4
Implementação

Neste capítulo, são consideradas as decisões de implementação do modelo de bloqueio proposto. Inicialmente são detalhados o design e as estruturas de dados do *Lock Manager*. Por fim, são apresentados dois componentes extras auxiliares: um vocabulário (ontologia) para facilitar a aquisição de bloqueios por parte da aplicação e uma DSL de *workflow* para transações Web.

Com exceção do vocabulário que foi escrito em OWL, todos os outros componentes foram implementados na linguagem Ruby, em função de seu caráter dinâmico e facilidades de metaprogramação.

PUC-Rio - Certificação Digital Nº 0721352/CA

4.1.
Lock Manager

O principal componente é o *Lock Manager*. Um componente centralizado e único (*singleton*), responsável por gerenciar todos os bloqueios adquiridos e liberados pelas transações. O *Lock Manager* implementa todos os passos do protocolo multigranular proposto, ou seja, bloqueios nos grânulos, propagação dos bloqueios planejados, enfim. Para tal, faz uso das três tabelas anteriores: matriz de compatibilidade (Tabela 16), matriz de conversão (Tabela 17) e tabela de downgrade de bloqueio real em bloqueio planejado (Tabela 15). Vale ressaltar que a responsabilidade do *Lock Manager* é apenas gerenciar a aquisição/liberação de bloqueios. Assim sendo, é assumido a existência de um *Transaction Manager* mais amplo, que faz uso deste *Lock Manager*, e é responsável por gerenciar todo o ciclo de vida das transações (início, fim e aborto), bem como prevenir (ou dectectar/recuperar) problemas como *starvation* ou *livelock*.

Basicamente, o *Lock Manager* oferece três operações:

- **`lock(transaction_id, granule, lock_mode, uris)`**
  Estabeler um bloqueio em um item de dado específico. O primeiro parâmetro `transaction_id` é o identificador único da transação

(em geral, um inteiro). `Granule` especifica o tipo de grânulo ("*Graph*", "*Property*", "*Resource*", "*Property of Resource*"). `Lock_mode` define o tipo de bloqueio, dentre todos propostos (p. ex.: *iW*). `uris` é uma tabela hash com as URIs do item a ser bloqueado, de acordo com o tipo de grânulo: para o grânulo "*Graph*", não há URIs; para o grânulo "*Property*", a URI da propriedade; para o grânulo "*Resource*", a URI do recurso; e para o grânulo "*Property of Resource*" devem ser fornecidas a URI da propriedade e a URI do recurso. Para os grânulos que envolvam propriedade, poder ser fornecido também a URI da propriedade inversa, caso haja. As chaves (*symbols* da linguagem Ruby) a serem usadas na tabela hash de URIs são: `:property`, `:resource`; `:inv_property`. Estes e outros pormenores ficarão mais claros no exemplo apresentado na seção 4.1.1 (Design).

- **`unlock(transation_id, granule, uris)`**
  Liberar um bloqueio em um item de dado específico.
- **`unlock_all(transaction_id)`**
  Liberar, de uma só vez, todos os bloqueios da transação especificada.

As estruturas de dados internas do *Lock Manager*, inspiradas pelo *Lock Manager* apresentado em [Weikum e Vossen, 2002], são mostradas na Figura 28.

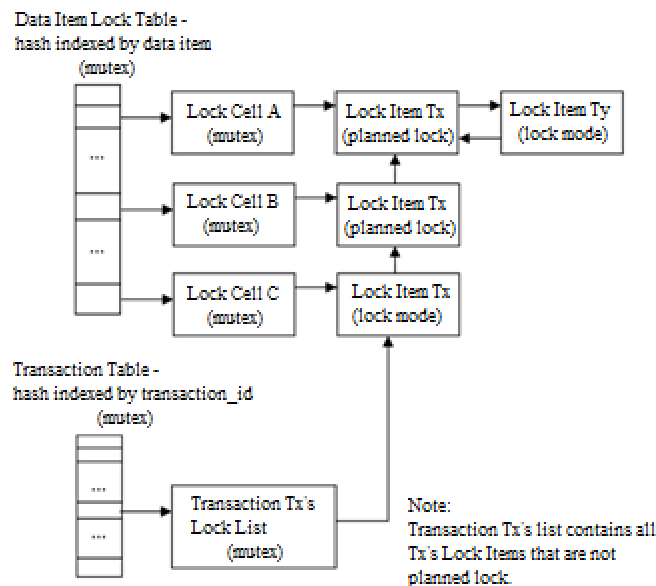

Figura 28 - Estruturas de dados do *Lock Manager*

PUC-Rio - Certificação Digital Nº 0721352/CA

O *Lock Manager* é composto por:

- ***Data Item Lock Table***

  Tabela hash que mapeia o item de dado (grânulo específico) em sua *Lock Cell* (célula de bloqueio). Como será visto adiante, no design do *Lock Manager*, cada item de dado tem sua correspondente *Lock Cell*, em um relacionamento um para um. Cada *Lock Cell* mantém a lista de todos os bloqueios sobre o item de dado (*Data Item Lock List*).

- ***Lock Cell***

  Mantém a lista (*Data Item Lock List*) contendo todos os bloqueios (*Locks Items*) do item de dado. Trata-se de uma lista duplamente encadeada para rapidamente remover o *Lock Item* durante a operação de *unlock* (na verdade é o próprio *Lock Item* que se adiciona e se remove desta lista). A *Lock Cell* aponta para o primeiro elemento desta lista.

- ***Lock Item***

  Lock Item (item de bloqueio) pertencente a uma transação. Para cada bloqueio estabelecido por uma transação sobre um item de dado, é criado um *Lock Item*, contendo: id da transação dona do bloqueio e o tipo de bloqueio *(lock mode)*. Além disso, o *Lock Item* contém um contador de *Lock Items* filho e mantêm referências para os *Lock Items* pai, formando a "hierarquia" de bloqueios multigranular da transação, de acordo com o protocolo proposto. Manter os *Lock Items* de uma transação "hierarquicamente" interligados, ou seja, os filhos conhecerem seus pais, é importante para velozmente propagar a remoção quando um *Lock Item* for removido, durante a execução de um *unlock*. E o contador de filhos controla quando um *Lock Item* (contendo um bloqueio planejado) pode ser removido ou não.

- ***Transaction Table***

  Tabela hash que mapeia o id da transação na sua lista de *Lock Items* (*Transaction Lock List*, a seguir). Esta lista é mantida para rapidamente ter acesso aos *Lock Items* de uma transação, em

PUC-Rio - Certificação Digital Nº 0721352/CA

especial, durante a operação *unlock_all*, quando todos os bloqueios da transação são liberados de uma só vez.

- ***Transaction Lock List***
  Lista de *Lock Items* da transação. Cada transação possui sua correspondente lista de *Lock Items*. Novamente, trata-se de uma lista duplamente encadeada para agilizar a remoção de bloqueios (e do mesmo modo, é o próprio *Lock Item* que se adiciona e se remove desta lista). A *Transaction Lock List* aponta para o primeiro elemento desta lista.

Para evitar condições de corridas nas estruturas de dados do *Lock Manager*, é necessário promover a exclusão mútua dos threads que as usam. Para tal foi criado o seguinte esquema de mutexes (semáforos binários):

- ***Data Item Lock Table Mutex***
  Controla as mudanças estruturais da tabela hash *Data Item Lock Table*. Mudanças estruturais significam inserção, obtenção e remoção. Ou seja, este mutex é usado apenas durante a inserção, obtenção ou remoção do *Lock Cell* de um item de dado específico.

- ***Lock Cell Mutex***
  Controla as mudanças estruturais da lista de *Lock Items* (*Data Item Lock List*) mantida pelo *Lock* Cell de um item de dado específico. Perceba, então, que cada item de dados específico possui seu próprio mutex, que controla sua lista de bloqueios. Isto é para evitar que o *Lock Manager* (especificamente o *Data Item Lock Table Mutex*) se torne um gargalo.

- ***Transaction Table Mutex***
  Similar ao Data Item *Lock* Table Mutex, este mutex serializa o acesso a tabela hash *Transaction Table*.

- ***Transaction Lock List Mutex***
  Similar ao *Lock Cell Mutex*, este mutex controla o acesso a lista de *Lock Items* de cada transação. Este mutex foi criado para cobrir os

PUC-Rio - Certificação Digital Nº 0721352/CA

casos onde a própria transação é multithreaded. Certamente, não é o caso da maioria das transações. Mas foi previsto.

Um ponto importante que cabe ressaltar é que apesar do *Lock Manager* ter sido projetado para o protocolo multigranular proposto, ele oferece dois modos de operação: **MGL** (multigranular) e **NO_MGL** (monogranular). O modo **NO_MGL** desativa a propagação de bloqueios planejados e pode ser usado por aplicações que, por alguma razão, prefiram trabalhar usando um único de tipo de grânulo. Mesmo nestes casos, onde o protocolo multigranular seja desativado, a aplicação continua se beneficiando dos novos tipos de bloqueio (de inserção e remoção) oferecidos. Ambos os modos de operação foram utilizados na simulação, apresentada mais adiante, para efeitos de comparação.

4.1.1. Design

O diagrama da Figura 29 mostra as classes que compõem o design[44] do *Lock Manager*, suas principais propriedades, mutexes e operações. Note que os parâmetros das operações foram suprimidos por razões de espaço.

Figura 29 - Modelo de classes do *Lock Manager*

[44] Veja Apêndice, seção A.2, para código fonte completo do *Lock Manager*.

PUC-Rio - Certificação Digital Nº 0721352/CA

A classe RdfLockManager é o *Lock Manager* propriamente dito. Perceba o estereótipo <<Singleton>> que denota que esta classe tem que ter uma única instância global. O atributo booleano mgl_mode é um flag que controla o modo de operação: *true* - MGL (*default*); *false* - NO_MGL. As tabelas hash *Data Item Lock Table* e *Transaction Table* são expressas por meio das associações qualificadas com as classes RDFLockCell e RdfTransactionLockList, respectivamente. Os qualificadores das associações são as chaves das tabelas hash. O qualificador data_item é a string resultante da concatenação do tipo de grânulo com as URIs que definem um grânulo específico.

A classe RdfLockItem corresponde ao *Lock Item,* que contém as informações de cada bloqueio estabelecido, por uma transação, sobre um item de dado (grânulo específico).

RDFTransactionList é implementação da *Transaction Lock List*, que é a lista duplamente encadeada de *Lock Items* de uma transação. Esta classe aponta para o primeiro elemento da lista (papel *first* do relacionamento com a classe RdfLockItem). O resto do encadeamento é estabelecido por meio dos papéis t_list_successor e t_list_predecessor do auto-relacionamento presente na classe RdfLockItem.

A classe RdfLockCell representa a *Lock Cell* associada a um item de dado. Esta classe aponta para o primeiro elemento da lista encadeada de *Lock Items* do item de dado (papel *first* do relacionamento com a classe RdfLockItem). O resto de encadeamento é realizado por meio dos papéis predecessor e successor no auto-relacionamento da classe RdfLockItem. Estas duas classes, RdfLockCell e RdfLockItem, juntas implementam toda lógica de aquisição, propagação e liberação de bloqueios. O relacionamento hierárquico entre os *Lock* Items de uma mesma transação é implementado por meio dos papéis children e parents do auto-relacionamento da classe RdfLockItem.

A classe RdfLockCell apresenta duas operações abstratas (propagate_read_*lock* e propagate_write_*lock*) relativa a propagação de bloqueios. Esta duas operações são apropriadamente invocadas pelo método propagate_*lock*, que é um Template Method [Gamma et al., 1994], durante a propagação de bloqueios para os grânulos maiores. Esta duas operações são abstratas porque a forma de propagação de bloqueios varia de acordo com o tipo de grânulo em questão. Para cada tipo de grânulo foi criada uma subclasse para implementar estas operações de acordo. Poderia ter implementado todos

PUC-Rio - Certificação Digital Nº 0721352/CA

os tipos de propagação dentro a classe `RdfLockCell`, mas resultaria em um método monolítico, cheio de condicionais, o que é uma boa oportunidade de aplicar polimorfismo. Caso, futuramente, descubra-se um novo tipo de grânulo, basta criar a subclasse correspondente e implementar as duas operações.

Associado à hierarquia das diferentes implementações da classe `RdfLockCell`, foi criada a classe `RdfLockFactory`, que é um factory parametrizado [Gamma et al., 1994], responsável por instanciar a subclasse de `RDFLockCell` correta, de acordo com tipo de grânulo passado como parâmetro. Este factory é usado pelo *Lock Manager* para obter a *Lock Cell* associada ao item de dado (grânulo específico) a ser bloqueado.

Para exemplificar o uso do *Lock Manager* no modo de operação default (ou seja, MGL - Multigranular), suponha que o *Lock Manager* esteja vazio (sem nenhum bloqueio), quando é executado o trecho de código Ruby apresentado na Tabela 20. Neste exemplo, uma transação T_1 (id = 1), requisita um bloqueio de leitura para remoção (*rR*) na propriedade *foaf:name* do recurso *ex:mark* e, em seguida, requisita um bloqueio de leitura para remoção e inserção (*riR*) na propriedade *foaf:age* do mesmo recurso *ex:mark*, obtendo ambos os bloqueios com sucesso. A seguir, uma segunda transação T_2 (id = 2), requisita um bloqueio de escrita para inserção (*iW*) na propriedade *foaf:name* do recurso *ex:mark*, obtendo o bloqueio com sucesso também. Afinal os tipos de bloqueio *rR* e *iW* são compatíveis.

Tabela 20 - Exemplo (código Ruby) de uso do *Lock* Manager

Exemplo (código Ruby) de uso do *Lock Manager*

```
. . .

     Synth::Transaction::RdfLockManager.instance.lock 1,
     Synth::Transaction::RdfLockManager.property_of_resource,
     Synth::Transaction::RdfLockManager.rR,
     :property => 'foaf:name', :resource => 'ex:mark'

     Synth::Transaction::RdfLockManager.instance.lock 1,
     Synth::Transaction::RdfLockManager.property_of_resource,
     Synth::Transaction::RdfLockManager.riR,
     :property => 'foaf:age', :resource => 'ex:mark'

     Synth::Transaction::RdfLockManager.instance.lock 2,
     Synth::Transaction::RdfLockManager.property_of_resource,
     Synth::Transaction::RdfLockManager.iW,
     :property => 'foaf:name', :resource => 'ex:mark'

. . .
```

PUC-Rio - Certificação Digital Nº 0721352/CA

Como resultado da execução deste trecho de código, o modelo de classes do *Lock Manager* é instanciado gerando o *snapshot* de objetos apresentado na Figura 30 (branco - *Lock Manager*, rosa - *tabelas hash*, azul - *Lock Cell*, amarelo - *Lock Item*, verde - *Transaction List*).

