliar o cromossoma, é levado em consideração o seguinte: as unidades curriculares de um determinado horário escolar devem estar em períodos de tempo diferentes, caso contrário o cromossoma é considerado inválido. No processo de avaliação, é criada uma lista onde consta períodos de tempo, o grupo da unidade curricular e a quantidade de ocorrências desta combinação no cromossoma. A nota é dada avaliando se algum período de tempo / sessão de aula tem mais do que uma ocorrência.

Exemplo:

Partindo do seguinte cromossoma:

UC1	UC2	UC3	UC4	UC5
SA1	SA3	SA4	SA4	SA5

Tabela 10 - Cromossoma (Choque de aulas)

São calculadas as repetições:

Sessões de aulas	SA1	SA2	SA3	SA4	SA5
Qt Ocorrências	1	0	1	2	1

Tabela 11 - Lista de ocorrências (Choque de aulas)

A função de avaliação para este cromossoma retorna o valor menos um, porque existem duas unidades curriculares no mesmo período de tempo, isto é, não respeita uma restrição forte.

Aulas teóricas seguidas (Uma turma não pode ter duas aulas teóricas seguidas)

As unidades curriculares estão divididas por tipo de aula/sessão (Figura 26). No momento da alocação de uma unidade curricular para um período de tempo, é necessário verificar se já não existem aulas, do mesmo tipo, seguidas.

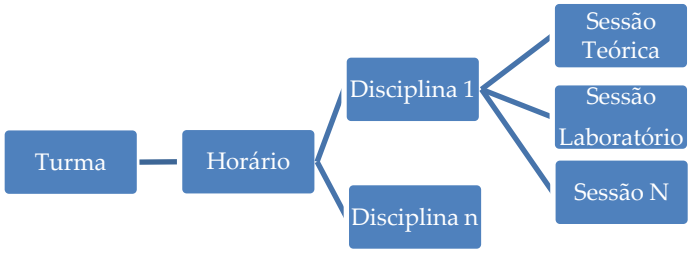


Figura 28 - Aulas teóricas seguidas

Para avaliar o cromossoma, é levado em consideração o seguinte: uma sessão/aula de uma determinada unidade curricular de um determinado horário escolar, caso seja do tipo teórica, não deve ser igual ao tipo de aula no período de tempo seguinte, caso contrário o cromossoma é considerado inválido. No processo de avaliação, é criada uma lista onde constam unidades curriculares por tipo de aulas e um factor de avaliação que é calculado da seguinte forma: caso o tipo de aula onde está posicionado não seja do tipo teórica, o factor tem valor zero, caso seja do tipo teórico, é necessário verificar se no período seguinte existe uma aula do mesmo tipo, em caso afirmativo o factor fica com valor um e caso negativo fica com o valor zero. A nota é dada avaliando a soma dos factores de todas as ocorrências, se for diferente de zero o cromossoma é considerado inválido.

Exemplo:

Partindo do seguinte cromossoma:

UC	UC1 - T	UC1 - TP	UC3 - T	UC4 - P	UC5 - L
Factor	0	0	1	0	0

Tabela 12 - Cromossoma (Aulas teóricas seguidas)

Efectuando a soma dos factores, verificamos que é diferente de zero, logo a função de avaliação para este cromossoma retorna o valor menos um, porque existem duas aulas / sessões do tipo teórico seguidas, isto é, não respeita uma restrição forte.

Minimizar o número aulas fora do período/turno definido para a turma

Os horários foram codificados e divididos por turnos, e podem ser agrupados de acordo com a necessidade do curso, que pode ser, por exemplo, manhã, tarde e pós-

laboral.

Normalmente os turnos são distribuídos conforme os semestres, por exemplo, se os estudantes do primeiro semestre têm aulas no turno da manhã, os estudantes do terceiro semestre têm aulas no turno da tarde. Este desfasamento é importante, para que os estudantes que têm unidades curriculares em atraso possam frequentar as aulas dos dois semestres. Só existem turmas em horário pós-laborais caso existam estudantes interessados suficientes.

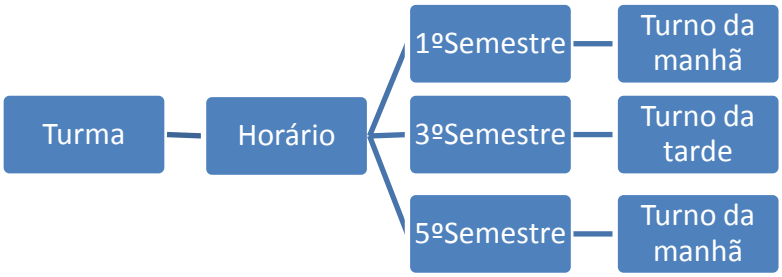


Figura 29 - Minimizar o número aulas fora do período

Para avaliar o cromossoma, é levado em consideração o seguinte: um horário só pode ter um determinado número de sessão/aula fora do turno correspondente. Inicialmente é definido qual o número máximo de infracções, se número máximo for ultrapassado o cromossoma é considerando inválido e devolve menos um, caso contrário devolve o número de infracções.

No processo de avaliação, é criada uma lista onde consta as turmas, os semestres, os turnos e o número de infracções.

Exemplo:

Partindo do seguinte cromossoma:

Turma	Turma1	Turma2	Turma 3	Turma 4	Turma 5
Semestre	1Sem	1Sem	3Sem	3Sem	5Sem
Turno	Manhã	Manhã	Tarde	Tarde	Manhã
Infracções	2	1	1	1	1

Tabela 13 - Cromossoma

Efectuando a soma das infracções, verificamos que o valor é seis, se consideramos

como número máximo de infracções dez (este número pode ser variável), logo a função de avaliação para este cromossoma retorna o valor seis.

4.3.1.3. Selecção

A selecção de cromossomas, pode ser efectuada através de um de dois operadores: um operador proporcional - o método de selecção da roleta - ou um operador misto - o método de selecção com normalização linear – que combina características de selecção proporcional e de selecção por ordem de classificação.

Para assegurar que as melhores soluções de cada geração não são destruídas pelo processo de reprodução foi implementado um mecanismo de elitismo, através do qual é possível a selecção automática dos melhores cromossomas da população actual e a respectiva introdução na população da próxima geração.

4.3.1.4. Reprodução

Cruzamento

O algoritmo de cruzamento é realizado independentemente das estruturas primárias e secundárias do cromossoma, de modo a manter a consistência nas turmas futuras. A herança de um gene da mãe num cromossoma, implica na troca, no cromossoma do pai, do gene da posição correspondente ao herdado pelo gene igual ao que acaba de ser herdado. Este cruzamento é semelhante ao de cruzamento de ciclo de Goldberg (Goldberg, 1989).

Entretanto, em vez de ser aplicado sobre todo o cromossoma, um ponto de corte é seleccionado aleatoriamente e o ciclo só é realizado com uma parte do cromossoma.

Cruzamento da Estrutura Primária

Para exemplificar a utilização do operador de cruzamento, considere o cromossoma descrito na Figura 30, onde C1 representa o curso 1, C2 o curso 2, e assim por diante.

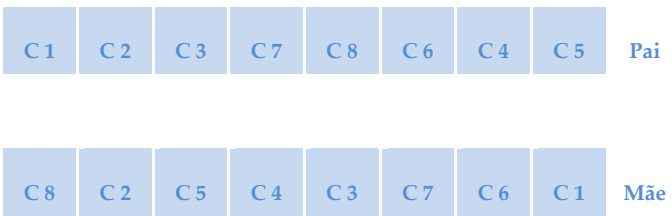


Figura 30 - Cruzamento da Estrutura Primária

Uma posição é seleccionada aleatoriamente como ponto de corte, dividindo o cromossoma pai e mãe em duas partes (Figura 31). Neste caso a posição será a três:

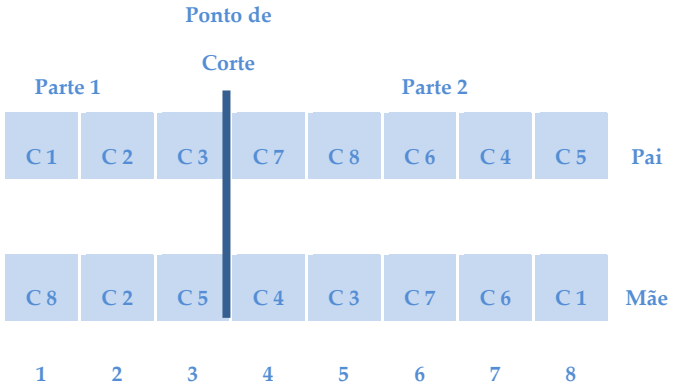


Figura 31 - Selecção do Ponto de Corte para o Cruzamento

O cruzamento é iniciado pela parte 1 do cromossoma mãe. O gene C8 é colocado na posição 1 do cromossoma filho. Em seguida, no cromossoma pai, o alelo C8 (que já foi herdado) é trocado com o alelo C1 (que está na posição já preenchida no filho). A nova constituição dos cromossomas pode ser vista na Figura 32.

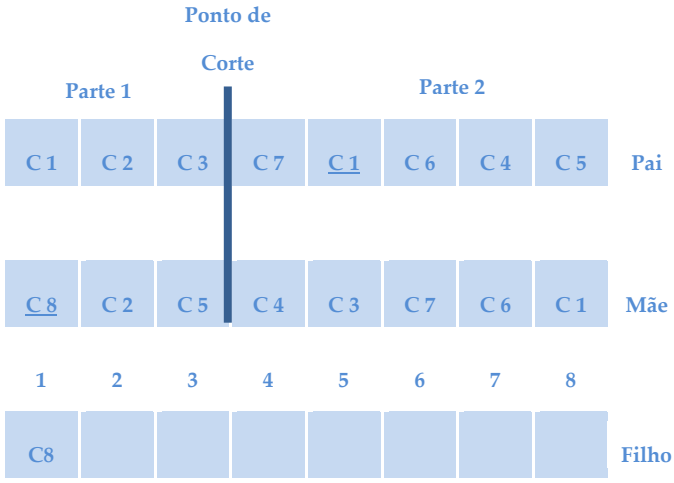


Figura 32 - Troca de Informações no Locus Gênico 1

Para a completar a posição 2 do cromossoma filho, é utilizado o alelo C2 do cromossoma mãe. Novamente, no cromossoma pai, o alelo C2 é trocado pelo alelo da

posição 2 (Figura 33). Por acaso, o alelo C2 do cromossoma mãe está na mesma posição no cromossoma pai.

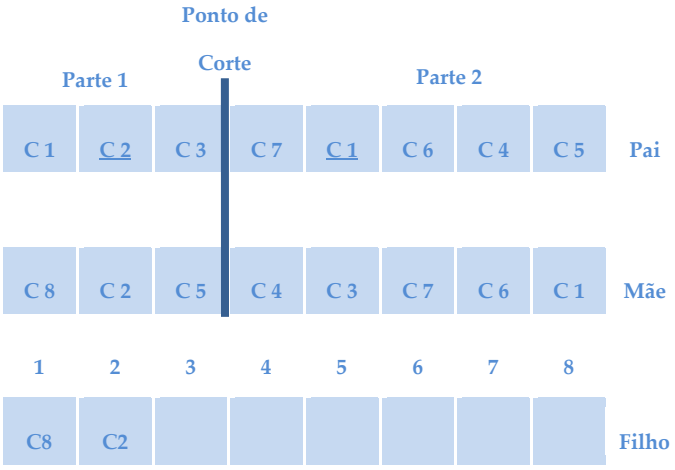


Figura 33 - Troca de Informações no Locus Gênico 2

O mesmo processo será realizado para a posição 3 do cromossoma filho. O alelo C5 é herdado do cromossoma mãe, e no cromossoma pai, o alelo C5 é substituído pelo alelo da posição 3 (Figura 34).

