UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANDRÉ CRISTIANO KALSING

Uma abordagem Incremental para Mineração de Processos de Negócio

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Cirano Iochpe
Orientador

Porto Alegre, julho de 2012.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Kalsing, André Cristiano

Uma abordagem Incremental para Mineração de Processos de Negócio / André Cristiano Kalsing. – Porto Alegre: PPGC da UFRGS, 2012.

110 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2012. Orientador: Cirano Iochpe.

1. Mineração de Processos. 2. Workflow. 3. Mineração Incremental de Processos. 4. Sistemas Legados. I. Iochpe, Cirano. II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Rui Vicente Oppermann
Pró-Reitora de Pós-Graduação: Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço a minha esposa Aline pelo apoio, conversas, compreensão e paciência. Aos meus familiares pelo estímulo adicional dado durante este período.

Agradeço ao meu orientador, Prof. Dr. Cirano Iochpe, pelas lições estimulantes que me ensinaram muito durante o desenvolvimento deste trabalho.

Aos meus colegas do grupo de pesquisa, pelo auxílio e amizade. Um agradecimento especial à Lucinéia Thom e Gleison Nascimento, pelo grandíssimo apoio, pelo estímulo e por todos os ensinamentos que contribuíram tanto para este trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS **6**
LISTA DE FIGURAS **7**
LISTA DE TABELAS **9**
RESUMO **10**
ABSTRACT **11**
1 INTRODUÇÃO **12**
1.1 Motivação e Contribuição **14**
1.2 Objetivo e Metodologia **15**
1.3 Trabalhos Relacionados **16**
1.3.1 Algoritmos de Mineração de Processos 16
1.3.2 Métodos de Extração a Partir do Código Fonte 17
1.4 Estrutura do Texto **18**
2 INTRODUÇÃO À MINERAÇÃO DE PROCESSOS **19**
2.1 Conceitos Básicos **19**
2.2 Modelos formais utilizados em *Process Mining* **22**
2.2.1 Redes de Petri 22
2.2.2 Redes de *Workflow* (WF-net) 24
2.3 Aspectos da Mineração de Processos **26**
2.3.1 Minerando Tarefas Ocultas 26
2.3.2 Minerando Tarefas Duplicadas 27
2.3.3 Construções Sem-Escolha-Livre 27
2.3.4 Minerando Laços 28
2.3.5 Tempo de Uso 28
2.3.6 Minerando Diferentes Perspectivas do Processo 29
2.3.7 Lidando com Ruídos 29
2.3.8 Lidando com Incompleteza 29
2.4 Minerando Decisões **30**
2.5 Métricas de Análise **35**
2.5.1 Cobertura (*Fitness*). 36
2.5.2 Precisão (*Behavioral Appropriateness*). 36
2.5.3 Generalização. 36
2.5.4 Estrutura (Structural Appropriateness). 36
2.6 Ferramenta ProM **37**
2.6.1 Arquitetura 37
2.6.2 Formato do *Log* de Processos 38
3 ALGORITMOS INCREMENTAIS DE MINERAÇÃO **40**
3.1 Mineração Incremental Baseada no Versionamento de Documentos **41**
3.1.1 Abordagem Incremental de Mineração 42

3.1.2 Vantagens e Limitações 45
3.2 Mineração Incremental com Padrões Opcionais 46
3.2.1 Definições do algoritmo 47
3.2.2 Algoritmo Incremental de Mineração 47
3.2.3 Descoberta de Padrões Opcionais 51
3.2.4 Vantagens e Limitações 52
3.3 Mineração Incremental com Laços 53
3.3.1 *Framework* do algoritmo de mineração 53
3.3.2 Mineração incremental com laços 54
3.3.3 Vantagens e Limitações 56
3.4 Comparativo das abordagens 56
4 ABORDAGEM INCREMENTAL DE MINERAÇÃO 58
4.1 Modelo Conceitual 59
4.1.1 Modelo Conceitual de Processo de Negócio 59
4.1.2 Conjunto de Regras de Negócio 61
4.2 Definição de Cenários de Execução dos Usuários 65
4.3 Execução dos Cenários 67
4.4 Mineração do *Log* de Execução 69
4.4.1 Operações de atualização 69
4.4.2 Definição do Algoritmo 71
4.4.3 Lidando com Ruídos no *Log* 78
4.4.4 Atualização de um Modelo Completo 79
4.4.5 Mapeando o Resultado da Mineração para o Modelo Conceitual de Processo . 82
4.5 Apresentação do Resultado da Mineração 84
5 EXPERIMENTOS 87
5.1 Experimentos Utilizando um Sistema de Informação Legado 87
5.1.1 Geração dos *Logs* de Eventos 87
5.1.2 Análise de Performance 89
5.1.3 Análise da Qualidade 91
5.1.4 Resultado da Mineração do Sistema de Informação 93
5.1.5 Validação do Modelo Resultante 95
5.2 Experimentos Utilizando Dados Simulados 97
5.2.1 Geração dos *Logs* de Eventos 98
5.2.2 Experimentos com Mineração sem Modificações (Não Incremental) 98
5.2.3 Experimentos com Mineração Incremental com Modificações nos Processos . 99
6 CONCLUSÃO 101
6.1 Principais Contribuições 101
6.2 Trabalhos Futuros 102
REFERÊNCIAS 104

LISTA DE ABREVIATURAS E SIGLAS

B2B	Business to Business
BAM	Business Activity Monitoring
BI	Business Inteligence
BPA	Business Process Analysis
BPEL	Business Process Engine Language
BPM	Business Process Management
CRM	Customer Relationship Management
DAG	Directed Acyclic Graph
EPC	Event-driven Process Chain
ERP	Enterprise Resource Planning
GM	GeneticMiner
HM	HeuristicMiner
I/O	Input/Output
IM	IncrementalMiner
LTL	Linear Temporal Logic
MXML	Log Metamodel XML
PAIS	Process-Aware Information System
ProM	Process Mining Tool
SAP	Systems, Applications and Products in Data Processing
SGWf	Sistema de Gerenciamento de Workflow
SNA	Social Network Analysis
UML	Unified Modeling Language
WF	WorkFlow
WfMC	Workflow Management Coalition

LISTA DE FIGURAS

Figura 2.1: O ciclo de vida do processo é usado para ilustrar a mineração de processos e analise Delta em relação ao projeto de processos tradicional................................. 20
Figura 2.2: Visão geral da mineração de processos 21
Figura 2.3: Um modelo de processo que corresponde ao *log* de processo da Tabela. 22
Figura 2.4: Outro modelo de processo tupla (P,T,F) 22
Figura 2.5: Um modelo de processo com duas tarefas escondidas 26
Figura 2.6: Um modelo de processo com tarefas duplicadas. 27
Figura 2.7: Um modelo de processo com construções sem-livre-escolha 28
Figura 2.8: Um modelo de processo com laços. 28
Figura 2.9: Um modelo de processo onde é dicífil identificar a sincronização. 30
Figura 2.10: A abordagem da mineração de decisões 31
Figura 2.11: Fase de Mineração de Processos 32
Figura 2.12: Fragmento de um log de exemplo no formato MXML usando a ferramenta XML Spy 33
Figura 2.13: Pontos de decisão representados como problemas de classificação 34
Figura 2.14: Modelo de Processo Melhorado 35
Figura 2.15: Avaliação do modelo de processo baseado em diferentes dimensões 37
Figura 2.16: Visão geral do framework ProM 38
Figura 2.17: Formato do log de processo MXML 39
Figura 3.1: Mineração do *log* de execução parcial 41
Figura 3.2: Mineração incremental dos novos eventos no *log* 41
Figura 3.3: Esquema de mineração a partir do versionamento de documentos. 42
Figura 3.4: Esquema de Mineração Incremental 43
Figura 3.5: Passos do algoritmo incremental 43
Figura 3.6: Modelo de processo extraído a partir do log de versionamento 44
Figura 3.7: Processo médico com uma tarefa opcional 46
Figura 3.8: LearnWorkflowNet: algoritmo que minera e gera o modelo de processo a partir do log. 48
Figura 3.9: Processo de mineração incremental de processos 49
Figura 3.10: (a) LearnOrdering: aprende uma relação de ordenação a partir do modelo (b) LearnIndependence: aprende uma relação de independência a partir do modelo. ... 50
Figura 3.11: (a) *OrderingForIncremental*: um algoritmo para combinar o relacionamento de ordenação obtido a partir da rede de workflow e dos dados de log de processo. (b) *IndependenceForIncremental*: um algoritmo para combinar o relacionamento de independência obtido a partir da rede de workflow e dos dados de log de processo. 51

Figura 3.12: AddOption: um algoritmo para adicionar padrões opcionais 52
Figura 3.13: Representação formal de uma estrutura de laço 53
Figura 3.14: Algoritmo LearnOraclesFromModel 54
Figura 3.15: Um exemplo de modelo existente e um novo log..................................... 55
Figura 3.16: Algoritmo *CompleteOracles*. 56
Figura 4.1: Mineração Incremental de Processos a partir de *logs* de execução de sistemas de informação... 58
Figura 4.2: Modelo conceitual de processos baseado no modelo da WfMC. 60
Figura 4.3: Exemplo de modelo de processo representando as entidades do modelo conceitual.. 61
Figura 4.4: Listagem com exemplo de código fonte instrumentado para geração de eventos no *log*.. 62
Figura 4.5: Listagem com exemplo de código fonte modificado, gerando trace de eventos modificados. 62
Figura 4.6: Cenários de execução do sistema de informação.. 66
Figura 4.7: *Log* resultante da execução dos cenários de usuário. 68
Figura 4.8: Pseudocódigo do algoritmo *IncrementalMiner*... 73
Figura 4.9: Árvores de Dependência. Mantém a confiança e suporte atualizados para cada relação de dependência extraída do *log*.. 76
Figura 4.10: Árvores das Melhores Relações extraídas a partir da Árvore de Dependência da Figura 77
Figura 4.11: Grafo de Dependência e os participantes extraídos a partir do log de execução *W*. 78
Figura 4.12: Cálculo das Relações Irmãs Obsoletas... 81
Figura 4.13: Grafo de dependência atualizado. 82
Figura 4.14: Modelo conceitual do grafo de dependência gerado pelo algoritmo IncrementalMiner 83
Figura 4.15: Ferramenta de mineração incremental 84
Figura 4.16: Visualização da semântica do processo 85
Figura 4.17: Visualização do processo de negócio.. 86
Figura 5.1: Sistema ERP legado e seus módulos e relacionamentos.............................. 88
Figura 5.2: Tempo total de processamento durante a mineração de novos traces adicionados ao *log* 90
Figura 5.3: Escalabilidade dos algoritmos (sem mineração incremental)..................... 90
Figura 5.4: *Zoom* da Figura 91
Figura 5.5: Exemplos de modelos de processos parciais extraídos do sistema legado .. 94
Figura 5.6: Processo de verificação do modelo minerado durante a mineração incremental. 100

LISTA DE TABELAS

Tabela 2.1: Um log de processo. 21
Tabela 3.1: Log obtido a partir do Sistema de Gerenciamento de Versões.... 44
Tabela 3.2: Resumo do comparativo entre as abordagens incrementais 57
Tabela 4.1: Taxonomia de Regras com Exemplos 64
Tabela 4.2: Exemplos de cenários de execução dos usuários do sistema apresentado na Figura.... 66
Tabela 4.3: Diferentes tipos de atualização de um modelo de processo durante a mineração incremental.... 70
Tabela 4.4: *Thresholds* do algoritmo *IncrementalMiner*.... 79
Tabela 4.5: Correspondência entre os elementos do modelo conceitual do algoritmo *IncrementalMiner* e o modelo conceitual de processo de negócio.... 83
Tabela 5.1: Comparativo de tempo (em segundos) 89
Tabela 5.2: Comparativo das métricas de qualidade entre *IncrementalMiner* (IM), *HeuristicMiner* (HM), *Genetic Miner* (GM) e *Alpha++* 92
Tabela 5.3: Elementos extraídos incrementalmente a partir do aplicação Financeira.... 93
Tabela 4: Matriz de responsabilidades por participante extraídas do processo da Figura 94
Tabela 5.5: Resultado da estatística Kappa 95
Tabela 5.6: Resultado do cálculo para o aspecto Mapeamento de Atividades (Regras de Negócio) 97
Tabela 5.7: Avaliação de qualidade para *IncrementalMiner*, α-*algorithm* e o método de Ma et al. 99
Tabela 5.8: Similaridade média entre os modelos resultantes da mineração incremental. 99

RESUMO

Até os dias de hoje, diversos algoritmos de mineração de modelos de processos já foram propostos para extrair conhecimento a partir de *logs* de eventos. O conhecimento que tais algoritmos são capazes de obter incluem modelos de processos de negócio, assim como aspectos da estrutura organizacional, como atores e papéis. A mineração de processos pode se beneficiar de uma estratégia incremental, especialmente quando as informações sobre um ou mais processos de negócio presentes no código fonte de um sistema de informação são logicamente complexas (diversas ramificações e atividades paralelas e/ou alternativas). Neste cenário, são necessárias muitas execuções da aplicação para a coleta de um grande conjunto de dados no arquivo de *log*, a fim de que o algoritmo de mineração possa descobrir e apresentar o processo de negócio completo. Outra situação que torna necessária a mineração incremental é a constante evolução dos processos de negócio, ocasionada geralmente por alterações nas regras de negócio de uma ou mais aplicações. Neste caso, o *log* pode apresentar novos fluxos de atividades, ou fluxos alterados ou simplesmente fluxos que não são mais executados. Estas mudanças devem ser refletidas no modelo do processo a fim de garantir a sincronização entre a aplicação (processo executado) e o modelo. A mineração incremental de processos pode ainda ser útil quando se faz necessária a extração gradual de um modelo de processo completo, extraindo modelos parciais (fragmentos de processo com início e fim) em um primeiro passo e integrando conhecimento adicional ao modelo em etapas até a obtenção do modelo completo. Contudo, os algoritmos atuais de mineração incremental de processos não apresentam total efetividade quanto aos aspectos acima citados, apresentando algumas limitações. Dentre elas podemos citar a não remoção de elementos obsoletos do modelo de processo descoberto, gerados após a atualização do processo executado, e também a descoberta de informações da estrutura organizacional associada ao processo como, por exemplo, os atores que executam as atividades.

Este trabalho propõe um algoritmo incremental para a mineração de processos de negócio a partir de *logs* de execução. Ele permite a atualização completa de um modelo existente, bem como o incremento de um modelo de processo na medida em que novas instâncias são adicionadas ao *log*. Desta forma, podemos manter ambos, modelo de processo e o processo executado sincronizados, além de diminuirmos o tempo total de processamento uma vez que apenas novas instâncias de processo devem ser consideradas. Por fim, com este algoritmo é possível extrair modelos com acurácia igual ou superior aqueles que podem ser extraídos pelos algoritmos incrementais atuais.

Palavras-Chave: Mineração de Processos, Workflow, Mineração Incremental de Processos, Sistemas Legados.

Incremental Approach to Business Process Mining

ABSTRACT

Even today, several process mining algorithms have been proposed to extract knowledge from event logs of applications. The knowledge that such algorithms are able to discovery includes business process models, business rules, as well as aspects of organizational structure, such actors and roles of processes. These process mining algorithms can be divided into two: non-incremental and incremental. The mining process can benefit from an incremental strategy, especially when information about the process structure available in the system source code is logically complex (several branches and parallel activities). In this scenario, its necessary several executions of the application, to collect a large set of log data, so that the mining algorithm can discover and present the complete business process. Another use case where incremental mining is usefull is during the changing structure of the process, caused by the change in the business logic of an application. In this case, the log may provide new traces of activities, modified traces or simply traces that are no longer running. These changes must be reflected in the process model being generated to ensure synchronization between the application and model. The incremental process mining can also be useful when it is necessary to extract a complete process model in a gradual way, extracting partial models (process fragments with begin and end) in a first step and integrating additional knowledge to the model in stages to obtain the complete model. However, existing incremental process mining algorithms are not effective to all aspects mentioned above. All of them have limitations with respect to certain aspects of incremental mining, such as deletion of elements in the process model (process model update). Additionally, most of them do not extract all the information present in the structure of the process, such as the actors who perform the activities.

This paper proposes an incremental process mining algorithm from execution logs of information systems. The new algorithm allows the full update (adding and removing elements) of an existing model, as well as the increment of a process model as new records are added to the log. Thus, we can keep process models and process execution syncronized, while reducting the total processing time, since only new process instances must be processed. Finally, are expected the extraction of process models with similar or higher accuracy compared to current incremental mining algorithms.

Keywords: Process Mining, Workflow, Incremental Process Mining, Legacy Systems.

1 INTRODUÇÃO

De acordo com os padrões de terminologia da WfMC – Workflow Management Coalition (WfMC, 1999), um processo de negócio compreende o conjunto de um ou mais procedimentos ou atividades relacionadas, as quais, coletivamente, realizam um objetivo de negócio no contexto de uma estrutura organizacional. Além disso, os processos de negócio encapsulam o conhecimento das operações e serviços prestados por uma organização. Por exemplo, o processo de negócio seguido quando um livro é comprado em um site pode consistir de uma série de tarefas como verificar a disponibilidade do livro, a necessidade de reabastecer o estoque, ou verificar a validade do cartão de crédito do cliente.

Normalmente, um fluxo de trabalho, ou *workflow*, representa um processo de negócio. Ele descreve as tarefas essenciais, sua ordem parcial de execução, os participantes, os papéis de negócio que assumem e os recursos necessários à execução de cada etapa do processo. Para o exemplo da compra de um livro citado acima, o fluxo de trabalho descreve os participantes (como o cliente e o fornecedor), as tarefas tomadas por cada participante, e a ordem de execução destas, juntamente com as decisões para suas execuções.

Durante a última década, os conceitos e tecnologia de gerenciamento do fluxo de trabalho (AALST; DESEL; OBERWEIS, 2000), (AALST, HEE, 2002), (JABLONSKI; BUSSLER, 1996), (LEYMANN, ROLLER, 1999) foram aplicados em sistemas de informação em muitas empresas. Sistemas de gestão de fluxo de trabalho, tais como Staffware, IBM MQSeries, e COSA, ofereceram modelos genéricos e capacidade de execução automatizada para processos de negócios estruturados. Ao fazer definições de processos em interfaces gráficas, ou seja, modelos que descrevem o ciclo de vida de um caso (instância do fluxo de trabalho) de forma isolada, pode-se configurar esses sistemas para apoiar os processos de negócios.

Além de sistemas específicos para a gestão de fluxo de trabalho, muitos outros sistemas de software têm adotado a tecnologia de workflow. Considere, por exemplo, sistemas ERP (Enterprise Resource Planning), como SAP, PeopleSoft, Baan e Oracle, e sistemas de gerenciamento do relacionamento com o cliente, como CRM (Customer Relationship Management). Outro exemplo são os serviços Web. Todas as linguagens de composição ou orquestrações de Web Services como BPEL, BPMN e EPC adotaram os principais conceitos de workflow.

Sistemas de gerenciamento de workflow mais recentes são guiados por modelos de processo explícitos. Ou seja, uma fase de projeto (modelagem) do fluxo de trabalho é necessária para definir um processo de negócio, o qual pode ser executado somente após esta fase. A criação de um modelo de fluxo de trabalho é um procedimento que, muitas vezes, pode ser complexo e demorado. Normalmente, existem discrepâncias

entre os processos de negócio reais, existentes nas organizações, e os processos modelados que estão sob gerenciamento destes sistemas. Outro problema está no fato de sistemas de informação mais antigos não possuirem um modelo de processo explícito. Isto resulta na demanda por novos métodos para apoiar este processo, em especial quando a transição para arquiteturas mais modernas está em pauta. As principais demandas para estas novas arquiteturas são a adoção de linguagens de programação orientadas a objetos, o advento das tecnologias *Web* e, especificamente, as arquiteturas orientadas a serviços (SOA – Service Oriented Architecture) (OASIS, 2008).

SOA está se tornando cada vez mais uma prática na engenharia de software. Relatórios mostram que mais de 50% das grandes aplicações recentemente desenvolvidas e processos de negócios projetados durante o ano de 2007 utilizaram de alguma forma conceitos de arquitetura orientada a serviços (ABRAMS; SCHULTE, 2008). Entretanto, a experiência também indica que as iniciativas de SOA raramente começam do zero. De acordo com (KRILL, 2006), em 2006 70% dos sistemas das grandes empresas eram sistemas legados (Bisbal; LAWLESS; WU; GRIMSON, 1999), rodando linguagens como COBOL, C e C++. A maioria das empresas necessitam manter sistemas legados ativos pelo fato destes implementarem, na maioria das vezes processos de negócio centrais à organização e de alta complexidade, aumentando de forma considerável o risco associado a falhas dos mesmos. Informações mais recentes da empresa Gartner estimam (com 0,8 de probabilidade) que, em 2011, mais de 80% das aplicações existentes estariam sendo ou já tinham sido, pelo menos parcialmente, modernizadas para participarem de arquiteturas orientadas a serviços (NATIS; PEZZINI; SCHULTE; IIJIMA, 2006). Isto representa um esforço significativo para os departamentos de TI das organizações.

Com o crescimento da adoção de SOA, a necessidade de uma abordagem sistemática para a reengenharia de sistemas legados se torna cada vez mais necessária. As abordagens de reengenharia visam extrair destes sistemas regras e processos de negócio, permitindo seu reaproveitamento na adaptação ou criação de novas aplicações. Assim, é possível economizar tempo e dinheiro durante o processo de modernização destes sistemas, pois boa parte do investimento e conhecimento aplicado no passado nestes sistemas pode ser resgato novamente.

Técnicas de mineração de processos de negócio (COOK; WOLF, 1998), (AALST; DONGEN; HERBST; MARUSTER; SCHIMM; WEIJTERS, 2003) possibilitam extrair informações a partir do *log* de eventos ocorridos durante reiteradas execuções de um sistema legado, levando à descoberta de modelos de processos de negócio inerentes ao código de tal sistema. Esta abordagem também pode ser usada para comparar o processo capturado e o processo de negócios projetado, para identificar discrepâncias entre ambos modelos. Além de sistemas legados, podemos aplicar as técnicas de mineração de processos a quaiquer sistemas de informação que suportem a geração de arquivos de *logs* a partir de sua execução. Alguns exemplos incluem sistemas de gerenciamento de workflow, BPM, sistemas ERP, CRM e B2B.

Adicionalmente aos algoritmos de mineração de processos, existem vários outros algoritmos e métodos para a extração de modelos de processos de negócio a partir de sistemas de informação, como por exemplo análise estática do código fonte. Técnicas baseadas na análise estática de código fonte (ZOU, 2006) (ZOU, 2004), (LIU, 1999) geralmente extraem o processo de negócio diretamente a partir de estruturas presentes no código fonte da aplicação, como comandos IF, WHILE, chamadas de função, etc. Assim, o modelo de processo final apresenta uma estrutura muito semelhante ao

algoritmo programado no código fonte, o que pode muitas vezes não revelar todas as informações sobre o processo, tais como tarefas concorrentes e os participantes que executam cada atividade, pois são informações que estão disponíveis apenas em tempo de execução da aplicação.

Algoritmos de mineração de processos existentes (MA; TANG; WU, 2011), (WEN; AALST; WANG; SUN, 2009), (MEDEIROS; WEIJTERS; AALST, 2007), (WEIJTERS; AALST; MEDEIROS, 2006) extraem informações sobre o comportamento do software diretamente a partir de *logs* de execução do sistema, usando técnicas de aprendizagem de máquina. Estes algoritmos podem identificar estruturas simples, bem como estruturas mais elaboradas, como por exemplo, controles de fluxo condicionais, concorrência, laços e participantes do processo, utilizando análise do comportamento dinâmico do sistema. Esses algoritmos também lidam com inconsistências presentes no *log*, como ruídos, que podem ser gerados a partir de erros presentes no código fonte do sistema ou por qualquer outro problema na execução da aplicação. Adicionalmente, uma parte destes algoritmos é capaz de minerar arquivos de log incrementalmente, permitindo a atualização de um modelo inicial de processo descoberto a medida que novas entradas são adicionadas ao arquivo de *log*.

1.1 Motivação e Contribuição

Embora os algoritmos de mineração de processos atuais sejam eficazes para a extração de modelos de processos de negócios a partir de sistemas legados, eles ainda apresentam algumas limitações. As duas principais são: (i) não possuem performance adequada durante a mineração de *logs* de execução com grande número de registros e (ii) não suportam todos os aspectos da mineração incremental destes *logs* como, por exemplo, a remoção de elementos que não mais fazem parde de um determinado modelo de processo.

A primeira limitação refere-se a degradação de performance (baixa escalabilidade) dos algoritmos quando o processo de mineração precisa ser repetido diversas vezes em um arquivo de log de execução grande (ex: milhares de instâncias) e que continua crescendo. Em tais casos, o comportamento completo do processo só pode ser aproximado por um conjunto de dados de execução cada vez maior a ser capturado no arquivo de log. Já a segunda limitação está relacionada à falta de suporte para a descoberta de eventos de atualização (remoção de elementos) do modelo durante a mineração incremental do *log*. Neste caso, após a descoberta de um modelo de processo parcial ou final, quaisquer mudanças ocorridas no processo (remoção de atividades, tarefas, transições e atores) necessitam ser descobertas e incorportadas a este modelo. Assim, a premissa desta pesquisa é de que, tanto a mineração de grandes arquivos de *logs*, como a mineração de *logs* com eventos representando modificações nos processos, poderiam ser mais eficiente, tanto do ponto de vista de tempo execução, como do ponto de vista de qualidade do modelo. E a hipótese é de que isso pode ser obtido através da aplicação de uma estratégia incremental de mineração modificada. A abordagem, aqui proposta, deve permitir que se extraia resultados parciais a partir do *log* (ou seja, modelos de processos parciais) o mais cedo possível, e posteriormente, o incremento do modelo com novos eventos gerados, quando estes estiverem disponíveis. Além disso, se espera que a mineração do log gerado, a partir da execução do sistema legado, possa revelar informações adicionais sobre os processos de negócios, como as atividades concorrentes e os atores de uma atividade, informações estas que seriam complexas de serem obtidas a partir da análise estática do código fonte.

Neste trabalho, é proposto um novo algoritmo para a extração, de forma incremental, de estruturas de processos de negócio a partir de sistemas de informação. O algoritmo aqui proposto utiliza técnicas de mineração de processos (COOK; WOLF, 1998), para extrair as estruturas que representam o comportamento dinâmico da aplicação. As maiores contribuições desta nova abordagem são: i) a extração incremental de estruturas de processo de negócio, considerando operações de adição e remoção de elementos (atividades, transições, controles de fluxo, etc) do modelo, a partir de sistemas de informação; e ii) a extração de informações complementares do processo de negócio, como os atores de cada tarefa, controles de fluxo que representam concorrência (ex: AND-split/join).

O algoritmo *IncrementalMiner* ((KALSING; IOCHPE; THOM, 2010a) desenvolvido durante a pesquisa para o mestrado, é responsável pela extração incremental de processos de negócio. Este algoritmo foi criado com base no algoritmo *HeuristicMiner* (WEIJTERS; AALST; MEDEIROS, 2006). A principal razão em adotar o *HeuristicMiner* como base ao invés de outros algoritmos de mineração existentes como, por exemplo, *α Miner* (AALST, 2004c), *Alpha++* (WEN, 2007), *ProbabilistMiner* (SILVA; ZHANG; SHANAHAN, 2005) ou *GeneticMiner* (MEDEIROS; WEIJTERS; AALST, 2007), foi basicamente a sua precisão satisfatória, assim como o seu bom suporte para extrair as principais construções de processos de negócios (por exemplo, XOR/AND-split/join, laços, etc).

O algoritmo foi aplicado na mineração incremental de *logs*, utilizando duas bases de dados de *logs* de processo: uma base empírica, gerada por um simulador de processos, e uma base real gerada a partir de reiteradas execuções de um sistema legado. Na sua avaliação de desempenho, o IncrementalMiner mostrou alta performance para extrair os processos de negócios de *log* com grandes volumes de eventos, sendo ainda mais rápido quando utilizado na mineração incremental de diversos *logs* parciais de eventos. O tempo total de processamento dos *logs* foi reduzido em 64% durante a mineração incremental. O tempo de resposta do *IncrementalMiner*, para o mesmo conjunto de eventos, foi cinco vezes menor do que o tempo de execução do algoritmo *Alpha++* (WEN, 2007) e oitenta vezes mais rápido do que o tempo de execução do *HeuristicMiner* para o arquivo com o log completo. Os modelos de processos extraídos mostraram precisão igualmente satisfatória quando comparados com os resultados dos demais algoritmos de mineração não incrementais. Já para os modelos gerados durante a mineração incremental, o algoritmo *IncrementalMiner* mostrou precisão superior, quando utilizado no processamento de *logs* que continham mudanças no processo, como por exemplo a remoções de atividades obsoletas do modelo. Isto foi possível devido ao seu suporte para descoberta de eventos implícitos de remoção de elementos do processo, tarefa esta não suportada pelos demais algoritmos incrementais.

1.2 Objetivo e Metodologia

O principal objetivo deste trabalho é a pesquisa e o desenvolvimento de uma abordagem incremental para a mineração de processos de negócio, abordagem esta capaz de absorver o conhecimento descoberto também de forma incremental. Esta pesquisa foi realiza no contexto de um projeto mais amplo que visa o desenvolvimento de uma técnica de reescrita de sistemas legados baseada em tecnologia SOA e BPM (NASCIMENTO; IOCHPE; THOM; REICHERT, 2009), (RADAELLI JUNIOR; NASCIMENTO; IOCHPE, 2011), (NASCIMENTO; IOCHPE; KALSING; THOM; MOREIRA, 2012).

Para atingir este objetivo, seguiu-se a seguinte metodologia de trabalho:

1. Levando em consideração a deficiência na extração de estruturas de processos da abordagem apresentada em (NASCIMENTO et al, 2009), realizou-se um levantamento bibliográfico sobre as principais técnicas de mineração de processos (Trabalho Individual de Mestrado, 2008);
2. Identificou-se que não havia até então, abordagens eficientes e completas para esta tarefa. Com isso, decidiu-se adaptar os algoritmos existentes para alcançar o objetivo esperado (KALSING et al, 2010a).
3. Aplicação do algoritmo derivado em um estudo de caso real (KALSING et al, 2010b);
4. Criação de uma extensão do algoritmo de mineração incremental para suporte a identificação de modificações na estrutura do processo (KALSING et al, 2012);
5. Avaliação do modelo resultante através do método de estatística Kappa (KALSING, 2010b).

1.3 Trabalhos Relacionados

A extração de processos de negócios a partir de sistemas de informação tornou-se o foco de muitos projetos de pesquisa na atualidade. Podemos dividir os trabalhos relacionados em dois grupos: 1) mineração de processo a partir do log de execução e 2) extração de processos de negócios a partir do código fonte.

1.3.1 Principais Algoritmos de Mineração de Processos

O primeiro grupo inclui os algoritmos de mineração de processos. Eles permitem a mineração de rastros (*traces*) de execução de um sistema para extrair informações como processos de negócios, aspectos de estruturas organizacionais e regras de negócio (van der Aalst, 2003). Um dos principais algoritmos que utiliza a abordagem de mineração de processos é o *GeneticMiner* (MEDEIROS, 2007). Este algoritmo utiliza métodos de busca adaptativa que simulam o processo de evolução. Quando comparado aos demais algoritmos de mineração, ele apresenta melhor precisão do modelo extraído, especialmente para a mineração de processos que apresentam comportamento mais complexo. No entanto, este algoritmo de mineração de processos muitas vezes necessita de alto tempo de processamento para chegar ao resultado, além de utilizar apenas registros históricos de *log*, não considerando a mineração incremental de processos de negócios a partir do *log* de eventos. A presente abordagem, por outro lado, suporta a mineração incremental, o que leva a uma redução considerável do tempo total de processamento sem comprometer a qualidade do resultado. Já outro algoritmo de mineração que pode ser destacado é o algoritmo criado por Ma et al (MA; TANG; WU, 2011). Ele permite a mineração incremental dos *logs* de execução de um sistema de informação, permitindo gerar um modelo parcial inicial e realizar sua atualização a medida que novas entradas são geradas no *log*. A grande restrição deste algoritmo é sua limitação com relação a atualização do modelo na mineração incremental. Atualmente apenas operações de complementação (adição de tarefas e transições) são suportadas pelo algoritmo.

Outros dois trabalhos de destaque proporam a identificação de modificações no processo a partir do *log*. No primeiro deles, Gunter el tal (GUNTHER; RINDERLE-MA; REICHERT; AALST, 2008) propos novas técnicas para lidar com as limitações

descritas anteriormente. Apesar de suas técnicas não preverem a mineração incremental de logs de execução, ele introduziu a mineração de alterações *ad hoc* em instâncias de processos de negócio. Para isto, foram criados eventos especializados no *log* para descrever modificações nas instâncias de processos (ex: evento de inserção de tarefa, evento de remoção de tarefa, etc). Estes eventos registram no *log* todas modificações possíveis em uma instância de processo. Desta forma, *logs* com modificações de processos são interpretados diferentemente de *logs* convencionais onde o conteúdo descreve a execução de um processo de negócio. Neste *log* especializado, o autor considera uma sequência de operações de adição e remoção obtidas a partir de eventos de modificação de uma instância de processo. O problema com esta abordagem é que, em alguns casos, sistemas de informação e SGWf não geram no *log* informações sobre modificações do processo (ex: sistemas ERP legados). Neste caso pode ser difícil descobrir mudanças no processo a partir destes sistemas.

No segundo trabalho proposto para identificação de modificações de processo no *log*, Bose et al (BOSE; AALST; ZLIOBAITÉ; PECHENIZKIY, 2011) introduziu o conceito de derivação de conceitos (*Concept Drift*) aplicado a mineração de processos. Ele aplicou técnicas para detecção, localização e classificação de modificações do processo diretamente no *log* de execução, sem a necessidade de um *log* especializado contendo tais modificações, conforme proposto por Gunter et al. A abordagem principal deste trabalho visa detectar potenciais modificações no controle de fluxo do processo, manifestados como desvios súbidos sobre um período de tempo, de acordo com os eventos registrados no *log*. Para isto, o autor utilizou uma técnica chamada teste estatístico hipotético, a qual utiliza procedimentos para avaliar uma hipótese contra um conjunto de dados de exemplo. O principal objetivo do trabalho de Bose et al era demonstrar os possíveis pontos de mudança de um modelo de processo, e não fornecer um algoritmo de mineração capaz de derivar modelos atualizados.

O algoritmo proposto neste trabalho permite a mineração incremental de *logs* de execução, suportando as principais operações de atualização do modelo, como adição de atividades e transições e também a remoção destes elementos do modelo. Além disso, foi possível notar que o presente algoritmo obteve em alguns casos melhores resultados com relação a qualidade dos modelos minerados, mesmo durante a mineração de *logs* históricos (sem a abordagem incremental).

1.3.2 Métodos de Extração a Partir do Código Fonte

O segundo grupo inclui os métodos de extração de processos de negócios a partir do código fonte de sistemas de informação. A abordagem proposta em (ZOU et al, 2004) apresenta um *framework* de recuperação de processos dirigido por modelos, o qual captura as características funcionais essenciais representadas em um processo de negócio. Ele usa técnicas de rastreamento estático e heurísticas para realizar o mapeamento de entidades do código fonte (ex: comandos If, While, etc) para entidades de processo de alto nível. Em outro trabalho, Zou (ZOU et al, 2006) compara as características estruturais do fluxo de trabalho projetado com o fluxo de trabalho implementado (existente na organização), utilizando um modelo intermediário de comportamento. Finalmente, Liu (LIU et al, 1999) usa uma abordagem de recuperação de requisitos que se baseia em três etapas básicas: 1) captura do comportamento; 2) modelagem do comportamento dinâmico, e 3) derivação dos requisitos como documentos formais. Todas as abordagens acima descritas usam somente informações estáticas, encontradas no código fonte, para extrair modelos de processos de negócios a partir da análise de sistemas de informação. Este tipo de abordagem não permite extrair

modelos de processo tão completos quanto os obtidos na abordagem de mineração, pois se restrigem apenas às informações disponíveis no código fonte. O algoritmo proposto neste trabalho utiliza o comportamento dinâmico disponível apenas em tempo de execução, permitindo extrair elementos como fluxos concorrentes e participantes de um fluxo de trabalho. É importante ressaltar novamente que nossa abordagem permite a integração das técnicas atuais de análise estática do código com o modelo de extração a partir da execução do sistema. Desta forma, é possível extrair uma estrutura de processos de negócios mais completa e mais próxima da representação real do processo que está implicitamente descrita no sistema de informação.

1.4 Estrutura do Texto

No Capítulo 2 é revisado o tema mineração de processos de negócio. Uma introdução conceitual sobre o assunto é apresentada, assim como as principais estruturas para representação do modelo de processo.

No Capítulo 3 são apresentados os principais algoritmos de mineração incremental, apresentando seu comportamento e principais vantagens e limitações. Ao final do capítulo é apresentado um resumo qualitativo dos algoritmos, baseado nas observações realizadas ao longo do capítulo.