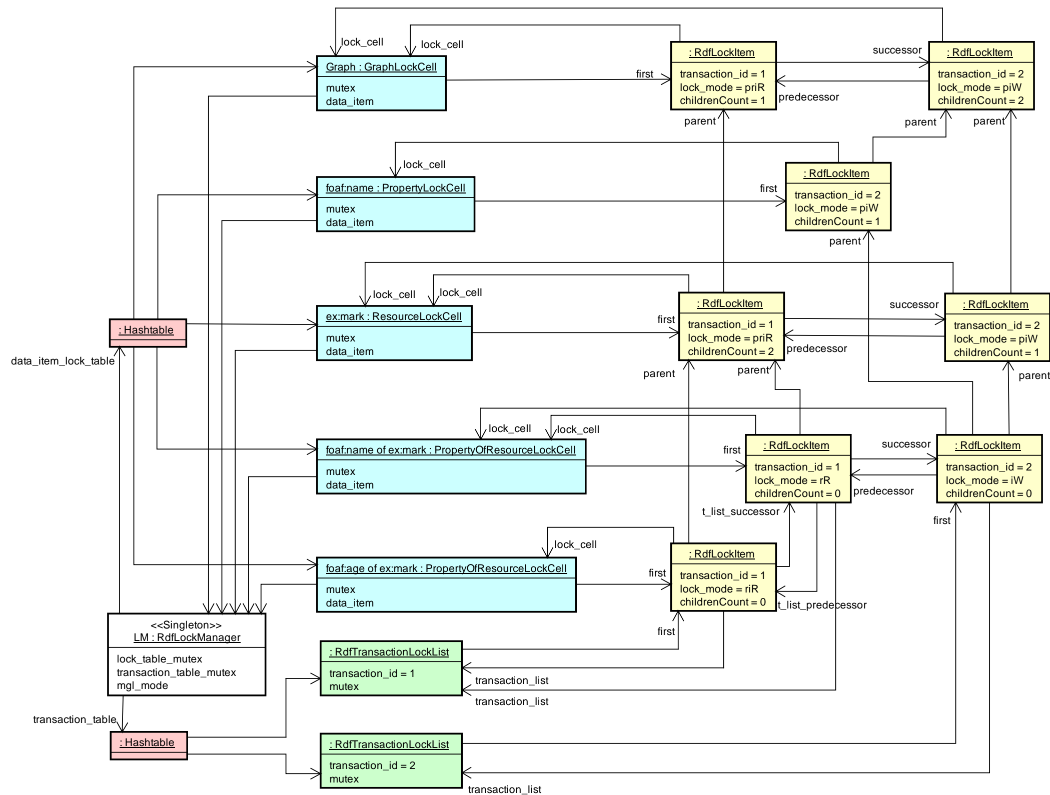

Figura 30 - Exemplo de instanciação do *Lock Manager*

Perceba que cada *Lock Cell* aponta para o primeiro elemento de sua lista de *Lock Items*, que, por sua vez, apontam para o sucessor e predecessor, formando a lista encadeada. Cada *Lock Item* da lista aponta para a *Lock Cell* em questão, e cada *Lock Cell* aponta para o *Lock Manager*. Na liberação de um bloqueio, o *Lock Item* se remove da lista encadeada e, caso seja o último da lista, ele se remove da *Lock Cell*.

Note também que cada *Transaction List* aponta para o primeiro elemento da lista de *Lock Items* (bloqueios reais) da transação, que, por sua vez, apontam para sucessor (`t_list_sucessor`) e predecessor (`t_list_predecessor`). Além disso, cada *Lock Item* (bloqueio real) aponta para a *Transaction List* de sua transação. Cada *Lock Item* também aponta para seus pais de acordo com protocolo multigranular proposto. Deste forma, na liberação de um bloqueio, o *Lock Item* se remove da *Transaction List* da sua transação e, se for o último da lista, a *Transaction List* se remove da *Transaction Table* do *Lock Manager*. Em

PUC-Rio - Certificação Digital Nº 0721352/CA

seguida, se não tiver filhos, o *Lock Item* se remove da lista encadeada da sua *Lock* Cell (se tiver filhos apenas faz *downgrade* de bloqueio real para o correspondente bloqueio planejado). Em seguida, o *Lock Item* propaga a remoção para seus pais. Se o pai não tiver mais outros filhos, ele também se remove da lista encadeada da sua *Lock Cell*.

4.2. DSL e Vocabulário de Apoio

Foi criada também uma DSL[45] interna em Ruby para elaboração do modelo de transição de estados de *workflow* para transações Web. Para auxiliar o uso do *Lock Manager*, também foi criado um pequeno vocabulário[46] em OWL, que juntamente com SPARQL CONSTRUCT, auxilia na definição dos itens a serem bloqueados pela transação. Esta duas ferramentas serão apresentas a seguir.

4.2.1. DSL para *Workflow* de Transações Web

Para prover um caminho bem definido a ser seguido pelo projetista, bem como definir uma arquitetura de execução para transações Web, foi criada uma DSL interna em Ruby, baseada em modelo de transições de estado, para especificação de *workflow* de transações Web (vale para qualquer tipo de transação Web, ou seja, é independente do model de dados adjacente, pondendo ser RDF ou qualquer outro).

Antes de apresentar a DSL propriamente dita, vale uma pausa para explicar o significado do termos **Transação Web** e **DSL interna**.

Transação Web

Rememorando, uma transação Web é uma transação ACID em nível de aplicação Web, ou seja, nada mais do que um "caso de uso" transacional em uma aplicação Web, onde a interação do usuário é pré-definida por um *workflow* até atingir um objetivo. Ao longo de uma transação Web ocorre uma sequencia de leituras de dados e uma única atualização ao final (espécie de *commit*) que garante que, em caso de sucesso, todos os efeitos da transação Web sejam persistidos de uma só vez (atomicidade).

---

[45] Veja Apêndice, seção A.4, para código fonte completo da DSL.

[46] Veja Apêndice, seção A.3, para o vocabulário de bloqueios em RDF/XML.

PUC-Rio - Certificação Digital Nº 0721352/CA

De acordo com [Distante e Tilley, 2005], uma transação Web é um passo do *workflow* de um processo ou transação de negócio maior. Como exemplo de um processo de negócio envolvendo transações Web, podemos citar o planejamento de uma viagem como descrito em [Schmit e Dustdar, 2005]. O *workflow* deste processo de negócio é composto de três transações Web: a reserva de um voo, o aluguel de um carro no destino e a reserva em um hotel no destino.

Vale destacar que transações Web duram minutos, e podem perfeitamente ser implementadas como transações ACID (e, com base em minha experiência, tipicamente o são). Por outro lado, transações de negócio são, na maioria dos casos, non-ACID, podendo durar dias ou meses. No exemplo anterior de [Schmit e Dustdar, 2005], suponha que o serviço de aluguel de carros precise de dois dias para enviar um e-mail de confirmação e a companhia aérea precise de cinco minutos para reservar um assento no voo. Seria inadmissível estabelecer um bloqueio no assento por dois dias. Além disso, os três passos deste processo de negócio não precisam estritamente respeitar a propriedade de atomicidade. Por exemplo, é perfeitamente aceitável ter a seguinte regra de negócio: a falha da reserva do voo implica em abortar toda a transação de negócio, mas quanto aos outros serviços, basta que pelo menos um deles tenha sucesso para que transação de negócio tenha sucesso, o outro serviço é apenas desejável. Obviamente, uma vez que atomicidade não é respeitada, mecanismos extras, como transações Web compensatórias, precisam ser utilizados no *workflow* de transações de negócio.

Outro ponto relevante, porém pouco discutido, é como combinar transação e navegação. Estritamente falando, navegação é um comportamento como qualquer outro. A única diferença é que sua semântica é conhecida a priori, tal como previsto pelos modelos de navegação hiper-textual de nó e *link* padrão, exemplificados pela WWW. Por isso, "modelos de navegação" especializados foram propostos para simplificar a especificação desta parte do comportamento da aplicação. Tipicamente, navegação é implementada como transação aninhada para selecionar itens a serem processados por uma transação irmã subsequente. Por exemplo, a transação Web "Criação de Pedido" tem duas transações aninhadas: "Seleção de Itens" e "*Checkout*". Sob este ponto de vista, navegação é um tipo especial de transação Web. Em [Jacyntho, 2007] é apresentada uma discussão mais ampla sobre o inter-relacionamento entre os paradigma hipermídia e transacional, bem como os requisitos de um meta-modelo para transações Web e o estado da arte dos modelos conceituais

PUC-Rio - Certificação Digital Nº 0721352/CA

propostos até então. Já [Jacyntho e Schwabe, 2010a] e [Jacyntho e Schwabe, 2010b] ambos descrevem uma DSL (definida por um meta-modelo e uma notação gráfica) para permitir a modelagem explícita de transações Web e de negócio, por meio da reificação do conceito de transação como um tipo (classe) de primeira ordem no meta-modelo, onde instâncias do tipo de transação correspondem às ocorrências (execuções) da transação e armazenam todo estado da ocorrência.

DSL interna

Diferentemente de linguagens de propósito geral, DSL (*Domain Specific Language*) é uma linguagem voltada para resolver problemas em um domínio específico (no caso, *workflow* de transações Web). Uma DSL é definida por dois componentes: um metamodelo e uma notação (que pode ser gráfica ou textual). A forma tradicional de se construir uma DSL é desenvolver seu próprio *parser* (interpretador ou compilador) para processar os modelos construídos com a DSL. Enfim, codificar uma nova linguagem. Martin Fowler chama esta abordagem tradicional de *DSL externa* [Fowler e Parsons, 2011]. A desvantagem deste tipo de DSL é clara: necessidade de criar uma nova linguagem de programação "do zero".

Uma alternativa mais simples é construir uma DSL utilizando uma linguagem de programação pré-existente como hospedeira, e, portanto, reaproveitar todas as construções desta linguagem (*statements*, classes, métodos, comentários, etc.). As palavras chave da DSL são, na verdade, chamadas de operações de classes criadas nesta linguagem. Ou seja, para cada componente do metamodelo da DSL cria-se uma classe na linguagem hospedeira. Depois cria-se uma ou mais classes que provejam operações para instanciar o modelo (instanciar as classes do metamodelo). Estes métodos serão utilizados como palavras chave da DSL. O termo utilizado por Fowler para esta outra abordagem é *DSL interna* [Fowler e Parsons, 2011]. Ruby, com seu poderoso suporte a metaprogramação e sua sintaxe flexível, se apresenta como uma excelente opção para desenvolvimento deste tipo de DSL.

DSL interna, em Ruby, para Transações Web

Retornando para a DSL em questão, na Figura 31, encontra-se o metamodelo da mesma. Mais especificamente o metamodelo da DSL compreende os modelos `structural` e `behavioral`, ambos em azul, e o modelo `predicate`, em amarelo. Os outros modelos (`user`, `action`

PUC-Rio - Certificação Digital Nº 0721352/CA

`specification` e `operation`) foram apresentados para melhor contextualizar o metamodelo da DSL.

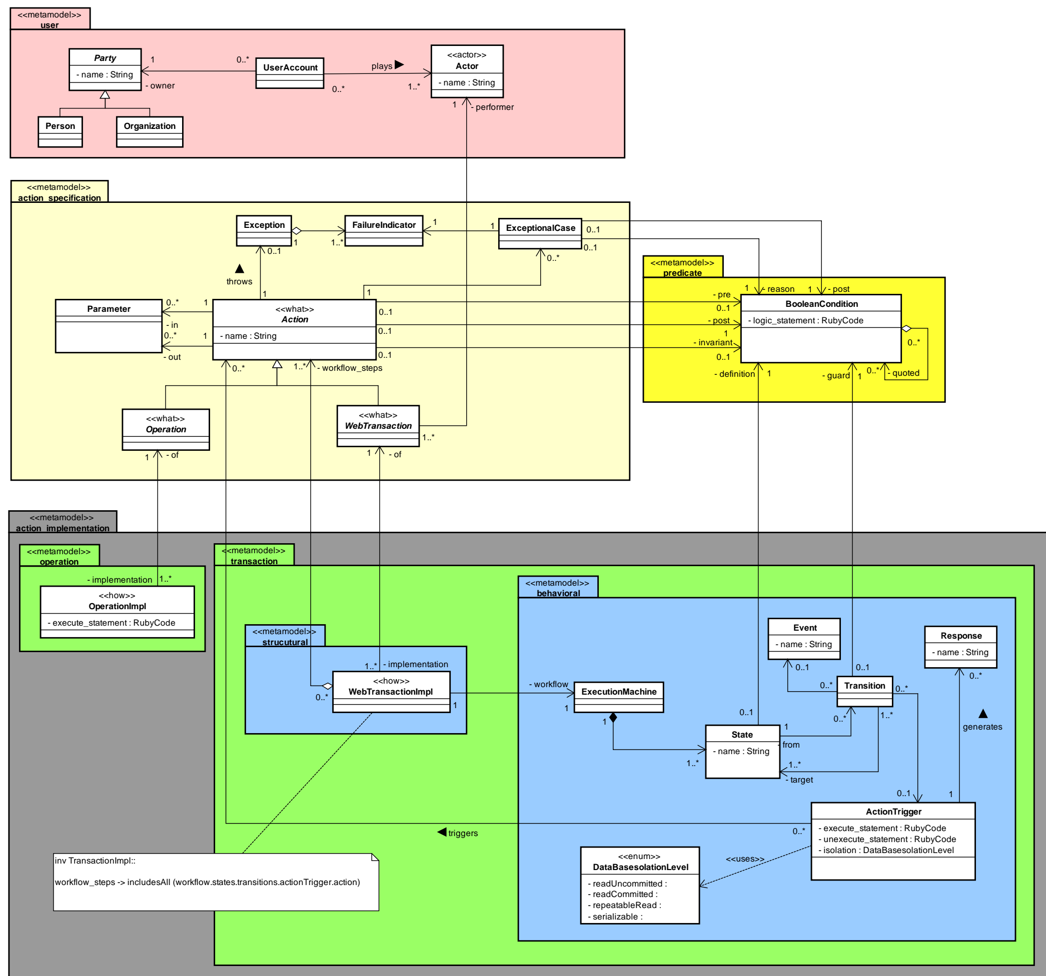

Figura 31 - Metamodelo da DSL para *workflow* de transações Web

O modelo `user` apresenta o conceito `Ator` como entidade que utiliza a aplicação, disparando uma transação Web, para atingir um objetivo. O modelo `action specification`, está presente para desacoplar a especificação ("*what*") de uma ação das várias possíveis formas de realizá-la ou implementá-la ("*how*"). A especificação define o contrato (*design by contract*) da ação, incluindo: pré-condição, pós-condição, invariantes, parâmetros de entrada e saída, e casos excepcionais. A implementação define uma, dentre as várias formas possíveis, de, dado que a pré-condição seja respeitada, chegar a pós-condição da ação. Caso a ação seja uma operação, que é a ação mais elementar prevista, a implementação consiste em um código Ruby. Já quando a ação é uma transação Web, a implementação consiste em dois componentes: `WebTransactionImpl` e `ExecutionMachine`. O primeiro corresponde ao

PUC-Rio - Certificação Digital Nº 0721352/CA

objeto de primeira ordem que representa a transação reificada, ou seja, a transação é um objeto, instanciado quando ela inicia. Portanto, para cada transação Web deve ser definida uma classe em Ruby. O segundo representa o modelo de estados da transação, ou seja, o *workflow* de sua execução, que é o objetivo da DSL interna aqui proposta.