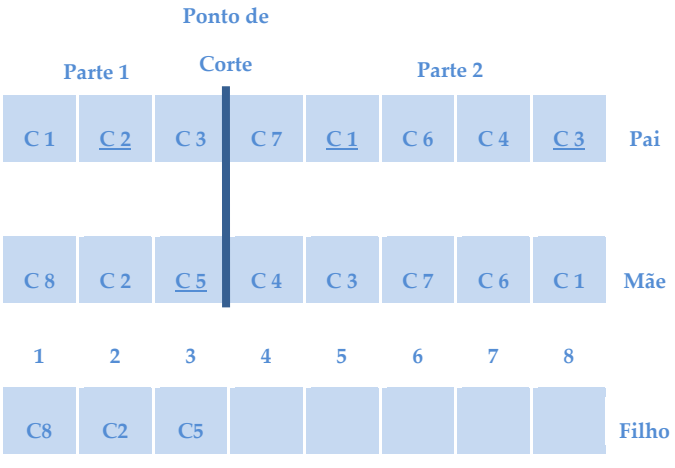


Figura 34 - Troca de Informações no Locus Gênico 3

Após realizar as trocas até ao ponto de corte definido, o preenchimento da segunda

parte do cromossoma filho é feita com a segunda parte do cromossoma pai, sem ocorrer mais nenhuma troca (Figura 35).

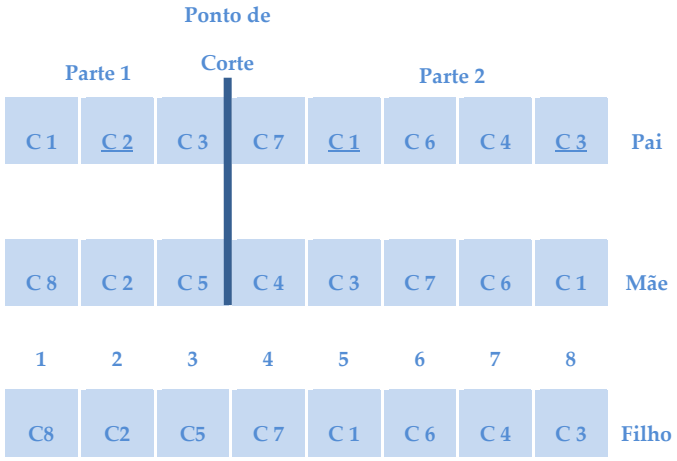


Figura 35 - Complementação do Cromossoma Filho

O processo de geração é repetido, com a inversão dos cromossomas pai e mãe, de forma a gerar um segundo filho. O resultado do cruzamento são dois filhos, como mostra a Figura 36. Isto é necessário para manter a população.

Como o cruzamento entre dois indivíduos gera dois filhos, a nova geração terá o mesmo número de indivíduos da população anterior.

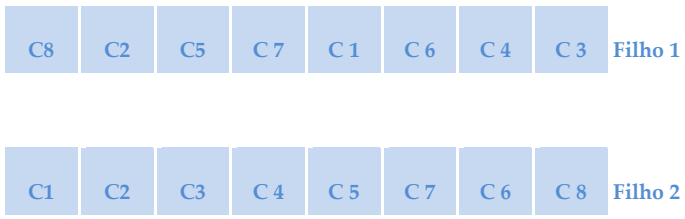


Figura 36 - Filhos do Cruzamento da Estrutura Primária do Gene

A operação de cruzamento da estrutura primária, ocorre sob as restrições de população e com uma probabilidade, definida pelo utilizador.

Cruzamento da Estrutura Secundária

O mesmo método de cruzamento aplicado na estrutura primária também será aplicado na estrutura secundária do gene (Figura 37), que contém as informações

sobre as unidades curriculares do currículo relacionadas com a turma do cromossoma.

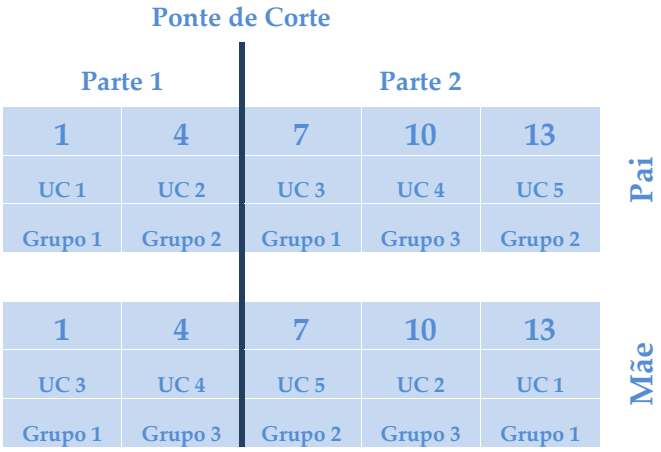


Figura 37 - Cruzamento da Estrutura Secundária

Pode-se notar que os genes 1 e 2 não se encontram na segunda parte do cromossoma pai. Com a utilização do cruzamento não há perda de informação no cromossoma, mantendo-se todas as unidades curriculares desta turma (Figura 38).

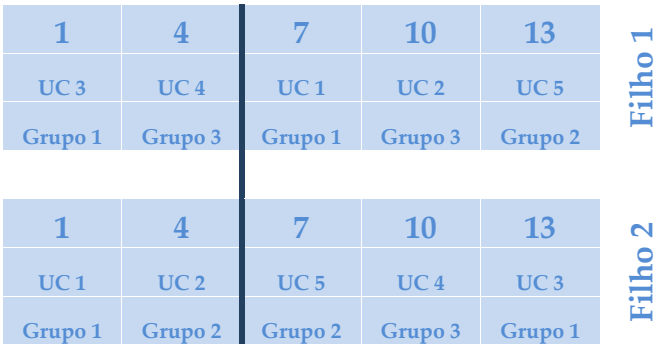


Figura 38 - Filhos do Cruzamento da Estrutura Secundária

A operação de cruzamento da estrutura secundária do gene ocorre sob as restrições de população com probabilidade, também definida pelo utilizador.

Mutação

O operador de mutação será aplicado sobre o cromossoma e a estrutura secundária do gene. A razão para tal é que poderão ocorrer locais do cromossoma que não

sofrem alterações ou mantêm a ordem da sequência da geração inicial.

Mutação da Estrutura Primária do Gene

Para ilustrar a aplicação do operador de mutação na estrutura primária do cromossoma, será estudado o cromossoma representado na Figura 39.

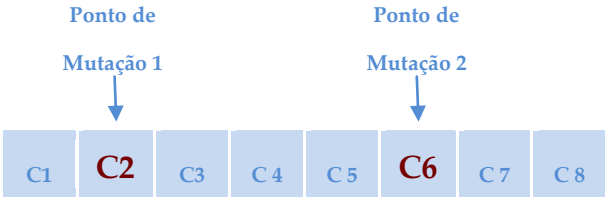


Figura 39 - Pontos de Mutação da Estrutura Primária

Após sortear dois pontos para fazer a mutação, os valores dos genes são trocados entre si. Neste exemplo foram sorteadas as posições 2 e 6. Como resultado da mutação da estrutura primária é gerado um novo cromossoma, representado na Figura 40.

Figura 40 - Resultado da Mutação na Estrutura Primária

Este operador não perde informações porque há somente uma troca das posições dos genes, sem alteração dos seus conteúdos.

A operação de mutação da estrutura primária do gene, de forma semelhante ao cruzamento, é aplicada sobre os horários de uma turma com probabilidade, definida pelo utilizador.

Mutação da Estrutura Secundária

A mutação da estrutura secundária é igual à da estrutura primária. Como demonstrado na Figura 41 e na Figura 42, duas unidades curriculares são escolhidas aleatoriamente e os seus horários são trocados.

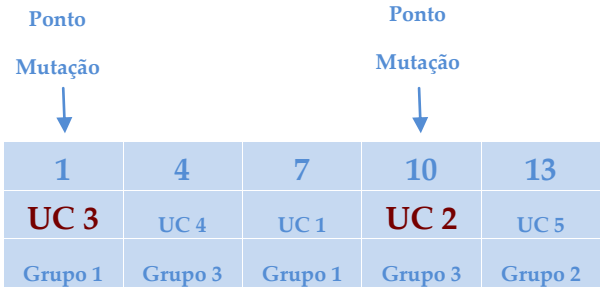


Figura 41 - Mutação da Estrutura Secundária

Da mesma forma que na estrutura primária, a utilização deste operador não leva à perda de informações.

1	4	7	10	13
UC 2	UC 4	UC 1	UC 3	UC 5
Grupo 3	Grupo 3	Grupo 1	Grupo 1	Grupo 2

Figura 42 - Resultado da Mutação na Estrutura Secundária

A mutação da estrutura secundária do gene também ocorre em função de uma probabilidade, definida pelo utilizador.

4.3.1.5. Critério de paragem

A finalização do algoritmo verifica-se quando ocorre uma das seguintes condições:

O algoritmo gera uma solução em que as restrições não sofrem qualquer infracção, o que significa que foi encontrada a solução óptima do problema;

Foi atingido o número máximo de gerações definido como limite de execução do algoritmo. Neste caso é terminada a execução do algoritmo adoptando-se a solução encontrada até ao momento que mais se adapte à resolução do problema.

4.3.1.6. Avaliação da Melhor Solução

A melhor solução será dada pelo cromossoma que tenha o maior número de restrições respeitadas. Aplicando-se as funções de avaliação, será a melhor solução o cromossoma que tiver a soma dos resultados mais próxima de zero.

4.3.2. Resultados computacionais

Para realizar os testes foi utilizada uma ferramenta chamada FET (versão fet-4.2.8) (FET).

FET é um *software open source*, que gera automaticamente horários escolares de média ou superior dificuldade. A versão 4.2.8 utiliza o algoritmo genético para a geração. Ele está licenciado sob a GNU GPL. Normalmente, FET é capaz de elaborar horários complicado no máximo 20 minutos. Para os horários mais simples, pode levar um tempo mais curto, menos de 5 minutos (em alguns casos, uma questão de segundos). Para horários extremamente difíceis, pode levar um longo tempo, uma questão de horas.

Foram criadas três instâncias do problema para efeitos de testes. A metodologia adoptada foi a abordagem do caso mais simplificado a uma situação mais crítica do problema.

4.3.2.1. Instâncias do problema

4.3.2.1.1. Teste 1

Nesta instância do problema procurou-se pontuar um caso mais simples. Considerou-se um subconjunto de catorze docentes e cem estudantes de um curso, cada estudante está inscrito a cinco unidades curriculares. As aulas do tipo teóricas práticas ou teóricas tem uma capacidade de cinquenta estudantes por aula e as aulas do tipo práticas/laboratórios tem uma capacidade de vinte e cinco estudantes por aula, esta regra aplica-se aos três testes. A distribuição de aulas foi dada a quinze docentes de forma uniforme e adicionadas seis restrições. Foram criadas onze salas e adicionadas três restrições.

Resultados

Para esta instância do problema, foram feitas cinco execuções do algoritmo, a fim de se aferir o número médio de gerações, assim como o tempo médio de alcance da melhor solução. O critério de parada utilizado foi o alcance da solução óptima.

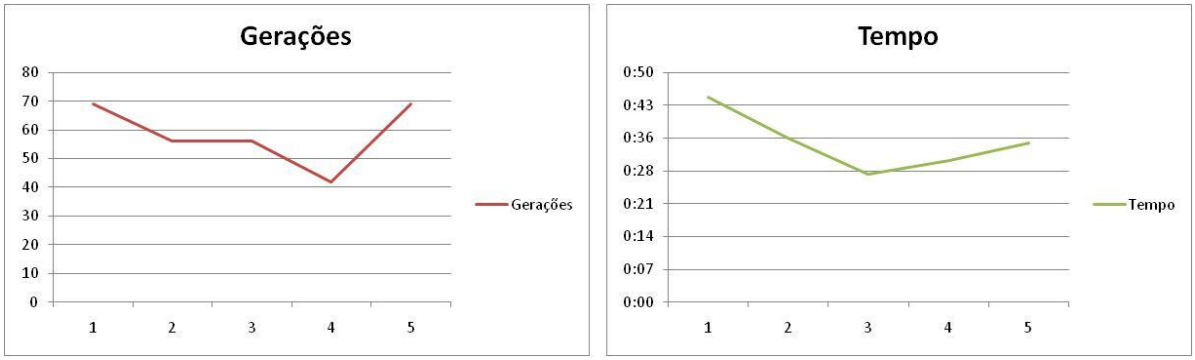


Figura 43- Resultados do teste 1

A solução óptima foi alcançada por três vezes num tempo médio de 35 segundos.