O Capítulo 4 apresenta a abordagem incremental de mineração proposta neste trabalho, a qual representa a principal contribuição desta pesquisa. Este mesmo capítulo descreve o funcionamento do algoritmo *IncrementalMiner*, componente fundamental na extração incremental das estruturas de processo do *log*. Complementarmente é apresentado o modelo conceitual de processo que foi utilizado para armazenar o resultado da mineração. Também é apresentada a ferramenta de mineração desenvolvida especialmente para dar suporte à extração incremental de modelos de processos a partir do *log*.

Um estudo de caso é apresentado no Capítulo 5. Neste foram desenvolvidos diversos experimentos que demonstraram a performance e qualidade dos modelos minerados a partir do algoritmo *IncrementalMiner*. Um sistema legado real foi utilizado para geração dos *logs* e mineração do modelo de processo. Para verificar se o modelo extraído representa o comportamento de processo contido no sistema legado, um modelo estatístico baseado no método Kappa foi utilizado. Esta verificação levou em consideração o nível de concordância entre observadores (analistas de negócio) sobre o modelo de processo minerado. Além do estudo de caso, neste capítulo foram também descritos experimentos realizados sobre arquivos de *logs* empíricos, gerados a partir de um simulador de execução de processos. Assim como no estudo de caso, nestes experimentos foram avaliados critérios de performance e qualidade dos modelos gerados. Porém, foram utilizados cenários não previstos no estudo de caso, como tratamento de ruídos, atualização incremental do modelo (remoção de atividades e atores do modelo) e atividades executadas em laço.

Finalizando, no Capítulo 6 são apresentadas as conclusões do trabalho, reiterando as principais contribuições do mesmo e fornecendo sugestões para trabalhos futuros envolvendo o tema abordado.

2 INTRODUÇÃO À MINERAÇÃO DE PROCESSOS

O objetivo da mineração de processos é extrair um modelo de processo explícito a partir de *logs* de execução de um sistema de informação. O desafio é criar o modelo de processo, dado um *log* de eventos, tal que o modelo seja consistente com o comportamento dinâmico observado neste *log*. Além disso, a mineração de processos também foca nas relações de causalidade entre atividades. Assim, ao invés de iniciar com a modelagem de um processo, como é tradicionalmente esperado, mineração de processos inicia obtendo informação sobre os processos como eles são executados. Inicia-se assumindo que é possível registrar eventos com informações sobre a ordem em que estes eventos são executados. Qualquer sistema de informação pode oferecer esta informação de alguma forma, mesmo que seja necessária a sua customização. Desta forma, não é necessária a presença de um Sistema Gerenciador de Workflow (SGWf) para geração destes *logs* de processo.

2.1 Conceitos Básicos

O objetivo das técnicas de mineração de processos é fazer a engenharia reversa de um processo de negócio a partir da coleta de dados em tempo de execução, para então suportar a analise e projeto de *workflows*. É fato que em muitos casos, antes da implantação de um sistema de automação de processos, este processo já estava presente na empresa de alguma forma. Ou seja, a organização já utilizava algum tipo de sistema de informação que de alguma forma gerava dados ou *logs* de processos de forma informal. A informação coletada em tempo de execução a partir destes *logs* pode ser usada para derivar um modelo que explicará os eventos registrados, conforme a Figura 2.2. Este modelo pode ser usado então em ambas fases de diagnóstico e reengenharia de processos. Assim, obtendo um conjunto de execuções reais de processos (eventos de *log*) como ponto de partida, é possível descobrir um processo e verificar sua conformidade com relação ao comportamento observado.

Para comparar a mineração de processos com a abordagem tradicional que vai em direção ao projeto e execução de processos, consideremos o ciclo de vida do processo mostrado na Figura 2.1. O ciclo de vida consiste em quatro fases: (i) projeto de *workflow*, (ii) configuração de workflow, (iii) execução do *workflow*, e (vi) diagnóstico do *workflow*. Na abordagem tradicional, a fase de projeto é usada para construir um modelo de *workflow*. Isto é tipicamente feito por um analista de negócio e é dirigido por idéias gerenciais sobre melhorias dos processos de negócio. Se o projeto é finalizado, o sistema de *workflow* (ou qualquer outro sistema que suporte o processo) é configurado como especificado nesta fase. Na fase de configuração é necessário lidar com as limitações e particularidades do sistema de gerenciamento de *workflow* que será usado. Na fase de execução, casos (instâncias de processos) são gerados pelo sistema de *workflow* como especificados na fase de projeto e realizados na fase de configuração.

Baseado na execução do processo, é possível coletar informações que serão analisadas na fase de diagnóstico. A fase de diagnóstico pode novamente prover entrada para a fase de projeto completando o ciclo de vida do processo. Na abordagem tradicional o foco está na fase de projeto e configuração. Menos atenção é dada para a fase de execução e poucas organizações sistematicamente coletam dados em tempo de execução, os quais são analisados como entrada para a reengenharia do processo, deixando assim a fase de diagnóstico geralmente ausente. Além disso, após a execução e mineração de *logs*, é possível ainda a realização de uma análise delta (AALST, 2004a) ou análise de conformidade (ROZINAT; AALST,2006b) para comparar o comportamento registrado no *log* com algum modelo de processo já existente a fim de detectar possíveis desvios. Esta análise é útil para verificar a existência de possíveis falhas, desvios ou melhorias que podem ser realizadas sobre o modelo de processo.

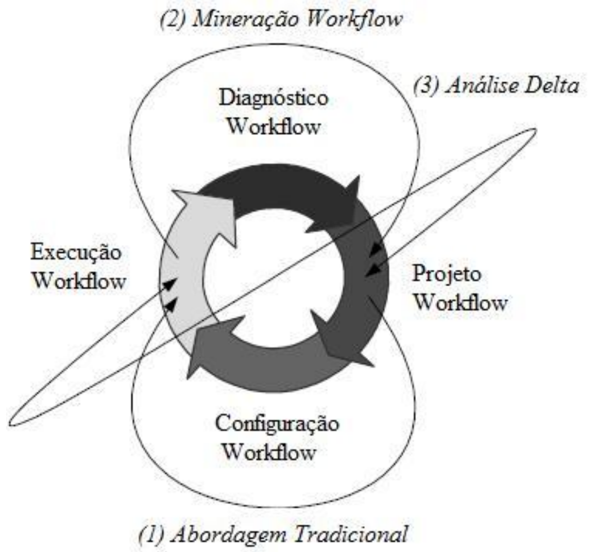


Figura 2.1: O ciclo de vida do processo é usado para ilustrar a mineração de processos e analise Delta (AALST, 2004a) em relação ao projeto de processos tradicional (AALST et al., 2003).

Os autores Cook e Wolf (COOK; WOLF, 1998) e van der Aalst et al (AALST; MARUSTER, 2004b) usam o termo mineração de processos para descrever o método de descoberta da estrutura de um processo de negócio a partir de um conjunto real de execuções de um sistema de informação. Para ilustrar os princípios da mineração de processos, consideremos o *log* de processo mostrado na Tabela 2.1. Este *log* contém informações sobre cinco casos (instâncias de processos). O *log* mostra que para quatro casos (casos 1, 2, 3 e 4) as tarefas A, B, C e D foram executadas. Para o quinto caso, somente duas tarefas diferentes foram executadas: tarefas E e F. Se a tarefa B é executada, então a tarefa C também será. Porém, para alguns casos, a tarefa C é executada antes da tarefa B. Baseado nesta informação mostrada na tabela e fazendo algumas suposições sobre a completeza do *log* (assume-se que os casos são representativos e um conjunto suficientemente grande de possíveis comportamentos são observados dentro do *log*), podemos deduzir, por exemplo, o modelo de processo mostrado na Figura . O modelo é representado em termos de uma rede de Petri (REISIG; ROZENBERG, 1998). A rede de Petri pode iniciar com a tarefa A e terminar com a tarefa D. Estas tarefas são representadas por transições. Após executar A, as tarefas B e C são habilitadas em paralelo. Note que para este exemplo é assumido que duas tarefas são paralelas se elas aparecem em qualquer ordem dentro do *log*.

Para este simples exemplo, é fácil construir um modelo de processo que é capaz de reproduzir um *log* de processo. Para modelos de processo maiores e mais elaborados (ex: diversas atividades paralelas, laços, etc), esta tarefa pode se tornar muito mais complexa. Por exemplo, se um modelo exibe alternativas e rotas paralelas, então o *log* de processo tipicamente não conterá todas possíveis combinações. Considere um processo com 10 tarefas, cada qual executada em paralelo. O número total de combinações poderá chegar a 10! = 3628800.

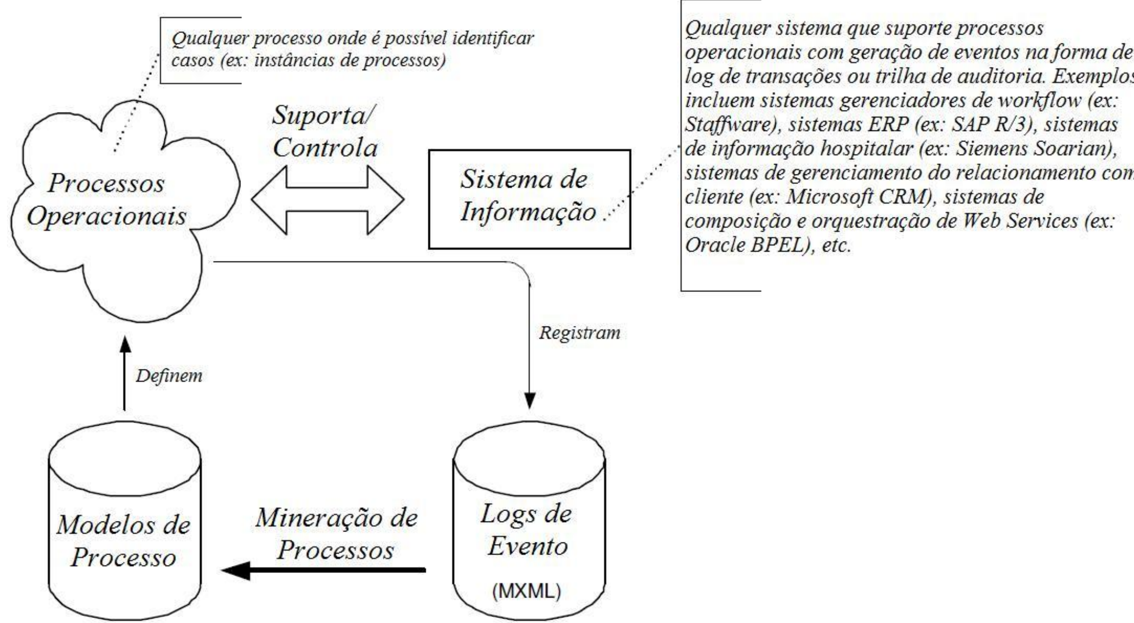


Figura 2.2: Visão geral da mineração de processos (MEDEIROS; WEIJTERS; AALST, 2007).

Apesar da constatação realiza acima, não faz parte da realidade presumir que todas as combinações estarão presentes no *log*. Além disso, certos caminhos através do modelo de processo poderão ter uma baixa probabilidade de executar e permanecerão indetectáveis. Dados com ruídos (ex: erros no *log*) podem complicar ainda mais esta questão. Assim, estes são apenas algums dos problemas com os quais é necessário lidar ao trabalhar com mineração de processos.

Tabela 2.1: Um *log* de processo.

ID Caso	ID Tarefa
Caso 1	Tarefa A
Caso 2	Tarefa A
Caso 3	Tarefa A
Caso 3	Tarefa B
Caso 1	Tarefa B
Caso 1	Tarefa C
Caso 2	Tarefa C
Caso 4	Tarefa A
Caso 2	Tarefa B
Caso 2	Tarefa D
Caso 5	Tarefa E
Caso 4	Tarefa C

Caso 1	Tarefa D
Caso 3	Tarefa C
Caso 3	Tarefa D
Caso 4	Tarefa B
Caso 5	Tarefa F
Caso 4	Tarefa D

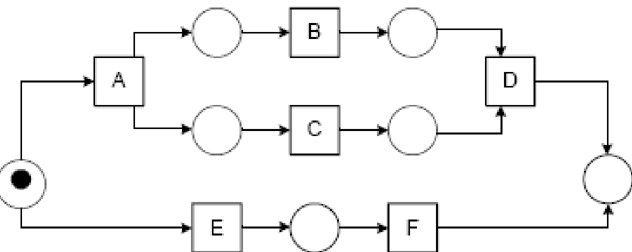


Figura 2.3: Um modelo de processo que corresponde ao *log* de processo da Tabela 2.1 (AALST; WEIJTERS, 2004).

2.2 Modelos formais utilizados em *Process Mining*

Os modelos formais descritos nesta seção são modelos utilizados na apresentação dos resultados da grande maioria dos algoritmos de mineração. Neste trabalho, estes modelos representam grafos de processos de negócio decorados com elementos semânticos específicos para este contexto.

2.2.1 Redes de Petry

Nesta seção é apresentado um modelo variante das redes de Petri clássicas, chamado rede Transição/Lugar (*Place/Transition*). Este modelo de rede de Petri será referenciado ao longo deste trabalho, através dos algortimos de mineração que utilizam este método formal para descrição dos processos de negócio. Detalhes do modelo tradicional de redes de Petri não serão abordados aqui e podem ser obtidos em (REISIG; ROZENBERG, 1998),(DESEL; ESPARZA, 1995) e (MURATA, 1989).

Definição 1 (P/T-nets) Uma rede de Transição/Lugar, ou simplesmente P/T-net, é uma rede onde:

1. P é um conjunto de lugares finito,

2. T é um conjunto de transições finito tal que $P \cap T = 0$, e

3. $F \subseteq (P \text{ x } T) \cap (T \text{ x } P)$ é um conjunto de arcos dirigidos, chamado relação de fluxo.

Uma rede P/T-net marcada é um par (N,s), onde (P,T,F) é uma rede P/T-net e onde s é um pacote sobre P denotando a marca da rede. O conjunto de todas as redes marcas P/T-net é denominado N.

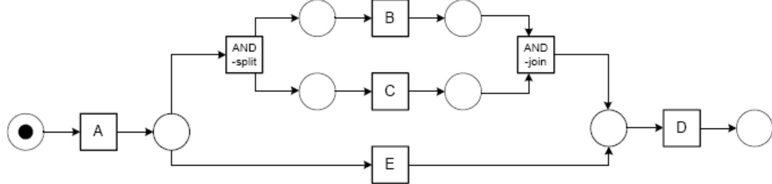


Figura 2.4: Outro modelo de processo (DER; A.; MARUSTER, 2004). tupla (P,T,F)

Uma marcação é um pacote sobre um conjunto de lugares P. Ela é uma função de P para os números naturais. Usa-se conchetes para enumerar os pacotes, como por exemplo $[a^2, b, c^3]$ para denotar o pacote com dois a's, um b, e três c's. A soma de dois pacotes $(X + Y)$, a diferença $(X - Y)$, a presença de um elemento no pacote $(a \in X)$ e a noção de sub-pacotes $(X \leq Y)$ é fortemente definida e eles podem ligar com uma mistura de conjuntos e pacotes.

Seja $N = (P, T, F)$ uma rede P/T-net. Elementos de $P \cup T$ são chamados nós. Um nó x é um nó de entrada de outro nó y se e somente se existe um arco dirigido de x até y, ou seja $(x, y) \in F$. O nó x é um nó de saída de y se e somente se $(y, x) \in F$. Para qualquer $\overset{N}{\bullet} x \in P \cup T$, $x \overset{N}{\bullet} = \{y \mid (x, y) \in F\}$ and $x = \{y \mid (x, y) \in F\}$.

A Figura 2.4 mostra uma rede P/T-net consistindo de 8 lugares e 7 transições. A Transição A tem um lugar de entrada e um lugar de saída, a transição AND-*split* tem um lugar de entrada e dois de saída, e a transição AND-*join* tem dois lugares de entrada e um de saída. O ponto preto no lugar de entrada de A representa um *token*. Este *token* denota a marca inicial. O comportamento dinâmico de uma rede P/T-net marcada é definido pela regra de disparo abaixo.

Definição 2 (Regra de Disparo) Seja $(N = (P, T, F), s)$ uma rede marcada P/T-net. A Transição $t \in T$ está habilitada, e é denotada por $(N, s)[t\rangle$, se e somente se $\bullet t \leq s$. A regra de disparo $_ [_ \rangle _ \subseteq N \times T \times N$ é a menor relação que satisfaz qualquer $(N = (P, T, F), s) \in N$ e qualquer $t \in T$, $(N, s)[t\rangle \Rightarrow (N, s)\ [t\rangle\ (N, s - \bullet t + t\bullet)$.

Na marca mostrada na Figura 2.4, um *token* no lugar de origem dá condições para a transição A ser habilitada. Disparando esta transição, remove-se o *token* do lugar de entrada e coloca-o no lugar de saída. Na marca resultante, duas transições estão habilitadas: E e AND-*split*. Embora ambos estejam habilitadas somente uma pode disparar. Se o AND-*split* dispara, um *token* é consumido e dois *token* são produzidos.

Definição 3 (Marcações Alcançáveis) Seja (N, s_0) uma rede marcada P/T-net em N. Uma marca s é alcançável a partir da marca inicial s_0 se e somente se existe uma sequência de transições habilitadas cujo disparo conduz s_0 até s. O conjunto de marcas alcançáveis de (N, s_0) é denotado por $[N, s_0\rangle$.

A rede marcada P/T-net mostrada na Figura 2.4 tem marcas alcançáveis. Algumas vezes é conveniente mostrar a sequência de transições que são disparadas para alcançar alguma dada marcação. A notação a seguir foi utilizada para descrever sequências. Seja A um alfabeto de identificadores. Uma sequência de tamanho n, para algum número natural $n \in IN$, sobre um alfabeto A é uma função $\sigma : \{0,..., n - 1\} \rightarrow A$. Uma sequencia de tamanho zero é chamada sequencia vazia e escrita como ϵ. Para maior legibilidade, uma sequência de tamanho positivo é usualmente escrito pela justaposição dos valores da função. Por exemplo, a sequencia σ: $\{(0, a), (1, a), (2, b)\}$, para a e $b \in A$, é escrito como *aab*. O conjunto de todas as sequências de tamanho arbitrário sobre o alfabeto A é escrito como A^*.

Definição 4 (Disparando sequências) Seja (N, s_0) com $N = (P, T, F)$ uma rede marcada P/T-net. Uma sequencia $\sigma \in T^*$ é chamada sequência de disparo de (N, s_0) se e somente se, para algum número natural $n \in IN$, existem marcações $s_1,...,s_n$ e transições $t_1,....t_n \in T$ tal que $\sigma = t_1...t_n$ e, para todo i com $0 \leq i < n$, $(N, {}_{si})[t_{i+1}\rangle$ e $s_{i+1} = si - \bullet t_{i+1}$

+ t_{i+1}•. Pode-se notar que $n = 0$ implica que $\sigma = \epsilon$ e que ϵ está disparando a sequência de (N, $s0$). A sequencia σ é dita habilitada na marcação s_0, denotado por $(N, s_0)[\sigma\rangle$. Disparando a sequencia σ resulta na marcação s_n, denotado por $(N, s_0)[\sigma\rangle\ (N, s_n)$.

Definição 5 (Conectividade ou Connectedness) Uma rede $N = (P, T, F)$ é fracamente conectada, ou simplesmente conectada, se e somente se, para cada dois nós x e y em $P \cup T$, $x(F \cup F^{-1})^{*}y$, onde R^{-1} é a inversa e R^{*} o encerramento reflexivo e transitivo da relação R. A rede N é fortemente conectada se e somente se, para cada dois nós x e y, $xF^{*}y$.

Assumi-se que todas as redes são fracamente conectadas e tem ao menos dois nós. A rede P/T-net mostrada na Figura 2.4 não é fortemente conectada porque não existe caminho direto de D para A.

Definição 6 (Limitação e Segurança) A rede marcada ($N = (P, T, F), s$) é limitada se e somente se o conjunto de marcações alcançáveis $[N, s\rangle$ é finito. Ele é seguro se e somente se, para qualquer $s' \in [N, s\rangle$ e qualquer $p \in P$, $s'(p) \leq 1$. Podemos notar que segurança implica em limitação. A rede marcada P/T-net mostrada na Figura Figura 2.4 é segura e também limitada pois nenhum dos oito estados alcançáveis coloca mais que um *token* em um lugar.

Definição 7 (Transições mortas e Liveness) Seja ($N = (P, T, F), s$) uma rede marcada P/T- net. A transição $t \in T$ é morta em (N,s) se e somente se não existirem marcações alcançáveis $s' \in [N, s\rangle$ tal que $(N, s')[t\rangle$. (N, s) é viva se e somente se, para cada marcação alcançável $s' \in [N, s\rangle$ e $t \in T$, existe uma marcação alcançável $s'' \in [N, s'\rangle$ tal que $(N, s'')[t\rangle$. Podemos notar que a vivacidade implica na ausência de transições mortas.

Na Figura 2.4 nenhuma das transições são mortas. Porém, a rede não está viva pois não é possível habilitar cada transição continuamente.

2.2.2 Redes de *Workflow* (WF-net)

A maioria dos sistemas de workflow oferecem blocos de construção padronizados como AND-*split*, AND-*join*, OR-*split* e OR-*join* (JABLONSKI; BUSSLER, 1996), (LEY-MANN; ROLLER, 1999). Estas construções são usadas para modelar sequências, condições, roteamentos paralelos e iterativos. Claramente, uma rede de Petri pode ser usada para especificar o controle de fluxo de um processo. Tarefas são modeladas por transições e dependências causais são modeladas por lugares e arcos. De fato, um lugar corresponde a uma condição, a qual pode ser usada como pré-condição e/ou pós-condição para tarefas. Um AND-*split* corresponde a transições com dois ou mais lugares de saída (paralelismo), e um AND-*join* corresponde a uma transição com dois ou mais lugares de entrada (sincronização). OR-*split*/OR-*join* correspondem a lugares com múltiplos arcos de saídas/entradas. Segundo Aalst et al (AALST; MARUSTER, 2004), uma rede de Petri que modela o fluxo de controle de um *workflow*, é chamada de rede de *Workflow* (WF-net). Uma rede WF-net especifica o comportamento dinâmico de um simples caso em isolado.

Definição 8 (Redes de *Workflow*) Seja $N = (P, T, \bar{F})$ uma rede P/T-net e t um identificador recente que não está em $P \cup T$. N é uma rede *workflow* (WF-net) se e somente se:

1) *Criação do objeto*: P contém um lugar de entrada i tal que $\bullet i = 0$,

2) *Objeto de conclusão*: P contém um lugar de saída o tal que $o\bullet = 0$,

3) Conectividade: $\mathrm{N} = (\mathrm{P}, \mathrm{T} \cup \{\mathrm{t}\}, \mathrm{F} \cup \{(\mathrm{o}, \mathrm{t}), (\mathrm{t}, \mathrm{i})\})$ é fortemente conectada.

Definição 9 (Soundness ou Solidez) Seja $N = (P, T, F)$ uma WF-net com lugar de entrada i e lugar de saída o. N é sólida se e somente se:

1) *Segurança*: $(N, [i])$ é segura,

2) Execução Correta: para qualquer marcação $s \in [N, [i]\rangle$, $o \in s$ implica $s = [o]$,

3) Opção para Completar: para qualquer marcação $s \in [N, [i]\rangle$, $[o] \in [N, s\rangle$, e

4) *Ausência de tarefas mortas*: $(N, [i])$ não contém transições mortas.

A Figura 2.4 é uma rede sólida. Solidez pode ser verificada usando técnicas de análise tradicionais de redes de Petri. Um exemplo de ferramenta adaptada para análise de WF-net é a Woflan (VERBEEK; BASTEN; AALST, 2001).

Definição 10 (*Log* de processo completo) Seja $N = (P, T, F)$ uma WF-net sólida (ver definição 9), ou seja $N \in W$. W é um *log* de processo de N se e somente se $W \in P(T^*)$ e cada trace $\sigma \in W$ é uma sequência de disparo de N iniciando no stado $[i]$ e terminando no estado $[o]$. Ou seja, $(N, [i])[\sigma\rangle(N, [o])$. W é um *log* de processo completo de N se e somente se:

1) para qualquer *log* de processo W' de N: $>_{w'} \subseteq >_w$, e

2) para qualquer $t \in T$ existe um $\sigma \in W$ tal que $t \in \sigma$.

A definição de *log* completo (*completeness*) dada na definição 10 pode parecer arbitrária, mas não é. Devemos notar que seria irrealista pensar que todas as seqüências de disparo possíveis estão presentes no *log*. Primeiro, porque o número de possíveis seqüências podem ser infinitas (no caso de laços). Segundo, processos normalmente têm um número exponencial de estados e, portanto, o número de seqüências de disparo possíveis pode ser muito grande. Finalmente, mesmo que não haja paralelismo e laços, mas apenas N escolhas binárias, o número de possíveis seqüências pode ser 2^N. Portanto, é necessária uma definição mais fraca de integralidade. Se não houver ocorrências de paralelismo e de laços, mas apenas N escolhas binárias, o número de casos pode ser tão pequeno quanto 2, usando a definição dada de integralidade. Claro que, para um N grande, é improvável que todas as escolhas sejam observadas em apenas dois casos, mas ela ainda indica que essa exigência é consideravelmente menos exigente do que observar todas as sequências possíveis. O mesmo vale para processos com laços e paralelismo. Se um processo possui N fragmentos sequenciais onde cada um expõe paralelismo, o número de casos necessários para observar todas as combinações possíveis é exponencial em número de fragmentos.

Definição 11 (SWF-net) Uma rede WF-net $N = (P, T, F)$ é uma rede de *workflow* estruturada (SWF-net), se e somente se:

1) Para cada $p \in P$ e $t \in T$ com $(p, t) \in F$: $|p\bullet| > 1$ implica $|\bullet t| = 1$.

2) Para cada $p \in P$ e $t \in T$ com $(p, t) \in F$: $|\bullet t| > 1$ implica $|\bullet p| = 1$.

3) Não existem lugares implícitos.

A primeira vista, os três requisitos na definição acima parecem ser restritivos. De um ponto de vista prático, este não é o caso. Primeiro, a rede estrutural de *workflow* permite todas as construções de rotas encontradas na prática. Sequências, paralelismo, rotas condicionais e iterativas, são possíveis e os blocos de construção básicos (AND-split, AND-join, OR-split e OR-join) são também suportados. Em segundo lugar, redes de *workflow* que não são estruturadas são tipicamente difíceis de ser compreendidas e deveriam ser evitadas. Em terceiro lugar, muitos sistemas de gerenciamento de *workflow* somente permitem processos que correspondem a redes estruturadas. A última observação pode ser explicada pelo fato destes sistemas usarem na maioria das vezes linguagens de modelagem que separam os blocos de construção para OR-split e AND-join.

2.3 Aspectos da Mineração de Processos

Mineração de processos levanta um número de questões científicas interessantes. Como indicado na seção anterior, algumas destas questões já foram tratadas por pesquisadores enquanto outras requerem ainda mais pesquisa. Nesta seção, serão revisados os principais aspectos da área de mineração de processos. Estes aspectos precisam ser considerados por todos os algoritmos de mineração para que os mesmos possam extrair modelos de processo corretos a partir do *log*.

2.3.1 Minerando Tarefas Ocultas

Uma das suposições básicas da mineração de processos é que cada evento (uma ocorrência de uma tarefa para um caso específico) é registrado no *log*. Claramente, não é possível localizar uma informação sobre uma tarefa se esta não estiver registrada no *log*. Porém, dada uma linguagem específica, é possível perceber que há uma tarefa chamada tarefa escondida. Considere por exemplo, que na Tabela 2.1 os eventos que referenciam a tarefa A são removidos. Apesar do log não revelar a tarefa A, é claro que ali existe uma estrutura AND-split se for assumido que as tarefas B e C possuem caracteristicas de execução em paralelo. Similarmente, podemos detectar que existe uma tarefa AND-join se forem removidos do log todos os eventos referenciando a tarefa D. Suponhamos que ambas tarefas A e D são removidas da Tabela 2.1. Neste cenário, ainda é possível automaticamente construir um modelo de processo similar a Figura 2.3 (ver a Figura 2.5). Porém, para processos mais complexos, é mais dificil adicionar estas tarefas escondidas, e este portanto coloca um problema interessante relacionado a questões como comportamento observável e ramificações (GLABBEEK; WEIJLAND, 1996).

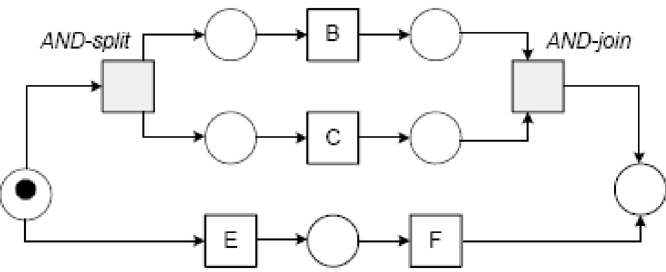


Figura 2.5: Um modelo de processo com duas tarefas escondidas (AALST; WEIJTERS, 2004).

2.3.2 Minerando Tarefas Duplicadas

O problema de tarefas duplicadas refere-se à situação em que pode-se ter um modelo de processo (uma rede de Petri) com dois nós referenciando a mesma tarefa. Suponhamos que na Tabela 2.1 e na Figura 2.3 a tarefa E fosse renomeada para B (ver Figura 2.6). Claramente, o *log* modificado pode ser o resultado de um modelo de processo modificado. Neste caso, torna-se muito difícil automaticamente construir um modelo a partir da Tabela 2.1 com a tarefa E renomeada para B porque não é possível distinguir o B no caso 5 dos B's em outros casos. Note que a presença de tarefas duplicadas está relacionada a tarefas escondidas. Muitos processos com tarefas escondidas mas com nenhuma tarefa duplicada podem ser transformados em processos equivalentes com tarefas duplicadas e sem nenhuma tarefa escondida.

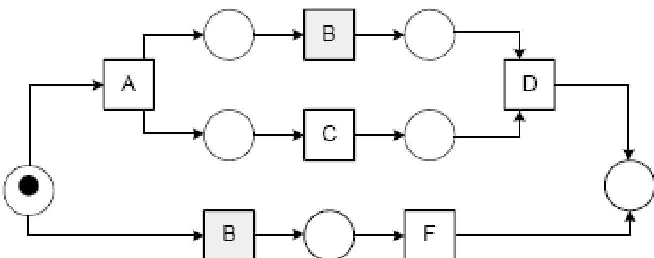


Figura 2.6: Um modelo de processo com tarefas duplicadas (AALST; WEIJTERS, 2004).

2.3.3 Construções Sem-Escolha-Livre

Redes de Petri de escolha-livre (*Free-choice* Petri) são redes onde não existem duas transições consumindo a partir do mesmo local de entrada, mas sim onde uma tem um local de entrada o qual não é local de entrada de outra (DESEL; ESPARZA, 1995). Isto exclui a possibilidade de escolhas de junções e sincronização. Redes de petri de escolha-livre são classes bem conhecidas e amplamente usadas deste tipo de rede. Porém, muitos processos não podem ser expressos em termos de redes de escolha-livre. Infelizmente, a maioria das técnicas de mineração (também aquelas que não usam redes de Petri) assumem modelos de processo correspondendo à classe redes de livre-escolha. Construções de redes sem-livre-escolha são difíceis de modelar pois elas representam "escolhas controladas". Ou seja, a escolha entre duas tarefas não é determinada dentro de um nó no modelo de processo, mas pode depender das escolhas feitas por outras partes do processo. Claramente, este comportamente externo é difícil de minerar e pode requerer muitas observações.

A Figura 2.3 é uma rede de livre-escolha pois a sincronização (tarefa D) é separada da escolha entre A e E. A Figura 2.7 mostra uma construção sem-livre-escolha. Após executar a tarefa C existe uma escolha entre a tarefa D e E. Porém, a escolha entre D e E é "controlada" pela escolha anterior entre A e B. Note que as tarefas D e E estão envolvidas na escolha mas também sincronizam dois fluxos. Claramente este tipo de construção é difícil de minerar pois a escolha não é local e o algoritmo de mineração precisa "relembrar" eventos passados.

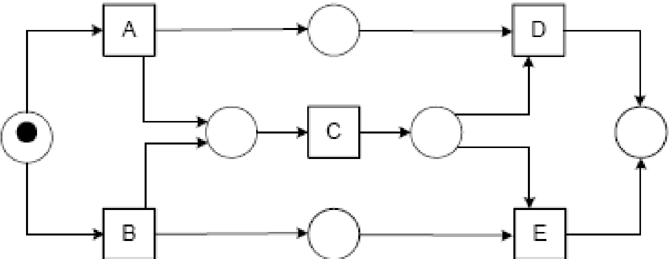


Figura 2.7: Um modelo de processo com construções sem-livre-escolha (AALST; WEIJTERS, 2004).

2.3.4 Minerando Laços

Em um processo de negócio, pode ser possível executar a mesma tarefa várias vezes. Se isto acontece, tipicamente este evento refere-se a uma iteração no modelo correspondente. A Figura 2.8 mostra um exemplo de modelo com este tipo de situação. Após executar a tarefa B, a tarefa C pode ser executada arbitrariamente várias vezes, onde eventos possíveis são BD, BCD, BCCD, BCCCD, etc. Iterações como as relatadas envolvendo a tarefa C são fáceis de serem descobertas. Porém, uma iteração pode também ser usada para voltar para qualquer tarefa anterior durante o fluxo do processo. Para processos mais complexos, minerar iterações está longe de ser trivial pois existem múltiplas ocorrências da mesma tarefa em um dado caso. Algumas técnicas numeram cada ocorrência destas estruturas. Por exemplo, a sequência de execução BCCCD pode ser representada por B1C1C2C3D1. Estas ocorrências são então mapeadas para uma tarefa simples.

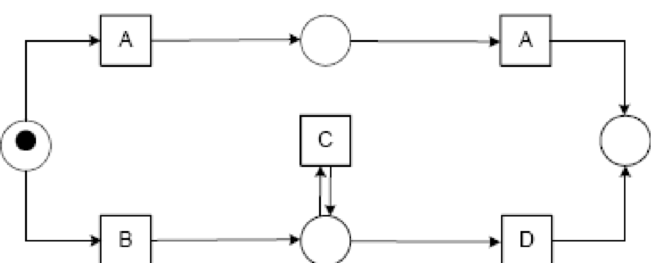


Figura 2.8: Um modelo de processo com laços (AALST; WEIJTERS, 2004).

2.3.5 Tempo de Uso

A Tabela 2.1 mostra a informação mínima necessária para conduzir alguma forma de mineração de processos. Como já foi dito, cada linha corresponde a uma evento gerado pela execução de uma tarefa de um caso específico. Em muitos casos, o *log* também contém informações temporais (cada evento contendo um *timestamp*). Para modelar a duração da execução de uma tarefa, pode-se registrar eventos de início e fim. Comparando as diferenças entre o *timestamp* dos eventos de início e fim, é possível descobrir o tempo de execução de um processo e suas atividades. A informação de tempo pode ser usada de duas formas: (*i*) adicionar a informação de tempo para o modelo de processo e (*ii*) melhorar a qualidade do modelo de processo descoberto. É relativamente simples incrementar o modelo de processo com a informação de tempo. Uma abordagem é primeiramente minerar o processo enquanto ignora-se o *timestamp* e então em seguida realizar um "*replay*" do *log* no modelo de processo.

2.3.6 Minerando Diferentes Perspectivas do Processo

Ao minerar um *log* de eventos de execução de processos, diversas perspectivas do processo podem ser descobertas. A principal perspectiva é a perspectiva de fluxo de controle. A principal característica desta perspectiva é a ordem das tarefas, que ao final da mineração formarão a estrutura do processo. Esta perspectiva pode também ser estendida para incluir outras informações importantes, como tempo (*timestamp*), por exemplo.

Em adição a perspectiva de fluxo de controle, podemos relacionar outras perspectivas: perspectiva organizacional, perspectiva da informação e perspectiva da aplicação. Na perspectiva da organização, a estrutura organizacional e sua população são especificadas, descrevendo relações entre papéis (classes de recursos baseado nos aspectos funcionais da empresa) e grupos (classes de recursos baseado nos aspectos organizacionais), além de outras informações que clarificam detalhes da organização, como responsabilidades e disponibilidade dos recursos. Recursos podem variar de humanos a dispositivos computacionais, sendo estes atribuídos a grupos ou papéis. A perspectiva da informação está ligada aos dados de controle e produção do processo. Dados de controle são aqueles introduzidos pontualmente para propósitos gerenciais, como por exemplo variáveis introduzidas para alterar uma rota do processo. Dados de produção são objetos de informação como documentos, formulários e tabelas, cuja existência não depende do gerenciamento do processo. A perspectiva de aplicação lida com as aplicações que irão ser usadas para executar as tarefas do processo, como editores de texto e aplicações ERP.