O modelo de estados (`ExecutionMachine`), obviamente, é composto de estados e transições. Onde cada estado é definido por uma expressão booleana (`BooleanCondition`), que é um código Ruby envolvendo propriedades dos objetos de domínio e do próprio objeto da transação Web reificada que indica se a transação *está* (*verdadeiro*) *ou não está* (*falso*) no referido estado. Cada transição tem um estado de origem e um ou mais estados de destino e, em geral, tem um evento que a dispara. Quando a transição não tem evento, ela ocorre automaticamente, de acordo com o estado dos objetos manipulados. Evento, em geral, corresponde a uma requisição HTTP, e carrega consigo os parâmetros desta requisição. Uma transição tem também uma condição de guarda (`BooleanCondition`), e somente ocorre caso esta condição seja verdadeira. Caso uma transição não tenha condição de guarda será usada a condição de guarda pré-definida `true` (que, obviamente, sempre será satisfeita). Por fim, uma transição pode ter um disparador de ação (`ActionTrigger`) associado, que é um objeto (*design pattern Command* [Gamma et al., 1994]) responsável por executar efetivamente a transação Web, adquirindo os bloqueios necessários, carregando objetos do banco de dados e invocando ações apropriadamente. Cada `ActionTrigger` é definido por dois códigos Ruby: `execute_`*`statement`* e `unexecute_`*`statement`*. O primeiro é o código a ser executado quando a transição ocorrer. Já o segundo é o código que deve desfazer os efeitos do primeiro, sendo utilizado para *rollback*, caso a transação Web seja abortada (os `ActionTriggers` são armazenados em uma pilha no objeto da transação reificada, para que seja possível fazer o *rollback*). Perceba que caso ação disparada pelo `ActionTrigger` seja uma outra transação Web, teremos uma transação Web aninhada (*nested Web transaction*). De fato, *rollback* somente é necessário para transações Web aninhadas. No caso da transação Web raiz da árvore, não há necessidade de rollback, pois, em caso de aborto, todos os objetos modificados (em memória) simplesmente serão descartados, não sendo transferidos para o banco de dados adjacente.

Com relação a interoperabilidade entre a transação Web e suas transações de banco de dados, o acesso ao banco de dados ocorre durante a

PUC-Rio - Certificação Digital Nº 0721352/CA

execução de um `ActionTrigger`, que é quem trata o evento externo (requisição HTTP). Sendo assim, em geral, uma transação de banco de dados é iniciada como primeiro passo do código `execute_statement` do `ActionTrigger`, sendo finalizada (com *rollback* ou *commit*) como último passo do mesmo código. Por esta razão, o `ActionTrigger` possui a propriedade `isolation`, onde pode ser definido o nível de isolamento a ser utilizado no momento de obter uma conexão com banco de dados, para iniciar uma transação com o mesmo.

Como obter uma conexão com o banco de dados, iniciar uma transação, confirmar (*commit*) ou abortar (*rollback*) são os mesmos passos em todos os `ActionTriggers`, uma solução mais elegante seria aplicar o *design pattern Decorator* [Gamma et al., 1994], e decorar todo os `ActionTriggers` com estes pré/pós-processamentos relativos a transações de banco de dados.

Juntamente com a DSL, é oferecido um pequeno framework com alguns componentes caixa-branca que reúnem características comuns, e devem ser reusados por meio de herança. Dentre estes componentes estão previstos *decorators* para os `ActionTriggers` (Figura 32), em especial o `TransactionalActionTriggerDecorator`, que perfaz todo o pré/pós processamento relativo a transações de banco de dados, antes e depois de delegar para a execução do `ActionTrigger` decorado. Basta a aplicação especializar este decorator, implementando as operações abstratas do Template Method [Gamma et al., 1994] `execute()`, de acordo com o banco de dados em questão, e todos os `ActionTriggers` passarão, automaticamente, a executar no escopo de uma transação de banco de dados, e com o nível de isolamento especificado.

PUC-Rio - Certificação Digital Nº 0721352/CA

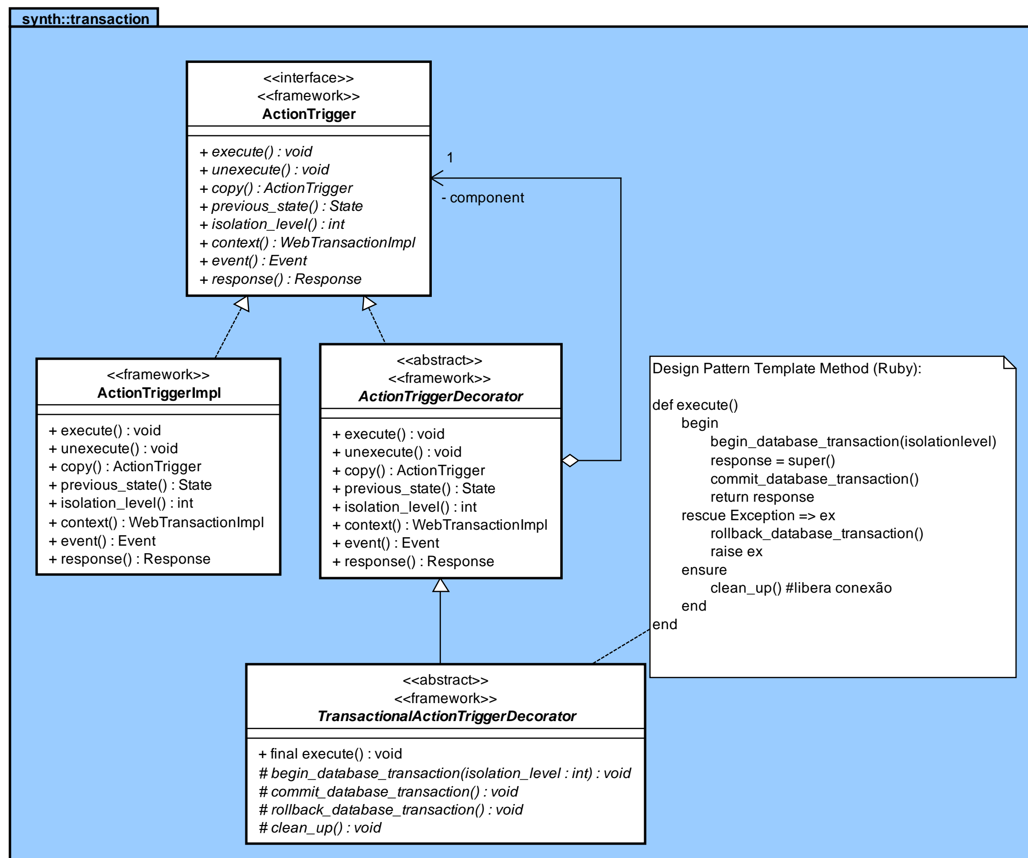

Figura 32 - TransactionalActionTriggerDecorator

Para ilustrar o uso da DSL será utilizado um exemplo bem simplório, o que torna mais fácil a compreensão da mesma.

Trata-se de uma transação Web, chamada `SimulacaoLampadaWebTransaction`, que simula o "*acender*" e o "*apagar*" de uma lâmpada (único objeto de domínio no exemplo) até que a lâmpada, ao acender, queime, quando então a transação termina (nos testes a lâmpada foi configurada para queimar quando acendesse pela segunda vez). A transação Web reificada, o domain model e os componentes do framework associado a DSL são mostrados na Figura 33. Perceba que a classe da transação Web especializa a classe `Implementation` do framework que reúne o que há de comum a todas as transações Web: operações de `commit` e `rollback`, tratamento de eventos, pilha de `ActionTriggers` para eventual *rollback*, e estado corrente no modelo de estados.

A classe `SimulacaoLampadaWebTransaction` da transação Web reificada possui um campo estático (variável de classe) onde fica armazenada seu modelo de estados de execução (`ExecutionMachine`), ou seja, o modelo que foi criado com a DSL (instanciação do metamodelo da Figura 31). Com

PUC-Rio - Certificação Digital Nº 0721352/CA

relação ao modelo de estados, a instância da transação Web corrente é chamada de `context`, ou seja, o contexto sobre o qual a máquina de estados esta sendo executada. Em outras palavras, a máquina de estados é totalmente *stateless* com relação a transação corrente. Todo o estado da transação é armazenado no próprio objeto da transação, o que é a principal razão para reificação de transações.

Para permitir instanciar o metamodelo foi criada uma classe Ruby para cada elemento do metamodelo e, para cada metaclasse criada, foi criada uma outra classe Ruby com sufixo `_dsl`, responsável por instanciar a metaclasse, oferecendo, para tal, operações que funcionem como se fossem verdadeiras palavras chave da DSL (notação textual).

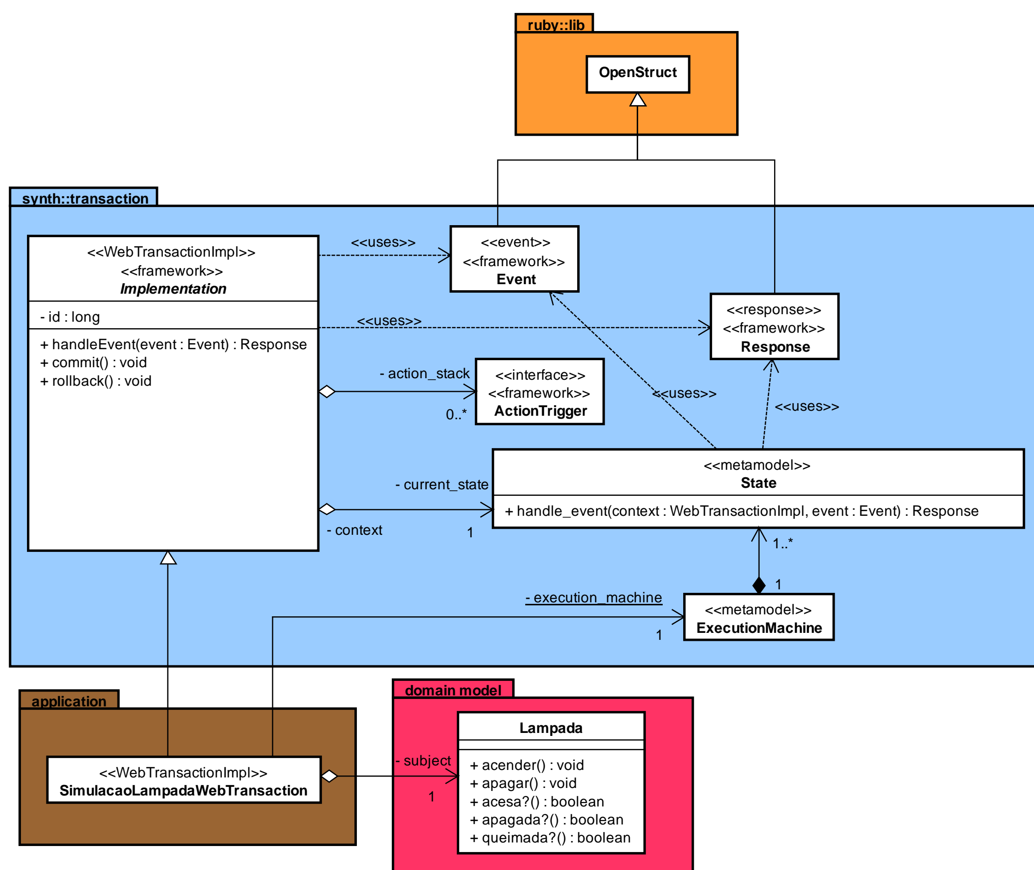

Figura 33 - Transação Web "Simulação Lâmpada" reificada

O modelo de estados do *workflow* da transação Web `SimulacaoLampadaWebTransaction` é ilustrado em um diagrama *statechart* na Figura 34. Para cada estado é mostrada sua definição por meio de uma expressão booleana (instância da metaclasse `BooleanCondition`). Para cada transição temos: evento (instância da metaclasse `Event`), condição de guarda

PUC-Rio - Certificação Digital Nº 0721352/CA

(instância da metaclasse `BooleanCondition`), ação (conteúdo do meta-atributo `execute_statement` da metaclasse `ActionTrigger`). *Apagada* é o estado inicial e *Queimada* é o estado final da transação Web.