4.3.2.1.2. Teste 2

Nesta instância do problema, abordou-se um caso um pouco mais complexo dentro do contexto da instituição, aumentando o número de estudantes. Considerou-se um subconjunto de catorze docentes e duzentos estudantes de um curso, cada estudante está inscrito a cinco unidades curriculares. A relação às restrições manteve-se as do teste anterior.

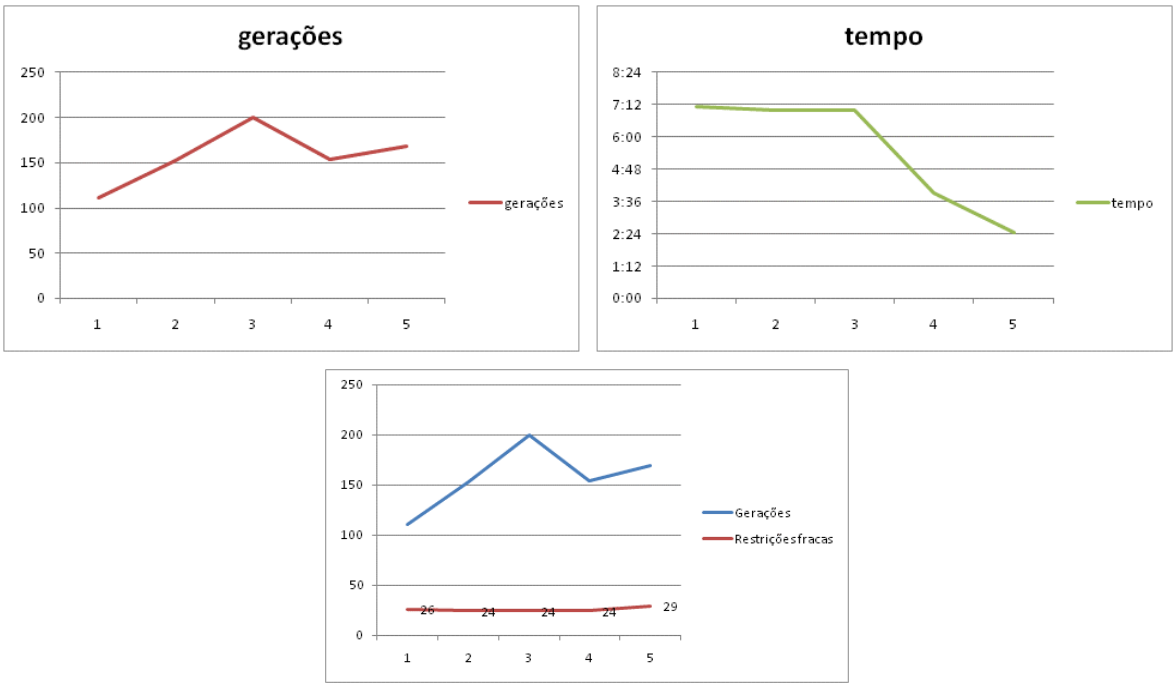


Figura 44- Resultados do teste 2

Neste teste não se alcançou a solução óptima, de todas as vezes que se efectuou a

geração de horário reduziu-se o número de restrições que não foram respeitadas. As restrições não respeitadas eram restrições fracas, e estavam todas relacionadas com a restrição "Minimizar os furos do horário do docente".

4.4. Programação Lógica com Restrições

A Programação Lógica com Restrições (PLR) evidencia um paradigma computacional que combina a natureza declarativa da programação em lógica com a eficiência dos métodos de resolução de problemas que recorrem a restrições. Esta tecnologia tem-se mostrado eficaz na resolução de problemas que são intratáveis quando se recorrem a outras técnicas. Entre esses problemas encontram-se diversas classes de problemas de natureza combinatória, tais como os problemas de escalonamento, de planeamento e de atribuição de recursos. É de notar que uma das vantagens mais significativas deste modelo passa por oferecer um menor tempo de desenvolvimento de programas e uma eficiência comparável à das linguagens imperativas. Freuder (Freuder, 1997) afirma que a programação com restrições representa a abordagem mais aproximada, alguma vez efectuada, ao santo Graal da programação, em que um utilizador declara o problema e o computador soluciona-o.

4.4.1. Conceitos Gerais Sobre Restrições

A problemática dos sistemas, que têm o seu comportamento condicionado pela existência de restrições, surge naturalmente em muitas áreas da actividade humana.

São meios naturais de expressão para formalizar as regularidades e dependências que estão na base dos mundos computacionais e físicos, bem como nas suas abstracções matemáticas. Na actividade humana as restrições constituem um ponto-chave do senso comum na condução do raciocínio. Frases como "eu posso estar lá amanhã das quatro às seis horas da tarde" englobam uma restrição típica usada para o planeamento de tempo. Dependendo do domínio do problema, poder-se-ão encontrar outros exemplos de restrições: "a soma dos ângulos de um triângulo é 180 graus"; "a soma das correntes que fluem num nó é igual a zero"; e "uma resistência num circuito eléctrico obedece à lei de Ohm".

Intuitivamente, uma restrição pode ser considerada como uma condicionante num espaço de possibilidades (Hentenryck & Saraswat, 1997). As restrições matemáticas especificam de uma forma clara e precisa as relações entre diversos parâmetros desconhecidos (variáveis), cada um tomando um dado valor no domínio considerado. As restrições restringem os valores possíveis que as variáveis podem tomar e representam alguma informação (parcial) sobre as variáveis em jogo. Estas apresentam algumas propriedades muito interessantes, e expressas no texto que se segue:

- Como se referiu, as restrições traduzem informação parcial acerca de uma dada questão ou problema, uma vez que de uma restrição, por si só, não é de

esperar que determine o valor das variáveis do problema;

- As restrições são aditivas. A uma restrição r1 pode ser adicionada uma outra restrição r2. A ordem segundo a qual as restrições são impostas é irrelevante, o que está em jogo é que no final à conjunção dos termos que denotam as restrições seja atribuído o valor de verdadeiro;
- As restrições raramente são independentes; ou seja, geralmente partilham variáveis. A colocação das restrições r1 e r2 resulta na obtenção da restrição X + Y = Z;
- As restrições são ainda não direccionais: Considerando a restrição X + Y = Z, esta pode ser usada para determinar a sua forma equivalente em X (X = Z - Y) ou em Y (Y = Z - X); e
- Finalmente, as restrições são de natureza declarativa pelo facto de apenas indicarem as relações que devem ser asseguradas entre variáveis sem especificar um procedimento computacional para estabelecer esse relacionamento.

4.4.2. Programação com Restrições

A Programação com Restrições (PR) passa pelo estudo de sistemas computacionais baseados em restrições. O princípio base em que assenta a PR, e a sua utilização na resolução de problemas, passa pela indicação das restrições para o problema em causa e, consequentemente, por se encontrarem as soluções que satisfaçam essas condicionantes. Os primeiros desenvolvimentos que conduziram à PR podem ser encontrados no domínio da inteligência artificial (IA) e remontam aos anos sessenta e setenta((Sutherland, 1963); (Fikes, 1968); (Montanari, 1972); (Waltz, 1975)).

Estes avanços tecnológicos focalizavam-se na representação e manipulação explícita de restrições em sistemas computacionais. No entanto, apenas nas décadas de 80 e 90 se desenvolveu uma crescente consciencialização de que estas ideias podiam fornecer a base para uma abordagem poderosa à temática da programação, modelação e resolução de problemas (Steele, 1980). Por outro lado foram desenvolvidos esforços para explorar estes pressupostos e unificá-los numa única estrutura conceptual (Jaffar & Lassez, 1987).

Actualmente, nota-se o aparecimento de duas correntes, pese embora o facto de serem complementares, para o tratamento da PR: a satisfação de restrições e a resolução de restrições. Ambas partilham a mesma terminologia, mas possuem origens distintas e socorrem-se de tecnologias diferentes de resolução de problemas.

A satisfação de restrições lida com os problemas definidos sobre domínios de valores discretos, do qual fazem parte os domínios finitos, sendo actualmente a mais usada

na maioria das aplicações. A resolução de restrições possui todas as propriedades base da PR. No entanto, neste caso as restrições são definidas (na sua maior parte) sobre domínios infinitos ou mais complexos. Em vez dos métodos combinatórios para a satisfação de restrições, os algoritmos de resolução de restrições são baseados em técnicas matemáticas tais como a diferenciação automática, séries de Taylor, método de Newton, ou a programação linear.

4.4.3. Programação Lógica e Programação com Restrições

A Programação em Lógica (PL) corporiza um paradigma de programação baseado num subconjunto da lógica de primeira ordem. Os programas em lógica são de natureza declarativa, dado que indicam os relacionamentos lógicos necessários para resolver um dado problema. O sistema subjacente é responsável por efectuar os cálculos lógicos que permitem que a computação ocorra. Um sistema baseado em lógica usa um conjunto de regras fornecidas pelo programador (programa) para responder às perguntas que lhe são endereçadas. Em geral, os programas em lógica podem ser usados para comprovar uma dada afirmação, ou para determinar qual o conjunto de instanciações das variáveis que faz com que a uma dada pergunta seja atribuído o valor de verdadeiro. A propriedade declarativa das linguagens de programação em lógica não só é apelativa para os programadores, como facilita a existência de uma semântica clara e bem definida, com uma base teórica bem fundamentada.

As restrições surgem como a extensão natural à estrutura da programação em lógica ao partilharem muitas das propriedades acima discutidas. Jaffar e Lassez (Jaffar & Lassez, 1987) mostraram que o Prolog pode ser visto como um exemplo de um esquema muito particular de PLR. De facto, a PL é baseada num paradigma computacional de natureza declarativa em que um programa é uma teoria lógica e a cada passo computacional apresenta uma solução para um sistema de equações de termos lógicos por intermédio do algoritmo de unificação. A sua natureza declarativa faz com que a PL esteja próxima do princípio base da programação com restrições, em que se declara o que tem de ser satisfeito mas não como. Por outro lado, o processo de pesquisa de soluções em profundidade com retrocesso é em tudo similar aos procedimentos de retrocesso padrão usados para solucionar problemas com restrições.

Em PLR a lógica é usada para especificar um conjunto de possibilidades para resolução do problema que são exploradas por intermédio de um simples método de pesquisa, enquanto as restrições são usadas para minimizar o espaço de soluções pela eliminação em avanço de vias alternativas para a resolução do problema que no futuro se irão mostrar inconsequentes. O programador de aplicações pode declarar

os factores que têm de ser tidos em conta em qualquer solução – as restrições –, declarar as possibilidades – o programa em lógica – e usar o sistema para combinar o raciocínio e a pesquisa. As restrições são usadas para restringir e guiar a pesquisa.

4.5. Identificação das restrições

4.5.1. Diagrama de classes

Pretende-se com este diagrama, desenhar de uma forma geral o problema dos horários, ou seja, desenhar as classes principais do problema. Não será abordado o problema da Distribuição de Serviço de Docente, matrículas nas unidades curriculares nem a inscrição nas turmas.