Neste trabalho, o algoritmo de mineração incremental proposto utiliza a perspectiva de fluxo de controle do processo e também alguns conceitos da perspectiva organizacional, onde o objetivo final é obter a estrutura do processo de negócio e os participantes (recursos que executam cada atividade do processo) . Desta forma, as demais perspectivas serão pouco abordadas e apenas citadas ao decorrer deste capítulo.

2.3.7 Lidando com Ruídos

A maioria dos algoritmos de mineração assume que a informação do *log* está correta. Embora isto seja uma afirmação coerente em muitas situações, o *log* pode conter ruídos (ex: informações registradas incorretamente). Por exemplo, pode ser possível em determinados casos que um evento não seja registrado, ou ainda em alguns casos ele seja registro fora de ordem. O algoritmo de mineração precisa ser robusto com respeito a este e outros tipos de ruídos. Além disso, este tipo de relação de causalidade pode não ser baseada em uma simples observação. De fato, poderia-se argumentar que o algoritmo de mineração deveria distinguir exceções (situações excepcionais com pouca ocorrência) de um "fluxo normal". Ao considerar uma informação como ruído, é necessário muitas vezes determinar um valor limite de corte (*threshold*) deste comportamento excepcional ou incorretamente registrado. Com este limite de corte, é possível definir um valor mínimo aceitável para os eventos serem registrados como relações de causalidade, sendo que qualquer valor abaixo seja automaticamente considerado como ruído, devendo ser descartado do resultado da mineração.

2.3.8 Lidando com Incompleteza

Um *log* é considerado imcompleto se ele não contém as informações suficientes para derivar um processo. Considerando novamente a Tabela 2.1 e o modelo de processo derivado apresentado na Figura 2.3. Suponhamos que a Figura 2.3 é uma

representação correta de um processo real mas a rota representada pelo caso 5 acontece raramente. Quando minera-se poucos casos, pode ocorrer que somente casos similares aos casos 1, 2, 3 e 4 sejam registrados. Como resultado, o modelo de processo descoberto não seria correto pois as tarefas E e F não estariam presentes no modelo. Este exemplo pode ser trivial, porém, para processos de organizações reais, pode-se chegar facilmente a milhões de possíveis caminhos quando for permitido paralelismo, rotas condicionais e iterativas. Considerando, por exemplo, a Figura 2.9, podemos notar que neste processo não existem escolhas, todas as tarefas são executadas somente uma vez. Porém, a tarefa B e a sequência de nove tarefas C1, C2, ..., C9 são executadas em paralelo. Como resultado, existem dez possíveis rotas. Mesmo não havendo escolhas a serem feitas, pelo menos, dez casos serão necessários para derivar o modelo de processo apresentado na Figura 2.9. De fato, observações onde B é executado após a sequência de nove tarefas C1, C2, ..., C9 pode ser altamente inprovável e talvez milhares de casos logados sejam necessários para descobrir o modelo correto. Se alterarmos o processo na Figura 2.9 para que as tarefas C1, C2, ..., C9 sejam executadas em paralelo entre si, então haverá 10! = 3628800 possíveis rotas a seguir. Estas heurísticas são tipicamente baseadas no princípio de Occam’s Razor (THORBURN, 1915), que diz "Quando você tem duas teorias competindo que fazem exatamente as mesmas predições, a mais simples tende a ser a melhor".

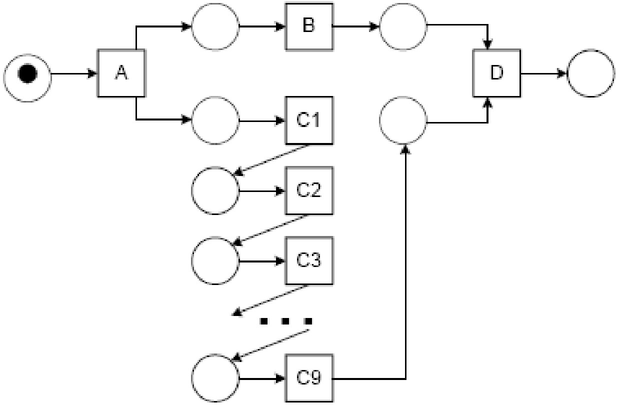


Figura 2.9: Um modelo de processo onde é dicífil identificar a sincronização (AALST; WEIJTERS, 2004).

2.4 Minerando Decisões

Uma análise apropriada dos *logs* de um sistema de informação orientado a processo pode revelar conhecimento importante e ajudar as organizações a melhorar a qualidade de seus serviços. Utilizando algoritmos de mineração de processos, podemos analisar como os atributos dos dados influenciam as escolhas feitas em um processo baseado nas suas execuções passadas. Uma das abordagens relacionadas a mineração de processos que trata da análise dos atributos dos dados é chamda mineração de decisões (ROZINAT; AALST, 2006a). Também referenciada como análise de pontos de decisão, ela objetiva a detecção de dependências entre os dados que afetam as rotas de uma instância de processo de negócio.

A grande maior das técnicas de mineração de processos não aproveitam todos os dados presentes no *log* e somente fazem uso de poucos atributos para construir um modelo de processo que reflete as dependencias causais que foram observadas entre as atividades. Para análise dos demais dados do *log*, assim como os atributos dos dados

consumidos pelo processo, algoritmos de aprendizagem de máquina tem tornado-se uma escolha ampalhamente utilizada para extração de conhecimento (WITTEN; FRANK, 2011). Nesta seção é explorado o potencial destas técnicas para mineração da perspectiva de dados dos processos de negócio. O fundamentado conceito de árvores de decisão (BREIMAN; FRIEDMAN; OLSHEN; STONE, 1984) é utilizado para contemplar a análise de pontos de decisão a fim de determinar quais propriedades de um caso podem lidar com a tomada de decisão em uma rota do processo.

Iniciando a partir do modelo de processo descoberto, podemos tentar melhorar o modelo integrando padrões que podem ser observados a partir de modificações realizadas sobre os dados, como por exemplo, cada escolha no modelo pode ser analisada e, se possível, relacionada a propriedades de casos e atividades individuais. Como é possível visualizar na Figura 2.10, o *log* de eventos pode conter informação sobre as pessoas que executaram as atividades (coluna *originator*), o tempo de execução destas atividades (coluna *timestamp*), e os dados envolvidos (coluna *data*). Porém, como já descrito anteriormente, abordagens mais clássicas de mineração de processos como o algoritmo *α Miner* (AALST, 2004c), tendem a utilizar somente as primeiras duas colunas para obter um modelo de processo. A rede de Petri mostrada na parte inferior central da figura ilustra o resultado da aplicação do algoritmo *α Miner* sobre um *log* de processo. O objetivo da mineração de decisões é localizar regras de decisão que explicam sobre quais circunstâncias a atividade de *Tempo Excedido* foi selecionada ao invés da atividade *Processar Resposta* (ver regra de decisão destacada na parte inferior direita da Figura 2.10). Esta regra de decisão indica que a vistoria de documentos enviados por carta para o participante não são muitas vezes retornadas a tempo, assim como documentos enviados pouco antes do Natal. Consequentemente, uma extensão do limite de tempo nestes casos poderia ajudar a reduzir despesas com postagem.

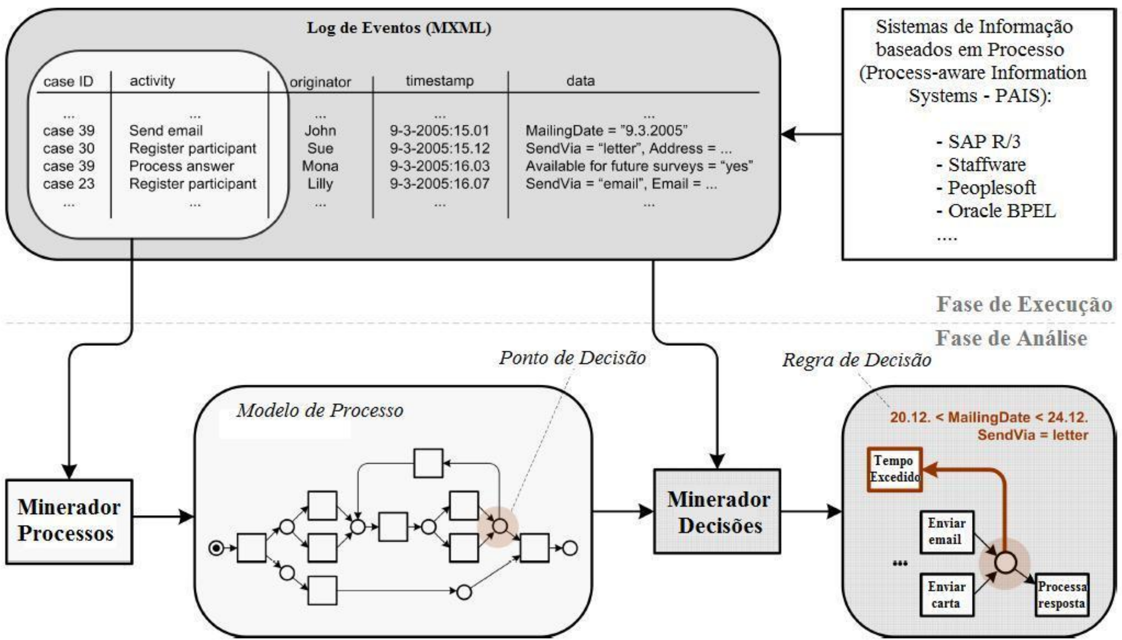


Figura 2.10: A abordagem de mineração de decisões (ROZINAT; AALST, 2006a)

Claramente, a aplicação das técnicas existentes de mineração de dados para detectar padrões frequentes no contexto de processos de negócio tem um potencial de ganho de informação sobre o processo, e permitem explicitar um conhecimento oculto. Além disso, o tipo de dependência que pode ser descoberta é muito ampla.

Além dos atributos de dados e informações sobre recursos e *timestamps*, informações mais quantitativas como indicadores chave de performance (ex: tempo de espera) e também qualitativas (propriedades desejáveis ou indesejáveis) podem ser incluídas na análise se as mesmas estiverem disponíveis.

O primeiro passo para a mineração de decisões compreende na aplicação de um algoritmo de mineração de processos para obter um modelo de processo. A Figura 2.11 (a) mostra um *log* de eventos de uma forma esquemática. O *log* foi agrupado por instâncias (de acordo com a coluna *ID Casos*) e todas as informações, exceto as atividades executadas, podem ser descartadas. Baseado neste *log*, o algoritmo α Miner, ou qualquer outro algoritmo de mineração de estruturas de processos, induz o modelo de processo mostrado na Figura 2.11 (b), que por sua vez serve como entrada para a fase de mineração de decisão. O processo de exemplo usado neste capítulo esboça um pedido de pagamento. Primeiramente, a atividade A é registrada. Nela, os dados relacionados ao pedido são registrados e então em seguida é realizada uma verificação completa ou verificação de políticas (atividades B ou C). Em seguida, o pedido é avaliado (atividade D), e então, como consequência, ele pode ser rejeitado (F) ou aprovado e pago (G e E). Finalmente, o caso é arquivado e fechado (H).

Entrando no segundo estágio da abordagem, o modelo de processo minerado é deixado de lado para darmos uma olhada novamente no *log* de eventos, agora também levando em consideração os atributos de dados. A Figura 2.12 mostra um trecho do *log* no formato MXML (o metamodelo MXML é explicado na seção 2.6.2). Podemos observar que somente as atividades A (*Registrar Pedido)* e D (*Avaliar Pedido)* possuem itens de dados associados.

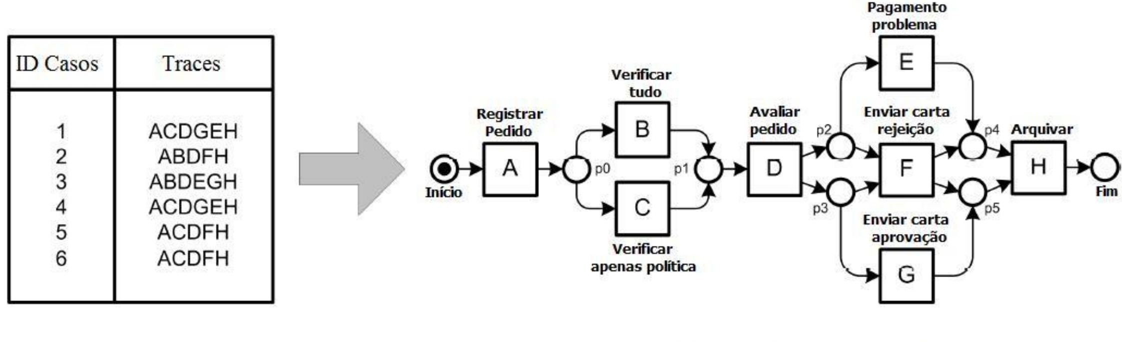


Figura 2.11: Fase de Mineração de Processos (ROZINAT; AALST, 2006a)

Para analisar as escolhas (rotas) no processo de negócio, primeiramente é necessário identificar as partes do modelo onde o processo será dividido em ramos alternativos, também chamados de controle de fluxo ou pontos de decisão. O objetivo é encontrar regras para seguir um caminho ou outro baseado nos atributos de dados associados aos casos do processo no *log* de eventos.

Em termos de uma rede de Petri, um ponto de decisão corresponde a um lugar com múltiplos arcos deixando um nó. Uma vez que um *token* pode ser consumido por apenas um das transições conectadas a esses arcos, caminhos alternativos podem ser seguidos durante a execução de uma instância de processo. O modelo de processo na Figura 2.11 exibe três pontos de decisão: p0 (se existe um *token*, ambos B ou C podem ser realizados), p2 (se existe um *token*, ambos E ou F podem ser executados) e p3 (se existe um *token*, ambos F ou G podem ser efetuados).

Tendo identificado um ponto de decisão em um processo de negócio, é necessário saber se esta decisão pode ser influenciada por casos de dados. Ou seja, se casos com certas propriedades tipicamente seguem uma rota específica. Técnicas de aprendizagem de máquina (MITCHELL, 1999) podem ser usadas para descobrir padões estruturais nos dados baseados em um conjunto de instâncias de treinamento. Por exemplo, pode haver algumas instâncias de treinamento que representam ou não uma tabela, acompanhada por um número de atributos como *height*, *width,* e *number of legs*. Baseado nestas instâncias de treinamento, um algoritmo de aprendizado de máquina pode "aprender" o conceito tabela para classificar instâncias desconhecidas como sendo uma tabela ou não, baseado nos valores dos atributos. O padrão estrutural inferido para cada problema de classificação é chamado descrição de conceito, e pode ser representado, em termos de regras como uma árvore de decisão (dependendo do algoritmo aplicado).

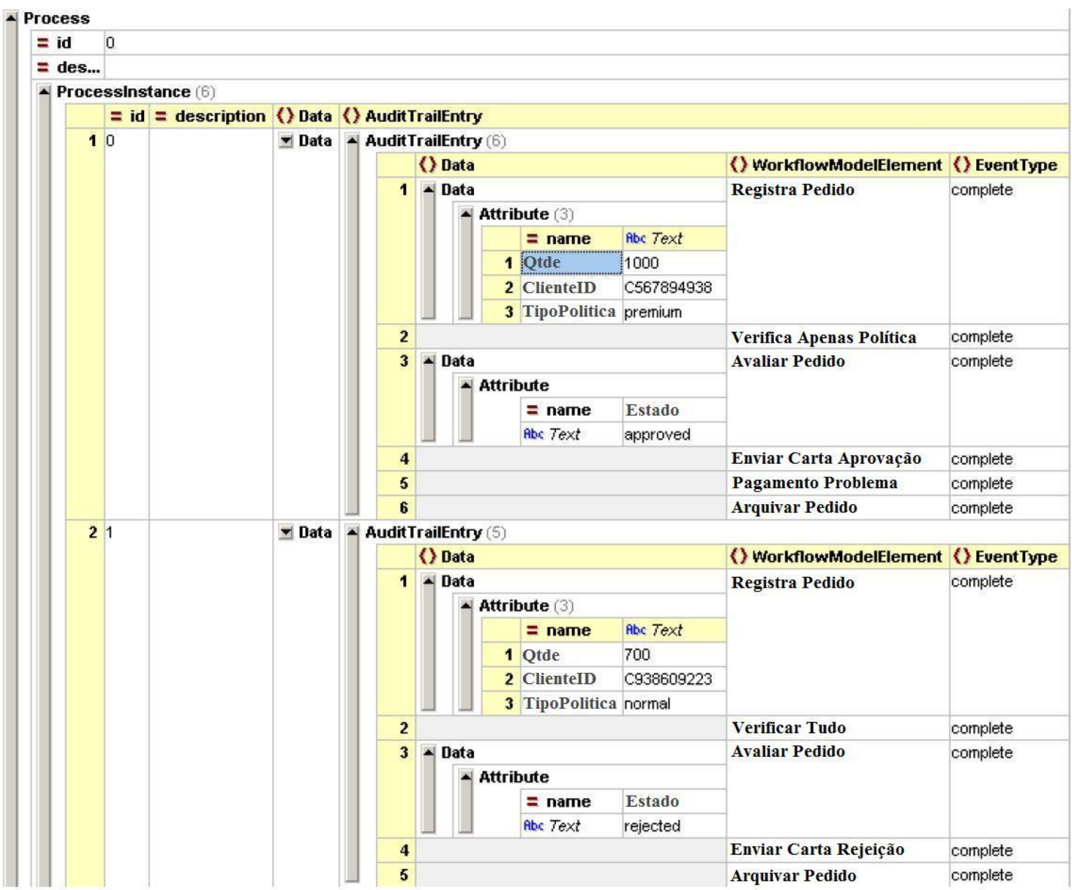
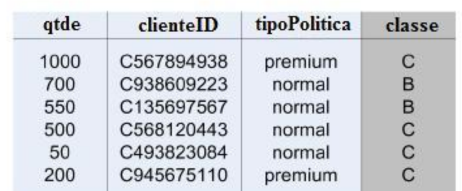


Figura 2.12: Fragmento de exemplo de um log no formato MXML usando a ferramenta XML Spy (ROZINAT; AALST, 2006a)

Embora uma descrição de conceito possa ser usada para predizer a classe de futuras instâncias, o principal benefício é tipicamente o conhecimento adquirido nos atributos de dependência, que são "explicados" pela representação estrutural explícita. Usando análise de pontos de decisão é possível extrair conhecimento sobre as regras de decisão do processo de exemplo, como mostrado na Figura 2.14. Cada um dos três pontos de decisão corresponde a uma das escolhas no processo. Com respeito ao ponto de decisão p0, a verificação estensiva (atividade B) é somente realizada se a quantidade é maior que 500 e o campo *tipoPolitica* é igual a "normal", considerando uma cobertura de verificação mais simples (atividade C) é suficiente se a quantidade é menor ou igual a

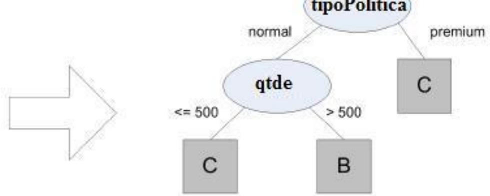
500, ou o campo *tipoPolitica* é igual a "premium". As duas escolhas no ponto de decisão p2 e p3 são ambas guiadas pelo atributo *estado*, o qual é a saída da atividade *Avaliar Pedido* (atividade D). No restante desta seção, será descrito como estas regras podem ser descobertas.

A idéia da aplicação de árvores de decisão é converter cada ponto de decisão em um problema de classificação (WITTEN; FRANK, 2011),(QUINLAN, 1993), (MITCHELL, 1999), onde as classes são as diferentes decisões que podem ser feitas. Como exemplos de treinamento podemos usar as instâncias de processos no *log*, para qual já é conhecido qual caminho alternativo eles terão de seguir com respeito ao ponto de decisão. Os atributos a serem analisados são os atributos dos casos contidos no *log*, e assumimos que todos os atributos que foram escritos no *log* antes da construção da escolha considerada podem ser relevantes para a decisão da escolha da rota no caso do processo. Assim, para pontos de decisão p0, somente os atributos de dados providos pela atividades A são considerados (*qtde*, *clienteID* e *tipoPolitica*), e na Figura 2.13 (a) os valores correspondentes contidos no *log* serão usados para construir um exemplo de treinamento de cada instância de processo (um exemplo de treinamento corresponde a um linha na tabela). A última coluna representa a classe de decisão, que denota a decisão que foi feita pela instância do processo com respeito ao ponto de decisão p0 (se a atividade B ou C foi executada). Similarmente, a Figura 2.13 (c) e (e) representam os exemplos de treinamento para os pontos de decisão p2 e p3, respectivamente. Aqui, um atributo adicional (*estado*) foi incorporado ao conjunto de dados pelo fato do mesmo ser provido pela atividade D, que é executada antes de p2 (E ou F) e p3 (G ou F).



qtde	clienteID	tipoPolitica	classe
1000	C567894938	premium	C
700	C938609223	normal	B
550	C135697567	normal	B
500	C568120443	normal	C
50	C493823084	normal	C
200	C945675110	premium	C

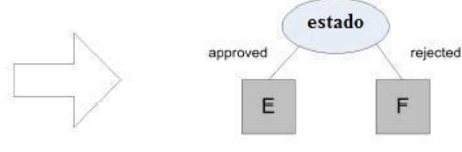
(a) exemplos de treinamento para ponto decisão"p0"



(b) árvore de decisão para o ponto decisão "p0"

qtde	clienteID	tipoPolitica	estado	classe
1000	C567894938	premium	approved	E
700	C938609223	normal	approved	E
550	C135697567	normal	rejected	F
500	C568120443	normal	approved	E
50	C493823084	normal	rejected	F
200	C945675110	premium	rejected	F

(c) exemplos de treinamento para ponto decisão "p2"



(d) árvore de decisão para o ponto decisão "p2"

qtde	clienteID	tipoPolitica	estado	classe
1000	C567894938	premium	approved	G
700	C938609223	normal	approved	G
550	C135697567	normal	rejected	F
500	C568120443	normal	approved	G
50	C493823084	normal	rejected	F
200	C945675110	premium	rejected	F

(e) exemplos de treinamento para ponto decisão "p3"

(f) árvore de decisão para o ponto decisão "p3"

Figura 2.13: Pontos de decisão representados como problemas de classificação (ROZINAT; AALST, 2006a)

Para resolver um problema de classificação, há vários algoritmos disponíveis (WITTEN; FRANK, 2011). ROZINAT et al em seus trabalhos decidiu utilizar o algoritmo de árvores de decisão C4.5 (QUINLAN, 1993), pois se trata de um dos mais populares algoritmos de inferência indutiva e disponibiliza um número grande de extensões que são importantes para aplicações práticas. Os itens (b), (d) e (f) da Figura 2.13 mostram a árvore de decisão que foi derivada a partir dos pontos de decisão p0, p2 e p3, respectivamente. A partir desta árvore, podemos inferir expressões lógicas que formam as regras de decisão apresentadas na Figura 2.14. Se uma instância é localizada em uma das folhas da árvore de decisão, ela cobre todos os predicados no caminho entre a raíz e a folha. Ou seja, eles são conectados por um operador boleano *E*. Por exemplo, a classe B na Figura 2.13 (b) é escolhida se ((*tipoPolitica* = "normal") *E* (*qtde* > 500)).

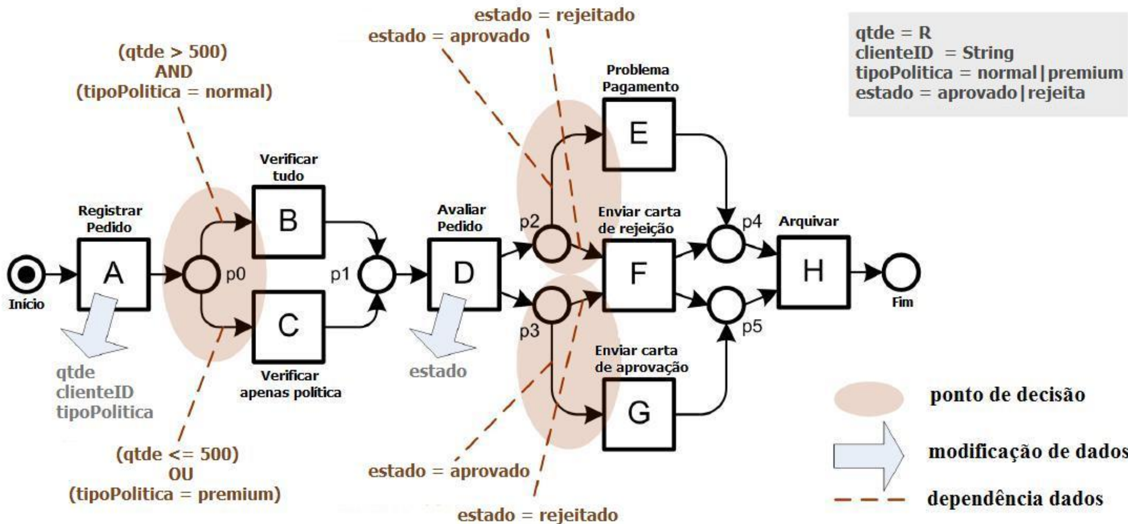


Figura 2.14: Modelo de Processo Melhorado (ROZINAT; AALST, 2006a)

2.5 Métricas de Análise de Qualidade

Mecanismos de análise sempre foram utilizados dentro do contexto de *data mining e machine learning (*MITCHELL, 1999*).* Estes mecanismos são utilizados para avaliar a performance (acurácia) de um modelo minerado. Similarmente, a qualidade de um modelo de processo gerado por um algoritmo de mineração de processos também precisa ser quantificada. Dois tipos de avaliação podem ser feitas no contexto de mineração de processos: i) avaliação utilizando métricas de validação e ii) avaliação utilizando técnicas do domínio de aprendizagem de máquina. O primeiro tipo realiza uma avalição utilizando métricas que analisam algumas propriedades do *log* em relação ao modelo minerado, ou em alguns casos até o modelo original (modelo utilizado para gerar o *log*). O segundo tipo de avaliação considera a utilização da técnica de validação cruzada *k-fold (*GEISSER, 1993*).*

Nesta seção, são introduzidas algumas métricas de validação utilizadas na mineração de processos e também utilizadas para avaliação do algoritmo proposto neste trabalho. Estas técnicas fornecem quatro perspectivas diferentes para análise da qualidade de um modelo: cobertura, precisão, generalização e estrutura (ROZINAT et al, 2007).

2.5.1 Cobertura (*Fitness*).

A métrica Fitness indica quanto do comportamento observado no *log* é capturado (ou coberto) pelo modelo de processo. *Traces* no *log* podem ser associados a caminhos de execução válidos especificados pelo modelo de processo descoberto. Por exemplo, o modelo apresentado na Figura 2.15 (c) reproduz apenas a sequência ABDEI, mas não as outras sequências no *log*. Desta forma, podemos dizer que a cobertura do modelo é baixa.

Uma maneira de medir se um *log* de eventos cobre um modelo de processo é reproduzir o *log* a partir do modelo e então medir os desvios encontrados. A reprodução de cada *trace* no *log* inicia com a adição de uma marca no local inicial do processo (ex: rede de Petri). Assim, as transições que pertencem aos eventos logados no *trace* serão disparados uma após a outra. Enquanto a reprodução ocorre, é necessário contar o número de *tokens* que precisam ser criados artificialmente (transições pertencentes ao evento registrado mas que não foram habilitados e desta forma não podem ser executados com sucesso) e o número de *tokens* que foram perdidos no modelo, o que indica que o processo não foi completado de forma esperada.

2.5.2 Precisão (*Behavioral Appropriateness*).

A métrica Precisão avalia modelos que podem ser excessivamente genéricos. Por exemplo, o modelo apresentado na Figura 2.15 (d) permite a execução das atividades A até I em qualquer ordem, assim como as sequências no *log*. Apesar da cobertura ser boa, a precisão do modelo é baixa. Já o modelo apresentado na Figura 2.15 (b) é considerado um modelo preciso, embora ele permita o trace ACGHDFI (que não está presente no *log*). Em processos onde atividades são executadas em paralelo (como o modelo b) é improvável que todo comportamento possível esteja presente no *log*. Assim, é necessário detectar estas atividades sem a obrigatoriedade de se observar cada relacionamento entre elas. Neste caso, o objetivo é medir a flexibilidade do modelo para detectar comportamento contendo estruturas paralelas (AND-Split/Join) e alternativas (XOR-Split/Join), mesmo elas não sendo utilizadas em execuções reais observadas no *log*.

2.5.3 Generalização.

Avalia modelos que podem ser excessivamente precisos. Por exemplo, o modelo apresentado na Figura 2.15 (e) somente permite a execução das cinco sequências exatas disponíveis no *log*. Ao contrário do modelo da Figura 2.15 (b), o qual também permite o *trace* ACGHDFI, nenhuma generalização foi realizada no modelo da Figura 2.15 (e).

2.5.4 Estrutura (Structural Appropriateness).

Nesta perspectiva, a estrutura sintática e semântica do processo de negócio é avaliada, a qual é determinada pelo vocabulário disponível na linguagem de modelagem (ex: elementos semânticos como AND e XOR). Existem diversas maneiras sintáticas de se expressar o mesmo comportamento, e algumas representações podem ser mais ou menos preferíveis por um projetista de processo. A métrica estrutural avalia se um modelo de processo de negócio é o mais simples possível, sem perder a cobertura e precisão. Por exemplo, a cobertura e precisão do modelo da Figura 2.15 (e) são bons, mas ele contém muitas tarefas duplicadas, o qual o torna de difícil compreensão.

Algumas estruturas podem pesar negativamente no valor da métrica estrutural, dentre elas podemos citar a ocorrência de tarefas duplicadas e tarefas ocultas no modelo

de processo avaliado. Estas estruturas podem “inflar” o modelo de processo, prejudicando assim sua compreensão. Desta forma, para obtermos modelos de processo com boa estrutura, é necessário, quando possível, evitar este tipo de construção.

No capítulo 5 são introduzidos experimentos que utilizam estas perspectivas de qualidade. O objetivo destes experimentos é medir a qualidade dos modelos minerados pelo algoritmo proposto neste trabalho e compará-los com os modelos gerados pelos principais algoritmos de mineração de processos.

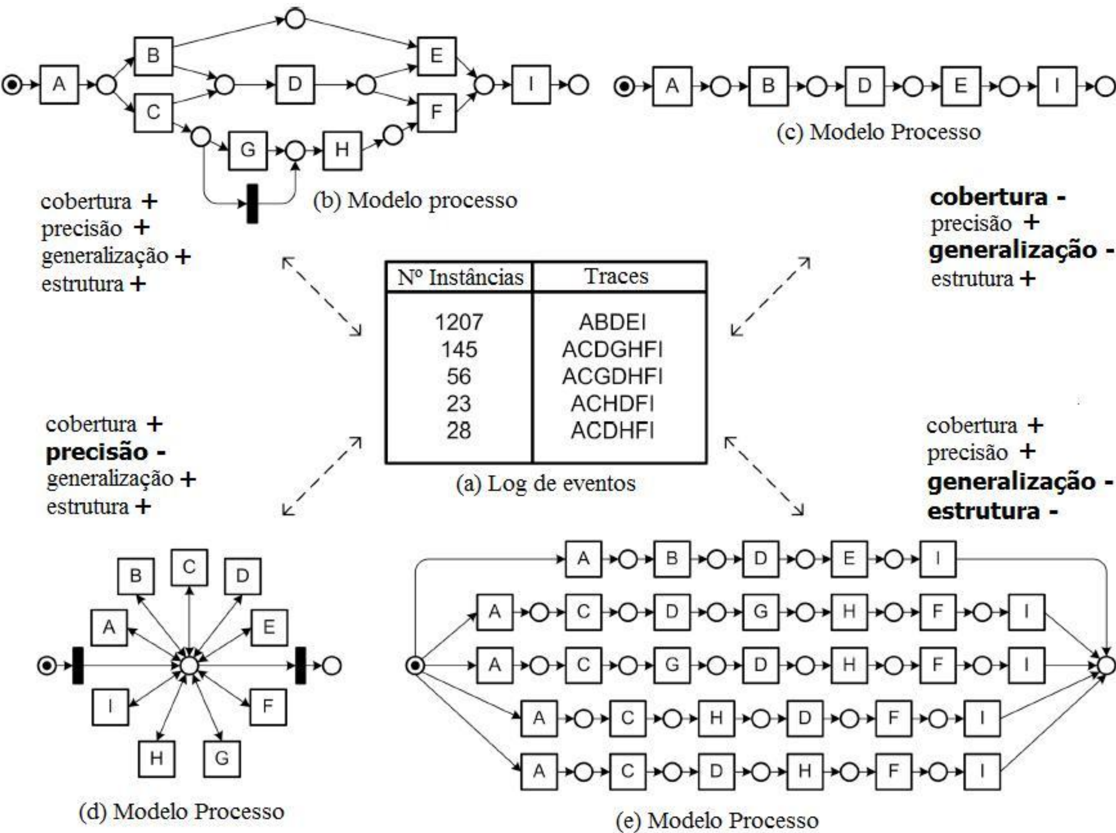


Figura 2.15: Avaliação do modelo de processo baseado em diferentes dimensões (ROZINAT et al, 2007)

2.6 Ferramenta ProM

O objetivo da ferramenta ProM (DONGEN; MEDEIROS; VERBEEK; WEIJTERS; AALST, 2005b) é fornecer um ambiente integrado para mineração de processos a partir do *log* de execução de *workflow*. A ferramenta utiliza *plug-ins* que permiten a flexibilidade durante o processo de mineração. Com isto, não existe restrição quanto a utilização de diversos formatos de entrada e saída do algoritmo. A ferramenta ProM é aberta suficientemente para permitir fácil reutilização de código durante a implementação de novas idéias e algoritmos de mineração de processos. Assim, através deste ambiente "plugável" e estendível, a ferramenta ProM permite minerar diferentes perspectivas do *log*, como a perspectiva fluxo de dados, perspectiva organizacional, perspectiva da informação e perspectiva da aplicação (ver Seção 2.3.6).

2.6.1 Arquitetura

Na Figura 2.16 é possível ter uma visão das principais funcionalidades da ferramenta ProM. Ela mostra as relações entre o *framework*, o formato do *log* de processo, e os *plug-ins*. Como podemos ver, a ferramenta pode ler arquivos no formato

XML através do componente *Filtro Log*. Este componente é capaz de lidar com grande volume de dados e filtrá-los antes da mineração iniciar. Através do *plug-in Importação*, uma grande variedade de modelos podem ser carregados variando de uma rede de Petri até fórmulas em lógica temporal (LTL). O *plug-in Mineração* realiza a mineração propriamente dita e o resultado é armazenado como um *Frame de Resultado*. Estes frames podem ser usados para visualização, como por exemplo uma rede de Petri, um modelo EPC, uma rede social, ou consequentemente uma análise e conversão do resultado. O *plug-in Análise* pega o resultado da mineração e aplica este conteúdo a alguma técnica de análise de qualidade do modelo. O *plug-in Conversão* transforma os dados do resultado da mineração em outro formato, como por exemplo de Rede de Petri para EPC ou vice-versa.

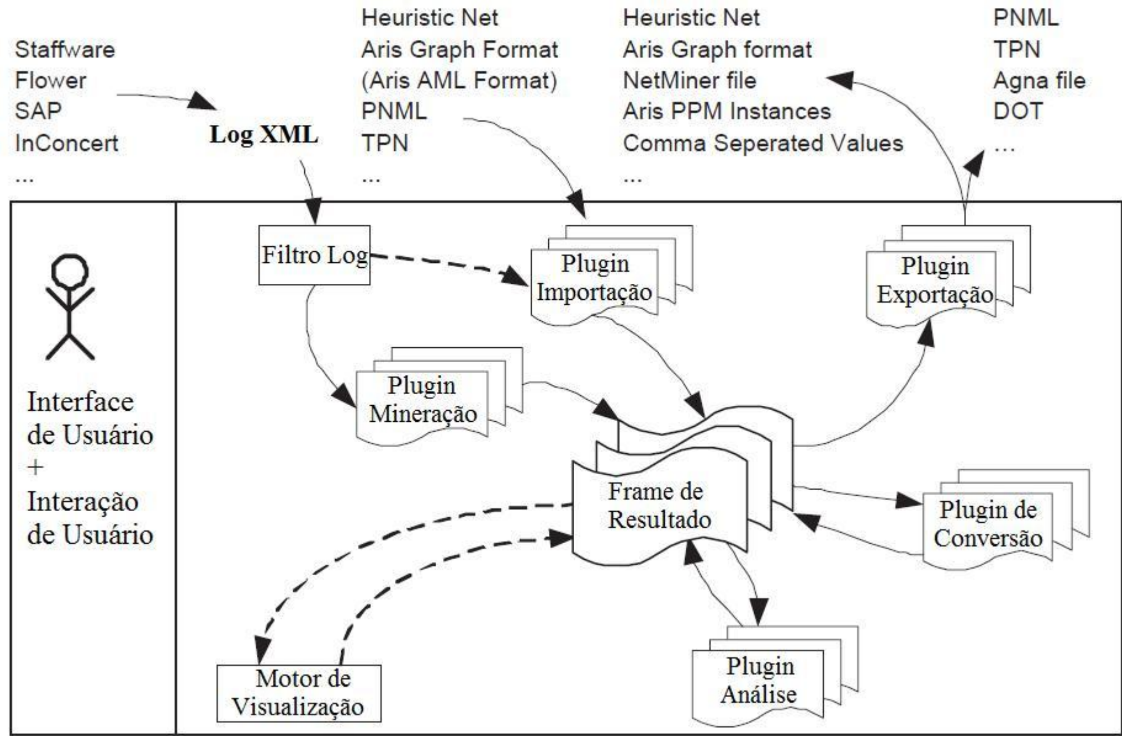


Figura 2.16: Visão geral do framework ProM (DONGEN et al., 2005b).

