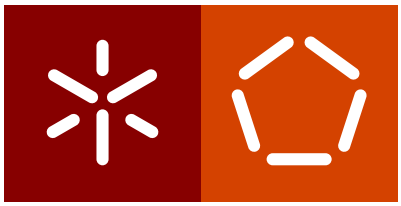
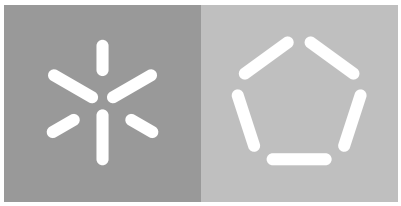


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mariana Almeida Brandão Capelo

Especificação e Validação de Processos ETL em Alloy

Novembro de 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mariana Almeida Brandão Capelo

Especificação e Validação de Processos ETL em Alloy

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Doutor Orlando Manuel de Oliveira Belo

Novembro de 2018

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu orientador, Professor Doutor Orlando Manuel de Oliveira Belo, por me ter acompanhado neste trabalho de dissertação, mostrando-se sempre disponível e pronto a ajudar. Sem o seu suporte este trabalho não seria possível. Gostaria, também, de agradecer ao Professor Bruno Oliveira, pela ajuda e acompanhamento na fase inicial deste projeto.

Ao Henrique, que esteve sempre ao meu lado, com as palavras e conselhos certos. Obrigada por todo o apoio, por toda a motivação e ajuda.

À minha família, que me apoiou sempre, incondicionalmente. Que me apoiou na teoria e na prática, das melhores formas possíveis. Obrigada, Ana, pela companhia neste processo e por teres sempre um ouvido disponível. Obrigada a todos, pelos bons exemplos que me deram ao longo dos anos.

Resumo

Especificação e Validação de Processos ETL em Alloy

O desenvolvimento de processos ETL é uma tarefa dispendiosa e complexa. Não admira, pois, o cuidado que os seus implementadores têm, em particular, durante as suas fases de planeamento e análise. Muito trabalho tem sido desenvolvido em prol do estabelecimento de novos e melhores métodos e técnicas de modelação conceptual e lógica destes processos. Todavia, ainda ocorrem inúmeros problemas durante as primeiras fases de execução dos processos de ETL, muitos deles provocados por erros de análise, de desenvolvimento, ou de simples esquecimento. Como tal, é vital que antes da entrada destes processos em produção, eles sejam submetidos a algum tipo de mecanismo que permita validá-los e comprovar a sua correção, relativamente àquilo que se espera que eles realizem. A utilização da linguagem Alloy na especificação e validação de processos ETL oferece esse tipo de validação. Neste trabalho de dissertação, suportado por um caso de estudo específico, Alloy é estudada, utilizada e avaliada quanto à sua aplicação na especificação formal e validação de processos ETL.

Palavras-chave: Data Warehousing, Sistemas de ETL, Especificação e Verificação de Sistemas de ETL, Alloy

Abstract

Specification and Validation of ETL Processes in Alloy

The development of ETL processes is an expensive and complex task, hence the attention and care given by its developers, especially during the planning and analysis stages. A lot of effort has been put into establishing new and improved methods and techniques for ETL processes logical and conceptual modelling. However, even with the given attention, several problems occur during the first stages of the execution of ETL processes, a lot of them caused by analysis errors, development errors, or simply due to forgetfulness. Thus, it is vital that, before these processes are deployed into production, they are submitted to some mechanism which enables their validation and offers proofs about their correctness. The use of Alloy language for the specification and validation of ETL processes provides this kind of validation. In this dissertation work, supported by a specific study case, the Alloy language is studied, applied and evaluated regarding its application in the formal specification and validation of ETL processes.

Keywords: Data Warehousing, ETL Systems, Specification and Validation of ETL, Alloy.

Índice

Índice de Figuras

Capítulo 1.

Introdução

1.1 Contextualização

Num contexto organizacional, o acesso à informação é um elemento chave para a compreensão do estado do negócio e para o suporte de um processo de tomada de decisão consciente. Os especialistas podem tomar decisões conscientes e fundamentadas em comportamentos passados ou previsões de tendências futuras, sendo que o impacto destas decisões frequentemente desempenha um papel muito importante no futuro de uma organização. Os sistemas de suporte à decisão (SSD) surgem, assim, como ferramentas essenciais para a gestão eficiente de uma empresa. Estes permitem o acesso a informação legível, em tempo útil, possibilitando a extração de conhecimento que fundamenta as decisões tomadas pelos seus utilizadores finais - agentes de tomada de decisão. Para isso, os SSD colecionam grandes volumes de dados e reduzem-nos a uma forma que pode ser usada para analisar padrões e comportamentos organizacionais, armazenando-os num repositório denominado *data warehouse*.

Usualmente, os *data warehouses* encontram-se na base dos SSD, e são especializados em centralizar dados provenientes de diferentes fontes, armazenando-os temporalmente e de acordo com as

perspetivas de análise envolvidas no processo de decisão. Contudo, a construção e desenvolvimento de um sistema de *data warehousing* é uma tarefa complexa, na qual o processo de ETL - extração, transformação, carregamento - surge como um dos mais críticos.

O processo de ETL é um processo complexo cujo objetivo final é alimentar um sistema de *data warehousing* com dados com qualidade, provenientes de diferentes fontes, frequentemente heterogéneas. Para que a qualidade dos dados que são processados seja garantida, os sistemas de ETL necessitam de mecanismos para tratamento de dados e recuperação de erros. Adicionalmente, o fluxo de execução de processos ETL apresenta, não raras vezes, as dependências existentes entre componentes, processamento de dados em *batch* e processos potencialmente paralelos.

Uma fraca implementação de um sistema de ETL terá consequências negativas, não só a nível do desenvolvimento e da manutenção do sistema, como na qualidade dos dados do próprio sistema de *data warehousing* (English, 1999). Apesar do peso que o desenvolvimento de processos ETL acarreta e da importância da sua correção num SSD, a comunidade apresenta ainda uma falha quanto a uma abordagem definitiva, simples e rigorosa, capaz de modelar e validar estes processos.

1.2 Motivação

Ao longo dos últimos anos foram apresentadas diversas propostas para suportar a modelação conceptual e lógica de processos ETL. Algumas destas propostas utilizam padrões, blocos de funcionalidades ou de serviços que atuam de forma autónoma, que são traduzidos num conjunto de componentes abstratos configurados de forma a permitir a sua instanciação para cenários aplicacionais específicos. Todavia, os sistemas de ETL apresentam um grau de especificidade muito elevado, acarretando requisitos de dados bastante complexos e rotinas de transformação elaboradas, cuja correção é de difícil validação.

A motivação deste trabalho de dissertação emergiu a partir do conhecimento adquirido com o trabalho de Oliveira, Belo, e Macedo (2016), no qual é apresentada uma proposta de uma abordagem especialmente orientada por padrões para a validação de processos ETL. Esta permite a implementação de processos ETL com um maior nível de abstração, sendo acompanhada por um

conjunto de especificações formais que lhes garante a sua robustez. Este trabalho foi um dos primeiros esforços na especificação formal de tarefas ETL utilizando Alloy, uma linguagem de especificação declarativa, no qual se apresenta uma especificação comportamental e estática de um padrão que suporta uma tarefa de ETL relacionada com o processamento de uma dimensão, com variação e com histórico – *Slowly Changing Dimension with History Maintenance* (SCD-H).

A Alloy (Jackson, 2012) é uma linguagem textual de especificação desenvolvida no MIT em 1997, que permite descrever estruturas complexas, bem como as suas restrições e comportamentos. Uma das suas principais características é a sua capacidade analítica, que permite validar as especificações declaradas, através da ferramenta Alloy Analyzer. A análise de especificações em Alloy é implementada através da descoberta de instâncias de um dado modelo ou contraexemplos de uma asserção relacionada com esse modelo, alertando apenas, em geral, para a existência de erros e não para a sua ausência. Contudo, a hipótese 'Small Scope' postula que a maior parte dos erros pode ser encontrada em modelos com um domínio de cardinalidade pequena. Desta forma, a linguagem Alloy tem vindo a ser usada com sucesso na modelação de problemas em diferentes áreas e de diferentes naturezas.

No contexto da modelação de processos ETL, a utilização da linguagem Alloy introduz um formalismo inovador perante as abordagens atualmente existentes, mantendo a flexibilidade necessária para lidar com comportamentos específicos de processos ETL. Adicionalmente, as especificações criadas são analisadas e validadas, oferecendo maior confiança quanto à sua correção - característica imprescindível para o sucesso de produtos de software complexos.

1.3 Objetivos

Este trabalho teve como principal objetivo estudar algumas das formas como poderíamos especificar e validar processos ETL - blocos de operações e as suas dependências - utilizando uma linguagem de especificação formal (Alloy). Inspirado pelos avanços registados nesta área de investigação, que evidenciam o potencial da utilização de uma linguagem formal para a modelação dos processos ETL, espera-se que a utilização de Alloy permita uma forma automática de análise e a procura de asserções falsas pela geração de contraexemplos. Com isto, as tarefas especificadas tornar-se-ão

mais simples de ler e utilizar e também de reutilizar para a produção de software com maior qualidade. De uma forma mais concreta, pretendemos estudar e avaliar a adequação da linguagem de especificação formal Alloy na definição de processos ETL, tendo em consideração:

- Adequação da linguagem para representar a natureza dinâmica de processos ETL (evolução de um estado para o próximo).

- Especificação de operações em *batch*.

- Adequação da utilização de asserções para garantia de qualidade de processos ETL.

- Facilidade de escrita da modelação estática do sistema.

1.4 Organização do documento

Além do presente capítulo, esta dissertação está organizada em mais seis capítulos, nomeadamente:

- Capítulo 2 – Trabalho Relacionado – no qual se explora o trabalho relacionado com o tema em questão e se apresentam as diversas abordagens à modelação conceptual e lógica de processos ETL. Adicionalmente, apresenta-se um breve estudo das diferentes linguagens e técnicas utilizadas na especificação formal de sistemas.

- Capítulo 3 – Especificação e Validação de Processos ETL em Alloy – dedicado a uma abordagem aprofundada da especificação de sistemas ETL em Alloy. Para tal, a filosofia e metodologia de modelação associadas à linguagem Alloy são estudadas e apresentam-se os primeiros exemplos de modelos em Alloy. Alguns dos componentes chave associados aos processos ETL são também explorados e é feita uma análise quanto à adequação da linguagem Alloy para a especificação de processos ETL.

- Capítulo 4 – O Caso de Estudo – este capítulo introduz o caso de estudo selecionado como base para a modelação do sistema ETL alvo. A apresentação do caso de estudo envolve a

sua contextualização, modelação dimensional e introdução das fontes de dados envolvidas. Por fim, as estruturas auxiliares ao processo de povoamento do caso de estudo (que compõem a área de retenção do sistema) e as tarefas de extração, transformação e carregamento de dados do processo de povoamento são apresentadas e modeladas com apoio a diagramas BPMN.

- Capítulo 5 – Especificação Formal do Sistema – neste capítulo são apresentadas as estratégias e abordagens definidas para a modelação do caso de estudo em Alloy, sendo apresentada em detalhe a modelação das estruturas de dados envolvidas no mesmo.

- Capítulo 6 – Especificação Formal do Processo ETL – a especificação das tarefas ETL em Alloy e do processo ETL enquanto um todo é apresentada, analisada e validada com base na modelação das estruturas de dados em Alloy do capítulo anterior e da modelação das tarefas do caso de estudo apresentadas no Capítulo 4.

- Capítulo 7 – Conclusões e Trabalho Futuro – neste último capítulo é feita uma análise crítica acerca do processo de modelação desenvolvido neste trabalho de dissertação e das decisões que foram sendo tomadas ao longo da sua execução. Complementarmente, tecem-se algumas considerações sobre potenciais linhas de trabalho futuro, com particular interesse na generalização da modelação desenvolvida.

Capítulo 2.

Trabalho Relacionado

2.1 Modelação de Processos ETL

A modelação de processos ETL tem sido alvo de estudo nas últimas décadas. A definição de uma metodologia ou notação padrão para a modelação de processos ETL aceite pela comunidade é um passo importante para a uniformização do próprio desenvolvimento de um sistema de *data warehousing*. A utilização de padrões é uma forma de tornar a implementação mais metódica e regularizada, evitando erros ou falhas conhecidas através da utilização de elementos comuns.

Uma padronização quanto aos processos ETL pode levar, ainda, à sua automação. Apesar de isto ser, implicitamente, realizado pelas ferramentas de ETL existentes no mercado, estas ferramentas são muitas vezes proprietárias, o que torna as metodologias, as notações ou o modo de funcionamento interno dessas ferramentas bastante específicos (ou mesmo desconhecidos) para os seus utilizadores.

Desde as abordagens propostas por Inmon (2005) e Kimball (2004; 1998) para o desenvolvimento de ETL, que são essencialmente *ad-hoc* e que demonstram alguma falta de rigor e formalismo, têm

surgido diferentes abordagens, que, de uma forma simples, tentam esquematizar os mecanismos inerentes aos processos ETL. A modelação com base em meta modelos e meta dados e a sua consequente instanciação foi o primeiro marco no processo da esquematização dos processos ETL, despoletando diferentes abordagens que tentaram utilizar linguagens de modelação como a UML ou a notação BPMN – na qual um processo de ETL é identificado como um processo de negócio –, enquanto outras abordagens, mais formais, apelaram à álgebra relacional ou linguagens de especificação formal para fazer esse mesmo trabalho.

Em Vassiliadis, Simitsis e Skiadopoulos (2002), é defendida uma arquitetura baseada em três níveis para suportar a modelação conceptual dos processos de ETL, nomeadamente: uma camada de meta modelos com construtores genéricos (por exemplo: atributo, relação, transformação), uma camada de *template* com construtores específicos de atividades ETL (por exemplo: tabela de factos, dimensão, atribuição de chaves de substituição) e, por fim, uma camada de esquema com instâncias das classes da camada de meta modelos e das subclasses contidas na camada de *template*. A arquitetura sugerida, apesar de conceptualmente bem dividida e de fácil compreensão, conduz ao desenvolvimento de diagramas bastante complexos e pouco legíveis, mesmo quando aplicada a casos de estudo simples. Além desta proposta, também no âmbito da modelação conceptual destes processos e de forma a padronizá-los, foi proposta uma extensão à notação UML (Trujillo & Luján-Mora, 2003). Nesta abordagem o processo de desenvolvimento de um *data warehouse* é estruturado com base num modelo integrado, que é constituído por um esquema de dados operacionais, um esquema conceptual do *data warehouse*, um esquema de armazenamento físico do *data warehouse* e um modelo de negócio – no qual são definidas as diferentes formas ou vistas de acesso ao *data warehouse* pelos utilizadores finais. Posteriormente, a este modelo são adicionados dois esquemas de mapeamento: processos ETL (mapeamento entre o esquema de dados operacionais e esquema conceptual do *data warehouse*) e processo de exportação (mapeamento entre o esquema conceptual do *data warehouse* e o seu esquema de armazenamento). Esta abordagem faz, também, uma distinção clara entre diferentes mecanismos de ETL, nomeadamente, agregação, conversão, filtro, incorreções, junções e carregamentos, entre outros.

Segundo El Akkaoui e Zimanyi (2009), os processos ETL devem ser modelados como processos de negócio. Para isso propuseram uma extensão à notação BPMN para que fosse possível lidar com as especificidades de certas tarefas ETL. A utilização da BPMN permite uma tradução dos modelos desenvolvidos para programas criados através da linguagem de programação BPEL. Posteriormente,

esta abordagem foi explorada por outros autores, que encontravam no mapeamento de modelos conceptuais para um conjunto de primitivas de execução grande valor. A partir dessa base, tais autores criaram modelos conceptuais específicos para diferentes processos ETL (Oliveira & Belo, 2013). Contudo, estes modelos não apresentavam uma componente lógica, que fosse capaz de reduzir a liberdade oferecida no momento de implementação das tarefas ETL.

Mais tarde, em Santos (2015) a especificação de processos ETL foi abordada utilizando operadores e árvores de álgebra relacional, adicionando, assim, uma componente formal à sua construção. Neste estudo foram especificados outros processos padrão ETL, como foi o caso das tarefas de captura de alteração de dados – *Changing Data Capture* (CDC) – e de garantia de qualidade de dados – *Data Quality Enforcement* (DQE) –, em particular em termos da decomposição, conformação e da conciliação de dados. Além disso, foram também estudados casos de padrões para suporte a tarefas que envolvem o processamento de dimensões com variação de dados de diferentes tipos e a atribuição de chaves de substituição – *Surrogate Key Pipelining* (SKP). Nestes casos, as tarefas de CDC usam principalmente o operador de subtração de álgebra relacional, enquanto que tarefas de DQE se baseiam no uso de funções de utilizador aplicadas aos atributos, ou tabelas auxiliares que mapeiam valores incorretos em valores corretos. Por sua vez, as tarefas de conciliação de dados foram implementadas utilizando tabelas de mapeamento com chaves de substituição através da utilização de operações de junção e subtração. Com base nas especificações criadas, o trabalho de Santos (2015) apresenta uma especificação completa de um sistema de ETL real, como caso de estudo. Contudo, esta abordagem falha na captação de aspetos intrínsecos dos processos em questão. Os operadores utilizados na especificação não permitem a modelação de comportamentos, como a manipulação de erros (como recomeçar o processo ou enviar mensagens de erro), uma vez que a álgebra relacional é orientada à manipulação de dados e não ao controlo de fluxos de operações. Assim, apesar do formalismo que acarreta, esta abordagem apresenta falhas quanto à sua capacidade de esquematização do processo ETL enquanto um todo.

O primeiro esforço em utilizar uma linguagem de especificação como a Alloy para modelar diferentes processos ETL foi registado no trabalho de Oliveira et al. (2016), no qual foi proposta uma abordagem orientada a padrões. Neste trabalho foi apresentada a definição de um padrão ETL de uma dimensão de variação com manutenção de histórico - SCD-H -, enquanto padrão de transformação, e a sua especificação em Alloy, incluindo a sua modelação estrutural e comportamental. O trabalho realizado evidencia o objetivo principal desta abordagem: a

implementação de processos ETL com um grau elevado de abstração e com formalismo na sua modelação, sem acarretar uma grande carga matemática. Adicionalmente, a especificação realizada pode ser validada, tirando partido das capacidades de análise da linguagem Alloy.

2.2 Linguagens de Especificação Formal

As especificações formais são técnicas matemáticas utilizadas para descrever um sistema e analisar o seu comportamento. O seu propósito é auxiliar na implementação de sistemas e produtos de software, ajudando à sua modelação através da verificação de propriedades chave. As especificações formais utilizam notações matemáticas para descrever de forma precisa as propriedades de um sistema. Contudo, fazem-no sem restringir a forma como essas propriedades são mantidas ou alcançadas – uma especificação formal descreve o que um sistema deve fazer sem especificar como é feito (Spivey, 1989).

A filosofia por detrás da utilização de especificações formais no desenvolvimento de software baseia-se na aplicação de abstrações no seu projeto. As abstrações em software implicam uma definição precisa e consistente das propriedades do sistema a desenvolver. Isto conduz a uma maior compreensão do próprio sistema e da sua modelação, ajudando na descoberta de problemas numa fase inicial de desenvolvimento. Adicionalmente, ao se definir as propriedades do sistema a modelar, a sua complexidade conceptual é reduzida e torna-se mais fácil a sua decomposição modular. Desta forma, as abstrações implicam, frequentemente, a solidez dos conceitos base do produto, trazendo interfaces simples e pequenas e produtos escaláveis, em termos das suas funcionalidades gerais.

As linguagens de especificação formal (LEF) são utilizadas para expressar especificações formais numa dada linguagem, cujo vocabulário, sintaxe e semântica são formalmente definidas, o que significa que são baseadas em conceitos matemáticos bem compreendidos. De acordo com Sannella e Wirsing (1999), os componentes base de qualquer linguagem de especificação formal incluem, entre outros:

- construtores para especificar propriedades de componentes individuais de programas como tipos e funções;

- mecanismos estruturais para construção de especificações extensas de uma forma modular;

- uma descrição da semântica da linguagem e um conjunto de mecanismos para executar provas de propriedades de especificações.

De seguida, são apresentadas algumas linguagens ou técnicas de especificação formal que hoje podemos utilizar na especificação de produtos de software.

2.2.1 VDM (VDM-SL)

A linguagem VDM-SL é baseada no Método de Desenvolvimento de Viena (VDM). Este método foi desenvolvido inicialmente nos laboratórios da IBM em Viena, na década de 1970, com o objetivo de ajudar no desenvolvimento de Sistemas Operativos. O VDM é uma das metodologias formais mais conhecidas, sendo uma técnica de especificação orientada por modelos, baseada em cálculo proposicional e predicativo e lógica de funções parciais. VDM é desenhado para suportar o refinamento consecutivo de modelos abstratos, em implementações concretas, utilizando-se para isso uma abstração de representações e uma abstração de operações (Plat & Larsen, 1992).

Um modelo em VDM pode ser expresso numa linguagem de especificação – VDM-SL. Um modelo consiste em dois componentes principais: um modelo do estado, com invariantes específicos definidos, e um conjunto de operações sobre os tipos de dados abstratos pertencentes ao estado. As operações são definidas com base na sua assinatura, pré-condições e pós-condições e podem aceder e alterar variáveis externas pertencentes ao estado do sistema, sendo obrigatória a definição de acesso de cada variável (apenas leitura ou escrita) em cada operação. As descrições de estado são definidas em níveis sucessivos de abstração, interligados por passos de implementação.

Atualmente, existem várias extensões à VDM-SL que lhe adicionam novas funcionalidades tendo, normalmente, como alvo um tipo específico de software. Algumas das extensões são a VDM++ (Dürr & van Katwijk, 1992), que introduz o conceito de programação orientada a objetos, e a VDM-RT (Mukherjee, Bousquet, Delabre, Paynter, & Larsen, 2000), previamente conhecida por VICE, que inclui suporte para modelos temporais discretos. Adicionalmente, tanto a nível comercial como académico, é possível encontrar inúmeras ferramentas de suporte ao VDM, incluindo suporte a teste

e prova de propriedades de modelos e geração de código a partir de modelos válidos – e.g. VDMTools[1] e OvertureTool[2].

2.2.2 Larch (LSL e LIL)

Larch é uma família de linguagens de especificação (Guttag & Horning, 1986) desenvolvida, primeiramente, nas décadas de 1980 e 1990, nos Estados Unidos da América. Este desenvolvimento ocorreu no âmbito de um projeto (Larch Project) que pretendeu explorar linguagens, métodos e ferramentas de especificação formal. Cada especificação em Larch tem componentes escritos em duas linguagens:

- uma desenhada para uma linguagem de programação específica, denominada de Larch Interface Language – LIL,

- uma independente de qualquer implementação, denominada de Larch Shared Language - LSL.

As especificações abstratas definidas em LSL podem ser transformadas para uma linguagem LIL para descrever unidades de programação (módulos, funções, tipos, etc.) usadas para implementação na linguagem de programação associada. Ao contrário de linguagens como a VDM-SL ou a Z, a Larch é orientada por propriedades, especificando indiretamente o comportamento do sistema. As linguagens baseadas em propriedades preocupam-se mais em identificar as propriedades externas do sistema, e o que este deve fazer, relacionando-se mais proximamente com a especificação de requisitos.

As especificações em LSL são modularizadas em *traits* que introduzem operadores e especificam as suas propriedades. *Traits* podem corresponder a tipos de dados abstratos ou capturar propriedades úteis, que podem ser partilhadas por outras *traits*. O conjunto de operadores é declarado em termos de símbolos de operadores e as suas assinaturas. As propriedades do conjunto de operadores

[1] http://fmvdm.org/
[2] http://overturetool.org/

descrevem uma teoria, que é, essencialmente, um conjunto de teoremas que podem ser derivados a partir das propriedades definidas no *trait*, utilizando axiomas e regras de inferência de lógica de primeira-ordem (Mišić & Velašević, 1997).

Existem várias linguagens de interface desenhadas para linguagens de programação específicas, entre as quais se destacam as desenvolvidas para C e Modula-3, Ada, CLU e C++. Com o estabelecimento da diferença entre linguagens LIL e LSL, a Larch encoraja uma separação de preocupações, as primeiras focando-se nos detalhes de implementação e a segunda nos construtores base da especificação do sistema. Existem múltiplas motivações para manter maior complexidade na especificação em LSL em vez de na LIL, uma vez que essas são mais prováveis de serem reutilizadas. Além disso, é mais fácil fazer asserções sobre propriedades semânticas de uma especificação em LSL do que propriedades semânticas de uma especificação de interface (Garland, Guttag, & Horning, 1993).

2.2.3 A notação Z (Z)

A linguagem Z é uma linguagem baseada na notação Z (Sufrin, 1986). Esta foi desenvolvida na década de 1980 na Universidade de Oxford, sendo, posteriormente, padronizada pela ISO (International Organization for Standardization & International Electrotechnical Commission, 2002). A notação Z é uma das mais conhecidas notações de especificação usada para descrever e modelar sistemas de computação, sendo orientada a modelos e baseada na teoria de conjuntos tipados. A Z utiliza conceitos matemáticos bem conhecidos, como conjuntos, relações, funções, entre outros, que são estruturados em esquemas (*schemas*). Um esquema é constituído por uma parte declarativa, que inclui a declaração de variáveis, e uma parte predicativa, que inclui a definição de predicados e axiomas que relacionam as variáveis (Mišić & Velašević, 1997), modelando, assim, os aspetos estáticos e dinâmicos do sistema. Desta forma, em Z, um sistema é modelado, normalmente, através da representação do seu estado - um conjunto de variáveis de estado e os seus valores - e operações que podem alterar o seu estado. Com a utilização de esquemas na estrutura da especificação, é possível utilizar operações para combinar dois ou mais esquemas, tanto a nível das suas partes declarativas ou das suas partes predicativas, como ao nível das duas, facilitando a modularidade das especificações desenvolvidas. Estas operações sobre esquemas constituem aquilo que podemos designar como o cálculo de esquemas - uma característica base que oferece à linguagem Z um

poderoso mecanismo estrutural. Aspetos como a integridade de objetos e a comunicação entre estes podem ser especificados formalmente com a utilização do sistema estrutural apresentado.

A notação Z é utilizada de forma exaustiva em diferentes sistemas de software, como a especificação do sistema de ficheiros UNIX e especificações de hardware. Atualmente, conta com uma comunidade ativa e com suporte em inúmeras ferramentas, académicas e comerciais. Veja-se, por exemplo, os casos de CZT[3], ProofPower[4], ProZ[5] e HolZ[6].

2.2.4 Alloy

A Alloy é uma linguagem textual que foi desenvolvida em 1997, no MIT, por uma equipa liderada por Daniel Jackson (Jackson, 2002a). A Alloy é capaz de descrever estruturas complexas, bem como as suas restrições e os seus comportamentos. Esta linguagem foi desenhada não só para ser descritiva, como também analisável, uma vez que as especificações escritas em Alloy podem ser validadas através da ferramenta Alloy Analyzer (Jackson, 2012).

A linguagem Alloy é profundamente enraizada na notação Z. A Alloy permite descrever as estruturas com um conjunto mínimo de notações matemáticas e utiliza, também, notações de modelação de objetos, simplificando a classificação de objetos e a associação de propriedades de acordo com essa classificação. Os processo de análise em Alloy são implementados através da descoberta de instâncias de um modelo ou de contraexemplos de uma asserção relacionada com o modelo. A procura das instâncias é feita de forma exaustiva dentro de um domínio com cardinalidade definida pelo utilizador. A abordagem seguida pela Alloy consegue, em geral, apenas revelar a presença de erros num modelo e não provar a sua ausência. Contudo, a hipótese 'Small Scope' postula que a maior parte dos erros pode ser encontrada em modelos com *scope* pequeno (Andoni, Daniliuc, Khurshid, & Marinov, 2003). Desta forma, a Alloy tem vindo a ser utilizada com sucesso na modelação de problema de diferentes áreas, trazendo um grau de confiança elevado.

[3] http://czt.sourceforge.net/
[4] http://www.lemma-one.com/ProofPower/index/
[5] https://www3.hhu.de/stups/prob/index.php/ProZ
[6] https://www.brucker.ch/projects/hol-z/

A análise incorporada na ferramenta Alloy Analyzer tira partido da tecnologia para a resolução de problema de satisfação de booleanos (SAT). Esta ferramenta traduz as restrições a serem resolvidas pela Alloy em restrições booleanas, alimentadas aos SAT *solvers*. Estes procuram uma valoração para as variáveis da fórmula recebida tal que esta seja satisfeita. O resultado obtido é mapeado pela ferramenta de volta para o contexto da especificação e apresentado ao utilizador.

A linguagem Alloy surge como uma resposta às dificuldades frequentemente encontradas em aplicar especificação formal a sistemas de software, oferecendo uma ferramenta que permite uma modelação rápida e análise que retorna *feedback* imediato. Ao contrário da maioria das notações conhecidas até ao momento, a Alloy não utiliza uma sintaxe matemática nem se baseia na prova completa de teoremas - a análise é feita apenas sobre um espaço limitado de casos, podendo, no entanto, analisar num espaço com biliões de casos e sendo completa dentro do espaço estabelecido. No capítulo seguinte abordaremos de forma mais detalhada esta linguagem de especificação, uma vez que, e dadas as suas características, a Alloy é um principal foco deste trabalho de dissertação, tendo sido selecionada para fazer a especificação de processos ETL.

Capítulo 3.

Especificação e Validação de Processos ETL em Alloy

3.1 Aplicação da Linguagem Alloy

3.1.1 Filosofia base

A Alloy é uma linguagem orientada a modelos, utilizando, para isso, um conjunto mínimo de notações matemáticas e tirando partido da teoria de conjuntos. Esta linguagem defende o desenho de abstrações, enquanto ideias reduzidas à sua forma essencial, para uma definição de interfaces simples e facilmente escaláveis, sem se prender a detalhes de implementação. Modelar um sistema em Alloy é um exercício que exige um nível alto de compreensão do problema e uma forma genérica de analisar a solução a desenvolver. Esta simplificação leva a uma maior clareza no processo de planeamento, permitindo detetar vários problemas de desenho nesta fase inicial. Além disso, a modelação em Alloy não é um processo complexo, tendo uma curva de aprendizagem suave e sendo o seu resultado final facilmente legível.

3.1.2 Modelos Alloy

Os modelos escritos em Alloy são micromodelos, declarativos, estruturais e analisáveis (Jackson, 2002b). O desenvolvimento de modelos Alloy é realizado no seu próprio ambiente, que disponibiliza a ferramenta Alloy Analyzer para a obtenção de soluções para o modelo construído e a validação de asserções sobre o mesmo. Assim, o desenvolvimento em Alloy é feito de forma interativa, utilizando os cenários obtidos através da análise do modelo para a sua construção incremental.

Uma das principais características da linguagem é o conceito de relação. Na verdade, a Alloy baseia-se em duas definições: átomos e relações. Os átomos são entidades indivisíveis, imutáveis e não-interpretadas (sem qualquer tipo de propriedades nativas). A forma como tudo evolui a partir dos átomos deve-se à entidade de relação, que é a responsável por relacionar os átomos. Uma relação é composta por um conjunto de tuplos (por exemplo, se *a* se relaciona com *b* pela relação R (*a* R *b*), então o tuplo (*a*,*b*) pertence à relação R). De referir que, um conjunto de átomos é uma relação unária, um escalar é uma relação unária com apenas uma entrada, e os relacionamentos mais complexos são traduzidos em relações entre átomos.

Um modelo Alloy é constituído por uma declaração de módulo, um conjunto de importações de módulos e um conjunto de parágrafos. Estes podem ser uma **declaração de assinatura**, uma **restrição** ou um **comando**. Uma declaração de assinatura representa um conjunto de átomos e pode introduzir campos que representam relações entre assinaturas. As restrições são divididas em **factos** (invariantes do modelo), **predicados** (restrições a serem usadas e cumpridas em diferentes contextos), **funções** (expressões reutilizáveis com capacidade de retornar um valor) e **asserções** (implicações a serem confirmadas). Os comandos são instruções para analisar o modelo construído – por exemplo, o comando ***run*** procura soluções ou instâncias de um predicado específico e o comando ***check*** procura contraexemplos que contrariem uma determinada asserção.

Para a escrita de um modelo em Alloy, é possível tirar partido de uma lógica relacional que combina quantificadores de lógica de primeira ordem e operadores de cálculo relacional, oferecendo um grande poder de expressividade, especialmente na declaração de restrições do sistema.

3.1.3 Exemplo de um Modelo Alloy

A nível de exemplo, podemos considerar o modelo de sistemas de ficheiros escrito em Alloy, inspirado no tutorial de Alloy disponível online[7]. Este modelo, tal como a maioria dos modelos Alloy, pode ser dividido na modelação estrutural das entidades do sistema e na modelação dinâmica do mesmo.

Modelação estrutural

```
1.  // File system objects
2.  abstract sig FSObject {
3.      size: one Int
4.  }
5.  sig File, Dir extends FSObject { }
6.
7.  // A File System
8.  sig FileSystem {
9.      live: set FSObject,
10.     root: Dir & live,
11.     contents: Dir lone -> FSObject,
12.     parent: FSObject -> lone Dir
13. }{
14.     // root has no parent
15.     no root.parent
16.     // live objects are those reachable from the root
17.     live = root.*contents
18.     // contents only defined on live objects
19.     contents in live->live
20.     // parent is the inverse of contents
21.     parent = ~contents
22. }
```

Figura 1 - Modelo estrutural em Alloy de um Sistema de Ficheiros

Na Figura 1 podemos ver que o modelo de sistema de ficheiros declara, primeiramente, as assinaturas que serão utilizadas para representar o sistema, nomeadamente: **FSObject** (objetos do sistema de ficheiros), **File** e **Dir** (objetos ficheiro e objetos diretoria) e **FileSystem** (sistemas de ficheiros). As assinaturas são, tecnicamente, conjuntos – relações unárias. Porém podem ser consideradas como classes, no paradigma de programação orientada a objetos. Segundo essa

[7] http://alloytools.org/tutorials/online/frame-FS-7.html

abordagem, a assinatura **FSObject**, por exemplo, pode ser interpretada como uma classe abstrata, e **File** e **Dir** classes que a estendem. A classe **FSObject** define um campo **size** que, em cada instância das classes que a estendem, aponta para uma instância da classe **Int**.

Seguindo uma interpretação mais próxima do modo de funcionamento interno da Alloy, a mesma modelação pode ser considerada como a definição dos conjuntos **FSObject**, **Dir**, **File** e **FileSystem**. Uma vez que **Dir** e **File** estendem **FSObject**, estes são subconjuntos disjuntos do último. Adicionalmente, uma vez que **FSObject** é uma assinatura abstrata, este conjunto é a união de **Dir** e **File**. **size** é uma relação que mapeia cada elemento de **FSObject** num elemento do conjunto **Int**. Por fim, a assinatura **FileSystem** representa o conjunto de todos os sistemas de ficheiros e é introduzida para permitir a modelação e visualização de vários sistemas simultaneamente. Esta assinatura declara a relação **live**, que indica o conjunto de objetos ativos no sistema, e **root**, que indica a raiz do sistema de ficheiros. Estas são duas relações binárias e expressam o seu significado ao relacionarem cada sistema de ficheiros com um conjunto de objetos (elementos **FSObject**), no caso da relação **live**, e cada sistema de ficheiros com uma diretoria que é um dos seus objetos (elementos da interseção **Dir & live**), no caso da relação **root**. Adicionalmente, são, ainda, declaradas as relações **contents** e **parents** – duas relações ternárias. Por seu lado, a relação **contents** mapeia cada sistema de ficheiros numa relação binária entre diretorias e objetos do sistema. A definição da relação com o multiplicador **lone** no lado esquerdo (**contents: Dir lone -> FSObject**) indica que cada objeto do sistema se relaciona com, no máximo, uma diretoria em cada sistema através da relação **contents**. A relação **parent** mapeia cada sistema numa relação entre objetos do sistema e diretorias. A utilização do multiplicador **lone** no lado direito (**parent: FSObject -> lone Dir**) indica que cada objeto do sistema tem como pai, no máximo, uma diretoria em cada sistema.

À declaração da assinatura **FileSystem** são, também, associados factos. Em Alloy, factos são restrições explícitas adicionadas ao modelo. A Alloy não vai considerar cenários ou soluções que violem os factos definidos. Estes podem ser declarados associados a uma assinatura ou, de forma isolada, através de uma declaração **fact**. A primeira abordagem é adotada, normalmente, para definir restrições de sanidade do sistema. A segunda abordagem é utilizada, normalmente, para restrições menos triviais e mais específicas. No exemplo em consideração, são declarados como factos associados à assinatura **FileSystem** algumas propriedades para a correta representação dos

elementos deste conjunto – como por exemplo, é declarado que o elemento **root** não tem nenhum elemento a si associado na relação de **parent**, e que as relações **parent** e **content** são simétricas.

Modelação dinâmica

Nesta altura, acrescentamos ao modelo anteriormente apresentado alguns comportamentos dinâmicos. O modelo apresentado na Figura 2, por exemplo, introduz a noção de remoção de objeto de um sistema com a definição de um predicado e a verificação de propriedades que queremos garantir com a definição de uma asserção. Sendo esta uma linguagem declarativa, em Alloy os comportamentos são definidos pela descrição de estados. O predicado **remove** recebe um objeto a ser removido e dois elementos de **FileSystem**, **fs** e **fs'** que, por convenção, representam o estado inicial (ou pré-estado) e final (ou pós-estado), isto é, o sistema de ficheiros antes e depois da remoção do objeto.

```
1.  // Delete the file or directory x
2.  pred remove [fs, fs': FileSystem, x: FSObject] {
3.      x in (fs.live - fs.root)
4.      fs'.parent = fs.parent - x->(x.(fs.parent))
5.      fs'.root = fs.root
6.  }
7.  run remove for 2 FileSystem, 4 FSObject
8.
9.  // remove removes exactly the specified file or directory
10. assert removeOkay {
11.     all fs, fs': FileSystem, x: FSObject | remove[fs, fs', x] => fs'.live = fs.live - x
12. }
13. check removeOkay for 5
```

Figura 2 - Modelação em Alloy da operação de remoção de um objeto num Sistema de Ficheiros

O predicado descrito, tal como todos os predicados em Alloy, é apenas uma restrição, apesar de ser introduzido para representar uma operação. Este predicado será, posteriormente, utilizado num contexto de restrição ou diretamente numa execução, e só será verdade quando o argumento **fs'** for válido tendo em conta os restantes argumentos de entrada.

Apesar de ser indiferente a ordem pela qual as linhas interiores de um predicado são organizadas, é normal fazer a sua divisão em três partes: a definição de pré-condições, pós-condições e condições de quadro (*frame conditions*). A primeira definição estabelece as propriedades que precisam de ser

verdade para o predicado ser satisfeito, a segunda dita as propriedades que se pretende verificar após o predicado - as alterações que se deseja obter - e a última é utilizada para manter inalterada alguma parte do sistema.