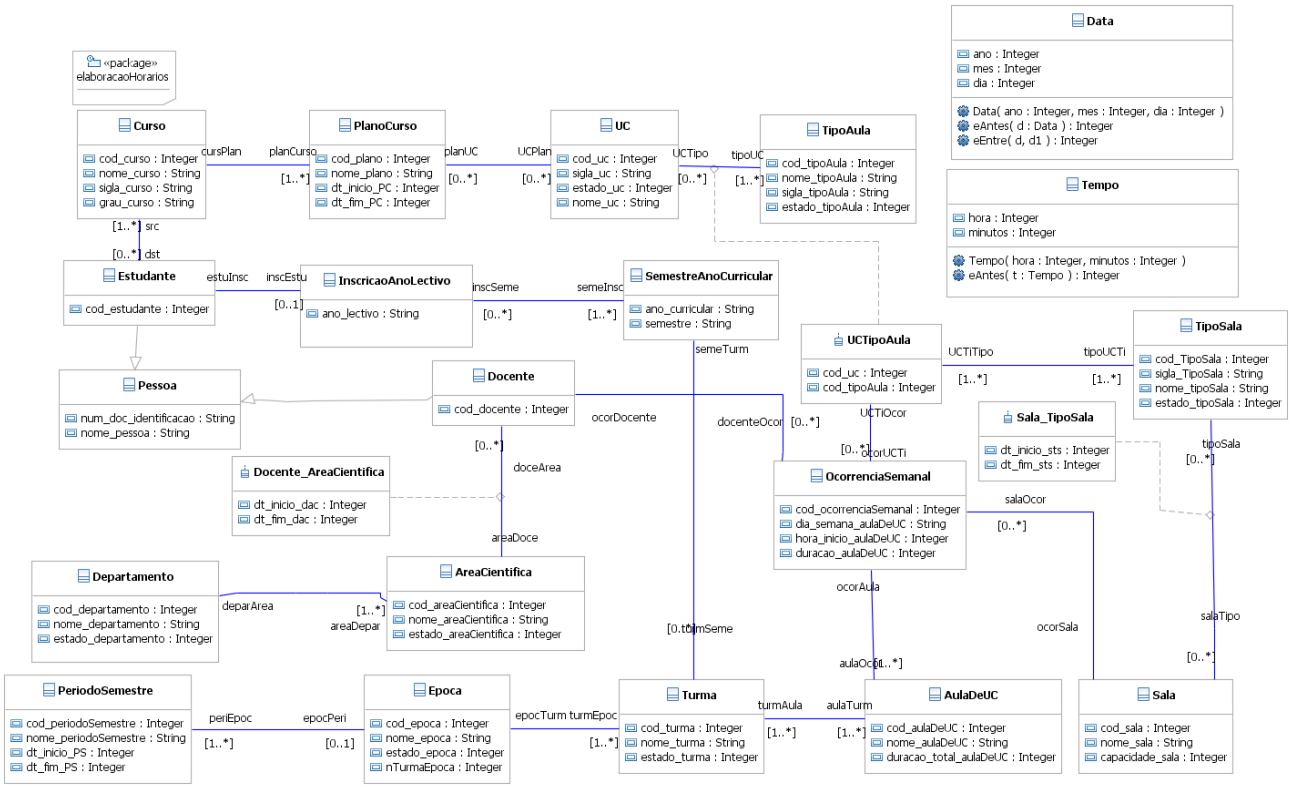


Figura 45 - Diagrama de classes

4.6. Representação e validação das restrições em OCL

Para a validação das restrições foi utilizada a ferramenta DOT (Dresden OCL Toolkit).

Este compilador é capaz de realizar a análise sintáctica (*parsing*) de expressões OCL, verificação de tipos e avaliação de restrições presentes em modelos comuns. Também é capaz de gerar, automaticamente, o código Java e SQL a partir das expressões OCL e do modelo UML fornecidos.

A DOT gera o código e avalia/controla o código-fonte da aplicação, para que sejam verificadas as restrições em tempo de execução.

A inserção do código para verificação das expressões na aplicação é feita conforme cada tipo de restrição – *invariantes*, *pré-condições* ou *pós-condições*. Para os *invariantes*, em vez de serem verificados antes e depois da execução de cada método público e após cada construtor, a sua execução ocorre sempre no fim dos métodos que modifiquem o valor de campos utilizados pelos *invariantes*.

Tive várias dificuldades na utilização desta ferramenta, como por exemplo:

- A versão existente para SQL disponibilizada na página oficial é muito antiga e desactualizada, não funciona nas últimas versões do Eclipse. Para ser possível efectuar os testes, foi disponibilizada pela Universidade Dresden uma versão ainda em desenvolvimento.
- Existe pouca documentação sobre a ferramenta (versão SQL).

4.7. Conversão das restrições em OCL para SQL

Para iniciar a conversão das restrições é necessário criar um modelo em UML (ficheiro com extensão .uml) onde será criado o diagrama de classes, e criar um ficheiro que contenha as restrições (ficheiro com extensão .ocl) onde estão as restrições necessárias, para isso utilizamos o Eclipse (instalação no anexo A). O anexo B tem descrito todos os passos necessários para a criação dos referidos ficheiros.

4.7.1. Análise do código sql gerado

Essa secção apresenta alguns exemplos de definições especificadas em OCL, produzidas a partir dos elementos definidos pelo diagrama de classes apresentado na Figura 45.

Uma restrição do tipo invariante é definida no contexto de um classificador por uma expressão através da palavra reservada *inv*. A restrição presente nas linhas 42 - 44 da Figura 46 aplica-se a todas as instâncias do Curso, enquanto que as restrições especificada nas linhas 46 - 48 aplicam-se a todas as instâncias do tipo Pessoa que, neste exemplo, correspondem às instâncias das classes Docente e Estudante.

```
42 context Curso
43 inv CursoCodUnico: Curso.allInstances()->isUnique(cod_curso)
44 and cod_curso >0
45
46 context Pessoa
47 inv PessoaCodUnico : Pessoa.allInstances() -> isUnique ( num_doc_identificacao)
48 and not self.num_doc_identificacao.oclIsUndefined()
49
```

Figura 46 - Exemplo de uma restrição do tipo invariante

```
37  CREATE OR REPLACE VIEW CursoCodUnico AS
38 (SELECT * FROM OV_Curso SELF
39  WHERE NOT (( AND (SELF.cod_curso > 0))));
40
41  CREATE OR REPLACE VIEW PessoaCodUnico AS
42 (SELECT * FROM OV_Pessoa SELF
43  WHERE NOT (( AND (NOT ()))));
```

Figura 47 - Exemplo do resultado gerado pelo OCL22SQL, restrição do tipo invariante

Pode-se inicializar um atributo através de uma expressão OCL associada ao atributo pela palavra reservada *init*. No exemplo da Figura 48, o valor inicial do atributo estado_turma e estado_epoca.

```
30 context Turma::estado_turma
31 init: true
32
33 context Epoca::estado_epoca
34 init: true
```

Figura 48 - Exemplo de expressão de inicialização de um atributo

```
22  CREATE OR REPLACE VIEW  AS
23 (SELECT * FROM OV_Turma AS SELF
24  WHERE NOT ((true)));
25
26  CREATE OR REPLACE VIEW  AS
27 (SELECT * FROM OV_Epoca AS SELF
28  WHERE NOT ((true)));
```

Figura 49 - Exemplo do resultado gerado pelo OCL22SQL, inicialização de um atributo

O resultado de uma operação de consulta pode ser definido por uma expressão OCL associada a essa operação através da palavra reservada *body*. A Figura 50 apresenta a

definição da operação de consulta eAntes.

```
16 context Data::eAntes(d:Data) :Boolean
17 body: ((ano <= d.ano) or ((ano > d.ano)
18         and ((mes <= d.mes) or ((mes >
19         d.mes) and (dia <= d.dia)))))
```

Figura 50 - Exemplo de definição do corpo de uma operação de consulta

```
6   CREATE OR REPLACE VIEW  AS
7  (SELECT * FROM OV_Data SELF
8  WHERE NOT (((SELF.ano <= SELF.ano) OR ((SELF.ano > SELF.ano)
9  AND ((SELF.mes <= SELF.mes) OR ((SELF.mes > SELF.mes)
10  AND (SELF.dia <= SELF.dia)))))));
```

Figura 51 - Exemplo do resultado gerado pelo OCL22SQL, operação de consulta

A semântica de uma operação é especificada através de pré-condições e pós-condições, conforme ilustrado pela Figura 52. Neste exemplo, três pré-condições foram definidas para a operação Data da classe Data.

```
23 context Data::Data(ano: Integer, mes: Integer, dia: Integer)
24     pre: dia >= 0 and dia <= 31
25     pre: mes >= 1 and mes <= 12
26     pre: ano > 0
27     post: self.oclIsNew()
```

Figura 52 - Exemplo de especificação de pré-condições e pós-condições

```
6   CREATE OR REPLACE VIEW data AS
7  (SELECT * FROM OV_Data AS self
8      WHERE NOT ((((((self.dia > 0)
9             AND (self.dia <= 31))
10            AND (self.mes >= 1))
11            AND (self.mes <= 12))
12            AND (self.ano > 0))));
```

Figura 53 - Exemplo do resultado gerado pelo OCL22SQL, pré-condições e pós-condições

4.7.1. Análise dos resultados

A referida ferramenta gera dois ficheiros, um que contém os *scripts* para a criação da base de dados e outro com as *views* resultantes das restrições OCL.

Para testar o código SQL gerar pelo DOT, foi criada uma base de dados e executados os ficheiros gerados. Para criar a referida base de dados foi utilizado o Microsoft SQL Server 2008. À *posteriori* foram inseridos alguns dados nas tabelas (os suficientes para realizar os testes).

O teste consiste em representar as restrições em OCL, posteriormente gerar o ficheiro

SQL e verificar se o código gerado detecta irregularidades nos dados registados na base de dados.

Exemplo nº1

Esta restrição valida se existe mais do que uma aula a ocorrer em simultâneo na mesma sala de aulas.

Não podem existir duas ou mais aulas a iniciar ao mesmo tempo na mesma sala ou a iniciar enquanto a sala ainda se encontra ocupada.

```
context OcorrenciaSemanal
  inv rh:
    OcorrenciaSemanal.allInstances()->forAll(e1, e2 |
    (e1<>e2 and e1.ocorSala =e2.ocorSala )
    implies
     (
    e1.dia_semana_aulaDeUC<>e2.dia_semana_aulaDeUC
    or
    (((e1.hora_inicio_aulaDeUC + e1.duracao_aulaDeUC) <= e2.hora_inicio_aulaDeUC or
    (e2.hora_inicio_aulaDeUC + e2.duracao_aulaDeUC) <= e1.hora_inicio_aulaDeUC)) and
        e1.hora_inicio_aulaDeUC<>e2.hora_inicio_aulaDeUC
    )
)
```

Figura 54 - Restrição representada em OCL (exemplo 1)

```
CREATE  VIEW rh AS
(SELECT * FROM OV_OcorrenciaSemanal AS self
WHERE NOT (NOT EXISTS((SELECT temp1.PK_OcorrenciaSemanal
FROM OV_OcorrenciaSemanal AS temp1
INNER JOIN (SELECT * FROM OV_OcorrenciaSemanal) AS temp2
ON NOT((NOT (NOT (temp1.PK_OcorrenciaSemanal = temp2.PK_OcorrenciaSemanal)
AND (temp1.FK_ocorSala = temp2.FK_ocorSala))
OR (NOT (temp1.dia_semana_aulaDeUC = temp2.dia_semana_aulaDeUC)
OR ((((temp1.hora_inicio_aulaDeUC + temp1.duracao_aulaDeUC) <= temp2.hora_inicio_aulaDeUC)
OR ((temp2.hora_inicio_aulaDeUC + temp2.duracao_aulaDeUC) <= temp1.hora_inicio_aulaDeUC))
AND NOT (temp1.hora_inicio_aulaDeUC = temp2.hora_inicio_aulaDeUC)))))))));
```

Figura 55 – Código SQL gerado pelo DOT (exemplo1)

PK_OcorrenciaS...	FK_UCTiOcor	FK_ocorSala	cod_ocorrencia...	dia_semana_aul...	duracao_aulaD...	hora_inicio_aula...	
1	1	1	1	Seg	1	10	
2	1	2	2	Ter	2	10	1
3	1	2	3	Ter	2	9	
4	1	3	4	Qui	2	16	2
5	1	3	5	Qui	2	15	
6	2	4	6	Sab	2	9	
7	3	5	7	Sab	1	10	

Figura 56 – Todos os dados referentes aos horários (exemplo 1)

	PK_OcorrenciaSemanal	FK_ocorSala	dia_semana_aulaDeUC	duracao_aulaDeUC	hora_inicio_aulaDeUC
1	2	2	Ter	2	10
2	3	2	Ter	2	9
3	4	3	Qui	2	16
4	5	3	Qui	2	15

Figura 57 – Detecção de dados incorrectos (exemplo 1)

Executado o código gerado pela ferramenta DOT na base de dados, consegue-se detectar que existem duas salas que têm duas aulas sobrepostas. A sala número dois (FK_ocorSala) tem uma aula que inicia às dez horas com duração de duas horas e outra que inicia às nove horas com duas horas de duração, as aulas encontram-se sobrepostas entre as dez horas e as onze horas. O mesmo acontece com a sala três que tem uma sobreposição idêntica.