2.6.2 Formato do *Log* de Processos

A ferramenta ProM trabalha com um formato padronizado de *log* de processos, chamado MXML (DONGEN; AALST, 2005a). Na Figura 2.17 podemos visualizar o esquema XML que especifica o formato deste metamodelo de *log*. O elemento *WorkflowLog* é o elemento raíz do metamodelo. Este elemento contém os sub-elementos opcionais *Data* e *Source*, e os demais elementos estão relacionados aos dados de execução do processo. Um elemento *Data* permite armazenar dados textuais arbitrários, utilizados pelo processo , e contém uma lista de elementos do tipo *Attribute*. Um elemento *Source* pode ser usado para armazenar informações sobre o sistema de informação que gerou o *log*. Um elemento *Process* refere-se a um processo especifico em um sistema de informação. Pelo fato da maioria dos sistemas de informações tipicamente controlarem diversos processos, multiplos elementos *Process* podem existir em um arquivo de *log* MXML. Um elemento *ProcessInstance* representa uma instância do processo, ou seja um caso propriamente dito. Um elemento *AuditTrailEntry* faz referência a uma atividade executada, a qual tem seu nome armazenado através do

elemento *WorkflowModelElement*. O elemento *AuditTrailEntry* ainda contém o tipo de evento relacionado a atividade executada (*Eventtype*), um *timestamp* (*Timestamp*) contendo o tempo de execução da atividade, e o participante que executou a atividade (*Originator*).

Assim como o elemento *WorkflowLog*, o elemento *AuditTrailEntry* também contém um elemento *Data*, o qual armazena os dados (ex: parâmetros de entrada e saída) relacionados a execução de uma atividade. Como visto na seção 2.4, este elemento também pode ser utilizado para realizar a mineração dos pontos de decisão do processo, revelando regras de decisão utilizados nos controles de fluxo do processo.

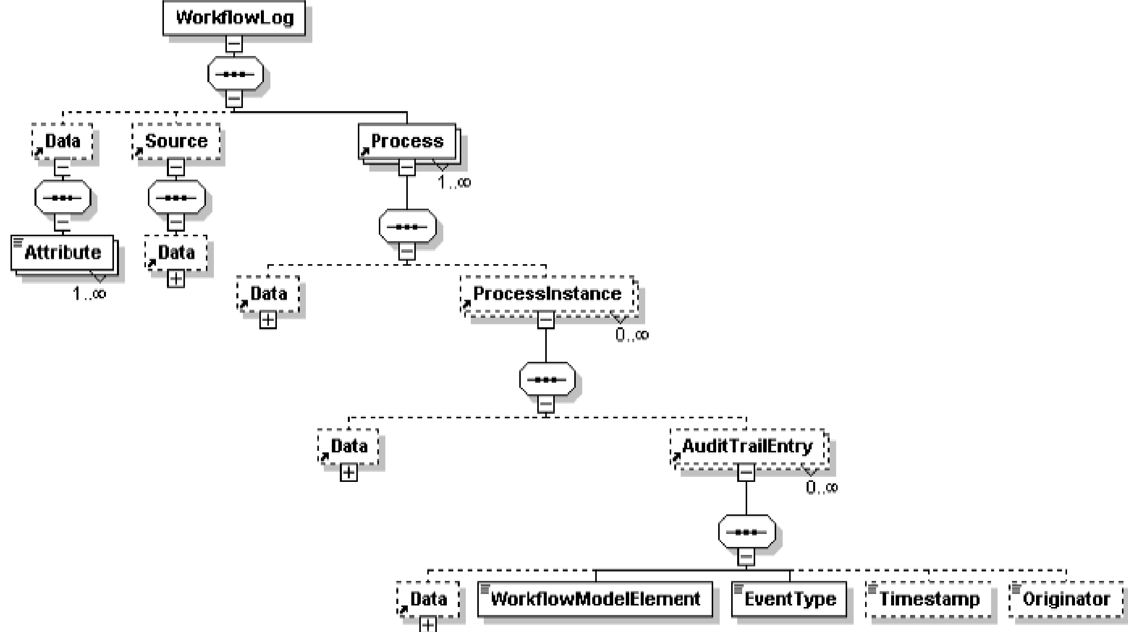


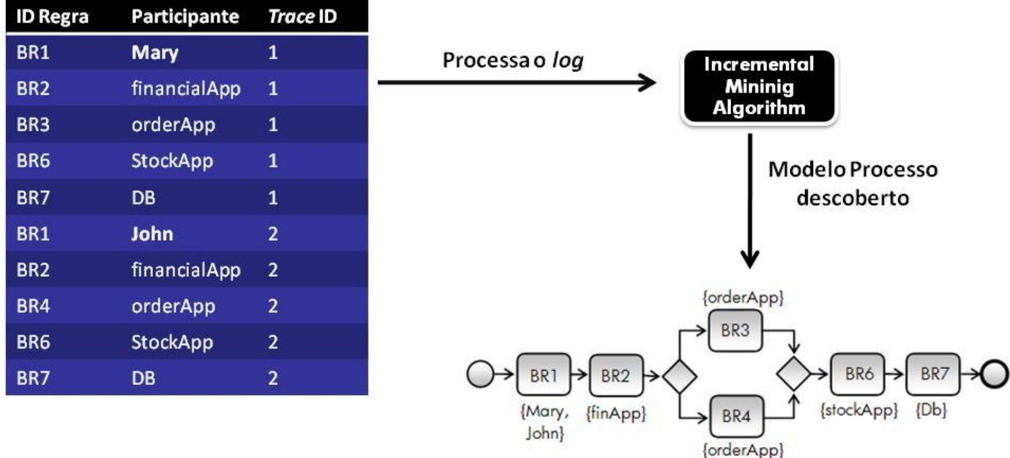
Figura 2.17: Formato do log de processo MXML (DONGEN et al., 2005a).

3 ALGORITMOS INCREMENTAIS DE MINERAÇÃO

Algoritmos incrementais de mineração não são exclusividade da área de mineração de processos. A alguns anos este tema já vem sendo abordado pela mineração de dados e aprendizagem de máquina (HAN et al, 2000), (HERNÁNDES-LEÓN, 2008). Estes algoritmos visam a descoberta de conhecido dinâmico ao longo do tempo. Ou seja, a medida que novos casos são gerados, o conhecimento adquirido até o momento deve ser atualizado. Neste capítulo, são apresentados os principais algoritmos de mineração incremental de processos. Estes algoritmos possuem a capacidade de realizar a mineração de logs de execução de processos de forma incremental, possibilitando (i) complementar um modelo de processo parcial previamente descoberto e/ou (ii) atualizar um modelo de processo existente, a fim de mantê-lo sincronizado com o processo em execução.

Para a compreensão do funcionamento da mineração incremental, podemos observar o *log* parcial apresentado na Figura 3.1 e Figura 3.2. Estes *logs* foram gerados a partir da execução de um sistema de informação. A partir da disponibilidade do *log* de execução, podemos realizar a mineração para a descoberta de um modelo de processo parcial. A Figura 3.1 apresenta o resultado da mineração de um *log* inicial parcial. Neste ponto, algoritmos incrementais e não incrementais não diferem no resultado, pois não há um modelo de processo previamente descoberto. Quando novas execuções do sistema ocorrem, novos registros são adicionados ao *log* representando neste caso novos *traces* de execução, conforme apresentado na Figura 3.2. Neste momento, o algoritmo incremental é executado novamente, processando apenas as novas entradas no *log* e realizando consequentemente a atualização de modelo de processo existente o qual podemos assumir ser mais completo.

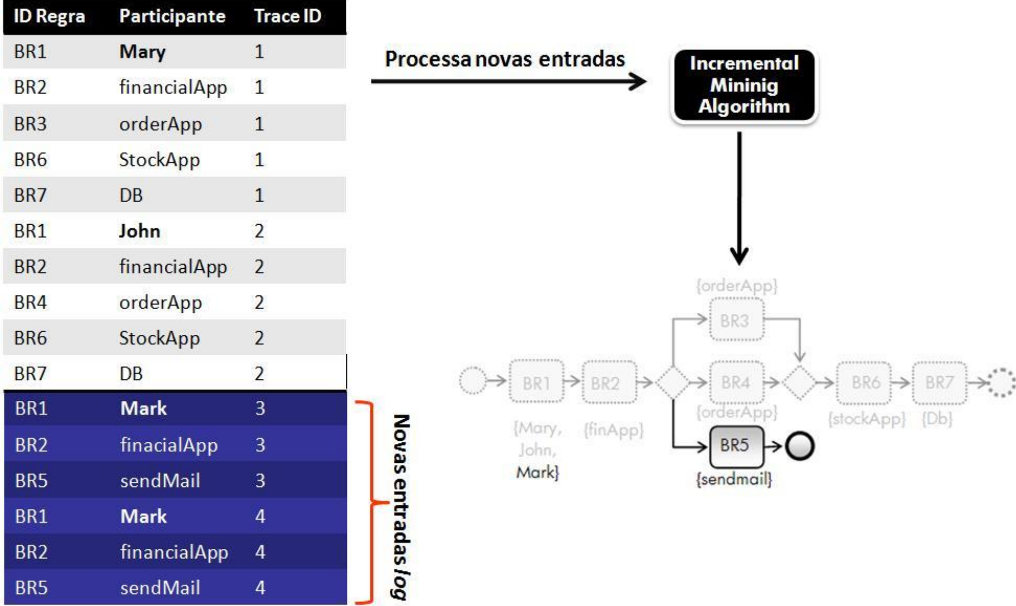
O cenário descrito acima pode ser classificado como um dos mais simples encontrados durante a mineração incremental. Porém, é possível também encontrar no *log* situações onde alterações no modelo de processo são registradas. São consideradas alterações a exclusão de uma atividade, participante, ou mesmo a exclusão de uma transição entre duas atividades existentes. Estes eventos são complexos de se identificar e geralmente não são registrados no *log*.



ID Regra	Participante	*Trace* ID
BR1	**Mary**	1
BR2	financialApp	1
BR3	orderApp	1
BR6	StockApp	1
BR7	DB	1
BR1	**John**	2
BR2	financialApp	2
BR4	orderApp	2
BR6	StockApp	2
BR7	DB	2

Figura 3.1: Mineração do *log* de execução parcial

Neste capítulo, serão introduzidos o comportamento básico e a estratégia de mineração adotada por cada algoritmo incremental para realizar a mineração do *log*. Também serão apresentados suas principais vantagens e limitações dentro da abordagem incremental.



ID Regra	Participante	Trace ID
BR1	**Mary**	1
BR2	financialApp	1
BR3	orderApp	1
BR6	StockApp	1
BR7	DB	1
BR1	**John**	2
BR2	financialApp	2
BR4	orderApp	2
BR6	StockApp	2
BR7	DB	2
BR1	**Mark**	3
BR2	finacialApp	3
BR5	sendMail	3
BR1	**Mark**	4
BR2	financialApp	4
BR5	sendMail	4

Figura 3.2: Mineração incremental dos novos eventos no *log*

3.1 Mineração Incremental Baseada no Versionamento de Documentos

Diferente das abordagens tradicionais de mineração que utilizam *log* de atividades, a abordagem apresentada nesta seção utiliza dados de versionamento de documentos para derivar as atividades de um processo (KINDLER; RUBIN; SCHAFER; 2006). Como é possível ver na Figura 3.3, esta abordagem consiste basicamente de quatro componentes: um *Sistema de Gerenciamento do Versionamento de Documentos* (ex: SVN, MS Source Safe, StarTeam, etc), um *Sistema Gerenciador de Recursos* e um *Sistema de Gerenciamento de Workflow* com sua *Interface de Usuário*. O *Sistema de Gerenciamento de Recursos* contém as informações sobre as interações do usuário

(participantes) com os documentos. Estas informações indicam quem criou um documento, ou mesmo quem modificou determinada versão de um documento existente. Já o *Sistema Gerenciador de Workflow* armazena as informações sobre todos os processos descobertos pelo algoritmo incremental. Ao final, a *Interface de Usuário* é utilizada para apresentar os modelos de processos descobertos ao usuário de uma forma amigável e visual.

Nesta abordagem, o usuário trabalha com o sistema de *Gerenciamento do Versionamento de Documentos* gravando novas versões dos seus documentos de trabalho (*check-in*/*check-out*) e consequentemente gerando as mensagens de versionamento no *log*. O *log* de versões consiste de um conjunto de registros onde cada entrada (*check-in*) contém informações sobre um documento, como nome, o usuário que o alterou, a data/hora da gravação e um comentário com informações adicionais.

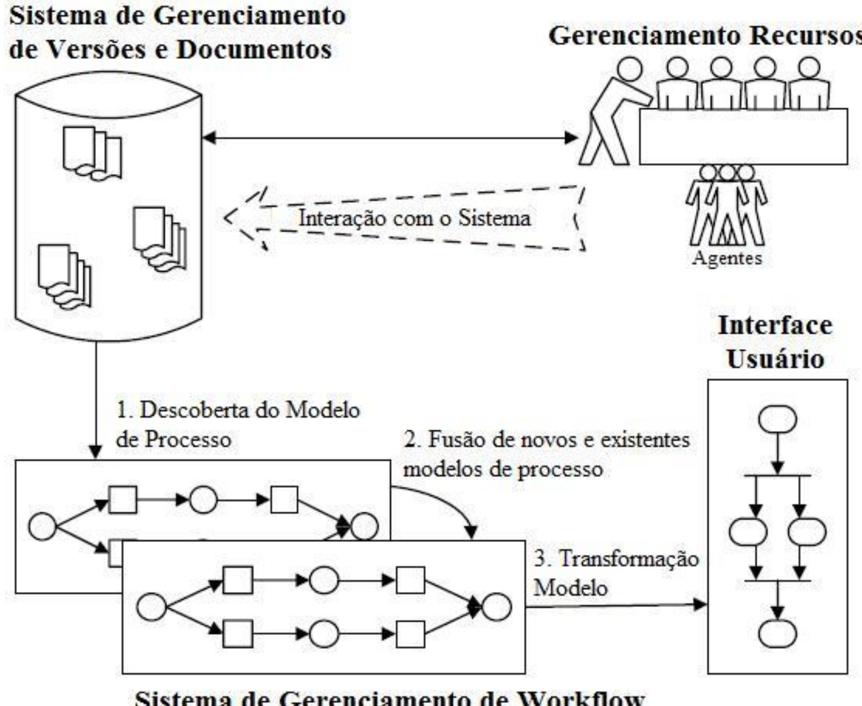


Figura 3.3: Esquema de mineração a partir do versionamento de documentos.

Seguindo a abordagem incremental de mineração de processos, os processos são descobertos pelo algoritmo e inseridos no *Sistema Gerenciador de Workflow*, onde são mantidos e executados posteriormente, caso assim for desejado. Os processos são então transformados e apresentados ao usuário em um formato compreensível do ponto de vista do negócio (ex: diagrama de atividades da UML). O usuário, considerando o processo apresentado, continua trabalhando com documentos, mesmo após a descoberta do modelo de processo. Este ciclo atualmente descreve o trabalho do usuário e os papéis dos sistemas envolvidos na mineração incremental de processos, os quais serão detalhados abaixo.

3.1.1 Abordagem Incremental de Mineração

Esta seção apresenta a idéia básica da mineração incremental de processos introduzida na seção anterior. Para isto, a Figura 3.4 apresenta os detalhes do esquema

de arquitetura de mineração. Os sistemas de *Gerenciamento do Versionamento de Documentos* e *Gestão de Recursos* servem como entrada para o algoritmo de mineração, gerando o *log* de versionamento de documentos. O *Sistema de Gerenciamento de Workflow* mantém os modelos de processos que são derivados durante as principais etapas da abordagem. Os retângulos arredondados da Figura 3.4 representam os algoritmos que derivam os modelos de processo, entre eles: Mineração, Fusão e Transformação. Os modelos de processos são derivados através da execução do algoritmo de mineração (*Mineração*). Em seguida, ocorre a integração dos modelos novos com os modelos já existentes (*Unificação*). Este processo é utilizado para realizar a atualização dos modelos de processo durante a mineração incremental, sendo este o principal passo dentro da mineração incremental. Por fim, os modelos são transformados (*Transformação*) para posterior apresentação ao usuário de negócio.

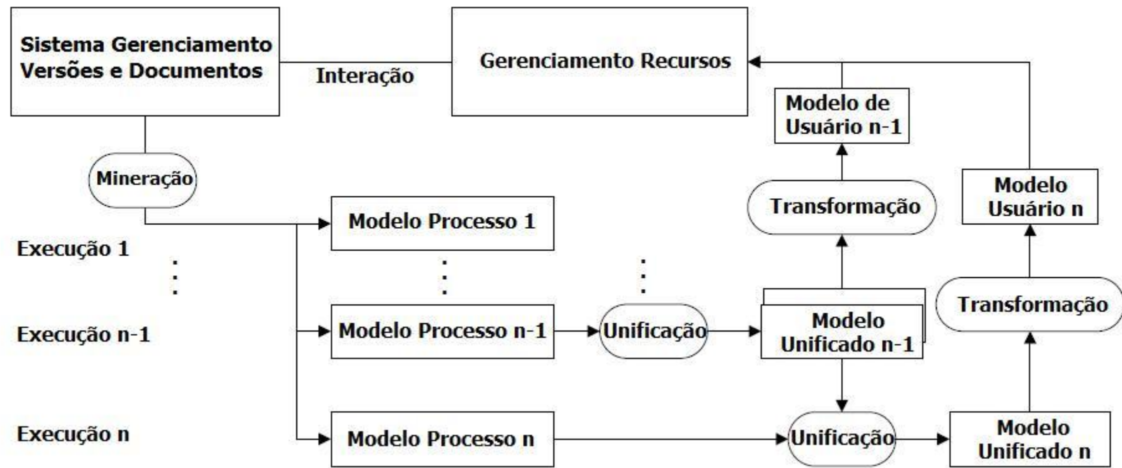


Figura 3.4: Esquema de Mineração Incremental

A abordagem de mineração incremental descrita acima pode ser representada pelo pseudocódigo da Figura 3.5. As etapas são enumeradas na parte esquerda do esquema.

```
Step 0. Let <Process Model>, <Merged Model> and
        <new Merged Model> be equal to null;
Step 1. Discover process model from
        versioning log ("Mining") -
        get <Process Model>;
Step 2. If <Merged Model> exists,
        then execute Merging algorithm ("Merging")
             and get <new Merged Model>,
        else let
             <Merged Model> be <Process Model>;
Step 3. If <new Merged Model> is not null
        then
             if <new Merged Model> is not equal
                to <Merged Model>
             then let <Merged Model> be
                  <new Merged Model>
                  and go to Step 1,
             else Stop:
        else go to Step 1.
```

Figura 3.5: Passos do algoritmo incremental

O primeiro passo (*Step* 1) obtém a primeira estrutura do processo a partir do *log* de versionamento. As dependências entre os documentos acessados são derivados pelo algoritmo incremental a partir do *log* gerado pelo *Sistema de Gerenciamento de Versionamento de Documentos*. Assim, para cada registro no *log* de execução, podemos derivar o contexto de uma atividade: pré-requisitos, documentos de saída e o usuário que criou a versão do documento. Os pré-requisitos são os documentos já atualizados no sistema. Esta atualização geralmente ocorre através de um comando de finalização (ex: "*commit*") no sistema de versionamento. O documento de saída é o documento gravado por uma atividade. O usuário representa o agente (participante) que gerou ou salvou o documento. A partir da data/hora da atualização do documento, é possível obter a ordem de execução das atividades e, conseqüentemente, o modelo de processo. Na Tabela 3.1, podemos ver um exemplo de *log* de versões de documentos contendo todas as informações discutidas acima.

Tabela 3.1: Log obtido a partir do Sistema de Gerenciamento de Versões

Documento	Revisão	Data/Hora	Autor	Comentário
Projeto	1.1	01-01-2011 14:00	Eng. Projeto	Modificação
Código	1.1	01-01-2011 14:30	Eng. Projeto	Modificação
Resultados Testes	1.1	01-01-2011 16:00	Eng. QA	Teste Unitário
Revisão Projeto	1.1	01-01-2011 17:00	Eng. Sistemas	Revisão Projeto

O segundo passo (*Step* 2) da abordagem consiste na execução do algoritmo de fusão de modelos (*Unificação*). Este passo é responsável por refinar um modelo existente (minerado anteriormente), incorporando-o a um modelo de processo produzido na primeira etapa. Este algoritmo é o ponto central da abordagem de mineração incremental do autor. Ele pode ser usado para o refinamento gradual do processo existente, depois de obter um novo modelo de processo por meio de descoberta de novas relações possíveis entre documentos versionados. Assim, com a ajuda da mineração e o algoritmo de *Unificação*, o modelo de processo é melhorado de forma incremental. Assim, um processo que era inicialmente executado de forma *ad-hoc* pelo usuário, pode de forma evolutiva se tornar um processo estruturado.

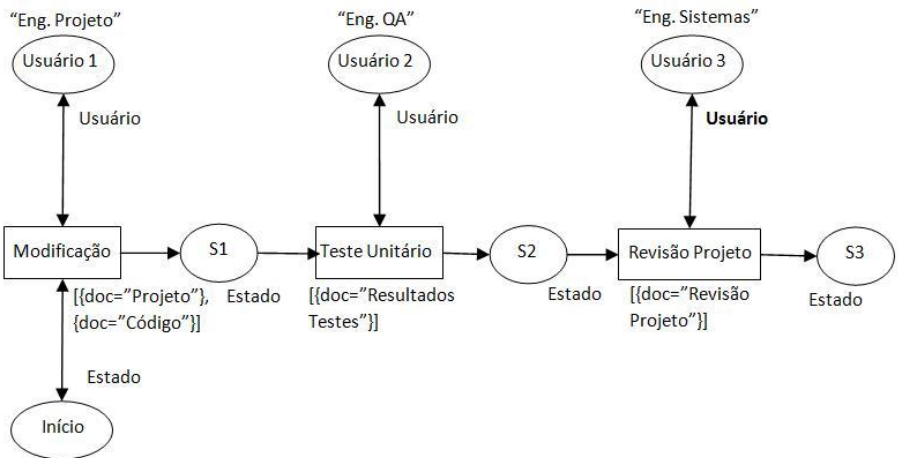


Figura 3.6: Modelo de processo extraído a partir do log de versionamento

A terceira etapa (*Step* 3) da abordagem apresentada nesta seção inclui a verificação das condições de laço do algoritmo. Ela verifica se a execução do algoritmo *Unificação*

deve ou não ser interrompida. A execução do algoritmo deve ser encerrada assim que o novo modelo incorporado e o modelo antigo se tornarem iguais. A implementação deste algoritmo deve conter as condições de guarda adicionais, o que exclui a situação de parada da execução em etapas iniciais por causa da coincidência de *logs* de execução. Como resultado, um modelo de processo representando uma sequência de atividades, usuários e informações sobre os documentos pode ser gerado. O modelo de processo do algoritmo é apresentado utilizando Redes de Petri, como mostrado na Figura 3.6. Ao final, o modelo descoberto pelo algoritmo é armazenado no *Sistema Gerenciador de Workflow* para futura utilização.

Para apresentar o modelo de uma forma mais direcionada ao usuário de negócio e com menos formalismos, a abordagem introduz também um algoritmo de transformação. Este algoritmo realiza a conversão do modelo representado por Redes de Petri para modelos em outros formatos, como diagrama de atividades da UML, por exemplo. Assim que um modelo de processo é gerado, o usuário é aconselhado a trabalhar com os documentos de acordo com o processo descoberto.

3.1.2 Vantagens e Limitações

A abordagem apresentada pelo autor permite a mineração incremental a partir de *logs* de versionamento de documentos. O *log* de versionamento de documentos pode revelar uma estrutura de processo implícita no gerenciamento de documentos de uma organização. Esta abordagem permite a descoberta de atividades de um processo sem a sua definição em tempo de projeto. Um grande diferencial desta abordagem está no fato do algoritmo considerar a descoberta de aspectos organizacionais, como o nome do participante que executou uma atividade do processo. Esta informação é desprezada pela grande maioria dos algoritmos de mineração, sendo ela de extrema importância para a geração de um modelo completo de processo.

Um aspecto interessante introduzido por esta abordagem é a possíbilidade da extração de processos *ad-hoc* a partir de documentos acessados, transformando-os em processos estruturados, e melhorando assim seu gerenciamento. A abordagem pode ainda ser estendida para funcionar não apenas com *Sistemas de Gerenciamento de Documentos*, mas também em sistemas ERP, por exemplo.

Outro aspecto que diferencia este trabalhado das abordagens mais clássicas, é o fato do algoritmo extrair as dependências entre atividades a partir de *logs* de documentos acessados e não a partir de dependências entre eventos de atividades registrados no *log* e sua ordem de execução.

Um dos principais problemas desta abordagem é a restrição quanto a atualização do modelo durante a mineração incremental. O algoritmo que realiza a unificação de modelos considera apenas a adição de elementos (ex: incorporação de novas atividades e transições), sem se preocupar com a remoção de elementos obsoletos, o que pode comprometer a qualidade do modelo resultante nestes casos.

Outro problema desta abordagem está na inferência dos nomes das atividades do modelo de processo. Atualmente está informação não está presente no *log* de versionamento. A única forma de obter o nome da atividade é através de sua extração a partir da coluna *Comentário* disponível no arquivo de *log*. Para esta abordagem ser efetiva, é necessário criar uma convensão durante a gravação ("*commit*") da entrada do *log*, informando um nome padronizado para a atividade, e esta sendo submetida como entrada para o algoritmo de mineração.

3.2 Mineração Incremental com Padrões Opcionais

Tarefas opcionais são tarefas que podem aparecer apenas em determinadas condições dentro da execução de um processo de negócio, sendo muitas vezes omitidas em diversas instâncias de processo. Na Figura Na Figura 3.7, é apresentado um exemplo de modelo processo onde a tarefa *Teste Laboratorial* é executada apenas na condição onde o paciente possui diabetes e sua idade é superior a 50 anos.

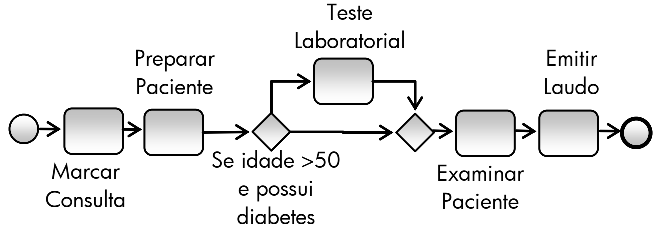


Figura 3.7: Processo médico com uma tarefa opcional

Algoritmos como α *Miner* (AALST; WEIJTER; MARUSTER, 2004) e o *Probabilistic Miner* (SILVA; ZHANG; SHANAHAN, 2005), possuem restrições para a mineração destas construções, tornando seus modelos incompletos quando encontram este tipo de situação.

Além da descoberta de tarefas opcionais, o algoritmo proposto por (Sun; Li; Peng, 2007) permite a atualização de modelos pré-existentes de processos ou a complementação de modelos de processos parciais, ambos através da mineração incremental. O algoritmo foi criado baseado no algoritmo *Probabilistic Miner* citado acima, com a diferença de permitir a mineração de padrões opcionais, além de representar o modelo de processo descoberto através de Redes de *workflow* (ver seção 2.2.2) ao invés de grafos acíclicos dirigidos (DAG).

As Redes de *Workflow* utilizadas para representar um modelo de processo minerado são derivadas de Redes de Petri, com a adição de elementos semânticos específicos para um modelo de processo, como AND/XOR-*Split/Join*, por exemplo. Sun et al definiu algumas premissas para o correto funcionamento das Redes de *Workflow,* onde $N = (P, T, F)$ neste algoritmo:

- Para todo $x \in P \cup T$, não existirá uma sequência $x_0,...,x_n$ tal que $x_i \in P \cup T$ e $(x_i, x_i{+}1) \in F$ onde $0 \leq i < n$ e $x = x_0 = x_n$;
- Para todo $x \in P \cup T$, se $|x^{\bullet}|>1$, então existe $y \in P \cup T$ tal que existe um caminho a partir de qualquer elemento em $x^{\bullet}$ até y.
- Se $x\ \sigma\ y$ e $x\ \alpha\ y$ são dois caminhos em N, z é qualquer nó da rede de *workflow* e existe um caminho em qualquer nó em uma sequência $\sigma\ \alpha\ y$ até z, então existe um caminho de y até z ou um caminho de z até y.

A primeira condição garante que a rede de *workflow* não terá ciclos. Ou seja, um grafo dirigido gerado pelo algoritmo precisa obrigatoriamente ser acíclico (DAG), impedindo que tarefas que executam repetidas vezes em sequência sejam descobertas. As últimas duas condições essencialmente fazem cumprir a sincronização de *threads* durante um *join* (AND-*join*) de atividades dentro do processo.

3.2.1 Definições do algoritmo

O autor utiliza uma guia ordenada O para o grafo de *workflow* G tal que $O(T_1, T_2)$ retorna *true*, *false* ou *exclusive* como descrito abaixo:

- Se T_1 e T_2 são sucessores observáveis diretos de um AND-*split*, então $O(T_1, T2) = O(T_2, T_1) = true$;
- Se T_1 é um antecessor de T_2, então $O(T_1, T_2) = true$;
- Se $O(T_1, T_2) = true$, então T_2 não é um antecessor de T_1;
- $O(T_1, T_2) = exclusive$ se e somente se T_1 e T_2 são mutuamente exclusivos;

De acordo com as definições acima, é possível ter $O(T_1, T_2) = true$ mesmo se T_1 não for um antecessor de T_2, e T_2 não for um antecessor de T_1.

Analogamente, assumindo neste momento que existe uma guia independente I para o grafo de *workflow* G tal que $I(T_i, T_j, T_k)$ é verdade se e somente se T_i e T_j são independentes dado $T_k = 1$ ($T_k = 1$ significa que o evento ocorreu, e $T_k = 0$ não ocorreu). Adicionalmente, T_i e T_j são independentes se:

- T_i e T_j são *d-separated* (SPIRTES; GLYMOUR; SCHEINES, 2000) dado um conjunto de tarefas T;
- T_i e T_j são *d-separated* dado um ancestral de alguma tarefa $T_k \in T$ tal que $T_k = 1$;
- T_i (ou T_j) é um ancestral de alguma tarefa $T_k \in T$ tal que $T_k = 1$;

3.2.2 Algoritmo Incremental de Mineração

O algoritmo de mineração de processos proposto pelo autor é dividido em duas fases:

1. A entrada para o algoritmo de mineração de processos inclui uma guia ordenada O e uma guia independente I. Além disso, o algoritmo recebe também o *log* de processos. Em seguida, técnicas de mineração de dados são aplicadas para obter os relacionamentos entre atividades, tal como ordenamento e independência (SILVA; ZHANG; SHANAHAN, 2005), grafo de dependência (AGRAWAL; GUNOPULOS; LEYMANN, 1998) e *log* baseado em relações de ordenação (AALST; MARUSTER, 2004). s

2. Modelo de processo é criado baseado nas relações entre atividades.

Para realizar a mineração inicial do *log* de processos, é utilizado o algoritmo *LearnWorklowNet* (ver Figura 3.8). O algoritmo processa as entradas existentes no *log* utilizando as fases definidas acima. Quando novos dados de *log* são disponibizados, o algoritmo simplesmente processa as novas entradas e gera um novo modelo de processo, sem se preocupar com o modelo minerado anteriormente. Para isto, o algoritmo inicia iterativamente adicionando nós filhos a um grafo parcialmente construído em uma ordem específica. Inicialmente a guia ordenada diz quais nós são “causas raiz” de todas as outras tarefas (ex: nós que não possuem um nó ancestral). Estes nós são identificados no Passo 3 do algoritmo. Se existirem mais de um nó como “causa raiz”, atividades não observáveis do tipo *split* precisam ser adicionadas ao grafo. Isto é realizado pelo algoritmo auxiliar *HiddenSplits.*

A cada iteração principal do algoritmo (Passos 11-16), é disponibilizado um conjunto de nós chamado *CurrentBlanket,* que contém todos os nós folha do grafo

corrente de *workflow N* (ex: todos os nós que não possuem filhos em *N*). A escolha inicial de nós no conjunto *CurrentBlanket* são exatamente as causas raíz. O próximo passo procura quais tarefas devem ser adicionados em *N*. O algoritmo deseja construir o grafo selecionando somente um conjunto de tarefas *NextBlanket* que:

- Não existir um par (T_1, T_2) no conjunto *NextBlanket* onde T_1 é um acestral de T_2 em *G*;
- Nenhum elemento no conjunto *NextBlanket* possui um ancestral em *G* que não está em *N*;
- Cada elemento em *NextBlanket* tem um ancestral em *G* que está em *N*;

```
Algorithm LearnWorkflowNet
Input     O, an ordering oracle for a set T of tasks;
          I, an independence oracle for T;
Output    N, workflow net
1.  N=(Ø,Ø,Ø) and G is a graph (V,E) where V=T and E=Ø
2.  E(G)←E(G)U{(t1,t2)|O(t1,t2)=true and O(t2,t1)=false)}
3.  CurrentBlanket={t|∀t1∈T:(t1,t)∉E(G)}
4.  T(N)←T(N)UCurrentBlanket
5.  OptionalSet←Ø
6.  (N, SplitNode,OptionalSet)←HiddenSplits(N,
       CurrentBlanket,O,OptionalSet)
7.  if SplitNode is transition
8.     Create a new place p
9.     add arc (p, SplitNode) to N
10. V(G)←V(G)-CurrentBlanket
11. While V(G)≠Ø
12.    NextBlanket←GetNextBlanket(CurrentBlanket,G,O,I)
13.    T(N)←T(N) U  NextBlanket
14.    Ancestors←Dependencies(CurrentBlanket,NextBlanket,O,I)
15.    (N,OptionSet)←InsertLatents(N,CurrentBlanket,
       NextBlanket,Ancestors,O,OptionalSet)
16.    G←G-NextBlanket
17. Let CurrentBlanket be the subset of T whose elements donot
    have a child in N
18. (N, JoinNode)←HiddenJoins(N,CurrentBlanket,O)
19. if JoinNode is transition
20.    Create a new place p
21.    add arc(JoinNode, p) to N
22. N← addOption(N, OptionalSet)
23. Return N
```

Figura 3.8: LearnWorkflowNet: algoritmo que minera e gera o modelo de processo a partir do log.

O algoritmo *GetNextBlanket* retorna um conjunto correspodendo a estas propriedades. Ainda é necessário identificar quais elementos no conjunto *NextBlanket* devem ser descendentes dos elementos em *CurrentBlanket*. Esta verificação é realizada pelo algoritmo *Dependencies*.

É provável que entre nós do conjunto *CurrentBlanket* e *NextBlanket* existam diversas atividades ocultas do tipo *join/split*. Estas atividades são detectadas e

adicionadas ao grafo *N* pelo algoritmo *InsertLatents*. Este procedimento é repetido até que todas as atividades observáveis no *log* sejam adicionadas à *N*. Ao final uma tarefa do tipo fim deve ser adicionado ao modelo. Esta tarefa garante que todas as tarefas anteriores (ex: *splits* de atividades) sejam sincronizadas ao final. Se uma tarefa final não for adicionada ao final do modelo, é possível que diversas tarefas fiquem pendentes durante a execução (ex: *threads* de execução poderiam ficar abertas em um *split* paralelo), gerando uma situação inconsistente. Para isto, ao final do algoritmo é realizada a chamada do algoritmo *HiddenJoins*. Este algoritmo adiciona ao modelo atividades ocultas de sincronização (ex: AND-*join ou XOR-join*) relacionadas à atividades *split* adicionadas anteriormente no modelo.

Após a descoberta do novo modelo de processo baseado em novas entradas no *log* e também da existência de um modelo minerado a partir das entradas inicias do *log*, é realizada a comparação entre o modelo pré-existe e o novo modelo. Esta comparação tem como objetivo obter as diferenças, e finalmente a realizar a fusão, atualizando ou complementando o modelo pré-existente (atualização incremental entre um modelo pré-existente e o novo novo). Para realizar esta comparação um modelo de relacionamento intermediário é gerado, conforme observado na Figura 3.9. A fusão dos modelos é composta por duas operações: *Complete* e *Update*. A operação *Complete* é utilizada quando existe um modelo parcial construído ou minerado anteriormente. Neste caso, esta operação adiciona novas tarefas ao modelo de acordo com o novo *log* e finalmente completa o modelo. Já a operação *Update* utiliza novas entradas no *log* para atualizar um modelo completo existente.