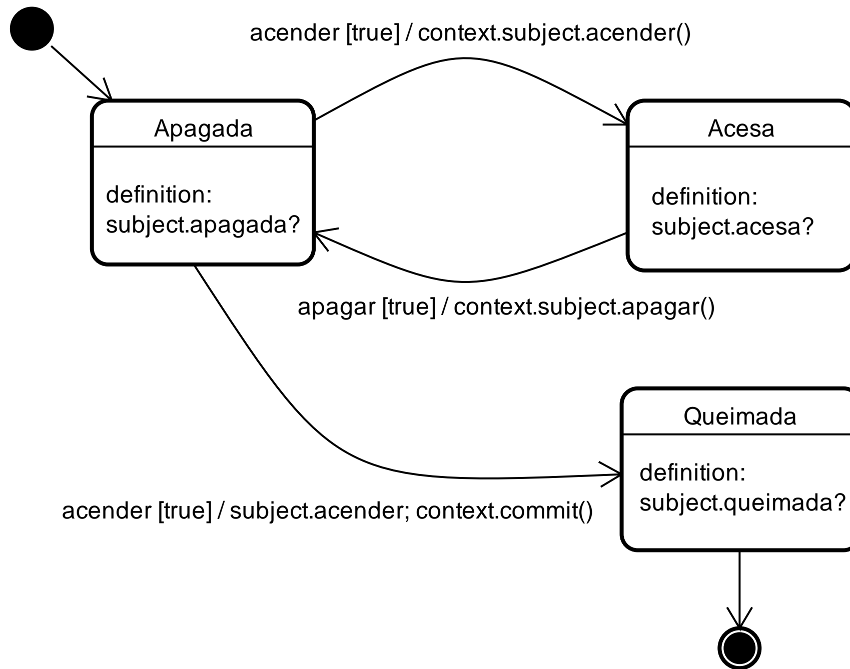

Figura 34 - Statechart da transação Web "Simulação Lâmpada"

O mesmo modelo descrito usando a DSL interna é exibido na Tabela 21.

Tabela 21 - Uso da DSL interna para a transação Web "Simulação Lâmpada"

Estados	Transições

Estados:

```
require 'execution_machine'
require 'implementation'

class SimulacaoLampadaTransaction
  include Synth::Transaction::Implementation

  attr_reader :subject

  def initialize(id, subject)
    super id
    @subject = subject
  end

Synth::Transaction::ExecutionMachine.new(self)
{

    state {
      initial
      name   :apagada
      definition {subject.apagada?}
    }

    state {
      name   :acesa
      definition {subject.acesa?}
    }


    state {
      name   :queimada
      definition {subject.queimada?}
    }
```

Transições:

```
    acender = transition {
      action {
        isolation_level   :read_comitted
        execute_statement {
          @memento = context.subject.create_memento
          context.subject.acender
          context.commit if context.subject.queimada?
        }
        unexecute_statement {
          context.subject.set_memento @memento
        }
      }
      event_name :acender_event
      target_states :acesa, :queimada
    }

    on_apagada acender

    apagar = transition {
      action {
        isolation_level :read_comitted
        execute_statement {
          @memento = context.subject.create_memento
          context.subject.apagar
        }
        unexecute_statement {
          context.subject.set_memento @memento
        }
      }
      event_name :apagar_event
      target_state :apagada
    }

    on_acesa apagar
  }
end
```

PUC-Rio - Certificação Digital Nº 0721352/CA

A classe da transação Web tem que incluir a *module* `Implementation` e, no escopo da definição da classes, instanciar uma `ExecutionMachine` definindo seu estado e transições, usando a DSL interna. Para cada elemento do metamodelo da DSL (Figura 31), existe uma correspondente palavra chave na DSL interna. Cada palavra chave na verdade é uma chamada de operação da classe `ExecutionMachine` que instancia um elemento e o insere na instância de `ExecutionMachine`, montando, passo a passo, a máquina de estados da classe da transação Web em questão. Por questão de conveniência, o componente `ActionTrigger` foi renomeado na DSL para a palavra chave `action`. Os outros componentes do metamodelo tiveram seu nomes mantidos.

Na instanciação de um estado, a palavra chave `name` define o nome do estado, e é com este nome que se faz referência ao estado posteriormente. Já `definition` é a instanciação do elemento `BooleanCondition` do metamodelo. A expressão booleana Ruby será avaliada no contexto da instância da transação Web (`context`), tendo então acesso a todas as propriedades da transação reificada, como *subject* (objeto de domínio de trabalho), por exemplo.

Na instanciação de uma transição, a definição dos `execute_`*`statement`* e `unexecute_`*`statement`* inclui o código Ruby que será executado na ocorrência da transição. Pode ser qualquer código Ruby, inclusive podem ser usadas variáveis de instância. Uma variável definida em um *statement* é visível no outro, pois este código é executado no escopo de uma instância deste `ActionTrigger`. Uma nova instância deste `ActionTrigger` é criada a cada `ocorrência` da transição (*design pattern command*), e esta instância é armazenada na pilha de `ActionTriggers` executados para eventual *rollback*. Para ter acesso ao objeto da transação, basta acessar `context`. Nos *statements* do exemplo, o `execute_`*`statement`* primeiro obtém um *memento*[47] do *subject* e armazena na variável de instância `@memento`. Esta mesma variável é usada pelo `unexecute_`*`statement`* para restaurar o estado do *subject*, em caso de um *rollback*.

A transição `acender` é especial. É especial porque ela pode levar ao estado final `queimada`. Portanto, caso ela leve ao estado final, é necessário invocar a operação `commit` da transação Web, para que todos os objetos

---

[47] *Design Pattern Memento* [Gamma et al., 1994] - sem quebrar encapsulamento, é retornado um *snapshot* do estado interno de um objeto, de forma que o objeto possa ser restaurado para este estado mais tarde.

PUC-Rio - Certificação Digital Nº 0721352/CA

modificados sejam transferidos da memória para o banco de dados, de uma só vez (Atomicidade). Isto é realizado pelo trecho de código "`context.commit if context.subject.queimada?`" do `execute_statement` da transição `acender`. Para se conseguir Atomicidade em nível da transação Web, poder-se-ia utilizar o padrão arquitetural *UnitOfWork* [Fowler et al., 2002] que é um objeto responsável por manter a lista de objetos afetados (criados, removidos e alterados) pela transação Web e, ao final, fazer todas as atualizações no banco de dados de uma só vez. Desta forma, a implementação da operação `commit` da transação Web nada mais seria do que delegar para o UnitOfWork persistir todo o trabalho.

Especificamente para transações Web RDF, o *UnitOfWork* manteria a lista de *statements* RDF a serem inseridos e removidos. Este é um dos trabalhos futuros previstos nesta tese, ou seja, Atomicidade em nível de transações Web RDF.

Para finalizar, note que como trata-se de uma DSL interna, podemos fazer uso de todas as construções da linguagem Ruby. Por exemplo, armazenar uma transição em um variável local e reusar esta mesma transição, sem precisar defini-la novamente, adicionando-a vários estados. Graças às facilidades de metaprogramação de Ruby, a adição de uma transição a um estado ficou bastante simples e elegante. Por exemplo, `on_apagada acender`, adiciona a transição `acender` (variável local `acender`) no estado `apagada`.

PUC-Rio - Certificação Digital Nº 0721352/CA

4.2.2.
Vocabulário para Aquisição de Bloqueios

Para facilitar a definição dos itens a serem bloqueados, foi criado um pequeno vocabulário em OWL que, juntamente com SPARQL CONSTRUCT, permite a aplicação, com base em uma consulta SPARQL, determinar exatamente quais itens de dados devem ser bloqueados. Relembrando, SPARQL CONSTRUCT permite criar um novo grafo a partir das triplas retornadas pela consulta. Especificamente será criado um novo grafo usando o vocabulário de bloqueios em questão. Uma vez de posse do grafo contendo os itens a serem bloqueados e qual o tipo de bloqueio para cada um, basta fazer um *parsing* no grafo, aplicando o *design pattern Builder* [Gamma et al., 1994], e gerar as chamadas da operação *`lock`* do *Lock Manager*, dinamicamente, via metaprogramação em Ruby [Perrota, 2010].

O vocabulário em si é muito simples. Para cada um dos 25 tipos de bloqueio previstos foi criada uma propriedade OWL (*object property*) correspondente (Figura 35). O grafo resultante de bloqueios conterá triplas com o seguinte *template*:

<recurso> tipo de bloqueio <propriedade>

Por exemplo, a tripla `ex:mark` *`locking`*`:rRLockAt foaf:name` indica que deve ser estabelecido um bloqueio de leitura para remoção (*rR*) na propriedade `foaf:name` do recurso `ex:mark`. Para bloquear os grânulos maiores, ou seja, *recurso*, *propriedade* e *grafo*, foi introduzido um pseudo recurso denominado *all* (todos), representando todos os elementos, onde quer que ele apareça no *template* de tripla anterior. Ou seja, se aparecer no *placeholder* `<recurso>`, representará todos os recursos. Se aparecer no *placeholder* `<propriedade>`, representará todas as propriedades. Por exemplo, para bloquear o grânulo recurso `ex:mark`, seria utilizada a tripla `ex:mark` *`locking`*`:rRLockAt` *`locking`*`:all`, ou seja, todas as propriedades do recurso `ex:mark`. Já para bloquear o grânulo propriedade `foaf:name`, seria *`locking`*`:all` *`locking`*`:rRLockAt foaf:name`, ou seja, a propriedade `foaf:name` de todos os recursos. Por fim, no caso do grafo todo, seria *`locking`*`:all` *`locking`*`:rRLockAt` *`locking`*`:all`, significando todas as propriedades de todos os recursos.

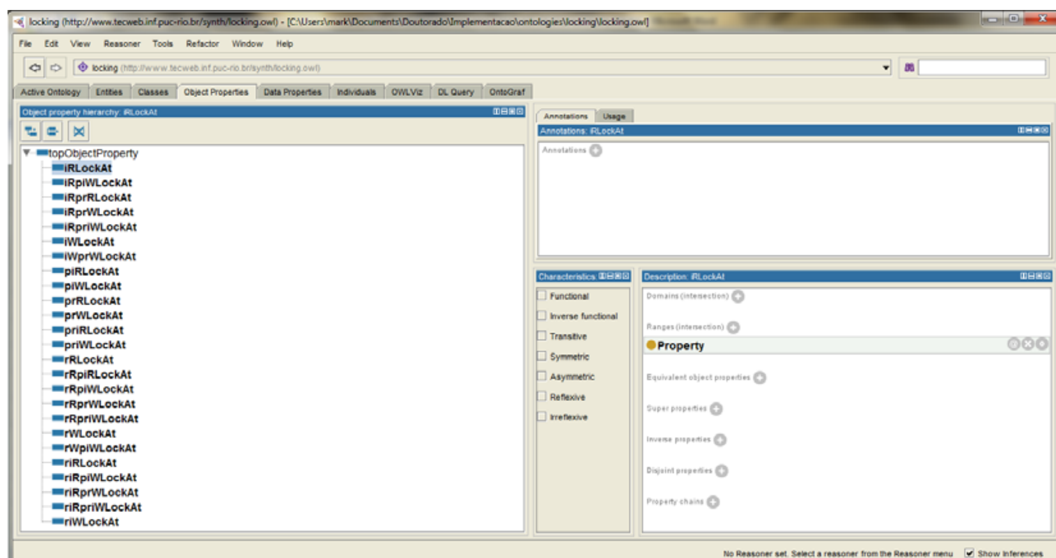

Figura 35 - *OWL Object properties* correspondentes aos tipos de bloqueio

PUC-Rio - Certificação Digital Nº 0721352/CA

4.2.2.1. Exemplos Motivacionais - Transações Web

Nesta seção, são descritas algumas transações Web de exemplo, no contexto de um CMS (*Conference Managment System*), destacando-se os problemas de concorrências e possíveis soluções utilizando os novos tipos de bloqueio, o vocabulário OWL para aquisição de bloqueios e, quando necessário, SPARQL CONSTRUCT. A ontologia OWL criada para este domínio é apresentada na Figura 36.