Exemplo nº2

Esta restrição valida se existem duas aulas a serem leccionadas pelo mesmo docente em simultâneo.

Não podem existir duas ou mais aulas com o mesmo docente ao mesmo tempo ou a iniciar enquanto o docente ainda se encontrar ocupado.

```
context OcorrenciaSemanal
  inv rh4:
    OcorrenciaSemanal.allInstances()->forAll(e1, e2 |
    (e1<>e2 and e1.ocorDocente=e2.ocorDocente) implies
     (
    e1.dia_semana_aulaDeUC<>e2.dia_semana_aulaDeUC
    or
    (((e1.hora_inicio_aulaDeUC + e1.duracao_aulaDeUC) <= e2.hora_inicio_aulaDeUC or
    (e2.hora_inicio_aulaDeUC + e2.duracao_aulaDeUC) <= e1.hora_inicio_aulaDeUC)) and
        e1.hora_inicio_aulaDeUC<>e2.hora_inicio_aulaDeUC))
```

Figura 58 - Restrição representada em OCL (exemplo 2)

```
CREATE OR REPLACE VIEW rh4 AS
(SELECT * FROM OV_OcorrenciaSemanal AS self
WHERE NOT (NOT EXISTS((SELECT temp1.PK_OcorrenciaSemanal
FROM OV_OcorrenciaSemanal AS temp1
INNER JOIN (SELECT * FROM OV_OcorrenciaSemanal) AS temp2
ON NOT((NOT (NOT (temp1.PK_OcorrenciaSemanal = temp2.PK_OcorrenciaSemanal)
AND (temp1.FK_ocorDocente = temp2.FK_ocorDocente))
OR (NOT (temp1.dia_semana_aulaDeUC = temp2.dia_semana_aulaDeUC)
OR ((((temp1.hora_inicio_aulaDeUC + temp1.duracao_aulaDeUC) <= temp2.hora_inicio_aulaDeUC)
OR ((temp2.hora_inicio_aulaDeUC + temp2.duracao_aulaDeUC) <= temp1.hora_inicio_aulaDeUC))
AND NOT (temp1.hora_inicio_aulaDeUC = temp2.hora_inicio_aulaDeUC)))))))));
```

Figura 59 – Código SQL gerado pelo DOT (exemplo2)

PK_OcorrenciaS...	FK_UCTiOcor	FK_ocorDocente	FK_ocorSala	cod_ocorrencia...	dia_semana_aul...	duracao_aulaD...	hora_inicio_aula...
1	1	1	1	1	Seg	1	10
2	1	4	2	2	Qui	1	10
3	1	3	2	3	Ter	2	9
4	1	4	3	4	Qui	2	9
5	1	1	3	5	Qui	2	15
6	2	2	4	6	Sab	2	9
7	3	3	5	7	Sab	1	10
8	3	3	1	8	Sab	2	9

1

2

Figura 60 – Todos os dados referentes aos horários (exemplo 2)

	PK_OcorrenciaSemanal	dia_semana_aulaDeUC	hora_inicio_aulaDeUC	duracao_aulaDeUC	FK_ocorDocente
1	2	Qui	10	1	4
2	4	Qui	9	2	4
3	7	Sab	10	1	3
4	8	Sab	9	2	3

Figura 61 – Detecção de dados incorrectos (exemplo 2)

Executado o código gerado pela ferramenta DOT na base de dados, consegue-se detectar que existem dois docentes que têm duas aulas sobrepostas. O docente com o código três (FK_ocorDocente) tem uma aula que inicia às dez horas com duração de uma hora e outra que inicia às nove horas com duas horas de duração, as aulas encontram-se sobrepostas entre as dez horas e as onze horas. O mesmo acontece com o docente com o código quarto que tem uma sobreposição idêntica.

Os testes efectuados mostram que os scripts SQL, gerados pelas restrições OCL, detectaram todas as situações que violavam as regras pretendidas. Dado que os comandos SQL são uma representação fiel das restrições OCL, podemos concluir que estas mostraram-se adequadas.

4.7.2. Problema detectado na conversão

A versão utilizada para efectuar os testes descritos nesta tese é uma versão ainda em desenvolvimento, tendo sido disponibilizada apenas para realizar testes no âmbito deste trabalho.

Durante os testes foram encontrados alguns erros, como por exemplo:

- não é possível definir chaves primárias do tipo Integer, quando é gerado o ficheiro que contém os scripts para a construção da base de dados, é gerada automaticamente uma chave primária do tipo Varchar(255);

- Não gera os ficheiros se existirem atributos sem tipo.
- Quando gera o código SQL (Standard(SQL)), gera com alguns pequenos erros de sintaxe.

5. Conclusões

5.1. OCL

Neste trabalho, a principal contribuição foi a de investigar a utilização de OCL como uma forma robusta de validar restrições em modelos.

Embora a OCL tenha sido definida com o objectivo de ser uma linguagem de uso mais fácil, se comparada às linguagens formais tradicionais (Warmer & Kleppe, 2003), as especificações produzidas com a OCL podem apresentar problemas de legibilidade e preservação, em função da presença de construções inadequadas, seja nas expressões OCL, seja no modelo subjacente.

Existem várias ferramentas que oferecem recursos para a avaliação sintáctica e para a verificação de tipos numa especificação OCL, o suporte automatizado para análises ainda é bastante limitado.

Para efectuar os testes, foi utilizada a ferramenta DOT (Dresden OCL Toolkit). Este compilador é capaz de realizar a análise sintáctica (*parsing*) de expressões OCL, verificação de tipos e avaliação de restrições presentes em modelos comuns. Também é capaz de gerar, automaticamente, o código Java e SQL a partir das expressões OCL e do modelo UML fornecidos.

Existiram várias dificuldades na utilização desta ferramenta, como por exemplo:

- A versão existente para SQL disponibilizada na página oficial é muito antiga e desactualizada, não funciona nas últimas versões do Eclipse. Para ser possível efectuar os testes, e só após vários contactos, foi disponibilizada pela Universidade Dresden, uma versão ainda em desenvolvimento.
- Existe pouca documentação sobre a ferramenta (versão SQL).

Os testes realizados tiveram apenas em consideração uma amostra das restrições, por este facto, as conclusões não poderão ser generalizadas para qualquer problema.

Tal como referido no capítulo 4.6, os testes efectuados mostram que os scripts SQL, gerados pelas restrições OCL, detectaram todas as situações que violavam as regras pretendidas. Dado que os comandos SQL são uma representação fiel das restrições OCL, podemos concluir que estas se mostraram adequadas.

Baseado nos estudos realizados neste trabalho, apresentamos algumas conclusões sobre o uso de OCL durante o processo de desenvolvimento de software:

- OCL ainda não é suficientemente madura para uso na indústria. A linguagem OCL possui uma grande capacidade de expressão, mas ainda não parece estar

madura o suficiente para ser usada na sua totalidade. Existem poucas ferramentas e pouca documentação nesta área e a documentação que existe nem sempre representa das restrições da mesma forma.

- OCL é difícil de compreender. A curva de aprendizagem da linguagem não é suave. Havendo muitas dificuldades durante a representação das restrições.

O objectivo inicial desta tese era criar uma aplicação para receber as restrições utilizadas na elaboração dos horários e no OCL para que ambas as aplicações partilhassem a mesma fonte de informação (Figura 62), mas não foi possível. Neste momento existe uma aplicação que gera os horários e outra que gera as restrições, não havendo qualquer ligação entre as duas.

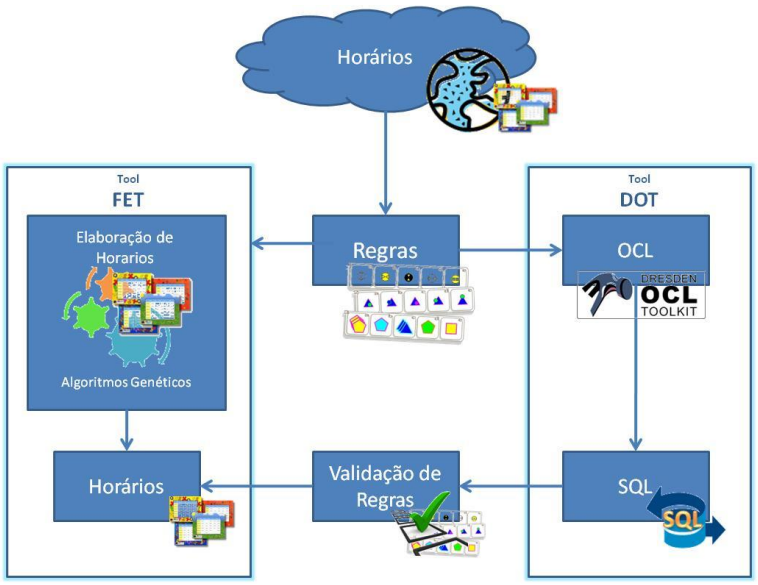


Figura 62 - Modelo pretendido da aplicação

Como trabalho futuro pretende-se adaptar a implementação dos algoritmos genéricos de modo a que este interprete as expressões OCL.

5.2. Algoritmos Genéticos

O modelo desenvolvido neste trabalho permite concluir que a utilização de algoritmos genéticos representa uma alternativa satisfatória para a solução do problema de elaboração de horários. O problema, por si só bastante complexo, foi abordado considerando-se restrições de recursos e docentes.

A representação utilizada, com uma estrutura primária e uma secundária no cromossoma, permite uma maior variação dos indivíduos.

Isto permite que uma maior combinação de horários seja encontrada numa

quantidade menor de gerações. A estruturação do cromossoma permite ainda uma redução do tamanho do mesmo, agilizando o processamento.

O modelo apresentado não tem a pretensão de ser o melhor para qualquer instituição. Naturalmente, os critérios de alocação de aula das unidades curriculares mudam de instituição para instituição. No entanto, um aspecto relevante do modelo é o de que a sua estrutura é flexível a qualquer número de cursos, semestre curricular, unidades curriculares, tipos de unidades curriculares e tipos de recursos.

Bibliografia

Abela, J., & Abramson, D. (1991). *A Parallel Genetic Algorithm For Solving The School Timetabling Problem.*

Abranson, D. (1991). Constructing School Timetables Using Simulated Annealing: Sequential and Parallel Algorithms. *Management Science* , 98-113.

Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., et al. (2005). *The KeY tool - integrating object oriented design and formal verification.*

Akehurst, D. H., & Patrascoiu, O. (2004). OCL 2.0 - implementing the standard for multiple metamodels.

Akkoyunlu, E. A. (1973). A linear algorithm for computing the optimum university. *The Computer Journal* , 347-350.

Algoritmos Genéticos. (s.d.). Obtido em 05 de 2010, de http://www.icmc.usp.br/~andre/research/genetic/

Almeida, V. C. (2006). *Uso da linguagem OCL no contexto de Diagramas de Classe da UML e programas em Java.* Belo Horizonte - Brasil.

Andrade, W. L., Barbosa, D. L., Machado, P. D., & Figueiredo, J. C. (2004). Um método automático para verificação funcional de componentes. *4ºWorkshop de Desenvolvimento Baseado em Compomentes.* Paraíba - Brasil.

Appleby, J. S., Blake, D. V., & Newman, E. A. (1961). Techniques for Producting School Timetables on a Computer and Their Application to Other Scheduling Problems. *The Computer Journal* , 237-245.

Baar, T. (2005). Non-deterministic Constructs in OCL – What Does any() Mean. *International SDL forum No12* (pp. 32-46). Grimstad, Noruega: Springer, Berlin, Allemagne.