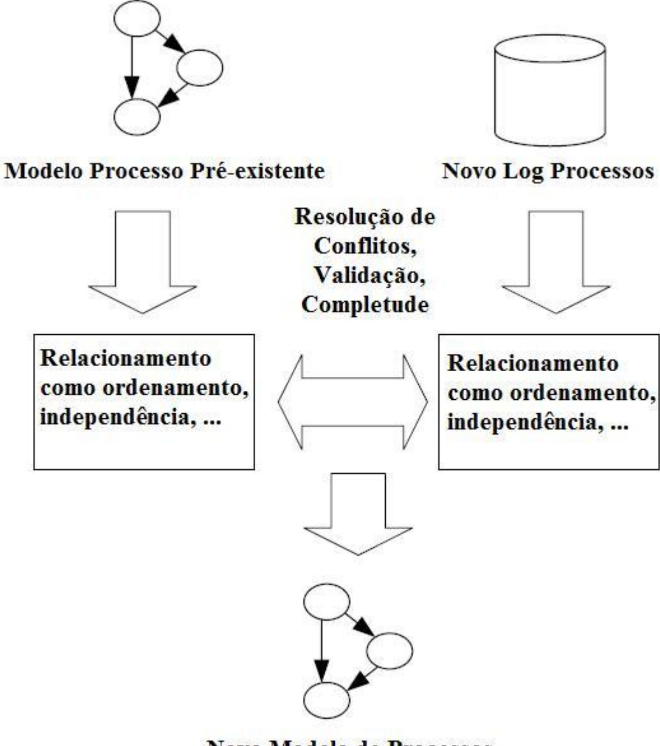


Figura 3.9: Processo de mineração incremental de processos

Normalmente, algoritmos de mineração de processos não fornecem procedimentos para extrair as relações intermediárias de um modelo de processo. O modelo de relacionamento intermediário é extraído tanto para o modelo pré-existe como para o

novo modelo a partir dos novos dados do *log*. Na Figura 3.10, o autor introduz algoritmos para a mineração das relações ordenadas e independentes, respectivamente, a partir de Redes de *Workflow* existentes. O algoritmo *LearnOrdering* recebe como entrada o modelo de processo pré-existente e também as atividade T_1 e T_2 do novo *log*. A partir destes dados o algoritmo retorna se a relação existe, se ela é exclusiva ou indefinida.

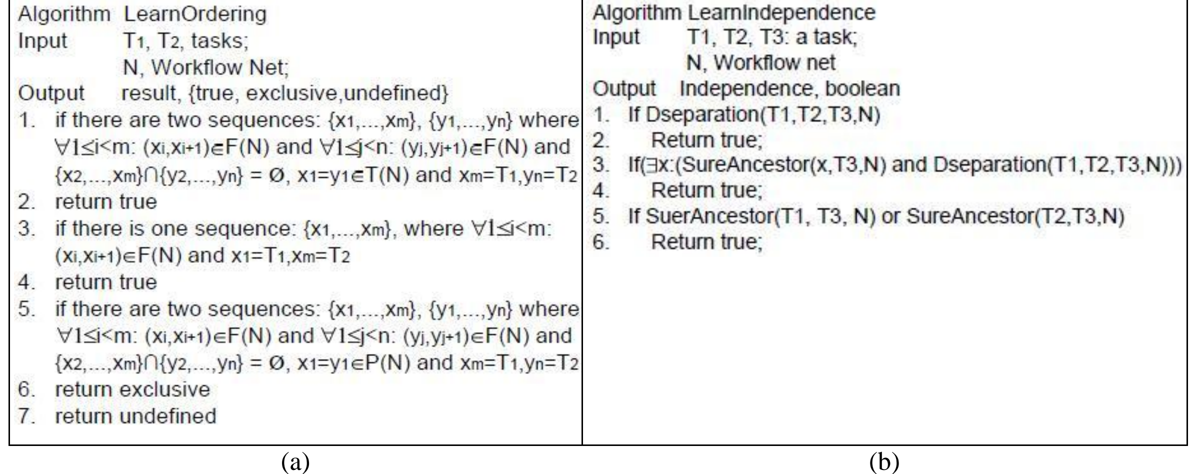


Figura 3.10: (a) LearnOrdering: aprende uma relação de ordenação a partir do modelo (b) LearnIndependence: aprende uma relação de independência a partir do modelo.

Quando obtemos as relações intermediárias a partir do modelo de processo pré-existente e do novo *log* de processo a ser processado, existe a possibilidade da ocorrência de inconsistências neste modelo e elas precisam ser removidas. Para isto, o autor criou os algoritmos *OrderingForIncremental* e *IndependenceForIncremental*, que permitem a extração incremental das relações a partir do modelo pré-existe e do novo *log* de processo. Estes algoritmos implementam a resolução de conflitos para eliminação de possíveis inconsistências no modelo resultante.

O algoritmo *OrderingForIncremental* mostrado na Figura 3.11 (a) ilustra como combinar as diferentes relações de ordenação entre o modelo pré-existe e as relações de ordenação extraídas do novo *log*. Da mesma forma, o algoritmo *IndependenceForIncremental* mostrado na Figura 3.11 (b) descreve como combinar as diferentes relações de independência. Nestes algoritmos foram definidos diferentes estratégias de resolução de conflitos, com fins específicos: complementação do modelo, atualização e comparação. Os algoritmos *LearnOrdering* e *LearnIndependence* são utilizados por estes algoritmos para a extração das relações de ordenação e independência.

```
Algorithm OrderingForIncremental
Input     T1,T2: tasks
          O, ordering oracle for new workflow log;
          N, a pre-existing workflow net;
          OP, operation
Output    R, ordering relationship between T1,T2
1.  result1←LearnOrdering(T1,T2,N)
2.  result2←O(T1,T2)
3.  If OP is Completion
4.      If result1 is undefined
5.          return result2
6.      else
7.          return result1
8.  If OP is Update
9.      If result2 is undefined
10.         return result1
11.     else
12.         return result2
13. if OP is Comparison
12.     if result1≠Result2
13.         print T1,T2, result1, result2
14.     return result1
```

(a)

```
Algorithm IndependenceForIncremental
Input     T1,T2,T3: tasks
          I, ordering oracle for new workflow log;
          N, a pre-existing workflow net;
          OP, operation
Output    R, independence relationship between T1,T2,T3
1.  result1←LearnIndependence(T1,T2,T3,N)
2.  result2←I(T1,T2,T3)
3.  if T1,T2 or T3∉ T(N)
4.  return result2
5.  if T1,T2 or T3 doesnot appear in new workflow log
6.  return result1
7.  If OP is Completion
8.          return result1
9.  If OP is Update
10.         return result2
11. if OP is Comparison
12.     if result1≠Result2
13.         print T1,T2,T3, result1, result2
14.     return result1
```

(b)

Figura 3.11: (a) *OrderingForIncremental*: um algoritmo para combinar o relacionamento de ordenação obtido a partir da rede de workflow e dos dados de log de processo. (b) *IndependenceForIncremental*: um algoritmo para combinar o relacionamento de independência obtido a partir da rede de workflow e dos dados de log de processo.

3.2.3 Descoberta de Padrões Opcionais

Com apenas relações de ordenação e independência descritas originalmente pelo algoritmo *Probabilistic Miner*, não é possível encontrar padrões opcionais. É necessário mais informações do *log* de processos. Desta forma, no algoritmo *LeranWorkflowNet* representado na Figura 3.8, é apresentado o conjunto *CurrentBlanket*. Este conjunto armazerna as tarefas formadas por nós folhas (sem descendente). Quando as tarefas neste conjunto são adicionadas à rede de *workflow*, essas tarefas serão removidas pelo algoritmo. Quando obtemos o conjunto *CurrentBlanket*, os traces de processo do *log* de processo corrente devem começar com *CurrentBlanket* se não existir nenhum padrão opcional no log. Se houver traces de processo iniciando com tarefas que não estão contidas no conjunto *CurrentBlank*, então é possível que estes traces sejam ruídos no log. Para estes casos, o autor definiu um *threshold* para frequência destes traces. Se a frequência destes eventos for superior ao *threshold,* então é alta a probabilidade deste ser uma tarefa opcional. Este *threshold* é fundamental pois ainda há a necessidade de se diferenciar eventos que são ruídos de eventos que serão opcionais. Desta forma, eventos com frequência inferior ao *threshold* serão tratados como ruídos, e consequentemente descartados. Este threshold é armazenado no conjunto *OptionalSet* quando o algoritmo *HiddenJoins* é executado. Devido ao fato do autor considerar que a estrutura de processo é aninhada, não é possível alcançar pontos internos desta estrutura de fora. Cada vez que o conjuto *CurrentBlanket* é dividido, é necessário verificar se o conjunto contém o conjunto *OptionalSet*. Se contém, então é possível obter o padrão opcional e registrá-lo no conjunto *OptionalSet*. Ao final do algoritmo *LeranWorkflowNet*, é executado o algoritmo *AddOption* para adicionar os padrões opcionais ao modelo de processo construído, conforme apresentado pela Figura 3.12.

```
Algorithm AddOption
Input       N, workflow net;
            OptionalSet, an optional Set;
Output    N, workflow net
1.  For each element x in OptionalSet
2.      For each element y in x3
3.          create a transition t and add it to N
4.          add arcs (x1, t), (t, y1) to N
5.          x3← x3-{y}
6.      If τ∈x2
7.          create a transition t and add it to N
8.          add arcs (x1, t), (t, z) to N where postSet(z) is empty
9.  Return N
```

Figura 3.12: AddOption: um algoritmo para adicionar padrões opcionais

3.2.4 Vantagens e Limitações

A abordagem incremental introduzida pelo autor possui características interessantes. Uma delas, é a possibilidade de comparar modelos de processos aparentemente diferentes executando em empresas similares. Neste caso, é possível comparar os diferentes modelos para obter diferenças e melhorar as propriedades de um modelo de processo, gerando um log destas diferenças no momento da construção do novo modelo durante o processo incremental.

Outro cenário similar que pode se beneficiar desta funcionalidade é a comparação de dois modelos similares onde um representa o modelo de processo projetado (fase de projeto) e o outro representa o modelo minerado a partir de sua execução (modelo implantado e executado).

Apesar do algoritmo se propor a resolver os principais problemas relacionados a mineração incremental dos processos, o autor não deixa claro em seu trabalho como estes pontos são tratados ou validados. Há um *déficit* em experimentos que comprovem sua eficácia ou mesmo sua performance quando comparados com outros algoritmos. Uma das grandes limitações deste algoritmo está na deficiência em minerar laços de atividades do processo. Ou seja, uma relação onde uma atividade executa ciclicamente (execução de duas ou mais vezes em sequência) não pode ser determinada pelo algoritmo. Outra limitação crítica do algoritmo, e que também foi observada pelo algoritmo anterior criado por Kindler et al, é a falta de suporte para a remoção de elementos durante a mineração incremental. Ou seja, se durante a modificação de um processo, atividades ou transições forem removidas, dificilmente o algoritmo conseguirá gerar modelos que reproduzam esta modificação. Por fim, é importante ressaltar também o aspecto negativo do algoritmo ao demonstrar acurácia infereior à dos outros algoritmos de mineração de processos, como *HeuristicMiner* e *GeneticMiner*. Isto ocorre principalmente nos cenários onde a mineração de laços de atividades é necessária, funcionalidade esta não suportada pelo algoritmo. Além disso, o algoritmo não extrai os participantes que executam cada atividade, conforme suportado pelo algoritmo de Kindler et al.

3.3 Mineração Incremental com Laços

O algoritmo apresentado nesta seção sugere melhorias sobre o algoritmo de mineração incremental apresentado anteriormente na seção 3.2. Aqui, (Ma; Tang; Wu, 2011) propõem como novidade a mineração de laços em um *log* de processos. O autor também propõe operações omitizadas de atualização e a complementação do modelo durante a mineração incremental.

3.3.1 *Framework* do algoritmo de mineração

Como o algoritmo anterior, proposto por (SUN; LI; PENG, 2007), este também utiliza o algoritmo de mineração probabilistico criado por (SILVA; ZHANG; SHANAHAN, 2005) como ponto de partida para implementação de um algoritmo incremental. *O* algoritmo *Probabilistic Miner* utiliza uma guia ordenada *O* e uma guia de independência *I*. *O* descreve a ordem de execução de tarefas: $t_1, t_2 \in Tarefa$, $O(t_1, t_2) = exclusivo$ se $t_1 \perp t_2$; $O(t_1, t_2) = true$ se $t_1 < t_2$ e caso contrário $O(t_1, t_2) = false$. Neste trabalho o autor faz uma variação desta definição. Ele utiliza relações de intersecção para explicitamente detactar paralelismo. Se $t_1 \times t_2$, então é possível definir diretamente $O(t_1, t_2) = O(t_2, t_1) = true$.

3.3.1.1 Estrutura de laço

Uma estrutura de laço representa um conjunto de tarefas que podem executar repetidamente. A Figura 3.13 mostra a representação formal de uma estrutura de laço. O autor decompos a estrutura em dois componentes: α *net* e β *net*. O conjunto das primeiras (ou últimas) tarefas executadas quando inicia (ou termina) um laço é chamado de *LS* (ou *LE*). Após executar a última tarefa, o processo determina se é necessário reexecutar o laço (para executar β é necessário seguir a seguinte ordem $\beta \rightarrow LS \rightarrow \delta \rightarrow LE$) ou deixar o laço para executar a tarefa γ.

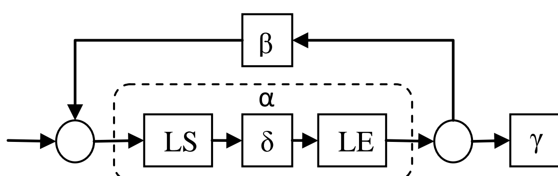


Figura 3.13: Representação formal de uma estrutura de laço

Definição 1 (Laço). Uma estrutura de laço pode ser representada como seis tuplas Laço=(*ID, TaskList, LS, LE,* L_{α}, L_{β}). *ID* é o identificador do laço; *TaskList* representa o conjunto de atividades dentro do laço que podem ser agrupadas e chamadas como tarefa de laço. *LS/LE* é o conjunto da primeira/última tarefa executada no laço. L_{α}, L_{β} respectivamente contém os *traces* de *workflow* das redes α e β.

Definição 2 (Guias ou *Oracles*). *Oracles* = (*O, I, LCS*). *O* e *I* são respectivamente as relações de ordenação e independência das tarefas mais externas e das tarefas virtuais que representam os laços externos. $LCS = \{LC_1, ..., LC_n\}$ onde $LC = (ID, Oracles_{\alpha}, Oracles_{\beta}) \in LCS$ registra as informações de um laço. ID é o identificador do laço; $Oracles_{\alpha}$, e $Oracles_{\beta}$ são respectivamente as relações das redes α e β. Se um modelo de processo não contém laços, então LCS = ∅.

3.3.1.2 Algoritmo

O método criado pelo autor possui dois passos: execução do algoritmo *LearnOracleFromLog* e *LearnWfNFromOracles*. O algoritmo *LearnOracleFromLog* calcula as relações a partir do *log* de processo. Primeiramente ele reconhece os laços mais externos e substitui o segmento da atividade no laço do *trace* de *workflow* por uma tarefa virtual. Esta operação é tratada especificamente como remoção do *loop* ou normalização do *log*. Em seguida, um *log* sem laço (*NoLoopLog)* e as informações sobre o laço (*LoopSet*) são obtidas. O aprendizado das relações é realizado de forma recursiva a partir do *log*. As relações *Oracles.O* e *Oracles.I* são computadas diretamente a partir do *log* sem laços *NoLoopLog*. A relação *Oracle.LCS* é computada com *LoopSet* como parâmetro de entrada.

O algoritmo *LearnWfNFromOracles* aprende o modelo de *workflow* a partir das relações da mesma forma recursiva que o algoritmo anterior. Primeiramente é realiza a descoberta do modelo de processo utilizando o algoritmo probabilistico descrito na seção 3.3.1 e então para cada laço é realizada a mineração das sub-redes α *net* e β *net*. Finalmente o algoritmo insere as sub-redes que representam o laço na rede principal.

3.3.2 Mineração incremental com laços

A operação de fusão (*Merging*) de um modelo existente e das novas entradas no *log* ocorre de forma similar ao algoritmo incremental original proposto por de Sun et al. Primeiramente é obtido a relação $Oracles_M$ a partir de um modelo existente e $Oracle_L$ a partir do novo *log* e então ocorre a fusão de ambas. Abaixo, será discutido dois problemas: (i) como obter a relação $Oracles_M$, e (ii) como eliminar as inconsistências entre $Oracles_M$ e $Oracles_L$.

3.3.2.1 Aprendendo as relações a partir do modelo

O algoritmo *LearnOraclesFromModel* apresentado na Figura 3.14 é responsável por aprender as guias de um modelo. Se dois nós estão em um laço, eles são mutuamente acessíveis na rede. As etapas 1 e 2 descobrem os conjuntos de nós que estão no laço mais externo. Uma vez que as estruturas estão devidamente aninhadas, existirá uma única entrada e uma única saída para cada laço mais externo. No passo 4.2 WfN_α WfN_β podem ser obtidos por um algoritmo *first-depth-traverse* (HOPCROFT; TARJAN, 1973), o qual realiza o percorrimento de um grafo visitando cada vértice adjacente a um nó específico. O passo 5 descobre as relações *Oracles.O* e *Oracles.I* a partir do modelo de processo sem laço, que já foram discutidos anteriormente.

Algorithm: *LearnOraclesFromModel*

1. Let $WfN = (P, T, F)$ be a workflow net, R be a relation over $P \bigcup T$: for $n_1, n_2 \in (P \bigcup T), n_1 R n_2$ if there is a reachable path from n_1 to n_2 in *WfN*.
2. Let $LoopSet_M$ be the set of nontrivial equivalence classes of R.
3. $Oracles.LCS \leftarrow \emptyset$
4. foreach $Loop_M \in LoopSet_M$
   - 4.1 Get $n_i : n_i \in Loop_M$, $\exists x \in (P \bigcup T) / Loop_M, (x, n_i) \in F$;
     Get $n_o : n_o \in Loop_M$, $\exists x \in (P \bigcup T) / Loop_M, (n_o, x) \in F$;
   - 4.2 The subgraph from n_i to n_o is WfN_α; the subgraph from n_o to n_i is WfN_β;
   - 4.3 $LC.Oracles_\alpha \leftarrow LearnOraclesFromModel(WfN_\alpha)$;
   - 4.4 $LC.Oracles_\beta \leftarrow LearnOraclesFromModel(WfN_\beta)$;
   - 4.5 Add *LC* into *Oracles.LCS* ;
5. $Oracles.O \leftarrow LearnOrdering()$; $Oracles.I \leftarrow LearnIndependence()$;
6. Return *Oracles*;

Figura 3.14: Algoritmo LearnOraclesFromModel

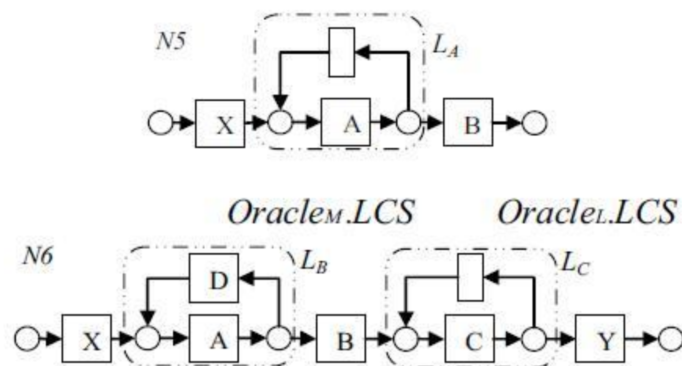


Figura 3.15: Um exemplo de modelo existente e um novo log

3.3.2.2 Realizando a fusão (Merging)

A fusão de guias com laços é uma tarefa mais complexa que a descrita na última seção. Existem dois problemas principais:

Definição 1 (Similaridade entre laços). Seja LC_1 and LC_2 dois laços. $Sim(LC_1, LC_2) = |Tarefas(LC_1) \cap Tarefas(LC_2)|$ é definido como a similaridade de LC_1 and LC_2.

Definição 2 (O laço mais encontrado). Seja LC um laço, $LCS = \{LC_1, LC_2, ..., LC_n\}$. $LC_M \in LCS$ é o laço mais encontrado de LC se $\neg \exists LC' \in LCS$, $Sim(LC', LC) > Sim(LC_M, LC)$. Se $\forall LC \in LCS$, $Sim(LC', LC) = 0$, então $LC_M = null$.

(i) $Oracles_M$ e $Oracles_L$ gravam respectivamente informações sobre seus laços. Como é mostrado na Figura 3.15, o modelo existente é *N5* e o novo *log* é *N6* (para melhor legibilidade, o autor utilizou um modelo para representar as ordens de execução de tarefas no *log*. Na verdade *N6* não é extraído). O problema é como combinar os laços correspondentes em $Oracles_M.LCS$ e $Oracles_L.LCS$.

Para este problema, o autor utilizou uma solução baseada em heuristica: dois laços que possuem tarefas em comum são provavelmente o mesmo laço.

(ii) A posição de uma tarefa com relação a um laço é alterada. É possível que uma tarefa esteja originalmente em um laço no modelo existente, mas é movido para fora do laço no novo *log*. Em um cenário mais complexo, um laço é dividido em dois ou mais laços menores, ou alguns laços são unidos em um único laço. Aqui é seguido o princípio do funcionamento das operações de *Modelo Completo* e *Atualização do Modelo* introduzidas anteriormente. Utilizando a operação de *Modelo Completo* como exemplo. Se uma tarefa está em um laço ou não, ou em qual laço a tarefa está, é determinada por seu lugar original no modelo existente. Se a tarefa é uma tarefa recém-adicionada, então o lugar da tarefa no novo modelo é determinado pelo local onde ela está no novo *log*.

Na Figura 3.16, é apresentada a operação *Completar Modelo* através do algoritmo *CompleteOracles*. O passo 3 é responsável por conferir cada LC em $Oracles_L.LCS$ de $Oracles_M.LCS$ para encontrar o laço LC_M mais frequente. O Passo 3.3 usa uma forma recursiva para completar LC_M com LC. Os passos 4-5 calculam *Oracles.O* e *Oracles.I* como definido no início da seção 3.2.1. A operação *Atualização do Modelo* possui comportamento similar, apenas invertendo o comportamento de $Oracles_M$ e $Oracles_L$.

```
Algorithm: CompleteOracles
1.  if ( Oracles_L is null) return Oracles_M ;
2.  Oracles.LCS ← ∅ ;
3.  foreach LC ∈ Oracles_L.LCS
    3.1 LC_M ← GetMostSimilarLC(Oracles_M.LCS, LC) ;
    3.2 if ( LC_M is null) add LC into Oracles.LCS ;
    3.3 if ( LC_M is not null )
        a) NewLC.Oracles_α ← CompleteOracles(LC_M.Oracles_α, LC.Oracles_α) ;
        b) NewLC.Oracles_β ← CompleteOracles(LC_M.Oracles_β, LC.Oracles_β) ;
        c) Add NewLC into Oracles.LCS ;
4.  Oracles.O ← Complete(Oracles_M.O, Oracles_L.O) ;
5.  Oracles.I ← Complete(Oracles_M.I, Oracles_L.I) ;
6.  Return Oracles ;
```

Figura 3.16: Algoritmo *CompleteOracles*.

Como ilustrado na Figura 3.15, *N5* contém *X*, L_A e *B*. L_A representa um laço que contém uma tarefa *A*. O novo *log* contém *X*, L_B, *B*, *Y* e L_C. L_B e L_C são, respectivamente, laços. Combinando a laços descobrimos que L_B corresponde a L_A e L_C é um laço recém-adicionado. As fusões de $Oracles_M.O$ e $Oracles_L.O$, $Oracles_M.I$ e $Oracles_L.I$ retornam a precedência entre as tarefas *X*, L_A (L_B), *B*, L_C e *Y*. Por fim, L_A e L_B são combinados recursivamente.

3.3.3 Vantagens e Limitações

Ma et al propuseram um novo algoritmo capaz de realizar a mineração incremental de um log de processos. O algoritmo permite a complementação de um modelo parcial, assim como a atualização de um modelo já completo quando mudanças ocorrem nos requisitos do processo de negócio. Como principal melhoria sobre o algoritmo proposto por Sun et al, neste algoritmo é possível minerar laços de atividades a partir do *log* de execução do processo. Porém, assim como todos os outros algoritmos incrementais apresentados, este também deixa de considerar a remoção de elementos do processo de negócio.

Quando comparado com algoritmos mais simples de mineração de processos, como *alpha miner* por exemplo, a acurácia do algoritmo ficou um pouco abaixo dos demais. Isto pode ser observado de forma qualitativa pelos experimentos apresentados pelo autor em seu trabalho publicado.

3.4 Comparativo das abordagens

Neste capítulo foram apresentados os principais algoritmos incrementais de mineração de processos. Para resumir as principais características de cada algoritmo descritas anteriormente, foi criada a Tabela 3.2 abaixo. Podemos verificar que os principais aspectos da mineração foram considerados (seção 2.3), como mineração de atividades invisíveis, atores, laços, ruídos no *log* e adição e remoção de elementos do modelo. A tabela ainda destaca as principais limitações de cada abordagem. Estes aspectos podem influenciar de forma positiva ou negativa a qualidade do modelo final extraído do *log* (métrica *fitness*), como será apresentado no Capítulo 5.

Na última coluna da tabela, foi adicionado o algoritmo *IncrementalMiner*, o qual foi proposto neste trabalho e apresentado no Capítulo 4. É possível observar que o algoritmo *IncrementalMiner* atende de forma satisfatória os principais aspectos da mineração, além de introduzir melhorias na mineração incremental, como o suporte a remoção de elementos obsoletos do processo (linha 2 da Tabela 3.2). Este aspecto não foi considerado por nenhum dos outros algoritmos incrementais. Desta forma, podemos inferir que modelos com baixa acurácia podem ser gerados pelos demais algoritmos durante a mineração de processos que envolvam este cenário.

Tabela 3.2: Resumo do comparativo entre as abordagens incrementais

	Aspecto	**Incremental Mining (Kindler et al)**	**Incremental Mining (Sun et al)**	**Incremental Mining (Ma et al)**	**Incremental Miner (Kalsing et al)**
1	Suporta mineração Incremental com adição de elementos do processo	Sim	Sim	Sim	**Sim**
2	Suporta mineração Incremental com remoção de elementos do processo	Não	Não	Não	**Sim**
3	Lida com Ruídos	Não	Sim	Sim	**Sim**
4	Tarefa Invisível (AND-Split/Join)	Sim	Sim	Sim	**Sim**
5	Tarefa Invisível (OR-Split/Join)	Sim	Sim	Sim	**Sim**
6	Tarefas Opcionais	Sim	Sim	Não	**Sim**
7	Laços	Sim	Não	Sim	**Sim**
8	*Log* MXML	Não	Não	Não	**Sim**
9	Participantes	Sim	Não	Não	**Sim**
10	Complexidade do Algoritmo	N/D	$O(n^3)$	$O(n^3)$	**$O(n^3)$**

É possível verificar na tabela que, com exceção do algoritmo proposto por Kindler et al, todos os demais algoritmos executam em tempo $O(n^3)$, onde n é o número de tarefas presente em um *trace* de processo no *log*.

Para um maior detalhamento de alguns dos aspectos apresentados acima, no Capítulo 5 foi realizado uma análise quantitativa entre alguns algoritmos. Este comparativo incluiu principalmente os algoritmos não incrementais, com a finalidade de comparar a qualidade dos modelos gerados. Infelizmente não foi possível realizar este comparativo com os algoritmos incrementais devido a falta de documentação (ex: código fonte, arquivos binários de execução dos programas, dados de teste, etc) disponibilizada pelos autores, o que impediu a execução de experimentos mais elaborados e precisos entre estes algoritmos.

4 ABORDAGEM INCREMENTAL DE MINERAÇÃO

A mineração de processos a partir de sistemas de informação pode ser uma tarefa inprodutiva e pouco eficaz se uma estratégia de execução não for definida *apriori*. Para que este processo seja eficaz e executado no menor tempo possível, foi definido um conjunto de passos que nos permite coordenar a mineração de acordo com a estratégia ou necessecidade de uma organização (ver Figura 4.1). Estes passos consideram a extração do processo de forma gradual e contínua. Desta forma, podemos definir quais módulos de uma aplicação serão considerados durante a mineração de um processo de negócio, ou mesmo escolher um conjunto específico de usuários que interagem e trabalham com estes sistemas. Ao final, é esperado a descoberta evolutiva de processos de negócio parciais ou completos a partir do comportamento dinâmico do sistema registrado no log durante a sua execução.

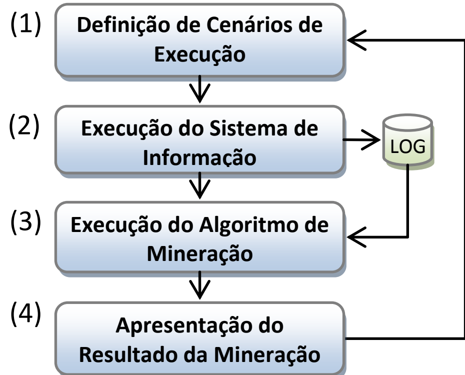


Figura 4.1: Mineração Incremental de Processos a partir de *logs* de execução de sistemas de informação.

Outra importante definição desta abordagem incremental é como e quais eventos gerados por um sistema de informação devem ser considerados na gravação do *log* de execução. Para isto, foi definido que todos os eventos gerados por um sistema de informação são instâncias de regras de negócio implementadas no código fonte da aplicação. Estas regras de negócio são identificadas e anotadas no código fonte para que seja então possível a geração do *log*. É importante salientar que a identificação das regras de negócio é pré-requisito para a correta mineração do *log*. Neste trabalho, foi assumido que a descoberta das regras não faz parte do processo de mineração. Esta descoberta envolve a aplicação de técnicas de análise estática de código fonte (PUTRYCZ et al, 2007), (CHIANG et al, 2006), (WANG et al, 2008), (SNEED et al, 1996), (HUANG et al, 1996), (RADAELLI JUNIOR; NASCIMENTO; IOCHPE, 2011) que não serão abordadas neste trabalho. Desta forma, é premissa que um código fonte já analisado e com suas regras identificadas, seja fornecido como entrada para o processo.

Este capítulo descreve os passos da mineração incremental e qual a sua importância para a mineração eficaz do *log* de execução. O primeiro passo do processo define os cenários de execução a serem considerados durante a execução e geração dos logs da aplicação. A partir da definição destes cenários, podemos selecionar quais os trechos de uma aplicação serão executados, ou mesmo escolher os usuários que executarão estas aplicações. Com isto, serão gerados no *log* registros de *trace* que apenas fazem parte destes cenários. No segundo passo, os sistemas são executados para geração dos *logs* de processo. É neste passo que o *log* de execução é gerado. A execução da aplicação é realizada com auxilio dos cenários definidos no passo anterior. Desta forma, uma execução orquestrada das aplicações é realizada de acordo com os passos definidos nos cenários. Já no terceiro e principal passo, é realizada a mineração do *log* de execução. Este passo é fundamental pois é nele que ocorre a descoberta das estruturas do processo pelo algoritmo *IncrementalMiner*. No último passo, é feita a apresentação do modelo descoberto ao usuário. A apresentação é feita de maneira gráfica a fim de permitir a visualização do fluxo do processo descoberto, assim como os demais componentes do modelo de processo.

Antes da apresentação dos passos da abordagem incremental, será introduzido abaixo o modelo conceitual de processos utilizado para representação dos modelos de processos utilizados neste trabalho.

4.1 Modelo Conceitual

Todas as informações relacionadas a processos e regras de negócio foram representadas por entidades de um modelo conceitual. Com este modelo conceitual é possível realizar o mapeamento entre o resultado gerado pelo algoritmo e a representação do processo em um nível de abstração mais alto e compreensível por analistas de negócio.

O modelo é composto basicamente pelos componentes que fazem parte da estrutura de um processo de negócio, como tarefas e suas regras de negócio, controles de fluxo, participantes, etc. As duas próximas seções descreverão cada uma das entidades que compõem este modelo conceitual.

4.1.1 Modelo Conceitual de Processo de Negócio

A estrutura de processo de negócio utilizado neste trabalho é representada pelo modelo conceitual da Figura 4.2 e graficamente apresentada pelo modelo de processo da Figura 4.3. Este modelo é formado por um subconjunto de elementos do modelo conceitual de processo criado e mantido pela WfMC (WFMC, 2008), conforme descrito abaixo.

1) *Processo* – a entidade de mais alto nível que representa um fluxo de trabalho completo ou simplesmente segmentos deste fluxo.

2) *Atividade* – a entidade de mais baixo nível no processo e que representa uma entidade abstrata no processo. Uma atividade pode ser especializada como uma *Tarefa Evento* ou *Gateway* (controle de fluxo).

3) *Tarefa* – atividade que representa uma unidade de trabalho atômica em um processo de negócio e que pode ser executada por um participante do processo. Neste trabalho, uma tarefa é sempre composta por uma regra de negócio do tipo Unidade de Trabalho (ver seção 4.1.2).

4) *Evento* – atividade que representa um evento de início ou fim no processo. Geralmente um processo inicia ou termina com este tipo de atividade.

5) *Transição* – representa uma sequência entre duas atividades. Ela pode ser representada por uma transição simples (transição entre duas atividade) ou por um laço (repetição de uma mesma atividade).

6) *Gateway* – entidade de controle de fluxo que define uma rota entre duas ou mais tarefas. Um *gateway* é um tipo de atividade dentro de um processo de negócio e também está associado a regras de negócio. Um *gateway* pode ser dividido em 3 tipos: *Inclusivo*, *Exclusivo* e *Paralelo*. O *Gateway Inclusivo* representa uma rota do tipo OR, onde uma ou mais rotas podem ser seguidas dependendo de uma decisão. A decisão por seguir uma rota (transição) ou outra está associada a uma condição atrelada à entidade *Transição*. Um *gateway Exclusivo* define uma rota do tipo XOR (OU exclusivo com tarefas mutuamente exclusivas) onde apenas um caminho pode ser seguido. Já o *gateway Paralelo* permite que duas ou mais atividades sejam executadas paralelamente.

7) *Participante* – um participante é um ator do processo que realiza uma tarefa. Um participante pode ser do tipo humano, sistema, papel ou uma unidade organizacional que realiza uma tarefa.

8) *Aplicação* – uma aplicação é responsável por realizar uma tarefa dentro do processo. Uma aplicação pode ser do tipo Regra de Negócio, *Web Services*, uma aplicação Java ou um *script*. Neste trabalho, cada tarefa será sempre associada a uma aplicação do tipo Regra de Negócio, extraída a partir do código de um sistema de informação. Esta regra de negócio é composta pela entidade *UnidadeTrabalhoBR* (ver seção 4.1.2) e representa uma unidade de trabalho atômica do processo.

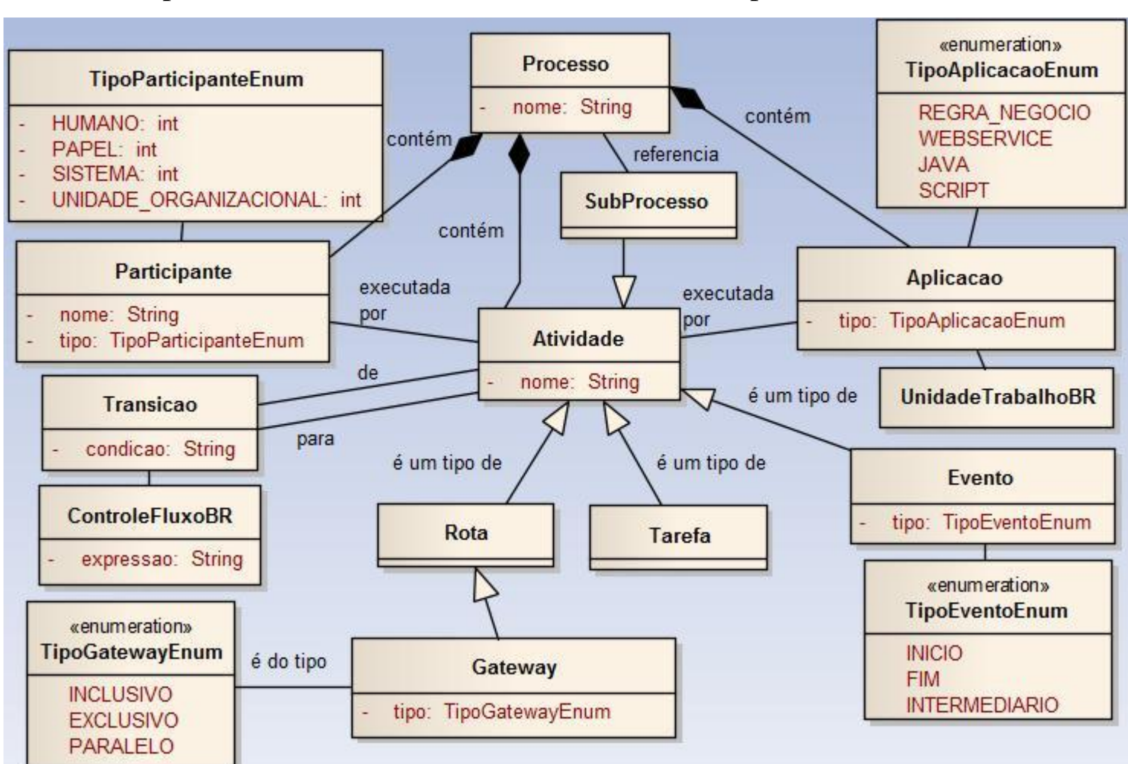


Figura 4.2: Modelo conceitual de processos baseado no modelo da WfMC.