O predicado **remove** apresentado na Figura 2 estipula, na primeira linha do seu corpo, que o ficheiro a ser removido pertence à diferença entre o conjunto de ficheiros ativos do estado inicial e o seu objeto raiz. Esta restrição pode ser considerada como a pré-condição do predicado. De seguida, é estabelecido que a relação **parent** do estado final (sistema de ficheiros **fs'**) é igual à relação **parent** do estado inicial (sistema de ficheiros **fs**) sem o tuplo constituído pelo objeto a remover e o seu pai. Este tuplo é construído usando o símbolo **->** para relacionar dois elementos, e usando o operador ponto **.** para aceder à relação binária de **fs.parent** e, nesta, aceder ao elemento mapeado a partir do objeto a ser removido. Uma vez que esta propriedade concretiza o objetivo do predicado (a remoção de um objeto num sistema de ficheiros), esta pode ser interpretada como a pós-condição do predicado **remove**. Quanto à última restrição, que dita que o estado final e o estado inicial (**fs** e **fs'**) têm o mesmo objeto raiz, esta pode ser considerada uma condição de quadro que mantem imutável a relação **root**.

Análise e Validação

A modelação apresentada continua com a definição de um comando **run** para o predicado **remove**. A execução deste comando tenta encontrar uma solução para o sistema na qual o predicado seja verdade. Os comandos em Alloy recebem, obrigatoriamente, um *scope* a ser aplicado (um limite superior do número total de elementos por assinatura). A análise em Alloy é sempre de confiança (sem falsos positivos) mas incompleta (limitada superiormente). No entanto, esta é completa dentro do limite definido e suporta limites significativos. A execução de um comando em Alloy permite a visualização e análise dos cenários encontrados, com a ferramenta Alloy Analyzer.

A segunda parte da modelação dinâmica define uma asserção – **removeOkay**. Uma asserção estabelece uma propriedade que esperamos que se verifique sempre, com base no comportamento do modelo. Neste caso, **removeOkay** pretende verificar que o predicado de remoção de um objeto remove exatamente esse objeto (deixando de incorporar o conjunto de objetos ativos). Para o expressar, é declarado que, para quaisquer sistemas de ficheiros (inicial e final) e qualquer objeto, a verificação do predicado de remoção com o sistema final, sistema inicial e objeto enquanto

argumentos, implica que o sistema final tenha como objetos ativos os mesmos objetos ativos do sistema inicial exceto pelo objeto removido.

A asserção definida é verificada com o comando **check**, que tenta encontrar um contraexemplo da asserção dentro do *scope* definido. Se a execução do comando não obtiver resultados, significa que nenhum cenário que invalide a asserção foi encontrado no limite utilizado e, por isso, a asserção pode ser válida. Se a execução do comando obtiver algum resultado, então foi encontrado um contraexemplo à asserção definida e esta, assim, é inválida.

3.1.4 A aplicação da Alloy

A construção de modelos de uma forma incremental, a facilidade de leitura do modelo obtido e o poder de verificação que a Alloy oferece são aspetos cada vez mais valorizados na comunidade académica e empresarial. A Alloy, sendo uma linguagem de especificação formal leve, tem sido adotada em diferentes cenários aplicacionais, construindo uma popularidade caso a caso, em crescendo. Esta adoção revela não só a utilização da Alloy enquanto uma simples ferramenta no processo de construção de software, mas também como uma abordagem e metodologia de desenvolvimento. Recentemente, esforços têm sido feitos para a construção de um ecossistema de ferramentas de suporte à Alloy – como a conversão entre Alloy e UML (Anastasakis, Bordbar, Georg, & Ray, 2007; Cunha, Garis, & Riesco, 2013), e o mapeamento entre especificações Alloy e implementações de bases de dados (Cunha & Pacheco, 2009). Esta corrente tem como objetivo incentivar e dar suporte à utilização desta linguagem para uma Engenharia Conduzida por Modelos - *Model-Driven Engineering* (MDE). A MDE é uma abordagem baseada em transformações de modelos em diferentes níveis de abstração, aplicadas sucessivamente até alcançar níveis mais concretos – como código. A adoção da linguagem Alloy para a modelação e análise de sistemas incluí exemplos como a análise de protocolos de configuração de rede, controlo de acesso, criptografia, entre outros[8]. A geração de soluções da Alloy é uma vantagem especialmente significativa na modelação de sistemas cuja configuração é indeterminada ou dinâmica.

[8] http://alloy.csail.mit.edu/alloy/faq.html#10

A Alloy foi aplicada, também, na área médica para explorar o desenho de algumas funcionalidades de sistemas de terapia de cancro (Jackson, 2006). Com a modelação obtida, foi possível detetar alguns cenários inesperados na calendarização, alocação e envio de máquinas para salas, para os quais o sistema não estava completamente preparado, e ainda perceber o motivo e possível solução a alguns comportamentos estranhos detetados nas máquinas. Este esforço reforça ainda mais a ideia de que uma especificação inicial, capaz de modelar um sistema de forma fiel e com as abstrações adequadas, conduz ao desenvolvimento de um produto de software mais preciso, claro e escalável.

3.2 Componentes Chave de um Processo ETL

Na sua essência, um processo ETL tem como principal objetivo a extração de dados de diferentes fontes, a sua limpeza e manipulação, e posterior carregamento para um sistema de *data warehousing*. Para tal, o processo é constituído por um conjunto de tarefas cuidadosamente planeadas e coordenadas, e sujeito a requisitos temporais, de negócio e físicos.

3.2.1 Natureza dinâmica

Um processo ETL é definido como um *workflow*, traduzindo-se na ordenação, calendarização e sincronização de tarefas ao longo de todo o processo. As tarefas ETL podem ser vistas como componentes, encadeados numa determinada ordem e num determinado contexto. A execução de uma tarefa ou bloco de tarefas tem como consequência a manipulação de dados nas entidades envolvidas (criação, alteração, remoção de dados), o que desmascara a natureza evolutiva e dinâmica da execução de um processo ETL.

3.2.2 Tarefas ETL

Apesar da especificidade de cada sistema de *data warehousing* e de cada ETL associado, é possível identificar tarefas feitas recorrentemente em processos ETL. O primeiro desafio deste processo passa pela extração de dados a partir de fontes heterogéneas. Os dados podem ser providenciados em diferentes formatos, como ficheiros CSV, ficheiros XML, bases de dados relacionais ou não

relacionais, entre outros. Para cada fonte de dados, é necessária a definição de uma política de extração, tanto para o processo de povoamento inicial como para o processo de Captura de Dados Alterados – *Change Data Capture* (CDC). Este processo é responsável pela deteção de alterações nos dados armazenados nas fontes, sendo, por isso, essencial para a alimentação regular de dados frescos ao sistema de *data warehousing* e o sucesso deste e do SSD associado. Como método de deteção de alterações podem ser adotadas diferentes abordagens, sendo as mais comuns a utilização de colunas de auditoria em cada objeto de dados das fontes, utilização de ficheiros *log* nas bases de dados operacionais, alimentação de tabelas de auditoria na fonte com uma política de *triggers* ou carregamento incremental – no qual é calculada a diferença entre o atual e o último carregamento de cada objeto de dados.

A segunda etapa de um processo ETL envolve a transformação dos dados extraídos, a qual encapsula uma categoria distinta de tarefas. São estas tarefas que limpam os dados, os conformam e conciliam. A limpeza e a conformação (padronização) dos dados relacionam-se com a garantia da sua correção, integridade e consistência a nível de atributo, de linha e estrutural. Em geral, algumas das anomalias detetáveis nos dados envolvem a falta de integridade referencial, valores contraditórios, erros ortográficos, valores fora do domínio ou nulos, entre outras. Em (Costa, 2006), é estabelecido um conjunto de métodos de resolução das anomalias de dados, dos quais se destacam a decomposição dos dados, a padronização (capitalização, acrónimos e abreviaturas), normalização e correção de valores nulos e entradas duplicadas. Quanto à conciliação dos dados, esta é um passo essencial para a qualidade da informação armazenada no *data warehouse*, uma vez que é o passo responsável por criar uma vista única dos objetos de dados provenientes de diferentes fontes. Usualmente, neste processo utilizam-se tabelas de conciliação (ou de equivalência), nas quais se armazenam os mapeamentos das diferentes chaves naturais de cada fonte e das suas correspondentes chaves de substituição - *Surrogate Key* (SK) -, que são utilizadas no *data warehouse*. Por último, a etapa de carregamento do ETL é responsável pela integração dos dados já transformados no *data warehouse*.

Um dos aspetos mais importantes num *data warehouse* é a consistência dos seus dados e, consequentemente, dos relatórios e análises realizadas sobre estes. Uma vez que um *data warehouse* é um sistema essencialmente histórico, o carregamento de dados alterados, especialmente o carregamento de dados nas tabelas de dimensão, deverá, frequentemente, preservar os seus dados históricos. Assim, na etapa de carregamento é dada especial atenção ao tipo de dimensão a ser alimentada. As dimensões cujos dados podem sofrer alterações são

classificadas quanto à sua variação e ao tipo de resposta face uma alteração, de acordo com a modelação dimensional do *data warehouse*, como as dimensões categorizadas como sendo de variação lenta (*Slowly Changing Dimensions* – SCD) (Kimball & Caserta, 2004). Uma SCD pode ser identificada como tipo 1 (no qual não é armazenada qualquer histórico perante uma alteração de dados), tipo 2 (no qual, perante uma alteração de dados de um registo existente, este é mantido e é inserido um novo registo com os dados alterados e uma nova chave, que será utilizada pelos factos que referenciarem o objeto daí em diante), tipo 3 (para o qual se cria um novo registo com os dados alterados e uma nova chave, com a introdução de um campo com o valor anterior) ou tipo 4 (no qual é mantida uma tabela de histórico separada com os atributos da dimensão e uma data inicial e final, sendo os dados na dimensão mantidos atualizados).

Na etapa de carregamento do ETL são também integrados dados nas tabelas de factos. Este processo requer, frequentemente, uma tarefa de transformação denominada de encadeamento de chaves de substituição (*Surrogate Key Pipelining* – SKP) (Kimball & Caserta, 2004). Esta atividade é responsável por atualizar os valores de dimensão que utilizam chaves de substituição nas entradas que vão integrar uma tabela de factos. Para isso, são utilizadas tabelas de equivalência para cada dimensão, sendo procurada a chave correspondente com base na fonte e na chave natural da dimensão em cada entrada.

3.2.3 Heterogeneidade de Sistemas e a Área de Retenção

Um cenário típico de ETL pode integrar variadíssimos sistemas. Em adição ao próprio *data warehouse* que receberá os dados, o ETL necessita de integrar e conhecer as fontes a partir das quais os dados são extraídos – os seus objetos de dados e funcionamento interno –, para que seja possível a seleção e aplicação da política de extração mais adequada. Adicionalmente, um processo ETL pode utilizar uma base de dados fisicamente separada do *data warehouse* para processar os dados até estes estarem prontos a incorporá-lo. A esta zona de trabalho costuma-se designar área de retenção ou área de preparação - *Data Staging Area* (DSA) na terminologia anglo-saxónica.

Essencialmente, a área de retenção é uma base de dados dedicada à execução do processo ETL. A sua utilização evita sobrecarregar o *data warehouse* (ou as fontes de informação) com as computações envolvidas no processamento dos dados e assegura que a integridade do último não é comprometida. Na área de retenção estão materializadas um conjunto de entidades auxiliares às

tarefas ETL que aí decorrem. Usualmente, são utilizadas tabelas de quarentena, tabelas de equivalência (*look up*), e tabelas de auditoria. Estas tabelas são desenhadas com base na modelação do próprio *data warehouse*, nos requisitos de dados estabelecidos e tendo em consideração os *schemas* das fontes de dados.

3.2.4 Garantia de Propriedades

Um dos principais objetivos e responsabilidades do ETL passa pela garantia da qualidade e integridade dos dados a incorporar no sistema de *data warehousing*. A integridade dos dados pode envolver a sua integridade de entidade, referencial, de domínio e, ainda, integridade relacionada com a lógica de negócio. Apesar dos Sistemas de Gestão de Base de Dados (SGBD) atuais suportarem, total ou parcialmente, a integridade dos dados através de restrições ou outros mecanismos (como *triggers*), é frequente estes serem desativados no *data warehouse*, especialmente durante a execução do processo ETL. Nestes casos, é o processo ETL que deverá manipular os dados de forma a garantir a sua integridade e a lidar com casos problemáticos sem perda de informação. Assim, o sucesso da execução de um processo ETL pode ser validado com a verificação de um conjunto de propriedades que este deve garantir.

3.3 Alloy e ETL

A aplicação da Alloy na modelação e verificação de processos ETL pode ser analisada de acordo com a sua adequação para especificar alguns dos componentes ETL chave identificados anteriormente. Para uma maior facilidade de leitura, esta análise será orientada sequencialmente segundo os passos naturais de uma modelação em Alloy. Assim, de acordo com (Jackson, 2006), a construção de um modelo Alloy começa pela definição dos objetos do sistema (e as suas relações) para uma modelação estrutural do mesmo, seguindo-se a especificação das operações do sistema, que introduzem comportamento dinâmico. Por último, faz-se a inclusão da especificação dos requisitos (predicados e asserções) a validar no sistema, e a sua consequente análise e verificação. Tal como a maioria dos processos de desenvolvimento de software, a especificação de um modelo tem um carácter iterativo.

3.3.1 Modelação Estrutural – Heterogeneidade dos Sistemas e a Área de Retenção

A modelação estrutural dos sistemas envolvidos num processo ETL pode seguir diferentes abordagens. Um dos primeiros aspetos a considerar é a representação de cada sistema enquanto um módulo Alloy, sendo a importação de módulos representativa do conhecimento e conexão entre sistemas. Adicionalmente, a modelação estrutural de uma base de dados relacional, por exemplo, pode ser feita tirando partido de entidades globais como linha, atributo, valor e tabela. Esta modelação é importante uma vez que tanto a área de retenção do sistema de *data warehousing* como o *data mart* ou *data warehouse* associado são frequentemente suportados por bases de dados relacionais.

Para a modelação estrutural de bases de dados relacionais considerámos o trabalho exposto em (Oliveira et al., 2016), no qual foi apresentada uma especificação de diversas estruturas e entidades genéricas para um processo ETL na linguagem Alloy. Um excerto do modelo utilizado está apresentado Figura 3, excerto este que será analisado de seguida[9].

```
1.  sig State {}
2.
3.  sig Value {}
4.
5.  abstract sig Field {}
6.  sig SKField, ControlField, VariationField, DescriptiveField extends Field {}
7.  sig DateField, OperationField, ErrorField extends ControlField {}
8.
9.  sig Row {
10.     values : Field -> lone Value
11. }
12.
13. sig DataObject {
14.     fields: some Field,
15.     keys : some fields,
16.     rows : Row -> State
17. }
18.
19. fact rows {
```

[9] A especificação modela a natureza evolutiva do sistema com a introdução de uma assinatura a representar estados do mesmo, e aplicando *Local State Idiom* (ver Secção 3.3.2).

```
20.     all f : Field | one fields.f
21. }
22.
23. fact dataObject {
24.     all s : State, o : DataObject, r : o.rows.s | r.values.Value = o.fields
25. }
26.
27. pred consistentDataObject[s:State,o:DataObject] {
28.     all f : o.keys, r : o.rows.s | one f.(r.values)
29.     all r1,r2 : o.rows.s | (all f : o.keys | f.(r1.values) = f.(r2.values)) => r1 = r2
30. }
```

Figura 3 - Modelo estrutural em Alloy para modelação ETL (Oliveira et al., 2016)

A modelação estrutural (Figura 3) começa pela declaração das assinaturas que representam o sistema: estados (com a assinatura **State)**, campos e valores (com as assinaturas **Field** e **Value**), linhas ou entradas de dados (com a assinatura **Row**), e objetos de dados (assinatura **DataObject**). **Field** é uma assinatura abstrata que representa um campo (ou atributo), sendo estendida por assinaturas que representam conjuntos de diferentes tipos de campos – campos de chave de substituição (**SKField**), campos com variação (**VariationField**), campos descritivos (**DescriptiveField**), e campos de controlo (**ControlField**). A última é, ainda, estendida por campos de data, de erro e de operação (**DateField, ErrorField** e **OperationField**). Por sua vez, a assinatura **Row** define o conjunto de elementos do tipo linha de dados e declara a sua relação **values** – uma relação ternária que associa cada elemento linha de dados a um mapeamento de elementos campo para, no máximo, um elemento valor.

A especificação de objetos de dados é, como referido, feita pela declaração da assinatura **DataObject**. Esta define os seus campos, chaves e linhas. Para a modelação dos campos e chaves, são declaradas duas relações binárias, **fields** e **keys**, que relacionam cada elemento do tipo com um conjunto não vazio de elementos **Field**. As linhas de um objeto de dados podem ser alteradas durante a evolução do sistema. Neste sentido, é definida a relação mutável **rows**, que é uma relação ternária entre **DataObject**, **Row** e **State**. Associados aos objetos definidos foram, ainda, adicionados factos que garantem a integridade da modelação (só são analisadas soluções que cumprem as restrições dos factos definidos). A especificação faz, também, uso de predicados com restrições relativas à consistência de um objeto de dados num dado estado (modelada pelo predicado **consistentDataObject**). Estas restrições permitem analisar a validade do sistema num determinado estado, ou momento.

3.3.2 Modelação Dinâmica – Natureza Dinâmica e Tarefas ETL

A natureza dinâmica do ETL pode ser capturada com a modelação de uma máquina de estados em Alloy. A lógica inerente à Alloy é bastante genérica e permite um alto nível de liberdade quanto ao estilo de especificação a adotar, como é o caso da modelação de comportamentos dinâmicos (Cunha et al., 2013; Jackson, 2012). A seleção de uma abordagem que seja mais adequada ao sistema que está a ser modelado e ao principal objetivo da modelação é encorajada pelos autores e pela comunidade. No entanto, são estabelecidos dois idiomas principais para a representação da evolução do estado de um sistema. Os dois introduzem o conceito de estado com uma assinatura **State**, ordenada de forma global, e defendem a utilização de predicados declarativos para representar operações, que se traduzem em transições entre um estado inicial e um estado final – usualmente recebidos como argumentos dos predicados. Um dos idiomas para a modelação de comportamento dinâmico é designado por *Global State Idiom*, no qual todas as relações mutáveis são declaradas dentro da assinatura de **State**, adicionando-se, assim, **State** como primeira coluna destas relações. O segundo idioma para representar a evolução de um sistema tem uma aplicação mais simples e é designado por *Local State Idiom*, no qual a assinatura **State** é adicionada como última coluna das relações mutáveis. Esta abordagem tem a vantagem de manter a declaração das relações mutáveis na sua assinatura original (e a aplicação do estado é feita de uma forma local).

A modelação de um processo ETL como uma máquina de estados conduz à definição de uma tarefa ETL como uma operação de transição entre estados. Assim, utilizando um dos principais idiomas para a representação da evolução de um sistema, cada tarefa ETL será modelada de forma a especificar o seu estado inicial e o seu estado final, traduzindo a alteração que deseja verificar. Serão as tarefas que darão o comportamento dinâmico ao modelo, fazendo-o evoluir por transições bem definidas. Num contexto de modelação de processos, seguir uma abordagem declarativa quanto aos estados do sistema e encarar a sua evolução como estando restrita à especificação de transições entre estados pode alterar a complexidade de leitura e interpretação do próprio processo. Como referido em (Wallace, 2003), no qual é explorada a viabilidade da Alloy na modelação de processos de carácter organizacional, esta abordagem afasta a perceção de um processo enquanto uma sequência de ações e comunicações entre diferentes entidades, e conduz a uma imagem mais orgânica e fluída do mesmo.

Em (Oliveira et al., 2016) foi apresentada uma especificação em Alloy de uma tarefa ETL recorrente – o povoamento de uma dimensão SCD do tipo 4. A especificação foi desenvolvida orientada a padrões. De forma geral, um padrão é uma formalização de procedimentos ou técnicas recorrentes, que identifica o conjunto de operações neles envolvidas e abstraí o seu comportamento. Um padrão pode ser considerado enquanto um regra composta por três componentes: o seu contexto, o problema ou forças que tipicamente ocorrem e a forma como a solução resolve o problema. No caso em questão, foi definido o padrão SCD-H, classificado como um padrão de transformação que resolve o problema de armazenamento de dados históricos e dados atuais numa dimensão SCD. Este padrão é definido por três mapeamentos de objetos de dados: mapeamento entre tabela de auditoria e dimensão, mapeamento entre tabela de auditoria e tabela de quarentena e mapeamento entre tabela de auditoria e tabela de log. Nesse trabalho foi especificada uma operação atómica para adicionar uma entrada à dimensão, sendo modelada com um predicado - **addToDimension** (Figura 4). O predicado especificado expressa uma operação e recebe um estado inicial e final, uma linha de dados inicial e final e a configuração do SCD. Além disso, é um predicado declarativo que, de acordo com a ideologia das linguagens de modelação e com a abordagem Alloy anteriormente vista, especifica o resultado que se espera observar no sistema com a adição de uma linha de dados a uma dimensão SCD. Adicionalmente, a especificação define uma asserção para garantir que a consistência do sistema é preservada por esta operação, um passo importante na garantia de propriedades de segurança do modelo.

```
1.  pred addToDimension [s,s': State, r,r': Row, scd: SCD]
2.  {
3.      r in scd.auditToDimension.inData.rows.s
4.      r' not in scd.auditToDimension.outData.rows.s
5.      scd.auditToDimension.inData.rows.s' = scd.auditToDimension.inData.rows.s - r
6.      scd.auditToDimension.outData.rows.s' = scd.auditToDimension.outData.rows.s + r'
7.      all f : scd.auditToDimension.association.Field | f.(r.values) =
            f.(scd.auditToDimension.association).(r'.values) (...)
8.  }
9.
10. assert addToDimensionCorrect
11. {
12.     all s: State, s': s.next, scd: SCD, r: Row |
13.         (consistentSCD[s,scd] and addToDimension[s,s',r,scd]) =>consistentSCD[s',scd]
14. }
```

Figura 4 - Modelo em Alloy do predicado addToDimension e asserção addToDimensionCorrect do padrão SCD (Oliveira et al., 2016)

3.3.3 Especificação de Requisitos – Garantia de Propriedades

Como referido anteriormente (Secção 3.1.2), a modelação em ambiente Alloy oferece duas formas de análise, ambas aplicadas dentro de um determinado limite. A primeira forma de análise consiste na obtenção de soluções para uma restrição, nomeadamente, com a execução do comando **run** de um predicado. Esta análise valida a viabilidade da restrição no contexto do sistema e permite a exploração das soluções obtidas. Isto é, é possível perceber se existem soluções dentro do limite definido nas quais a restrição especificada é cumprida e, no caso de serem encontradas instâncias, aceder à sua descrição completa. A segunda forma de análise consiste na verificação de uma asserção (através da execução do comando **check** de uma asserção). Esta análise procura contraexemplos para a propriedade, e permite perceber se a asserção definida é inválida ou possivelmente válida, ou seja, no caso de serem encontrados exemplos que contradigam a asserção, a sua invalidade é certa. Porém, no caso de não serem encontrados tais exemplos, a validade da asserção é confirmada, mas apenas dentro do limite estabelecido (esta pode ser inválida num limite maior). Assim, a especificação isolada de tarefas ETL em Alloy permite visualizar diferentes soluções de execução, de forma a refinar a especificação construída e detetar possíveis anomalias. Além desta análise, é possível tirar partido de asserções para garantir algumas propriedades desejadas, nomeadamente que sejam capazes de validar a sua correta execução.

Adicionalmente, num contexto de modelação de sistemas evolutivos e de máquinas de estados, as propriedades de um sistema podem ser divididas em propriedades de segurança (*safety*) e propriedades de vivacidade (*liveness*). As propriedades de segurança são cumpridas em todos os estados alcançáveis do sistema e garantem que uma situação negativa ou indesejável não aconteça. As propriedades de vivacidade são satisfeitas num estado futuro em todos caminhos possíveis, e garantem que o objetivo positivo que se pretende é, mais cedo ou mais tarde, alcançado. Estes dois tipos de propriedades são capazes de garantir que um sistema cumpre com os requisitos estabelecidos.

Como exemplo de propriedades de segurança e vivacidade, pode-se considerar o clássico puzzle da travessia de um rio, no qual se pretende transportar itens de uma margem para a outra[10]. Neste

[10] https://pt.wikipedia.org/wiki/Quebra-cabe%C3%A7a_de_travessia_de_rio

quebra-cabeças, um fazendeiro pretende transportar um saco de grão, uma galinha e uma raposa, utilizando um barco capaz de transportar o próprio fazendeiro e um outro item. No entanto, sabe-se que, deixados sozinhos, a raposa comerá a galinha e a galinha comerá o saco de grão. A solução do puzzle é a sequência mínima de travessias que devem ser feitas para que todos os itens fiquem na margem correta, sem que nenhum deles seja comido. Ao modelar este sistema, a propriedade de segurança que se pretende garantir é que nenhum dos itens é comido, isto é, que não são deixados sem supervisão do fazendeiro elementos tais que um deles coma um outro. A propriedade de vivacidade que se pretende alcançar diz respeito à localização de todos os itens na margem desejada do rio.

As propriedades de segurança e de vivacidade podem ser verificadas através de dois métodos: indutivo e direto. No método indutivo verifica-se o cumprimento das propriedades de segurança nos estados iniciais e a sua preservação por todas as operações. Para a análise das propriedades de vivacidade, é utilizada uma métrica positiva que toma o valor de zero quando as propriedades de vivacidade são cumpridas e verifica-se que o seu valor decresce estritamente com todas as operações. Este método permite verificar as propriedades – isto é, se a verificação for possível as propriedades são verdade, caso contrário não é possível tirar qualquer conclusão. No método direto assume-se que os caminhos válidos na máquina de estados estão corretamente modelados. Desta forma é possível verificar o cumprimento das propriedades de segurança diretamente em todos os estados, e o cumprimento das propriedades de vivacidade num dos seus estados. A análise realizada neste método é orientada à invalidação das propriedades: se um contraexemplo for encontrado as propriedades são falsas, caso contrário não é possível tirar qualquer conclusão. Assim, na modelação do ETL enquanto uma máquina de estados, as propriedades de segurança dizem respeito à qualidade e integridade dos dados armazenados no *data warehouse* (esta deve ser preservada em todos os momentos) e as propriedades de vivacidade envolvem, principalmente, o futuro carregamento de todos os dados provenientes das diferentes fontes que sejam válidos. Teoricamente, a verificação das propriedades de vivacidade do ETL através do método indutivo é capaz de devolver uma solução para a coordenação das tarefas envolvidas no ETL, oferecendo uma sequência de estados e transições envolvidas até ser alcançado um estado que satisfaça as propriedades definidas.

Em conclusão, uma verificação das propriedades definidas relativamente a cada tarefa de extração, transformação e carregamento e, possivelmente, em relação ao próprio processo ETL enquanto um

todo, traz um nível de confiança dificilmente alcançado com um planeamento *ad-hoc* ou com o desenho do processo em ferramentas comerciais.

3.3.4 Conclusões sobre a Aplicação da Alloy

A exploração de componentes chave associados aos processos e tarefas ETL que foram identificados, bem como dos seus requisitos, revelou-se agradavelmente compatível com a linguagem e ferramenta em estudo. A modelação dos sistemas físicos envolvidos nos processos ETL pode ser abordada de diferentes formas. O trabalho de Oliveira et al. (2016) oferece uma especificação simples e clara de alguns dos objetos estruturais dos sistemas integrados no ETL. Os objetos genéricos especificados podem, ainda, ser evoluídos para a modelação de objetos mais complexos e significativos no contexto ETL, como tabelas da área de retenção e tabelas de factos ou dimensões do próprio *data warehouse*.

Adicionalmente, a Alloy apresenta diversas soluções para a modelação dinâmica de um sistema, suportando a modelação do ETL enquanto um *workflow* e das suas tarefas enquanto operações modeladas por predicados. Mais uma vez, o trabalho apresentado (Oliveira et al., 2016) pode servir como exemplo de especificação do comportamento de uma tarefa ETL. No entanto, a abordagem apresentada no estudo é feita a um nível muito atómico – a operação especificada é definida para a manipulação de uma linha de dados. Tendo em consideração a grande quantidade de dados manipulados pelo ETL, e a execução, frequentemente, em *batch* das suas tarefas, considera-se mais adequada a modelação das operações aplicadas a uma tabela de dados.

Por último, o poder de análise associado à Alloy toma um papel de grande importância, especialmente no cenário em estudo. De acordo com a natureza da linguagem, não é possível garantir a ausência total de erros de um sistema, uma vez que a análise é feita restrita a um limite definido pelo utilizador. Contudo, a prova é completa dentro desse mesmo limite. Adicionalmente, como já referido, a hipótese 'Small Scope' postula que a maior parte dos erros pode ser encontrada em modelos com um domínio de cardinalidade pequena. Neste sentido, desde que definidas corretamente as propriedades associadas aos processos ETL, a análise em Alloy é capaz de oferecer um grau de confiança difícil de alcançar por outros meios. Assim, com a análise das características e potencialidades da modelação em Alloy e a forma como esta poderia suportar alguns dos

componentes chave associados aos processos e tarefas ETL, a utilização do Alloy surge como um forte candidato à modelação de subsistemas ETL.

Capítulo 4.

O Caso de Estudo

4.1 Contextualização

O caso de estudo selecionado assenta num processo de negócio básico de venda de livros, envolve duas fontes de dados e um *data mart* orientado para a análise de vendas dos mesmos. No centro do caso de estudo encontra-se uma empresa de comercialização de livros, que possui uma loja *online* para venda de livros eletrónicos, em funcionamento há cerca de um ano, e uma cadeia de lojas físicas, que utiliza um sistema operacional com dados há desde dez anos atrás. As duas são suportadas por dois sistemas operacionais distintos, não havendo qualquer ligação entre os clientes nem livros comercializados pelos dois. A organização em questão pretende aceder de forma centralizada e conciliada aos dados das vendas realizadas nas suas várias lojas, de forma a conseguir não só maximizar o volume de vendas, mas também perceber interesses comuns entre os clientes das lojas físicas e *online*, de forma a prestar-lhes um serviço globalmente melhor. Para tal, pretende-se manter um *data warehouse* capaz de agregar a informação relativa aos clientes, livros e vendas da organização. Este sistema alimentará um SSD capaz de permitir análises relevantes como quais os livros mais procurados, os melhores clientes da empresa, o valor total e sazonalidade das vendas, etc.

4.2 A Estrutura Dimensional do *Data Warehouse*

A modelação dimensional do *data warehouse* do nosso trabalho foi realizada com base na identificação da área de suporte à decisão e da granularidade da tabela de factos envolvida, bem como na identificação e caracterização das dimensões de análise e na identificação das medidas dos factos. No *data warehouse* concebido definimos um *data mart* – **OrdersDM** – para acolher informação sobre a venda de livros de uma livraria, com o objetivo de suportar análises relativas às vendas da livraria pelos seus agentes empresariais.

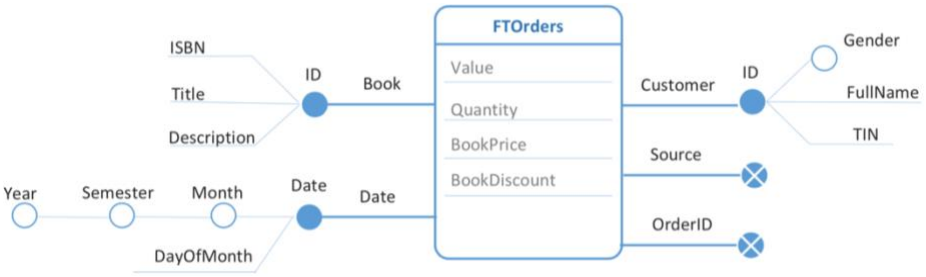


Figura 5 - Esquema dimensional do *data mart* OrdersDM

O *data mart* referido (Figura 5) é constituído apenas por uma tabela de factos do tipo transacional – **FTOrders**. Cada entrada armazenada nessa tabela de factos representa uma venda de exemplares de um livro, a um cliente, a partir de uma livraria ou da loja online. As dimensões presentes no cenário em estudo e integradas no *data mart* são:

- **Date** – dimensão temporal, que acolhe os atributos que sustentam as análises ao longo do tempo.

- **Book** – dimensão com variação lenta e sem manutenção de histórico (SCD Tipo 1), que sustenta as análises com informação relativa a livros. Cada livro é unicamente identificado pelo seu valor textual de ISBN. A chave primária desta dimensão é uma chave de substituição atribuída internamente pelo *data warehouse*.

- **Customer** – dimensão com variação lenta e sem manutenção de histórico (SCD Tipo 1), que permite realizar análises das vendas de acordo com a informação relativa ao cliente que efetuou a compra. Cada cliente é unicamente identificado pelo seu valor de TIN, utilizando como chave primária uma chave de substituição do *data warehouse*.

- **Source** – dimensão degenerada, que caracteriza a fonte da venda.

- **OrderID** – dimensão degenerada, que permite apenas identificar uma dada venda.

Adicionalmente, cada facto tem definido um conjunto de medidas agregáveis, nomeadamente:

- **Value**, que acolhe o valor de aquisição dos exemplares dos livros.

- **Quantity**, que representa o número de exemplares adquiridos do livro.

- **BookPrice**, que armazena o preço unitário do livro.

- **BookDiscount**, que contém o valor do desconto unitário realizado.

4.3 Caracterização das Fontes de Dados

O povoamento do *data mart* que escolhemos foi feito com informação extraída de duas fontes de dados, nomeadamente: SakilaBooks e EbookStore. Analisemos um pouco cada uma destas fontes de informação, vendo as suas características base.

A fonte SakilaBooks[11] é uma base de dados relacional MySQL utilizada pelo sistema operacional de um conjunto de livrarias físicas, sendo o seu domínio de ação o negócio de venda de livros. Esta fonte contempla um leque bastante interessante de informação empresarial relativa a lojas,

[11] O esquema da base de dados da SakilaBooks foi adaptado a partir do conhecido caso de estudo Sakila (https://dev.mysql.com/doc/sakila/en/).

funcionários, clientes, stocks e vendas, bem como outra informação relativa a livros, autores e categorias de livros. Os principais objetos desta fonte de dados são: clientes, lojas, livros, autores e vendas. O esquema lógico desta base de dados está apresentado na Figura 6.

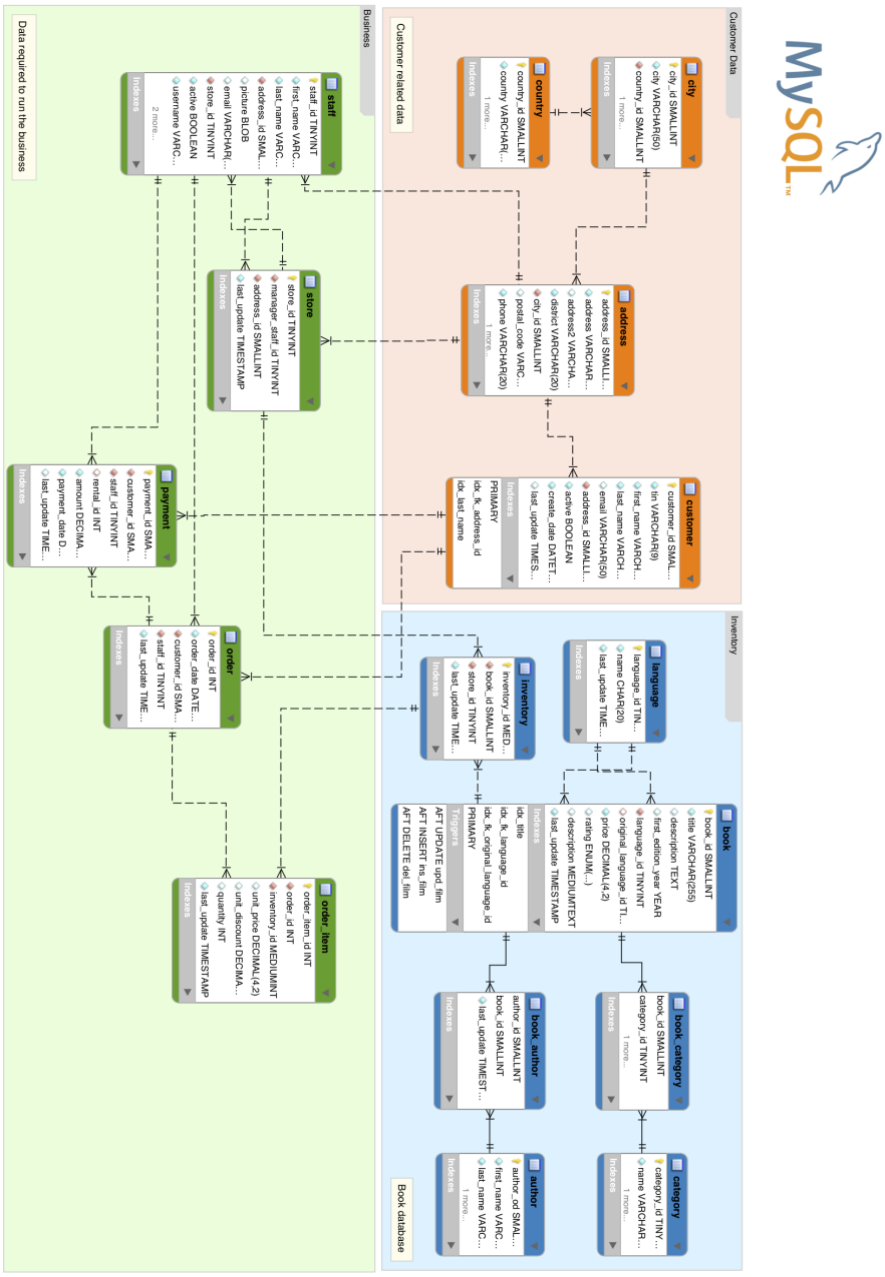


Figura 6 - Esquema lógico da base de dados SakilaBooks

Uma análise atenta ao modelo de dados da fonte SakilaBooks revela que as relações de interesse para o sistema de *data warehousing* são: a tabela **customer**, para obtenção de dados relativos a cliente, e a tabela **book**, para recolha de dados relativos a livros. Quanto aos dados relativos às vendas realizadas, estes podem ser obtidos a partir das tabelas **order**, **order_item** e **inventory** – a primeira tabela armazena dados de vendas (incluindo a data e o cliente associado), a segunda tabela mantem dados de itens das vendas (incluindo valores como preço e quantidade de exemplares e o registo de inventário vendido), e a última tabela contem os dados de inventário que relacionam um livro a uma loja.

As relações de interesse identificadas serão envolvidas num processo de extração de dados, sendo que as tabelas **customer** e **book** serão utilizadas diretamente. Quanto ao acesso aos dados de vendas, foi definida uma vista na fonte de dados - **orders_view** -, que ficou responsável por apresentar os dados de uma forma mais padronizada. Esta vista foi construída com base num conjunto de operações de junção realizado entre as tabelas **order_item**, **order** e **inventory**, selecionando os dados apropriados de cada relação. O resultado é uma relação com os atributos **order_item_id**, **order_id**, **customer_id**, **book_id**, **quantity**, **value**, **unit_price**, **unit_discount**, **order_date** e **last_update**.