Barbosa, D., Andrade, W., Machado, P., & Figueiredo, J. (2004). SPACES - uma ferramenta para teste funcional de componentes. *XI Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software*, (pp. 55–60). Brasília, Brasil.

Barraclough, E. D. (1965). The application of a digital computer to the construction of timetables. *The Computer Journal* , 136-146.

Bauerdick, H., Gogolla, M., & Gutsche, F. (2004). Detecting OCL traps in the UML 2.0 superstructure: An experience report. Alemanha: Springer Berlin- Heidelberg.

Beckert, B., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Schmitt, P. H., et al. *The Key System: Integrating Object-Oriented Design and Formal Methods.*

Berghuis, J., van der Heiden, A. J., & Bakker, R. (2005). The preparation of school time tables by electronic computer. *BIT Numerical Mathematics* , 106-114.

Bohling, J., Gogolla, M., & Richters, M. (2003). Validation of UML and OCL Models by Automatic Snapshot Generation. In Grady Booch, Perdita Stevens, and Jonathan Whittle, editors. *Proceedings of the 6th International Conference on Unified Modeling Language* . San Francisco.

Bohling, J., Richters, M., & Gogolla, M. (2005). Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling* , 386-398.

Boufflet, J. P., & Nègre, S. (2006). *Three methods used to solve an examination timetable problem .*

Bregalda, P. F., Oliveira, A. A., & Bornstein, C. T. (1988). *Introdução à Programação Linear.* Rio de Janeiro.

Briand, L. C., Dzidek, W., & Labiche, Y. (2004). Using aspect-oriented programming to instrument OCL contracts in Java.

Broder, S. (1964). Final Examination Scheduling. *Comm. A. C. M*, (pp. 494 - 498).

Burke, E. K., & Newall, J. P. (1999). A multistage evolutionary algorithm for the timetable problem. *Asia-Pacific Journal of Operational Research, v. 3, n. 1* , p. 63 – 74.

Caldeira, J. C., & Rosa, A. C. (1997). *School Timetabling using Genetic Search.* 115-122.

Carvalho, A. C., Braga, A. P., & Ludermir, T. B. (2002). Computação Evolutiva. In S. O. Rezende, *Sistemas Inteligentes - Fundamentos e Aplicações* (pp. 225-248). Editora Manole.

Cole, A. J. (1964). The preparation of examination time-tables using a small-store computer. *Comput. Journal* , 117-121.

Colmerauer, A. (1973). *Sur Les bases Théoriques de Prolog.* Paris.

Costa, E. O., & Bruna, M. D. (2003). Resolução de "Timetabling" utilizando Evolução Cooperativa. 20.

Cotta, C., & Moscato, P. (2003). An Introduction to Memetic Algorithms. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial.* , 131–148.

Davis, L. D. (1991). Handbook of Genetic Algorithms.

Demuth, B., & Hussmann, H. (1999). *Using UML/OCL Constraints for Relational Database Design .* Alemanha: Springer Berlin / Heidelberg.

Devos, F., & Steegmans, E. (2005). *Specifying business rules in object-oriented analysis.* Software and Systems Modeling.

Dresden OCL Toolkit. (s.d.). Obtido em 11 de 01 de 2010, de http://dresden-ocl.sourceforge.net/

Dzidek, W., Briand, L., & Labiche, Y. (s.d.). Lessons learned from developing a dynamic OCL constraint enforcement tool for Java.

Elliman, D. G., Burke, E., & Weare, R. F. (1995). The automation of the timetabling process in higher education. *Journal Educational Technol Systems* , 257-266.

Fang, H. (1994). *Genetic Algorithms in Timetabling and Scheduling.*

Faria, João Pascoal. (11 de 10 de 2004). *Laboratório de Engenharia de Software - Ano lectivo de 2004/2005, 1º Semestre*. Obtido em 2010 de 06 de 11, de FEUP: http://paginas.fe.up.pt/~jpf/teach/LES0405/

Feldman, R., & Golumbic, M. C. (2005). Interactive scheduling as a constraint satisfiability problem . *Annals of Mathematics and Artificial Intelligence* , 49-73.

Feo, T. A., & Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization* , 109-133.

Fernandes, C., Caldeira, J., & Rosa, A. (2002). *Infected Genes Evolutionary Algorithm for School Timetabling.* Evolutionary Systems and Biomedical Engineering Lab.

FET. (s.d.). *FET Free Timetabling Software*. Obtido em 11 de 2010, de http://www.lalescu.ro/liviu/fet/

Fikes, R. E. (1968). *A heuristic program for solving problems stated as non-deterministic procedures.* PhD thesis, Carnegie Mellon University.

Freuder, E. C. (1997). In Pursuit of the Holy Grail. *An International Journal* , 57-61.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

Giese, M., & Heldal, R. (2004). From Informal to Formal Specifications in UML. 197-211.

Gogolla, M., & Richters, M. (2003). Aspect-Oriented Monitoring of UML and OCL Constraints. Alemanha.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gotlieb, C. (1962). The Construction of Class-Teacher Timetables. *Proceeding of the IFIP Congress*, (pp. 73-77).

Gupta, G., & Akhter, S. F. (2000). *Knowledgesheet: A Graphical Spreadsheet Interface for Interactively Developing a Class of Constraint Programs.*

Hahnle, R., Giese, M., & Daniel, L. (2004). Rule-Based Simplification of OCL Constraints. *OCL and Model Driven Engineering, UML 2004 workshop*, (pp. 84-98). Lisboa, Portugal.

Hähnle, R., Johannisson, K., & Ranta, A. (2002). An Authoring Tool for Informal and Formal Requirements Specifications.

Hansen, P., & Mladenović, N. (2006). Variable Neighborhood Search.Operations Research. 1097–1100.

Haralick, R. M., & Elliott, G. L. (1980). *Increasing tree search efficiency for constraint satisfaction problems.* Artificial Intelligence.

Heijer, E., & Adriaans, P. (1996). The Application of Genetic Algorithms in a Career Planning Environment: CAPTAINS. (pp. 343-361). International Journal of Human-Computer Interaction.

Henderson-Sellers, B. (2005). *UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts.* Springer Berlin / Heidelberg.

Hentenryck, P. V., & Saraswat, V. (1997). Constraint Programming: Strategic Directions. *An International Journal, Kluwer Academic Publishers* , 7-33.

Hertz, A. (1992). *Finding a feasible course schedule using Tabu Search.* Amsterdam.

Holland, J. H. (1975). *Adaptation in natural and artificial systems.* The University of Michigan Press, 1th. Edition.

Holland, J. H. (s.d.). *Biographical Sketches*. Obtido em 12 de Maio de 2010, de http://www.cs.oswego.edu/~blue/hx/courses/cogsci1/s2001/section05/subsection5/main.html

Hussmann, H., Demuth, B., & Finger, F. (2002). Modular architecture for a toolset supporting OCL.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development.* Addison Wesley Longman, Inc.

Jaffar, J., & Lassez, J. L. (1987). *Constraint Logic Programming.* Munich - Germany.

Johnson, D. (1990). Timetabling university examinations. *Journal of the Operational Research* , 39-47.

Júnior, O. d. (2000). *Otimização de horários em Instituições de Ensino Superior através de algoritmos genéticos.* Florianópolis - Brasil.

Karp, R. M. (1972). *Reducibility Among Combinatorial Problems.* New York.

Kirkpatrick, S., Gellat, D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science* , 671–680.

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison Wesley.

Laddad, R. (2003). AspectJ in Action - Practical Aspect-Oriented Programming. Manning Publications Co.

Larsson, D., & Giese, M. (2005). *Simplifying Transformations of OCL Constraints.*

Lavazza, L., & Barresi, G. (2005). Automated support for process-aware definition and execution of measurement plans. *In Proceedings of the 27th International Conference on Software Engineering (ICSE'05),*, (pp. 234 - 243). St. Louis, MO, USA.

Linington, P., & Akehurst, D. (2003). OCL 2.0: Implementing the standard. Technical Report TR-12-03.

Lobo, E. L. (2005). *Uma solução do problema de horário escolar via algoritmo genético paralelo.* Brasil.

Loecher, S., & Ocke, S. (2003). A metamodel-based OCL-compiler for UML and MOF.In Elsevier, editor, Proceedings of the 6th International Conference on the Unified Modelling. San Francisco.

Mak, J. K., Choy, C. S., & Lun, D. P. (2004). Precise Modeling of Design Patterns in UML. *In Proceedings of the 26th International Conference on Software Engineering (ICSE'04)* (pp. 252 - 261). DC, USA: IEEE Computer Society Washington.

Meyer, B. (1997). *Object-Oriented Software Construction, Second Edition.* Santa Barbara : Prentice Hall Professional Technical Reference.

Mladenovic, N., & Hansen, P. (1997). *Variable Neighborhood Search. Computers and Operations Research.*

Montanari, U. (1972). Networks of constraints: Fundamental properties and applications to picture processing. *Information Science* , 95-132.

Morgan, T. (2001). Business Rules and Information Systems: Aligning IT with Business Goals. Addison Wesley.

Moscato, P. (1989). *On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms.* California, USA.

Norman, M., & Moscato, P. (1992). *A 'memetic' approach for the traveling salesman.Implementation of a Computational Ecology for Combinatorial Optimization on*

Message-Passing Systems. Amsterdam.

(1998). *Object Constraint Language - OMG-UML V1.1.*

Ol'khovich, L., & Koznov, D. V. (2003). OCL-Based Automated Validation Method for UML Specifications. *Programming and Computer Software*, (pp. 323 - 327).

OMG. (s.d.). *OMG UML 2.0 superstructure specification*. Obtido em August de 2005, de http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf.

Paechter, B., A., C., Luchian, H., & Petriuc, M. (1994). Two solutions to the general timetable problem using evolutionarymethods.

Penker, M., & Eriksson, H.-E. (2000). *Business Modeling with UML: Business Patterns at Work.* John Wiley and Sons.

Puccini, A. L. (1981). *Introdução à Programação Linear.* Rio de Janeiro: Livros Técnicos e Científicos Editora S.A.

Qu, R. (2002). *Case-Based Reasoning for course timetabling problems.* University of Nottingham.

Radaelli, J. L., & Terra, I. P. (2007). Utilização dos Métodos de Otimização em problemas de timetabling.

Richters, M. (2002). A Precise Approach to Validating UML Models and OCL Constraints. Universität Bremen, Logos Verlag, Berlin.

Romero, B. P. (1982). *Examination Scheduling in a Large Engineering School: A Computer-Assisted Participative...* Madrid.

Schmidt, G., & Ströhlein, T. (1980). Timetable construction – an annotated bibliography. *Computer Journal* , 307-316.

Sendall, S., & Küster, J. (2004). Taming Model Round-Trip Engineering. *Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications).* Vancouver,Canada.

Silva, G. Z., Almeida, V. T., Souza, J. M., Sulaiman, A., & Neto, F. G. (2001). *ATENAS: Um Sistema Gerenciador de Regras de Negócio.* Rio de Janeiro, Brasil: Apresentado e Publicado na Seção Técnica de Ferramentas do XV Simpósio Brasileiro de Engenharia de Software.

Souza, M. J., Costa, F. P., & Guimarães, I. F. (2002). *Um Algoritmo Evolutivo Híbrido para o Problema de Programação de Horários em Escolas.* Rio de Janeiro, Brasil.

Steele, G. L. (1980). *The Definition and Implementation of a Computer Programming*

Language Based on Constraints. PhD thesis, MIT.

Straeten, R. V., & Verheecke, B. (2002). Specifying and implementing the operational use of constraints in object-oriented applications. Australia.

Sutherland, I. E. (1963). *Sketchpad: A man-machine graphical communication system.* In Proceedings of the IFIP Spring Joint Computer Conference.

Sywerda, G. (1989). Uniform crossover in genetic algorithms. pp. 2 - 9.

Trevelin, L. C. (1983). *Sobre a proposta de um modelo linear para solução do problema de geração de horários de aulas de uma universidade.* São Carlos.