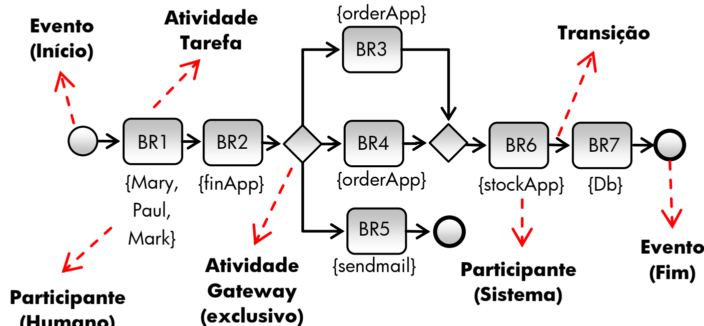


Figura 4.3: Exemplo de modelo de processo representando as entidades do modelo conceitual.

4.1.2 Conjunto de Regras de Negócio

De acordo com a entidade *Business Rule Group* (BRG, 2000), a partir da perspectiva de negócios, regras de negócio são uma obrigação de conduta, ação, prática ou procedimento dentro de uma determinada atividade ou esfera. Do ponto de vista de um sistema de informação, uma regra de negócio é uma declaração que define ou restringe algum aspecto do negócio (OMG, 2009). Neste trabalho, uma regra de negócio representa um fragmento de código de um sistema de informação que executa alguma lógica de negócio. Ela é utilizada como origem da geração dos eventos de *log*. Ou seja, quando uma regra de negócio é executada em um sistema, um evento é gerado registrando os dados de execução desta regra. É importante ressaltar que o termo evento utilizado aqui é diferente da entidade *Evento* do modelo conceitual de processo descrito anteriormente na seção 4.1.1. Aqui, um evento é qualquer atividade executada e gravada no *log*, podendo ela ser uma tarefa ou até mesmo um evento de processo.

O processo de geração de *log* a partir da execução d sistema é realizado através da instrumentação do sistema de informação. A instrumentação é o processo de anotação das regras de negócio no código fonte para posterior registro de eventos no *log* quando este código for executado. Na tabela abaixo, é possível ver um exemplo de código fonte instrumentado para geração de *logs* a partir da execução de regras de negócio. As linhas 4, 11, 16, 21, 28, 34 e 39 da Figura 4.4 representam as instruções adicionadas (instrumentação) ao início de cada regra de negócio para a gravação do *log*. Estas instruções são adicionadas ao código fonte antes do processo de mineração (ex: durante o processo de análise estática do código fonte o qual não faz parte deste trabalho). Para tal, o sistema é geralmente alterado, recompilado e disponibilizado novamente com a instrumentação ativada.

```
1  . . . . . . . . . . . . .
2  /* MAIS CÓDIGO AQUI */
3  . . . . . . . . . . . . .
4  EscreveLog(usuarioCorrente, “BR1”, traceId);
5  /* @INICIO_REGRA 1 – Interação Usuário */
6  printf("PEDIDO:PRODUTO/PREÇO/CLIENTE:");
7  scanf("%s", pedido.produto);
8  scanf("%f", pedido.preco);
9  scanf("%s", pedido.cliente);
10 /* @FIM_REGRA 1 */
11 EscreveLog (“finApp”, “BR2” , traceId);
12 /* @INICIO_REGRA 2 – Invocação Função */
13 aprovado=fin.analiseCredito(pedido.cliente);
14 /* @FIM_REGRA 2*/
15 if (aprovado && preco >1000) {
16       EscreveLog (“orderapp”, “BR3” , traceId);
17        /* @INICIO_REGRA 3 – Cálculo Mate. */
18       novoPreco =   preco - (preco * 0.10);
19       /* @FIM_REGRA 3 */
20 } else if  (aprovado) {
21       EscreveLog (“orderapp”, “BR4” , traceId);
22       /* @INICIO_REGRA 4 – Cálculo Matem. */
23        novoPreco = preco - (preco * 0.05);
24       /* @FIM_REGRA 4 */
25 }
26
27 else {
28    EscreveLog(“sendMail”, “BR5” , traceId);
29    /* @INICIO_REGRA 5 – Invocação Fun. */
30    sendMail.Enviar(cust,"crédito rejeitado!");
31    /* @FIM_REGRA 5*/
32    return;
33 }
34 EscreveLog(“stockApp”, “BR6” , traceId)
35  /* @INICIO_REGRA 6 – Invocação Função */
36  estoque.Atualizar(produto, pedido.qtde);
37  /* @FIM_REGRA 6*/
38  Query q(db);
39 EscreveLog (“Db”, “BR7” , traceId);
40 /* @INICIO_REGRA 7 – Persistência Dados */
41 q.execute (“INSERT INTO pedidos VALUES (“ +
42 pedido.Id  +”,”  +  pedido.preco  +  “,”  +
43 pedido.produto + “)”);
44 /* @FIM_REGRA 7*/
45 . . . . . . . . . . . . .
46 /* MAIS CÓDIGO AQUI */
47  . . . . . . . . . . .
```

Figura 4.4: Listagem com exemplo de código fonte instrumentado para geração de eventos no *log*.

Já a listagem da Figura 4.5 ilustrada abaixo apresenta o mesmo código fonte da Figura 4.4, porém com uma pequena alteração no código. Esta alteração introduz a remoção de regras de negócio que eram executadas na aplicação (linhas 15-19 da Figura 4.4 foram removidas na Figura 4.5), modificando consequentemente todos os novos *traces* de execução da aplicação gerados no *log*. Esta modificação representa um ponto chave deste processo de mineração e será abordada em detalhes na seção 4.4.4.

```
1  . . . . . . . . . . . . .
2  /* MAIS CÓDIGO AQUI */
3  . . . . . . . . . . . . .
4  EscreverLog(usuarioCorrente,“BR1”, traceId);
5  /* @INICIO_REGRA 1 – Interação Usuário*/
6  printf("PEDIDO:PRODUTO/PREÇO/CLIENTE:");
7  scanf("%s", pedido.produto);
8  scanf("%f", pedido.preco);
9  scanf("%s", pedido.cliente);
10 /* @FIM_REGRA 1 */
11 EscreverLog (“finApp”, “BR2” , traceId);
12 /* @INICIO_REGRA 2 – Invocação Função */
13 aprovado=fin.analiseCredito(pedido.cliente);
14 /* @FIM_REGRA 2*/
15 if  (aprovado) {
16       EscreverLog(“orderapp”, “BR4” , traceId);
17      /* @INICIO_REGRA 4 – Cálculo Matem. */
18       novoPreco =   preco - (preco * 0.05);
19      /* @FIM_REGRA 4 */
20 }
21
22 else {
23       EscreverLog(“sendMail”,“BR5”, traceId);
24       /* @INICIO_REGRA 5 – Invoca Função*/
25      sendMail.Enviar(cust,"crédito rejeitado!");
26       /* @FIM_REGRA 5*/
27      return;
28 }
29 EscreverLog(“stockApp”, “BR6” , traceId)
30  /* @INICIO_REGRA 6 – Invocar Função*/
31  estoque.Atualizar(produto, pedido.qtde);
32  /* @FIM_REGRA 6*/
33 Query q(db);
34 EscreverLog(“Db”, “BR7” , traceId);
34 /* @INICIO_REGRA 7 – Persistência Dados*/
35 q.execute (“INSERT INTO pedidos VALUES (“ +
36 pedido.Id  +”,”  +  pedido.preco  +  “,”  +
37 pedido.produto + “)”);
38 /* @FIM_REGRA 7*/
39 . . . . . . . . . . . . .
40 /* MAIS CÓDIGO AQUI  */
41  . . . . . . . . . . .
```

Figura 4.5: Listagem com exemplo de código fonte modificado, gerando trace de eventos modificados.

A instrução adicionada ao código fonte é responsável pela gravação dos eventos no *log* e é basicamente composta por uma chamada de procedimento que encapsulada a persistência do evento em um arquivo ou banco de dados. Nas figuras acima, um procedimento chamado *EscreveLog* foi utilizado para representar a instrução de gravação do *log*. Como parâmetros de entrada do procedimento, são esperados no mínimo 3 informações: i) o usuário ou aplicação que executou a regra de negócio, ii) o identificador da regra de negócio e iii) o identificador do *trace* de execução do processo. O primeiro parâmetro é responsável por gravar qual o participante que executou a regra de negócio. Quando interações humanas são necessárias no sistema (ex: preencher um formulário de pedido), estas são executadas por usuários (participante humano) da organização. Quando uma tarefa não é realizada por um usuário e sim por um sistema, então este parâmetro recebe o nome do sistema (participante sistêmico) que executou a regra de negócio. O segundo parâmetro recebe o identificador da regra de negócio (ex: BR1) que foi executada. Este identificador foi utilizado para compor o nome da atividade dentro do modelo de processo de negócio. O terceiro e último parâmetro recebido pelo procedimento é o identificador do *trace* ou caso do processo. Este identificador permite agrupar os diversos eventos no *log* a fim de definir a qual instância de processo eles pertencem. Para definir o escopo de um *trace* de execução, várias abordagens podem ser definidas, dependendo do tipo e complexidade das aplicações envolvidas na execução do processo. Duas possíveis abordagens são apresentadas aqui para definir o identificador do *trace* de execução a partir do código fonte da aplicação:

(i) Identificador formado pelo identificador da instância da aplicação: Um *trace* pode ser basicamente o fluxo de execução de uma instância da aplicação, sendo este *trace* identificado por um valor numérico incremental e único. Assim, todos os eventos gerados a partir da mesma execução do sistema receberão este identificador único referente a instância da aplicação. A restrição quanto a esta abordagem está no fato de alguns processos serem formados pela execução de diversas aplicações, impedindo assim a sua utilização.

(ii) Identificador formado por uma variável comum dentro do processo: Nos casos onde o identificador único da instância da aplicação não é efetivo para representar o identificar do *trace*, é necessário utilizar informações que estão disponíveis durante toda a execução de uma ou mais aplicações. Muitas vezes, as informações que estão presentes no fluxo de dados da aplicação são úteis para esta função. Como exemplo, podemos utilizar um processo de atendimento ao paciente. Neste processo o identificador do paciente pode ser um dado constante e disponível durante toda a execução de uma ou mais aplicações e poderia ser perfeitamente utilizado como identificador do *trace* de execução do processo.

Para definir os possíveis eventos (regras de negócio) que podem ser gerados a partir da execução do sistema de informação, foi definido um conjunto composto por oito regras de negócio. Estas regras cobrem as principais regras que podem ser encontradas no código fonte de um sistema. Este conjunto de regras de negócio é baseado no conjunto definido por (WEIDEN; HERMANS; SCHREIBER, 2002). Para uma melhor compreensão destas regras, o conjunto de regras de negócio foi dividido neste trabalho em dois grupos, os quais são apresentados na Tabela 4.1. O primeiro grupo representa as regras de negócio do tipo unidade de trabalho (entidade *UnidadeTrabalhoBR* no modelo conceitual de processo). Elas representam unidades de trabalho que são indivisíveis (processamento atômico dentro de uma aplicação), e que devem ser

executados em uma única transação. O segundo grupo representa as regras de negócio de controle de fluxo (entidade *ControleFluxoBR* do modelo conceitual de processo). Estas regras são condições que controlam o fluxo de execução de um processo de negócio. Elas representam expressões condicionais associadas a transições e laços e que geralmente controlam a transição e execução de uma ou mais regras de negócios do tipo unidade de trabalho.

Além de listar os tipos de regras de negócio, a Tabela 4.1 ainda apresenta alguns detalhes sobre cada regra, como o nome, descrição resumida da regra e um pseudocódigo utilizado como exemplo da regra. É importante ser enfatizado que algumas regras de negócio não são anotadas no código fonte da aplicação, mesmo elas possuindo um pseudocódigo de exemplo. Isto pode ser observado no código fonte de exemplo da Figura 4.4. As regras de negócio de controle de fluxo não são anotadas no código fonte (ver linhas 15, 20 e 27). Isto se deve ao fato do próprio algoritmo de mineração realizar esta descoberta. Esta característica pode trazer benefícios durante a mineração do processo. A primeira delas é a possibilidade da descoberta de regras de controle de fluxo que geralemente não estão presentes de forma explícita no código fonte, como controles de fluxos paralelos (AND-*split*), por exemplo. O segunda vantagem está associada a complexidade a qual uma linguagem pode representar o controle de fluxo. Linguagens mais modernas podem representar um controle de fluxo através de construções bem definidas e estruturadas, como por exemplo *If*, *While*, *For*, *RepeatUntil*, etc. Já linguagens mais antidas, podem representar controles de fluxo como construções mais primitivas, como o comando *Goto*, por exemplo. Desta forma, passando a responsabilidade para o algoritmo de mineração, é possível tratar estes diversos tipos de comandos de desvio de uma forma homogênea e transparente. Além disso, nem sempre um controle de fluxo presente no código fonte de um sistema representa um controle de fluxo em um modelo de processo. Muitas vezes um controle de fluxo no código fonte pode simplesmente ser parte de uma regra de negócio atômica e indivisível do tipo unidade de trabalho, como pode ser visto nas regras de Pré-Processamento e Pós-Processamento apresentadas na Tabela 4.1.

Tabela 4.1: Taxonomia de Regras com Exemplos

Tipo de Regra de Negócio	Descrição
Regra de Negócio de Unidade de Trabalho	
Cálculo Matemático ou Aritmético	Representa um cálculo matemático. Ela pode ser uma simples expressão matemática bem como um cálculo complexo envolvendo diversas linhas de código. Ex: *Discount = (price * 20) / 100;*
Chamada de Função	Representa uma chamada de função externa no código fonte. Uma função externa é geralmente uma caixa preta, onde o código fonte não é disponibilizado ou não é considerado em um primeiro momento. Ex: *ret=InvokeCreditAnalysis(customer);*
Chamada de Procedimento	Representa uma chamada de procedimento externo no código fonte. Um procedimento externo é geralmente uma caixa preta, onde o código fonte não é disponibilizado ou não é considerado em um primeiro momento. Ex: *InvokePrintReport(order);*

Tipo de Regra de Negócio	Descrição
Regra de Negócio de Unidade de Trabalho	
Persistência de Dados	Representa comandos associados a manipulação de dados. Ela pode ser um comando SQL ou uma operação E/S em um arquivo. Ex: *sql = “SELECT * FROM customer WHERE name = “ + customer_name;*

Interação com Usuário	Representa um formulário de entrada e saída de dados, onde o usuário interage com um sistema. Ex:
	printf("*Informe o nome/endereço do cliente: *");
	scanf("%s", customer_name);
	scanf("%s", customer_address);

Pré-Processamento	Representa uma pré-condição para uma validação de dados, geralmente ocorre após a entrada de dados em um formulário. Ex:
	/* quantidade foi informada anteriormente */
	IF (quantity <= 0) {
	printf("Valor da quantidade inválido!");
	return(0);
	}

Pós-Processamento	Similar à regra de Pré-processamento. A diferença é a origem dos dados. A regra geralmente aparece após um processamento interno do sistema. Ex.:
	tax = (total * tax_percent) / 100;
	/* pós-processamento inicia aqui */
	IF (tax <= 0) {
	printf("O valor da taxa é inválido!");
	exit(0);
	}

Regra de Negócio de Controle de Fluxo	

Controle de Fluxo	Representa um comando de controle de fluxo. Ela pode ser uma decisão condicional (e.g. *if-then-else, switch*, etc), um laço condicional (ex: *while*), um controle de exceção (ex: *try-catch-finally*), etc. Ex:
	IF (approved && price > 1000) {
	// Outras regras de negócio aqui
	}

4.2 Definição de Cenários de Execução dos Usuários

O primeiro passo do processo de mineração incremental é responsável por identificar os cenários de execução necessários para realizar a execução coordenada de um sistema de informação. Em outras palavras, a execução coordenada define quais cenários do sistema precisam ser executados para a geração do *log* do processo. Um cenário aqui é definido basicamente como o conjunto de funcionalidades executadas por um ou mais usuários. Geralmente, um cenário de usuário está associado a atividades executadas por um participante humano e estão relacionadas à interface de usuário de uma aplicação, como por exemplo o preenchimento de um formulário de entradas de

dados. Neste trabalho estas atividades foram mapeadas para regras de negócio de interação com usuário as quais representam atividades manuais executadas por um usuário (ver regra *Interação com Usuário* na Tabela 4.1). Na Figura 4.6, podemos ver um exemplo de interação do usuário através de um cenário de execução do sistema. Podemos ver a usuária Mary executando o *Cenário A*. Neste cenário Mary executa o menu *Preecher Pedido* de uma aplicação, o qual instancia a regra de interação Preencher Pedido (BR1 – *Preencher Pedido*). Esta regra, conseguentemente é seguida pela execução de outras regras no fluxo da aplicação, como regras de persistência, por exemplo, formando um *trace* de eventos que será registrado no *log*.

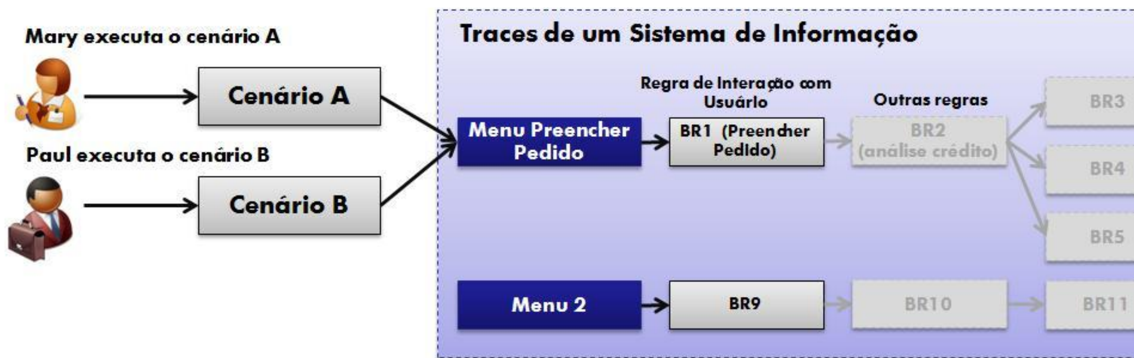


Figura 4.6: Interação do usuário com os cenários de execução do sistema de informação

A utilização de cenários permite extrair do sistema funcionalidades que são importantes para um determinado processo de negócio ou módulo do sistema. Esta característica pode ser importante quando utilizamos o processo de mineração para extrair processos de forma incremental a partir de sistemas de informação. Quando pensamos na utilização da mineração no processo de reengenharia de um sistema legado, fica evidente que não é possível extrair os processos em um único passo. Um sistema pode implementar dezenas de processos complexos os quais podem gerar milhares de eventos. Neste caso, é adequado utilizar uma abordagem incremental com cenários onde apenas parte do sistema (ex: um módulo ou usuário específico) é considerado inicialmente para a extração de modelos de processos parciais e então extrair o processo completo ou os demais processos relacionados em uma etapa posterior.

Tabela 4.2: Exemplos de cenários de execução dos usuários do sistema apresentado na Figura 4.4Figura

Usuário	Cenário	Menu a Executar	Parâmetros Entrada
Mary	Cenário A	Preencher Pedido	*Pedido*: [Produto1; R$ 10,00; Cliente1]
Paul	Cenário B	Preencher Pedido	*Pedido*: [Produto2; R$ 60,00; Cliente2]
Mary	Cenário A	Preencher Pedido	*Pedido*: [Produto3; R$ 40,00; Cliente3]
Paul	Cenário B	Preencher Pedido	*Pedido*: [Produto1; R$ 10,00; Cliente2]

Como foi dito no início desta seção, cenários de execução definem como um ou mais usuários interagem com um sistema. Desta forma, na maioria das vezes, para compor um cenário, é necessário apenas identificar como o usuário utiliza um ou mais sistemas e quais informações de entrada ele fornecerá, como por exemplo o preenchimento de um pedido de compra (regra de negócio de interação). Usando novamente o exemplo de aplicação da Figura 4.4. Podemos observar nas linhas 6-9 da listagem a criação de um formulário de entrada de dados onde alguns dados são

solicitados. O acesso a este formulário se dá geralmente através do acesso a uma opção de menu. Por exemplo, quando a usuária Mary acessar o menu *Preencher Pedido*, o formulário será aberto e ela deverá informar os dados do pedido (dados presentes no cenário definido na linha 1 da Tabela 4.2). Desta forma, os dados solicitados neste formulário de pedido (ex: cliente, preço e produto) são referentes a regra de negócio de interação que está associada ao trecho de código das linhas 6-9 da Figura 4.4.

Os cenários definidos acima são exemplos simples onde apenas um usuário interage simulataneamente em um trace de processo (Mary ou Paul). Em aplicações mais complexas, é possível encontrar casos onde dois ou mais usuários colaboram no mesmo fluxo de uma instância do processo. Nestes casos, um processo representa um fluxo de atividades onde diversas pessoas colaboram para a conclusão de uma demanda de negócio, formando assim um processo colaborativo.

4.3 Execução dos Cenários

Nesta etapa do processo, o sistema de informação é executado de acordo com os cenários de usuários definidos no passo anterior (ver Tabela 4.2). O objetivo aqui é executar o sistema para a coleta do comportamento dinâmico registrado nos traces de execução. Estes traces são então gravados no *log* de execução da aplicação para posterior mineração. O formato do *log* utilizado neste trabalho segue o metadelo MXML, conforme apresentado na seção 2.6.2. Por questões de simplificação, utilizaremos uma representação gráfica no formato de tabela, conforme apresentado na Figura 4.7.

A execução do sistema pode ser realizada de duas formas: (i) manualmente pelo próprio usuário ou (ii) de forma automática por uma ferramenta de automação de testes, como por exemplo Apache JMeter (HALILI, 2008). Quando realizada de forma manual, o próprio usuário é responsável pela execução do sistema seguindo os cenários de execução definidos. Já quando feita de forma automática, a ferramenta de automação de testes executa a aplicação de acordo com os passos definidos no cenário, sem a necessidade da interação do usuário com o sistema de informação. Neste caso, é possível obter o resultado (log de execução) de forma mais rápida e precisa.

Como é de se esperar de um processo incremental, a execução do sistema pode ser feita gradualmente. Ou seja, o sistema é executado inicialmente para um grupo de cenários e seus usuários, gerando um conjunto de registros no *log* que já podem ser trabalhados pela mineração. Em seguida, podemos executar o sistema novamente, agora para um novo conjunto de usuários, gerando um conjuntos de entradas complementares no *log*. A execução incremental de cenários pode ser representada pela Figura 4.7. Na primeira etapa (Figura 4.7-a), os cenários A e B são executados para os usuários Mary e Paul, gerando o *log* inicial de eventos. Em seguida o cenário C é executado para o usuário Mark (Figura 4.7-b), gerando novas entradas que represetam o *log* complementar. Este log complementar contém muitas vezes novos traces da aplicação, associados a trechos da aplicação que não haviam sido executados anteriormente. É possível verificar também que instâncias alteradas são geradas no log (Figura 4.7-c). Estas intâncias representam a reexecução dos cenários A e C após a modificação da aplicação (conforme código apresentado na Figura 4.5) e serão detalhadas na seção 4.4.4 deste capítulo.

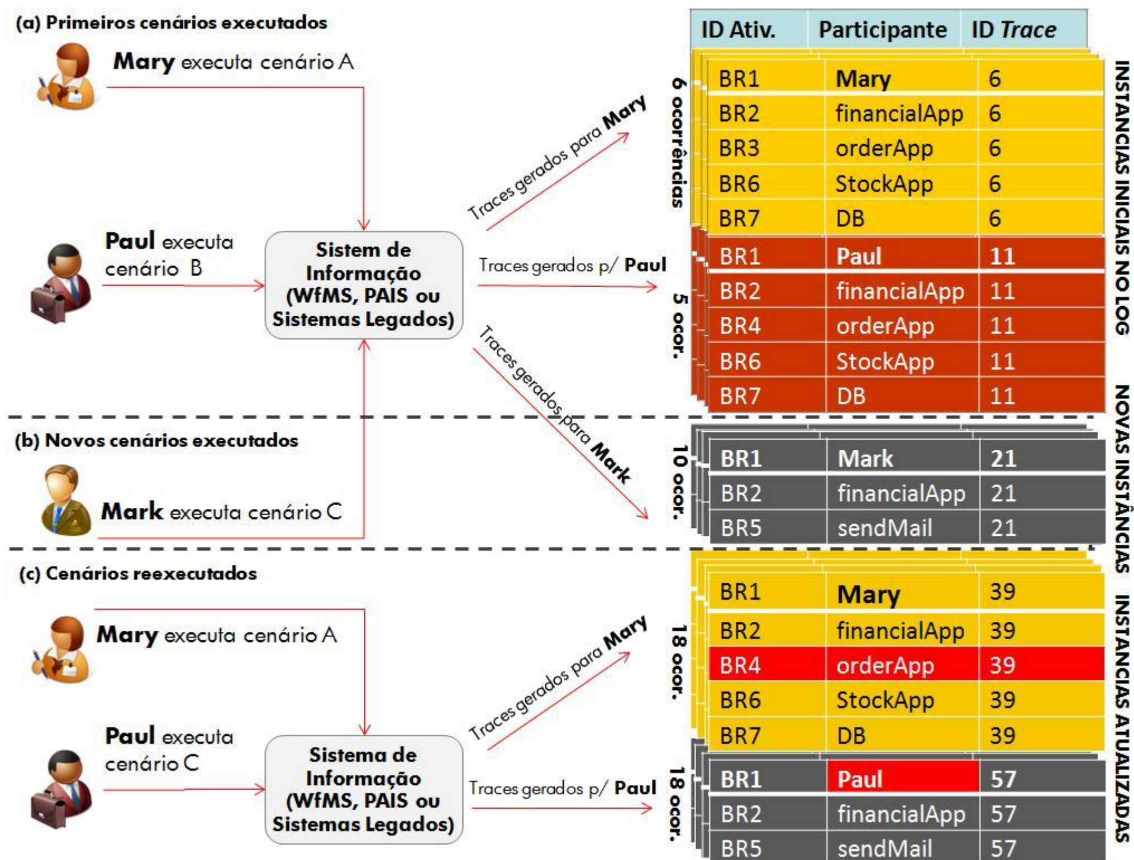


Figura 4.7: *Log* resultante da execução dos cenários de usuário.

Como explicado anteriormente, cada entrada no *log* apresentado na figura acima representa um evento gerado pela execução de uma regra de negócio no código fonte de um sistema de informação. Neste caso, o código fonte apresentado na Figura e Figura 4.5. Cada regra de negócio executada e gravada no log é composta por três informações: identificador da regra, participante e o identificador do trace (instância do processo). O identificador da regra é representado pela coluna *Rule* ID. Ela representa o identificador único de uma regra de negócio no código fonte e que representará posteriormente uma atividade única dentro do processo de negócio. A coluna Participante representa o ator que executou a regra de negócio. Neste caso, um participante poderá ser humano ou um sistema. Quando uma regra de negócio do tipo Interação com Usuário é executada, esta é sempre executada por um usuário do sistema, o qual representa um participante humano. Quando uma regra de negócio é executada por um sistema, agente ou *script*, este representa sempre um participante sistêmico. A última coluna representa o identificador único da instância da aplicação (*Trace* ID). Este identificador permite agrupar todas as regras de negócio que são executadas dentro do mesmo trace de execução. Desta forma, todas as instâncias de regras de negócio que possuem o mesmo *Trace* ID irão pertencer consequentemente à mesma instância do processo.

As informações contidas no *log* de execução definidas acima, são o mínimo necessário para o correto comportamento do algoritmo de mineração criado para a descoberta do processo a partir do *log*. Com estas informações, podemos extrair a estrutura do processo representado pelo comportamento registrado no *log*. Os detalhes da mineração deste *log* serão discutidos na próxima seção deste capítulo.

4.4 Mineração do *Log* de Execução

O principal passo dentro do processo de mineração incremental definido no início deste capítulo é a mineração do *log* de execução do sistema informação. Nesta etapa, a extrutura do processo é extraída a partir do *log*. Esta extração é realizada pelo algoritmo *IncrementalMiner* (KALSING; IOCHPE; THOM, 2010a), um algoritmo incremental de mineração de processos que extrai as estruturas do processo a partir de um log de execução. O algoritmo utiliza várias heurísticas para a mineração das diversas relações entre os eventos apresentados no *log*. Ao final, um grafo dirigido representa a estrutura do processo de negócio, juntamente com os aspectos semânticos do processo (ex: XOR/AND-split/join).

Como dito no início deste trabalho, a decisão em utilizar algoritmos de mineração de processo foi feita com base no fato destes algoritmos suportarem a extração das principais estruturas do processo de negócio a partir do *log* (ex: XOR/AND-split/join, participantes, laços, etc). Além disso, o algoritmo *IncrementalMiner* foi criado para suportar a extração incremental do processo, suportando a adição e remoção de elementos do modelo a medida que novos registros são disponibilizados no log. Assim, o modelo de processo de negócios extraído a partir do *log* pode ser criado e atualizado cada vez que novas entradas são geradas no *log* pela execução de um ou mais sistemas de informação.

A mineração incremental de um processo de negócio pode ser dividida em dois tipos: i) mineração incremental para complementação de um modelo parcial e ii) mineração incremental para atualização de um modelo de processo completo. A mineração incremental para complementação de um modelo parcial ou incompleto é realizada sobre um *log* de processo que inicialmente não continha todos os eventos necessários para geração de um modelo completo. Esta situação é comumente encontrada durante a mineração de processos que necessitam de milhares de eventos para geração do processo, mas devido a uma necessidade de negócio deseja extrair um modelo parcial para visualização do usuário. O segundo tipo de mineração trata da atualização de um modelo de processo já completo. Neste caso, instâncias de processos modificadas são registradas no log, geradas a partir de atualizações nas regras, estrutura ou fluxo de dados de um processo de negócio.

O algoritmo proposto no processo incremental de mineração deste trabalho possui a habilidade de realizar a mineração incremental de ambos tipos, sendo capaz de manter o modelo atualizado mesmo após alterações realizadas na implementação do processo. Este pode ser considerado o ponto central deste trabalho, pois introduz o mecanismo capaz de tornar possível a extração incremental do processo a partir de um sistema de informação. Os detalhes do algoritmo e como ele realiza a mineração incremental serão apresentados ao longo desta seção.

4.4.1 Operações de atualização

Diversos tipos de alterações podem ser realizadas em um modelo de processo. A Tabela 4.3 apresenta graficamente as possíveis operações de atualização que podem realizadas em um processo de negócio. Estas alterações podem ser geradas de diversas maneiras, como por exemplo a execução de um novo cenário de usuário, a alteração ou remoção de um cenário existente, ou pela adição, alteração ou remoção de regras de negócio de um processo. Abaixo, é listado com mais detalhes cada uma das principais possíveis alterações no modelo de processo:

i)***Adição de uma tarefa*** (Tabela 4.3-a): uma nova tarefa é inserida em um determinado ponto do processo. Esta alteração ocorre devido a adição de um novo cenário ou mesmo a adição de novas regras ao processo. A adição de uma tarefa ao modelo não ocorre sozinha. Além da tarefa, é adicionado também o ator que a executou e um elemento de transição entre a atividade adicionada e a atividade anterior no fluxo.

ii) ***Adição de um ator*** (Tabela 4.3-b): um ator é adicionado a uma tarefa existente ou a uma nova tarefa do processo. A adição de um ator é decorrente da adição de novos cenários de execução de um processo.

iii) ***Adição de controle de fluxo*** (Tabela 4.3-c): uma atividade do tipo controle de fluxo é inserida em um ponto do processo. Esta atividade pode ser do tipo controle de fluxo AND ou OR exclusivo. Geralmente a adição de um controle de fluxo é seguido da adição de um conjunto de novas transições que ligam as demais atividades existentes ao controle de fluxo. A adição de um controle de fluxo pode ser consequência da adição de uma regra de negócio do tipo controle de fluxo ao processo ou pela alteração de um cenário de execução existente.

iv) ***Adição de uma transição*** (Tabela 4.3-d): uma nova transição é inserida entre duas atividades em um processo de negócio. Esta alteração é requerida quando ocorre a adição de uma tarefa ou controle de fluxo ao processo ou simplesmente quando duas atividades existentes no modelo que não eram executadas em sequência agora são.

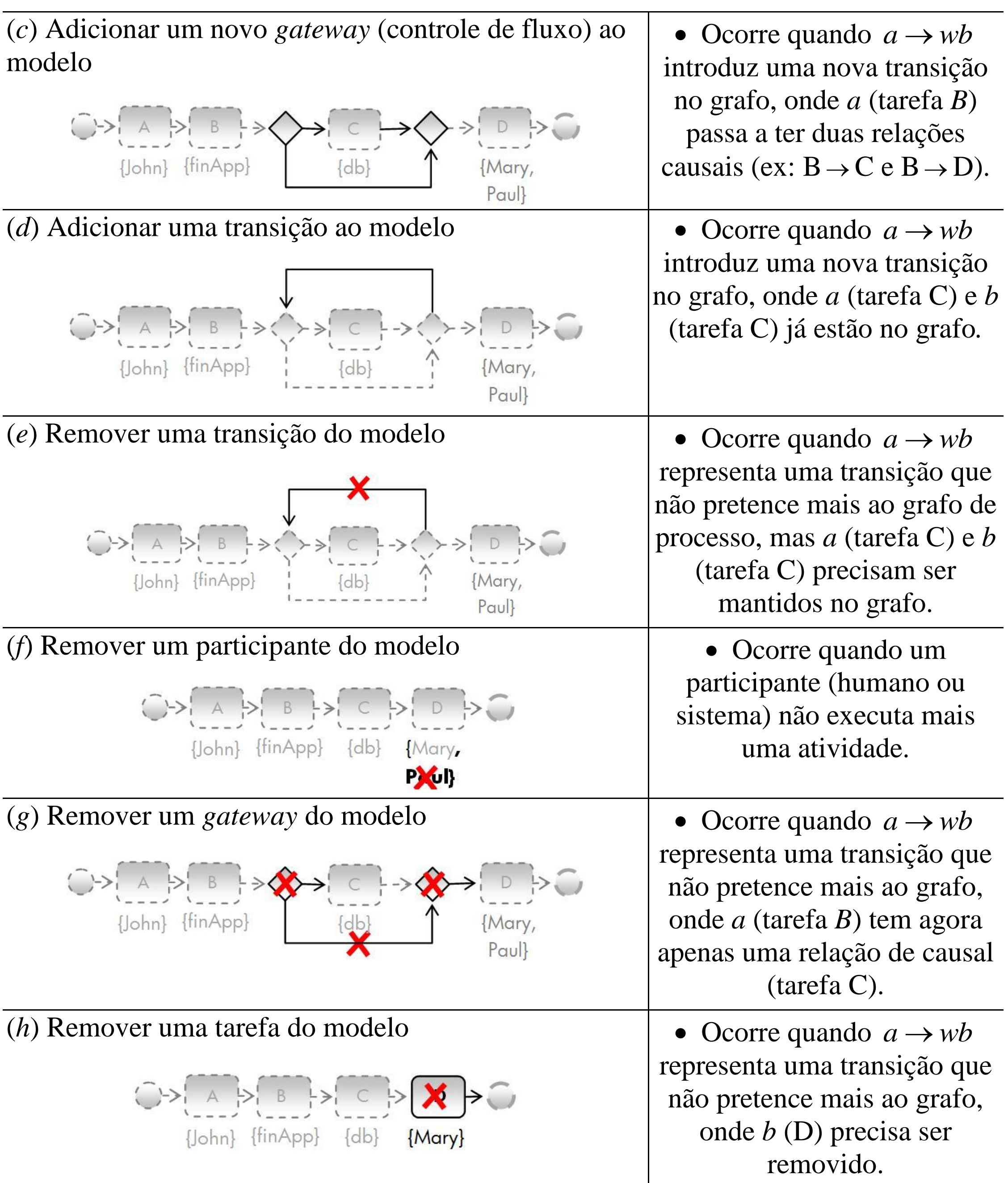
v) ***Remoção de uma transição*** (Tabela 4.3-e): a remoção de uma transição geralmente está associada a mudança de um controle de fluxo do processo. Associada a remoção de uma transição, pode estar também a remoção de uma atividade.

vi) ***Remoção de um ator*** (Tabela 4.3-f): um ator é removido do processo de negócio. A remoção geralmente ocorre quando um ator não executa mais uma determinada atividade no processo ou quando uma tarefa que era executada por este ator foi removida do modelo de processo (ex: Tabela -h).

vii) ***Remoção de uma atividade*** (Tabela 4.3-g e h): a remoção de uma atividade pode estar associada a remoção de uma tarefa ou controle de fluxo do processo. Associada a remoção de uma atividade pode estar também a remoção de uma ou mais transições do modelo que estavam associadas à atividade removida. A remoção de um atividade ocorre quando determinada regra de negócio não é mais executada no processo.