Para a inclusão dos dados armazenados por esta fonte de dados ao longo do tempo no sistema de *data warehousing*, é necessária a definição do método utilizado para a sua extração – uma política de Captura de Dados Alterados. Para a fonte SakilaBooks definimos um método de extração diferencial. Nesta abordagem, é mantida uma cópia da última extração de cada relação de interesse da fonte de dados na área de retenção. Em cada extração, as relações de interesse (tabelas e vista definida) são copiadas para a área de retenção, sendo carregadas por inteiro (em tabelas de carregamento contendo os dados mais atuais) e comparadas com a última cópia guardada (em tabelas de carregamento contendo os últimos dados carregados). O resultado da diferença entre uma tabela de carregamento atual e uma tabela de carregamento anterior representa os dados alterados nessa tabela, no processo de extração atual. No final de cada execução, as tabelas referentes ao carregamento anterior são eliminadas e as tabelas referentes ao carregamento atual são renomeadas para tabelas de carregamento anterior.

Quanto à fonte EbookStore, esta é, também, uma base de dados operacional, mas relativa a uma loja *online* de livros eletrónicos, que, tal como a primeira fonte, tem o mesmo domínio de aplicação

- a venda de livros. Esta segunda fonte de informação, uma base de dados relacional, armazena a informação relativa a utilizadores, livros, categorias, tags de livros, e encomendas. Esta fonte de informação está suportada por um sistema MySQL. O seu esquema lógico pode ser observado na Figura 7.

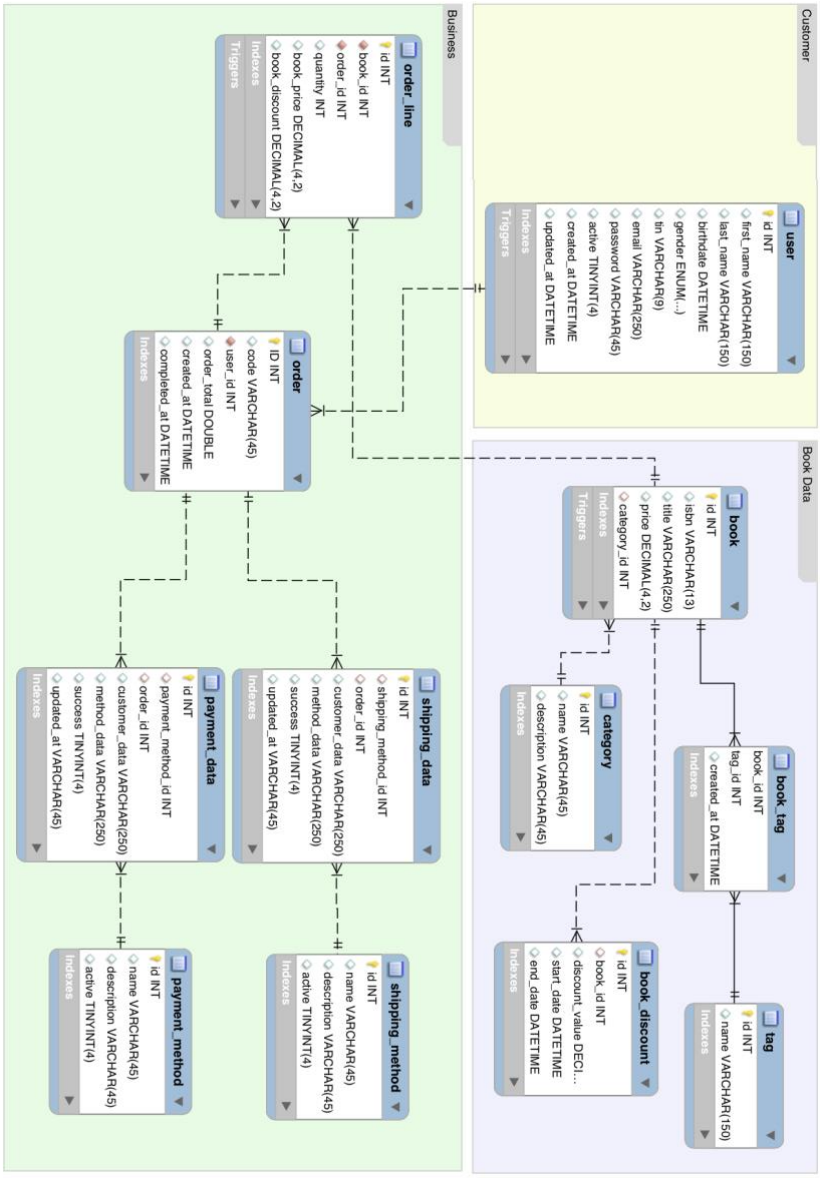


Figura 7 - Esquema lógica da base de dados EbookStore

A análise desta segunda fonte de dados permite a identificação das relações de interesse para o sistema de *data warehousing*, nomeadamente, as tabelas **user** e **book** para a obtenção dos dados relativos a clientes e livros, e as tabelas **order** e **order_line** para a obtenção de dados relativos a vendas de livros. De forma semelhante à primeira fonte de dados analisada, os dados de vendas encontram-se nas últimas duas tabelas – a tabela **order** armazena dados relativos a encomendas (como o valor total da venda e o cliente que a realizou), e a tabela **order_line** armazena dados relativos a cada linha de encomenda (como o livro associado, a quantidade e os preços envolvidos).

As tabelas de interesse identificadas – **user** e **book** – serão utilizadas no processo de extração de dados. De forma semelhante ao observado na primeira fonte de dados analisada, para o acesso facilitado aos dados relativos às encomendas na fonte em questão, optámos por definir uma vista específica na própria fonte de dados – **orders_view**. Esta vista é construída com base num conjunto de operações de junção realizado sobre as tabelas **order_lines** e **orders**, fazendo a seleção dos dados que necessitamos em cada uma destas tabelas. A vista incorpora na sua estrutura os atributos **order_line_id**, **order_id**, **user_id**, **book_id**, **value**, **quantity**, **book_price**, **book_discount**, **order_date** e **last_update**.

Para a inclusão no sistema de *data warehousing* dos dados armazenados por esta fonte de dados ao longo do tempo, é necessária a definição de uma política de Captura de Dados Alterados. Para esta fonte de dados foi selecionado o mesmo método de extração aplicado à primeira fonte de dados: método de extração diferencial. Assim, para fazer a extração dos dados da fonte EbookStore, as tabelas de interesse referidas e a vista definida anteriormente são carregadas por inteiro para a área de retenção do sistema, sendo os seus dados armazenados nas tabelas criadas para acolherem os dados relativos ao carregamento atual. Depois, para se obter os dados que foram alterados na fonte entre o último processo de extração e o atual, as tabelas com os dados do carregamento atual são comparadas com as tabelas com os dados do carregamento anterior. Como já visto, no final de cada processo de extração seguindo o método escolhido, as tabelas referentes ao carregamento anterior são eliminadas e as tabelas referentes ao carregamento atual são renomeadas para tabelas de carregamento anterior.

4.4 A Área de Retenção do Sistema

A área de retenção é uma zona de trabalho, especialmente orientada para a preparação de dados, que separa os sistemas operacionais do *data warehouse*. No sistema de *data warehousing* que concebemos para estudo, a área de retenção acolhe as tabelas de auditoria e as tabelas de quarentena para as tabelas de dimensão **Customer** e **Book,** e para a tabela de factos modelada - **FTOrders**. Adicionalmente, definimos também algumas tabelas de equivalência, em particular para as dimensões que utilizam chaves de substituição, nomeadamente **Customer**, **Book** e **Date,** e outras tabelas de controlo para auxiliar o processo de povoamento definido para o *data warehouse* em questão. O esquema lógico da área de retenção projetada pode ser consultado na Figura 8. Nas secções seguintes presentaremos, de forma sucinta, as estruturas de dados que criámos na área de retenção para sustentação do processo de povoamento do caso em estudo.

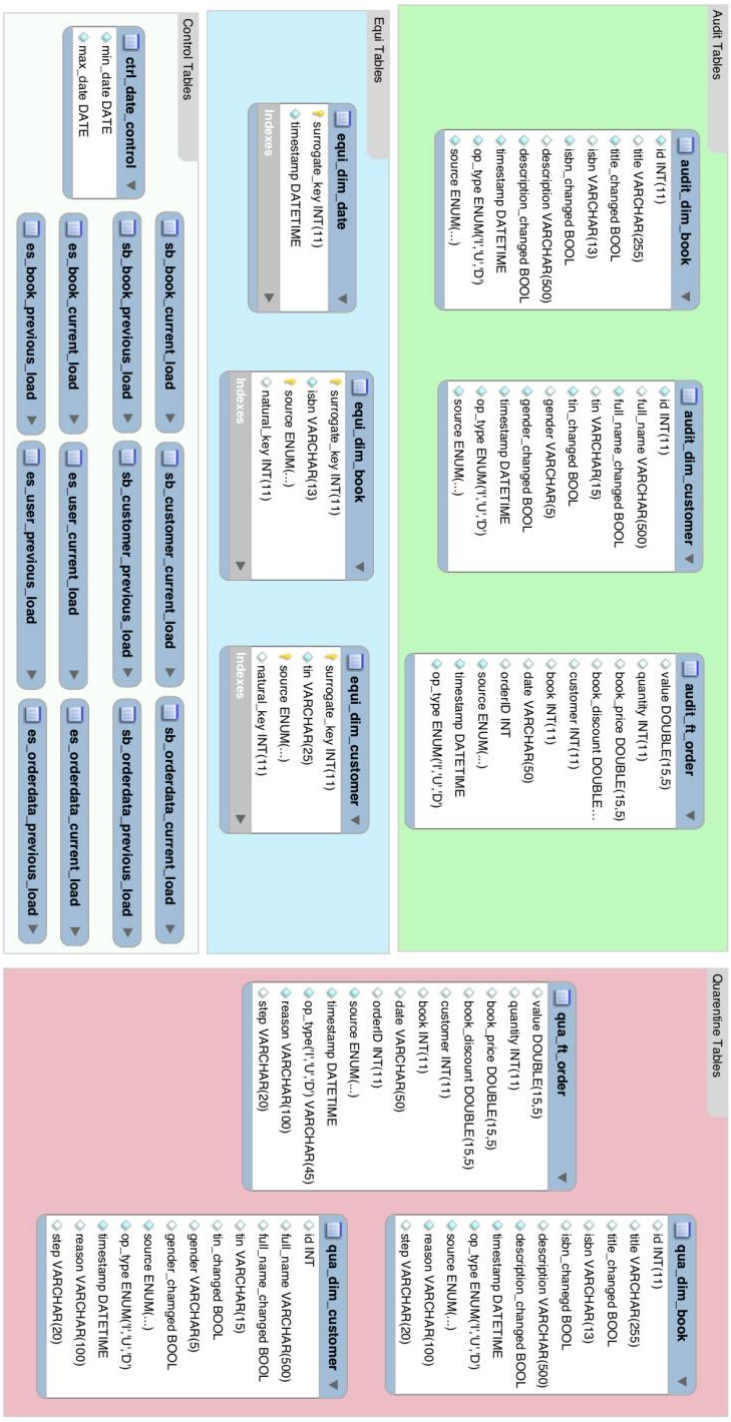


Figura 8 - Esquema lógico da área de retenção do sistema

Tabelas de auditoria

As tabelas de auditoria definidas no sistema (**audit_dim_book**, **audit_dim_customer** e **audit_ft_order**) destinam-se, essencialmente, a receber os dados extraídos das diferentes fontes de informação, já minimamente conformados, e a armazená-los ao longo da execução das transformações que são necessárias para que estes possam ser considerados prontos a integrar o *data warehouse*. Nestas tabelas é incluída informação de controlo a nível da operação que está a ser representada (inserção, remoção ou alteração de dados – atributo **op_type**), a nível da fonte de informação a partir da qual os dados são provenientes (atributo **source**) e a nível de atributos alterados, sendo designado um atributo de controlo para cada atributo descritivo que indica se esse sofreu alteração (por exemplo: atributo **title_changed** da tabela **audit_dim_book**). No momento em que os dados das tabelas de auditoria são considerados prontos a integrar o *data warehouse*, será feito o carregamento dos dados a partir destas tabelas para as tabelas de dimensão ou para a tabela de factos correspondente. No caso de, durante a manipulação dos dados, serem encontrados registos com algum tipo de anomalia, estes serão removidos da tabela de auditoria, sendo marcados como inválidos, complementados com informação de controlo (como o motivo da sua invalidade) e depois inseridos numa tabela de quarentena.

Tabelas de quarentena

As tabelas de quarentena do sistema (**qua_dim_book**, **qua_dim_customer** e **qua_ft_order**) armazenam os dados que foram assinalados como inválidos durante o processo de ETL. Estas tabelas têm uma estrutura de dados bastante semelhante à tabela de auditoria associada, sendo a principal diferença a presença de atributos de controlo adicionais (como o atributo **reason**). Os dados nelas mantidos são posteriormente tratados por um administrador do sistema, de forma a serem recuperados e reintegrados na tabela de auditoria associada para serem mais tarde reintroduzidos no processo de ETL. Caso não seja possível fazer a sua recuperação, poderão ser removidos do sistema.

Tabelas de equivalência

As tabelas de equivalência **equi_dim_customer**, **equi_dim_book** e **equi_dim_date** são, basicamente, tabelas de *look up*, utilizadas no processo de atribuição de chaves de substituição que

envolvam dados relativos às dimensões **Customer**, **Book** e **Date**. Estas tabelas armazenam as chaves naturais e as correspondentes chaves de substituição dos diversos objeto de dados de cada fonte de informação que impliquem a utilização de chaves de substituição. Para que seja possível obter uma vista única dos dados, dois registos de diferentes fontes que representam a mesma entidade, foram mapeados para a mesma chave de substituição. Para tal, foram utilizados atributos de equivalência cujos valores determinam, unicamente, cada um dos objetos que lhes correspondam. Por exemplo, a tabela de equivalência para a dimensão **Book** contém o atributo **ISBN**, e a tabela de equivalência para a dimensão **Customer** contém o atributo **TIN**. Assim, cada uma das tabelas referidas armazena a chave de substituição, o valor do campo de equivalência, a chave natural na fonte e a fonte à qual a chave natural pertence. No caso da tabela de equivalência para a dimensão **Date** apenas foi necessário mapear uma chave de substituição com base no atributo **Date**.

Tabelas de controlo

As tabelas de controlo servem, maioritariamente, para suportar a etapa de extração de dados das diferentes fontes de informação. A extração de dados é realizada através de um método de extração diferencial. Este método mantém uma cópia integral de cada tabela de interesse de cada fonte no momento da última extração. Na extração seguinte, cada relação de interesse é novamente copiada, de forma a serem determinadas as diferenças entre a última cópia e a cópia atual, e derivadas as alterações de dados dessa extração. Assim, são mantidas duas tabelas para cada relação de interesse de cada fonte de informação. As tabelas utilizam como prefixo uma abreviatura da fonte da qual são extraídas (nomeadamente, **sb** para SakilaBooks e **eb** para EbookStore), e como sufixo a indicação de que armazenam dados relativos a um carregamento anterior ou atual (nomeadamente, **previous_load** e **current_load**).

4.5 O Processo de Povoamento

Vejamos, agora, o processo de ETL que idealizámos para o caso de estudo apresentado. No planeamento do processo ETL utilizámos BPMN para fazer a sua esquematização geral. Na Figura 9, podemos observar o esquema BPMN do funcionamento geral deste processo. Na modelação realizada foi excluído o carregamento da dimensão temporal, uma vez que se considerou que este é

um caso especial, cujo carregamento é autónomo e realizado por um processo independente. Adicionalmente, considerou-se também que não existem alterações aos dados que são armazenados na tabela de factos do *data warehouse* – isto é, após o carregamento para o sistema de um facto relativo a uma venda, assume-se que esta não será alterada nem removida.

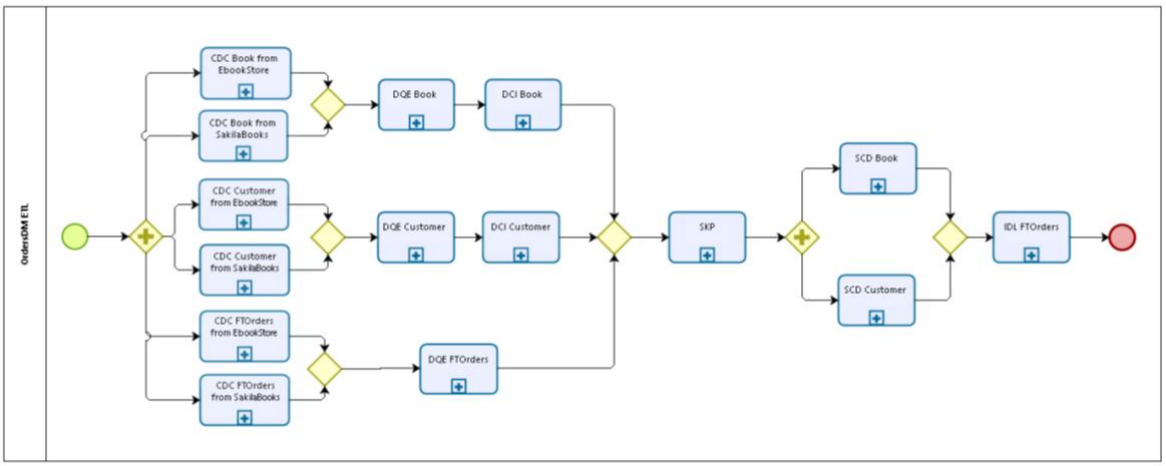


Figura 9 - Esquema em BPMN do ETL para o data mart OrdersDM

De uma forma sumária, o processo de ETL é iniciado com operações de extração de dados sobre as fontes de informação, que são executadas de forma paralela, pelas atividades de Captura de Alteração de Dados (**CDC**) sobre as tabelas das diferentes fontes de informação que alimentam as tabelas de dimensão **Book** e **Customer** e a tabela de factos **FTOrders**. Estas tarefas colocam os dados que angariam e processam nas tabelas de auditoria da área de retenção. Após a extração de dados relativos a cada dimensão ter terminado, o processo de ETL continua com a realização das tarefas de transformação de dados que foram projetadas, de forma a garantir a correção e integridade dos dados coletados – Garantia de Qualidade de Dados (**DQE**) –, e de os conciliar – Conciliação de Dados (**DCI**). Os dados relativos à tabela de factos são sujeitos, também, a uma outra tarefa de garantia da qualidade dos dados - **DQE**. Uma vez todos os dados extraídos e transformados, são aplicadas as tarefas de **SKP** no processo de povoamento da tabela de factos **FTOrders**. Durante esta etapa de transformação, os dados são manipulados nas suas tabelas de auditoria, utilizando-se as tabelas de quarentena no caso de serem registadas anomalias nos dados. Por último, realiza-se o carregamento dos dados para as tabelas no *data warehouse*. As tarefas **SCD** representam o carregamento dos dados das dimensões com variação lenta, e a tarefa **ILD** (*Intensive Load Data*) representa o carregamento dos dados de tabela de factos.

4.5.1 Extração dos Dados

A extração de dados é realizada pela tarefa **CDC**. Esta tarefa é realizada sobre as duas fontes de informação do sistema, para cada uma das dimensões e para a tabela de factos de forma bastante semelhante.

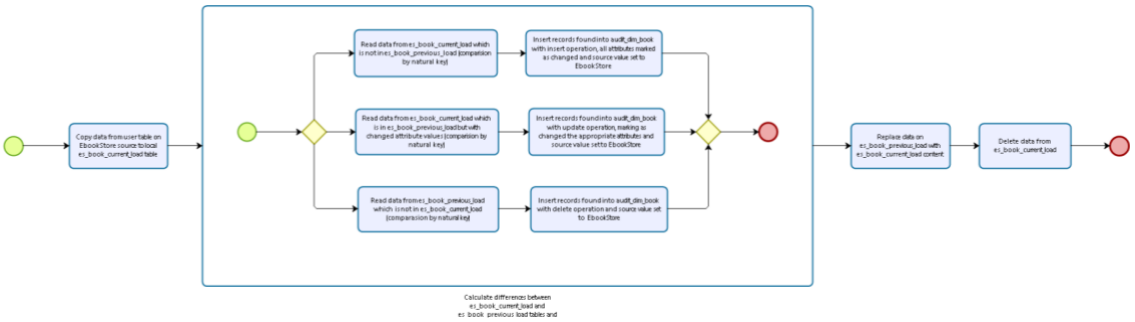


Figura 10 - Esquema em BPMN de CDC para a dimensão Book a partir da fonte EbookStore

Na Figura 10 podemos ver o diagrama BPMN do processo CDC para a dimensão **Book** sobre a fonte **EbookStore**, utilizando o método de extração diferencial. Aqui, o processo começa por copiar a tabela da fonte de informação para uma tabela local (**es_book_current_load**). Depois, esta tabela é comparada com a tabela copiada na última extração (**es_book_previous_load**), sendo o resultado desta comparação usado para inserir os dados envolvidos na tabela de auditoria da dimensão (**audit_dim_book**). Isto significa que, os registos que estiverem presentes na tabela de extração e que não estejam na tabela da última extração – detetados fazendo a comparação dos registos através da sua chave natural (**id**) –, são inseridos na tabela de auditoria correspondente como novas entradas (associadas à operação de inserção). Os registos que estão na tabela do processo de extração atual e na tabela da última extração realizada, mas com valores de atributos diferentes, são inseridos na tabela de auditoria como atualizações (associadas à operação de atualização). Os registos que estão na tabela da última extração e não estão na tabela da extração atual (comparando os registos pela sua chave natural) são inseridos como remoções (associados à operação de remoção). Este processo realiza-se de forma semelhante para a tabela de dimensão **Book** e tabela de dimensão **Customer**.

Para a captura de dados relativos à tabela de factos, este engloba, apenas, a deteção de novas entradas, o que resulta na captura de dados que foram gerados por operações de inserção.

A inserção dos registos que representam as alterações de dados nas diferentes tabelas de auditoria utiliza um mapeamento entre os atributos da tabela da fonte de informação da qual a alteração é proveniente e os atributos da tabela de auditoria correspondente. Adicionalmente, os atributos de controlo da tabelas de auditoria tomam os valores adequados: o atributo **source** toma o valor da fonte de informação envolvida no processo, o atributo **op_type** toma o valor de **I**, **D** ou **U** (consoante se trata de uma operação de inserção, remoção ou atualização), e são atribuídos aos campos de controlo a nível de cada atributo o valor correspondente de **true** (no caso do registo acarretar uma alteração de valor do atributo) ou **false** (caso contrário). Adicionalmente, a inserção dos registos nas tabelas de auditoria utiliza algumas pequenas tarefas de transformação para a conformação dos dados.

Quanto ao mapeamento realizado para sustentar o processo de extração dos dados relativos à dimensão **Book,** a partir da fonte SakilaBooks, este é feito de forma direta a partir das tabelas de carregamento da tabela **book**, com os seguintes mapeamentos: **book.id → id, book.ISBN → ISBN, book.title → title, book.description → description, book.last_update → timestamp**. De forma semelhante, os dados da tabela **books,** da fonte EbookStore, foram mapeados da seguinte maneira: **books.id → id , books.title → title, books.summary → description, books.ISBN → ISBN, book.last_update → timestamp**.

A extração dos dados para povoamento da dimensão **Customer,** a partir da fonte SakilaBooks, é feito utilizando as tabelas de carregamento da tabela **customer**, aplicando os seguintes mapeamentos: **customer.id → id, customer.tin → tin, (customer.first_name + customer.last_name) → fullname, customer.last_update → timestamp**. Uma vez que esta fonte de informação não armazena dados relativos ao género do cliente, é adicionado um mapeamento direto do valor **D** (desconhecido) ao atributo **gender** da tabela de auditoria. Por sua vez, os dados das cópias da tabela **users** da fonte EbookStore são mapeados para o processo de extração de dados, relativos à dimensão **Customer,** segundo os seguintes mapeamentos: **users.id → id , users.tin → tin, users.gender → gender, customer.first_name + customer.last_name → fullname, book.last_update → timestamp**.

O processo de extração projetado relativo ao povoamento da tabela de factos **FTOrders,** a partir da fonte SakilaBooks, envolve tabelas de carregamento associadas à vista **orders_views** definida na fonte de informação envolvida. Sobre os dados que se obtêm a partir dessa vista aplicamos os

seguintes mapeamentos: **order_id → orderID, customer_id → customer, book_id → book, value → value, quantity → quantity, unit_price → bookPrice, unit_discount → bookDiscount, order_date → date, last_upatate → timestamp**. Também no caso do processo de extração de dados da fonte de informação EbookStore, foram utilizadas tabelas de carregamento para a vista **orders_view** definida na própria fonte, e povoadas com base nos seguintes mapeamentos: **order_id → orderID, user_id → customer, book_id → book, value → value, quantity → quantity, book_price → bookPrice, book_discount → bookDiscount, order_date → date, last_upatate → timestamp**.

4.5.2 Transformação dos Dados

A etapa de transformação de dados do processo de ETL envolve tarefas de DQE, DCI e SKP, cada uma delas escalonadas de acordo com o modelo estabelecido para o funcionamento do processo de ETL.

De uma forma geral, um processo de DQE utiliza uma tabela de auditoria, que contém os dados que devem ser validados, e uma tabela de quarentena, que acolhe os registos considerados inválidos. O DQE envolve um conjunto de tarefas de filtragem e de transformação de dados, que realizam operações como a validação de registos ou a manipulação de valores isolados – e.g. normalização de acrónimos ou correção de valores nulos.

Para a dimensão **Book**, a tarefa de DQE projetada (**Error! Reference source not found.**) realiza operações de filtragem de forma a excluir os registos da tabela de auditoria (**audit_dim_book**) que não tenham valores para o conjunto de atributos obrigatórios (**ISBN** e **title**). Os registos que forem excluídos são transferidos para a tabela de quarentena correspondente (**qua_dim_book**), com a inclusão do motivo da sua exclusão (atributo **title** em falta ou atributo **ISBN** em falta) de forma a poderem ser tratados posteriormente.

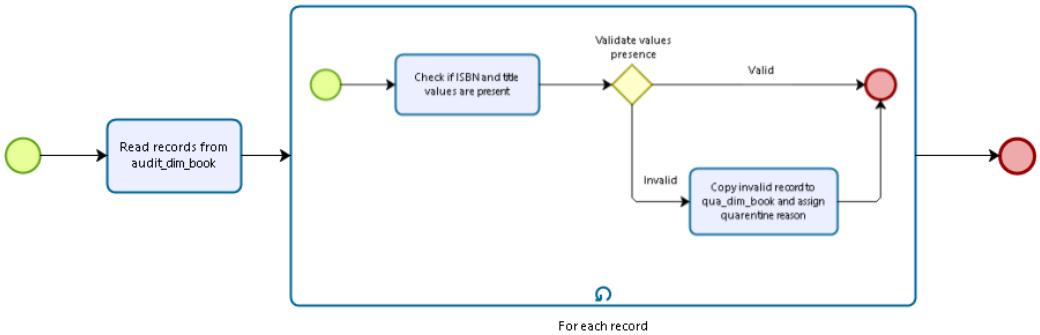


Figura 11 - Esquema em BPMN para DQE da dimensão Book

De forma equivalente, para o caso do povoamento da tabela de factos **TFOrders**, a tarefa DQE (Figura 12) cinge-se à aplicação de um filtro para exclusão de registos sem valores obrigatórios (neste caso, é requerida a existência de todos os valores dos atributos referentes a dimensões e de todas as medidas), bem como assegurar o cumprimento da integridade referencial dos factos.

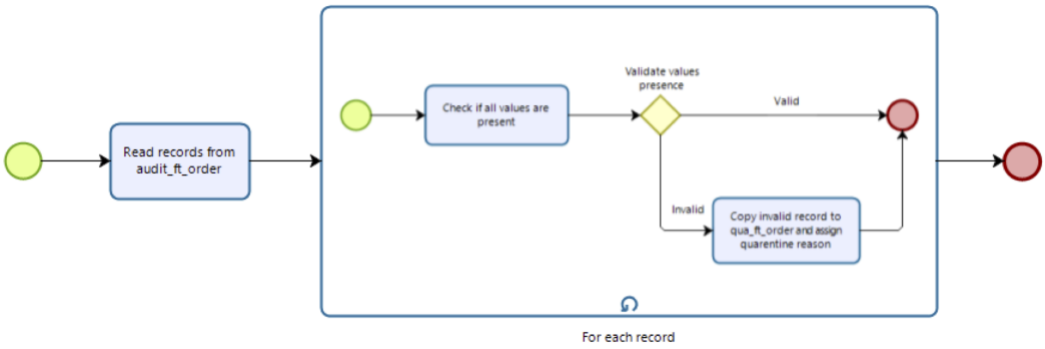


Figura 12 - Esquema em BPMN para DQE da tabela de factos FTOrders

Quanto à tarefa DQE para a dimensão **Customer** (**Error! Reference source not found.**), tal como o verificado nas tarefas DQE apresentadas anteriormente, esta realiza um conjunto de operações de filtragem e, complementarmente, uma operação de transformação de forma a fazer a padronização dos dados envolvidos. Os registos lidos a partir da tabela de auditoria (**audit_dim_customer**) são validados no sentido de verificar se os atributos **TIN** e **full_name** contêm valores não nulos e se o atributo **gender** contém valores válidos – 'F', 'M', 'D' (desconhecido). Os registos que forem considerados inválidos são adicionados à tabela de quarentena associada e retirados da tabela de

auditoria. Adicionalmente, os registos que foram considerados válidos são normalizados (se necessário) para garantir que cada nome de cliente apresenta o mesmo formato – basicamente, aplica-se uma operação de capitalização das palavras do valor do atributo **full_name**.

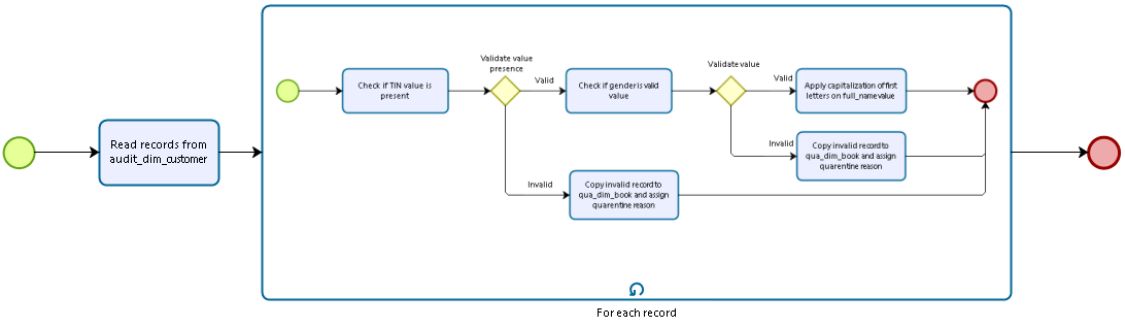


Figura 13 - Esquema em BPMN para DQE da dimensão Customer

Passemos agora à descrição das tarefas DCI. Como pudemos verificar anteriormente através do diagrama geral do sistema de ETL (Figura 9) existem várias tarefas DCI no processo projetado. O processo de povoamento das dimensões envolvidas está sujeito a tarefas de conciliação de dados, que são suportadas na sua operação por um conjunto de tabelas de equivalência. De uma forma geral, a conciliação dos dados traduz-se na atribuição de uma chave única (chave de substituição) para registos provenientes de diferentes fontes de informação que representem uma mesma entidade.

Na Figura 14 é possível analisar uma das tarefas DCI que está aplicada no processo de povoamento da dimensão **Book**. Neste processo, cada registo da tabela de auditoria (**audit_dim_book**) é atualizado de forma a usar como chave (valor do atributo **id**) uma chave de substituição em vez da sua chave natural – no mapeamento de um registo da tabela de auditoria numa chave de substituição foi utilizado o valor do atributo **ISBN**.

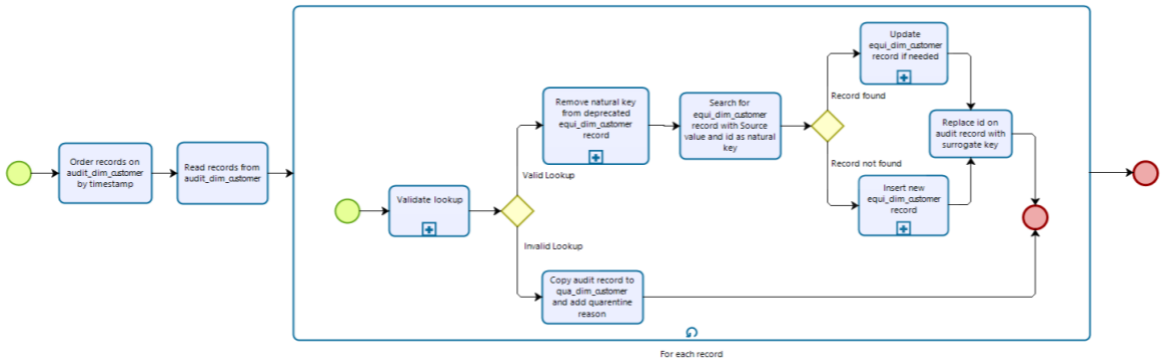


Figura 14 - Esquema em BPMN para DCI da dimensão Book

O processo de substituição de chaves para cada registo de auditoria teve que verificar também a validade da conciliação de dados a ser feita e fazer a manutenção da tabela de mapeamento de forma que esta se mantenha atualizada. Para tal, durante o processo, cada registo de auditoria é envolvido em três passos: 1) validação da operação de *lookup* a ser realizada; 2) atualização de registos da tabela de mapeamento descontinuados; e 3) inserção ou atualização de um registo na tabela de mapeamento. Neste processo, a validação da operação de *lookup* pretende verificar que não existem vários registos na tabela de equivalência com o mesmo valor de fonte e de campo de equivalência, mas com diferentes chaves naturais. Por exemplo, no caso apresentado, quando se encontra uma entrada na tabela de equivalência **equi_dim_book**, com os mesmos valores de **ISBN** e **source** do registo de auditoria, mas que não tenha o **id** deste como valor de **natural_key**, o registo presente na tabela de auditoria é assinalado como inválido – posteriormente será colocado em quarentena, com a indicação da ocorrência de um erro de conciliação (Figura 15).

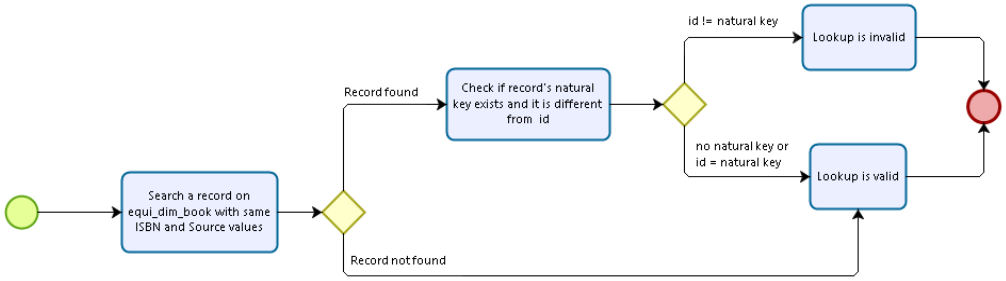


Figura 15 - Esquema em BPMN para validação da operação de *lookup* em DCI da dimensão Book

A atualização de registos descontinuados da tabela de mapeamento surge apenas em situações nas quais exista, para um registo em auditoria, uma entrada na tabela de mapeamento que tenha o seu **id** como valor de **natural_key** e que partilhe o mesmo valor de **source,** mas que tenha um valor diferente no campo de equivalência. Nestas situações, far-se-á a remoção da **natural_key** do registo da tabela de equivalência. No exemplo apresentado na Figura 16, procura-se um registo na tabela de equivalência **equi_dim_book** que tenha o mesmo valor de **source** do registo em auditoria e que tenha o seu **id** como valor do campo **natural_key**, mas diferente valor de **ISBN**. No caso de ser encontrado um registo com essas características, a sua **natural_key** é removida (colocando-se o seu valor a *null*).

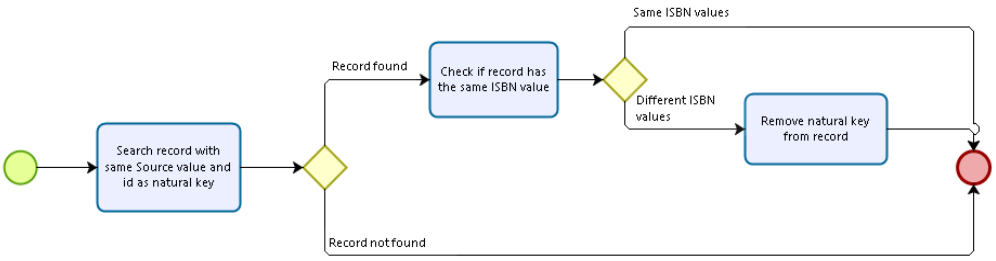


Figura 16 - Esquema em BPMN para atualização de registos descontinuados em DCI da dimensão Book

Por fim, insere-se ou atualiza-se, conforme o caso, o registo correto na tabela de mapeamento. Para isso, procura-se um registo na tabela de mapeamento que tenha os mesmos valores para **natural_key** e **source** do registo de auditoria. Caso este seja encontrado, se a operação do registo

de auditoria for uma remoção, então remove-se o valor de **natural_key** do registo encontrado. Caso contrário, insere-se um novo registo com os valores de **source** e campo de equivalência do registo de auditoria. Este novo registo terá como **surrogate_key** a chave de substituição existente para o valor de campo de equivalência ou um novo valor caso nenhum outro exista. Adicionalmente, se a operação do registo de auditoria for de inserção ou de atualização, o seu valor de **id** é tomado como **natural_key** do registo de equivalência. O processo de inserção de novos registos na tabela de mapeamento, para o caso da dimensão **Book**, pode ser observado na Figura 17.

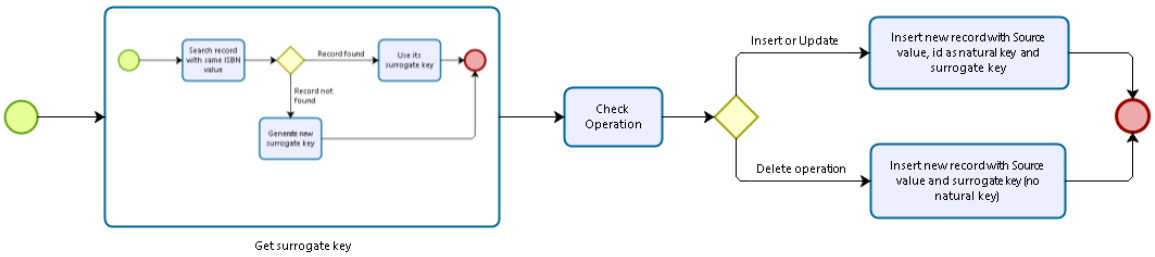


Figura 17 - Esquema em BPMN para inserção de novos registos de mapeamento em DCI da dimensão Book

De forma semelhante, para a tarefa DCI da dimensão **Customer** aplica-se o mesmo procedimento, utilizando-se a tabela de equivalência **equi_dim_customer**, e atualizando-se o valor do campo **id** de cada registo da tabela de auditoria **audit_dim_customer** com a chave de substituição que for encontrada. Para mapear um registo de auditoria numa chave de substituição utiliza-se o valor do atributo **TIN** (Figura 18).