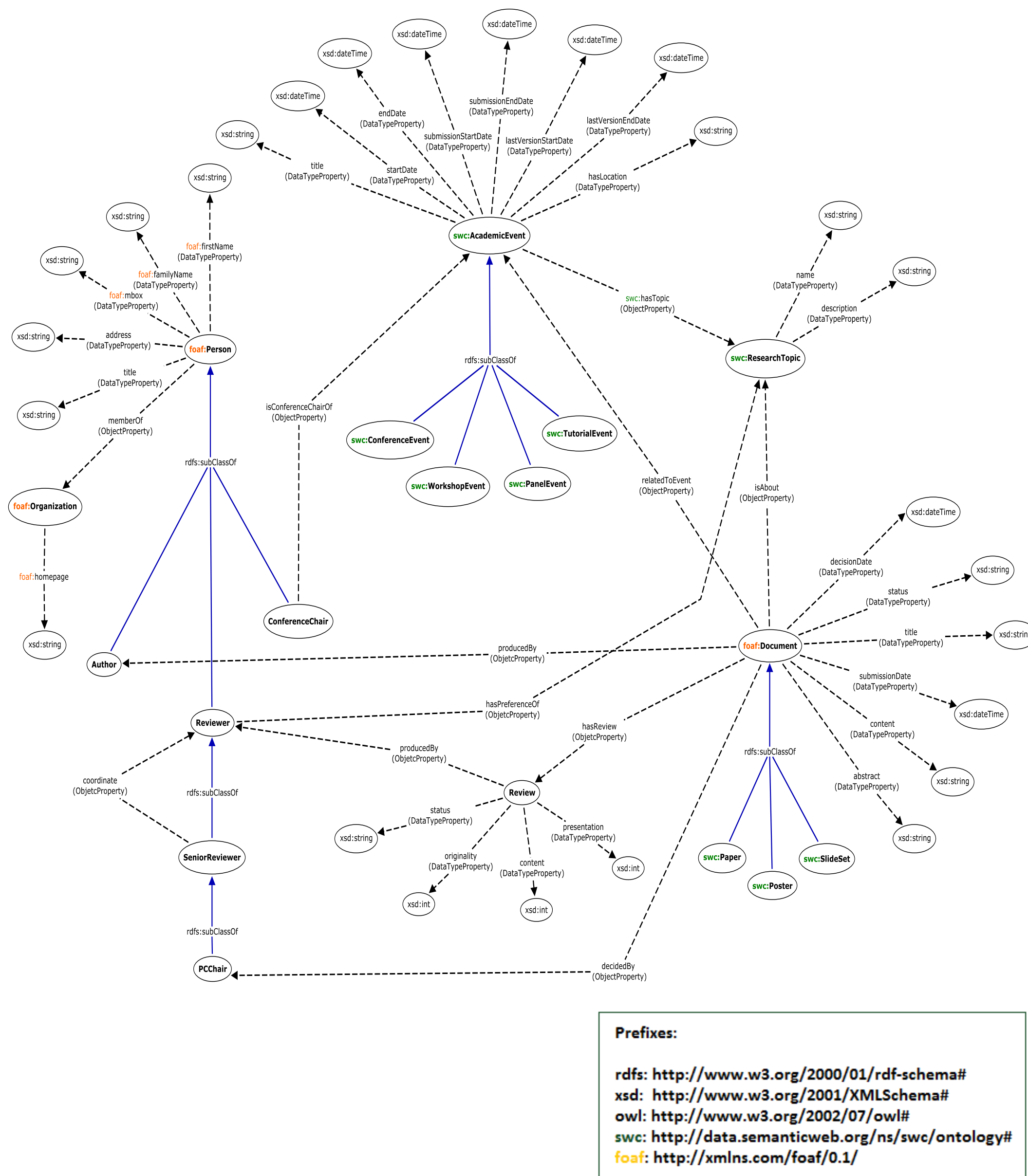

Figura 36 - Ontologia OWL para o domínio de conferências

Conforme dito anteriormente, uma transação Web é um passo do *workflow* de um processo ou transação de negócio maior. Neste exemplo, serão descritas

PUC-Rio - Certificação Digital Nº 0721352/CA

transações Web ACID que correspondem aos passos do *workflow* de uma transação de negócio non-ACID chamada "*Document Analysis*". Esta transação de negócio tem como objetivo "analisar" um documento (que pode ser um artigo, um poster ou uma palestra) submetido para uma conferência, desde de sua submissão até sua aceitação ou rejeição. Como de costume, esta transação de negócio tem duração de alguns meses. Cada documento passa por três revisões, cada uma realizada por um revisor. Finalmente, o documento é aceito ou rejeitado por um dos *pc-chairs* da conferência. Caso seja aceito, uma versão final deve ser submetida. Se a versão final não for enviada, o documento deixa de ser considerado aceito.

Vale deixar claro, que, conforme já foi dito, não faz sentido aquisição de bloqueios em nível de transações de negócio, caso contrário um bloqueio iria perdurar meses. No cenário descrito no parágrafo anterior, são as transações Web que fazem uso de bloqueios.

A transação de negócio "*Document Analysis*" tem seu *workflow* realizado por meio das seguintes transações Web:

- ***Document Submission***
  Um dos autores submete o documento.

- ***Reviewers Assignment***
  O *pc-chair* da conferência aloca os revisores para avaliar o documento.

- ***Obtaining Document To Review***
  Cada revisor obtém o documento para avaliar.

- ***Document Review Submission***
  Cada revisor submete sua avaliação.

- ***Document Decision***
  Um dos *pc-chairs* da conferência aceita ou rejeita o documento.

- ***Document Final Version Submission***
  Se o documento for aceito, o autor submete a versão final.

PUC-Rio - Certificação Digital Nº 0721352/CA

- ***Document Withdrawal***
  O autor desiste e retira o documento.

A Figura 37 descreve o *workflow* da transação de negócio "*Document Analysis*", utilizando a DSL proposta em [Jacyntho e Schwabe, 2010a] e [Jacyntho e Schwabe, 2010b] cujo meta-modelo de *workflow* é uma versão especializada do meta-modelo do diagrama de atividades da UML[48] 2.0, que é baseado em redes de Petri [Hee e Aalst, 2004].

Em linhas gerais, a notação gráfica da DSL tem o seguinte significado: *retângulo arredondado* - transação Web com seu parâmetros de entrada e saída; *retângulo tracejado* - escopo de suspensão por um evento externo ou por uma exceção; *seta* - fluxo de dados e controle; *diamante* - *decision/merge*; *barra* - *fork/join*; *círculo sólido* - início; *círculo com X* - final de fluxo; *círculo sólido/branco* - final; *retângulo com seta para baixo* - captura de exceção; *retângulo côncavo* - captura de evento externo. E as *raias de natação* delimitam as transações Web executadas por cada ator (também conhecido por *performer*).

Além destas, mais a seguinte transação Web <CRUD>[49] será descrita: ***Register Author*** - Cadastro de Autor.

[48] *Unified Modeling Language* (UML) - http://www.uml.org/

[49] CRUD - Acrônimo para ***C****reate* ***R****ead* ***U****pdate* ***D****elete*

PUC-Rio - Certificação Digital Nº 0721352/CA

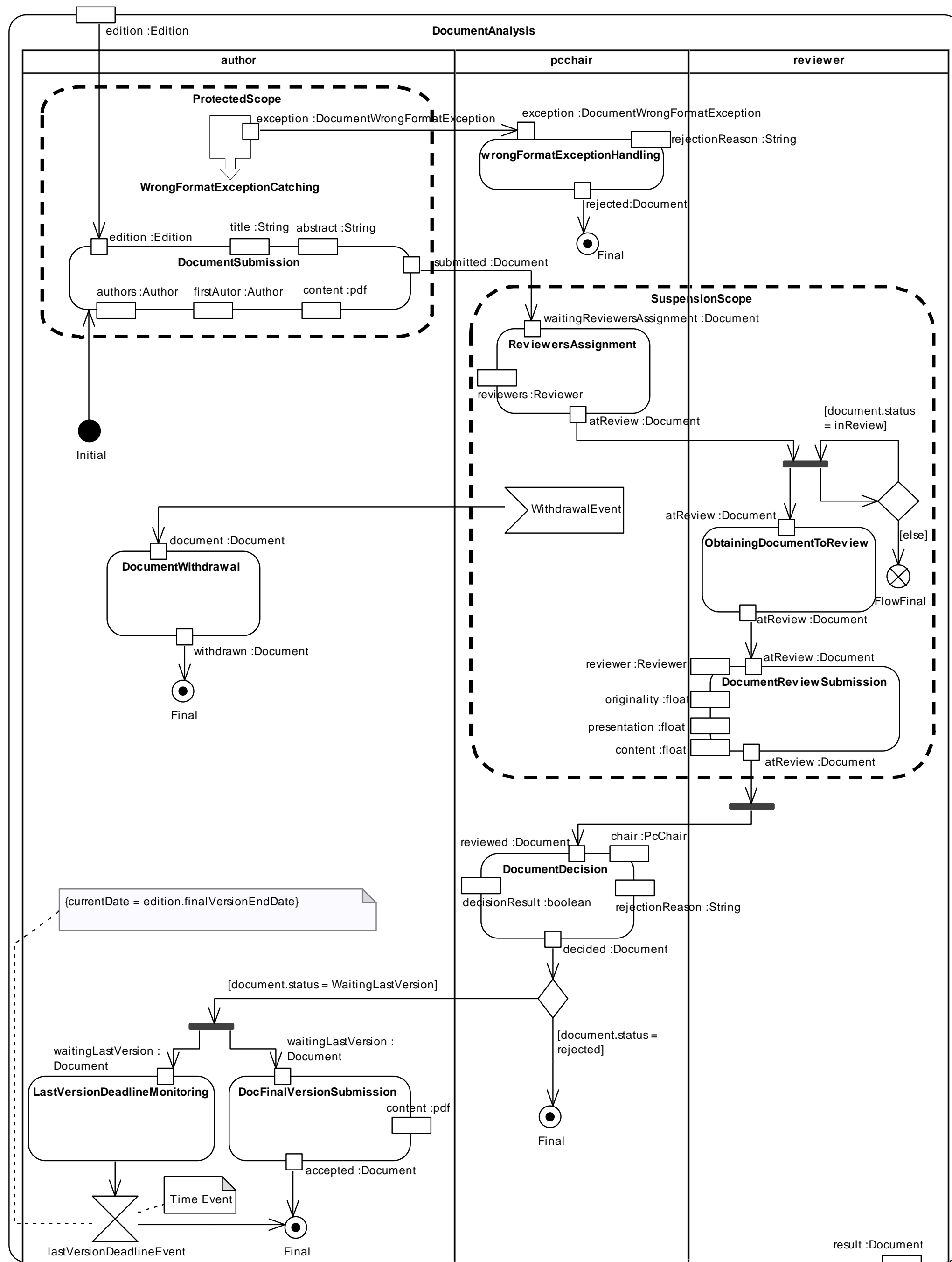

Figura 37 - Transação de negócio "*Document Analysis*" adaptado de [Jacyntho e Schwabe, 2010a]

A seguir, são especificadas todas as transações Web anteriores, apresentando: ator primário, ator secundário, pré-condição, breve descrição de execução, problemas de concorrência (*lost updates* e *inconsistent reads*) a serem evitados, e, principalmente, possíveis soluções para estes problemas usando o modelo proposto. Todas as especificações estão descritas em *Turtle*, usando interpolação de strings em Ruby. Nas especificações, `args` representa

PUC-Rio - Certificação Digital Nº 0721352/CA

uma tabela hash contendo os argumentos, no caso de especificações parametrizadas. Este estilo de especificação poderia perfeitamente fazer parte de uma ferramenta de autoria para aplicações na Web semântica, como o *Synth* proposto em [Bomfim, 2011]. *Synth* é um ambiente para construção e execução de aplicações projetadas de acordo com o método SHDM (*Semantic Hypermedia Design Method*) [Lima, 2003]. Um dos trabalhos futuros desta tese é justamente acrescentar especificação de transações Web no ambiente *Synth*.

Sendo assim, temos:

Document Submission

Ator Primário:

Autor responsável.

Ator Secundário:

Não tem.

Pré-condição:

O Autor tem que ter feito login AND data atual <= data final de submissão.

Descrição:

O sistema exibe a lista de documentos do autor. O usuário (autor responsável) seleciona um documento para editar ou escolhe submeter um novo documento (que corresponde a edição de um documento vazio). O sistema exibe o documento. O usuário atualiza o resumo, título, os tópicos aos quais o documento está relacionado, seleciona os outros autores e faz "upload" do arquivo contendo o conteúdo do documento. O sistema exibe o documento atualizado. O usuário confirma suas alterações. O sistema atualiza a data de submissão e o status (para "Submetido") e salva o documento.

Lost Updates:

1. O Documento por inteiro.

   Todas as propriedades podem ter valores inseridos e/ou removidos.

   Solução:

   No inicio da transação Web, bloqueio de escrita para remoção e inserção no documento.

PUC-Rio - Certificação Digital Nº 0721352/CA

```
%{
 @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
 #{args[:document]} locking:riWLockAt locking:all.
}
```

Inconsistent Reads:

Não tem.

Nada a fazer.

Reviewers Assignment

Ator Primário:

Pc-chair.

Ator Secundário:

Revisor.

Pré condição:

O pc-chair tem que ter feito login AND data atual > data final de submissão AND data atual <= data final de revisão.

Descrição:

O sistema exibe a lista de documentos com status "Esperando Alocação de Revisores". O usuário (*pc-chair*) seleciona um documento para alocar revisores. O sistema exibe o documento, bem com uma lista de revisores que não têm conflito com o documento. O usuário seleciona 3 (três) revisores da lista. O sistema exibe o documento e a lista com os 3 (três) revisores selecionados. O usuário confirma a alocação. O sistema cria 3 (três) avaliações com status "Não iniciada" e associa ao documento e ao respectivo revisor. O sistema atualiza o status (para "Em Revisão") e salva o documento e as avaliações, e notifica, por e-mail, cada um dos revisores.

Lost Updates:

1. Inserção de Revisões ao Documento. É criada uma revisão para cada revisor, com status "não iniciada".

   Inserção de novos valores na propriedade "hasReview" do Documento.

   Solução:

   No início da transação Web, bloqueio de escrita para inserção na propriedade "*hasReview*" do documento.

PUC-Rio - Certificação Digital Nº 0721352/CA

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
   #{args[:document]} locking:iWLockAt conf:hasReview.
}
```

Inconsistent Reads:

Por questão de conflito de interesse, temos:

1. Não pode inserir um novo Autor ao Documento.

   Não pode inserir um novo valor na propriedade "producedBy" do Documento

   Solução:

   No início da transação Web, bloqueio de leitura pra inserção na propriedade "*producedBy*" do documento.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
   #{args[:document]} locking:iRLockAt conf:producedBy.
}
```

2. Não pode associar os Autores do Documento a uma nova Organização.

   Não pode inserir um novo valor na propriedade "memberOf" de cada Autor do Documento.