Tabela 4.3: Diferentes tipos de atualização de um modelo de processo durante a mineração incremental.

Operação	Quando Ocorre
(*a*) Adicionar uma nova tarefa ao modelo A → B → C → **D** {John} {finApp} {db} **{Mary}**	• Ocorre quando $a \rightarrow wb$ introduz uma nova transição no grafo, onde *b* (tarefa D) é uma nova tarefa.
(*b*) Adicionar um novo participante ao modelo A → B → C → D {John} {finApp} {db} {Mary, **Paul}**	• Ocorre quando um participante (humano ou sistema) começa a executar uma nova tarefa.



(*c*) Adicionar um novo *gateway* (controle de fluxo) ao modelo	• Ocorre quando $a \rightarrow wb$ introduz uma nova transição no grafo, onde *a* (tarefa *B*) passa a ter duas relações causais (ex: B→C e B→D).
(*d*) Adicionar uma transição ao modelo	• Ocorre quando $a \rightarrow wb$ introduz uma nova transição no grafo, onde *a* (tarefa C) e *b* (tarefa C) já estão no grafo.
(*e*) Remover uma transição do modelo	• Ocorre quando $a \rightarrow wb$ representa uma transição que não pretence mais ao grafo de processo, mas *a* (tarefa C) e *b* (tarefa C) precisam ser mantidos no grafo.
(*f*) Remover um participante do modelo	• Ocorre quando um participante (humano ou sistema) não executa mais uma atividade.
(*g*) Remover um *gateway* do modelo	• Ocorre quando $a \rightarrow wb$ representa uma transição que não pretence mais ao grafo, onde *a* (tarefa *B*) tem agora apenas uma relação de causal (tarefa C).
(*h*) Remover uma tarefa do modelo	• Ocorre quando $a \rightarrow wb$ representa uma transição que não pretence mais ao grafo, onde *b* (D) precisa ser removido.

4.4.2 Definição do Algoritmo

O algoritmo *IncrementalMiner* utiliza três informações básicas do *log* para realizar a mineração: *Id* da Atividade, Participante e o *Id* do *trace* (conforme apresentado na Figura 4.7). Cada evento no *log* refere-se a uma atividade no processo e cada evento pertence a uma instância de processo. Um log de eventos pode ser definido como: Seja *T* um conjunto de atividades de processo. $\sigma \in T^*$ é um *trace* de eventos que consistem de uma sequência arbitrária de identificadores de atividade. $W \subseteq T^*$ é um *log* de eventos que consiste de um conjunto de *traces* de eventos, onde cada *trace* de evento pode aparecer diversas vezes no *log*.

Algoritmo 1. *IncrementalMiner*
Entrada w: Log, g_1: *GrafoDep* ***Saída*** g_2: *GrafoDep*

1. $g_2 \leftarrow g_1$
2. ***Para cada*** nova instancia $\sigma \in W$
   (a) ***Para cada*** evento $e_i \in \sigma$
      (i) $e_1 \leftarrow e_i$
      (ii) $e_2 \leftarrow e_{i+1}$
      (iii) $e_3 \leftarrow e_{i+2}$
      (iv) ***ProcessaRelacao***(e_1, e_2, e_3)
      (v) ***VerificaRelacoesObsoletas*** (e_1)
      (vi) ***ProcessaParticipante***(e_1)

Algoritmo 2. *ProcessaRelacao*
Entrada e_1, e_2, e_3: evento

1. $r_{12} \leftarrow$ ***CriarRelacao***(e_1, e_2).
2. *conf* $\leftarrow a \Rightarrow wb$, onde $a = e_1$ e $b = e_2$.
3. ***Se*** $e_1 = e_2$ então
   (a) *loop1* $\leftarrow a \Rightarrow wa$, onde $a = e_1$ e $a = e_2$
   (b) ***Se*** *loop1 > LOOP1_THSLD* então
      (i) ***AdicionaRelacaoGrafo***(e_1, e_2).
4. ***Senão*** **Se** $e_1 = e_3$ então
   (a) *loop2* $\leftarrow a \Rightarrow 2wb$, onde $a = e_1$ e $b = e_3$.
   (b) ***Se*** *loop2 > LOOP2_THSLD* então
      (i) ***AdicionaRelacaoGrafo***(e_3, e_1).
5. $n_1 \leftarrow$***AtualizaArvoreDepend***(r_{12}, *conf*)
6. *cand* $\leftarrow a \Rightarrow wb \wedge c$, onde $a=e_1, b=e_2$ e $c=e_3$
7. a*md*←obtém a árvore das melhores relações de e_1
8. a*mc*← obtém a árvore das melhores causas de e_2
9. ***AtualizarMelhorRel***(n_1, a*md*)
10. ***AtualizarMelhorRel*** (n_1, *amc*)
11. $n_3 \leftarrow$ obtém a raiz da árvore *amd*
12. $n_1 \leftarrow$ *amd.obtemNodo*(e_2)
13. $n_2 \leftarrow$ *amc.obtemNodo*(e_1)
14. ***Se*** ($\neg(n_1 = \emptyset) \vee \neg(n_2 = \emptyset) \vee$ *conf* > *DEPENDENCY_THSLD*) $\wedge$ ***suporte***(r_{12}) > *POSITIVEOBSERVATION_THSLD* $\wedge$ *conf* – n_3.*conf* $\leq$ *RELATIVETOBEST_THSLD* então
   (a) ***AdicionaRelacaoGrafo***(e_1, e_2).
15. ***CalculaRelacaoIrmaObsoleta***(e_1, e_2).

Algoritmo 3. *AtualizarMelhorRel*
Entrada n_1: nodo, a_1: arvore

1. *valor*← ***VerificarConfianca***(n_1.conf, a_1)
2. ***Se*** *valor = -1 então //valor menor*
   (*a*) ***RemoveRelacaoDoGrafo***(n_1.*relacao*).
3. **Senão** ***Se*** *valor = 1* então *//valor maior*
   (*a*) ***Para cada*** nó $n_2 \in a_1$
      (*i*) Remover n_2 da a_1
      (*ii*) ***RemoveRelacaoDoGrafo***(e_1, e_2)
   (*b*) Adicionar n_1 à a_1
4. ***Senão Se*** *valor = 0* então *//valor igual*
   (a) Adicionar n_1 à a_1

Algoritmo 4. *AdicionaRelacaoAoGrafoDep*
Entrada e_1, e_2: evento

1. $ve_1 \leftarrow$ obém vértice para e_1 do grafo g_2
2. $ve_2 \leftarrow$ obém vértice para e_2 do grafo g_2
3. *aresta*$_1$←obtém aresta{ve_1, ve_2} *no grafo* g_2
4. ***Se*** $ve_1=\emptyset$ então
   (*a*) $ve_1 \leftarrow$ Cria novo vértice para evento e_1
   (*b*) Adiciona ve_1 *ao grafo* g_2
5. ***Se*** $ve_2=\emptyset$ então
   (*a*) $ve_2 \leftarrow$ Cria novo vértice para evento e_2
   (*b*) Adiciona ve_2 *ao grafo* g_2
6. ***Se*** *aresta*$_1=\emptyset$
   (*a*) *aresta*$_1$← {ve_1, ve_2} *//cria aresta*
   (*b*) Adicionar *aresta*$_1$ *ao grafo* g_2
7. ***CriarRelacaoIrma***(e_1, e_2)

Algoritmo 5. *RemoveRelacaoDoGrafo*
Entrada e_1, e_2: evento

1. ve_1←obém vértice para e_1 do grafo g_2
2. ve_2←obém vértice para e_2 do grafo g_2
3. Remover aresta entre ve_1 e ve_2 do grafo g_2
4. out_{ve1}←obtém vértices de saída de ve_1
5. out_{ve2}←obtém vértices de saída de ve_2
6. in_{ve1}←obtém vértices de entrada de ve_1
7. in_{ve2}←obtém vértices de entrada de ve_2
8. ***Se*** $out_{ve1} = \emptyset \wedge in_{ve1} = \emptyset$ então
   (a) Remover ve_1 do *grafo* g_2
9. ***Se*** $out_{ve2} = \emptyset \wedge in_{ve2} = \emptyset$ então
   (a) Remover ve_2 do *grafo* g_2

Algoritmo 6. *VerificaRelObsoletas*
Entrada e_1: evento

1. *melhoresRel*←∅ *//cria conjunto temporário*
2. *pioresRel*← ∅ *//cria conjunto temporário*
3. ve_1←obém o vértice para e_1 do grafo g_2
4. out_{ve1}←obtém vértices de saída de ve_1
5. ***Para cada*** vertice $v_i \in out_{ve1}$
   (*a*) $r_{12} \leftarrow$ ***ObtemRelacao***(e_1, v_i.*evento*).
   (*b*) sib_{vi}←obtém relações irmãs de v_i
   (*c*) ***Para cada*** relação irmã $ri_1 \in sib_{vi}$
      (*i*) ***Se*** ri_1.conf $\geq$ *OBSOL_THSLD* então
         (x) Adicionar ri_1 a *melhoresRel*
      (*ii*) ***Senão***
         (x) Adicionar ri_1 a *pioresRel*
   (*d*) ***Se*** $sib_{vi} \subseteq$ *melhoresRel então*
      (*i*) ***RemoveRelacaoDoGrafo***(r_{12})
   (*e*) ***Senão***
      (*i*) ***Para cada*** *relação* $r_1 \in$ *pioresRel*
         (*x*) ***Para cada*** *relação irmã* $ri_1 \in r_1$
            a. ***Se*** *ri.conf>OBSOL_THSLD então*

i. *excluir*$\leftarrow$ $ri_1 \in$ *melhoresRel* $\wedge$ *excluir*
(ii) ***Se*** $\neg$ (*melhoresRel*=∅) $\wedge$ *excluir=true então*
(x) ***Para cada*** *relação* $r_1 \in$ *pioresRel*
a. ***RemoveRelDoGrafo***(r_1)
(*y*) ***RemoveRelDoGrafo***(r_{12})

Algoritmo 7. *CalculaRelacaoIrmaObsoleta*
Entrada e_1, e_2: evento

1. ve_1←obém vértice para e_1 do grafo g_2
2. r_{12} ← ***ObtemRelacao***(e_1, e_2).
3. sib_r←obtém conjunto rel. irmãs de r_{12}
4. ***Para cada*** relação irmã $ri_1 \in sib_r$
   (*a*) ***Se*** $ri_1 = r_{12}$ *então*
   (*i*) *conf*← $a \Rightarrow wb \mapsto c$, onde $a > wb$ = suporte de r_{12}, $a > wc$ = *suporte* de *r* e $[a \rightarrow wc]_{a \mapsto wb}$ = suporte de ri_1
   (*ii*) ri_1.*conf*←*conf*

Algoritmo 8. *AtualizarArvoreDependencia*
Entrada *r*: relação, *c*: Número ***Saída*** n_1: nodo

1. a_1←árvore relações dependência de *r.*e_1
2. n_1← a_1.*obtemNodo*(*r.*e_2) *//binary search*
3. ***Se*** n_1 *não existe então*
   (*a*) n_1←***CriarNodo***(*r, c*)
   (*b*) adicionar n_1 à a_1
4. n_1.*suporte*← n_1.*suporte + 1*
5. n_1.*confianca*← *c*

Algoritmo 9. *VerificarConfianca*
Entrada conf: *Numero*, a_1: Arvore ***Saída: v:*** *Numero*

1. n_1←obtém a raiz de a_1
2. ***Se*** *conf* > n_1.*conf então*
   (*a*) *v*← *1*
3. ***Senão Se*** n_1.*conf = conf então*
   (*b*) *v*← *0*
4. ***Senão***
   (*c*) *v*← *-1*

Algoritmo 10. *CriarNodo*
Entrada* r**: relação, **c**: número, **s**: suporte ***Saída n_1*:*nodo

1. n_1←*criar nodo*
2. n_1.*conf*←*c*
3. n_1.*suporte*←*s*
4. n_1.*relacao*←*r*

Algoritmo 11. *CriarRelacaoIrma*
Entrada e_1, e_2: evento

1. ve_1← obém vértice para e_1 do grafo g_2
2. ve_2← obém vértice para e_2 do grafo g_2
2. out_{e1}←obtém vértices de saída de ve_1
3. ***Para cada*** *vértice* $v_s \in out_{ve1}$
   (*a*) r_1←**ObtemRelacao**(e_1, v_s)
   (*b*) sib_1←obtém relações irmãs de r_1
   (*c*) ri_1← cria relação irmã entre e_1 e e_2
   (*d*) ***Se*** $ri_1 \notin sib_1$
   (*ii*) *Adiciona* ri_1 *à* sib_1

Algoritmo 12. *ObtemRelacao*
Entrada e_1, e_2: evento ***Saida r***: relação

1. a_1←árvore relações dependência de e_1
2. n_1← a_1.*obtemNodo*(e_2) *//binary search*
3. *r*← n_1.*relacao*

Algoritmo 13. *ProcessaParticipante*
Entrada e_1

1. ve_1←obtém vertice para e_1 do grafo g_2
2. p_1← e_1.*participante*
3. **Se** $\neg$ $p_1 \in ve_1$.*participantes*
   (*a*) Adiciona p_1 *em* ve_1.*participantes*

Figura 4.8: Pseudocódigo do algoritmo *IncrementalMiner*

Para ser possível a identificação de um modelo de processo a partir de um *log* de eventos, este *log* precisa ser analisado em busca de dependências causais. Ou seja, se uma atividade é sempre seguida por outra, é provável que exista uma relação de dependência entre ambas atividades. Para analisar estas relações, serão apresentadas abaixo as notações que descrevem todas as relações de dependência utilizadas pelo algoritmo *IncrementalMiner*. Estas notações foram introduzidas inicialmente por Weijters et al (WEIJTERS; AALST; MEDEIROS, 2006) e estendidas neste trabalho para contemplar a mineração incremental. Assim, seja *W* uma *log* de eventos sobre *T*, representado por $W \subseteq T^*$ e seja $a, b \in T$ então*:*

1*)* $a > wb$ se e somente se existe um *trace* $\sigma = t_1t_2t_3\ldots t_n$ e $i \in \{1, ..., n\text{-}1\}$ tal que $\sigma \in W$ e $t_i = a$ e $t_{i+1} = b$. A relação $> w$ nesta notação descreve quais atividades aparecem em sequência no *trace*.

2) $a \rightarrow wb$ se e somente se $a > wb$ e $b \not> w\ a$. A relação $\rightarrow$w representa a relação de dependência direta derivada a partir do *log W*.

3) $a\#wb$ se e somente se $a \not> wb$ e $b \not> wa$. Esta relação define um par de relações onde uma nunca é seguida pela outra.

4) $a \| wb$ se e somente se $a > wb$ e $b > wa$. A relação $\| w$ sugere um potencial paralelismo entre *a* e *b*. Se duas tarefas podem aparecer em sequência e em qualquer ordem, elas provavelmente serão executadas em paralelo.

5) $a >>> wb$ se e somente se existe um *trace* $\sigma = t_1 t_2 t_3 \ldots t_n$ e $i<j$ e $i, j \in \{1, ..., n\}$ tal que $\sigma \in W$ e $t_i = a$ e $t_j = b$. A relação $>>>w$ indica que *b* é sempre seguida por *a* no *log* de eventos *W*.

6) $a >> wb$ se e somente se existe um *trace* $\sigma = t_1 t_2 t_3 \ldots t_n$ e $i \in \{1, ..., n\text{-}2\}$ tal que $\sigma \in W$ e $t_i = a$ e $t_{i+1} = b$ e $t_{i+2} =$ a. Esta relação sugere um laço simples entre *a* e *b*.

Adicionalmente às relações definidas acima, foi necessária a criação de um novo tipo de relação para identificação de relações obsoletas em um *log* que contém atualizações do processo (mineração incremental com atualização do modelo), onde:

7) $[a > wc]_{a \mapsto wb}$ se e somente se existe um *log* de eventos $W = \sigma_1 \sigma_2 \sigma_3 \ldots \sigma_n$ e $i<j$ e $i, j \in \{1, ..., n\text{-}1\}$ e um *trace* $\sigma = t_1 t_2 t_3 \ldots t_n$ e $t_i = a$ e $t_{i+1} =$ (*b* or *c*) e $\sigma_i \subset a > wc$ e $\sigma_j \subset a \mapsto wb$. A relação $\mapsto w$ nesta notação representa a relação irmã da relação $> w$ derivada do *log* de eventos *W*, onde a relação $> w$ é mais antiga no *log* que a relação irmã $\mapsto w$.

O algoritmo de mineração descrito neste capítulo utiliza uma abordagem heurística para extrair as relações de dependências dos logs. Estas heurísticas utilizam métricas baseadas em frequência para indicar a probabilidade de uma relação de dependência entre dois eventos ser verdadeira. O algoritmo principal chama-se *IncrementalMiner* e é apresentado no pseudocódigo da Figura 4.8. O comportamento do algoritmo pode ser dividido em três etapas principais: 1) mineração do grafo de dependência, composto por relações de dependência $> w$, $\| w$ and $>>w$, além da semântica de controle de fluxo, 2) descoberta das relações obsoletas e 3) mineração dos participantes do processo. O primeiro passo descobre todas as relações de dependência entre atividades a partir do log de eventos, tais como seqüências, iterações, paralelismo e controle de fluxos. A descoberta é realizada aplicando as heurísticas definidas no algoritmo 2 *ProcessRelation (*ver item *2.iv* do Algoritmo 1). No passo seguinte, o algoritmo descobre as relações de dependência obsoletas a partir do log. Essas relações representam elementos removidos da definição do processo (processo que gera o log) e que devem ser excluídas do modelo descoberto. Na última etapa do algoritmo IncrementalMiner é realizada a mineração dos participantes do processo. Esta etapa descobre e gerencia o conjunto de todos os participantes que executam determinada tarefa. Este comportamento é descrito pelo algoritmo *ProcessaParticipante*, apresentado ao final do pseudocódigo da Figura 4.8.

Para exemplificar como a mineração do *log* é realizada, consideremos o *log* parcial $W=\{BR1BR2BR3BR6BR7^6, BR1BR2BR4BR6BR7^5\}$, formado pelos *traces* apresentados na Figura 4.7-a. O algoritmo 1 (*IncrementalMiner*) apresentado na Figura 4.8 inicia iterando sobre cada instância de processo e seus eventos presentes no *log*. Cada instância é composta por um conjunto de eventos (ex: um *trace* de execução) que representam as instâncias das regras de negócios executadas. Em cada iteração sobre

uma instância de processo, três eventos (e_1, e_2 e e_3) são submetidos ao algoritmo 2 (*ProcessaRelacao*). Este algoritmo é responsável em aplicar as heurísticas que descobrem as principais relações de dependência entre os eventos do *log*. Assim, considerando o par dos dois primeiros eventos do *log* (BR1 e BR2), a primeira heurística a ser aplicada é a relação de dependência (heurística 1). Esta heurística verifica se a dependência entre dois elementos de um *trace* existe. Seja W um *log* de eventos T, e $a, b \in T$. Então $|a > wb|$ é o número parcial de vezes que $a > wb$ ocorre em W, e:

$$a \Rightarrow wb = \left(\frac{|a > wb| - |b > wa|}{|a > wb| + |b > wa| + 1} \right) \quad (1)$$

A heurística (1) apresentada acima realiza o cálculo da relação de dependência entre dois eventos. É importante notar que esta e as demais heurísticas possuem uma sensível diferença com relação às heurísticas originais apresentadas pelo algoritmo *HeuristicMiner* do autor Weijters et al. Aqui, a notação $|> w|$ foi substituída pela notação $|> w|$, a qual representa um número parcial (incremental) de vezes que determinada relação ocorreu. Desta forma, o resultado do cálculo desta heurística representa a confiança parcial da relação (confiança calculada até o momento neste exemplo), usando o suporte de $a > wb$ (ex: número parcial de vezes que BR1 vem antes de BR2) e o suporte de $b > wa$ (número parcial de vezes que BR2 vem antes de BR1). Neste exemplo, $a \Rightarrow wb =$(11-0) / (11+0+1) = 0.916 é a confiança parcial para a relação BR1→BR2. O resultado desta e de todas as heurísticas utilizadas pelo algoritmo possue sempre variação de valores entre -1 e 1, onde valores que se aproximam de 1 são considerados valores de alta confiança.

O valor calculado pela heurística (1) e todos os dados associados a uma relação de dependência são inseridos pelo algoritmo *ProcessaRelacao* em uma estrutura de dados do algoritmo chamada *Árvore de Dependência* (ver Figura 4.9). A inserção é realizada invocando o algoritmo *AtualizaArvoreDependencia* (ver linha 5). A *Árvore de Dependência* é uma estrutura de dados formada por árvores binárias balanceadas - AVL (ADELSON-VELSKII; LANDIS, 1962). Estas árvores mantém as relações candidatas (ex: relações que poderão ser consideradas no grafo final do processo), juntamente com a confiança e suporte da relação, respectivamente. Cada tipo de evento gerado no *log* possui uma árvore de dependência que agrupa todas as relações derivadas deste evento. A Figura 4.9 mostra as árvores de dependência para a heuristica $a \Rightarrow wb$ calculada acima (ver árvore de dependência BR1 com relação de dependência BR1→BR2 no nó raíz BR2) e para todas as outras relações de dependência presentes no *log*. O algoritmo *IncrementalMiner* utiliza árvores AVL devido a seu tempo satisfatório para as operações de busca, inserção e remoção, que são requeridos pela abordagem incremental de mineração.

As duas próximas heurísticas (heurísticas 2 e 3) utilizadas pelo algoritmo *ProcessaRelação* verificam no *log* a ocorrência de laços simples entre eventos. Ou seja, ela verifica a existência de iterações entre atividades em um *trace*. A Heurística 2 calcula a confiança de laços curtos com tamanho um (ex: laços com apenas uma tarefa, BR1BR1). Já a heurística 3 considera laços de tamanho dois (laços com duas atividades, ex: BR1BR2BR1). Assim, seja *W* um *log* de eventos sobre *T*, e *a, b* ∈ *T*. Então $|a > wa|$

é o número parcial de vezes que $a >wa$ ocorre em *W*, e $|a >> wb|$ é o número parcial de vezes que $a >>_W b$ ocorre em *W*:

$$a \Rightarrow wa = \left(\frac{|a > wa|}{|a > wa| + 1} \right) \quad (2)$$

$$a \Rightarrow 2wb = \left(\frac{|a >> wb| - |b >> wa|}{|a >> wb| + |b >> wa| + 1} \right) \quad (3)$$

A última heurística utilizada pelo algoritmo *ProcessaRelação* verificar a ocorrência de atividades não-observáveis no *log* (ex: *gateways* AND/XOR-split/join). Seja *W* um *log* de eventos sobre *T*, e $a, b, c \in T$, e *b* e *c* são relações de dependência com *a*. Então:

$$a \Rightarrow wb \wedge c = \left(\frac{|b > wc| + |c > wb|}{|a > wb| + |a > wc| + 1} \right) \quad (4)$$

A parte inferior da heuristica (4) é formada pela fórmula $|a > wb| + |a > wc|$ que representa o número parcial de observações positivas e $|b > wc| + |c > wb|$ representa o número parcial de vezes que *b* e *c* aparecem diretamente no *log*, uma após a outra. Considerando o *log* de eventos parcial {BR1BR2BR3…, BR1BR2BR4…}, o valor de $a \Rightarrow wb \wedge c$ = (0 + 0) / (6 + 5 + 1) = 0,0 indica que *BR3* e *BR4* estão em uma relação XOR após BR2. Valores baixos para $a \Rightarrow wb \wedge c$ indicam a existência de uma relação XOR (OR exclusivo) entre as atividades. Já valores altos para esta heurística indicam uma relação paralela entre as atividades do tipo AND. Ou seja, um valor próximo a 1 para a heurística poderia indicar que BR3 e BR4 executam em paralelo.

As heurísticas definidas acima são utilizadas para calcular as relações candidatas que podem ser adicionadas ao grafo de dependência final do processo de negócio (ex: o grafo do processo com as melhores relações, representada na Figura 4.11). Desta forma, para selecionar as relações candidatas que irão compor o grafo de dependência, foi criado uma nova estrutura chamada *Árvore das Melhores Relações* (ver Figura 4.10). Estas árvores mantém as melhores dependências (Figura 4.10-a) e também as melhores relações causais (Figura 4.10-b), ambas extraídas das *Árvores de Dependência* apresentadas na Figura 4.9. Assim como as *Árvores de Dependência*, as *Árvores das Melhores Relações* são atualizadas após a avaliação de cada relação de dependência, calculada pelas heurísticas 1, 2 e 3.

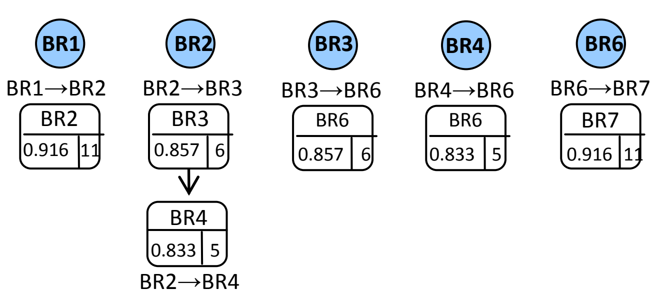


Figura 4.9: Árvores de Dependência. Mantém a confiança e o suporte atualizados para cada relação.

O primeiro passo na atualização das *Árvores das Melhores Relações* é realizado pelo algoritmo *ProcessaRelação* (linhas 9 e 10). O algoritmo *ProcessaRelação* realiza a chamada do algoritmo *AtualizarMelhorRelação* para atualizar a *Árvore das Melhores Dependências* (Figura 4.10-a) para a relação corrente. Considerando a relação corrente do exemplo (BR1→BR2), primeiramente é necessário verificar se o valor da confiança da relação (ex: 0.916) é menor que a confiança das relações na *Árvore das Melhores Dependências* de BR1 (ver a primeira árvore da Figura 4.10-a). Se for menor, é necessário remover a relação BR1→BR2 do *Grafo Final de Dependências* (ex: remover vértices e arestas do grafo da Figura 4.11). A remoção se dá através do algoritmo 5 (*RemoveRelacaoDoGrafo)*. Em contra partida, se a confiança da relação BR1→BR2 for maior que as relações na *Árvore das Melhores Dependências* de BR1, é necessário remover todas as relações antigas da árvore (ex: relações com confiança manor que a da relação corrente). Finalmente, é necessário adicionar a relação corrente ao final da *Árvore Melhores Dependências* de BR1 e por fim adicionar a relação ao *Grafo Final de Dependências* através do algoritmo 4 (*AdicionaRelacaoAoGrafo*). O mesmo processo é seguido para o segundo elemento (BR2). A diferença aqui é a utilização da *Árvore das Melhores Causas* de BR2 (ver a primeira árvore da Figura 4.10-b). Todos os elementos da árvore de dependência foram considerados como melhores dependências neste exemplo simples. Assim, as duas árvores presentes na Figura 4.10-a e Figura 4.9 são muito similares. Desta forma, o grafo de dependência inicial formado pelas melhores relações pode ser visto na Figura 4.11-a.

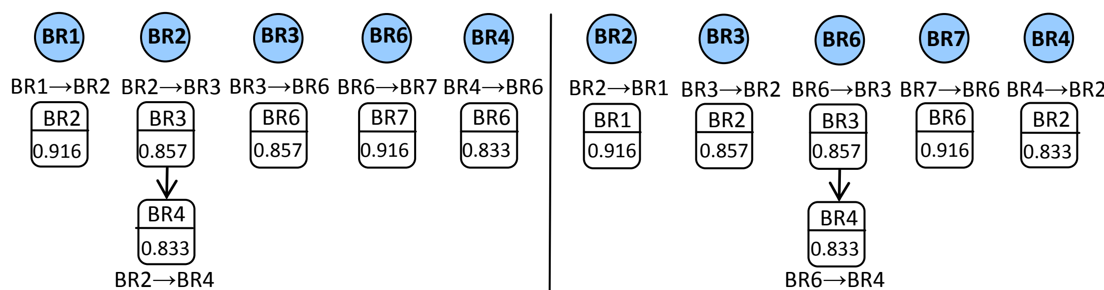
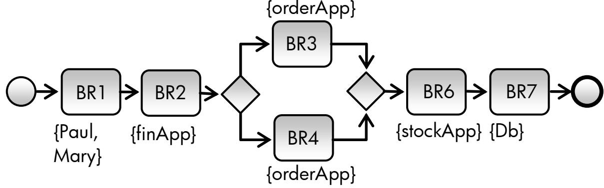
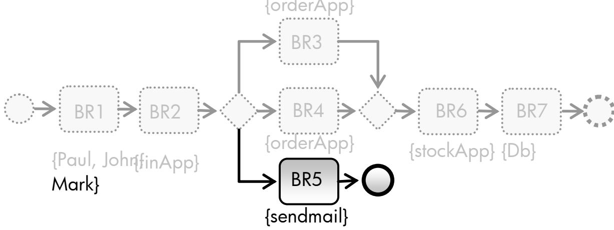


Figura 4.10: Árvores das Melhores Relações extraídas a partir da Árvore de Dependência da Figura 4.9.

As relações apresentadas até este momento são derivadas de eventos disponibilizados no *log* parcial presentes na Figura 4.7-a. Desta forma, o grafo de dependência da Figura 4.11-a representa um modelo parcial e incompleto, derivado deste *log*. Para geração de um modelo completo, é necessário processar os eventos adicionais presentes no *log* complementar, conforme apresentado na Figura 4.7-b. A partir deste *log*, é possível obter o conjunto complementar e completo, declarado como $W = \{BR1BR2BR3BR6BR7^{6}, BR1BR2BR4BR6BR7^{5}, \mathbf{BR1BR2BR5^{10}}\}$. Após executar o algoritmo *IncrementalMiner* para este *log*, é possível obter o grafo de dependência final apresentado na Figura 4.11-b. Este grafo apresenta estruturas adicionais extraídas a partir dos novos *traces* (ex: tarefa BR5 e os participantes Mark e *sendMail*). É importante esclarecer aqui que todos os *traces* antigos disponíveis no *log* (*traces* disponíveis na Figura 4.7-a) são descartados pelo algoritmo durante o processamento do *log* complementar. Isto é possível pelo fato do algoritmo *IncrementalMiner* manter uma referência para o último identificador de *trace* previamente processado (ex: *TraceId* = 11), obtido a partir da coluna *TraceId* do *log*.



(a) Grafo de Dependência Inicial extraído da Árvore das Melhores Relações da Figura 4.10



(b) Grafo de Dependência final

Figura 4.11: Grafo de Dependência e os participantes extraídos a partir do log de execução *W*.

Paralelamente a extração das relações a partir do *log*, o algoritmo *IncrementalMiner* também é capaz de extrair os participantes associados às atividades identificadas no *log* (ver coluna Participante na Figura 4.7). Ou seja, ele extrai todos os participantes que realizam uma ou mais atividades em um processo de negócio. Estas informações fazem parte da perspectiva organizacional do processo, a qual apresenta informações sobre a estrutura organizacional da empresa onde o processo é executado. Para isto, o algoritmo mantém um conjunto dos participantes associados a cada nó do grafo de dependência e atualiza-o a cada novo evento processado. Esta informação é mantida e atualizada de forma simples, através de uma estrutura de dados *hash map* (KNUTH, 1973), onde o nome do participante é utilizado como chave da estrutura (função *hash*).

4.4.3 Lidando com Ruídos no *Log*

Adicionalmente às heurísticas utilizadas para o cálculo da confiança das relações de dependência, o algoritmo *IncrementalMiner* utiliza diversos *thresholds* para decidir se uma relação será considerada ou não no grafo final de dependência (ver Tabela 4.4). Alguns destes *thresholds* foram herdados do algoritmo original *HeuristicMiner* criado por Weijters et al (WEIJTERS; AALST; MEDEIROS, 2006). A principal função do *threshold* aqui é ajudar a previnir a geração de falsas relações de dependência geradas por ruídos no *log*. O primeiro *threshold* utilizado pelo *IncrementalMiner* chama-se *threshold* de *Dependência*. Este *threshold* determina valores mínimos aceitáveis de confiança calculados pela heurística 1 (relação de dependência) quando uma relação for descartada pelo algoritmo por não ser considerada a melhor relação de dependência (relação removida da *Árvore das Melhores Relações* apresentada na Figura 4.10). O segundo *threshold* (Observações Positivas) verifica o número mínimo de observações positivas (ocorrências) que serão necessárias para o algoritmo considerar uma relação como válida. Este *threshold* foi implementado no algoritmo 2, item 9 e tem como objetivo verificar se uma relação aparece poucas vezes em um *log* (número menor do que o definido no *threshold*). Se este número for abaixo do *threshold,* é provável que se trate de um caso de ruído no *log*. A terceira heurística considera o valor máximo aceitável de diferença para o valor de confiança calculado para duas relações, sendo uma delas a melhor relação de dependência. Já os *thresholds ShortLoop1* e *ShortLoop2,*

foram declarados nos itens 3(b) e 4(b) do algoritmo 2. Eles consideram o valor mínimo de confiança calculado para um laço simples (heurística 2 e 3). O *threshold ANDThreshold* define o valor aceitável para a confiança calculada pela heurística 4. Valores de confiança acima do *threshold* considera que as relações fazem parte de um controle de fluxo AND (paralelismo), e o contrário reflete em um controle de fluxo XOR. O último *threshold* (Relação Obsoleta) considera o valor mínimo aceitável para uma relação ser considera obsoleta em relação à suas relações irmãs. Este *threshold* é utilizado pela heurística 5, introduzida neste trabalho e que será apresentada na próxima seção do trabalho.

Os *thresholds* introduzidos acima são muitas vezes utilizados em conjunto, principalmente os aplicados em relações de dependência calculadas pela heurística 1. Desta forma, o algoritmo *IncrementalMiner* (ver linha 14 do algoritmo 2) aceitará apenas relações de dependência entre eventos que possuirem basicamente (i) uma medida de dependência acima do valor do *threshold* de *Dependência*, e (ii) possuirem uma frequência acima do valor do *threshold* de *Observações Positivas*, e (iii) possuirem uma medida de dependência (confiança) para qual a diferença da "melhor" medida de dependência seja menor que o valor do *threshold Diferença Relativa à Melhar Dependência* (WEIJTERS; AALST; MEDEIROS, 2006).

Tabela 4.4: *Thresholds* do algoritmo *IncrementalMiner*

Thereshold	***Valor Padrão***	***Valores Aceitáveis***
Dependência	0,90	0,00 até 1
Observações Positivas	10	>= 1
Diferença Relativa à Melhar Dependência	0,05	0,00 até 1
Loop1Threshold	0,90	0,00 até 1
Loop2Threshold	0,90	0,00 até 1
ANDThreshold	0,10	0,00 até 1
Relação Obsoleta	0,99	0,00 até 1

4.4.4 Atualização de um Modelo Completo

Um modelo de processo pode ser dito completo quanto ele reproduz todos os eventos presentes em um *log* de eventos completo (ver definição 10 da seção 2.2.2). Neste caso, novos *traces* adicionados ao *log* não mudarão ou agregarão novo comportamento ao modelo. Esta afirmação pode não ser verdade quando alterações ocorrem na estrutura de um processo. Estas alterações podem ser requeridas quando as necessidades de negócio de uma empresa mudam. Porém, atualmente a mineração de processos padece de soluções que lidam com esta situação. Os algoritmos apresentados no Capítulo 3 não possuem mecanismos eficientes para esta atividade, se preocupando na maioria das vezes apenas com a simples adição de elementos a um modelo parcial ou completo (operações *a-d* da Tabela 4.3), sem considerar a remoção de elementos obsoletos do processo.

Voltando ao *log* de processo apresentado na Figura 4.7, podemos verificar que neste *log* são apresentados *traces* que representam alterações no fluxo existente (Figura 4.7-c). Estas alterações foram realizadas sobre cenários existentes do sistema de informação executado. É possível verificar na figura que os cenários *A* e *C* foram reexecutados gerando *traces* modificados no *log*. A primeira alteração pode ser observada quando a usuária Mary executou novamente o cenário *A*, gerando dezoito ocorrências do *trace*

{BR1, BR2, **BR4**, BR6, BR7}. O cenário *A* executava anteriormente a sequência de atividades {BR1, BR2, **BR3**, BR6, BR7}, ocorrendo desta vez uma mudança no evento BR3 (evento foi suprimido, tornando-se obsoleto). Esta mudança foi originada pela alteração do código fonte da aplicação, que pode ser vista na Figura 4.5. Originalmente, as linhas 15-19 do código fonte (ver Figura 4.4) executavam a regra de negócio BR3 e agora passaram a executam a regra BR4. Desta maneira, mudanças como esta podem ser muitas vezes imperceptíveis no *log*, pois dependem de diversos fatores, como a frequência dos eventos, incidência de ruídos, etc.