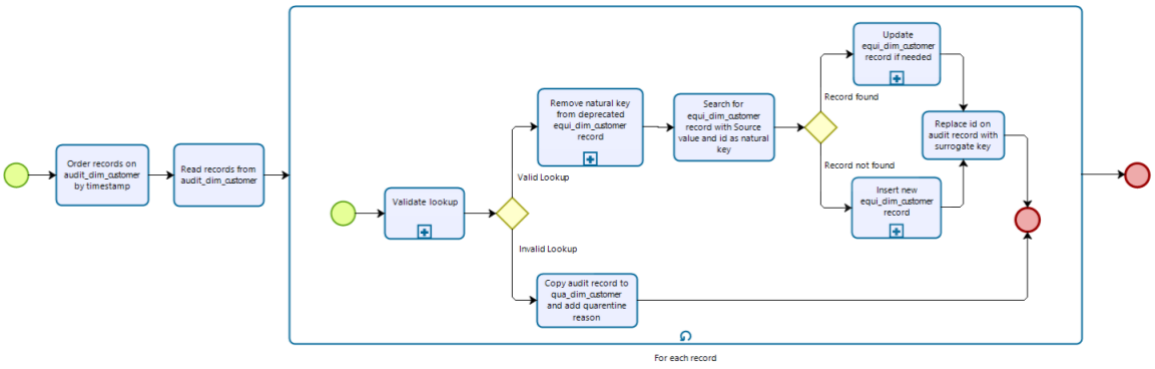


Figura 18 - Esquema em BPMN para DCI da dimensão Customer

Por fim, falta-nos abordar uma tarefa com algum relevo em todo o processo de povoamento do *data warehouse*: a tarefa *Surrogate Key Pipelining* (SKP). Esta é aplicada no processo de povoamento da tabela de factos **FTOrders**, no qual os registos da tabela de auditoria são atualizados de forma a serem atribuídas a respetivas chaves de substituição para os atributos relativos às dimensões **Book**, **Customer** e **Date** (Figura 19).

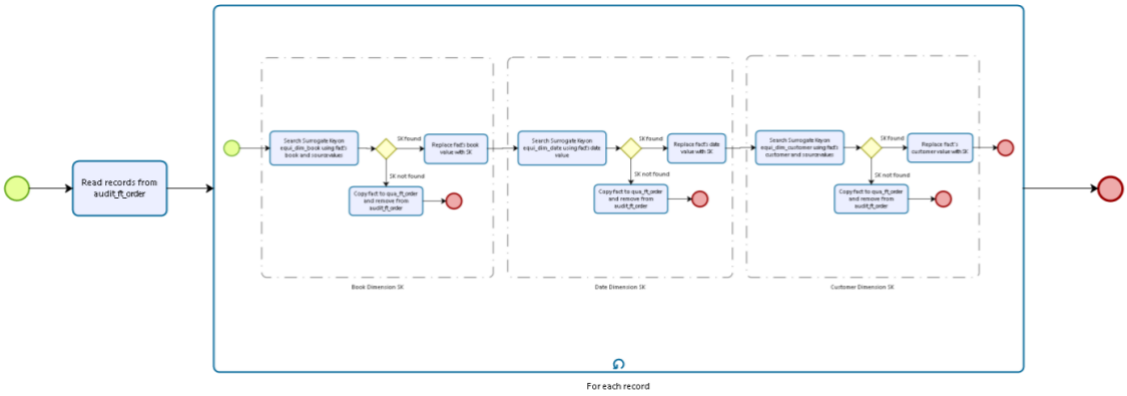


Figura 19 - Esquema BPMN para SKP na tabela de factos FTOrders

Assim, para fazer a atualização dos valores de dimensão de cada registo da tabela de auditoria (**audit_tf_order**), é utilizada a tabela de equivalência da dimensão em questão, o valor da sua chave natural e a fonte de informação da qual é proveniente (valor do atributo **source** do registo da tabela de auditoria dos factos). Para cada dimensão, procura-se uma chave de substituição na tabela de equivalência correspondente, que tenha uma chave natural referente à fonte de informação da qual o facto provém, utilizando-se o valor da dimensão desse facto. Caso seja encontrada uma chave de substituição, o facto é atualizado com esse valor, substituindo-se o valor da chave operacional anterior. No caso de não ser encontrada nenhuma chave de substituição, o facto é colocado em quarentena (na tabela **qua_ft_order**) e removido da tabela de auditoria.

4.5.3 Carregamento dos Dados

A etapa de carregamento dos dados no *data warehouse* é assegurada pelas tarefas de carregamento SCD e IDL. As tarefas de carregamento SCD são responsáveis por lidar com o carregamento de dados das dimensões com variação e sem manutenção de histórico (**Book** e **Customer**). Neste processo, os dados da tabela de auditoria, já transformados e prontos a integrar o *data warehouse*,

são carregados nas estruturas do *data warehouse* segundo a ordem temporal pela qual foram colocadas nas tabelas de auditoria de cada uma das dimensões. Este carregamento pode envolver registos novos ou registos atualizados, que correspondem aos registos de auditoria com referência a operações de inserção ou atualização, respetivamente. Depois, essas mesmas operações serão refletidas apropriadamente nas tabelas de dimensão em questão. Os dados são, assim, movimentados das tabelas **audit_dim_book** e **audit_dim_customer,** da área de retenção, para as tabelas de dimensão **dim_book** e **dim_customer,** do *data warehouse*.

Quanto às tarefas IDL, estas asseguram o carregamento intensivo de dados no *data warehouse*, e, no nosso caso, estão aplicadas no processo de povoamento da tabela de factos **FTOrders**. No nosso caso de estudo, assegurámos que todos os dados de dimensão relacionados com os factos já estão devidamente carregados quando uma tarefa IDL for iniciada. Assim, para o carregamento dos dados da tabela de factos, basta ter em consideração os dados na tabela de auditoria **audit_ft_order**. Os registos desta tabela – apenas referenciados com *tags* de operação de inserção – são carregados para a tabela de factos do *data warehouse* (**ft_order**) como se fosse executada uma operação *batch*.

Capítulo 5.

Especificação Formal do Sistema

5.1 Estratégias de Modelação

Antes de começarmos com a modelação do sistema em Alloy foram definidas algumas estratégias gerais. Em primeiro lugar, a abordagem de uma modelação genérica – utilizando objetos como linhas e tabelas abstratas – foi abandonada em prol de uma modelação específica do caso de estudo. Esta decisão baseou-se na necessidade de queremos uma modelação rápida, mais fácil de ler e com uma análise e interpretação mais diretas. Desta forma, pretendeu-se eliminar o ruído adicionado pela generalização do modelo, quer na própria modelação do sistema, quer na execução de comandos e na interpretação e visualização de resultados de análise.

Em segundo lugar, e no sentido de se obter uma modelação estruturada, os modelos definidos foram divididos em diferentes módulos Alloy, nomeadamente:

- um módulo para o modelo do *data warehouse* (**DW**);

- um módulo para o modelo da fonte SakilaBooks e da fonte EbookStore (respetivamente, **SakilaBooks** e **EbookStore**);

- um módulo para a área de retenção (**StagingArea**);

- um módulo com a modelação das tarefas ETL (**ETL**).

Adicionalmente, utilizámos também um módulo com modelações gerais, que é utilizado pelos restantes modelos – **DataTypes**.

A modelação do caso de estudo seguiu um dos princípios da modelação de sistemas e da própria linguagem Alloy, princípio este que dita que não se devem procurar modelos completos, mas sim modelos adequados ao propósito da modelação. Assim, excluímos do processo de modelação todos os objetos de dados e propriedades dos subsistemas sem interesse para o processo de ETL a modelar.

A modelação específica dos diversos elementos estruturais de cada subsistema (fontes de dados, área de retenção e *data warehouse*) foi realizada com a definição de uma assinatura para cada tabela de dados. Adicionalmente, uma vez que não foram utilizados objetos como "Campo", nem definidas relações que caracterizavam, de forma abstrata, uma tabela (como os seus campos e as suas chaves), a estrutura das linhas de uma tabela foi modelada por uma assinatura separada e específica. Esta assinatura define a estrutura básica de cada registo armazenado, traduzindo aspetos como o tipo de atributo e a sua anulabilidade. Veja-se, como exemplo, a modelação de uma tabela de dimensão **Book** apresentada na Figura 20.

```
1.  /*
2.   * Signature: DimBook
3.   */
4.  one sig DimBook  extends Relation {
5.     rows:  Book -> State
6.  }
7.
8.  /*
9.   * Signature: Book
10.  */
11. sig Book extends Row {
12.    id: one Int,
13.    cv: lone Int,
```

```
14.     title: one Text,
15.     description: one Text,
16.     ISBN: one Text
17. }
```

Figura 20 - Exemplo de uma modelação específica em Alloy para a tabela de dimensão Book

A modelação realizada para a tabela **Book** (Figura 20) pode ser interpretada da seguinte forma: **DimBook** é um *singleton* (um conjunto com apenas um elemento) que estende a assinatura **Relation**. Esta assinatura está envolvida numa relação de **rows** com elementos **Book**. A assinatura de **Book** define os seus atributos **id**, **cv**, **title**, **description** e **ISBN**. Destes atributos, os dois primeiros acolhem números inteiros e os restantes valores textuais. Adicionalmente, o atributo **cv** é anulável, isto é, pode tomar um valor nulo, que aqui foi representado pela ausência de valor relacionado - a relação **cv** é definida como **lone Int**. As definições estabelecidas significam que no sistema modelado existe um elemento **Relation DimBook**, com um conjunto de elementos **Book** associado em cada momento.

Por sua vez, a modelação do tempo foi realizada com a introdução de uma assinatura **State** e com a sua aplicação no modelo pela estratégia de *local state idiom* (discutida anteriormente na Secção 3.3.2). A evolução do sistema foi desenhada com base nas transições entre estados, sendo cada transição um predicado declarativo. Este predicado, e de acordo com o exposto na secção mencionada, especifica o estado anterior e o estado seguinte, de forma a representar a consequência da ação que representa.

De forma geral, a modelação do caso de estudo envolve a validação da propriedade de consistência das tabelas envolvidas. Assim, foi modelado também um predicado de consistência para cada tabela, e realizada a validação de que cada operação preserva a consistência das tabelas que manipula. Nesta modelação procurou-se fazer um balanço entre as restrições impostas ao modelo e as restrições (propriedades) a validar. Nomeadamente, a utilização de factos é evitada para que não haja sobre restrição do sistema. Os factos são utilizados, essencialmente, para impor restrições físicas específicas das tabelas (como a restrição de chave primária) e as restrições consequentes de alguns comportamentos específicos, como pode suceder com a positividade dos valores de um atributo auto incremental.

5.2 Modelação do Sistema

5.2.1 Modelação das Estruturas de Dados

A modelação do caso de estudo foi separada em módulos Alloy. Na modelação das estruturas de dados estão envolvidos os módulos **DataTypes** (módulo auxiliar), **SakilaBooks**, **EbookStore**, **DW** e **StagingArea**. O modelo de cada um dos subsistemas (fontes de dados, área de retenção e *data warehouse*) representa as várias tabelas envolvidas no processo de ETL, baseando-se no esquema lógico de cada entidade. Assim, e tendo em consideração as estratégias definidas anteriormente, para a modelação estrutural de uma tabela devem-se seguir, de forma geral, quatro passos: a definição da assinatura de tabela, a definição da assinatura de linha, a declaração de factos para garantia estrutural e, por fim, a especificação de um predicado para validar a sua consistência.

O Módulo DataTypes

O módulo **DataTypes** (Figura 21) define os objetos que são utilizados de forma transversal nos restantes modelos do sistema, circunstância que auxiliou bastante a sua modelação estrutural. Este módulo introduziu a assinatura **State** e aplicou uma ordenação global aos seus elementos, utilizando o módulo **ordering**. Foram também definidas duas assinaturas abstratas – **Relation** e **Row** –, para serem estendidas pelas assinaturas de tabelas e linhas específicas do sistema. Por fim, definiram-se outros tipos de dados para valores de atributos, nomeadamente **Text**, **Float**, **Datetime**, **Year**, **Month**, **Bool**, **SourceEnum** e **GenderEnum**. As últimas assinaturas foram definidas como **enum**, isto é, conjuntos com elementos específicos, tais como **SakilaBooks** e **EbookStore** da assinatura **SourceEnum**. De realçar, ainda, a utilização de uma ordenação global nos elementos da assinatura **Datetime**, através do módulo **ordering**. Esta aplicação permite a manipulação destes elementos enquanto registos temporais, simulando uma ordem cronológica entre estes.

```
1.  open util/ordering[State] as S
2.  open util/ordering[Datetime] as DO
3.  open util/integer
4.
5.  sig State {}
6.
7.  abstract sig Row {}
8.  abstract sig Relation {}
```

```
9.
10.  sig Text {}
11.  sig Float {}
12.  sig Datetime {}
13.  sig Year {}
14.  sig Month {}
15.
16.  enum Bool { True, False }
17.  enum GenderEnum { Female, Male, Unknown }
18.
19.  enum OpEnum { I, U, D }
20.  enum SourceEnum { SakilaBooks, EbookStore }
```

Figura 21 - Especificação do módulo DataTypes

O Módulo SakilaBooks

O módulo **SakilaBooks** representa uma fonte de dados (SakilaBooks), modelando as relações de interesse para o sistema em estudo. Esta modelação foi realizada utilizando o módulo auxiliar **DataTypes** e definindo-se as assinaturas de relação e de linha, bem como os factos e predicados associados a estes. As tabelas desta fonte foram modeladas pelas assinaturas *singleton* **Books**, **Customers**, **Orders**, **OrderItems** e **Inventories,** que estendem a assinatura **Relation**. Adicionalmente, foi também definida a assinatura *singleton* **OrdersView**, representando uma vista que unifica, de forma mais normalizada, a informação sobre a venda de um livro. A modelação das assinaturas referidas está apresentada na Figura 22.

```
1.   /* Books table */
2.   one sig Books extends Relation { rows:  Book -> State }
3.   /* Customers table */
4.   one sig Customers extends Relation { rows: Customer -> State }
5.   /* Orders table */
6.   one sig Orders extends Relation { rows: Order -> State }
7.   /* OrderItems table */
8.   one sig OrderItems extends Relation { rows: OrderItem -> State }
9.   /* Inventories table */
10.  one sig Inventories extends Relation { rows: Inventory -> State }
11.  /* OrdersView */
12.  one sig OrdersView extends Relation { rows: JoinedOrderData -> State }
```

Figura 22 - Modelo em Alloy das relações de SakilaBooks

Cada assinatura de relação utiliza uma assinatura de linha correspondente, nomeadamente **Book**, **Customer**, **Order**, **OrderItem**, **Inventory** e **JoinedOrderData**. As assinaturas de linha definem os atributos do objeto de dados que é armazenado, utilizando assinaturas definidas em **DataTypes** e características nativas da Alloy. Por exemplo, um atributo que pode tomar o valor *null* é definido como uma relação com **lone** (zero ou um) elementos. Assim, valores *null* são representados pela ausência de elemento relacionado. A modelação de cada uma das entidades consideradas pode ser observada na Figura 23.

```
1.  /* Book entity */
2.  sig Book extends Row {
3.      id: one Int,
4.      ISBN: one Text,
5.      title: lone Text,
6.      description: lone Text,
7.      first_edition_year: lone Year,
8.      language_id: lone Int,
9.      original_language_id: lone Int,
10.     price: lone Float,
11.     last_update: lone Datetime
12. } { pos[id] }
13.
14. /* Customer entity */
15. sig Customer extends Row {
16.     id: one Int,
17.     tin: lone Text,
18.     first_name: lone Text,
19.     last_name: lone Text,
20.     email: lone Text,
21.     address_id: lone Int,
22.     active: one Bool,
23.     create_date: lone Datetime,
24.     last_update: lone Datetime
25. } { pos[id] }
26.
27. /* Order entity */
28. sig Order extends Row {
29.     id: one Int,
30.     order_date: lone Datetime,
31.     customer_id: one Int,
32.     staff_id: lone Int,
33.     last_update: lone Datetime
34. } { pos[id] }
35.
36. /* OrderItem entity */
37. sig OrderItem extends Row {
38.     id: one Int,
```

```
39.     order_id: one Int,
40.     inventory_id: one Int,
41.     unit_price: one Int,
42.     unit_discount: one Int,
43.     quantity: one Int,
44.     last_update: lone Datetime
45.  } { pos[id] }
46.
47.  /* Inventory entity */
48.  sig Inventory extends Row {
49.     id: one Int,
50.     book_id: one Int,
51.     store_id: one Int,
52.     last_update: lone Datetime
53.  } { pos[id] }
54.
55.  /* JoinedOrderData */
56.  sig JoinedOrderData extends Row {
57.     order_item_id: one Int,
58.     order_id: one Int,
59.     customer_id: lone Int,
60.     book_id: lone Int,
61.     quantity: one Int,
62.     value: one Int,
63.     unit_price: one Int,
64.     unit_discount: one Int,
65.     order_date: lone Datetime,
66.     last_update: lone Datetime
67.  }
```

Figura 23 - Modelo em Alloy das entidades de SakilaBooks

A modelação de cada tabela envolveu, ainda, a declaração de um bloco de factos para expressar algumas das suas restrições físicas, nomeadamente **constraintsBooks**, **constraintsCustomers**, **constraintsOrders**, **constraintsOrderItems** e **constraintsInventories**, e a definição de um predicado de consistência (predicado **consistent** que recebe como primeiro argumento implícito a relação à qual se refere – por exemplo: **Books.consistent**). Uma vez que esta fonte de dados garante a sua integridade referencial utilizando restrições de chave estrangeira nas tabelas desejadas, esta restrição foi expressa enquanto um facto. Os predicados de consistência das tabelas deste modelo são, frequentemente, desnecessários, sendo definidos enquanto predicados vazios. Este comportamento pode ser analisado na Figura 24, na qual é apresentada a modelação dos factos e predicado referentes à tabela com informação de livro.

```
1.  /*
2.   * Fact: constraintsBooks
3.   * Books table constraints - involves the definition of id as the primary key and the ISBN unique index.
4.   */
5.  fact constraintsBooks
6.  {
7.      -- Primary key is id
8.      all s: State, b1, b2: (Books.rows).s | b1.id = b2.id implies b1 = b2
9.      -- Unique index: ISBN
10.     all s: State, b1, b2: (Books.rows).s | b1.ISBN = b2.ISBN implies b1 = b2
11. }
12.
13. /*
14.  * Predicate Books.consistent
15.  * Consistency properties of Books that must hold in the received state.
16.  * No consistency property is defined.
17.  */
18. pred Books.consistent[s: State]
19. {}
```

Figura 24 - Modelo em Alloy de factos e predicado de consistência para a relação Books de SakilaBooks

Na Figura 24, a declaração do facto **constraintsBooks** define o atributo **id** como chave primária de cada elemento **Book** nas linhas de **Books** e simula o índice único no atributo **ISBN**. O predicado de consistência de **Books** num determinado estado é vazio, o que significa que é sempre verdade – a consistência da tabela **Books** é garantida pela sua estrutura e pelas restrições aplicadas.

Quanto à modelação da vista **OrdersView** desta fonte, em adição à definição da assinatura de **Row JoinedOrderData**, foi utilizada uma declaração de factos para expressar a natureza do conteúdo desta relação. O conteúdo desta vista é uma seleção dos dados da tabela **OrderItems**, unidos com a tabela **Orders** por **order_id** para selecionar o identificador do cliente e a data da encomenda, e com a tabela **Inventories** por **inventory_id,** para selecionar o identificador do livro associado.

```
1.  /*
2.   * OrdersView definition.
3.   */
4.  fact ordersView
5.  {
6.      -- OrdersView contains all data from OrderItems joined with Orders and Inventories tables
7.      all s: State |
8.          all joined_o: (OrdersView.rows).s |
9.              -- Select from OrderItems
```

```
10.          one oi : (OrderItems.rows).s | joined_o.order_item_id = oi.id and
11.             joined_o.unit_price = oi.unit_price and joined_o.unit_discount = oi.unit_discount and
12.             joined_o.value = oi.quantity.mul[oi.unit_price.sub[oi.unit_discount]] and
13.             joined_o.quantity = oi.quantity and joined_o.last_update = oi.last_update and
14.             joined_o.order_id = oi.order_id and
15.             -- Join with Orders to fetch customer_id and order_date
16.             (
17.                ( one o: (Orders.rows).s | oi.order_id = o.id )
18.                implies ( one o: (Orders.rows).s | oi.order_id = o.id and joined_o.customer_id = o.customer_id and
    joined_o.order_date = o.order_date)
19.                else ( no joined_o.customer_id and no joined_o.order_date )
20.             ) and
21.             -- Join with Inventory to fetch book_id
22.             (
23.                ( one i: (Inventories.rows).s | oi.inventory_id = i.id )
24.                implies ( one i: (Inventories.rows).s | oi.inventory_id = i.id and joined_o.book_id = i.book_id)
25.                else ( no joined_o.book_id )
26.             )
27.
28.       -- All OrderItems data is selected in OrdersView (joined with Orders and Inventories)
29.       all s: State |
30.          all oi: (OrderItems.rows).s |
31.             -- Selected from OrderItems
32.             one joined_o: (OrdersView.rows).s | joined_o.order_item_id = oi.id and
33.                joined_o.unit_price = oi.unit_price and joined_o.unit_discount = oi.unit_discount and
34.                joined_o.quantity = oi.quantity and joined_o.last_update = oi.last_update and
35.                joined_o.order_id = oi.order_id and
36.                -- Joined with Orders and selected customer_id and order_date (or no values)
37.                (
38.                   ( one o: (Orders.rows).s | oi.order_id = o.id )
39.                   implies ( one o: (Orders.rows).s | oi.order_id = o.id and joined_o.customer_id = o.customer_id and
    joined_o.order_date = o.order_date)
40.                   else ( no joined_o.customer_id and no joined_o.order_date )
41.                ) and
42.                -- Joined with Inventories and selected book_id (or no value)
43.                (
44.                   ( one i: (Inventories.rows).s | oi.inventory_id = i.id )
45.                   implies ( one i: (Inventories.rows).s | oi.inventory_id = i.id and joined_o.book_id = i.book_id)
46.                   else no joined_o.book_id
47.                )
48.    }
```

Figura 25 - Modelo em Alloy para definição da vista OrdersView de SakilaBooks

Assim, o facto definido (ver Figura 25) dita que, para todos os estados e para todos as entradas na vista **OrdersView** nesse estado, existe uma entrada na tabela **OrderItems,** com os mesmos valores de **order_item_id**, **order_id**, **last_update**, **unit_price**, **unit_discount** e **quantity**. O

valor de **value** da entrada da vista é o resultado de multiplicar o preço unitário menos o desconto pela quantidade de livros. Foi estabelecido, também, que caso exista uma entrada na tabela de **Orders** com o mesmo **order_id**, a entrada da vista partilha o seu valor de **customer_id** e de **order_date**. Caso contrário, a entrada da vista não tem valor para esses atributos. De forma semelhante, caso exista uma entrada na tabela **Inventories** com o mesmo **inventory_id** que a entrada da vista, esta partilha o seu valor de **book_id**; caso contrário não tem nenhum valor de **book_id**. O facto foi reforçado com a propriedade de que, para todos os estados, todos os dados da relação **OrdersItems** nesse estado estão presentes, isto é, são selecionados, por uma entrada na vista **OrdersView,** sendo os valores de **book_id**, **customer_id** e **order_date** desta entrada lidos da tabela **Inventories** e **Orders** da forma já descrita.

Por fim, foi definido um predicado **consistentSakila**, que recebe um estado como argumento e valida a consistência de todas as relações definidas no módulo no estado recebido.

O Módulo EbookStore

O módulo **EbookStore** representa a fonte de dados **Ebookstore** e modela as relações de interesse para o sistema em estudo. Esta fonte foi modelada com recurso ao módulo **DataTypes** e baseia-se na definição de assinaturas de relação e de linha, e dos factos e predicados associados a cada uma. As tabelas desta fonte foram modeladas pelas assinaturas *singleton* **Books**, **Users**, **Orders** e **OrderLines,** que estendem a assinatura **Relation** (Figura 26). Tal como no módulo SakilaBooks, foi também modelada uma vista para recolha dos dados relativos a encomendas. Para isso, é introduzida a assinatura *singleton* **OrdersView,** que também estende **Relation**.

```
1.  /* Books table */
2.  one sig Books extends Relation { rows:  Book -> State }
3.  /* Users table */
4.  one sig Users extends Relation { rows: Customer -> State }
5.  /* Orders table */
6.  one sig Orders extends Relation { rows: Order -> State }
7.  /* OrderLines table */
8.  one sig OrderLines extends Relation { rows: OrderLine -> State }
9.  /* OrdersView */
10. one sig OrdersView extends Relation { rows: JoinedOrderData -> State }
```

Figura 26 - Modelo em Alloy das relações de EbookStore

A modelação desta fonte de dados utiliza as assinaturas de linha **Book**, **User**, **Order**, **OrderLine** e **JoinedOrderData**. Estas assinaturas definem a estrutura básica do objeto de dados armazenado, representando aspetos como a anulabilidade e o tipo de dados dos seus atributos. A modelação de cada uma destas entidades pode ser analisada na Figura 27.

```
1.  /* Book entity */
2.  sig Book extends Row {
3.      id: one Int,
4.      title: lone Text,
5.      summary: lone Text,
6.      category_id: lone Int,
7.      price: lone Int,
8.      ISBN: one Text,
9.      last_update: lone Datetime
10. } { pos[id] }
11.
12. /* User entity */
13. sig User extends Row {
14.     id: one Int,
15.     tin: lone Text,
16.     first_name: lone Text,
17.     last_name: lone Text,
18.     gender: one GenderEnum,
19.     birthdate: lone Datetime,
20.     email: lone Text,
21.     password: one Text,
22.     active: one Bool,
23.     create_date: lone Datetime,
24.     last_update: lone Datetime
25. } { pos[id] }
26.
27. /* Order entity */
28. sig Order extends Row {
29.     id: one Int,
30.     code: one Text,
31.     user_id: one Int,
32.     order_total: one Int,
33.     order_date: lone Datetime,
34.     last_update: lone Datetime
35. } { pos[id] }
36.
37. /* OrderLine entity */
38. sig OrderLine extends Row {
39.     id: one Int,
40.     order_id: one Int,
41.     book_id: one Int,
42.     quantity: one Int,
43.     book_price: one Int,
```

```
44.       book_discount: one Int
45.   } { pos[id] }
46.
47.   /* JoinOrderData */
48.   sig JoinedOrderData extends Row {
49.       order_line_id: one Int,
50.       order_id: one Int,
51.       user_id: lone Int,
52.       book_id: one Int,
53.       quantity: one Int,
54.       book_price: one Int,
55.       book_discount: one Int,
56.       value: one Int,
57.       order_date: lone Datetime,
58.       last_update: lone Datetime
59.   }
```

Figura 27 - Modelo em Alloy das entidades de EbookStore

Para a modelação das tabelas da fonte de dados, foram, ainda, utilizados factos com as suas restrições físicas adicionais e um predicado para verificar a sua consistência. Os factos definidos para cada tabela (em **constraintsBooks**, **constraintsUsers**, **constraintsOrders** e **constrainstOrderLines**) refletem apenas a definição de índice de chave primária e índice único (para o atributo **ISBN** das entradas de **Books**). Os predicados de consistências definidos para cada tabela, por outro lado, permitem validar algumas propriedades, bem como a integridade referencial, quando adequado, nomeadamente, a integridade da relação **Orders** e da relação **OrderLines**.

```
1.    /*
2.     * Fact: constraintsOrderLines
3.     * OrderLines table constraints - involves the definition of id as the primary key.
4.     */
5.    fact constraintsOrderLines
6.    {
7.        -- Primary keys is id
8.        all s: State, oi1, oi2: (OrderLines.rows).s | oi1.id = oi2.id implies oi1=oi2
9.    }
10.
11.   /*
12.    * Predicate: OrderLines.consistent
13.    * Consistency properties of OrderLines that must hold in the received state.
14.    * Referencial integrity properties are included in consistency of OrderLines table
15.    * (relation with Orders and Books tables).
16.    */
17.   pred OrderLines.consistent[s: State]
18.   {
```

```
19.     -- Foreign key to Orders
20.     all oi: (OrderLines.rows).s | oi.order_id in ((Orders.rows).s).id
21.     -- Foreign key to Books
22.     all oi: (OrderLines.rows).s | oi.book_id in ((Books.rows).s).id
23.     -- Positive quantities
24.     all oi: (OrderLines.rows).s | pos[oi.quantity] and nonneg[oi.book_discount] and oi.book_price > oi.book_discount
25. }
```

Figura 28 - Modelo em Alloy de factos e predicado de consistência para a relação OrderLines de EbookStore

A modelação apresentada na Figura 28 refere-se à tabela **OrderLines,** estabelecendo a sua chave primária **id** como facto em **constraintsOrderLines** e validando a sua integridade referencial e a positividade dos seus valores monetários no predicado **OrderLines.consistent**. A restrição de chave estrangeira do atributo **order_id**, por exemplo, foi expressa pela propriedade de, para todas as linhas de **OrderLines**, o valor de **order_id** está presente no conjunto de valores **id** das linhas de **Orders** no estado recebido como argumento do predicado de consistência (ver linha 20 da Figura 28).

Para a modelação da vista **OrdersView** desta fonte, em adição à definição da assinatura de **Row JoinedOrderData**, foi utilizada uma declaração de factos para expressar a natureza do conteúdo desta relação. O conteúdo desta vista é uma seleção dos dados da tabela **OrderLines**, unidos com a tabela **Orders** por **order_id** para selecionar o identificador do cliente e a data da encomenda.

```
1.  /*
2.   * OrdersView definition.
3.   */
4.  fact ordersView
5.  {
6.      -- OrdersView contains data from OrderLines joined with Orders table
7.      all s: State |
8.          all joined_o: (OrdersView.rows).s |
9.              -- Select from OrderLine
10.             one oi : (OrderLines.rows).s | joined_o.order_line_id = oi.id and
11.                 joined_o.book_price = oi.book_price and joined_o.book_discount = oi.book_discount and
12.                 joined_o.value = oi.quantity.mul[oi.book_price.sub[oi.book_discount]] and
13.                 joined_o.quantity = oi.quantity and joined_o.last_update = oi.last_update and
14.                 joined_o.book_id = oi.book_id and joined_o.order_id = oi.order_id and
15.                 -- Join with Orders to fetch user_id and order_date
16.                 (
17.                     one o: (Orders.rows).s | oi.order_id = o.id )
18.                     implies ( one o: (Orders.rows).s | oi.order_id = o.id and joined_o.user_id = o.user_id
```

```
19.  and joined_o.order_date = o.order_date )
20.                  else ( no joined_o.user_id and no joined_o.order_date )
21.              )
22.
23.     -- All OrderLines data is selected in OrdersView (joined with Orders table)
24.     all s: State |
25.       all oi: (OrderLines.rows).s |
26.         -- Selected from OrderLine
27.         one joined_o: (OrdersView.rows).s | joined_o.order_line_id = oi.id and
28.           joined_o.book_price = oi.book_price and joined_o.book_discount = oi.book_discount and
29.           joined_o.quantity = oi.quantity and joined_o.last_update = oi.last_update and
30.           joined_o.book_id = oi.book_id and joined_o.order_id = oi.order_id and
31.           -- Joined with Orders and selected order_date and user_id (or no values)
32.            (
33.              ( one o: (Orders.rows).s | oi.order_id = o.id )
34.              implies ( one o: (Orders.rows).s | oi.order_id = o.id and joined_o.user_id = o.user_id
35.  and joined_o.order_date = o.order_date )
36.              else ( no joined_o.user_id and no joined_o.order_date )
37.           )
38.  }
```

Figura 29 - Modelo em Alloy para definição da vista OrdersView de EbookStore

O facto definido (ver Figura 29), à semelhança da abordagem seguida na definição do conteúdo da vista da fonte SakilaBooks, dita que, para todos os estados e para todos as entradas na vista **OrdersView** nesse estado, existe uma entrada na tabela **OrderLines** com os mesmos valores de **order_line_id**, **order_id**, **last_update**, **book_price**, **book_discount** e **quantity**. O valor de **value** da entrada da vista é o resultado da multiplicação do preço unitário menos o desconto pela quantidade de exemplares. Foi estabelecido, também, que caso exista uma entrada na tabela de **Orders,** com o mesmo **order_id**, a entrada da vista partilha o seu valor de **user_id** e de **order_date**; caso contrário, a entrada da vista não tem valor para esses atributos. Ao facto foi, ainda, adicionada uma restrição para que não falte dados na seleção de conteúdo feita — os dados de **OrderLines** são todos selecionados e os valores de **user_id** e **order_date** são selecionados a partir de uma entrada de **Orders** correspondente, da forma já descrita anteriormente.

O módulo define, ainda, o predicado **consistentEbookStore**, que recebe um estado como argumento, e valida a consistência das relações no estado recebido.

O Módulo DW

Como o próprio nome indica, este módulo modela o *data warehouse* do caso de estudo. À semelhança dos modelos realizados para as diversas fontes de dados, este módulo utiliza **DataTypes** para auxiliar na modelação das suas tabelas. De acordo com a modelação dimensional realizada, foram definidas as assinaturas das relações **FTOrders**, **DimBook**, **DimCustomer** e **DimDate** (Figura 30). Cada assinatura de tabela estende **Relation** e define uma relação **rows** com elementos de uma assinatura de linha específica.

```
1.  /* FTOrders table */
2.  one sig FTOrders extends Relation { rows:  Order -> State }
3.  /* DimBook table */
4.  one sig DimBook extends Relation { rows: Book -> State }
5.  /* DimCustomer table */
6.  one sig DimCustomer extends Relation { rows: Customer -> State }
7.  /* DimDate table */
8.  one sig DimDate extends Relation { rows: Date -> State }
```

Figura 30 - Modelo em Alloy das tabelas de DW

As assinaturas de linha definem os atributos do objeto de dados que é armazenado, estabelecendo o seu tipo e anulabilidade. As assinaturas definidas são **Order**, **Book**, **Customer** e **Date** (Figura 31). Estas assinaturas foram definidas da forma mais liberal possível, assumindo a maioria dos atributos como anuláveis, exceto pela relação de dimensão temporal.

```
1.  /* Order entity */
2.  sig Order extends Row {
3.     -- measures
4.       value: lone Int,
5.       quantity: lone Int,
6.       bookPrice: lone Int,
7.       bookDiscount: lone Int,
8.       -- dimensions
9.       book: lone Int,
10.      customer: lone Int,
11.      date: lone Int,
12.      -- deg. dimensions
13.      orderID: lone Int,
14.      source: lone SourceEnum
15. }
16.
17. /* Book entity */
```

```
18.  sig Book extends Row {
19.      id: one Int,
20.      title: lone Text,
21.      description: lone Text,
22.      ISBN: lone Text
23.  } { pos[id] }
24.
25.  /* Customer entity */
26.  sig Customer extends Row {
27.      id: one Int,
28.      gender: lone GenderEnum,
29.      fullName: lone Text,
30.      tin: lone Text
31.  } { pos[id] }
32.
33.  /* Date entity */
34.  sig Order extends Row {
35.      id: one Int,
36.      dayOfMonth: one Int,
37.      month: one Month,
38.      semester: one Semester,
39.      year: one Year
40.  } { pos[id] and pos[dayOfMonth] }
```

Figura 31 - Modelo em Alloy das entidades de DW

A modelação das tabelas utiliza factos e o predicado de consistência. No entanto, poucas restrições são aplicadas enquanto factos. A abordagem para a modelação do *data warehouse*, uma vez que se pretende validar a eficácia e sucesso do processo de ETL no sentido deste ser capaz de garantir o povoamento de dados com qualidade e integridade, oferece uma maior liberdade ao sistema de *data warehouse,* permitindo validar a sua consistência através de predicados específicos. A dimensão temporal foi modelada, essencialmente, através de restrições de factos, uma vez que o seu carregamento não é englobado no caso de estudo. Assim, os factos definidos para as tabelas de **FTOrders**, **DimBook** e **DimCustomer** definem, apenas, as suas chaves primárias (conjunto de chaves de dimensão e atributos **id**, correspondentemente). Por seu lado, a relação **DimDate** tem associado o facto para restrição de chave primária e de índice único no conjunto de atributos **dayOfMonth**, **month** e **year**. Neste seguimento, o predicado de consistência das tabelas valida, essencialmente, a presença de valores em todos os atributos, a integridade referencial e restrições adicionais que se pretende garantir nos seus dados. Por exemplo, na Figura 32, está apresentada a modelação dos factos e predicado de consistência para a tabela de dimensão **Book**. O facto **constraintsDimBook**, tal como referido, aplica apenas a restrição de chave primária para o atributo

id. O predicado de consistência envolve a definição de valores de **title**, **description** e **ISBN**, bem como a unicidade de valores de **ISBN**.

```
1.  /*
2.   * Fact: constraintsDimBook
3.   * DimBook table constraints - involves the definition of id as primary key.
4.  */
5.  fact constraintsDimBook
6.  {
7.      -- Primary keys as id
8.      all s: State, b1,b2: (DimBook.rows).s | b1.id = b2.id implies b1 = b2
9.  }
10.
11. /*
12.  * Predicate: DimBook.consistent
13.  * Consistency properties of DimBook that must hold in the received state.
14.  * DimBook table is consistent when all of its values are defined and the uniqueness
15.  * of ISBN attribute is checked.
16.  */
17. pred DimBook.consistent[s: State]
18. {
19.     -- Defined values
20.     all b: (DimBook.rows).s | one b.title and one b.description and one b.ISBN
21.     -- Unique ISBN
22.     all b1,b2: (DimBook.rows).s | b1.ISBN = b2.ISBN implies b1 = b2
23. }
```

Figura 32 - Modelo em Alloy de factos e predicado de consistência para a relação DimBook de DW

Este módulo incluí, ainda, o predicado **consistentDW**, que conjuga a consistência das tabelas modeladas no estado recebido, e o predicado **frame,** que garante que as tabelas mantêm as mesmas linhas nos dois estados recebidos.

O Módulo StagingArea

O módulo **StagingArea** representa a área de retenção do sistema em estudo, modelando as estruturas de dados (tabelas) presentes nesta área, bem como um conjunto de predicados de carácter funcional. Uma vez que a área de retenção atua como um meio de comunicação entre as fontes de dados e o *data warehouse*, precisando como tal de se conectar aos dois, os módulos **SakilaBooks**, **EbookStore** e **DW** são importados, em adição ao módulo auxiliar **DataTypes**. Este

modelo incluí as tabelas de auditoria, de quarentena, de equivalência e de controlo da área de retenção, tendo sido definida uma assinatura *singleton* que estende **Relation** para cada uma delas.