USE. (28 de 10 de 2009). Obtido em 10 de 01 de 2010, de USE: http://www.db.informatik.uni-bremen.de/projects/USE/

Varella, A., Pereira, V., Held, V. V., Zimbrão, G., & Silva, J. C. (2004). *Uma interface em linguagem natural para a verificação de regras de negócio em bases de dados.* Brasil: In 4o. Simpósio de Desenvolvimento e Manutenção de Software da Marinha (SDMS'04).

Verheecke, B. (2001). From declarative constraints in conceptual models to explicit constraint classes in implementation models.

Waltz, D. L. (1975). *Understanding line drawings of scenes with shadows.* McGraw-Hill.

Wand, M. (2003). Understanding aspects (extended abstract).

Warmer, J. B., & Kleppe, A. G. (2003). *The object constraint language: getting your models ready for MDA.* The Addison-Wesley Object Technology Series. Addison-Wesley.

Welsh, D. J., & Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems . *Computer Journal* , 85-86.

White, G. M., & Wong, S. K. (1988). Interactive timetabling in universitie. *Computers & Education* , 521 - 529.

Yunes, T. H. (2000). *Problemas de escalonamento no transporte coletivo : programação por restrições e outras tecnicas .* Campinas, Brasil.

Zdun, U., & Avgeriou, P. (2005). Modeling architectural patterns using architectural. *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (pp. 133 - 146). San Diego, California, USA.

Zschaler, S., & Hussmann, H. (2004). The Object Constraint Language for UML 2.0 – Overview and Assessment. *Digital Journal of CEPIS (Council of European Professional Informatics Societies* , 25–28.

Anexo A

Instalação da ferramenta Eclipse

Fazer *download* no seguinte *link*:

http://www.eclipse.org/downloads/

1. Instalação

Eclipse Modeling Tools

http://www.bluage.com/?cID=eclipse_free_download_eclipse-modeling-galileo-incubation-win32.zip

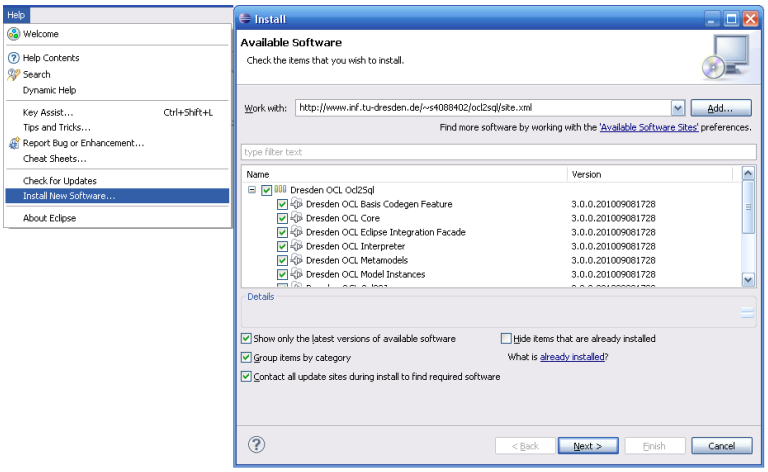
Descompacte o ficheiro (coloque num caminho que não tenha espaços), não necessita de instalação adicional.

2. Iniciar o Eclipse

Para iniciar o Eclipse, duplo clique sobre o *eclipse.exe* que se encontra no arquivo de instalação.

3. Instalação do ocl22sql

Para instalar o ocl22sql deve seleccionar a opção *Help* / *Install New Software*. Na janela de instalação em *Work with* escreva o seguinte *link* http://www.inf.tu-dresden.de/~s4088402/ocl2sql/site.xml (Versão beta, disponibilizada apenas para testes).



Anexo B

Validação das restrições e geração dos ficheiros

Criar um novo projecto Java em *File* / *New* / *Java Project*. Colocar o nome do projecto em *Project name.*

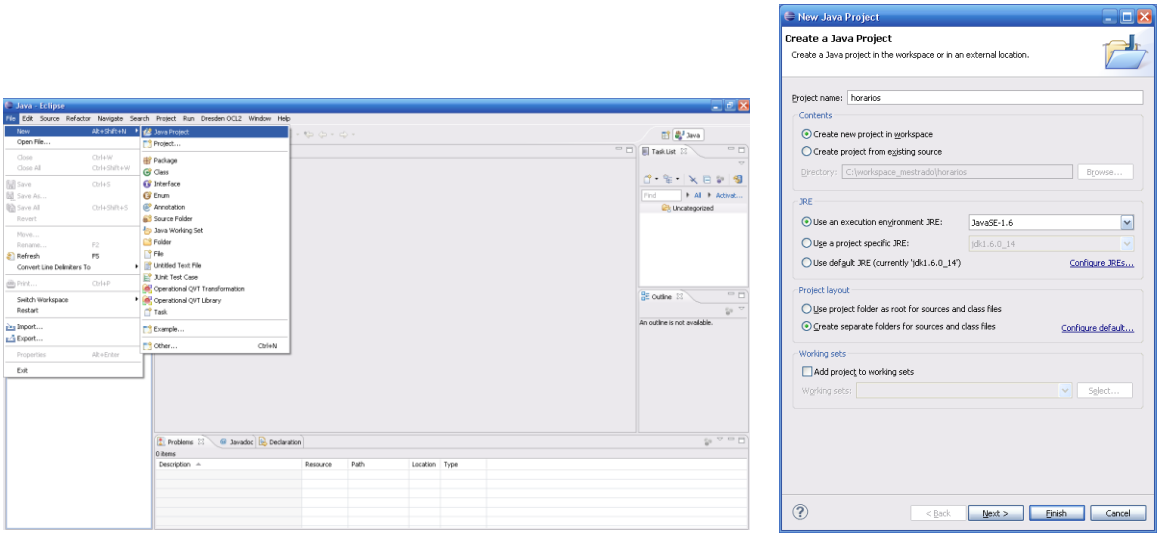


Figura 63 - Criação um novo projecto Java

Criar um *package* dentro da pasta src em *New* / *Package* (clicar no botão do lado direito do rato em cima da pasta *src*).

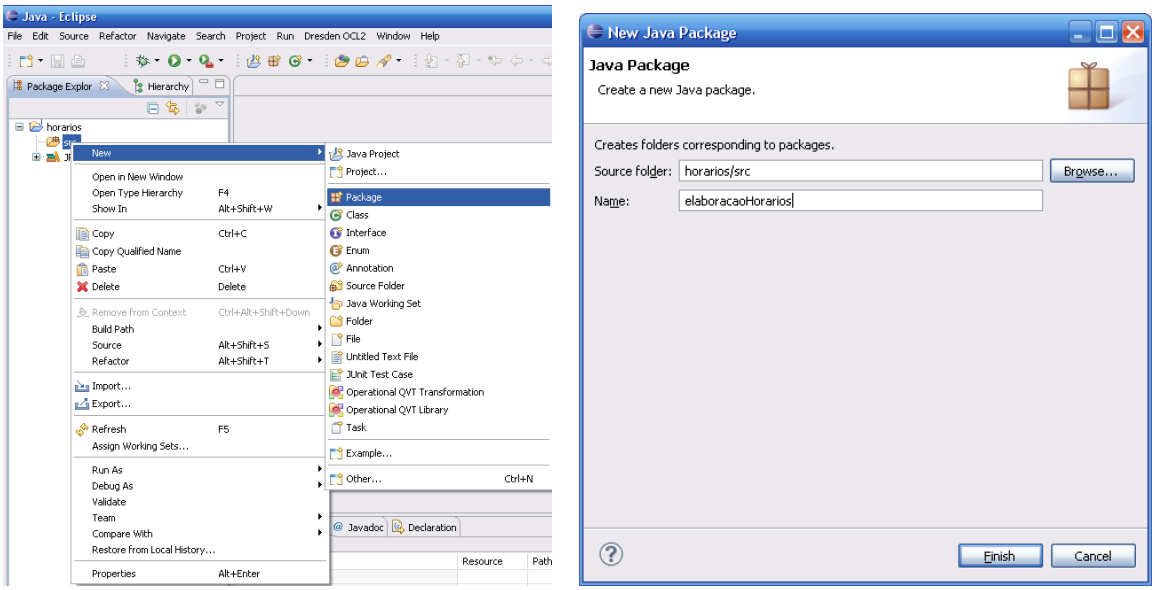


Figura 64 - Criação um Package

Criar um diagrama de classes dentro do *package* criado anteriormente, em *New* /

Other / UML 2.1 Diagrams / Class Diagram

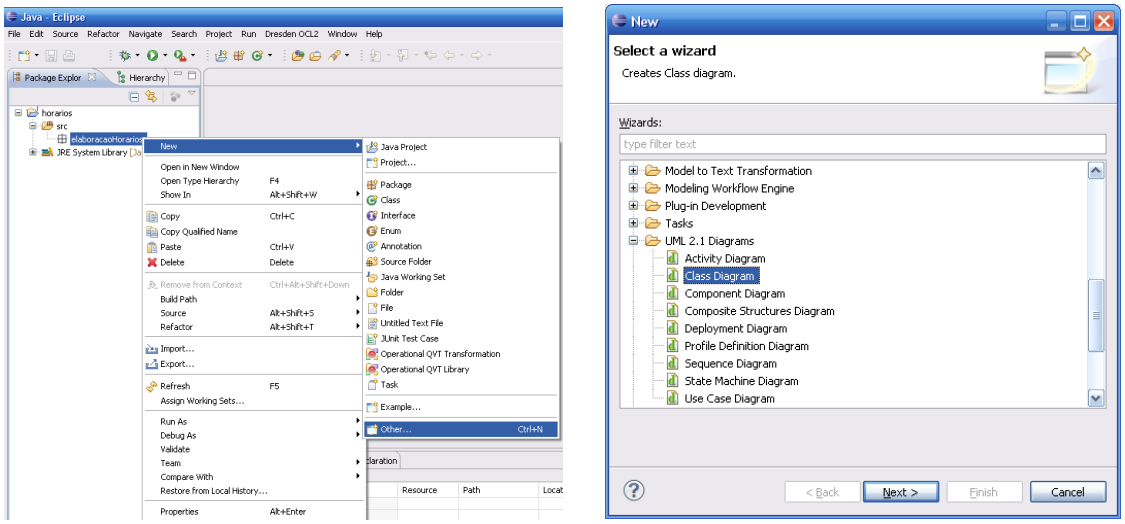
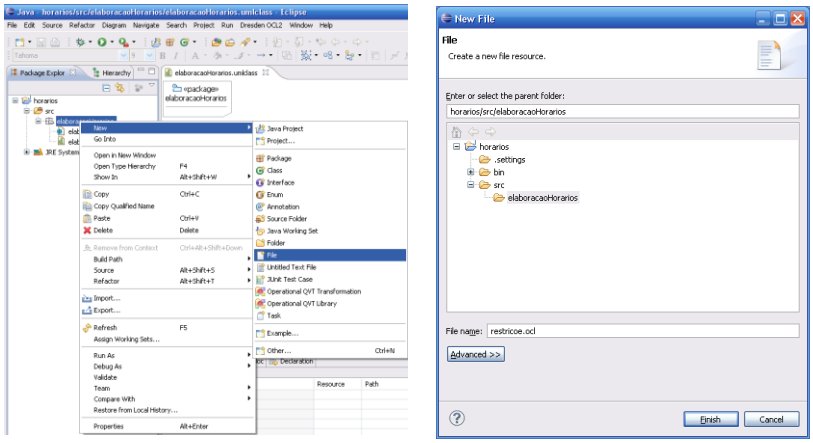


Figura 65 - Criação um diagrama de classes

Criar um ficheiro com extensão OCL (nomeExemplo.ocl) em *New* / *File*. Depois de criar ficheiro, escrever o seguinte texto no ficheiro:

package elaboracaoHorarios -- **Deve de ter o mesmo nome que o ficheiro do modelo**.