   Solução:

   No início da transação Web, bloqueio de leitura para inserção na propriedade "*memberOf*" de cada autor do documento.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  CONSTRUCT
  {
    ?author locking:iRLockAt conf:memberOf
  }
  WHERE
  {
    #{args[:document]} conf:isProducedBy ?author.
  }
}
```

PUC-Rio - Certificação Digital Nº 0721352/CA

3. Não pode associar os Revisores selecionados a uma nova Organização e os Revisores selecionados não podem deixar de ser Revisores.
   Não pode inserir um novo valor na propriedade "memberOf" de cada Revisor selecionado.
   Não pode remover um valor na propriedade "rdf:type" de cada Revisor selecionado.

Solução:

No início da transação Web, bloqueio de leitura para inserção na propriedade "*memberOf*" e bloqueio de leitura para remoção na propriedade "*rdf:type*" dos revisores existentes até então, que não tenham conflito com as organizações dos autores do documento.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  @prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  CONSTRUCT
  {
    ?reviewer locking:iRLockAt conf:memberOf.
    ?reviewer locking:rRLockAt rdf:type.
  }
  WHERE
  {
    ?reviewer a conf:Reviewer;
              conf:memberOf ?organization.
    FILTER(
              #{
                   args[:organizations].collect {
                       |org| "?organization" != " + org
                   }.join(" && ")
               }
          )
  }
}
```

PUC-Rio - Certificação Digital Nº 0721352/CA

O resultado desta interpolação seria:

```
@prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
@prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
CONSTRUCT
{
  ?reviewer locking:iRLockAt conf:memberOf.
  ?reviewer locking:rRLockAt rdf:type.
}
WHERE
{
  ?reviewer a conf:Reviewer;
            conf:memberOf ?organization.
  FILTER(
            ?organization != <org_1> &&
            ?organization != <org_2> &&
            . . .
            ?organization != <org_N>
        )
}
```

Importante:

Há um protocolo a ser seguido para estabelecer este último bloqueio. É necessário bloquear, temporariamente, todo o grafo, com um bloqueio de leitura para inserção, pois não podem surgir revisores "*phantoms*" após o estabelecimento dos bloqueios e antes da transação Web carregar os revisores. Caso contrário, a transação Web pode carregar um revisor que não estará bloqueado. Portanto, o protocolo compreende, nesta ordem: adquirir bloqueio de leitura para inserção em todo o grafo, adquirir o referido bloqueio usando o resultado do *SPARQL Construct*, carregar os revisores (que não têm conflito) fazendo uma consulta *SPARQL Select*, liberar o bloqueio do grafo.

Obtaining Document to Review

Ator Primário:

Revisor.

Ator Secundário:

Não tem.

PUC-Rio - Certificação Digital Nº 0721352/CA

Pré condição:

O Revisor tem que ter feito login AND status do Documento tem que ser "Em Revisão".

Descrição:

O sistema exibe a lista de documentos que possuam revisão cujo revisor é o usuário. O usuário (revisor) seleciona um documento para fazer download do arquivo para avaliar. O sistema executa o download do arquivo. O usuário confirma que o download foi realizado.

Lost Updates:

Não tem.

Nada a fazer.

Inconsistent Reads:

Não tem.

Nada a fazer.

Review Submission

Ator Primário:

Revisor.

Ator Secundário:

Não tem.

Pré-condição:

O Revisor tem que ter feito login AND data atual <= data final de revisão.

Descrição:

O sistema exibe a lista de documentos que possuam revisão cujo revisor é o usuário. O usuário (revisor) seleciona um documento para atualizar sua revisão. O sistema exibe a revisão cujo revisor é o usuário e o título do documento associado. O usuário atualiza as notas de originalidade, conteúdo e apresentação, bem como o parecer (qualquer observação que queira fazer a respeito das suas decisões). O sistema exibe a revisão atualizada. O usuário confirma suas alterações. O sistema atualiza a data de submissão e o status e salva a revisão.

Lost Updates:

1. A Revisão por inteiro.

   Todas as suas propriedades podem ter valores inseridos e/ou removidos.

PUC-Rio - Certificação Digital Nº 0721352/CA

Solução:

No inicio da transação Web, bloqueio de escrita para remoção e inserção na revisão.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  CONSTRUCT
  {
    ?review locking:riWLockAt locking:all
  }
  WHERE
  {
    #{args[:document]} conf:hasReview ?review.
    ?review conf:producedBy ?reviewer.
    FILTER(
              ?reviewer = #{args[:reviewer]}
          )
  }
}
```

PUC-Rio - Certificação Digital Nº 0721352/CA

Inconsistent Reads:

1. Não pode remover a Revisão em questão do Documento
   Não pode remover nenhum valor da propriedade "hasReview" do Documento em questão.

Solução:

No início da transação Web, bloqueio de leitura para remoção na propriedade "hasReview" do documento.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  #{args[:document]} locking:rRLockAt conf:hasReview.
}
```

Document Decision

Ator Primário:

Pc-chair.

Ator Secundário:

Não tem.

Pré-condição:

O *pc-chair* tem que ter feito login AND data atual > data final de revisão.

Descrição:

O sistema exibe a lista de documentos que estão aguardando decisão (ou aceitação ou rejeição). O usuário (*pc-chair*) seleciona um documento para decidir. O sistema exibe o documento. O usuário fornece sua decisão, bem como o parecer (qualquer observação que queira fazer a respeito de sua decisão). O sistema exibe o documento atualizado. O usuário confirma suas alterações. O sistema atualiza a data da decisão e o status e salva a documento.

Lost Updates:

1. Propriedades *status*, *data da decisão*, *decidido por* (pc-chair) do Documento em questão

   Todas estas propriedades podem ter valores inseridos e/ou removidos.

   Solução:

   No início da transação Web, bloqueio de escrita para remoção e inserção na propriedades "*status*", "*decisionDate*", "*decidedBy*" do documento.

```
%{
 @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
 @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
 #{args[:document]} locking:riWLockAt conf:status.
 #{args[:document]} locking:riWLockAt conf:decisionDate.
 #{args[:document]} locking:riWLockAt conf:decidedBy.
}
```

Inconsistent Reads:

1. Todas as outras propriedades do Documento não podem ser alteradas

   Todas estas propriedades não podem ter nenhum valor inserido ou removido.

   Solução:

   No início da transação Web, bloqueio de leitura para remoção e inserção em todas as propriedades do documento.

PUC-Rio - Certificação Digital Nº 0721352/CA

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  #{args[:document]} locking:riRLockAt conf:all.
}
```

Document Final Version Submission

Ator Primário:

Autor responsável.

Ator Secundário:

Não tem.

Pré-condição:

O autor tem que ter feito login AND status do Documento igual a "Esperando Versão Final" AND data atual <= data final de submissão da versão final.

Descrição:

Idem a transação Web "*Document Submission*", com uma única diferença: ao atualizar, o status do Documento passará para "Aceito".

Lost Updates:

Idem a transação Web "*Document Submission*".

Inconsistent Reads:

Idem a transação Web "*Document Submission*".

Document Withdrawal

Ator Primário:

Autor responsável.

Ator Secundário:

Não tem.

Pré-condição:

O autor tem que ter feito login AND o status do Documento tem que ser diferente dos seguintes valores: "Esperando Versão Final", "Aceito", "Rejeitado", "Sem Versão Final".

Descrição:

O sistema exibe a lista de documentos do autor. O usuário (autor responsável) seleciona um documento retirar. O sistema exibe o documento. O usuário confirma sua retirada. O sistema remove o documento (ou seja, remove todos os *statements* que mencionam o documento, quer seja como sujeito, quer seja como objeto).

PUC-Rio - Certificação Digital Nº 0721352/CA

Lost Updates:

1. O Documento como um todo. Afinal o Documento será removido do sistema.
   Todos os valores de todas as propriedades do Documento serão removidos.
   Serão removidos todos os statements que tenham o Documento como sujeito.

   Solução:
   No início da transação Web, bloqueio de escrita para remoção em todas as propriedades do documento.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  #{args[:document]} locking:rWLockAt conf:all.
}
```

2. Todas as propriedades, de todos os recursos que tenham o Documento como valor
   Também serão removidos todos statements que tenham o Documento como objeto.

   Solução:
   No início da transação Web, bloqueio de escrita para remoção nas propriedades dos recursos que são *sujeito* de *statements* que possuam o documento a ser removido, como *objeto*.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  CONSTRUCT
  {
    ?subject locking:rWLockAt ?property
  }
  WHERE
  {
    ?subject ?property #{args[:document]}.
  }
}
```

PUC-Rio - Certificação Digital Nº 0721352/CA

Inconsistent Reads:

1. *Não podem ser inseridos novos statements que mencionem o Documento como objeto.*

Solução:

No início da transação Web, bloqueio de leitura para inserção, em todas as propriedades, de todos os recursos. Enfim, bloquear todo o grafo para inserções.

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  conf:all locking:iRLockAt conf:all.
}
```

Register Author

Ator Primário:

Autor.

Ator Secundário:

Não tem.

Pré-condição:

Se for um Autor pré-existente, então o Autor tem que ter feito login.

Descrição:

O sistema exibe o autor (um autor vazio, caso seja a criação de um novo autor). O usuário (autor) ou se atualiza (o nome, sobrenome, e-mail, endereço, titulação, organização) ou se remove. Para permitir atualizar a organização, o sistema exibe uma lista de instituições de acordo com a invariante ***AssignNotConflictingInstitution*** . O sistema exibe o autor atualizado. O usuário confirma sua alterações. O sistema salva o autor.

Importante:

Se o autor não tiver documentos associados a ele, ele pode ser removido. No entanto, apesar desta ser uma transação Web CRUD, a remoção de um autor deve ser realizada em uma transação Web separada, pois, de acordo com o *abordagem* pessimista, serão necessários bloqueios mais custosos para garantir a consistência. A remoção de um autor não será descrita pois os problemas de concorrência a serem evitados são similares ao descritos na transação Web ***Document Withdrawal***, porém num contexto diferente.

PUC-Rio - Certificação Digital Nº 0721352/CA

Invariantes:

<u>*AssignNotConflictingInstitution*</u>**:**

A Organização do autor tem que ser diferente das instituições de todos os revisores alocados a todos os seus documentos.

```
NOT [documents->select(a | a.producedBy
author).hasReview.producedBy.memberOf.includes(author.institution)]
```

Lost Updates:

1. O Autor por inteiro.

   Todas as propriedades podem ter valores inseridos e/ou removidos.

   Solução:

   No início da transação Web, bloqueio de escrita para remoção e inserção no autor.

```
%{
 @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
 #{args[:author]} locking:riWLockAt locking:all.
}
```

Inconsistent Reads:

Por questão de conflito de interesse, temos:

1. Não pode inserir um novo Documento associado ao Autor

   Não pode inserir um novo statement com a *propriedade "producedBy"* e o Autor como objeto.

   Solução:

   No início da transação Web, bloqueio de leitura para inserção na propriedade "*producedBy*".

```
%{
 @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
 @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
 Locking:all locking:iRLockAt conf:producedBy.
}
```

2. Não pode inserir um novo Revisor aos Documentos do Autor

   Não pode inserir um novo valor na propriedade "hasReview" de todos os Documentos do Autor.

   E para cada Revisão de cada Documento do Autor, não pode inserir um novo valor na propriedade "producedBy".

PUC-Rio - Certificação Digital Nº 0721352/CA

3. Não pode inserir uma nova Organização nos Revisores associado aos Documentos do Autor
Para cada Revisor, de cada Revisão, de cada Documento do Autor, não pode inserir um novo valor na propriedade "memberOf".

Solução para os itens 2 e 3 juntos:
No início da transação Web, bloqueio de leitura para inserção na propriedade "*hasReview*" de todos os documentos do autor. E para cada revisão, da cada documento do autor, bloqueio de leitura para inserção na propriedade "*producedBy*". E para o revisor de cada revisão, de cada documento do autor, bloqueio de leitura para inserção na propriedade "*memberOf*".

```
%{
  @prefix locking:<http://www.tecweb.inf.puc-rio.br/synth/locking.owl#>
  @prefix conf:<http://www.tecweb.inf.puc-rio.br/example/conference.owl#>
  CONSTRUCT
  {
    ?document locking:iRLockAt conf:producedBy.
    ?review   locking:iRLockAt conf:producedBy.
    ?reviewer locking:iRLockAt conf:memberOf.
  }
  WHERE
  {
    ?document conf:producedBy #{args[:author]}.
    ?document conf:hasReview ?review.
    ?review   conf:producedBy ?reviewer.
  }
}
```

4.3.Avaliação de Desempenho

O modelo de bloqueio proposto foi avaliado por meio de simulações, a fim de se obter evidências que:

- Os quatros grânulos propostos ("*Graph*", "*Property*", "*Resource*" e "*Property of Resource*") permitem obter um melhor desempenho, de acordo com os tamanhos das transações, se comparado apenas com o grânulo "*Resource"* (que é o que mais se aproxima de uma tupla no esquema relacional);

PUC-Rio - Certificação Digital Nº 0721352/CA

- O protocolo multigranular proposto, de fato, melhora o desempenho, com relação ao protocolo monogranular, quando temos transações de vários tamanhos;
- Os novos tipos de bloqueio de leitura e escrita, para inserção e remoção, compatíveis entre si, melhoram o nível de multiprogramação quando comparados com os tipos de bloqueio de leitura e escrita convencionais, incompatíveis.

Esta seção atém-se às conclusões oriundas da análise dos resultados das simulações. Contudo, todos detalhes das simulações, desde a estratégia e design até a implementação, bem como a exposição gráfica de todos os resultados obtidos, encontram-se no apêndice A.1 Simulação - Estratégia e Resultados.

Como vimos, o menor item de dados que pode ser bloqueado é o grânulo *"Property of Resource*", ou seja, par (propriedade, recurso). Todas as simulações foram conduzidas com um conjunto de 1000 (mil) transações, utilizando um banco de dados fictício contendo 300 (trezentos) recursos e 100 (cem) propriedades, o que resulta em um conjunto de 30.000 (trinta mil) pares (propriedade, recurso) passíveis de bloqueio. Adicionalmente, todas as simulações foram realizadas em dois cenários de escrita: 80% de operações de escritas e 20% de operações de escrita.

Com relação à carga, para cada transação foi definido, aleatoriamente, o respectivo perfil de acesso, ou seja, o conjunto de pares (propriedade, valor) a serem acessados. Dentre este pares, foram selecionados, aleatoriamente, os pares a serem escritos e lidos, numa quantidade correspondente ao cenário de escrita (80%, 20%).

Com relação à política de bloqueio, foi utilizado o esquema *abortar-reiniciar* (em inglês, *no-wait*), em caso de não obtenção de um bloqueio, sem espera para reiniciar (min_*restart_time e max_restart_time,* ambos iguais a zero). Para garantir um escalonamento serializável, foi empregado o protocolo de bloqueio em duas fases estrito (*strict 2PL*) [Bernstein e Newcomer, 2009].

A valor escolhido como parâmetro de desempenho foi o tempo de resposta médio, também conhecido como *turnaround* médio. Este compreende desde do início da transação até a obtenção da resposta, contabilizando, pois, eventuais abortos e reinícios. A unidade de medida usada foi *segundo*.