Modelação de Tabelas de Auditoria

As tabelas de auditoria foram modeladas pelas assinaturas **AuditBooks**, **AuditCustomers** e **AuditOrders** (ver Figura 33). Estas utilizam assinaturas de linha específicas que caracterizam os dados armazenados em cada tabela.

```
1.  /** Auditory **/
2.  /* AuditBooks table */
3.  one sig AuditBooks extends Relation { rows:  AuditBook -> State }
4.  /* AuditCustomers table */
5.  one sig AuditCustomers extends Relation { rows: AuditCustomer -> State }
6.  /* AuditOrders table */
7.  one sig AuditOrders extends Relation { rows: AuditOrder -> State }
```

Figura 33 - Modelo em Alloy das tabelas de auditoria de StagingArea

Cada linha de uma tabela de auditoria de dimensão contém atributos de dados para armazenar a informação do registo, e atributos de controlo. Estes incluem atributos para armazenar: a operação associada à alteração de dados em auditoria (pelo atributo **op**), a fonte da qual a alteração de dados é proveniente (no atributo **source**) e uma marca temporal referente ao momento do registo da alteração dos dados (atributo **timestamp**). Em adição a estes, é definido um atributo de controlo para cada atributo de dados, que indica se este foi, ou não, alterado. Estes atributos de controlo tomam valores booleanos e são designados pela composição do nome do atributo de dados ao qual se referem e o sufixo **_changed** (por exemplo: **name_changed**). As assinaturas de linha capturam estes atributos — a sua modelação está apresentada na Figura 34. De referir que a tabela de auditoria de factos (**AuditOrders**) contem apenas atributos para armazenar a informação do registo (facto), uma vez que, de acordo com o processo modelado, factos são inalteráveis e refletem sempre operações de inserção.

```
1.  /* AuditBook entity */
2.  sig AuditBook extends Row {
3.      id: one Int,
4.      -- Data attributes
5.      title: lone Text,
6.      ISBN: lone Text,
```

```
7.      description: lone Text,
8.      -- Control attributes
9.      title_changed: one Bool,
10.     description_changed: one Bool,
11.     ISBN_changed: one Bool,
12.     timestamp: one Datetime,
13.     op: one OpEnum,
14.     source: one SourceEnum
15. }
16.
17. /* AuditCustomer entity */
18. sig AuditCustomer extends Row {
19.     id: one Int,
20.     -- Data attributes
21.     fullname: lone Text,
22.     gender: lone GenderEnum,
23.     tin: lone Text,
24.     -- Control attributes
25.     fullname_changed: one Bool,
26.     gender_changed: one Bool,
27.     tin_changed: one Bool,
28.     timestamp: one Datetime,
29.     op: one OpEnum,
30.     source: one SourceEnum
31. }
32.
33. /* AuditOrder entity */
34. sig AuditOrder extends Row {
35.     -- Data attributes (natural keys)
36.     orderID: lone Int,
37.     customer: lone Int,
38.     book: lone Int,
39.     date: lone (Datetime + Int),
40.     source: one SourceEnum,
41.     -- Data attributes (measures)
42.     value: lone Int,
43.     quantity: lone Int,
44.     bookPrice: lone Int,
45.     bookDiscount: lone Int,
46.     -- Control attributes
47.     timestamp: one Datetime,
48.     op: one OpEnum
49. }
```

Figura 34 - Modelo em Alloy das entidades de auditoria de StagingArea

Associado a cada tabela de auditoria foi definido um facto que define a chave primária de cada tabela como o conjunto de todos os atributos da assinatura de linha (**constraintsAuditBooks**,

constraintsAuditCustomers e **constraintsAuditOrders**). Os predicados de consistência para as tabelas de auditoria de dimensão (**AuditBooks.consistent** e **AuditCustomers.consistent**) validam a consistência entre o valor do atributo de operação e o valor dos atributos de controlo. Por sua vez, esta é validada pela restrição de que uma operação de inserção implica que todos os atributos sejam marcados como alterados, e que uma operação de atualização ou de inserção é equivalente a algum dos atributos estar marcado como alterado. O predicado de consistência para a tabela de auditoria de factos (**AuditOrders.consistent**) é vazio – a consistência é sempre válida.

À modelação de cada tabela de auditoria foi adicionada um predicado de validade. Este predicado pretende validar por completo as propriedades de qualidade que se pretendem garantir nos dados antes destes serem carregados para o *data warehouse*. Assim, em adição aos predicados de consistência, foram também implementados os predicados de **AuditBooks.valid**, **AuditCustomers.valid** e **AuditOrders.valid**, que validam a presença de valores dos atributos nas entradas das tabelas. Adicionalmente, a validade das tabelas de auditoria de dimensão incluí, ainda, a integridade da operação que está a ser carregada — por exemplo, um registo de atualização ou remoção de um elemento implica a presença desse elemento na tabela de dimensão no *data warehouse* (ou um registo de inserção do mesmo atualmente em auditoria). A título de exemplo, a modelação para a validade de **AuditCustomers** é apresentada na Figura 35.

```
1.  /*
2.   * Predicate: AuditCustomers.valid
3.   * Validity properties of AuditCustomers that must hold in the received state so  that its
4.   * data is considered valid and ready to be loaded into the dimension.
5.   * AuditCustomers table is valid when all its rows in the given state are valid.
6.   * Uses AuditCustomer.valid predicate.
7.   */
8.  pred AuditCustomers.valid[s: State]
9.  {
10.     all p: (this.rows).s | p.valid[s]
11. }
12.
13. /*
14.  * Predicate: AuditCustomer.valid
15.  * Validates the (implicit) AuditCustomer argument in the received state. Predicate
16.  * is satisfied when the received audit customer is valid.
17.  * Uses AuditCustomer.validAttributeValues and AuditCustomer.validRowOperation
18.  * predicates.
19.  */
20. pred AuditCustomer.valid[s: State]
21. {
22.     this.validAttributeValues and this.validRowOperation[s]
```

```
23.  }
24.
25.  /*
26.   * Predicate: AuditCustomer.validAttributeValues
27.   * Predicate with an implicit first AuditBook argument which validates its fullname,
28.   * gender and tin values existence.
29.   */
30.  pred AuditCustomer.validAttributeValues
31.  {
32.      ( one this.fullname and one this.gender and one this.tin )
33.  }
34.
35.  /*
36.   * Predicate: AuditCustomer.validRowOperation
37.   * Validates the operation of the first implicit AuditCustomer argument in the received
38.   * state. The operation is valid when it is an insert or is a delete or update operation
39.   * and there is one customer in dimension with the same tin value, or there is an insert
40.   * operation in the set of audit customers with the same tin value.
41.   */
42.  pred AuditCustomer.validRowOperation[s: State]
43.  {
44.      let
45.          entities = (DW/DimCustomer.rows).s,
46.          audits = (AuditCustomers.rows).s
47.      {
48.          ( this.op = U or this.op = D) implies ( ( one e: entities | e.tin = this.tin ) or
49.              ( some pb: audits | pb.tin = this.tin and pb.op = I) )
50.      }
51.  }
```

Figura 35 - Modelo em Alloy do predicado de validade da tabela de auditoria AuditCustomers de StagingArea

Por último, foram modelados alguns predicados funcionais relativos às assinaturas definidas, nomeadamente, predicados para copiar valores de atributos entre dois elementos do mesmo conjunto – **AuditBook.copy[a: AuditBook, deep: lone Bool]**, **AuditCustomer.copy[a: AuditCustomer, deep: lone Bool]** e **AuditOrder.copy[a: AuditOrder]**.

Modelação de Tabelas de Quarentena

As tabelas de quarentena foram modeladas de forma semelhante às tabelas de auditoria. Nelas foram definidas as assinaturas **QuaBooks**, **QuaCustomers** e **QuaOrders**, estendendo **Relation** (Figura 36).

```
1.  /** Quarentine **/
2.  /* QuaBooks table */
3.  one sig QuaBooks extends Relation { rows:  QuaBook -> State }
4.  /* QuaCustomers table */
5.  one sig QuaCustomers extends Relation { rows: QuaCustomer -> State }
6.  /* QuaOrders table */
7.  one sig QuaOrders extends Relation { rows: QuaOrder -> State }
```

Figura 36 - Modelo em Alloy das tabelas de quarentena de StagingArea

As assinaturas das tabelas de quarentena utilizam assinaturas de linha específicas – **QuaBook**, **QuaCustomer** e **QuaOrder**. Estas entidades têm a mesma estrutura que as assinaturas de linhas de auditoria correspondentes, adicionando apenas um atributo de controlo com o motivo de quarentena. Para tal, as assinaturas definem uma nova relação (atributo **reason**), com um elemento do conjunto **QuarentineReason**. Esta assinatura é uma enumeração, tendo um elemento para simbolizar cada motivo de quarentena possível (Figura 37).

```
1.  enum QuarentineReason {
2.      BookMissingTitle, BookMissingISBN, BookMissingDescription,
3.      CustomerMissingFullname, CustomerMissingTin, CustomerMissingGender,
4.      OrderMissingBook, OrderMissingCustomer, OrderMissingDate,
5.      OrderMissingOrderID, OrderMissingMeasureValue,
6.      InvalidOperation, SKPError, ConciliationError
7.  }
```

Figura 37 - Modelo em Alloy de motivos de quarentena de StagingArea

A modelação das tabelas de quarentena envolveu uma declaração de facto para cada tabela, que define como chave primária o conjunto de todos os atributos das entidades. Os predicados de consistência destas tabelas são vazios. O módulo incluí como predicados funcionais os predicados **copy** e **copyAudit,** para todas as assinaturas de linha das tabelas de quarentena. O predicado **copy** copia os valores de atributos entre dois elementos da mesma assinatura, enquanto que o predicado **copyAudit** copia os valores de atributos de um elemento de auditoria (recebido como argumento) para um elemento de quarentena. Na Figura 38 estão apresentados os predicados descritos para a assinatura de **QuaCustomer**.

```
1.  /*
2.   * Predicate: QuaCustomer.copy
3.   * Predicate holds when the received QuaCustomer atom and the implicit this QuaCustomer
```

```
4.    * atom have the same data attribute values (fullname, gender and tin) and the same
5.    * same control attribute values (fullname_changed, gender_changed, tin_changed,
6.    * timestamp, op and source).
7.    * Unless deep argument is False, the QuaCustomer atoms will also share the id value.
8.    */
9.   pred QuaCustomer.copy[a: QuaCustomer, deep: lone Bool]
10.  {
11.      -- Copy Id value (deep by default)
12.      ( not (deep = False) implies this.id = a.id )
13.      -- Data attributes
14.      this.fullname = a.fullname and this.gender = a.gender and this.tin= a.tin and
15.      this.reason = a.reason and
16.      -- Control attribites
17.      this.fullname_changed = a.fullname_changed and this.tin_changed = a.tin_changed
18.      and this.gender_changed = a.gender_changed and
19.      this.timestamp = a.timestamp and this.op = a.op and this.source = a.source
20.  }
21.
22.  /*
23.   * Predicate: QuaCustomer.copyAudit
24.   * Predicate holds when the received AuditCustomer atom and the implicit this QuaCustomer
25.   * atom have the same data attribute values (fullname, gender and tin) and the
26.   * same control attribute values (fullname_changed, gender_changed, tin_changed,
27.   * timestamp, op and source).
28.   * Unless deep argument is False, the AuditCustomer and QuaCustomer atoms will
29.   * also share the same id value.
30.   */
31.  pred QuaCustomer.copyAudit[a: AuditCustomer, deep: lone Bool]
32.  {
33.      -- Copy Id value (deep by default)
34.      ( not (deep = False) implies this.id = a.id )
35.      -- Data attributes
36.      this.fullname = a.fullname and this.gender = a.gender and this.tin= a.tin and
37.      -- Control attribites
38.      this.fullname_changed = a.fullname_changed and this.tin_changed = a.tin_changed
39.      and this.gender_changed = a.gender_changed and
40.      this.timestamp = a.timestamp and this.op = a.op and this.source = a.source
41.  }
```

Figura 38 - Modelo em Alloy dos predicados de cópia de elementos QuaCustomer de StagingArea

Modelação de tabelas de equivalência

A modelação das tabelas de equivalência da área de retenção inclui as assinaturas **EquiBooks**, **EquiCustomer** e **EquiDates** (Figura 39).

```
/** Equi **/
/* EquiBooks table */
one sig EquiBooks extends Relation { rows:  EquiBook -> State }
/* EquiCustomers table */
one sig EquiCustomers extends Relation { rows: EquiCustomer -> State }
/* EquiDates table */
one sig EquiDates extends Relation { rows: EquiDate -> State }
```

Figura 39 - Modelo em Alloy das tabelas de equivalência de StagingArea

As assinaturas de tabelas de equivalência utilizam assinaturas de linha específicas (Figura 40). Estas assinaturas definem a estrutura do objeto de dados armazenado, incluindo os atributos necessários para o mapeamento de registos das dimensões numa chave de substituição.

```
/* EquiBook entity */
sig EquiBook extends Row {
    surrogate_key: one Int,
    ISBN : one Text,
    natural_key: lone Int,
    source: one SourceEnum
} { pos[surrogate_key] }

/* EquiCustomer entity */
sig EquiCustomer {
    surrogate_key: one Int,
    tin : one Text,
    natural_key: lone Int,
    source: one SourceEnum
} { pos[surrogate_key] }

/* EquiDate entity */
sig EquiDate {
    surrogate_key: one Int,
    timestamp : one Datetime
} { pos[surrogate_key] }
```

Figura 40 - Modelo em Alloy das entidades de equivalência de StagingArea

A modelação das tabelas de equivalência é suportada pela declaração das chaves primárias de cada tabela enquanto factos. As entradas de **EquiBooks** e **EquiCustomers** têm como chave primária os seus atributos **surrogate_key** e **source**, e as entradas de **EquiDates** têm como chave primária **surrogate_key**. Porém os predicados de consistência são mais complexos para as tabelas de equivalência. Estas são relações manipuladas diretamente no processo de ETL a modelar –

principalmente **EquiBooks** e **EquiCustomers** –, sendo, por isso, importante a garantia de um conjunto de propriedades específicas.

Na Figura 41 está apresentado o predicado de consistência **EquiBooks.consistent**. O predicado de consistência para a tabela **EquiCustomers** é muito semelhante. Neste predicado é validada a unicidade dos valores de **source** e **natural_key** (não existem duas linhas com os mesmos valores de **natural_key** e **source**) e a unicidade dos valores de **source** e **ISBN**. Também se valida que as linhas na tabela de equivalência têm o mesmo valor de **ISBN** se e só se tiverem o mesmo valor de **surrogate_key**. Por fim, garante-se que as entradas da tabela de dimensão no *data warehouse* utilizam chaves de substituição válidas.

```
1.  /*
2.   * Predicate EquiBooks.consistent
3.   * Consistency properties of EquiBooks that must hold in the received state.
4.   * EquiBooks table is consistent when a set of properties, which are rooted in the process
5.   * of assigning the correct surrogate key, hold amongst its rows, and a set of
6.   * properties, which keep the sync between the equi table and the dimension,
7.   * hold amongst its rows and the associated Book dimension rows.
8.   *
9.   * Consistency properties include:
10.  * - no duplicate row with same natural key and source values (each (source,natural_key)
11.  * value pair only exists in one row);
12.  * - no duplicate  row with same equivalence field and source values (each (source, ISBN)
13.  * value pair only exists in one EquiBook row);
14.  * - two rows have the same equivalence field value exactly when have the same surrogate
15.  * key;
16.  * - all books in Book dimension have a corresponding row in EquiBooks with same id and
17.  * equivalence field values.
18.  */
19. pred EquiBooks.consistent[s: State]
20. {
21.     -- Each source - natural key value pair only exist in one EquiBooks row
22.     no disj e1, e2: (this.rows).s | some e1.natural_key & e2.natural_key and e1.source = e2.source
23.     -- Each source - ISBN value pair only exist in one EquiBooks row
24.     no disj e1, e2: (this.rows).s | e1.ISBN = e2.ISBN and e1.source = e2.source
25.     -- Rows with same ISBN have same surrogate key values
26.     all e1, e2: (this.rows).s | e1.ISBN = e2.ISBN iff e1.surrogate_key = e2.surrogate_key
27.     -- Consistency between book dimension and equi rows
28.     all b: (DimBook.rows).s | some e: (this.rows).s | e.surrogate_key = b.id and e.ISBN = b.ISBN
29. }
```

Figura 41 - Modelo em Alloy do predicado de consistência para EquiBooks de StagingArea

Adicionalmente, na modelação das tabelas de equivalência foi especificado uma função para consulta da chave de substituição, com base no valor de fonte e de chave natural. A especificação da função para a relação **EquiBooks** pode ser observada na Figura 42 — esta modelação introduz funcionalidade à estrutura de dados representada.

```
1.  /*
2.   * Function: EquiBooks::lookUp
3.   * Looks up the surrogate key of the EquiBook row in the received state, with the received
4.   * natural key and source values (or none if no row is found).
5.   */
6.  fun EquiBooks::lookUp[s: State, nk: one Int, sr: one SourceEnum] : lone Int
7.  {
8.      ((this.rows).s & (natural_key.nk) & (source.sr)).surrogate_key
9.  }
```

Figura 42 - Modelo em Alloy da função de lookup da relação EquiBooks de StagingArea

Modelação de tabelas de controlo

A modelação de tabelas de controlo da área de retenção envolve a definição de assinaturas para representar as tabelas utilizadas no processo de CDC – Figura 43.

```
1.  /** CDC auxiliary structures **/
2.  /* SakilaBookPreviousLoad and SakilaBookCurrentLoad tables */
3.  one sig SakilaBookPreviousLoad extends Relation { rows:  SakilaBooks/Book-> State }
4.  one sig SakilaBookCurrentLoad extends Relation { rows:  SakilaBooks/Book-> State }
5.  /* SakilaCustomerPreviousLoad and SakilaCustomerCurrentLoad tables */
6.  one sig SakilaCustomerPreviousLoad extends Relation { rows:  SakilaBooks/Customer -> State }
7.  one sig SakilaCustomerCurrentLoad extends Relation { rows:  SakilaBooks/Customer -> State }
8.   /* SakilaOrderPreviousLoad and SakilaOrderCurrentLoad table */
9.  one sig SakilaOrderPreviousLoad extends Relation { rows:  SakilaBooks/JoinedOrderData-> State }
10. one sig SakilaOrderCurrentLoad extends Relation { rows:  SakilaBooks/ JoinedOrderData -> State }
11.
12. /* EbookStoreBookPreviousLoad and EbookStoreBookCurrentLoad tables */
13. one sig EbookStoreBookPreviousLoad extends Relation { rows:  EbookStore/Book-> State }
14. one sig EbookStoreBookCurrentLoad extends Relation { rows:  EbookStore/Book-> State }
15. /* EbookStoreUderPreviousLoad and EbookStoreUserCurrentLoad tables */
16. one sig EbookStoreUserPreviousLoad extends Relation { rows:  EbookStore/User -> State }
17. one sig EbookStoreUserCurrentLoad extends Relation { rows:  EbookStore/User -> State }
18. /* EbookStoreOrderPreviousLoad and EbookStoreOrderCurrentLoad tables */
19. one sig EbookStoreOrderPreviousLoad extends Relation { rows:  EbookStore/JoinedOrderData-> State }
20. one sig EbookStoreOrderCurrentLoad extends Relation { rows:  EbookStore/ JoinedOrderData -> State }
```

Figura 43 - Modelo em Alloy das tabelas de controlo de StagingArea

As assinaturas das tabelas modeladas utilizam assinaturas de linha definidas nos módulos de cada fonte. Esta definição simula a cópia da estrutura de dados a partir das fontes. As assinaturas referentes às tabelas de controlo são utilizadas para o processo de extração do ETL, sendo o seu conteúdo copiado das fontes de dados.

Capítulo 6.

Especificação Formal do Processo ETL

6.1 Especificação de Tarefas ETL

As tarefas ETL foram especificadas no módulo **ETL**. Este importa o módulo **StagingArea**, tendo acesso, desta forma, aos subsistemas envolvidos no processo. O módulo ETL contém a modelação dos predicados para as tarefas ETL e de predicados específicos auxiliares a esses. A especificação de cada tarefa utiliza como base de modelação os diagramas BPMN apresentados na Figura 9.

6.1.1 As Tarefas CDC

As tarefas de captura de dados alterados foram modeladas por predicados específicos, nomeadamente, **CDCEbookStoreBooks**, **CDCEbookStoreOrders**, **CDCEbookStoreCustomers**, **CDCSakilaBooks**, **CDCSakilaOrders** e **CDCSakilaCustomers**. Estes predicados recebem um estado inicial (anterior) e um estado final (seguinte) como argumentos. O estado final é caracterizado pela atualização das tabelas de carregamento envolvidas

no processo e pela possível inserção de novas entradas na tabela de auditoria, com base nas diferenças detetadas entre as tabelas de carregamento.

Na Figura 44 está apresentada, de forma parcial, a modelação do predicado **CDCEbookStoreBooks**, predicado que modela a operação de captura de dados alterados relativos à dimensão **Book**, a partir da fonte **EbookStore**. Neste caso, as tabelas de carregamento atual e de carregamento anterior são modeladas pelas assinaturas **EbookStoreBookCurrentLoad** e **EbookStoreBookPreviousLoad**, correspondentemente. A tabela da fonte de dados a partir da qual é realizada a extração é representada pela assinatura **Books** do módulo **EbookStore** (acedido por **StagingArea/EbookStore/Books**), sendo utilizada a tabela de auditoria **AuditBooks**. Ao longo desta secção, este predicado será utilizado como exemplo base na descrição geral dos predicados de CDC. O predicado pode ser analisado, de forma completa, no Anexo 1.

```
1.  /*
2.   * Predicate: CDCEbookStoreBooks
3.   * Receives two states (previous and next). The next state has the AuditBooks table with the new audit
4.   * books from EbookStore and an updated EbookStoreBookPreviousLoad table.
5.  */
6.  pred CDCEbookStoreBooks[s, s': State]
7.  {
8.      // Pre-conditions
9.      (EbookStoreBookCurrentLoad.rows).s = (StagingArea/EbookStore/Books.rows).s
10.
11.     // Post-conditions
12.     // Calculate differences between current and previous load and feed AuditBooks accordingly.
13.     // ...
14.     // Copy entries of current load in previous state to previous load in the next state
15.     (EbookStoreBookPreviousLoad.rows).s' = (EbookStoreBookCurrentLoad.rows).s
16.     // Empty entries of current load in the next state
17.     (EbookStoreBookCurrentLoad.rows).s' = none
18.
19.     // Frame-conditions
20.     StagingArea/frameExcept[AuditBooks, s, s']
21.     StagingArea/DW/frame[s,s']
22. }
```

Figura 44 - Modelo em Alloy do predicado CDCEbookStoreBooks

Especificação

Os predicados podem ser analisados quanto às suas pré-condições, pós-condições e condições de quadro. A pré-condição envolvida nos predicados CDC refere-se à cópia dos dados na tabela da fonte para a tabela local da área de retenção. Para tal, na pré-condição define-se que o conjunto de linhas da tabela de carregamento atual no estado inicial é exatamente igual ao conjunto de linhas da tabela da fonte no mesmo estado. No predicado apresentado na Figura 44, tal é definido na linha 9, através da restrição **(EbookStoreBookCurrentLoad.rows).s = (StagingArea/EbookStore/Books.rows).s**.

As pós-condições dos predicados em estudo envolvem a definição do conteúdo da tabela de auditoria (que recebe novas entradas com as alterações de dados registadas entre os carregamentos), da tabela de carregamento anterior e da tabela de carregamento atual no estado final do predicado.

Assim, a tabela de auditoria, no estado final, contém as entradas do estado anterior e novas entradas baseadas nas diferenças entre as tabelas de carregamento. Se **newEntries** simbolizar o conjunto de novas linhas de auditoria, calculadas a partir das tabelas de carregamento, **prevStateEntries** simbolizar as entradas na tabela de auditoria do estado anterior e **nStateEntries** representar o conjunto de todas as linhas na tabela de auditoria no estado final, queremos especificar que $nStateEntries = prevStateEntries \cup newEntries$. Em Alloy, a igualdade é reduzida [12] e é modelada por três inclusões:

- $newEntries \subseteq nStateEntries$
  As novas entradas de auditoria estão no conjunto de linhas da tabela de auditoria no estado seguinte (linhas 15-53 do Anexo 1);

- $prevStateEntries \subseteq nStateEntries$
  As entradas da tabela de auditoria do estado anterior estão presentes no conjunto de linhas da tabela de auditoria no estado seguinte (linha 56 do Anexo 1);

[12] $A = B \cup C \equiv (A \subseteq B \cup C) \wedge (B \cup C \subseteq A) \equiv (A \subseteq B \cup C) \wedge (B \subseteq A \wedge C \subseteq A) \equiv (A \subseteq B \cup C) \wedge (B \subseteq A) \wedge (C \subseteq A)$

- $nStateEntries \subseteq prevStateEntries \cup newEntries$

  As entradas da tabela de auditoria do estado seguinte estão presentes no estado anterior ou foram geradas a partir da diferença entres as tabelas de carregamento (linhas 59-84 do Anexo 1).

Cálculo das novas entradas de auditoria

As novas entradas de auditoria são calculadas pela diferença entre as tabelas de carregamento, utilizando a chave primária das tabelas de carregamento. As entradas de auditoria podem ser o resultado de uma operação de inserção, atualização ou remoção.

A inclusão na tabela de auditoria de entradas relativas à remoção de registos baseia-se na seguinte implicação: para todos os registos da tabela de carregamento anterior, se não existir um registo na tabela de carregamento atual com o mesmo valor de chave primária, então a tabela de auditoria tem, no estado seguinte, uma entrada com os valores de atributos do registo anterior. Esta entrada é assinalada com operação de remoção (sem nenhum atributo alterado). O comportamento descrito, aplicado ao predicado **CDCEbookStoreBooks**, está expresso na Figura 45.

```
1.  // D - All books from PreviousLoad which are not present in CurrentLoad in the previous state s
2.  // generate an AuditBook entry with Delete operation and EbookStore source in the next state s'
3.  all pl : (EbookStoreBookPreviousLoad.rows).s |
4.     ( no cl : (EbookStoreBookCurrentLoad.rows).s | cl.id = pl.id )
5.     implies
6.     ( one a' : (AuditBooks.rows).s' | a'.id = pl.id and a'.title = pl.title and a'.ISBN = pl.ISBN and
7.        a'.description = pl.summary and a'.timestamp = pl.last_update and a'.source = EbookStore and
8.        a'.op = D and a'.title_changed = False and a'.ISBN_changed = False and a'.description_changed = False
9.     )
```

Figura 45 - Modelo em Alloy do carregamento de entradas de auditoria do predicado CDCEbookStoreBooks (1)

Quanto às inserções, estas são adicionadas à tabela de auditoria com uma implicação semelhante: para todos os registos da tabela de carregamento atual, se não existir um registo na tabela de carregamento anterior com o mesmo valor de chave primária, então a tabela de auditoria tem, no estado seguinte, uma entrada com os valores de atributos do registo, assinalada com operação de inserção e com todos os atributos alterados. A especificação apresentada na Figura 46, expressa a implicação descrita, aplicada ao predicado **CDCEbookStoreBooks**.

```
1.    // I - All books from CurrentLoad which are not present in PreviousLoad in the previous state s
2.    // generate an AuditBook entry with Insert operation and EbookStore source in the next state s'
3.    all cl : (EbookStoreBookCurrentLoad.rows).s |
4.       ( no pl : (EbookStoreBookPreviousLoad.rows).s | cl.id = pl.id )
5.       implies
6.       ( one a' : (AuditBooks.rows).s' | a'.id = cl.id and a'.title = cl.title and (...) and
7.          a'.source = EbookStore and a'.op = I and
8.          a'.title_changed = True and a'.ISBN_changed = True and a'.description_changed = True
9.       )
```

Figura 46 - Modelo em Alloy do carregamento de entradas de auditoria do predicado CDCEbookStoreBooks (2)

As entradas que traduzem uma atualização de dados de registos já existentes são calculadas com base na presença de registos na tabela de carregamento atual e na tabela de carregamento anterior com o mesmo valor de chave primária, mas com algum valor de atributo alterado. Para cada registo detetado, a tabela de auditoria tem, no estado seguinte, uma entrada com operação de atualização, os seus valores atuais e informação sobre quais os atributos alterados. Uma exceção a este comportamento ocorre se o valor do atributo usado no processo de conciliação como campo de equivalência é alterado - neste caso, todos os atributos são assinalados como atualizados. Este comportamento é especificado na Figura 47.

```
1.    // U - All books from CurrentLoad which are present in PreviousLoad in the previous state s, but with
2.    // different values, generate an AuditBook with Update operation in the next state s'
3.    all cl : (EbookStoreBookCurrentLoad.rows).s |
4.       ( some pl : (EbookStoreBookPreviousLoad.rows).s | cl.id = pl.id and
5.          (cl.title != pl.title or cl.summary != pl.summary or cl.ISBN != pl.ISBN)
6.       )
7.       implies
8.       ( one a' : (AuditBooks.rows).s', pl : (EbookStoreBookPreviousLoad.rows).s |
9.          cl.id = pl.id and (cl.title != pl.title or cl.summary != pl.summary or cl.ISBN != pl.ISBN) and
10.         a'.id = cl.id and a'.title = cl.title and (...) and a'.source = EbookStore and a'.op = U and
11.         (
12.            ( cl.ISBN != pl.ISBN )
13.            implies
14.            ( a'.ISBN_changed = True and a'.title_changed = True and a'.description_changed = True)
15.            else
16.            (  a'.ISBN_changed = False and
17.               (cl.title != pl.title implies a'.title_changed = True else a'.title_changed = False) and
18.               (cl.summary != pl.summary implies a'.description_changed = True else a'.description_changed = False)
19.            )
20.         )
21.      )
```

Figura 47 - Modelo em Alloy do carregamento de entradas de auditoria do predicado CDCEbookStoreBooks (3)

Nas pós-condições dos predicados de CDC são, ainda, incluídas restrições para simular o comportamento de substituir o conteúdo da tabela de carregamento anterior pelo conteúdo da tabela de carregamento atual, bem como a remoção dos dados da tabela de carregamento atual. Estas restrições são especificadas por: **(EbookStoreBookPreviousLoad.rows).s' = (EbookStoreBookCurrentLoad.rows).s** e **(EbookStoreBookCurrentLoad.rows).s' = none** (ver linhas 15-17 da Figura 44).

Por último, os predicados em estudo definem que, no estado final, todas as relações mutáveis da área de retenção e do *data warehouse* se mantêm iguais ao estado inicial, excetuando a tabela de auditoria manipulada. Para isso, são usados os predicados de **frame** definidos nos módulos de **StagingArea** e **DW** (linhas 20-21 da Figura 44).

Análise

Os predicados CDC são inicialmente analisados com a execução de um comando **run** que inclui o predicado. A este podem ser adicionadas restrições adicionais no sentido de obter soluções mais significativas. Na Figura 48 está apresentado um exemplo de um comando de execução simples para análise do predicado **CDCEbookStoreBooks**.

```
1.  run extractEbookStoreBookExample {
2.      some s: State, s': s.next | CDCEbookStoreBooks[s,s'] and ((EbookStoreBookPreviousLoad.rows).s) !=
    ((EbookStoreBookCurrentLoad.rows).s)
3.  } for 2 but exactly 2 State
```

Figura 48 - Comando de execução para CDCEbookStoreBooks

Uma das soluções que obtemos está apresentada na Figura 49. Uma análise mais atenta da solução permite concluir que, no estado inicial (**State0**), **EbookStoreBookCurrentLoad** contém o mesmo conjunto de linhas que **Books** de **EbookStore** (um único registo de livro, com id 6). Nesta solução, **EbookStoreBookPreviousLoad** não se relaciona com qualquer registo no estado inicial. No estado seguinte, contudo, **EbookStoreBookPreviousLoad** relaciona-se com o registo de livro referido (com id 6), e **EbookStoreBookCurrentLoad** tem um conjunto vazio de linhas. A relação

AuditBooks de **StagingArea**, que tem um conjunto de linhas vazio no estado inicial, encontra-se relacionada com um registo de auditoria no estado final (**AuditBook**). Este é um registo com operação de inserção, com valor de **source** (fonte) **EbookStore** e **ISBN_changed**, **description_changed** e **title_changed** com valor de **True**. Os seus atributos têm valores obtidos através de um mapeamento do novo registo de livro (**id** com valor de 6, **title** com valor **Text0**, etc.).

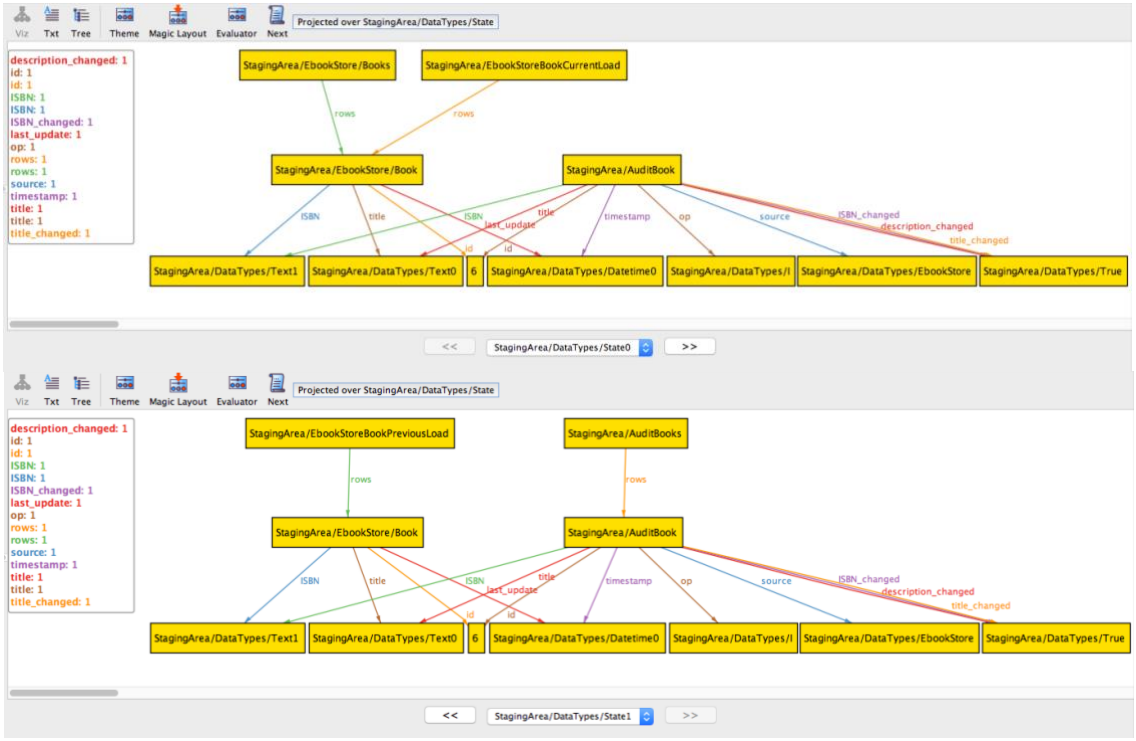


Figura 49 - Visualização de solução de execução de CDCEbookStoreBooks projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Validação

Os predicados desenvolvidos são sujeitos a asserções para validação da sua correção. Uma das asserções modeladas pretende verificar a manutenção da consistência do sistema, que neste caso é a consistência da área de retenção, uma vez que os predicados CDC apenas alteram tabelas que pertencem à área de retenção (tabelas de carregamento e de auditoria associadas). A verificação é feita pela implicação de que, para quaisquer estados consecutivos, a consistência da área de retenção no estado anterior (modelada pelo predicado **consistentStagingArea**) e o predicado CDC

com os estados definidos implicam a consistência da área de retenção no estado seguinte. Continuando o exemplo de captura de dados alterados relativos à dimensão **Book**, a partir da fonte **EbookStore** (predicado **CDCEbookStoreBooks**), a asserção relativa à sua capacidade de manutenção de consistência foi denominada **CDCEbookStoreBooksConsistent** (Figura 50). Adicionalmente, os predicados são também sujeitos a uma validação quanto à sua completude. Pretende-se, assim, verificar que nenhuma alteração de dados é descartada durante o processo: todas as alterações de dados na fonte causam um registo na tabela de auditoria. Para tal, é verificada a implicação de que, para quaisquer dois estados consecutivos, o predicado CDC que os tem como argumentos implica a existência de um registo na tabela de auditoria para todos os dados que foram inseridos, removidos ou atualizados na extração atual (em relação ao carregamento anterior). Na Figura 50 podemos ver que a asserção **CDCEbookStoreBooksComplete** utiliza as tabelas de carregamento **EbookStoreBookCurrentLoad** e **EbookStoreBookPreviousLoad** e a tabela de auditoria **AuditBooks**.

```
1.  /*
2.   * Assertion: CDCEbookStoreBooksConsistent
3.   * Consistency of staging area in a previous state and CDCEbookStoreBooks predicate
4.   * from previous to next state implies the consistency of staging area in the next state.
5.   */
6.  assert CDCEbookStoreBooksConsistent
7.  {
8.      all s: State, s': s.next |
9.          (consistentStagingArea[s] and CDCEbookStoreBooks[s,s']) implies consistentStagingArea[s']
10. }
11.
12. /*
13.  * Assertion: CDCEbookStoreBooksComplete
14.  * All necessary audit books are generated.
15.  */
16. assert CDCEbookStoreBooksComplete
17. {
18.     all s: State, s': s.next | CDCEbookStoreBooks[s,s'] implies
19.         (
20.             // Complete calculation (inserted book; deleted books; updated books are taken into account)
21.             (
22.                 all cl : (EbookStoreBookCurrentLoad.rows).s |
23.                     ( no pl : (EbookStoreBookPreviousLoad.rows).s | cl.id = pl.id ) implies
24.                     ( some a': (AuditBooks.rows).s' | a'.id = cl.id and a'.source = EbookStore and a'.op = I)
25.             )
26.             and
27.             (
28.                 all pl : (EbookStoreBookPreviousLoad.rows).s |
29.                     ( no cl : (EbookStoreBookCurrentLoad.rows).s | cl.id = pl.id ) implies
```

```
30.                 ( some a': (AuditBooks.rows).s' | a'.id = pl.id and a'.source = EbookStore  and a'.op = D)
31.             )
32.             and
33.             (
34.                 all cl : (EbookStoreBookCurrentLoad.rows).s |
35.                     ( one pl : (EbookStoreBookPreviousLoad.rows).s | ( cl.id = pl.id and (cl.title != pl.title or cl.summary !=
    pl.summary or cl.ISBN != pl.ISBN) ) ) implies
36.                     ( some a': (AuditBooks.rows).s' | a'.id = cl.id and a'.source = EbookStore and a'.op = U)
37.             )
38.         )
39.  }
```

Figura 50 - Validação de CDCEbookStoreBooks com asserções CDCEbookStoreConsistent e CDCEbookStoreComplete

As asserções de consistência e completude foram modeladas para todos os predicados CDC e, por último, verificadas com *scope* de 8 (mas apenas para dois estados). O facto da Alloy não ter encontrado nenhum contraexemplo para as asserções definidas indica que não foi encontrado nenhum erro na especificação e, como tal que esta pode estar consistente.