Endpackage



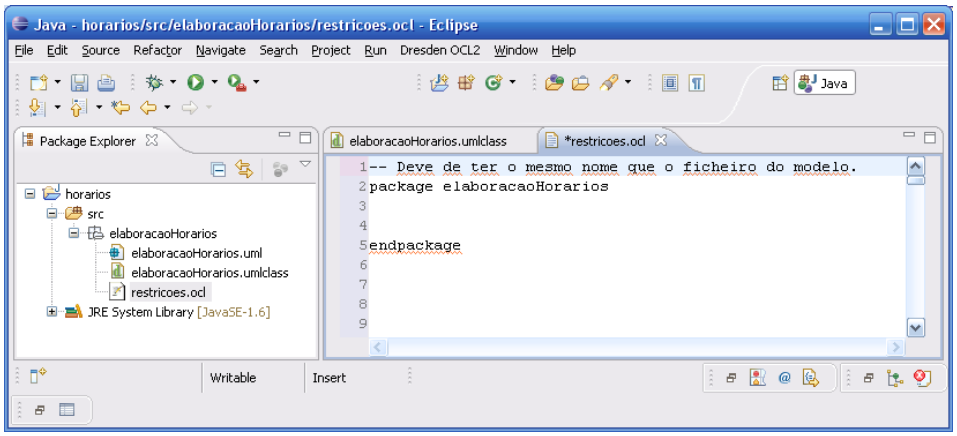


Figura 66 - Criação do ficheiro OCL

Neste momento já temos os dois ficheiros necessários para iniciar a conversão.

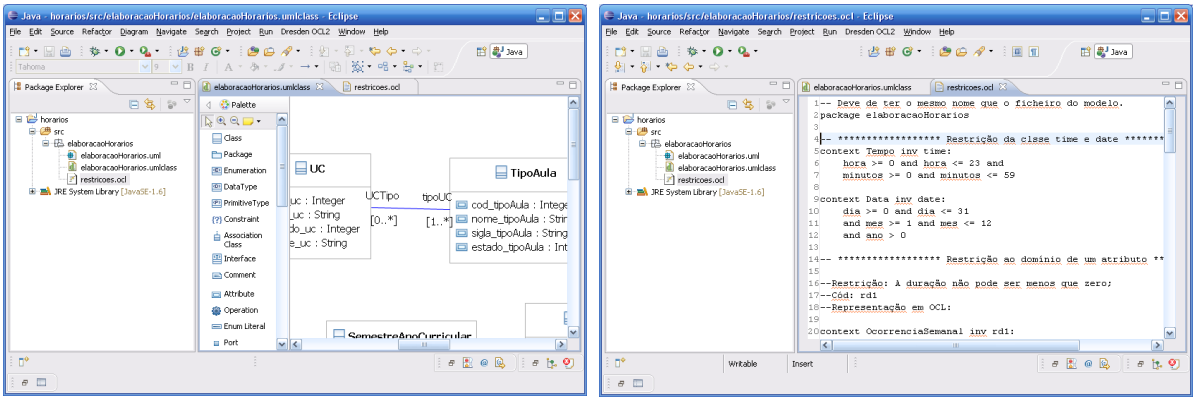


Figura 67 - Ficheiros necessários para a conversão

Para carregar o modelo UML selecciona-se a opção *Dresden OCL2* / *Load Model* / *UML2(eclipseMDT)* / *Browse Workspace*. Para que as restrições seja adicionadas ao modelo é apenas necessário abrir o ficheiro com a extensão *.ocl*.

Aplicação da OCL à especificação do problema de elaboração de horários

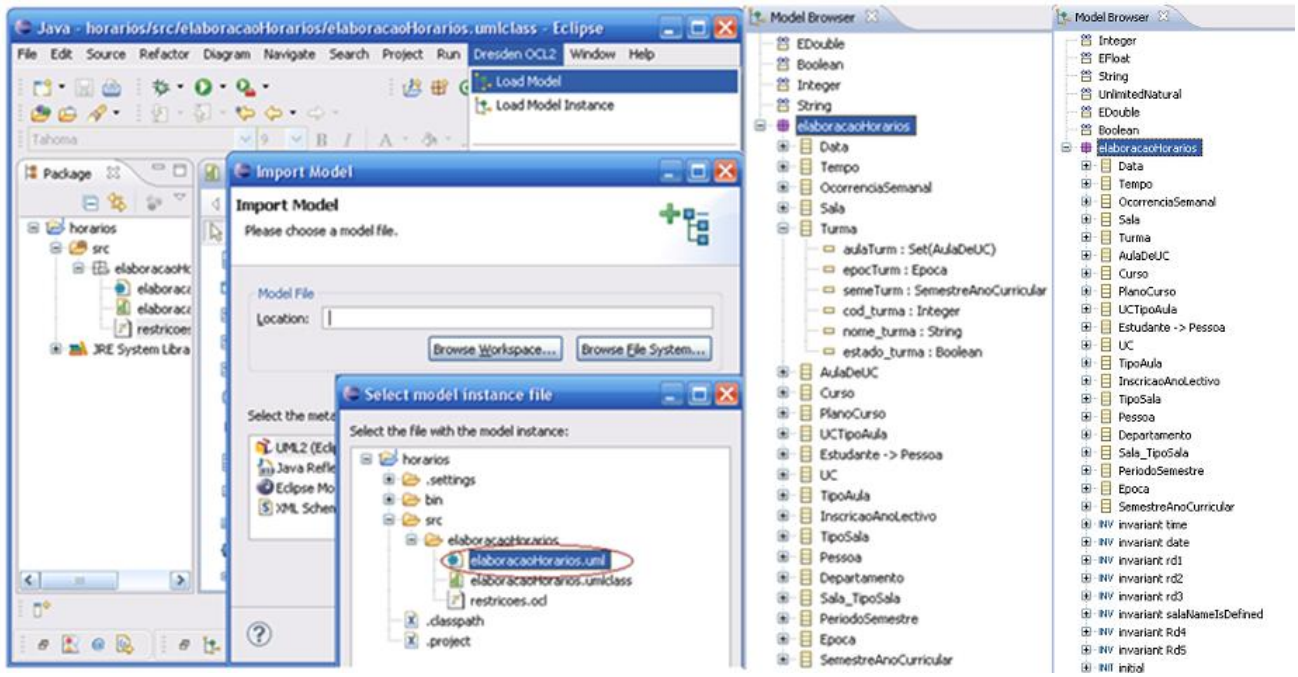


Figura 68 - Selecção do modelo

Para gerar o código SQL selecciona-se a opção *Dresden OCL2* / *Generate SQL Code*.

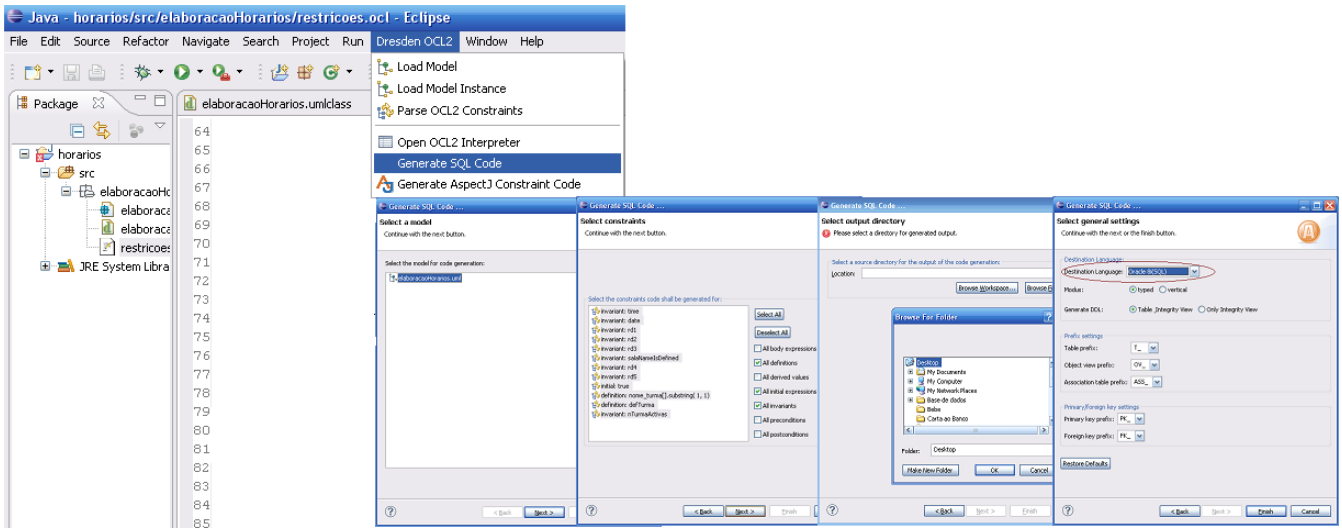


Figura 69 – Geração do código SQL

A referida ferramenta gera dois ficheiros, um que contém os *scripts* para a criação da base de dados e outro com as *views* resultantes das restrições OCL.

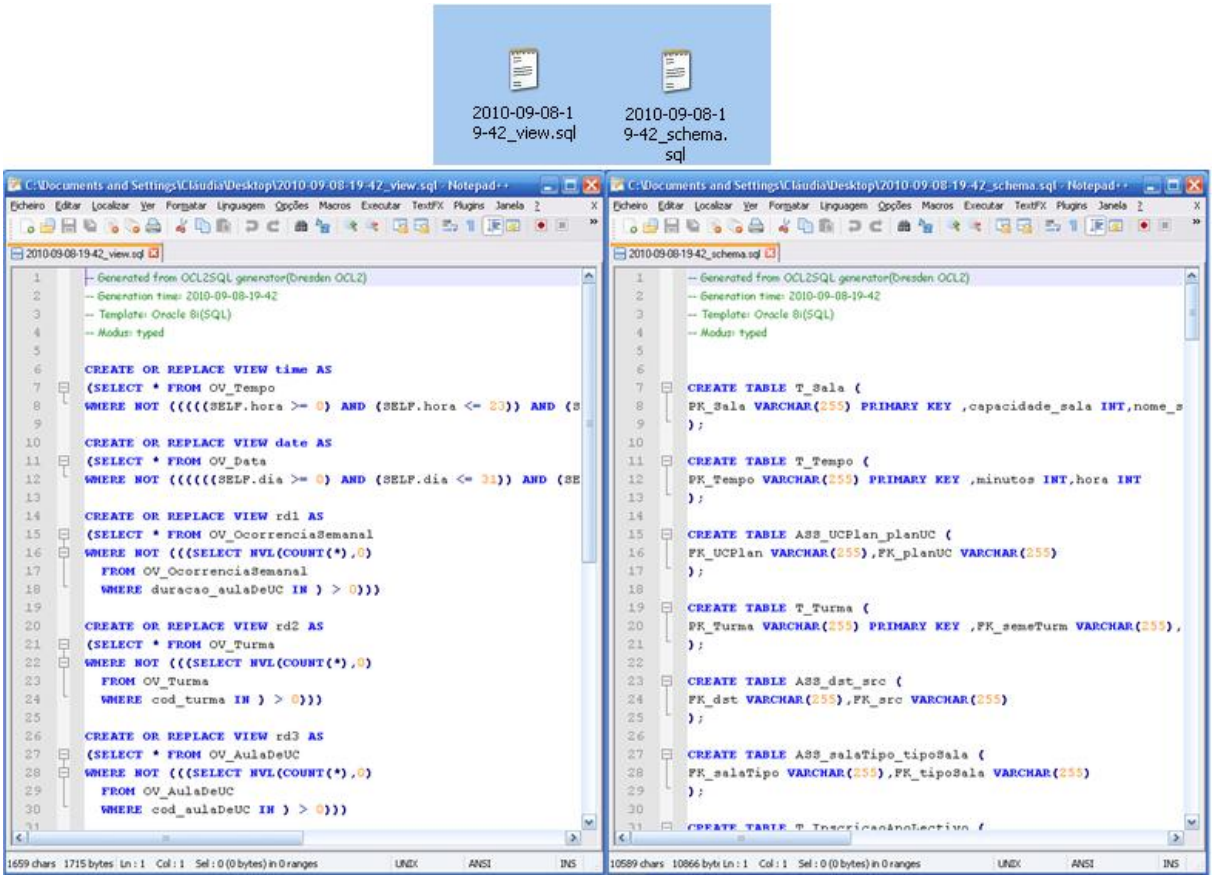


Figura 70 - Resultado da ferramenta OCL22SQL