Antes das conclusões propriamente ditas, vale reiterar que o desempenho de qualquer modelo de bloqueio é influenciado, dentre outros fatores, por dois

PUC-Rio - Certificação Digital Nº 0721352/CA

componentes preponderantes e antagônicos, a saber: *taxa de conflitos* e *overhead de gerenciamento de bloqueios*. Quando o primeiro aumenta, o segundo diminui, e vice-versa. Em outras palavras, quanto maior o número de conflitos, menor a quantidade de bloqueios, e vice-versa.

Para aferir o desempenho de cada um dos quatro grânulos individualmente, empregou-se o protocolo monogranular, usando cada grânulo em separado. Para cada cenário de escrita (80% e 20%), foram simulados três conjuntos de transações de mesmo tamanho de acesso: 0.1%, 1% e 10% do banco de dados. Para cada conjunto destes, foram usados cada um dos quatro grânulos, separadamente. Em ambos os cenários de escrita, os resultados, do melhor para o pior desempenho, foram:

- 1000 transações de tamanho 0.1%
  Property of Resource, Resource, Property, Graph (Figura 38)
- 1000 transações de tamanho 1%
  Property of Resource, Resource, Property, Graph.
- 1000 transações de tamanho 10%
  Property, *Resource*, *Graph*, *Property of Resource*.

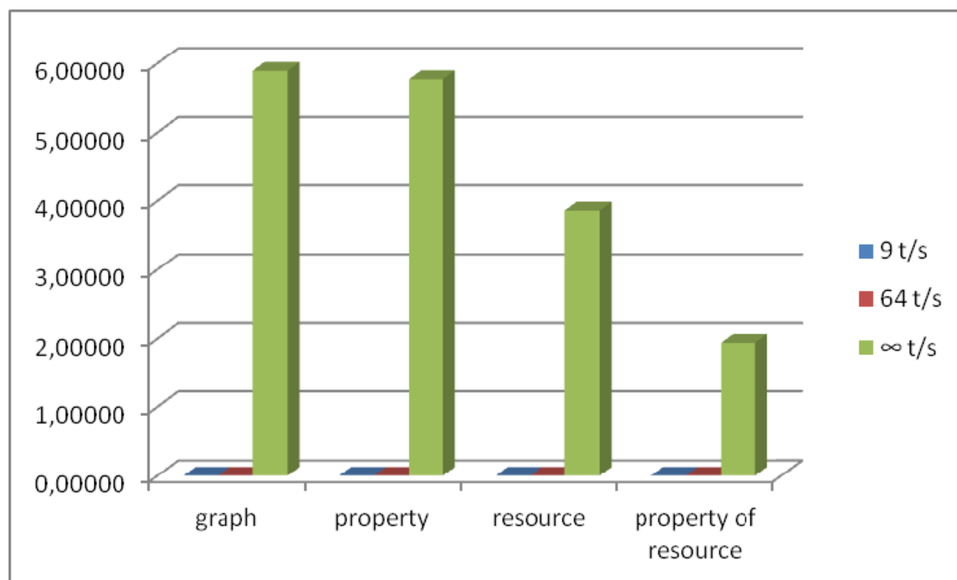

Figura 38 - *Turnaround* (seg) - Monogranular, Tamanho 0.1%, 80% de Escrita

Como era de se esperar, o menor grânulo "*Property of Resource*" obteve o melhor resultado. Contudo, com aumento do tamanho das transações, devido ao *overhead* de bloqueios, o desempenho do grânulo "*Property of Resource*" foi piorando e os grânulos maiores o superaram. Diante disso, conclui-se que para

PUC-Rio - Certificação Digital Nº 0721352/CA

transações pequenas, o grânulo "*Property of Resource*" aparece como uma ótima opção. Para transações de tamanho intermediário o grânulo "*Resource*" e, em alguns casos mais abrangentes, "*Property*" podem ser usados. Já para transações grandes, o "*Graph*" deve ser utilizado. Para ratificar esta última conclusão, uma vez que nos experimentos realizados até então não havia transações grandes o suficiente, foi realizada uma simulação extra, com 1000 (mil) transações de tamanho de acesso 20%, no cenário de 80% de escrita, onde o grânulo "*Graph*" obteve o *turnaround* mais curto, como pode ser visto na Figura 39.

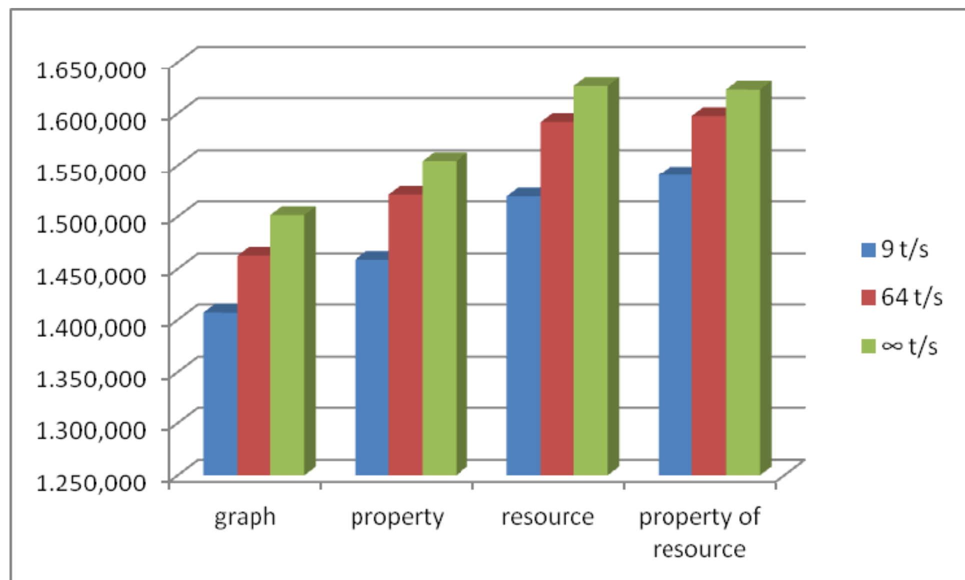

Figura 39 - *Turnaround* (seg) - Monogranular, Tamanho 20%, 80% de Escrita

Para a comparação entre o protocolo multigranular e o monogranular, para cada cenário de escrita (80% e 20%), foi feita uma simulação envolvendo transações com tamanho de acesso variando entre 0.1% e 10% do banco de dados. Como passo inicial, os grânulos foram simulados separadamente de novo, usando o protocolo monogranular e, em seguida, as mesmas transações foram simuladas com o protocolo multigranular. Na simulação multigranular, para decidir qual grânulo a transação deve bloquear é necessário decidir quando uma transação é considerada grande (ou abrangente) o suficiente para bloquear todo o grânulo. Isto se traduz em qual porcentagem mínima do total de pares (propriedade, recurso) "contidos" no grânulo, uma transação deve acessar para então bloquear todo o grânulo. Para este propósito, inspirado pelo experimento realizado em [Ries e Stonebraker, 1979], foram definidos alguns *threshold percentages* (tp) (percentuais de limiar) para se bloquear todo o grânulo. O

PUC-Rio - Certificação Digital Nº 0721352/CA

algoritmo tenta bloquear do maior grânulo ("*Graph*") para o menor ("*Property of Resource*"), de acordo com estes *thresholds.*

Em ambos os cenários, o tp 5% obteve o melhor desempenho (*turnaround* mais curto). Percebeu-se que neste tp, houve um melhor balanceamento ou distribuição dos bloqueios entre os quatro grânulos, corroborando o princípio básico da estratégia multigranular. Para efeitos de comparação, o melhor grânulo no protocolo monogranular, "*Property of Resource*", obteve, no cenário de 80% de escrita, um *turnaround* 33% mais demorado do que o *turnaround* do tp 5% e, no cenário de 20%, um *turnaround* 26% mais demorado do que o *turnaround* do tp 5%. A Figura 40 contém os *turnarounds* no cenário de 80%.

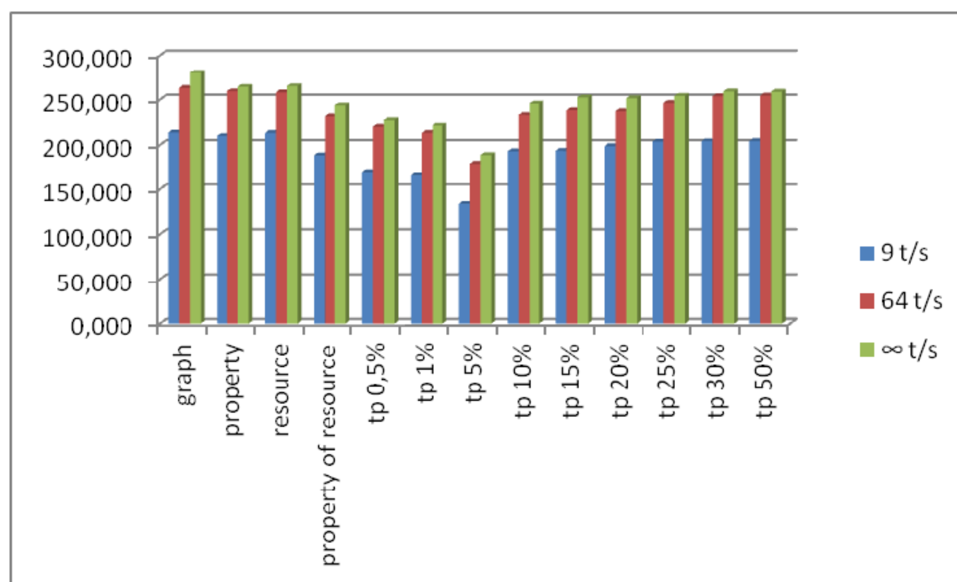

Figura 40 - *Turnaround* (seg) - Monogranular versus Multigranular, 80% de Escrita

Vale destacar que nem sempre o melhor desempenho é acompanhado pelo menor número de abortos. Como pode ser visto na Figura 41, no mesmo cenário de 80% anterior, o tp 5%, apesar de ter sido o melhor, gerou bem mais abortos do que alguns de seus concorrentes (tp 20%, tp 25%, etc.). O que aconteceu foi que o tp 5% obteve a melhor relação entre a taxa de conflitos (abortos) e o *overhead* gerado pelo gerenciamento de bloqueios.

PUC-Rio - Certificação Digital Nº 0721352/CA

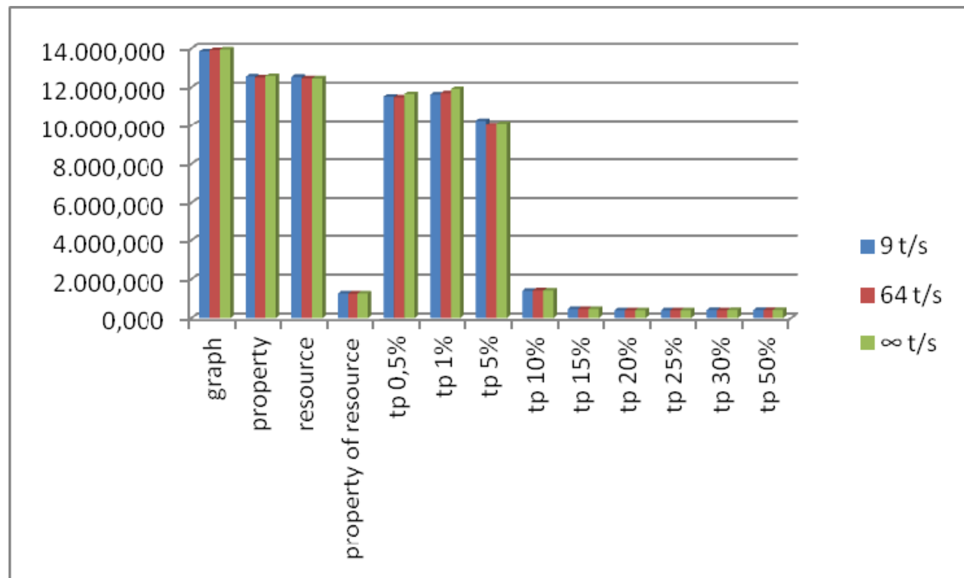

Figura 41 - Número de abortos - Monogranular versus Multigranular, 80% de Escrita

Para confirmar o aumento do paralelismo trazido pelos novos tipos de bloqueio, foi empregado o protocolo monogranular, usando o menor grânulo "*Property of Resource*". Para cada cenário de escrita (80% e 20%), foram simulados três conjuntos de transações de mesmo tamanho de acesso: 0.1%, 1% e 10% do banco de dados. Cada conjunto destes foi simulado duas vezes: ora usando os tipos de bloqueio de leitura e escrita convencionais incompatíveis (riR & riW), ora usando os novos tipos compatíveis (rR & iW), leitura para remoção e escrita para inserção. Os resultados foram os seguintes:

- 1000 transações de tamanho 0.1%

  Em ambos os cenários de escrita (80% e 20%) o ganho foi imperceptível. Com o aumento da taxa de chegada das transações, os tipos de bloqueio convencionais até foram ligeiramente melhores. Isto ocorreu porque são transações muito pequenas com baxíssima taxa de conflitos, logo quase não houve conflitos a reduzir.

- 1000 transações de tamanho 1%

  Com o aumento do tamanho das transações, o ganho dos novos tipos de boqueio se evidenciou.

  - No cenário de 80% de escrita, o número de abortos com os bloqueios convencionais foi 56% maior do que o número de abortos com os novos bloqueios. E

PUC-Rio - Certificação Digital Nº 0721352/CA

o *turnaround* com os bloqueios convencionais foi cerca de 25% mais demorado do que o *turnaround* com os novos bloqueios.

- No cenário de 20% de escrita, o número de abortos com os bloqueios convencionais foi 206% maior do que o número de abortos com os novos bloqueios (Figura 42). E o *turnaround* com os bloqueios convencionais foi cerca de 26% mais demorado do que o *turnaround* com os novos bloqueios (Figura 43).

- 1000 transações de tamanho 10%

  Com o aumento do tamanho das transações e o consequente aumento da quantidade de bloqueios, o *overhead* de gerenciamento de bloqueios começou a superar o efeito da redução do número de abortos e, desta forma, os bloqueios convencionais obtiveram um desempenho ligeiramente melhor.