6.1.2 As Tarefas DQE

As tarefas de garantia de qualidade de dados foram modeladas com os predicados **DQEBooks**, **DQECustomers** e **DQEOrders**. Estes predicados recebem um estado inicial e um estado final como argumentos. O estado final é caracterizado pela atualização das tabelas de auditoria e, possivelmente, de quarentena, envolvidas no processo. As tarefas DQE do caso de estudo envolvem a aplicação de filtros, quanto à existência ou validação de valores de alguns atributos, e a aplicação de uma transformação de estandardização, com a capitalização do atributo de nome do cliente. No modelo em Alloy, o nível de abstração aplicado não permite a especificação da transformação definida. Neste sentido, os predicados especificados focam-se, essencialmente, na verificação da validade de cada registo da tabela de auditoria e na sua possível exclusão para a tabela de quarentena.

Para a verificação da validade num registo de auditoria foi utilizado um predicado definido na estrutura de dados – **AuditBook.valid**, **AuditCustomer.valid** e **AuditOrder.valid**. Adicionalmente, foi também utilizada uma função auxiliar para atribuição do motivo da quarentena

do registo no caso deste ser excluído - **AuditBook::getQuarentineReason**, **AuditCustomer::getQuarentineReason** e **AuditOrder::getQuarentineReason**. O elemento de **QuarentineReason** devolvido pela função auxiliar será utilizado como valor da relação de **reason** do novo registo de quarentena.

Na Figura 51 está apresentado o predicado de garantia de qualidade de dados para a dimensão **Book** – **DQEBooks**. Neste predicado, os registos de **AuditBooks** do estado inicial que não são válidos segundo o predicado de validade definido são excluídos da tabela de auditoria e adicionados à tabela de quarentena (modelada pela assinatura **QuaBooks**) no estado final. Este predicado será utilizado como exemplo da modelação desenvolvida.

```
1.  /*
2.   * Predicate: DQEBooks
3.   * Receives two states (previous and next). The next state has the audit books with quality ensured.
4.   * Applies filters to exclude invalid audit books and creates quarentine entries for each invalid one.
5.   * Uses AuditBook.valid predicate and AuditBook.getQuarentineReason function.
6.   */
7.  pred DQEBooks[s, s': State]
8.  {
9.     // Pre-conditions
10.    AuditBooks.consistent[s]
11.    QuaBooks.consistent[s]
12.
13.    // Post-conditions
14.    // AuditBooks: s -> s'
15.    // All audit books of previous state that are valid are kept as audit book in the next  state. The ones
16.    // that are not valid generate a new quarentine book in the next state with the same data and the correct quarentine
       reason.
17.    all ab : (AuditBooks.rows).s |
18.       ( ab.valid[s'] )
19.       implies
20.       ( ab in (AuditBooks.rows).s' )
21.       else
22.       (
23.          ab not in (AuditBooks.rows).s' and
24.          one qb' : (QuaBooks.rows).s' | qb' not in (QuaBooks.rows).s and qb'.copyAudit[ab, True]
25.             and qb'.reason = ab.getQuarentineReason[s]
26.       )
27.
28.    // AuditBooks: s' <- s
29.    // All audit books of the next state were present in the previous state and are valid
30.    all ab' : (AuditBooks.rows).s' | ab' in (AuditBooks.rows).s and ab'.valid[s]
31.
32.    // QuarentineBooks: s -> s'
33.    // All quarentine books of the previous state are kept in the next state
```

```
34.     all qb : (QuaBooks.rows).s | qb in (QuaBooks.rows).s'
35.
36.     // QuarentineBooks: s' <- s
37.     // All quarentine books of the next state either are present in the previous state or are created by an invalid audit
    book of the previous state
38.     all qb' : (QuaBooks.rows).s' |
39.       ( qb' in (QuaBooks.rows).s )
40.       iff not
41.       ( one ab : (AuditBooks.rows).s | ( not ab.valid[s] ) and qb'.copyAudit[ab, True] and
42.         qb'.reason = getQuarentineReason[ab, s]
43.       )
44.
45.     // Frame-conditions
46.     StagingArea/frameExcept[AuditBooks + QuaBooks, s, s']
47.     StagingArea/DW/frame[s, s']
48. }
```

Figura 51 - Modelo em Alloy do predicado DQEBooks

Especificação

Os predicados DQE modelados definem pré-condições, pós-condições e condições de quadro. As pré-condições envolvem a consistência da tabela de auditoria e da tabela de quarentena associadas à tarefa no estado inicial. No predicado apresentado na Figura 51, as pré-condições estão expressas nas linhas 9-10.

As pós-condições especificam a transição entre estados relativamente às entradas da relação de auditoria e da relação de quarentena. Para tal, todos os registos do conjunto de linhas da tabela de auditoria no estado inicial, se forem válidos, estão presentes no conjunto de linhas da mesma no estado final, caso contrário, não estarão presentes no estado final. Em vez deles estará presente uma entrada no conjunto de linhas da tabela de quarentena do estado seguinte – que é uma cópia dos dados do registo inválido e que tem correspondente razão de quarentena atribuída. A modelação deste comportamento para o predicado **DQEBooks** pode ser analisado na Figura 51, nas linhas 15-24. Adicionalmente, foi também imposta a restrição de que, no conjunto de linhas da tabela de auditoria no estado seguinte estão apenas presentes linhas que estavam no conjunto de linhas do estado anterior e que são válidas (linha 28 da Figura 51). O conteúdo da tabela de quarentena é controlado por duas restrições: 1) todas as entradas presentes no estado inicial estão presentes no estado final; e 2) as entradas do estado final ou estão presentes no estado anterior ou são uma cópia de um registo de auditoria inválido (Figura 51, linhas 32-41).

Por fim, as condições de quadro do predicado DQE implicam que, no estado final, todas as relações mutáveis do *data warehouse* e da área de retenção se mantenham iguais ao estado inicial – exceto pelas assinaturas de auditoria e quarentena associadas ao predicado. As restrições descritas, aplicada ao predicado **DQEBooks**, podem ser observadas nas linhas 43-44 da Figura 51.

Análise

Os predicados DQE são analisados com execuções do comando **run** que os incluem. Ao comando podem ser adicionadas restrições para obter soluções que representem casos mais significativos ou situações de extremos que possam causar erros. Na Figura 52 está apresentado como exemplo um comando de execução para análise do predicado **DQEBooks**, no qual se declara uma situação em que, no estado inicial, a tabela de auditoria (modelada pela assinatura **AuditBooks**) contém registos válidos e registos inválidos.

```
1.  run DQEBooksExample {
2.      some s: State, s': s.next | DQEBooks[s,s']
3.          and ( some a : (AuditBooks.rows).s | a.valid[s] ) and ( some a : (AuditBooks.rows).s | not a.valid[s] )
4.  } for 4 but exactly 2 State
```

Figura 52 - Comando de execução para DQEBooks

Uma das soluções que foram obtidas pode ser observada na Figura 53. A solução contém a relação de **AuditBooks**, no estado inicial, com dois registos associados (elementos **AuditBook0** e **AuditBook1**). No estado final, **AuditBooks** tem apenas um elemento na sua relação de linhas (**AuditBook1**), enquanto que **QuaBooks** tem um novo elemento no seu conjunto de linhas. Este registo de quarentena (**QuaBook**) tem os mesmos valores de **AuditBook0** (neste caso, **id**, **timestamp**, **op**, **source**, **title_changed**, **description_changed** e **ISBN_changed**), e relaciona-se com **BookMissingISBN** na relação de **reason**.

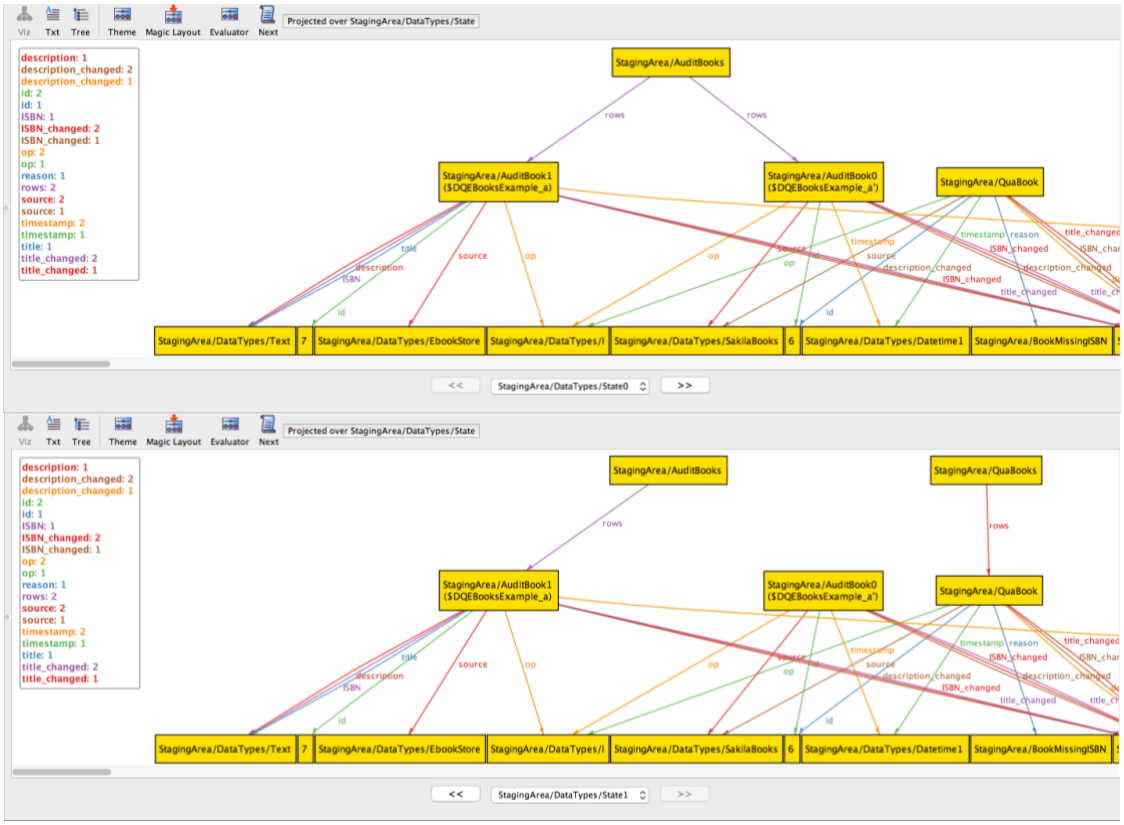


Figura 53 - Visualização de solução de execução de DQEBooks projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Validação

A validação dos predicados que foram desenvolvidos foi feita segundo uma asserção de manutenção de consistência no sistema e de garantia de validade. A manutenção de consistência do sistema é expressa pela implicação de que, para quaisquer dois estados consecutivos, a consistência do sistema no estado inicial e o predicado DQE com os dois estados como argumentos, implicam a consistência do sistema no estado final. Neste caso, uma vez que os predicados desenvolvidos apenas manipulam tabelas da área de retenção, a consistência do sistema é representada pelo predicado **consistentStagingArea**. Por sua vez, a garantia de validade verifica que, estando satisfeito o predicado DQE, com um qualquer estado inicial e estado final, todos os elementos no conjunto de linhas de auditoria no estado final são válidos. Um exemplo da asserção descrita, utilizando o predicado **DQEBooks**, pode ser observado na Figura 54.

```
1.  /*
2.   * Assertion: DQEBooksEnsured
3.   * With DQEBooks, all entries of AuditBooks in the next state are valid.
4.   */
5.  assert DQEBooksEnsured
6.  {
7.      all s: State, s': s.next | DQEBooks[s,s'] implies ( all a' : (AuditBooks.rows).s' | a'.valid[s'] )
8.  }
```

Figura 54 - Asserção DQEBooksEnsured para validação do predicado DQEBooks

As asserções de consistência e de garantia de qualidade foram modeladas para todos os predicados DQE e verificadas com *scopes* limitados. A validação em Alloy das asserções não encontrou nenhum contraexemplo, o que indica que não foi encontrado nenhum erro na especificação e que esta pode estar consistente.

6.1.3 As Tarefas DCI

As tarefas DCI para as dimensões **Book** e **Customer** foram modeladas por predicados específicos – **DCIBooks** e **DCICustomer**. Estes predicados recebem um estado anterior e um estado final, e são responsáveis por: a) atualizar os registos em auditoria, substituindo o valor da sua chave primária pela sua chave de substituição, ou excluir os registos para os quais não é encontrada uma chave de substituição; b) inserir os registos para os quais não é encontrada uma chave de substituição em quarentena; e c) manipular os registos na tabela de equivalência utilizada para o mapeamento. Para a modelação desta tarefa foram utilizados diversos predicados e funções auxiliares. Uma vez que os predicados definidos são declarativos e que se pretendeu modelar o comportamento em formato *batch*, o exercício de especificação desta tarefa foi complexo.

O predicado relativo à dimensão **Book** (**DCIBooks**) utiliza assinaturas que representam as tabelas de auditoria (**AuditBooks**), de quarentena (**QuaBooks**) e de equivalência (**EquiBooks**), e pode ser consultado no Anexo 2. Para este predicado, foram utilizadas, de forma auxiliar, as declarações:

- **EquiBooks::SK[s: State, i: one Text]: lone Int** – uma função que retorna o valor de chave de substituição presente no conjunto de linhas da tabela de equivalência da dimensão no estado recebido como argumento, para o valor de **ISBN**, também recebido como

argumento. Esta função é utilizada para descobrir o mapeamento atual de um valor de **ISBN** num valor de chave de substituição.

- **EquiBooks.newSK[s: State, s': State, new_sk: one Int, i: one Text]** – um predicado que recebe o argumento implícito de **EquiBooks**, o argumento de estado inicial e final, um valor de chave de substituição (**new_sk**) e um valor textual de **ISBN**. O predicado é satisfeito quando **new_sk** é uma chave de substituição nova válida (não é utilizado na tabela de equivalência no estado inicial e só é utilizado no estado final por registos com o mesmo valor de **ISBN**). Este predicado é utilizado para determinar que um dado valor é uma nova chave de substituição para registos da dimensão **Book**, associada a um valor textual de **ISBN** específico.

- **AuditBook.addSK[ab: AuditBook, e: EquiBook]** – um predicado que define o argumento implícito de **AuditBook** como partilhando todos os valores do elemento **AuditBook** recebido como argumento, exceto pelo valor de **id**, que é igual ao valor de **surrogate_key** do último argumento **EquiBook**. Este predicado é utilizado para determinar que o primeiro argumento (registo de auditoria da dimensão **Book**) é uma cópia do seguinte argumento e tem a chave de substituição do registo de equivalência recebido.

- **AuditBook.isLastRecord[books: set AuditBook]** – um predicado que valida que o argumento implícito de **AuditBook** é o registo mais recente, entre o conjunto de **AuditBook** recebido como segundo argumento, com os seus valores de **id** e **source**.

Especificação

Os predicados declarativos DCI definem pré-condições, pós-condições e condições de quadro. Estes predicados implicam, como pré-condição, a consistência das tabelas de auditoria e de equivalência associadas, e, ainda, a validade da primeira. Para isso, são utilizados os predicados de consistência e validade das assinaturas que representam as tabelas – apresentados no capítulo anterior. No caso do predicado **DCIBooks**, são utilizados os predicados **AuditBooks.consistent**, **AuditBooks.valid** e **EquiBooks.consistent**.

As condições de quadro impõem que, no estado final, todas as relações mutáveis do *data warehouse* e da área de retenção se mantêm iguais ao estado inicial, exceto pelas assinaturas relativas à tabela de auditoria, tabela de quarentena e de equivalência associadas ao predicado.

As pós-condições dos predicados englobam a manipulação dos registos na tabela de auditoria e na tabela de equivalência, e são divididas em diferentes restrições. De forma geral, pretendeu-se expressar:

- a atualização de todos os registos da tabela de auditoria no seu valor de chave primária ou a sua remoção caso não fosse encontrada nenhuma chave de substituição válida;

- a inserção de registos de quarentena consequentes da remoção de registos em auditoria para a qual não é encontrada nenhuma chave de substituição válida;

- a atualização de registos da tabela de equivalência (entradas da tabela de equivalência que mapeiam um objeto de dados de uma fonte numa chave de substituição são atualizadas com a remoção da chave natural guardada quando são descontinuadas – o objeto é removido ou muda de valor de campo de equivalência – e entradas que mapeiam um objeto com certo valor de campo de equivalência de uma fonte numa chave de substituição são atualizadas com a atribuição de uma chave natural quando são recuperadas – o objeto da fonte está em auditoria com o mesmo valor de campo de equivalência e operação diferente de remoção).

- a inserção de novos registos na tabela de equivalência (para objetos de dados de uma fonte que não são mapeados pela tabela, são criadas novas entradas com o seu valor de campo de equivalência, fonte, com a chave de substituição adequada (nova ou já existente para o valor de campo de equivalência) e com o valor adequado de chave natural (valor de chave primária do objeto de dados, ou sem valor – caso o objeto de dados seja posteriormente removido ou mude de valor de campo de equivalência).

Em primeiro lugar, a atualização ou remoção dos registos da tabela de auditoria é simulada. Para tal, é definido que, para cada registo na tabela de auditoria no estado inicial, caso a existência de um registo de equivalência para o mapeamento desse registo numa chave de substituição implique

que a tabela de equivalência continue válida e que não existam dois registos com o mesmo valor de campo de equivalência e de fonte, então o registo na tabela de auditoria será atualizado. Caso contrário, o registo será removido e existe um registo na tabela de quarentena, no estado final, com os mesmos dados que o registo de auditoria e com o motivo de quarenta de **ConciliationError**.

A atualização de um registo de auditoria implica que este não fará parte dos registos da tabela no estado final, e que é garantida a existência de um novo registo em auditoria, no estado final, resultante de adicionar o registo original e uma entrada da tabela de equivalência do estado final (especificado com o predicado **addSK**). A entrada de equivalência referida tem o mesmo valor no campo de equivalência (**ISBN** para dimensão **Book** e **tin** para dimensão **Customer**) e o mesmo valor de **source** do registo de auditoria anterior. Adicionalmente, está definido que, caso exista uma chave de substituição para o valor do campo de equivalência (se a função **SK** retornar um valor), então a entrada de equivalência terá esse valor como **surrogate_key**. Caso contrário, terá uma nova chave de substituição (utilizando o predicado **newSK**). Ainda, a entrada de equivalência terá como **natural_key** o valor de chave primária do registo de auditoria, caso o último registo em auditoria para essa mesma chave primária e fonte tenha o valor de campo de equivalência igual e não seja uma operação de remoção – se isso não se verificar, então o valor de **natural_key** da entrada de equivalência será removido. É, ainda, adicionada a restrição de que todas as entradas presentes nas linhas da tabela de auditoria do estado final são o resultado de adicionar um registo de auditoria do estado inicial e um registo de tabela de equivalência do estado final.

O conteúdo da tabela de quarentena é controlado por duas restrições: 1) todos os registos presentes no estado inicial, estão, também, presentes no estado final; e 2) todos os registos presentes no estado final estão presentes no estado anterior ou são uma cópia dos dados de um registo de auditoria do estado inicial para o qual não é possível obter uma entrada na tabela de equivalência para o seu mapeamento numa chave de substituição. A restrição seguinte simula a correção de chaves de equivalência descontinuadas – entradas da tabela de equivalência cujo elemento a ser mapeado mudou de mapeamento. Para tal, a restrição implica que, para todas as entradas na tabela de auditoria no estado inicial, todos os registos da tabela de equivalência do estado inicial que tenham a mesma fonte e a chave primária do registo de auditoria como valor de chave natural, e que exista uma outra entrada na tabela de auditoria que é o registo mais recente e que tem um valor de campo de equivalência diferente ou que é de operação de remoção, não estão presentes

na tabela de equivalência no estado seguinte, e existe uma entrada corrigida nesse estado (com os mesmos valores mas sem **natural_key**).

São também adicionadas restrições para manipular o conteúdo da tabela de equivalência de forma individual. A primeira restrição dita que as entradas do estado inicial que não são envolvidas no processo (nem atualizadas como consequência deste), mantêm-se nas linhas da tabela do estado final (caso contrário, serão removidas). A segunda restrição dita que as entradas da mesma tabela do estado final estão presentes no estado anterior ou foram utilizadas no processo.

Análise

Os predicados DCI são analisados com execuções do comando **run** que os incluem. Aos comandos definidos são adicionadas diferentes restrições para obter soluções que representem casos mais significativos ou situações que possam causar erros. São analisados cenários de coexistência de registos em auditoria referentes ao mesmo objeto, mas provenientes de diferentes fontes, cenários com múltiplas operações sobre o mesmo objeto de dados a partir da mesma fonte (registos de auditoria para inserção, atualização e remoção de um objeto), situações de atualização de registos de equivalência, ou de geração de novas chaves de substituição, entre outros. Como exemplo de um comando de execução para a análise do predicado **DCIBooks**, especifica-se a restrição de existência de dois registos na tabela de auditoria, com o mesmo valor de **ISBN** (campo de equivalência) mas com diferentes fontes (ver Figura 55).

```
1.  run DCIBooksExample {
2.      some s: State, s': s.next | DCIBooks[s,s'] and
3.          ( some a1, a2 : (AuditBooks.rows).s | a1.ISBN = a2.ISBN and a1.source != a2.source )
4.  } for 6 but exactly 2 State
```

Figura 55 - Comando de execução para DCIBooks

Na Figura 56 está apresentada uma parte da visualização de uma das soluções obtidas. Esta solução contém a relação de **AuditBooks**, no estado inicial, com dois registos associados com operação de inserção (elementos **AuditBook3** e **AuditBook2**), um elemento com **id** 7 da fonte **SakilaBooks** e outro com **id** 6 e fonte **EbookStore**. Os dois elementos partilham o mesmo valor de **ISBN**. Neste estado, **EquiBooks** relaciona-se com um conjunto vazio de registos. No estado final, **EquiBooks** relaciona-se com dois elementos, que têm o mesmo valor de **ISBN** e **surrogate_key** (valor 4). Um

dos elementos tem como fonte **SakilaBooks** e **natural_key** 7, e outro tem como fonte **EbookStore** e **natural_key** 6. Neste estado, **AuditBooks** relaciona-se, apenas, com dois novos elementos: **AuditBook0** e **AuditBook1**. **AuditBook0** tem os mesmos valores que **AuditBook3**, mas com valor de chave primária 4, enquanto que **AuditBook1** tem os mesmos valores que **AuditBook2** exceto pela chave primária (também com o valor 4).

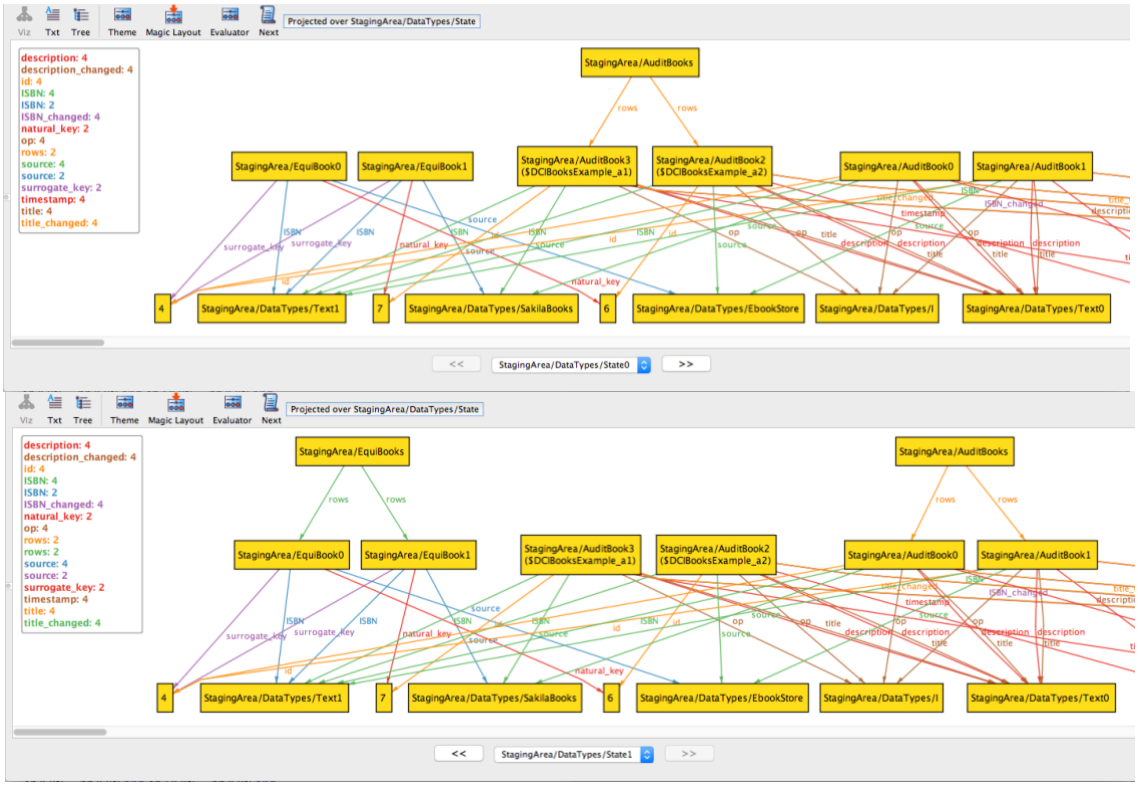


Figura 56 - Visualização de solução de execução de DCIBooks projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Validação

A validação dos predicados desenvolvidos foi feita segundo uma asserção de manutenção de consistência no sistema e uma asserção relativa à correção do mapeamento realizado. A asserção relativa ao mapeamento verifica que os predicados DCI utilizam de forma válida as chaves de substituição através da validação de duas propriedades. Uma vez satisfeito o predicado DCI com um qualquer estado inicial e estado final, a primeira propriedade garante que todos os registos na tabela de auditoria, no estado final, utilizam como chave primária o valor de uma chave de substituição de

um registo da tabela de equivalência, do mesmo estado, que é responsável pelo seu mapeamento (os registos partilham os mesmos valores de fonte e de campo de equivalência). Já a segunda propriedade garante que dois registos em auditoria, no estado final, têm o mesmo valor de chave primária se e só se partilharem o mesmo valor de campo de equivalência.

Um exemplo da asserção descrita pode ser observado na Figura 57, na qual se garante que a tarefa DCI para a dimensão **Books** concilia registos referentes à mesma entidade (com mesmo valor de **ISBN**) e provenientes de diferentes fontes no mesmo objeto de dados no contexto do sistema de *data warehousing*. O mapeamento num objeto de dados único é feito pela atribuição da mesma chave de substituição, utilizando, para isso, uma tabela de equivalência.

```
1.  /*
2.   * Assertion: DCIBooks
3.   * With DCIBooks, all entries of AuditBooks in the next state use as id a surrogate_key mapped by an
4.   * entry of EquiBooks, and all entries of AuditBooks in the next state have same ISBN value if and only
5.   * if they have the same id.
6.   */
7.  assert DCIBooks
8.  {
9.      all s: State, s': s.next | DCIBooks[s,s'] implies (
10.         (
11.             all a': (AuditBooks.rows).s' | one e': (EquiBooks.rows).s' |
12.                 a'.source = e'.source and a'.ISBN = e'.ISBN and a'.id = e'.surrogate_key
13.         ) and
14.         (
15.             all a1, a2: (AuditBooks.rows).s' | a1.ISBN = a2.ISBN iff a1.id = a2.id
16.         )
17.     )
18. }
```

Figura 57 - Asserção DCIBooks para validação do predicado DCIBooks

As asserções foram modeladas e verificadas com um scope limitado para todos os predicados de DCI. A validação em Alloy das asserções não encontrou nenhum contraexemplo, o que indica que não foi encontrado nenhum erro na especificação e que esta pode estar consistente.

6.1.4 As Tarefas SKP

Uma tarefa *Surrogate Key Pipelining* é modelada pelo predicado **SKP**, que recebe como argumentos um estado inicial e um estado final. O estado final é caracterizado pela atualização do conteúdo da tabela de auditoria de **TFOrders**, sendo os seus registos atualizados de forma a utilizarem chaves de substituição para as dimensões **Book**, **Customer** e **Date**, e, possivelmente, pela atualização de tabela de quarentena associada, à qual são adicionados os registos da tabela de auditoria para os quais alguma das chaves de substituição não foi encontrada. Para a atribuição das chaves de substituição foram utilizadas funções auxiliares de pesquisa nas diferentes estruturas de dados, nomeadamente: **EquiBooks::lookUp[s: State, nk: one Int, sr: one SourceEnum] : lone Int**, **EquiCustomers::lookUp[s: State, nk: one Int, sr: one SourceEnum] : lone Int** e **EquiDates::lookUp[s: State, d: one Datetime] : lone Int**. As funções de *lookup,* para as dimensões **Book** e **Customer,** recebem um estado, um valor de chave natural e um valor de fonte, enquanto que a função para a dimensão **Date** recebe apenas um estado e um valor temporal. A especificação das funções auxiliares pode ser observada na Figura 58.

```
1.  /*
2.   * Function: EquiBooks.lookUp
3.   * Looks up the surrogate key of the EquiBook row in the received state, with the received
4.   * natural key and source values (or none if no row is found).
5.   */
6.  fun EquiBooks::lookUp[s: State, nk: one Int, sr: one SourceEnum] : lone Int
7.  {
8.      ((this.rows).s & (natural_key.nk) & (source.sr)).surrogate_key
9.  }
10.
11. /*
12.  * Function: EquiCustomers.lookUp
13.  * Looks up the surrogate key of the EquiCustomer row in the received state, with the received
14.  * natural key and source values (or none if no row is found).
15.  */
16. fun EquiCustomers::lookUp[s: State, nk: one Int, sr: one SourceEnum] : lone Int
17. {
18.     ((this.rows).s & (natural_key.nk) & (source.sr)).surrogate_key
19. }
20.
21. /*
22.  * Function: EquiCalendar.lookUp
23.  * Looks up the surrogate key of the EquiCalendar row in the received state, with the received
24.  * datetime as timestamp (or none if no row is found).
25.  */
26. fun EquiDates::lookUp[s: State, d: one Datetime] : lone Int
```

```
27.  {
28.      ((this.rows).s & (timestamp.d)).surrogate_key
29.  }
```

Figura 58 - Modelo em Alloy de funções auxiliares de lookup

Especificação

A especificação do predicado SKP pode ser observado na Figura 59. Este predicado define pré-condições, pós-condições e condições de quadro. As pré-condições envolvem a consistência das tabelas de auditoria e de quarentena que são manipuladas no predicado – representadas pelas assinaturas **AuditOrders** e **QuaOrders** (linhas 10-11 da Figura 59).

As pós-condições do predicado são restrições quanto ao conteúdo das tabelas de auditoria e de quarentena. Em primeiro lugar, todas as entradas da tabela de auditoria para as quais são encontradas as chaves de substituição necessárias (isto é, a função de **lookUp** de cada tabela de mapeamento retorna um valor), são removidas do conjunto de linhas da tabela no estado seguinte, sendo garantida a existência de uma nova entrada no seu conjunto de linhas, com os mesmo valores exceto pelos valores de **book**, **customer** e **date** que são atualizados com as chaves de substituição encontradas. As entradas da tabela de auditoria do estado inicial para as quais uma ou mais chaves não são encontrada, são, também, removidas da tabela de auditoria no estado seguinte, sendo garantida a existência de uma entrada no conjunto de linhas da tabela de quarentena do estado seguinte, com todos os valores iguais aos do registo anterior e com **SKPError** como valor de razão de quarentena. Este comportamento está modelado nas linhas 15-29 da Figura 59. Além disso, é, ainda, imposta a restrição de que todas as entradas da tabela de auditoria do estado seguinte têm uma entrada correspondente no estado anterior (com os mesmos valores, exceto pela atualização das chaves de substituição). As pós-condições que restringem o conteúdo da tabela de quarentena estabelecem que todos os registos que lhes estão associados no estado inicial, são mantidos na relação no estado final. Adicionalmente, os registos do estado final ou pertencem ao conjunto de linhas da tabela no estado anterior ou têm **SKPError** como valor de razão de quarentena, partilhando todos os valores de uma entrada na tabela de auditoria do estado inicial para a qual uma das chaves de substituição não foi encontrada - estas restrições podem ser observadas nas linhas 37-46 da Figura 59.

As condições de quadro deste predicado implicam que, no estado final, todas as relações mutáveis do *data warehouse* e da área de retenção se mantenham iguais ao estado anterior – exceto pelas assinaturas de tabela de auditoria e de quarentena associadas. Estas condições podem ser vistas nas linhas 49-50 da Figura 59.

```
1.  /*
2.   * Predicate: SKP
3.   * Receives two states (previous and next). The next state has the entries of AuditOrders table
4.   * with the appropriate surrogate keys of dimensions. If a surrogate key can not be found for an entry,
5.   * the entry is removed from AuditOrders and added to QuaOrders table.
6.   */
7.  pred SKP[s,s': State]
8.  {
9.     // Pre-conditions
10.    AuditOrders.consistent[s]
11.    AuditOrders.consistent[s]
12.
13.    // Post-conditions
14.    // AuditOrders: s -> s' (entries which have all sk looked up will be updated, others are put in quar)
15.    all o: (AuditOrders.rows).s | o not in (AuditOrders.rows).s' and
16.       (
17.          ( one lookUp[EquiBooks,s,o.book,o.source] and one lookUp[EquiCustomers,s,o.customer, o.source] and one
   lookUp[EquiDates, s, o.date] ) implies
18.          (
19.             one o': (AuditOrders.rows).s' | o' not in (AuditOrders.rows).s and
20.                o'.copyMeasures[o] and o'.copyControlAttributes[o] and
21.                o'.orderID = o.orderID and o'.source = o.source and
22.                o'.book = lookUp[EquiBooks, s, o.book, o.source] and o'.customer = lookUp[EquiCustomers, s, o.customer,
   o.source] and o'.date = lookUp[EquiDates, s, o.date]
23.          ) else
24.          (
25.             one qo': (QuaOrders.rows).s' | qo' not in (QuaOrders.rows).s and qo'.copyAudit[o] and qo'.reason = SKPError
26.          )
27.       )
28.
29.    // AuditOrders: s' <- s (entries have a corresponding entry in previous state – before updating SK values)
30.    all o': (AuditOrders.rows).s' | one o: (AuditOrders.rows).s |  o'.copyMeasures[o] and
31.       o'.copyControlAttributes[o] and o'.orderID = o.orderID and o'.source = o.source and
32.       one o'.book and one o'.customer and one o'.date and
33.       o'.book = lookUp[EquiBooks, s, o.book, o.source] and o'.customer = lookUp[EquiCustomers, s, o.customer,
   o.source] and o'.date = lookUp[EquiDates, s, o.date]
34.
35.    // QuaOrders: s -> s'
36.    all qo: (QuaOrders.rows).s | qo in (QuaOrders.rows).s'
37.
38.    // QuaOrders: s' <- s
39.    all qo': (QuaOrders.rows).s' | ( qo' in (QuaOrders.rows).s ) iff not
40.       (
```

```
41.             one o: (AuditOrders.rows).s | qo'.copyAudit[o] and qo'.reason = SKPError and
42.                 not ( one lookUp[EquiBooks, s, o.book, o.source] and one lookUp[EquiCustomers, s, o.customer, o.source]
    and one lookUp[EquiDates, s, o.date] )
43.           )
44.
45.       // Frame-conditions
46.       StagingArea/frameExcept[AuditOrders + QuaOrders, s, s']
47.       StagingArea/DW/frame[s, s']
48.   }
```

Figura 59 - Modelo em Alloy do predicado SKP

Análise

O predicado **SKP** é analisado com a execução de um comando **run**. Este comando pode definir restrições adicionais para a obtenção de soluções mais significativas. Na Figura 60 pode-se ver um exemplo de um comando de execução simples, no qual se restringe a análise de casos em que existam registos na tabela de auditoria **AuditOrders,** tanto no estado inicial como no estado final do predicado.

```
1.    run SKPExample {
2.        some s: State, s': s.next | SKP[s,s'] and some (AuditOrders.rows).s and some (AuditOrders.rows).s'
3.    } for 3 but exactly 2 State
```

Figura 60 - Comando de execução para SKP

Uma das soluções obtidas está apresentada, em modo visual, na Figura 61, na qual um registo da tabela de auditoria é corretamente atualizado para utilizar chaves de substituição no estado final. A solução contém a relação **AuditOrders**, no estado inicial, com um registo associado (elemento **AuditOrder1**). O elemento em auditoria é proveniente da fonte **EbookStore** e tem o valor 7 como valor da maioria dos seus atributos – tanto para valores de medidas como para valores de dimensão. Neste estado, a relação **EquiBooks** associa-se a um elemento de equivalência pertencente à fonte **EbookStore**, com chave natural 7 e chave de substituição 5. Também a relação **EquiCustomers** se relaciona com um elemento pertencente à fonte **EbookStore**, com chave natural 7 e chave de substituição 1. A relação de equivalência da dimensão temporal (**EquiDates**) apresenta um elemento que mapeia o valor de **Datetime0** para a chave de substituição 5. No estado seguinte, **AuditOrders** relaciona-se, unicamente, com o elemento **AuditOrder0**. Este registo toma o valor 7 em grande parte dos seus atributos. Contudo, associa-se ao valor 5 pelos atributos **date** e **book** e

ao valor 1 por **customer**. As restantes relações mantêm-se inalteradas em relação ao estado anterior.

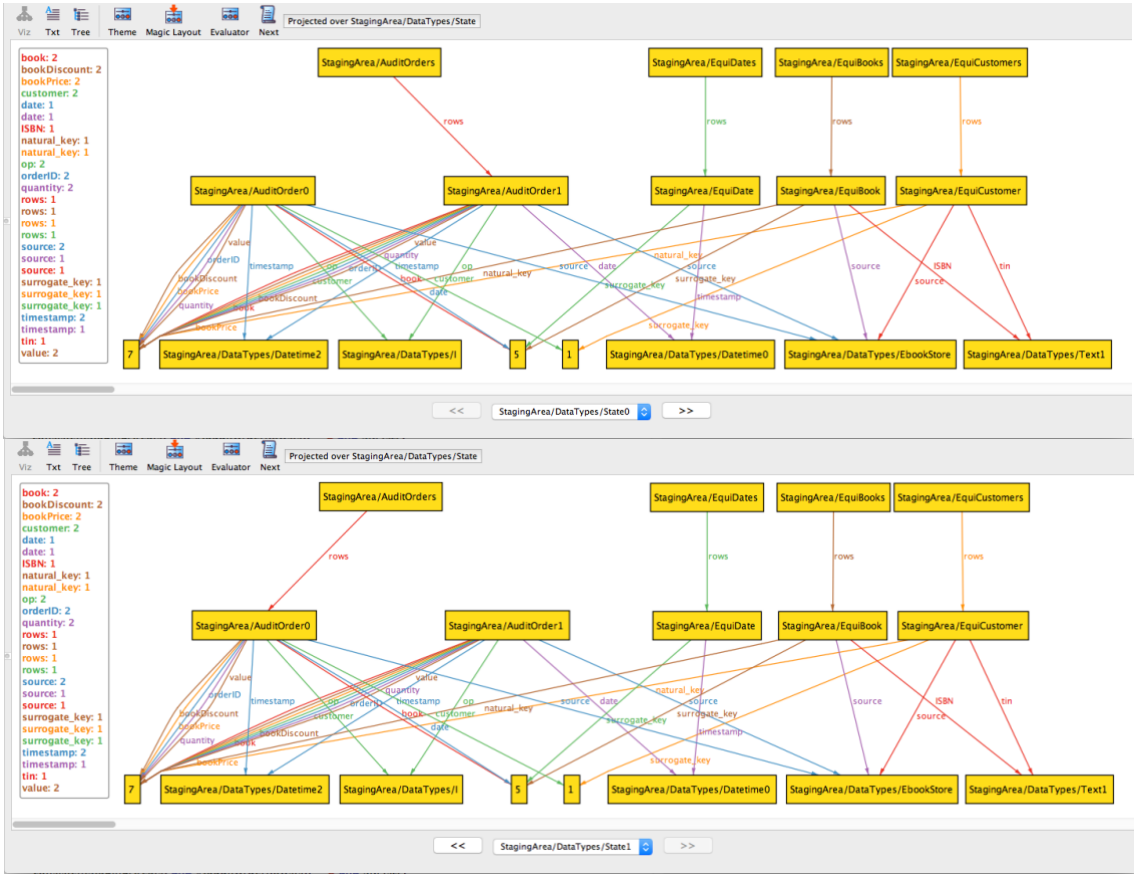


Figura 61 - Visualização de solução de execução de SKP projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Com a execução de um diferente comando (Figura 62) é possível analisar o comportamento do sistema quando uma das chaves de substituição das dimensões envolvidas não é encontrada, causando a exclusão de registos da tabela de auditoria.

```
1.  run SKPExampleQuarentine {
2.      some s: State, s': s.next | SKP[s,s'] and
3.          ( some o: (AuditOrders.rows).s | no lookUp[EquiBooks, s, o.book, o.source] or
4.              no lookUp[EquiCustomers, s, o.customer, o.source] or no lookUp[EquiDates, s, o.date] )
5.  } for 3 but exactly 2 State
```

Figura 62 - Execução em Alloy de SKP com registos inválidos

Uma das soluções obtidas está apresentada, parcialmente e em modo visual, na Figura 63. Nesta solução, **QuaOrders** encontra-se relacionado com um elemento (**QuaOrder1**) no estado inicial, assim como **AuditOrders** (relacionada com o elemento **AuditOrder**). No entanto, as relações de **EquiCustomers**, **EquiBooks** e **EquiDates** têm um conjunto vazio de linhas. No estado final do predicado, **AuditOrders** já não se relaciona com nenhum elemento, sendo um novo registo (**QuaOrder0**) adicionado a **QuaOrders**. Este novo elemento partilha os mesmos valores que **AuditOrder**, e relaciona-se com **SKPError** pela relação **reason**.

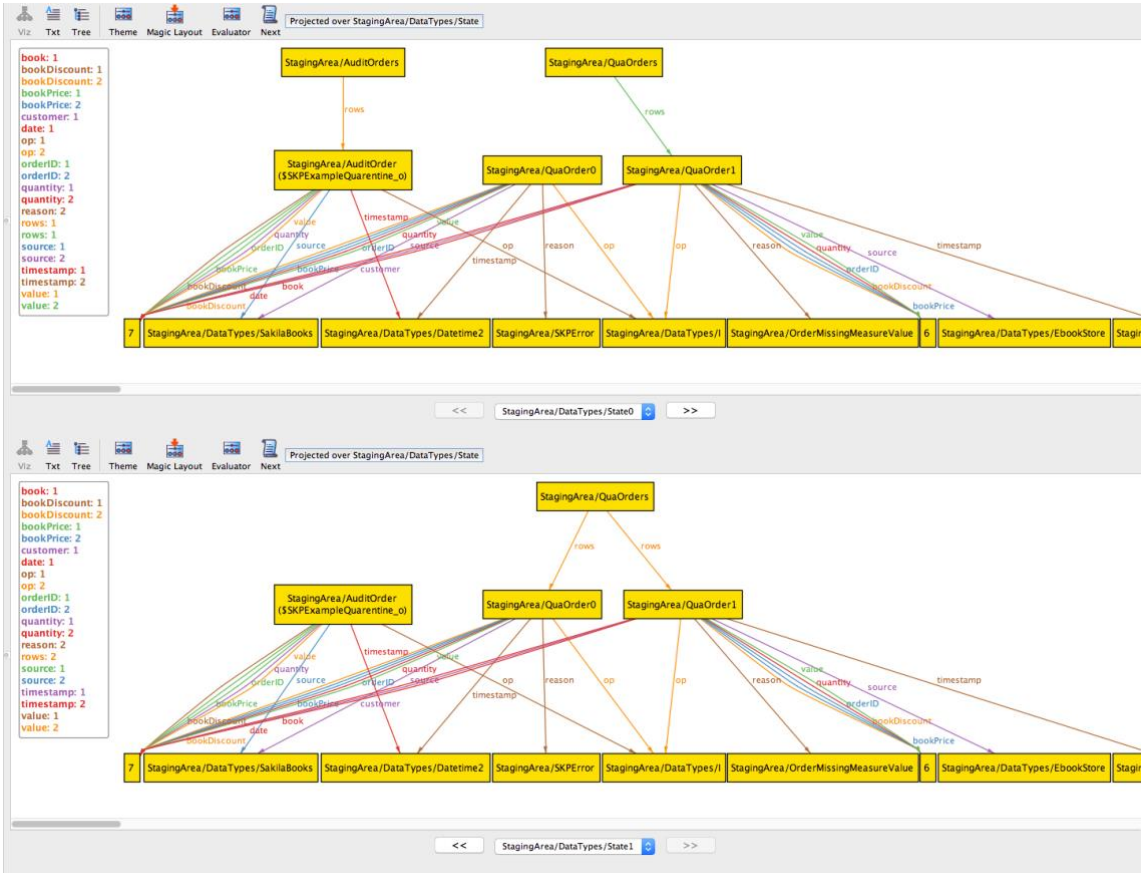


Figura 63 - Visualização de solução de execução de SKP com registos inválidos projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Validação

A validação do predicado **SKP** foi feita com base na análise realizada e na verificação de uma asserção de manutenção de consistência no sistema. Uma vez que o predicado apenas altera

assinaturas que representam tabelas pertencente à área de retenção, a consistência do sistema é aplicada ao seu nível, estando modelada pelo predicado **consistentStagingArea**. A verificação é feita pela implicação de que, para quaisquer dois estados consecutivos, a consistência da área de retenção, no estado anterior, e o predicado de SKP, com os estados definidos, implicam a consistência da área de retenção no estado seguinte – esta asserção está apresentada na Figura 64.

```
1.  /*
2.   * Assertion: SKPConsistent
3.   * Consistency of staging area in a previous state and SKP predicate with previous
4.   * and next state implies the consistency of staging area in the next state.
5.   */
6.  assert SKPConsistent
7.  {
8.      all s: State, s': s.next | ( consistentStagingArea[s] and SKP[s,s'] ) implies consistentStagingArea[s']
9.  }
```

Figura 64 - Asserção SKPConsistent para validação do predicado SKP

A verificação da asserção num *scope* limitado de 8 não encontra nenhum contraexemplo, o que significa que a especificação desenvolvida pode ser consistente.

6.1.5 As Tarefas SCD Loader

O carregamento de dados das duas dimensões das tabelas de auditoria para as tabelas de dimensão do *data warehouse* foram modeladas por dois predicados específicos: **SCDBooks** e **SCDCustomers**. Estes predicados recebem como argumentos dois estados (inicial e final) e utilizam as assinaturas relativas às tabelas de auditoria da área de retenção (de onde os dados são lidos e removidos) e às tabelas de dimensão do *data warehouse* (**DimBook** e **DimBook** do módulo **DW**). Estes predicados pretendem modelar o carregamento em *batch* dos dados, sendo que o resultado da operação simulada incluí nas tabelas de dimensão os dados mais recentemente alterados. Assim, **SCDBooks** e **SCDCustomers** utilizam predicados auxiliares para descobrir o registo mais recente na tabela de auditoria com a alteração de um dado valor, de forma a respeitar as marcas temporais das entradas e simulando operações de reescrita. Um exemplo dos predicados auxiliares desenvolvidos é **AuditBook.mostRecentChangeOfTitleAttribute[books: set AuditBook]** (Figura 65). Este predicado é satisfeito quando o argumento implícito **AuditBook** tem o atributo **title** assinalado como alterado e é o elemento com maior valor de **timestamp** dentro do conjunto

de elementos recebido como segundo argumento. A comparação de valores **Datetime** é conseguida através da modelação desta assinatura com a aplicação do módulo **ordering**, que estabelece uma ordem global entre os seus elementos. Este módulo define um conjunto de funções e predicados – como é o caso da função **nexts** que retorna o conjunto de elementos com ordenação superior ao elemento recebido como argumento. Esta função é utilizada na comparação realizada de valores do atributo temporal.

```
1.  /*
2.   * Predicate: AuditBook.mostRecentChangeOfTitleAttribute
3.   * Predicate with an implicit first AuditBook argument and a set of AuditBook as second argument.
4.   * The predicate is satified when this audit book has title changed and, amongst the received collection,
5.   * is the most recent audit book element for its book id which has title attribute changed (there is no
6.   * other audit book in the received set with the same id and changed title and a higher timestamp).
7.   */
8.  pred AuditBook.mostRecentChangeOfTitleAttribute[books: set AuditBook]
9.  {
10.    // Pre-condition: AuditBook is present in received set
11.    this in books
12.
13.    this.title_changed = True and
14.    ( no other_b : books |  (other_b != this) and (other_b.id = this.id) and (other_b.op != D ) and
15.      (other_b.title_changed = True) and other_b.timestamp in nexts[this.timestamp]
16.    )
17. }
```

Figura 65 - Modelo em Alloy do predicado auxiliar AuditBook.mostRecentChangeOfTitleAttribute

Na Figura 66 está apresentada a modelação do predicado **SCDBooks**, que representa a operação de carregamento dos dados relativos à dimensão **Book**. Neste caso, a tabela de auditoria é modelada pela assinatura **AuditBooks** e a tabela da dimensão pela assinatura **DimBook** do módulo **DW** (acedido por **StagingArea/DW/DimBook**). Este predicado será utilizado como exemplo na descrição da especificação dos predicados de carregamento de dados **SCD**.

```
1.  /*
2.   * Predicate: SCDBooks
3.   * Receives two states (previous and next). The next state has the audit book table emptied and
4.   * the dimension book table with the unchanged dimension books and new or updated entries.
5.   */
6.  pred SCDBooks[s, s': State]
7.  {
8.    // Pre-conditions
9.    AuditBooks.consistent[s] and AuditBooks.valid[s]
10.   EquiBooks.consistent[s]
```

```
11.     StagingArea/DW/DimBook.consistent[s]
12.
13.     // Audit books have the surrogate key values
14.     all ab: (AuditBooks.rows).s | some e: (EquiBooks.rows).s | e.surrogate_key = ab.id and e.ISBN = ab.ISBN
15.
16.     // Post-conditions
17.     // AuditBooks: s -> s' (all audit books with op=I or op=U are considered in the process. If the audit book entry is the
    most recent change for an attribute, the next state will have a dimension book with the same id and its attribute value)
18.     all ab: (AuditBooks.rows).s | (ab.op = I or ab.op = U) implies
19.         (   (
20.               ab.mostRecentChangeOfTitleAttribute[(AuditBooks.rows).s]  implies
21.               ( one b': (StagingArea/DW/DimBook.rows).s' | b'.id = ab.id and b'.title = ab.title )
22.             ) and (
23.               ab.mostRecentChangeOfDescriptionAttribute[(AuditBooks.rows).s] implies
24.               ( one b': (StagingArea/DW/DimBook.rows).s' | b'.id = ab.id and b'.description = ab.description )
25.             ) and ( one b': (StagingArea/DW/DimBook.rows).s' | b'.id = ab.id and b'.ISBN = ab.ISBN )
26.         )
27.
28.      // AuditBooks: s' <- s (empty audit books)
29.      (AuditBooks.rows).s' = none
30.
31.     // DimBooks: s -> s' (books in dimension in the previous state which are not changed by audit books
32.     // remain in dimension in the next state)
33.     all b: (StagingArea/DW/DimBook.rows).s | ( no ab: (AuditBooks.rows).s | ab.id = b.id and ab.op != D     )
34.       implies ( b in (StagingArea/DW/DimBook.rows).s' )
35.
36.     // DimBooks: s' <- s (book entries of the next state either are present in the previous state and were not changed by
    any audit books entry or were inserted or partially updated by audit books entries)
37.     all b': (StagingArea/DW/DimBook.rows).s' |
38.      (  // b' in previous state and no audit book entry (I/U) to change it
39.        b' in (StagingArea/DW/DimBook.rows).s and ( no ab: (AuditBooks.rows).s | ab.id  = b'.id and ab.op != D ) ) iff not (
40.        // some change of book id in audit books
41.        ( some ab: (AuditBooks.rows).s | ab.id = b'.id and ab.op != D ) and
42.        (
43.           // If there is(are) audit book(s) with title changed, then b has the most recent change. Otherwise, the previous
    value is kept.
44.           ( some ab: (AuditBooks.rows).s | ab.id = b'.id and ab.title_changed = True ) implies
45.           ( some ab: (AuditBooks.rows).s | ab.id = b'.id and
46.              ab.mostRecentChangeOfTitleAttribute[(AuditBooks.rows).s] and b'.title = ab.title
47.           ) else
48.           ( one b: (StagingArea/DW/DimBook.rows).s | b.id = b'.id and b.title = b'.title )
49.  ) and
50.        (    // If there is(are) audit book(s) with description changed, then b has the most recent change. Otherwise, the
    previous value is kept. (...)
51.         )
52.
53.     // Frame-conditions
54.     StagingArea/frameExcept[AuditBooks, s, s']
55.     StagingArea/DW/frameExcept[StagingArea/DW/DimBook, s, s']
56. }
```

Figura 66 - Modelo em Alloy do predicado SCDBooks

Especificação

As pré-condições dos predicados que modelam as tarefas de **SCD Loader** englobam a consistência das tabelas de auditoria, de equivalência e de dimensão, e a validade dos dados da primeira. Adicionalmente, foi definida uma restrição que implica que os dados em auditoria utilizem uma chave de substituição atual, a partir da tabela de equivalência associada.

Os predicados definem pós-condições referentes ao conteúdo das tabelas de auditoria e tabela de dimensão associadas. Para os registos na tabela de auditoria no estado inicial que tenham operações de inserção ou de atualização, se forem o registo com a alteração mais recente de um atributo, então existe um elemento pertencente ao conjunto de linhas da tabela de dimensão no estado final, com o mesmo valor de chave de substituição e com o valor do atributo alterado. A exceção ocorre apenas para o atributo de campo de equivalência, uma vez que o seu valor é único para a chave de substituição e a chave de substituição é única para o seu valor. Neste sentido, para cada registo da tabela de auditoria que tenha operação de inserção ou atualização, existe um elemento na tabela de dimensão do estado seguinte com o mesmo valor de chave de substituição e com o mesmo valor do campo de equivalência. A modelação desse comportamento pode ser observada na Figura 66, nas linhas 18-29, na qual são utilizados os predicados definidos com a assinatura de **AuditBook**, nomeadamente **mostRecentChangeOfTitleAttribute** e **mostRecentChangeOfDescriptionAttribute**. No estado seguinte, é definido que o conjunto de linhas da tabela de auditoria é vazio.

O conteúdo da tabela de dimensão no estado final é controlado no sentido de manter os registos inalterados do estado inicial e de inserir ou atualizar registos consoante os elementos em auditoria. Para a manutenção dos registos inalterados foi definido que os registos presentes no conjunto de linhas da tabela de dimensão no estado inicial, caso não exista nenhum elemento na tabela de auditoria com a mesma chave e com uma operação de inserção ou atualização, estão presentes nas linhas da tabela no estado final. Para refletir a inserção e a atualização de registos na tabela de dimensão foi definido que todos os registos no conjunto de linhas dessa tabela no estado final ou são mantidos desde o estado inicial (estão presentes no conjunto de linhas do estado e não existe nenhum registo na tabela de auditoria que os altera) ou são alterados por uma ou mais linhas da

tabela de auditoria. No último caso, é adicionada uma restrição para determinar o valor de cada atributo: no caso de existir algum registo que altere o mesmo, então o registo da tabela de dimensão tem como valor do atributo o valor do registo de auditoria mais recente, caso contrário, o valor de atributo é mantido inalterado a partir do estado anterior, ou seja, existe um registo na tabela de dimensão no estado anterior, que partilha a mesma chave de substituição e o mesmo valor do atributo com o registo da tabela de dimensão do estado final. O comportamento descrito está apresentado na Figura 66, nas linhas 33-52.

As condições de quadro dos predicados definem que todas as relações da área de retenção e do *data warehouse* se mantêm inalteradas, exceto pela tabela de auditoria e pela tabela de dimensão associadas aos predicados.

Análise

A análise dos predicados de carregamento de dados de dimensão foi realizada pela execução de diferentes comandos **run** que utilizam os predicados definidos, podendo adicionar restrições no sentido de observar o comportamento do sistema em diferentes condições. Um exemplo de execução do predicado **SCDBooks** está apresentado na Figura 67, no qual se analisa o carregamento de dados para a dimensão **Book**, existindo, no estado inicial, um ou mais registos na tabela de dimensão, um ou mais registos na tabela de auditoria com operação de remoção e um ou mais registos com operação de inserção ou de atualização.

```
1.  run SCDBooksExample {
2.      all s: State, s': s.next | SCDBooks[s,s'] and some (StagingArea/DW/DimBook.rows).s and
3.          ( some a: (AuditBooks.rows).s | a.op=D ) and ( some a: (AuditBooks.rows).s | a.op=I or a.op=U )
4.  } for 5 but exactly 2 State
```

Figura 67 - Comando de execução para SCDBooks

Uma das soluções obtida está parcialmente apresentada na Figura 68. A solução apresenta, no estado inicial, a assinatura referente à tabela de auditoria – **AuditBooks** – com dois elementos no seu conjunto de linhas relacionadas (elementos **AuditBook0** e **AuditBook1**). O elemento **AuditBook0** é uma operação de remoção da entidade com chave de substituição 7, enquanto que **AuditBook1** reflete uma operação de atualização da mesma entidade, tendo apenas a descrição alterada (**description_changed** toma o valor de **True**, e **description** toma o valor de **Text0**).

Neste estado, a assinatura que modela a tabela de dimensão – **DimBook** do módulo **DW** – tem um elemento relacionado (**Book0**), que tem como valor de chave primária a chave de substituição 7, e os atributos **title**, **description** e **ISBN** com valor de **Text1**. No estado seguinte, **AuditBooks** tem um conjunto vazio de linhas, e **DimBook** relaciona-se com um elemento **Book1**. Este elemento tem os valores de atributos iguais ao registo anterior **Book0**, exceto pelo valor de **title**, que passou a ser **Text0**.

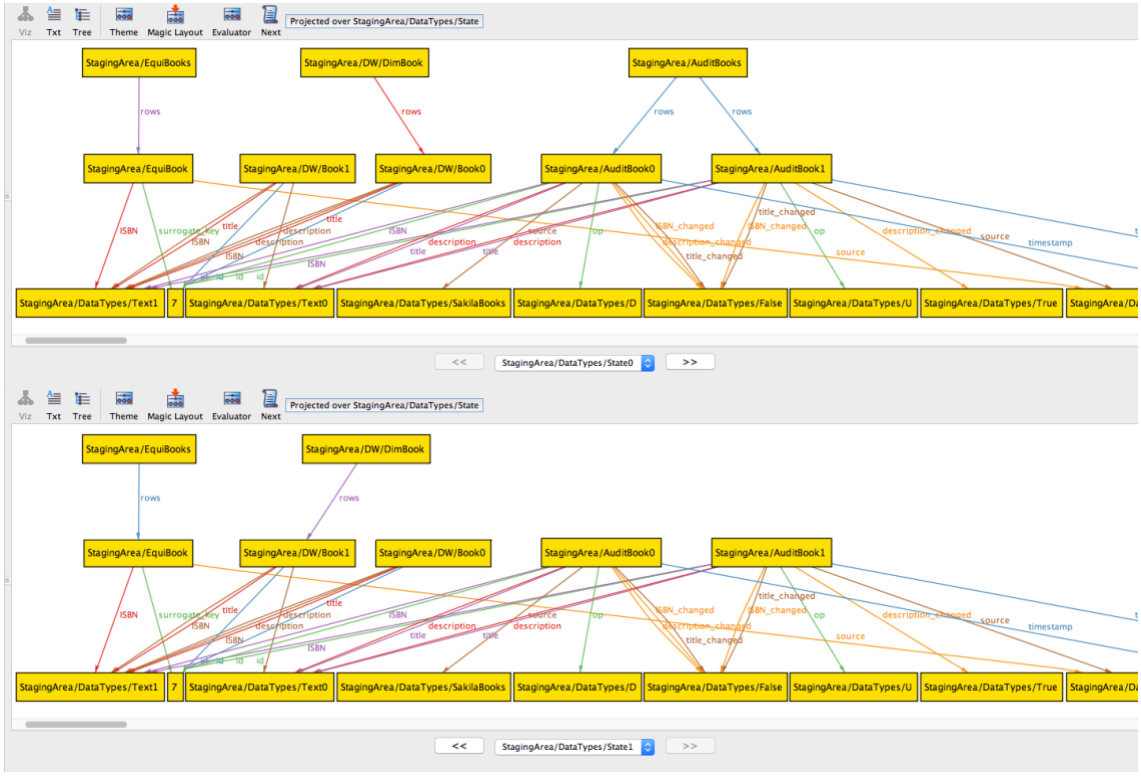


Figura 68 - Visualização parcial de solução de execução de SCDBooks projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Validação

Os predicados desenvolvidos foram sujeitos a asserções para validação da sua correção. Uma das asserções modeladas pretendeu verificar a manutenção da consistência do sistema – neste caso, da consistência da área de retenção e do *data warehouse*, uma vez que os predicados alteram tabelas de auditoria (pertencentes à área de retenção) e tabelas de dimensão do *data warehouse*. A verificação é feita pela implicação de que, para quaisquer estados consecutivos, a consistência da

área de retenção (modelada pelo predicado **consistentStagingArea**) e do *data warehouse* (modelada pelo predicado **consistentDW**) no estado anterior e o predicado de carregamento de SCD com os estados definidos implicam a consistência dos dois subsistemas no estado seguinte. Continuando o exemplo de carregamento de dados da dimensão **Book**, a asserção relativa à manutenção de consistência pelo predicado associado é denominada por **SCDBooksConsistent** e está apresentada na Figura 69. Adicionalmente, os predicados são sujeitos a uma validação quanto às alterações nos dados da tabela de dimensão. Pretende-se com isso verificar que todos os registos presentes na dimensão no estado inicial são mantidos no estado final do predicado – os seus valores de chave primária e campo de equivalência mantêm-se os mesmos, e caso não exista um registo em auditoria que cause uma alteração de valor de um atributo (registo com a mesma chave e o atributo assinalado como alterado), os seus valores de atributos também são mantidos. Na Figura 69 podemos ver a asserção **SCDBooksDataMaintenance** a validar a manutenção de dados descrita.

```
1.  /*
2.   * Assertion: SCDBooksConsistent
3.   * Consistency of staging area and consistency of DW in a previous state and SCDBooks predicate with
4.   * previous and next states implies the consistency of staging area and DW in the next state.
5.   */
6.  assert SCDBooksConsistent
7.  {
8.      all s: State, s': s.next | ( consistentStagingArea[s] and consistentDW[s] and SCDBooks[s,s'] ) implies
9.          (consistentStagingArea[s'] and consistentDW[s])
10. }
11.
12. /*
13.  * Assertion: SCDBooksDataMaintenance
14.  * SCDBooks with an initial and final state keeps dimension records from initial state ( maintains
15.  * every id and ISBN values) and keeps the existing attribute values when no audit record changes it.
16.  */
17. assert SCDBooksDataMaintenance
18. {
19.     all s: State, s': s.next | SCDBooks[s,s'] implies (
20.         all b: (StagingArea/DW/DimBook.rows).s |
21.             -- ISBN are mantained from state s to s'
22.             ( one b': (StagingArea/DW/DimBook.rows).s' | b'.id = b.id and b.ISBN = b'.ISBN)
23.             and
24.             -- Attributes (title or description), when not changed, keep previous values
25.             (
26.                 ( no ab: (AuditBooks.rows).s | ab.id = b.id and ab.title_changed = True )
27.                 implies
28.                 ( one b': (StagingArea/DW/DimBook.rows).s' | b'.id = b.id and b'.title = b.title )
29.             )
```

```
30.             and
31.             (
32.                 ( no ab: (AuditBooks.rows).s | ab.id = b.id and ab.description_changed = True )
33.                 implies
34.                 ( one b': (StagingArea/DW/DimBook.rows).s' | b'.id = b.id and b'.description = b.description )
35.             )
36.         )
37.     }
```

Figura 69 - Validação de SCDBooks com asserções SCDBooksConsistent e SCDBooksDataMaintenance

As asserções foram modeladas e verificadas, com um *scope* limitado, para todos os predicados de carregamento de dados de dimensões. A validação em Alloy das asserções não encontrou nenhum contraexemplo, o que indica que não foi encontrado nenhum erro na especificação e que esta pode ser consistente.

6.1.6 As Tarefas Facts Loader

O carregamento de dados da tabela de factos **TFOrders** está modelada através do predicado **loadTFOrders.** Este predicado recebe como argumentos um estado inicial e um estado final, e utiliza as assinaturas relativas à tabela de auditoria dos factos na área de retenção (**AuditOrders**) e à tabela de factos do *data warehouse* (**FTOrders** do módulo **DW**). O principal objetivo do predicado é simular o carregamento dos factos em auditoria no estado inicial para a tabela de factos do *data warehouse* no estado seguinte. O comportamento especificado é semelhante ao comportamento dos predicados de carregamento de dados das dimensões de variação lenta sem manutenção de histórico, descrito anteriormente. Porém, trata-se de uma versão mais simplificada, uma vez que apenas são consideradas inserções de novos factos.

```
1.  /*
2.   * Predicate: loadTFOrders
3.   * Receives two states (previous and next). The next state has the AuditOrders table emptied and
4.   * the FTOrders table loaded with the new inserted facts.
5.   */
6.  pred loadFTOrders[s, s': State]
7.  {
8.      // Pre-conditions
9.      AuditOrders.consistent[s]
```

```
10.     AuditOrders.valid[s]
11.     StagingArea/DW/FTOrders.consistent[s]
12.     // All foreign key data already loaded
13.     all ao: (AuditOrders.rows).s | ao.book in ((StagingArea/DW/DimBook.rows).State).id
14.     all ao: (AuditOrders.rows).s | ao.customer in ((StagingArea/DW/DimCustomer.rows).State).id
15.     all ao: (AuditOrders.rows).s | ao.date in ((StagingArea/DW/DimDate.rows).State).id
16.
17.     // Post-conditions
18.     // AuditOrders: s -> s' (all audit orders with op=I are considered in the process and inserted in fact table)
19.     all ao: (AuditOrders.rows).s | ao.op = I implies ( one o': (StagingArea/DW/FTOrders.rows).s' |
20.       -- new fact is not present in previous state
21.       o' not in (StagingArea/DW/FTOrders.rows).s and
22.       -- new fact has same measures
23.       o'.value = ao.value and o'.quantity = ao.quantity and o'.bookPrice = ao.bookPrice and o'.bookDiscount =
    ao.bookDiscount and
24.       -- new fact has same dimension values
25.       o'.source = ao.source and o'.orderID = ao.orderID and
26.       o'.date = ao.date and o'.book = ao.book and o'.customer = ao.customer
27.     )
28.
29.     // AuditOrders: s' <- s (empty audit orders)
30.      (AuditOrders.rows).s' = none
31.
32.     // FactOrders: s -> s' (order entries of the previous state remain in fact table in the next state)
33.     all o: (StagingArea/DW/FTOrders.rows).s | o in (StagingArea/DW/FTOrders.rows).s'
34.
35.     // FactOrders: s' <- s (order entries of the next state either are present in the previous state or were inserted by an
    audit orders entry)
36.     all o': (StagingArea/DW/FTOrders.rows).s' | ( o' in (StagingArea/DW/FTOrders.rows).s ) iff not (
37.         one ao: (AuditOrders.rows).s |
38.             o'.value = ao.value and o'.quantity = ao.quantity and o'.bookPrice = ao.bookPrice and o'.bookDiscount =
    ao.bookDiscount and
39.             o'.source = ao.source and o'.orderID = ao.orderID and
40.             o'.book = ao.book and o'.customer = ao.customer and o'.date = ao.date
41.       )
42.
43.     // Frame-conditions
44.     StagingArea/frameExcept[AuditOrders, s, s']
45.     StagingArea/DW/frameExcept[StagingArea/DW/FTOrders, s, s']
46. }
```

Figura 70 - Modelo em Alloy do predicado loadFTOrders

Especificação

O predicado **loadFTOrders** está apresentado na Figura 70. Este predicado define como pré-condição a consistência e validade da tabela de auditoria dos factos e a consistência da tabela de

factos do *data warehouse*. Para isso, são utilizados os predicados **AuditOrders.consistent[s: State]**, **AuditOrders.valid[s: State]** e **FTOrders.consistent[s: State]**. Adicionalmente, todos os dados relacionados com os factos em auditoria estão já carregados no *data warehouse* – nomeadamente as chaves estrangeiras para as tabelas de dimensão **Book**, **Date** e **Customer** (linhas 12-15 na Figura 70).

As pós-condições do predicado destinam-se a definir o conteúdo da tabela de auditoria e da tabela de factos do *data warehouse*. Aqui, em primeiro lugar, é estabelecido que, para todos os registos pertencentes ao conjunto de linhas da tabela de auditoria no estado inicial do predicado, caso tenham operação de inserção, é adicionado um novo registo à tabela de factos no estado final, com os mesmos valores de dimensão e de medidas do registo em auditoria. No estado final, a tabela de auditoria de factos tem um conjunto vazio de linhas. Estas restrições podem ser observadas na Figura 70, nas linhas 19-32. Adicionalmente, foi estabelecido que todos os registos da tabela de factos no estado inicial do predicado são mantidos no estado final. Os registos da tabela no estado final ou estão presentes no conjunto de linhas do estado inicial ou existe um registo na tabela de auditoria no estado inicial, que partilha com o registo da tabela de factos os valores de dimensão e de medidas. A modelação deste comportamento pode ser vista na Figura 70, nas linhas 35-43.

As condições de quadro do predicado modelado especificam que todas as relações mutáveis da área de retenção e do *data warehouse* são mantidas inalteradas no estado final do predicado – exceto pela tabela de auditoria dos factos e pela tabela de factos do *data warehouse*.

Análise

A análise do predicado de carregamento de dados da tabela de factos foi realizada através da execução de diferentes comandos **run** que utilizam o predicado modelado, adicionando restrições no sentido de observar o comportamento do sistema em diferentes condições. Um exemplo de comando de execução é apresentado na Figura 71, no qual se analisa o carregamento de dados da tabela de factos **TFOrders** com um estado inicial caracterizado pela presença de alguns registos nas linhas da tabela de factos e nas linhas da tabela de auditoria de factos.

```
1.  run loadFTOrdersExample {
2.      all s: State, s': s.next | loadFTOrders[s,s'] and some (StagingArea/DW/FTOrders.rows).s  and some
    (AuditOrders.rows).s
3.  } for 6 but exactly 2 State
```

Figura 71 - Comando de execução para loadFTOrders

Uma das soluções analisada está parcialmente apresentada na Figura 72. A solução, no estado inicial, a assinatura referente à tabela de auditoria – **AuditOrders** – com um elemento no seu conjunto de linhas relacionadas (elemento **AuditOrder**). O elemento **AuditOrder** é uma operação de inserção de um facto da fonte **EbookStore**. Neste estado, a tabela de factos do *data warehouse* – modelada pela assinatura **FTOrders** – relaciona-se com um elemento – **Order0**. No estado seguinte, **AuditOrders** não se relaciona com nenhum registo pela relação de **rows**. Porém, **FTOrders** relaciona-se com dois elementos – **Order0** e **Order1**. O elemento **Order1** é um novo facto e tem os mesmos valores de atributos que **AuditOrder**.

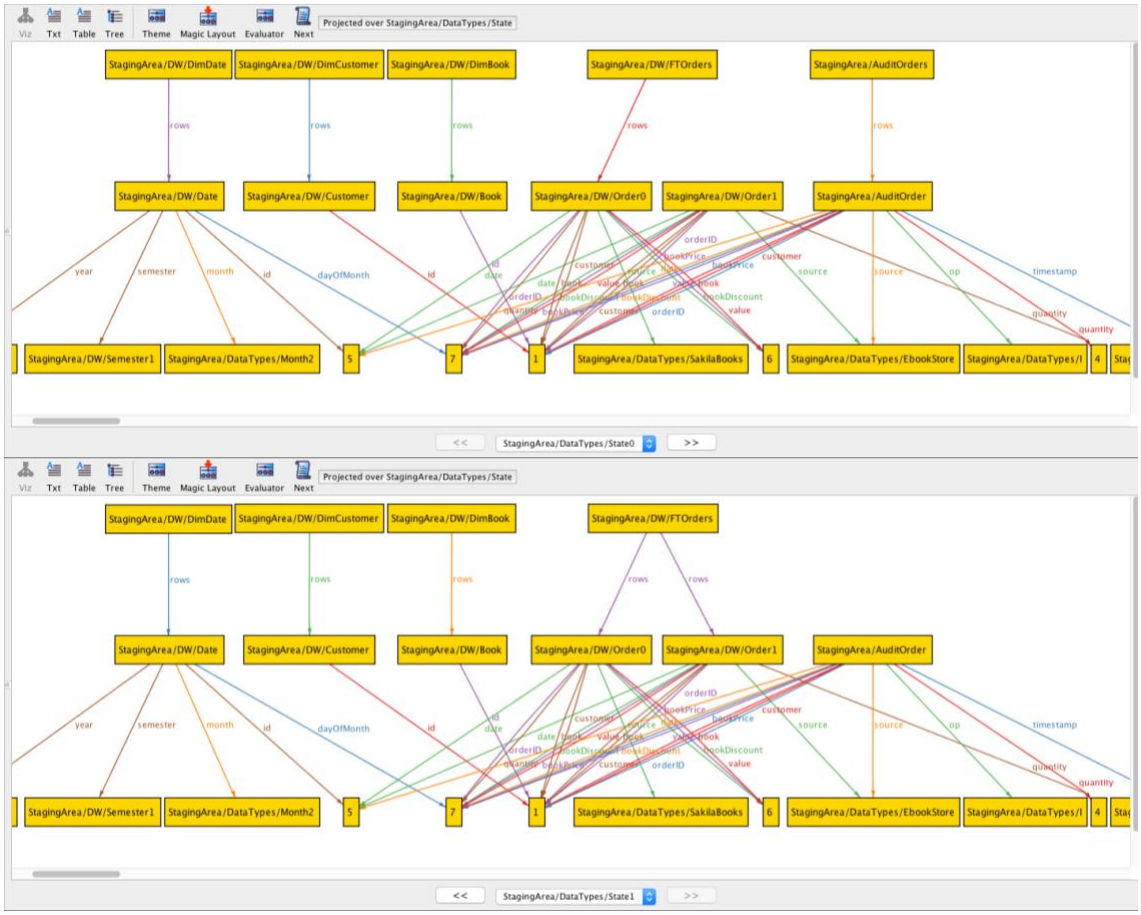


Figura 72 - Visualização parcial de solução de execução de loadFTOrders projetada no estado anterior (imagem acima) e projetada no estado seguinte (imagem abaixo)

Validação

A validação do predicado desenvolvido para carregamento de dados da tabela de factos foi feita com uma asserção de manutenção de consistência no sistema e uma asserção relativa à manutenção de factos. A consistência envolvida na asserção diz respeito à consistência da área de retenção (modelada pelo predicado **consistentStagingArea**) e à consistência do *data warehouse* (modelada pelo predicado **consistentDW**). A asserção estabelece que, para quaisquer dois estados consecutivos, a consistência dos dois subsistemas no estado inicial e o predicado **loadFTOrders** com os dois estados implica a consistência dos subsistemas no estado seguinte. Esta asserção foi denominada por **loadFTOrdersConsistent**. A asserção relativa à manutenção de factos,

denominada **loadFTOrdersDataMaintenance**, estabelece que o predicado com dois quaisquer estados consecutivos mantém os factos presentes na tabela de factos do estado inicial no estado final. As duas asserções modeladas estão apresentadas na Figura 73.

```
1.  /*
2.   * Assertion: loadFTOrdersConsistent
3.   * Consistency of staging area and consistency of DW in a previous state and loadTFOrders
4.   * predicate  with previous and next state implies the consistency of staging area and DW
5.   * in the next state.
6.   */
7.  assert loadFTOrdersConsistent
8.  {
9.      all s: State, s': s.next | ( consistentStagingArea[s] and consistentDW[s] and loadFTOrders[s,s'] )
10.         implies (consistentStagingArea[s'] and consistentDW[s])
11. }
12.
13. /*
14.  * Assertion: loadFTOrdersDataMaintenance
15.  * Predicate loadFTOrders with an initial state and a final state keeps existing facts from FTOrders
16.  * (in the initial state) in the final state.
17.  */
18. assert loadFTOrdersDataMaintenance
19. {
20.     all s: State, s': s.next | loadFTOrders[s,s'] implies
21.         ( (StagingArea/DW/FTOrders.rows).s in (StagingArea/DW/FTOrders.rows).s')
22. }
```

Figura 73 - Validação de loadFTOrders com asserções loadFTOrdersConsistent e loadFTOrdersDataMaintenance

As asserções modeladas foram verificadas dentro de um *scope* limitado. Isto significa que, neste caso, não foi encontrado nenhum contraexemplo e que a especificação pode estar consistente.

6.2 Especificação do Processo ETL

Para a modelação do processo ETL como um todo foi definido um novo módulo – **ETLExecution**. Este módulo incluí o módulo com a modelação individual das tarefas – **ETL** –, utilizando cada predicado definido como meio para analisar a execução completa do processo. Adotando a abordagem de modelação do processo enquanto uma máquina de estados, foi declarado um facto

para estabelecer as transições possíveis entre estes. Este facto define os possíveis caminhos do sistema (Figura 74).

```
1.  /*
2.   * Fact: valid_path_prefixes
3.   * Defines a specified initial state and restricts all state evolutions to the defined predicates.
4.   */
5.  fact valid_path_prefixes {
6.     init[first]
7.     all s : State - last | CDCSakilaBooks[s,s.next] iff not CDCSakilaCustomers[s,s.next] iff not
8.        CDCSakilaOrders[s,s.next] iff not CDCEbookStoreBooks[s,s.next] iff not
9.        CDCEbookStoreCustomers[s,s.next] iff not CDCEbookStoreOrders[s,s.next] iff not
10.       DQEBooks[s,s.next] iff not DQECustomers[s,s.next] iff not DQEOrders[s,s.next] iff not
11.       DCIBooks[s,s.next] iff not DCICustomers[s,s.next] iff not SKP[s,s.next]
12.       iff not SCDBooks[s,s.next] iff not SCDCustomers[s,s.next] iff not loadFTOrders[s,s.next]
13. }
```

Figura 74 - Modelo em Alloy do facto valid_path_prefixes

O facto declarado utiliza o predicado **init[s: State]** – que estabelece a consistência da área de retenção e do *data warehouse* no estado inicial e a cópia dos dados das diferentes tabelas nas tabelas de carregamento apropriadas na área de retenção. Adicionalmente, o facto define que para todos os estados um dos predicados especificados (que modelam as tarefas de CDC, DQE, DCI, SKP, carregamento de dados de SCD e carregamento de factos) é satisfeito – concretizando uma transição entre o estado **s** e o seu estado seguinte.