  - No cenário de 80% de escrita, o número de abortos com os bloqueios convencionais foi 33% maior do que o número de abortos com os novos bloqueios. Todavia, o *turnaround* com os novos bloqueios foi cerca de 2% mais demorado do que o *turnaround* com os bloqueios convencionais.
  - No cenário de 20% de escrita, o número de abortos com os bloqueios convencionais foi 808% maior do que o número de abortos com os novos bloqueios (Figura 44). Não obstante, mesmo com esta expressiva diferença no número de abortos, a medida que a taxa de chegada das transações aumentou, o *turnaround* com os novos bloqueios foi cerca de 0.2% mais demorado do que o *turnaround* com os bloqueios convencionais (Figura 45).

PUC-Rio - Certificação Digital Nº 0721352/CA

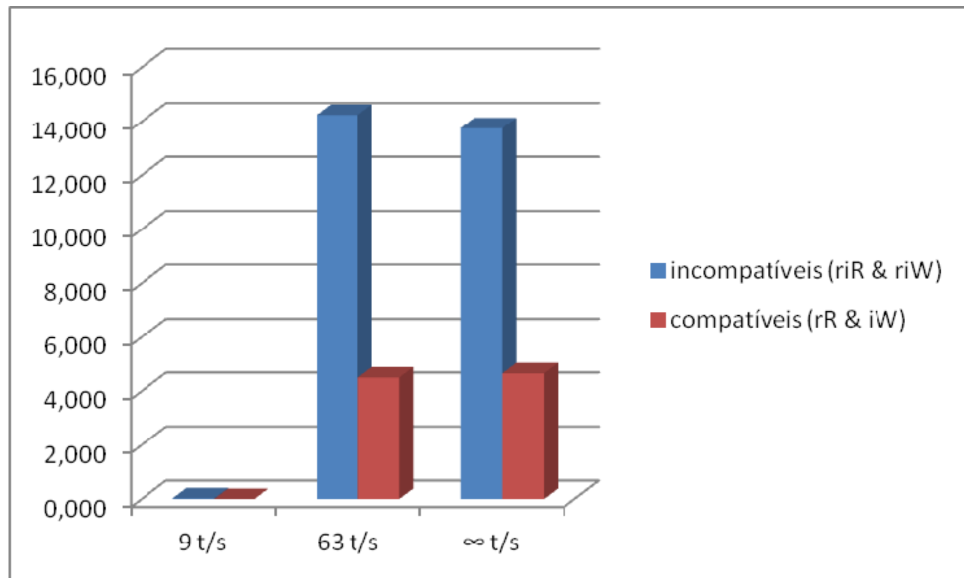

Figura 42 - Número de abortos - Tipos de Bloqueio, Tamanho 1%, 20% de Escrita

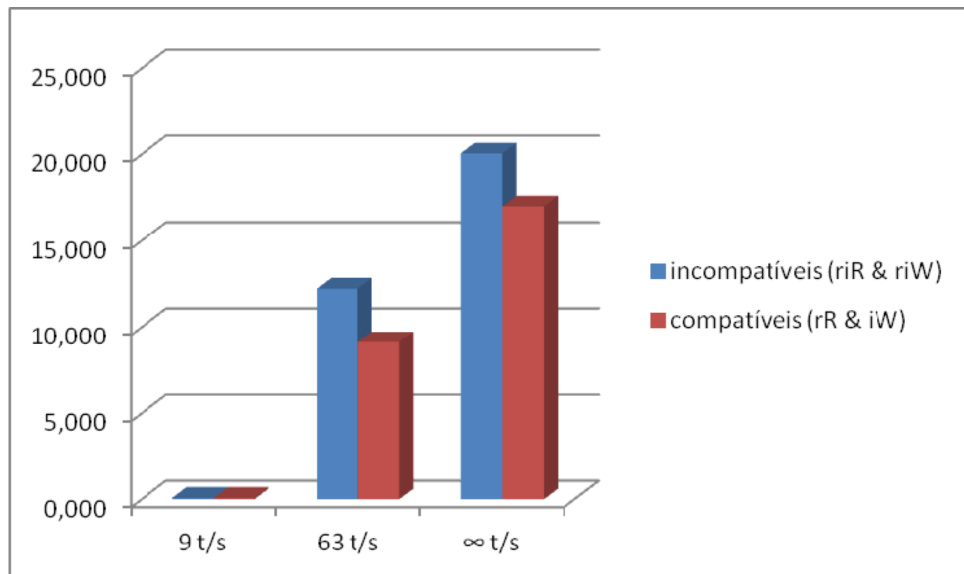

Figura 43 - *Turnaround* (seg) - Tipos de Bloqueio, Tamanho 1%, 20% de Escrita

PUC-Rio - Certificação Digital Nº 0721352/CA

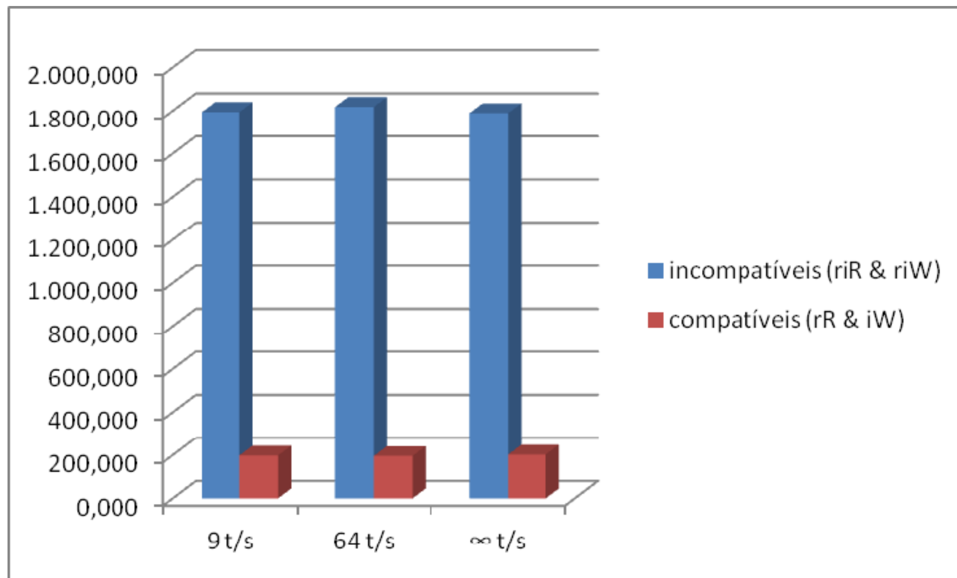

Figura 44 - Número de Abortos - Tipos de Bloqueio, Tamanho 10%, 20% de Escrita

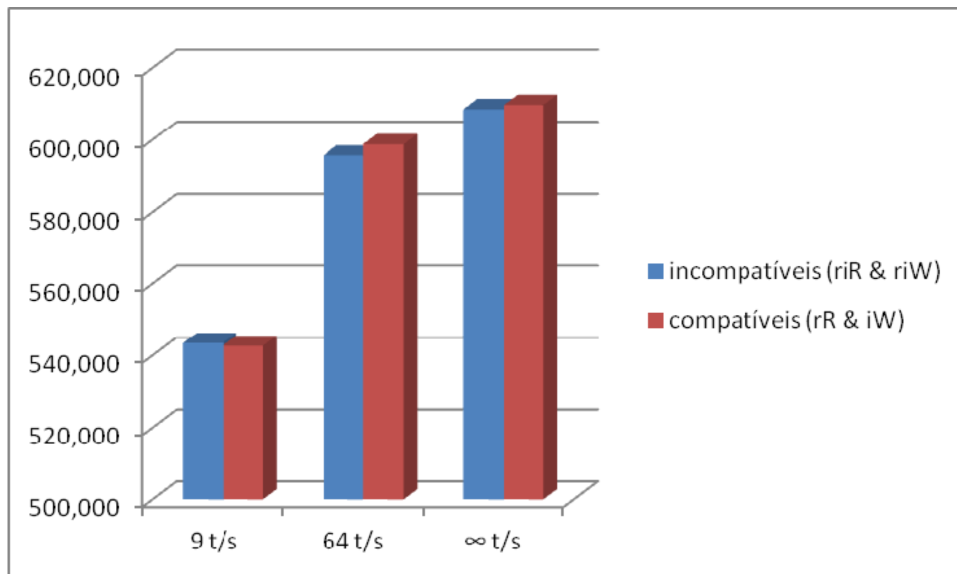

Figura 45 - *Turnaround* (seg) - Tipos de Bloqueio, Tamanho 10%, 20% de Escrita

Com base nos resultados, conclui-se que, de fato, os novos tipos de bloqueio reduzem a taxa de conflitos (abortos), em especial quando a maioria das operações é de leitura. No entanto, em função do *trade-off* fundamental entre taxa de conflitos e *overhead* de gerenciamento de bloqueios, existe um tamanho máximo de transação, onde a partir do qual a redução dos conflitos passa a não mais trazer ganho, pelo contrário, traz prejuízo, ocasionando uma espécie de *thrashing*.

PUC-Rio - Certificação Digital Nº 0721352/CA

4.4.Diretrizes de Utilização

Esta seção reune algumas diretrizes de utilização do modelo de bloqueio proposto. Como sabemos, concorrência é um tema bastante capcioso, logo, além destas diretrizes, cada transação deve sempre ser especificamente analisada.

Ao analisar os requisitos não funcionais de concorrência de uma transação, devemos começar pelos requisitos de escrita (*lost updates*) e, em seguida, os requisitos de leitura (*inconsistent reads*). Isto porque um bloqueio de escrita é sempre exclusivo com relação a outro bloqueio de escrita, logo, às vezes, um bloqueio de escrita já cobre um ou mais requisitos de leitura.

4.4.1. Bloqueio Otimista versus Bloqueio Pessimista

Devemos utilizar a abordagem otimista quando a probabilidade de conflitos for baixa e as consequências da perda do trabalho do usuário forem suaves ou toleráveis.

Já a *abordagem* pessimista deve ser escolhida quando a probabilidade de conflitos for alta ou as consequências trazidas por um conflito forem severas, acarretando grande insatisfação por parte do usuário.

4.4.2. Tipos de Bloqueio

Obviamente devemos sempre escolher a opção que contribua para um maior nível de multiprogramação.

Em se tratando de navegação sobre o conteúdo, na maoria dos casos, o objetivo consiste em mera exibição. Portanto, basta obter a última versão confirmada (*committed*) do dado, o que se traduz em utilizar bloqueio de leitura apenas durante o carregamento dos dados, liberando logo em seguida.

Quando for necessária a aquisição de bloqueio, devemos, sempre que possível, utilizar bloqueios apenas para inserção ou apenas para remoção, pois como vimos aumentam o grau de multiprogramação. Para isso, não basta identificar as propriedades sujeitas a alteração. Precisamos ir além e pensar em "como" (inserção e/ou remoção de valores) estas propriedades devem ser alteradas e qual a relação deste "como" com o nosso requisito de concorrência.

PUC-Rio - Certificação Digital Nº 0721352/CA

Com relação aos bloqueios de escrita, para as transações de edição de dados (as chamadas CRUD), em geral, utilizamos bloqueio de escrita para inserção e remoção, pois estamos fazendo alterações no conteúdo, o que, em RDF, significa inserir e remover. Já para outros tipos de transação, que disparam operações de negócio de domínio, precisamos analisar cuidadosamente quais as pós-condições das operações disparadas, para determinarmos o efeito que será gerado e, por conseguinte, os tipos de bloqueio de escrita mais adequados para cada propriedade afetada.

Já os bloqueios de leitura estão relacionados às pré-condições da operações de domínio. Devemos analisá-las para identificarmos quais propriedades não podem ser alteradas (inserção e/ou remoção) por uma segunda transação concorrente.

4.4.3.
Grânulos

Nos experimentos realizados, vimos que o limiar de acesso em torno de 5% do total de pares (propriedade, recurso) logicamente contidos no grânulo, já foi suficiente para se bloquear todo o grânulo.

Aferir qual porcentagem do grânulo uma transação acessa pode ser deveras trabalhoso. Envolve análise das ontologias utilizadas, "domain" das propriedades, etc. Esta certamente deveria ser uma tarefa automatizada, realizada não pelo projetista, mas por um *Lock Manager* um pouco mais inteligente. Seria a noção de bloqueio dinâmico multigranular, previsto como um dos trabalhos futuros desta tese.

No entanto, algumas heurísticas podem ser elencadas para ajudar o projetista, na escolha do grânulo a ser bloqueado.

Quando se tratar de uma transação CRUD, em geral, estamos editando um recurso (*subject* de trabalho) como um todo, portanto, devemos bloquear todo o recurso (grânulo "*Resource*").

No caso da chamada de uma ou mais operações de negócio de domínio, envolvendo algumas propriedades de alguns poucos recursos (*subjects* de trabalho) isolados, devemos bloquear, separadamente, as propriedades dos recursos envolvidos (grânulo "*Property of Resource*").

Quando se tratar de uma operação de domínio envolvendo propriedade(s) de vários recursos, devemos bloquear a(s) propriedade(s) como um todo (grânulo "*Property*"). Um exemplo comum seria o reajuste de preço de vários

PUC-Rio - Certificação Digital Nº 0721352/CA

produtos. Ao invés de bloquear todos os produtos, bloqueamos apenas a propriedade "preço".

Lembre-se que ao bloquear um grânulo envolvendo uma propriedade, devemos bloquear também a propriedade inversa (grânulo "Property") com o mesmo tipo de bloqueio.

Quando não purdermos precisar quais propriedades envolvidas, ou seja, quando quisermos bloquear quaisquer propriedades, então temos que bloquear o todo o grafo (grânulo "*Graph*"). Quando bloquearmos todo o grafo, devemos, obviamente, fazer a transação o mais curta possível.

Nas transações CRUD, a remoção de um recurso demanda que o mesmo não seja mencionado em nenhum *statement*, quer seja como subjeito, quer seja como objeto. Isto implica em bloquear todo o grafo (vide transação *Document Withdrawal*, seção 4.2.2.1- Exemplos Motivacionais - Transações Web). Portanto, devemos sempre projetar a remoção de um recurso como uma transação mais curta, separada da transação de edição CRUD.

PUC-Rio - Certificação Digital Nº 0721352/CA