Com a inclusão do facto descrito, tem-se a garantia que não existem manipulações adicionais aos dados envolvidos no sistema, e que este apenas evolui pelos predicados modelados. Os predicados que modelam as tarefas ETL foram previamente validados, o que significa que, definindo o sistema do módulo como uma sequência de um subconjunto desses predicados, herda-se, por implicação, um conjunto de propriedades já validadas. Independentemente disso, é possível adicionar validações diretas ao sistema do módulo atual – como, por exemplo, o reforço da garantia da característica histórica do *data warehouse*. Para tal, é definida uma asserção que garante que, em qualquer situação, os dados na tabela de factos no primeiro estado estão presentes no estado final. Para os dados das tabelas de dimensão, foi expressa a propriedade de que os registos destas tabelas no estado inicial, estão presentes (ou uma versão atualizada dos mesmos) nas tabelas de dimensão no estado final. Esta asserção foi denominada **DWDataMaintenance** (Figura 75).

```
1.  /*
2.   * Assertion: DWDataMaintenance
3.   * All entries of fact table in the initial state remain in the final state. All entries of dimension tables
4.   * (SCD) in the initial state remain in the final state (untouched or updated).
5.   */
6.  assert DWDataMaintenance
7.  {
8.     all d: (ETL/StagingArea/DW/FTOrders.rows).first | d in (ETL/StagingArea/DW/FTOrders.rows).last
9.     all d: (ETL/StagingArea/DW/DimBook.rows).first | ( d in (ETL/StagingArea/DW/DimBook.rows).last or
10.       some updated_d: (ETL/StagingArea/DW/DimBook.rows).last | updated_d.id = d.id )
11.    all d: (ETL/StagingArea/DW/DimCustomer.rows).first | ( d in (ETL/StagingArea/DW/DimCustomer.rows).last or
12.       some updated_d: (ETL/StagingArea/DW/DimCustomer.rows).last | updated_d.id = d.id )
13. }
```

Figura 75 - Modelo em Alloy da asserção DWDataMaintenance do módulo ETLExecution

Análise de soluções

No sentido de analisar uma execução do processo completo de ETL foram utilizados dois predicados adicionais. O primeiro predicado auxiliar utilizado foi **mandatory_jobs** (Figura 76), que simula a obrigatoriedade de execução de todas as tarefas definidas. Este predicado é satisfeito quando cada predicado que modela uma tarefa ETL for satisfeito por algum dos estados do sistema.

```
1.  /*
2.   * Predicate: mandatory_jobs
3.   * Predicate holds when some state satisfies each of the defined predicates.
4.   */
5.  pred mandatory_jobs
6.  {
7.     some s : State - last { CDCSakilaBooks[s,s.next] }
8.     some s : State - last { CDCSakilaCustomers[s,s.next] }
9.     some s : State - last { CDCSakilaOrders[s,s.next] }
10.    some s : State - last { CDCEbookStoreBooks[s,s.next] }
11.    some s : State - last { CDCEbookStoreCustomers[s,s.next] }
12.    some s : State - last { CDCEbookStoreOrders[s,s.next] }
13.    some s : State - last { DQEBooks[s,s.next] }
14.    some s : State - last { DQECustomers[s,s.next] }
15.    some s : State - last { DQEOrders[s,s.next] }
16.    some s : State - last { SCDBooks[s,s.next] }
17.    some s : State - last { SCDCustomers[s,s.next] }
18.    some s : State - last { loadFTOrders[s,s.next] }
19. }
```

Figura 76 - Modelo em Alloy do predicado mandatory_jobs do módulo ETLExecution

Adicionalmente foi modelado um predicado para a especificação das dependências entre as tarefas ETL – **dependencies** (Figura 77). Este permite aplicar restrições quanto à ordem dos predicados definidos, traduzindo as dependências entre os diferentes componentes. Nomeadamente:

- O predicado DQE de uma dimensão ou tabela de factos sucede aos predicados de CDC dos seus dados a partir das fontes envolvidas.

- O predicado DCI de uma dimensão sucede ao predicado DQE da mesma.

- O predicado SKP sucede às tarefas de DCI das dimensões.

- O predicado SCD de uma dimensão sucede ao predicado DCI da mesma.

- O predicado de carregamento da tabela de factos (**loadFTOrders)** sucede aos predicados de carregamento de dados das dimensões (SCD).

```
1.  /*
2.   * Predicate: dependencies
3.   * Defines dependencies between predicates.
4.   * Predicate holds when all dependencies are respected.
5.   */
6.  pred dependencies
7.  {
8.     // DQEBooks only happens after CDC of Books
9.     all s: State - last | DQEBooks[s,s.next] implies (
10.       ( some previous_s : prevs[s] | CDCSakilaBooks[previous_s, previous_s.next] )
11.       and
12.        ( some previous_s : prevs[s] | CDCEbookStoreBooks[previous_s, previous_s.next] )
13.    )
14.    // DQECustomers only happens after CDC of Customers
15.    all s: State - last | DQECustomers[s,s.next] implies (
16.       ( some previous_s : prevs[s] | CDCSakilaCustomers[previous_s, previous_s.next] )
17.       and
18.        ( some previous_s : prevs[s] | CDCEbookStoreCustomers[previous_s, previous_s.next] )
19.    )
20.    // DQEOrders only happens after CDC of Orders
21.    all s: State - last | DQEOrders[s,s.next] implies (
22.       ( some previous_s : prevs[s] | CDCSakilaOrders[previous_s, previous_s.next] )
23.       and
24.       ( some previous_s : prevs[s] | CDCEbookStoreOrders[previous_s, previous_s.next] )
25.    )
26.    // DCIBooks only happens after DQE of Books
```

```
27.     all s: State - last | DCIBooks[s,s.next] implies (
28.         some previous_s : prevs[s] | DQEBooks[previous_s, previous_s.next]
29.     )
30.     // DCICustomers only happens after DQE of Customers
31.     all s: State - last | DCICustomers[s,s.next] implies (
32.         some previous_s : prevs[s] | DQECustomers[previous_s, previous_s.next]
33.     )
34.     // SKP only happens after DCI of Books and Customers
35.     all s: State - last | SKP[s,s.next] implies (
36.         ( some previous_s : prevs[s] | DCIBooks[previous_s, previous_s.next] )
37.         and
38.         ( some previous_s : prevs[s] | DCICustomers[previous_s, previous_s.next] )
39.     )
40.     // SCDBooks only happens after DCI of Books
41.     all s: State - last | SCDBooks[s,s.next] implies (
42.         some previous_s : prevs[s] | DCIBooks[previous_s, previous_s.next]
43.     )
44.     // SCDCustomers only happens after DCI of Customers
45.     all s: State - last | SCDCustomers[s,s.next] implies (
46.         some previous_s : prevs[s] | DCICustomers[previous_s, previous_s.next]
47.     )
48.     // load FTOrders only happens after load of Books and Customers
49.     all s: State - last | loadFTOrders[s,s.next] implies (
50.         ( some previous_s : prevs[s] | SCDBooks[previous_s, previous_s.next] )
51.         and
52.         ( some previous_s : prevs[s] | SCDCustomers[previous_s, previous_s.next] )
53.     )
54. }
```

Figura 77 - Modelo em Alloy do predicado dependencies do módulo ETLExecution

Assim, uma solução capaz de satisfazer o predicado de obrigatoriedade de satisfação de cada tarefa ETL e que respeite as dependências entre elas, é uma solução perfeita para a análise de uma possível execução do sistema ETL em estudo.

Para a obtenção de uma solução foi definido o comando de execução **run** (Figura 78). No entanto, a execução do comando com um *scope* limitado não foi capaz de obter nenhuma solução. Neste sentido, a análise do processo através de soluções geradas por comandos **run** foi inconclusiva.

```
1.  run etl {
2.      dependencies and mandatory_jobs
3.  } for 10
```

Figura 78 - Comando de execução para dependencies e mandatory_jobs do módulo ETLExecution

Capítulo 7.

Conclusões e Trabalho Futuro

Um modelo é definido como uma abstração de um sistema, tendo como propósito perceber o mesmo antes da sua construção ou desenvolvimento, omitindo detalhes não essenciais (Rumbaugh et al., 1991). Por se tratar de uma simplificação do sistema, um modelo é mais fácil de manipular do que a entidade original. A chave de um bom modelo, segundo (Jackson, 2002b), é a abstração aplicada, que permite capturar os aspetos importantes e eliminar o ruído dos aspetos sem importância. A modelação de componentes ETL revela-se um desafio com certo grau de dificuldade. Se, por um lado, é vantajoso aplicar abstrações para simplificar o sistema no contexto de modelação, por outro, as tarefas associadas ao processo ETL frequentemente exigem uma atenção ao detalhe e envolvem especificidades próprias de cada sistema.

Neste trabalho de dissertação foi utilizada a linguagem de especificação Alloy para descrever um sistema ETL de um caso de estudo simples. A linguagem foi analisada quanto à sua adequação ao desafio de modelação, provando ser uma boa candidata à especificação de sistemas ETL. Um dos fatores que contribuiu para a seleção desta linguagem foi o estudo realizado por Oliveira et al. (2016) – um primeiro esforço em utilizar a linguagem Alloy para descrever padrões ETL.

A modelação do caso de estudo foi realizada segundo uma abordagem específica. Foram adotadas metodologias para a descrição dos sistemas — e.g. fontes utilizadas, *data warehouse* e área de

retenção — de forma simples e realista. Como consequência, foi eliminado algum ruído associado a uma modelação mais genérica, conduzindo a especificações maioritariamente simples e a uma análise fácil de soluções com cardinalidade baixa. Nesta especificação, as tarefas ETL do caso selecionado foram modeladas enquanto predicados de transição entre dois estados. As tarefas foram analisadas e validadas, e permitiram a especificação final do processo ETL enquanto um *workflow*, definindo-se as dependências entre as tarefas envolvidas.

Problemas encontrados

Um dos problemas associados à modelação específica do caso de estudo foi a quantidade total de assinaturas declaradas. As tabelas envolvidas no processo foram definidas como assinaturas *singleton*, o que significa que existe um elemento da assinatura sempre presente no sistema. Esta modelação conduziu a uma representação constante de todas as tabelas no modelo, mas provocou um aumento da complexidade do mesmo. O aumento de complexidade levou a uma degradação da performance das análises executadas, especialmente visível na ausência de soluções de uma execução completa para o processo ETL. Adicionalmente, outros predicados mais complexos foram afetados ao nível da complexidade de leitura da sua análise.

Com o grau de abstração utilizado nas especificações, perdeu-se, ainda, a capacidade de modelar algumas operações atómicas – nomeadamente transformações textuais, como a concatenação de atributos que foi utilizada como uma forma de conformação inicial no processo de CDC, e a capitalização de valores utilizada no processo de DQE. Como consequência, a especificação destas operações foi impossibilitada.

Adicionalmente, uma das maiores dificuldades verificadas na utilização da linguagem Alloy relacionou-se com a descrição do estado final de uma operação a nível de tabelas (conjunto de entradas). Operações desenhadas a nível mais atómico (por exemplo, aplicadas a uma só entrada) aproximam-se de uma abordagem processual, usualmente associada às tarefas realizadas. A descrição do resultado de uma operação a nível de tabelas foi um exercício mais complexo, uma vez que foi necessário ter em consideração o resultado final de diversas manipulações.

O desenvolvimento segundo uma abordagem específica do caso de estudo levou a um modelo que não é reutilizável para diferentes casos. No entanto, a modelação obtida desmascara semelhanças

entre tarefas aplicadas em diferentes contextos – como tarefas associadas a duas dimensões similares. Estas semelhanças revelam que uma modelação genérica das tarefas seria um avanço com grande valor.

Resultado final e generalização da modelação em Alloy

Uma especificação genérica de tarefas ETL pode tomar como base os predicados modelados de forma específica no presente trabalho de dissertação. No entanto, esforços realizados na generalização dos mesmos revelaram que a análise resultante é mais complexa e ruidosa. Adicionalmente, a generalização da modelação não resolveu o problema de complexidade encontrado, especialmente quando se tenta especificar um sistema ou cenário realista.

Com as decisões tomadas ao longo do desenvolvimento da presente dissertação, teve-se como resultado final uma especificação clara para um caso de estudo específico. A escrita de asserções foi um exercício direto e permitiu a validação da consistência do *data warehouse* e a correção das tarefas definidas. A natureza dinâmica do sistema foi devidamente capturada, o que tornou possível planear o processo ETL enquanto um todo. Contudo, algumas especificidades de comportamentos ETL e o facto de a modelação envolver a manipulação de dados a nível de tabelas deram origem a partes de especificação mais complexas. Prevemos que a aplicação da mesma metodologia a um caso mais complexo enfatizaria ainda mais as dificuldades encontradas.

Trabalho futuro

A inclusão de formalismo no planeamento e desenvolvimento de processos ETL revela-se, cada vez mais, não só desejável, mas também imprescindível para o sucesso global no desenvolvimento de sistemas de *data warehousing*. Para este formalismo são necessárias a definição dos processos envolvidos e a seleção de uma ferramenta adequada. A utilização de diferentes linguagens ou notações para especificar processos ETL pode ditar o futuro da abordagem quanto à sua legibilidade, usabilidade e reusabilidade, nível de formalismo, instanciação e adesão da comunidade. A modelação apresentada neste trabalho de dissertação apresenta falhas quanto à sua reusabilidade e instanciação. O caso de estudo utilizado foi um componente essencial ao desenvolvimento da modelação, contudo restringiu de forma clara a reusabilidade da mesma.

Apesar dos problemas encontrados, acreditamos que a Alloy se mantém como uma boa candidata para a especificação formal de subsistemas ETL. A sua natureza formal e a abstração com a qual se associa cria alguma distância entre a modelação e a aplicação real de tarefas ETL. No entanto, estas características, e o facto de ser uma linguagem analisável, permitem a validação de propriedades de segurança e de vivacidade do sistema, trazendo uma segurança difícil de alcançar de outra forma.

Tendo em conta as conclusões retiradas do presente trabalho de dissertação, não seria surpresa se Alloy tivesse uma presença marcante no futuro do desenvolvimento ETL. Contudo, a sua utilização, a metodologia e a abordagem adotadas teriam que ser as ideais para o contexto destes processos. Consideramos que dar continuidade ao estudo da aplicação ou utilização da Alloy para a modelação de processos ETL é um passo interessante e merecedor de atenção.

Bibliografia

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2007). UML2Alloy: A challenging model transformation. *10th International Conference on Model Driven Engineering Languages and Systems*, 1–15. https://doi.org/10.1007/978-3-540-75209-7_30

Andoni, A., Daniliuc, D., Khurshid, S., & Marinov, D. (2003). Evaluating the "small scope hypothesis." *Unpublished*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.4000&rep=rep1&type=pdf

Costa, A. M. P. M. da. (2006). *A gestão da qualidade dos dados em ambientes de data warehousing na prossecução da excelência da informação*. University of Minho.

Cunha, A., Garis, A., & Riesco, D. (2013). Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software and Systems Modeling*, *14*(1), 5–25. https://doi.org/10.1007/s10270-013-0353-5

Cunha, A., & Pacheco, H. (2009). Mapping between alloy specifications and database implementations. In *SEFM 2009 - 7th IEEE International Conference on Software Engineering and Formal Methods* (pp. 285–294). https://doi.org/10.1109/SEFM.2009.27

Dürr, E., & van Katwijk, J. (1992). VDM++, a formal specification language for object-oriented designs. *Computer Systems and Software Engineering*. https://doi.org/10.1109/CMPEUR.1992.218511

El Akkaoui, Z., & Zimanyi, E. (2009). Defining ETL workflows using BPMN and BPEL. *DOLAP '09 Proceedings of the ACM Twelfth International Workshop on Data Warehousing and OLAP*, 41–48. https://doi.org/10.1145/1651291.1651299

English, L. P. (1999). *Improving data warehouse and business information quality: methods for reducing costs and increasing profits* (Vol. 1). Wiley New York.

Garland, S. J., Guttag, J. V, & Horning, J. J. (1993). An Overview of Larch. *Functional*

Programming, Concurrency, Simulation and Automated Reasoning. International Lecture Series 1991--1992, *693*, 329–348.

Guttag, J. V., & Horning, J. J. (1986). Report on the larch shared language. *Science of Computer Programming*, *6*(C), 103–134. https://doi.org/10.1016/0167-6423(86)90021-3

Inmon, W. H. (2005). *Building the data warehouse*. John wiley & sons.

International Organization for Standardization, & International Electrotechnical Commission. (2002). Information technology--Z formal specification notation--Syntax, type system and semantics, *2002*, 101. Retrieved from ftp://195.215.30.152/jtc1/sc22/def/n3399.pdf

Jackson, D. (2002a). Alloy : A Lightweight Object Modelling Notation, *11*(2), 256–290.

Jackson, D. (2002b). *Micromodels of Software: Lightweight modelling and analysis with Alloy*. Cambridge: MIT Lab for Computer Science.

Jackson, D. (2006). Dependable software by design. *Scientific American*. https://doi.org/10.1038/scientificamerican0606-68

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis* (Revised Ed). The MIT Press.

Kimball, R., & Caserta, J. (2004). The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data.

Kimball, R., Reeves, L., Ross, M., & Thornthwaite, W. (1998). *The data warehouse lifecycle toolkit: expert methods for designing, developing, and deploying data warehouses*. John Wiley & Sons.

Mišić, V. B., & Velašević, D. (1997). Formal Specifications in Software Development: An Overview. *Yugoslav Journal for Operations Research*, *7*(1), 79–96.

Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., & Larsen, P. G. (2000). Exploring timing properties using VDM++. In *Proceedings of the Second VDM Workshop*.

Oliveira, B., & Belo, O. (2013). Approaching ETL conceptual modelling and validation using BPMN and BPEL. *DATA 2013 - Proceedings of the 2nd International Conference on Data Technologies and Applications*, (section 2), 191–198. Retrieved from http://www.scopus.com/inward/record.url?eid=2-s2.0-84887165580&partnerID=40&md5=26111d4512723b63b508edb1f613ac80

Oliveira, B., Belo, O., & Macedo, N. (2016). Towards a formal validation of ETL patterns behavior. *4rd International Conference on Data Management Technologies and Applications*. https://doi.org/10.1007/978-3-319-45547-1_13

Plat, N., & Larsen, P. G. (1992). An overview of the ISO/VDM-SL standard. *ACM Sigplan Notices*,

27(8), 76–82.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E., & others. (1991). *Object-oriented modeling and design* (Vol. 199). Prentice-hall Englewood Cliffs, NJ.

Sannella, D., & Wirsing, M. (1999). Specification Languages BT. In E. Astesiano, H.-J. Kreowski, & B. Krieg-Brückner (Eds.), *Algebraic Foundations of Systems Specification* (pp. 243–272). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-59851-7_8

Santos, V. N. C. dos. (2015). *A Relational Algebra Approach to ETL Modeling*. University of Minho.

Spivey, J. M. (1989). An introduction to Z and formal specifications. *Software Engineering Journal*, *4*(1), 40–50.

Sufrin, B. A. (1986). *Z handbook, draft 1.1*. Oxford, UK: Programming Research Group.

Trujillo, J., & Luján-Mora, S. (2003). A UML based approach for modeling ETL processes in data warehouses. *Conceptual Modeling-ER 2003*, *2813*, 307–320. https://doi.org/ETL processes, Data warehouses, conceptual modeling, UML

Vassiliadis, P., Simitsis, A., & Skiadopoulos, S. (2002). Conceptual Modeling for ETL Processes. *Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP - DOLAP '02*, 1–25. https://doi.org/10.1145/583890.583893

Wallace, C. (2003). Using Alloy in Process Modelling. *Information and Software Technology*, *45*, 1031–1043. https://doi.org/10.1016/S0950-5849(03)00131-9

Anexos

1. Modelo completo em Alloy do predicado CDCEbookStoreBooks

```
1.  /*
2.   * Predicate: CDCEbookStoreBooks
3.   * Receives two states (previous and next). The next state has the AuditBooks table with the new audit
4.   * books from EbookStore and an updated EbookStoreBookPreviousLoad table.
5.  */
6.  pred CDCEbookStoreBooks[s, s': State]
7.  {
8.     // Pre-conditions
9.     (EbookStoreBookCurrentLoad.rows).s = (StagingArea/EbookStore/Books.rows).s
10.
11.    // Post-conditions
12.    // Calculate differences between current and previous load and feed AuditBooks accordingly.
13.    // D - All books from PreviousLoad which are not present in CurrentLoad in the previous state s
14.    // generate an AuditBook entry with Delete operation and EbookStore source in the next state s'
15.    all pl : (EbookStoreBookPreviousLoad.rows).s |
16.       ( no cl : (EbookStoreBookCurrentLoad.rows).s | cl.id = pl.id )
17.       implies
18.       ( one a' : (AuditBooks.rows).s' | a'.id = pl.id and a'.title = pl.title and a'.ISBN = pl.ISBN and
19.          a'.description = pl.summary and a'.timestamp = pl.last_update and a'.source = EbookStore and
20.          a'.op = D and a'.title_changed = False and a'.ISBN_changed = False and a'.description_changed = False
21.       )
22.
23.    // I - All books from CurrentLoad which are not present in PreviousLoad in the previous state s
24.    // generate an AuditBook entry with Insert operation and EbookStore source in the next state s'
25.    all cl : (EbookStoreBookCurrentLoad.rows).s |
26.       ( no pl : (EbookStoreBookPreviousLoad.rows).s | cl.id = pl.id )
27.       implies
28.       ( one a' : (AuditBooks.rows).s' | a'.id = cl.id and a'.title = cl.title and (...) and
29.          a'.source = EbookStore and a'.op = I and
30.          a'.title_changed = True and a'.ISBN_changed = True and a'.description_changed = True
31.       )
```

```
32.
33.     // U - All books from CurrentLoad which are present in PreviousLoad in the previous state s, but with
34.     // different values, generate an AuditBook with Update operation in the next state s'
35.     all cl : (EbookStoreBookCurrentLoad.rows).s |
36.         ( some pl : (EbookStoreBookPreviousLoad.rows).s | cl.id = pl.id and
37.             (cl.title != pl.title or cl.summary != pl.summary or cl.ISBN != pl.ISBN)
38.         )
39.         implies
40.         ( one a' : (AuditBooks.rows).s', pl : (EbookStoreBookPreviousLoad.rows).s |
41.             cl.id = pl.id and (cl.title != pl.title or cl.summary != pl.summary or cl.ISBN != pl.ISBN) and
42.             a'.id = cl.id and a'.title = cl.title and (...) and a'.source = EbookStore and a'.op = U and
43.             (
44.                 ( cl.ISBN != pl.ISBN )
45.                 implies
46.                 ( a'.ISBN_changed = True and a'.title_changed = True and a'.description_changed = True)
47.                 else
48.                 (  a'.ISBN_changed = False and
49.                     (cl.title != pl.title implies a'.title_changed = True else a'.title_changed = False) and
50.                     (cl.summary != pl.summary implies a'.description_changed = True else a'.description_changed = False)
51.                 )
52.             )
53.         )
54.
55.     // AuditBooks: s -> s' (entries present in s are also present in s')
56.     all a: (AuditBooks.rows).s | a in (AuditBooks.rows).s'
57.
58.     // AuditBooks: s' -> s (entries present in s' are either present in s or were generated by a calculated difference from s)
59.     all a' : (AuditBooks.rows).s' | ( a' in (AuditBooks.rows).s ) iff not
60.         (
61.             ( one pl: (EbookStoreBookPreviousLoad.rows).s | a'.id = pl.id and (...) and
62.                 a'.source = EbookStore and a'.op = D and a'.title_changed = False and (...) and
63.                 ( no cl : (EbookStoreBookCurrentLoad.rows).s | cl.id = pl.id )
64.             ) or
65.             ( one cl: (EbookStoreBookCurrentLoad.rows).s | a'.id = cl.id and (...) and
66.                 a'.source = EbookStore and a'.op = I and a'.title_changed = True and (...) and
67.                 ( no pl : (EbookStoreBookPreviousLoad.rows).s | cl.id = pl.id )
68.             ) or
69.             ( one cl: (EbookStoreBookCurrentLoad.rows).s |
70.                 some pl : (EbookStoreBookPreviousLoad.rows).s |
71.                     cl.id = pl.id and (cl.title != pl.title or cl.summary != pl.summary or cl.ISBN != pl.ISBN) and
72.                     a'.id = cl.id and (...) and a'.source = EbookStore and a'.op = U and
73.                     (
74.                         ( cl.ISBN != pl.ISBN )
75.                         implies
76.                         ( a'.ISBN_changed = True and a'.title_changed = True and a'.description_changed = True)
77.                         else
78.                         (   a'.ISBN_changed = False and
79.                             (cl.title != pl.title implies a'.title_changed = True else a'.title_changed = False) and
80.                             (cl.summary != pl.summary implies a'.description_changed = True else a'.description_changed = False)
81.                         )
```

```
82.                     )
83.                )
84.           )
85.
86.       (EbookStoreBookPreviousLoad.rows).s' = (EbookStoreBookCurrentLoad.rows).s
87.       (EbookStoreBookCurrentLoad.rows).s' = none
88.
89.       // Frame-conditions
90.       StagingArea/frameExcept[AuditBooks, s, s']
91.       StagingArea/DW/frame[s,s']
92.   }
```

2. Modelo completo em Alloy do predicado DCIBooks

```
1.    /*
2.     * Predicate: DCIBooks
3.     * Receives two states (previous and next). The next state has the audit books
4.     * from previous state in AuditBook but with replaced surrogate keys.
5.     * Uses EquiBooks to look up, insert or update a record with a surrogate key and
6.     * the appropriate natural key from audit book. If the last operation of a certain
7.     * book (identified by its equi field value - ISBN ) is delete, the surrogate key will
8.     * be looked up but the appropriate natural key removed.
9.     * Uses AuditBook's isLastRecord and addSK predicates and EquiBook's
10.    * SK function and newSK predicate (to lookup and to generate a surrogate key).
11.    */
12.   pred DCIBooks[s, s': State]
13.   {
14.     // Pre-conditions
15.     AuditBooks.consistent[s]
16.     AuditBooks.valid[s]
17.     EquiBooks.consistent[s]
18.
19.     -- Unique operation on source element at a time
20.     all disj ab1, ab2: (AuditBooks.rows).s | ( ab1.id = ab2.id and ab1.source = ab2.source ) implies
21.       ab1.timestamp != ab2.timestamp
22.
23.     // Post-conditions (replace id with surrogate key on AuditBooks and use (insert,
24.     // update, or do nothing) EquiBooks)
25.     // AuditBooks: specify from s -> s' + s' <- s (what happens to post books of
26.     // previous state and what has happened to post books of next state)
27.      // EquiBooks: specifiy from s -> s' + s' <- s (what happens to equi books
28.     // of previous state and what has happened to equi books of next state)
29.
30.     // AuditBooks: s -> s'
31.     // (Part 1: ) For all audit books of the previous state, there is an audit book
32.     // in the next state with same values and the id replaced by the surrogate key
33.     // of an existing equi book of the previous state.
34.     // The audit books of the previous state are not present in the next state, and
```

```
35.   // are replaced by new audit books (simulating an update operation).
36.   // The equi book used to get the surrogate key has the same isbn and source
37.   // value as the audit book that generated it, and its uses an existing surrogate
38.   // key for its isbn (if possible) - otherwise uses a brand new surrogate key, used
39.   // only by equi books with same isbn. The natural key of the equi book used is
40.   // dictated by the last audit book with the same source and id values - if the last
41.    // operation of the said book id from the said source has a different isbn value
42.   // or is a delete  operation, then the equi book will have no natural_key value,
43.   // otherwise it will have the said id.
44.   all ab: (AuditBooks.rows).s |
45.     ab not in (AuditBooks.rows).s'    and
46.     (
47.       (
48.         ( one eb': (EquiBooks.rows).s' | eb'.ISBN = ab.ISBN and eb'.source = ab.source and
49.           -- surrogate_key:
50.           (
51.           -- if an equi with same isbn value exists, eb' will use its surrogate key
52.           -- otherwise, eb' uses a complety new surrogate key value (not used by
53.           -- any equi in the previous state and not used by any equi in the next state
54.           -- with a different isbn, which will be the surrogate key of all entries with same ISBN)
55.             ( one EquiBooks.SK[s, ab.ISBN] )
56.             implies ( eb'.surrogate_key = EquiBooks.SK[s, ab.ISBN] )
57.             else ( EquiBooks.newSK[s, s', eb'.surrogate_key, ab.ISBN ] )
58.           )
59.           and
60.           -- natural_key:
61.           (
62.           -- Define natural key value on eb' according to last occurence of id,source in audit
63.           --           books. If last occurrence of id, source in audit books has the same ISBN and is
64.           -- not a delete operation, eb' should hold the audit book id has natural key.
65.           -- Otherwise, eb' should be marked as deprecated (hold no nk value)
66.             ( some last_ab : (AuditBooks.rows).s | last_ab.id = ab.id and last_ab.source = ab.source and
67.               last_ab.isLastRecord[(AuditBooks.rows).s] and
68.               last_ab.ISBN = ab.ISBN and last_ab.op != D
69.             )
70.             implies ( eb'.natural_key = ab.id )
71.             else ( no eb'.natural_key )
72.           )
73.         )
74.         implies
75.         ( no disj e1,e2 : (EquiBooks.rows).s' | e1.ISBN = e2.ISBN and e1.source = e2.source )
76.       )
77.       implies
78.       (
79.         one ab': (AuditBooks.rows).s' |
80.           one eb': (EquiBooks.rows).s' | eb'.ISBN = ab.ISBN and eb'.source = ab.source and
81.           (
82.             ( one EquiBooks.SK[s, ab.ISBN] )
83.             implies ( eb'.surrogate_key = EquiBooks.SK[s, ab.ISBN] )
84.             else ( EquiBooks.newSK[s, s', eb'.surrogate_key, ab.ISBN ] )
```

```
85.           ) and
86.           (
87.             ( some last_ab : (AuditBooks.rows).s | last_ab.id = ab.id and last_ab.source = ab.source and
88.               last_ab.isLastRecord[(AuditBooks.rows).s] and
89.               last_ab.ISBN = ab.ISBN and last_ab.op != D
90.             )
91.             implies ( eb'.natural_key = ab.id )
92.             else ( no eb'.natural_key )
93.           ) and
94.            ab' not in (AuditBooks.rows).s and ab'.addSK[ab, eb']
95.       )
96.       else
97.       (
98.         one qb': (QuaBooks.rows).s' | qb'.copyAudit[ab, True] and qb'.reason = ConciliationError
99.       )
100.
101.  )
102.
103.  // (Part 2: ) Deprecate outdated equi books - for any equi book from previous
104.  // state that match the source and id of an audit book from the previous state
105.  // which last record is a delete operation or has a different ISBN value will be
106.  // removed and fixed.
107.  // For all audit books of the previous state, any equi books that hold its id as
108.  // natural key value and have the same source value and it happens to exist
109.  // an audit book with same source and id that is the last record on the audit
110.  // books of the previous state and is a delete operation or has a different isbn
111.  // value, then said equi books will be removed and new fixed equi books will be
112.  // in the next state (with same values as the wrong equi books but no natural key
113.  // value).
114.  all ab: (AuditBooks.rows).s |
115.      -- when a wrong equi row exists in the previous state, with same source and
116.      -- natural key as ab but different isbn than the last audit book, or the last
117.      -- post book is a delete operation, the row will be removed and fixed
118.      all wrong_eb: (EquiBooks.rows).s |
119.        (
120.          wrong_eb.source = ab.source and wrong_eb.natural_key = ab.id and
121.          ( some last_ab : (AuditBooks.rows).s | last_ab.id = ab.id and last_ab.source = ab.source and
122.            last_ab.isLastRecord[(AuditBooks.rows).s] and ( last_ab.ISBN != wrong_eb.ISBN
123.            or last_ab.op = D )
124.          )
125.        )
126.        implies
127.        (
128.          wrong_eb not in (EquiBooks.rows).s' and
129.          (
130.            one fixed_eb': (EquiBooks.rows).s' | fixed_eb'.surrogate_key = wrong_eb.surrogate_key
131.              and fixed_eb'.ISBN = wrong_eb.ISBN and fixed_eb'.source = wrong_eb.source and
132.              no fixed_eb'.natural_key
133.          )
134.        )
```

```
135.
136.
137.  // AuditBooks: s' <- s
138.  // AuditBooks on the next state contains entries from the previous state with
139.  // replaced id
140.  all ab': (AuditBooks.rows).s' |
141.    ( one ab: (AuditBooks.rows).s, eb' : (EquiBooks.rows).s' |
142.      ab'.copy[ab, False] and eb'.ISBN = ab'.ISBN and eb'.source = ab.source and ab'.id = eb'.surrogate_key
143.    )
144.
145.  // EquiBooks: s -> s'
146.  // EquiBooks entries of the previous state that were not involved in the proccess
147.  // are kept in the next state
148.  all eb : (EquiBooks.rows).s |
149.    ( eb in (EquiBooks.rows).s') iff not
150.    ( one eb': (EquiBooks.rows).s' |
151.      eb.source = eb'.source and eb'.ISBN = eb.ISBN and eb'.surrogate_key = eb.surrogate_key and
152.      (
153.        -- added natural key
154.        ( no eb.natural_key and
155.          ( one ab: (AuditBooks.rows).s | ab.source = eb.source and ab.ISBN = eb.ISBN and
156.            ab.isLastRecord[(AuditBooks.rows).s] and ab.op != D and eb'.natural_key = ab.id
157.          )
158.        ) or
159.        -- removed natural key
160.        ( no eb'.natural_key and
161.          ( one ab: (AuditBooks.rows).s | ab.source = eb.source and ab.id = eb.natural_key and
162.            ab.isLastRecord[(AuditBooks.rows).s] and (ab.op = D or ab.ISBN != eb.ISBN )
163.          )
164.        )
165.      )
166.    )
167.
168.  // EquiBooks: s' <- s
169.  // EquiBooks entries of the next state are either present in the previous one or
170.  // were used in process by an audit book with same id, src
171.  all eb' : (EquiBooks.rows).s' |
172.    -- eb' is present in the previous state
173.    eb' in (EquiBooks.rows).s or
174.    -- eb' is needed by some audit book
175.    (
176.      -- eb' is is a correctly generated equi row (uses existing surrogate key for
177.      -- its isbn (if possible) or a completly new surrogate key )
178.      (
179.        ( one EquiBooks.SK[s, eb'.ISBN] )
180.        implies
181.          eb'.surrogate_key =  EquiBooks.SK[s, eb'.ISBN]
182.        else
183.          EquiBooks.newSK[s, s', eb'.surrogate_key, eb'.ISBN ]
184.      )
```

```
185.    and
186.    (
187.      -- Case 1: eb' is a deprecated eb from s
188.      (
189.        no eb'.natural_key and
190.        (
191.          -- eb existed in s, with same values as eb' (except for natural_key)
192.          -- but it was deprecated by an audit book (with same source and natural_key)
193.          -- that changed ISBN value
194.          some eb: (EquiBooks.rows).s |
195.            -- same values as eb'
196.            eb.source = eb'.source and eb.ISBN = eb'.ISBN and
197.            eb.surrogate_key = eb'.surrogate_key and
198.            -- eb natural key is not passed to eb' because of a change of isbn on the
199.            --  last audit book record
200.            (
201.              some last_ab : (AuditBooks.rows).s | last_ab.source = eb.source and
202.                last_ab.id = eb.natural_key and last_ab.isLastRecord[(AuditBooks.rows).s]
203.                and (last_ab.ISBN != eb'.ISBN or last_ab.op = D)
204.            )
205.        )
206.      )
207.      or
208.      -- Case 2: eb' is a created + deprecated eb'
209.      (
210.        no eb'.natural_key and
211.        (
212.          -- audit book with same isbn and source existed in s, but the most recent audit
213.          -- book for its id and source had different ISBN value or has a delete operation
214.          some ab: (AuditBooks.rows).s |
215.            -- same source and isbn
216.            ab.source = eb'.source and ab.ISBN = eb'.ISBN and
217.            (
218.              some last_ab : (AuditBooks.rows).s | last_ab.source = ab.source and
219.                last_ab.id = ab.id and last_ab.isLastRecord[(AuditBooks.rows).s] and
220.                (last_ab.ISBN != eb'.ISBN or last_ab.op = D)
221.            )
222.        )
223.      )
224.      or
225.      -- Case 3: eb' is a created eb'
226.      (
227.        some eb'.natural_key and
228.        (
229.          -- audit book with same isbn, natural key and source existed in s, and was the
230.          -- last record of its natural key and was not a delete action PLUS originated a
231.          -- copied audit book in the next state with the eb's surrogate key as id
232.          one ab: (AuditBooks.rows).s | one ab': (AuditBooks.rows).s' |
233.            ab'.addSK[ab, eb'] and ab.source = eb'.source and ab.id = eb'.natural_key and
234.            ab.ISBN = eb'.ISBN and ab.isLastRecord[(AuditBooks.rows).s] and ab.op != D
```

```
235.          )
236.        )
237.      )
238.    )
239.
240.   all qb: (QuaBooks.rows).s | qb in (QuaBooks.rows).s'
241.   all qb': (QuaBooks.rows).s' | (qb' in (QuaBooks.rows).s ) iff not (
242.     one ab: (AuditBooks.rows).s | qb'.copyAudit[ab, True] and qb'.reason = ConciliationError and
243.       not ( one eb': (EquiBooks.rows).s' | eb'.ISBN = ab.ISBN and eb'.source = ab.source
244.           and (
245.             ( one EquiBooks.SK[s, ab.ISBN] )
246.             implies ( eb'.surrogate_key = EquiBooks.SK[s, ab.ISBN] )
247.             else ( EquiBooks.newSK[s, s', eb'.surrogate_key, ab.ISBN ] )
248.           ) and
249.           (
250.             ( some last_ab : (AuditBooks.rows).s | last_ab.id = ab.id and last_ab.source = ab.source and
251.               last_ab.isLastRecord[(AuditBooks.rows).s] and
252.               last_ab.ISBN = ab.ISBN and last_ab.op != D
253.             )
254.             implies ( eb'.natural_key = ab.id )
255.             else ( no eb'.natural_key )
256.           ) and
257.           ( no disj e1,e2 : (EquiBooks.rows).s' | e1.ISBN = e2.ISBN and e1.source = e2.source )
258.       )
259.
260.   )
261.   // Frame-conditions
262.   StagingArea/frameExcept[AuditBooks + EquiBooks + QuaBooks, s, s']
263.   StagingArea/DW/frame[s, s']
264. }
```