Uma relação obsoleta é basicamente uma relação de dependência $>w$ presente no modelo de processo minerado mas que não é mais executada. Para realizar a mineração do *log* prevendo a descoberta de relações de dependência obsoletas no grafo de dependência do processo, foram criados algoritmos auxiliares capazes de identificar tais relações (KALSING; IOCHPE; THOM; NASCIMENTO, 2012). O algoritmo *IncrementalMiner* utiliza os algoritmos *CalculaRelacaoIrmaObsoleta* e *VerificaRelacoesIrmas* (ver algoritmos 6 e 7 apresentados na seção 4.4.2) para realizar a descoberta destas relações. O algoritmo *CalculaRelacaoIrmaObsoleta* é responsável por calcular a confiança das relações candidatas obsoletas. Para isto, ele utiliza a heurística (5) definida abaixo. A heurística calcula a confiança de uma relação $>w$ em relação a uma relação irmã $\mapsto w$. Para exemplificar graficamente uma relação irmã, podemos ver a Figura 4.12-a. A relação BR2→BR3 é uma relação que contém as relações BR2$\mapsto$ BR4 e BR2$\mapsto$BR5 como relações irmãs. Desta forma, a heurística deve ser aplicada a cada relação irmã de BR2→BR3. Assim, seja W um *log* de eventos T, e $a,b,c \in T$. Então $|a>wb|$ é o número parcial de vezes que $a>wb$ ocorre em W, $|a>wc|$ é o número parcial de vezes que $a>wc$ ocorre em W, $|[a>wc]_{a \mapsto wb}|$ é o número parcial de vezes que $[a>wc]_{a \mapsto wb}$ ocorre em *W* e:

$$a \Rightarrow wb \mapsto c = \left(\frac{\left(1+\frac{|a>wb|}{|a>wc|}\right) \times \left(|a>wb| - |[a>wc]_{a \mapsto wb}|\right)}{\left(1+\frac{|a>wb|}{|a>wc|}\right) \times \left(|a>wb| - |[a>wc]_{a \mapsto wb}|\right)+1} \right) \quad (5)$$

Considerando o *log* complementar $W = \{..., BR1BR2BR4BR6BR7^{18}, BR1BR2BR5^{18}\}$ apresentado na Figura 4.7-c como sendo os *traces* gerados após a alteração da aplicação, podemos verificar que o evento BR3 está ausente. Todas as relações de dependência que estão no grafo de dependência e possuirem a atividade BR3 podem ser possíveis relações obsoletas. No exemplo, a relação BR2→BR3 é a única relação associada à atividade BR3 (ver as *Árvores das Melhores Relações de Dependência* da Figura 4.10-a). Para identificarmos se ela é realmente uma relação obsoleta, precisamos analisá-la do ponto de vista das relações irmãs (BR2$\mapsto$BR4 e BR2$\mapsto$BR5). Desta forma é necessário aplicar a heurística para cada uma delas. Assim, considerando primeiramente a relação BR2→BR4 no cálculo da confiança da relação obsoleta, temos o suporte de $|a>wb|$ como o número de parcial de vezes que BR2→BR4 ocorreu no *log* (23 vezes), o suporte de $|a>wc|$como o número parcial de vezes que BR2→BR3 ocorreu no *log* (6 vezes), e $[a>wc]_{a \mapsto wb}$ como o suporte da relação irmã BR2$\mapsto$ BR4 antes da última ocorrência no *log* da relação BR2→BR3. É possível visualizar na Figura 4.12-a que o suporte da relação irmã BR2$\mapsto$BR4 era zero (nenhuma ocorrência)

quando a relação BR2→BR3 ocorreu pela última vez. Com isto, temos a confiança da relação BR2→BR3 como sendo obsoleta em relação a irmã BR2$\mapsto$BR4 calculada como $a \Rightarrow wb \mapsto c$ = ((1 + 23 / 6) x (23 – 0)) / (((1 + 23 / 6) x (23 – 0)) + 1) = 0.991. Para a próxima relação irmã BR2$\mapsto$BR5, temos o valor de confiança $a \Rightarrow wb \mapsto c$ = ((1 + 28 / 6) x (28 – 0)) / (((1 + 28 / 6) x (28 – 0)) + 1) = 0.993. Como resultado do algoritmo *CalculaRelacaoIrmaObsoleta* para as relações irmãs BR2$\mapsto$BR4 e BR2$\mapsto$BR5 da relação BR2→BR3 temos os valores 0.991 e 0.993 de confiança, respectivamente.

É importante observarmos na heurística 5 a expressão parcial $(1 + |a > wb| / |a > wc|) \times (...)$. Esta fórmula é utilizada para maximizar o valor calculado pela heurística. Assim, sendo $|a > wb|$ o número parcial de vezes que BR2→BR4 ocorre no *log* e $|a > wc|$ o número parcial de vezes que BR2→BR3 ocorre, quanto mais eventos contendo a relação BR2→BR4 forem disponibilizados no *log* maior poderá ser a confiança da relação BR2→BR3 ser obsoleta com relação à esta relação irmã. No exemplo, temos 1 + 23 / 6 = 4,83, sendo este valor o fator de multiplicação para o restante da fórmula. Em contra partida, o inverso pode ser observado na Figura 4.12-b. Podemos ver que a confiança da relação BR2→BR4 ser obsoleta em relação à relação irmã BR2→BR3 terá fator de multiplicação calculado como 1 + 6 / 23 = 1,26, sendo este menor devido a baixa incidência de relações BR2→BR3 em relação à BR2→BR4.

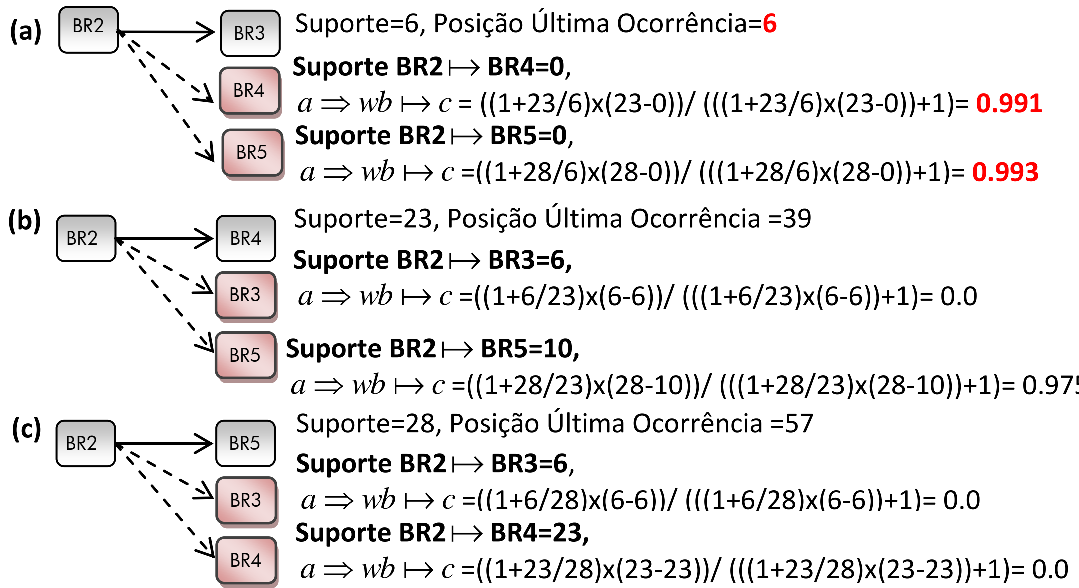


Figura 4.12: Cálculo das Relações Irmãs Obsoletas.

Para verificarmos se a relação pode ser definida como uma relação obsoleta, utilizamos o algoritmo 6 (*VerificaRelacoesObsoletas*). O algoritmo verifica o valor de confiança de cada relação irmã $\mapsto w$ de uma relação $> w$. Desta forma, uma relação é considerada obsoleta se ela obedece a seguinte definição:

Definição 1 (relação obsoleta) Seja uma relação $> w$ e suas relações irmãs $\mapsto w$, uma relação é considerada obsoleta se e somente se todas as relações irmãs possuirem o valor de confiança calculado por $a \Rightarrow wb \mapsto c$ acima do valor do *threshold Relação Obsoleta*.

Esta definição é seguida pelo algoritmo *VerificaRelacoesObsoletas* como pode ser observado no item 2 (a) do algoritmo 6. Para este trabalho, utilizou-se o valor do *threshold Relação Obsoleta* como sendo 0,99. Assim, se todas as relações irmãs de uma relação qualquer ultrapassarem este valor, então a relação $>w$ deve ser removida do grafo de dependência do processo, conforme apresentado no item 2 (b) do algoritmo 6. Voltando ao exemplo da relação BR2→BR3 gerada no *log* de exemplo, podemos verificar na Figura 4.12-a, que ambas relações irmãs apresentam valor de confiança acima de 0.99 (0.991 para BR2$\mapsto$BR4 e 0.993 para BR2$\mapsto$BR5), necessitando assim a remoção da relação BR2→BR3 do grafo de dependência, conforme apresentado na Figura 4.13.

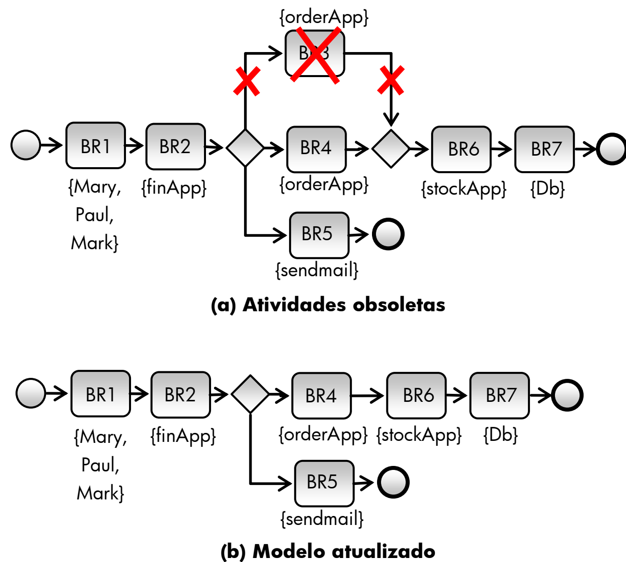


Figura 4.13: Grafo de dependência atualizado.

4.4.5 Mapeando o Resultado da Mineração para o Modelo Conceitual de Processo

Até este ponto do trabalho, foi apresentado a mineração incremental de estruturas de processos a partir de um sistema de informação. O resultado desta mineração utiliza um grafo de dependência, onde as principais estruturas semânticas de um processo de negócio são representadas. O modelo extraído pelo algoritmo *IncrementalMiner* utiliza uma representação interna de processo, formado pelo modelo conceitual apresentado na Figura 4.14. Esta representação contempla um modelo simples de grafo e não contém informações detalhadas sobre o modelo de processo de negócio. Desta forma, é necessário realizar o mapeamento dos conceitos do modelo interno de grafo para o modelo conceitual de processo de negócio (conforme modelo conceitual descrito em 4.1.1). O mapeamento segue as regras definidas na Tabela 4.5. O modelo conceitual do algoritmo é formado pelos elementos *Grafo*, *Vertice, Aresta, Semantica* e *Participante* e são mapeados diretamente para os elementos do modelo conceitual de processo, que são respectivamente *Processo*, *Tarefa*, *Transição, Gateway e Participante*. As entidades *Gateway XOR/AND* são controles de fluxos do processo de negócio e são representados pelo algoritmo *IncrementalMiner* através da entidade *Semantic*. Uma instância da entidade *Semantic* pode agrupar todos os *Vertex* que estão em uma relação XOR (ver a agregação *Vertices na Relação XOR* da entidade *Semantica* na Figura 4.14). Ou seja, uma instância da entidade *Semantica* que faz parte da agregação *semantica de entrada* ou *semantica de saída da* entidade *Vertice* pode conter os vértices (atividades) que

fazem parte de um controle de fluxo XOR. Quando uma instância da entidade *Semantica* existir na agregação *semantica de saida* é provavel a existência de um controle de fluxo *Gateway* XOR-*split*. Já quando existirem instâncias na agregação *semantica de entrada,* é provável a existência de uma relação do tipo *Gateway* XOR-*join*. Em contra partida, se existerem duas ou mais instâncias da entidade *Semantica* na agregação *semantica de entrada* ou *semantica de saida* da entidade *Vertice*, é provável que esta relação signifique uma relação *Gateway* AND (paralelismo) entre dois ou mais *Vertex*. Adicionalmente, a entidade *Vertice* é mapeada para a entidade *Evento* quando a agregação *semantica de saida* (evento de fim do processo) ou *semantica de entrada* (evento de início do processo) não conter nenhum item.

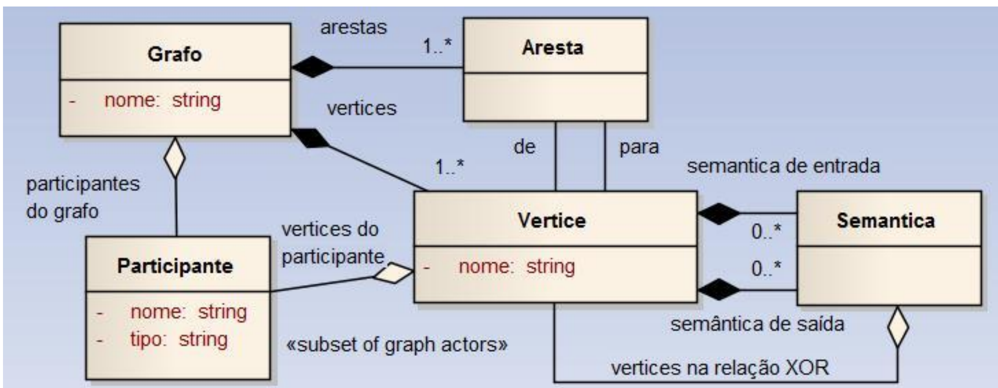


Figura 4.14: Modelo conceitual do grafo de dependência gerado pelo algoritmo IncrementalMiner

O mapeamento entre os modelos conceituais permite representarmos o resultado da mineração dos *logs* em um modelo de processo de negócio de mais alto nível, que pode ser apresentável a pessoas com menor conhecimento computacional, como analistas de negócio e usuários. Devido a simplicidade do modelo conceitual de processo de negócio utilizado neste trabalho, podemos realizar o mapeamento deste metamodelo para os principais modelos e linguagens de especificação de processos existentes (ex: BPMN, BPEL, UPC, UML, etc), sem perdermos elementos semânticos ou mesmo diminuirmos a expressividade do processo de negócio.

Tabela 4.5: Correspondência entre os elementos do modelo conceitual do algoritmo *IncrementalMiner* e o modelo conceitual de processo de negócio.

Modelo algoritmo IncrementalMiner		***Modelo Conceitual Processo***
Entidade	***Símbolo***	***Entidade***
Vertice		Tarefa
Aresta	→	Transição
Participante	{nome_part}	Participante
AND-*Join*: Ocorrências da entidade *Semantica* em *Semantica de Entrada* deve ser maior que um. AND-*Split*: ocorrências da entidade *Semantica* em *Semantica de Saida* deve ser maior que um.	+	*Gateway* (tipo: paralelo - AND)
XOR-*Join*: para uma entidade *Semantica* da relação *Semantica de Entrada,* a relação *Vertices na Relação XOR* desta entidade deve conter uma ou mais entidades *Vertice.*		*Gateway* (tipo: OU exclusivo - XOR)

Modelo algoritmo IncrementalMiner		*Modelo Conceitual Processo*
Entidade	***Símbolo***	***Entidade***
XOR-*Split*: para uma entidade *Semantica* da relação *Semantica de Saída,* a relação *Vertices na Relação XOR* desta entidade deve conter uma ou mais entidades *Vertice*		
A relação *Semantica de Entrada* não contém nenhuma entidade *Vertice*	○	Evento (tipo: INICIO)
A relação *Semantica de Saida* não contém nenhuma entidade *Vertice*	○	Evento (tipo: FIM)

4.5 Apresentação do Resultado da Mineração

Para visualização do resultado da mineração, foi criado uma ferramenta capaz de suportar o processo incremental de mineração de processos de negócio e apresentar o grafo de dependência gerado a cada ciclo de mineração (ver Figura 4.15). Esta ferramenta permite que diversos arquivos de *log* sejam importados e processados pelo algoritmo *IncrementalMiner,* a fim de gerar um modelo parcial ou completo de processo. Outra característica importante da ferramenta é a utilização do metamodelo de *log* MXML. A utilização deste metamodelo permite a padronização da geração do *log* e sua interoperabilidade entre diversas ferramentas de mineração. Adicionalmente, através desta ferramenta é possível visualizar todas as informações sobre o processo de negócio, como atividades, controle de fluxo sequencial ou paralelo (XOR/AND-*split/join*), participantes do processo e também a matriz de atividades por participante.

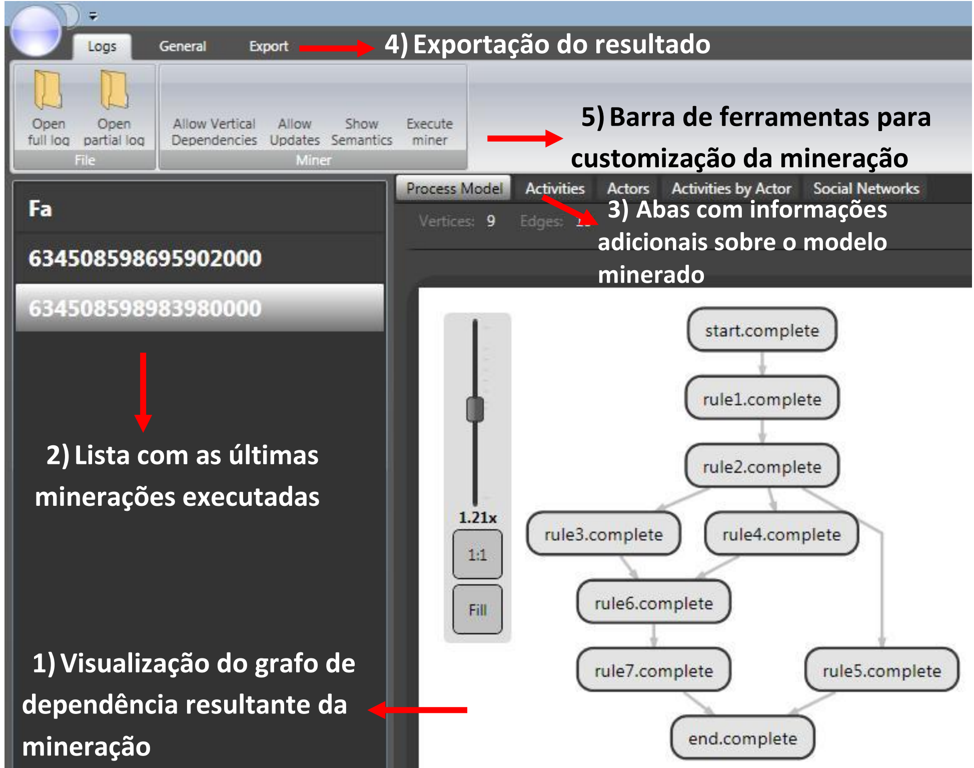


Figura 4.15: Ferramenta de mineração incremental

Como principais funcionalidades da ferramenta podemos listar:

- Mineração incremental de um *log* de processo no formato MXML;

- Visualização gráfica do grafo minerado com apresentação da semântica do processo (ex: Controle de Fluxo AND/XOR);
- Exportação do modelo para o formato XPDL;
- Visualização dos participantes do processo;
- Visualização de atividades por participante (matriz de responsabilidades de cada participante);
- Gravação do resultado da mineração para posterior visualização;

Outra característica importante da ferramenta de mineração é a possíbilidade de customizar certos aspectos da apresentação do modelo (grafo de dependência). Através da barra de ferramentas da aplicação é possível configurar o que deve ou não ser apresentado. Como exemplo, podemos utilizar a Figura 4.16. O modelo de processo apresentado difere sensivelmente da Figura 4.15. Apesar de serem modelos iguais, o modelo da figura abaixo apresenta os aspectos semânticos do processo (ex: controle de fluxo XOR repersentado no grafo pelo nó *xor-out.rule2.complete.0*), os quais foram omitidos do grafo da Figura 4.15 através da customização da ferramenta.

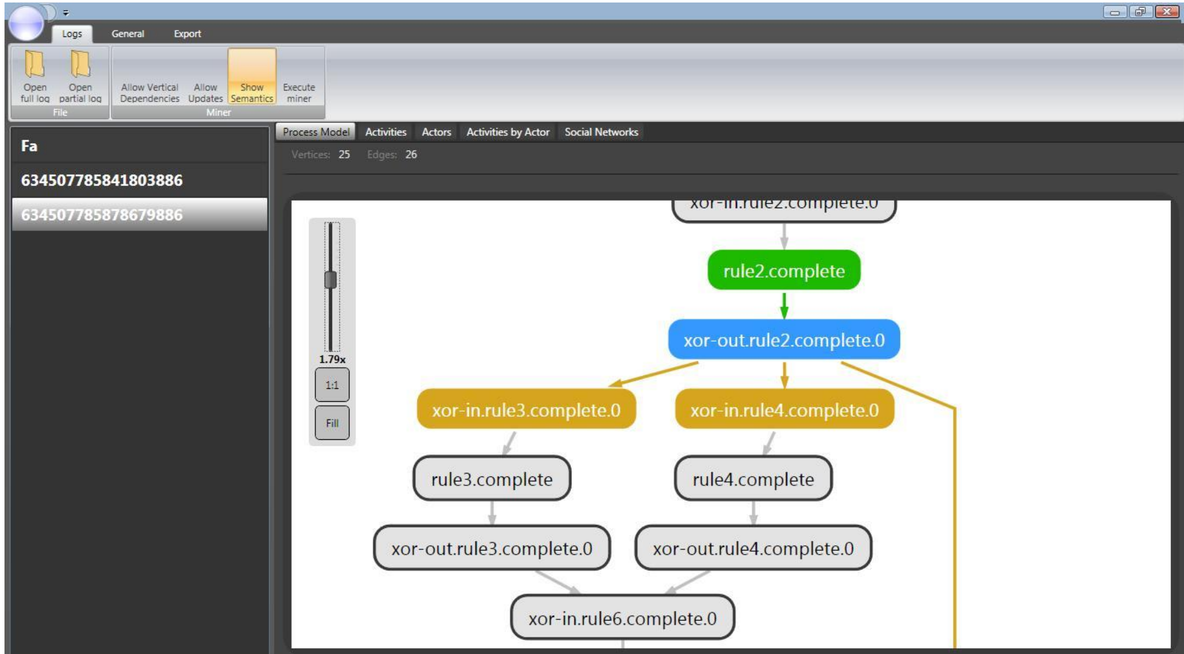


Figura 4.16: Visualização da semântica do processo

O modelo apresentado na Figura 4.15 representa o grafo de dependência parcial visualizado na Figura 4.11-b. Como foi explicado ao longo deste capítulo, este resultado é obtido a partir da mineração do *log* apresentado na Figura 4.7-a e b. Para realizarmos a geração do processo de negócio final, que será visualizado pelo usuário de negócio, é necessário a utilização das regras de mapeamento apresentadas na Tabela 4.5. Desta forma, para chegarmos ao modelo final, foi necessário realizar a exportação do modelo para o formato XPDL, através da ferramenta de mineração, e sua posterior conversão para a notação BPMN, através da ferramenta BizAgi Modeler. O modelo de processo final pode ser conferido na Figura 4.17.

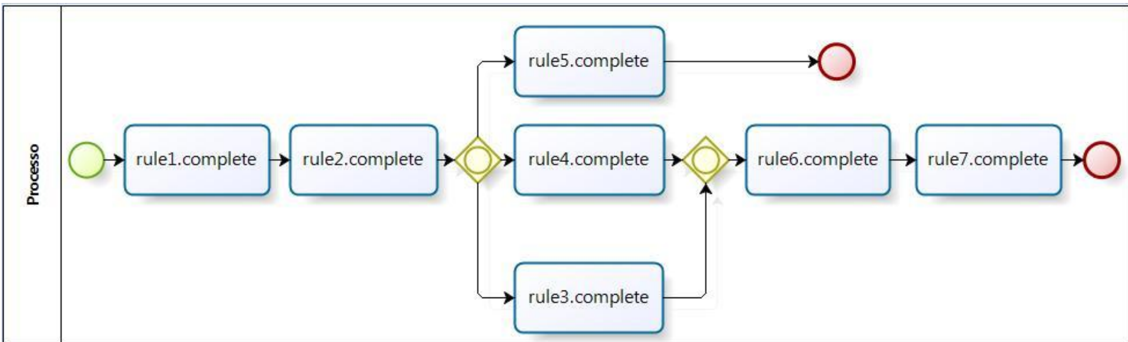


Figura 4.17: Visualização do processo de negócio

O mapeamento entre o metamodelo XPDL utilizado por este trabalho (modelo definido na seção 4.1.1) e a notação gráfica BPMN é possível graças ao suporte nativo do modelo XPDL em contemplar entidades da notação BPMN. Em outras palavras, a notação BPMN permite o mapeamento 1:1 para os elementos do modelo XPDL, permitindo a conversão de forma simples. Esta característica é suportada deste a versão 2.0 da XPDL, onde houve a reestruturação do metamodelo, passando a suportar de forma completa esta notação.

5 EXPERIMENTOS

O objetivo deste capítulo é demonstrar através de uma análise quantitativa a acurácia do algoritmo *IncrementalMiner.* Os experimentos aqui foram divididos em dois grupos. O primeiro grupo utilizou um sistema de informação legado para realizar a mineração do *log*. A idéia foi aplicar o processo incremental definido no capítulo 4 a fim de extrair modelos de processo a partir do código fonte de um sistema. O segundo grupo de experimentos utilizou dados empíricos gerados a partir de um simulador de execução de processos. Este simulador teve a função de gerar um conjunto de modelos de processos e seus respectivos conjuntos de registros de *log* de execução. Os logs gerados foram suficientemente grandes para realizar a mineração incremental do processo. A utilização de um simulador foi necessária devido a dificuldade em se reproduzir certos aspectos da mineração em sistemas de informação reais, como por exemplo a incidência de ruídos e alterações nos modelos de processos. Além desta divisão, os experimentos foram subdivididos em: i) experimentos que avaliam o tempo de processamento e ii) experimentos que avaliam a qualidade dos modelos descobertos.

Para execução destes experimentos foi utilizado um PC com processador Intel Core 2 Duo 2GHz e 4GB de memória RAM. O algoritmo *IncrementalMiner* foi implementado utilizando a linguagem de programação Java. Adicionalmente, foi implementada uma ferramenta gráfica desenvolvida para suportar a mineração incremental do algoritmo (ver seção 4.5) e também o *parser* do *log* de eventos no formato MXML (DONGEN; AALST, 2005a).

5.1 Experimentos Utilizando um Sistema de Informação Legado

Esta seção discute um estudo de caso que foi realizado com base em um sistema de informação legado real. O sistema legado utilizado consiste de um ERP que dá suporte a diversos processos de negócio. O sistema é composto por aproximadamente 3.000.000 de linhas de código, distribuídos em cinco aplicações.

A idéia deste experimento foi extrair incrementalmente deste sistema modelos de processos a partir do comportamento dinâmico executado e registrado no *log*. Os resultados obtidos com este estudo de caso serviram para avaliar de um ponto de vista prático a performance e qualidade dos resultados disponibilizados na mineração utilizando o algoritmo *IncrementalMiner*. Nestes experimentos não foram consideradas modificações nos processos de negócio (ex: modificações no código fonte da aplicação durante o processo de mineração), sendo esta característica abordada nos experimentos da seção 5.2.

5.1.1 Geração dos *Logs* de Eventos

Para todos os experimentos discutidos nesta seção, foram utilizados um conjunto de *logs* gerados através de sucessivas execuções do sistema legado. O sistema ERP legado

possuia diversos módulos (ex: Gerenciamento Financeiro, Gerenciamento Vendas, Gerenciamento de Serviços, etc), a grande maioria deles interconectados, como apresentado na Figura 5.1. Cada um destes módulos é responsável por implementar de forma implícita (sem um modelo projetado apriori) inúmeros processos de negócio.

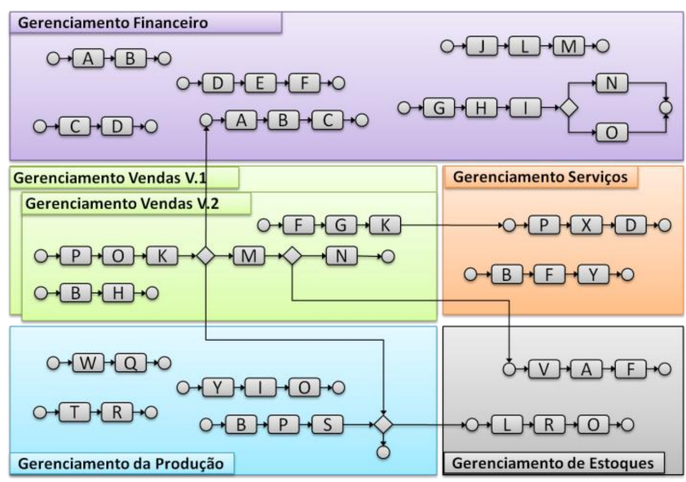


Figura 5.1: Sistema ERP legado e seus módulos e relacionamentos

Antes de iniciar o processo de mineração do sistema legado, foi necessário instrumentar a aplicação para que cada atividade presente no código fonte fosse gravada no *log* de execução. Novamente aqui, como definido no Capítulo 4, cada atividade registrada no *log* foi derivada da execução de uma regra de negócio presente no código fonte da aplicação. Desta forma, cada regra de negócio foi anotada através da adição de comandos para gravação do seu conteúdo no *log*, similarmente como apresentado na seção 4.1.2.

Após a instrumentação da aplicação, alguns cenários de execução foram definidos para coordenar a execução do sistema e a geração do conteúdo do *log*. Estes cenários serviram também para reduzir a amostra de processos extraídos do legado, lembrando que a idéia do experimento não era extrair todos os processos do legado, e sim uma pequena amostra para ser avaliada.

Os cenários de usuário foram divididos e executados em sete grupos. Cada grupo representou diversos cenários relacionados a um usuário específico do sistema legado. Seguindo sucessivas execuções destes cenários, foram gerados sete arquivos de log (um para cada usuário) com um total de aproximadamente 10.500 instâncias de traces de aplicação. Os arquivos foram gerados e rotulados respectivamente como *A, B, C, D, E e F*. Os cinco primeiros arquivos foram gerados de forma a conter conteúdos complementares, onde $A \subseteq B \subseteq C \subseteq D \subseteq E$. Em outras palavras, o arquivo B possuia todos os traces do arquivo de *log* A e mais 2.000 novas instâncias. O arquivo C possuia todos os traces dos arquivos de *log* A e B e mais 2.000 novas instâncias, e assim sucessivamente até o arquivo E conter os 10.000 traces de processo relacionados a todos os módulos do sistema legado. O arquivo de log A possuia *traces* incrementais relacionados a execução do módulo de Gerenciamento Financeiro. Já o arquivo B, possuia *traces* relacionados a execução do módulo Gerenciamento de Vendas. Em seguida, o arquivo de *log* C armazenava *traces* relacionados a execução do módulo Gerenciamento de Serviços. Já os arquivos D e E registraram *traces* dos módulos Gerenciamento de Produção e Estoques, respectivamente. Por último, o arquivo F, diferente dos demais arquivos, introduziu *traces* modificados, gerados a partir da execução da versão 2 do módulo Gerenciamento de Vendas (ver representação das

versões na Figura 5.1). Os *traces* modificados foram decorrentes de modificações realizadas no código fonte deste módulo, ocorridas propositalmente durante o processo de mineração.

5.1.2 Análise de Performance

O primeiro resultado obtido pode ser visto na Figura 5.2. Após processar completamente o primeiro arquivo de *log* (primeiras 2000 instâncias do arquivo A), o algoritmo *IncrementalMiner* considerou somente novos *traces* contidos nos novos arquivos de *log* submetidos para processamento, diferente dos demais algoritmos não incrementais apresentados no gráfico. Adicionalmente, o algoritmo mostrou linearidade no tempo de execução, a medida que novos arquivos com novas adicionais eram mineradas. Ao final, o algoritmo *IncrementalMiner* processou o conjunto de *logs* completo quatro vezes (400%) mais rápido que o algoritmo Alpha++ e oitenta vezes mais rápido que o algoritmo HeuristicMiner (8000%). Isto é possível devido ao algoritmo (i) considerar somente instâncias com novos identificadores, (ii) implementar uma estrutura de dados eficiente (ex: árvores AVL) e (iii) realizar o processamento de cada *trace* de forma otimizada (cada elemento no trace é visitado apenas uma vez). É importante observarmos nos testes apresentados na Figura 5.2 que o algoritmo Alpha++ também mostrou um bom tempo de resposta, mesmo sem possuir a característica de processar os *logs* incrementalmente. O grande problema deste algoritmo está na baixa acurácia dos modelos extraídos, quando comparados com os obtidos com o algoritmo *IncrementalMiner* e demais algoritmos, conforme será abordado na próxima seção.

Tabela 5.1: Comparativo de tempo (em segundos)

	Instâncias no *log*						
	A	**B**	**C**	**D**	**E**		
Abordagem de Mineração	2000	4000	6000	8000	10000	**Total**	
Mineração Completa (seg)	1.4	2.7	4.0	5.4	6.7	**20.2s**	
Mineração Incremental (seg)	1.4	1.5	1.4	1.5	1.5	**7.2s**	64%

Adicionalmente, na Tabela 5.1 é mostrado o comparativo do processamento dos *logs* com e sem a mineração incremental realizada pelo algoritmo *IncrementalMiner*. Na primeira linha da tabela, o tempo total de processamento da mineração do conteúdo completo dos cinco arquivos de *log* (sem a mineração incremental) foi apresentado. Podemos verificar que foram necessários 20,2 segundos para processar todo o conteúdo dos arquivos. Já na segunda linha, o tempo total de processamento do *log* levou em consideração a mineração incremental do *logs*, reduzindo-o à 7,2 segundos. Ao final, foi obtido um ganho total de 64% no tempo total de processamento usando a mineração incremental dos *logs*. Apesar destes números parecerem relaltivamente óbvios, é importante salientar a importância do tempo de execução durante a mineração de um sistema legado. Neste cenário, a descoberta de conhecimento pode se apresentar bastante dinâmica, necessitando a reexecução sucessiva do algoritmo e também das aplicações. Isto se torna ainda mais evidente quando é somado à este cenário alterações nos processos de negócio, necessitando além de agilidade, a capacidade de adaptar os modelos às novas necessidades de forma incremental.

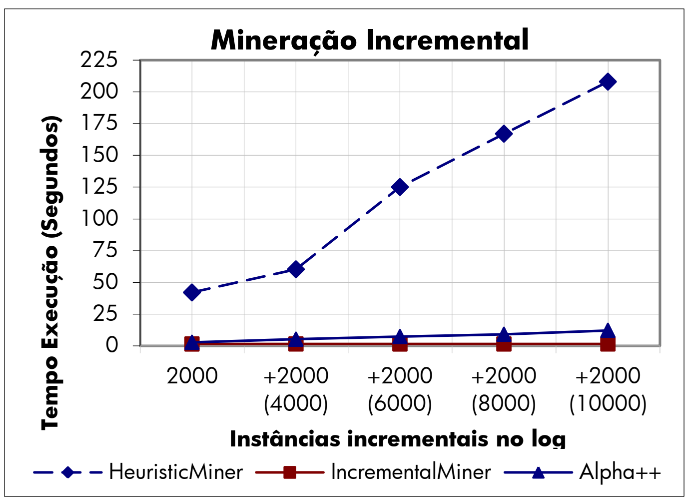


Figura 5.2: Tempo total de processamento durante a mineração de novos traces adicionados ao *log*

Para verificar como o algoritmo *IncrementalMiner* e os demais algoritmos operam com grandes arquivos de *log* (ex: arquivos de *log* com grande volume de instâncias processados a partir do zero), foram realizados alguns experimentos levando em consideração a escalabilidade dos algoritmos. Aqui, foram utilizados os mesmos cinco arquivos de *log* utilizados anteriormente. Porém, desta vez, os arquivos foram utilizados de forma independente, sem considerar a mineração incremental. O resultado pode ser visto na Figura 5.3 e Figura 5.4. O tempo de execução do algoritmo *IncrementalMiner* e *Alpha++* aumentam linearmente a medida que o número de instâncias de processo também aumenta, o que não ocorre com o algoritmo *HeuristicMiner*. Isto demonstra que o algoritmo *IncrementalMiner* escala melhor e é mais rápido que os demais algoritmos, mesmo não utilizando a mineração incremental. Nos experimentos realizados, também foi considerado a utilização do algoritmo *GeneticMiner* (MEDEIROS et al, 2007), o qual é considerado um dos principais algoritmos da área. Porém, este algoritmo mostrou tempo de execução dezenas de vezes superior ao dos demais algoritmos apresentados. *GeneticMiner* precisou de diversas horas para processar os mesmos arquivos de *log*, e por esta razão não foi considerado no gráfico de análise de performance.

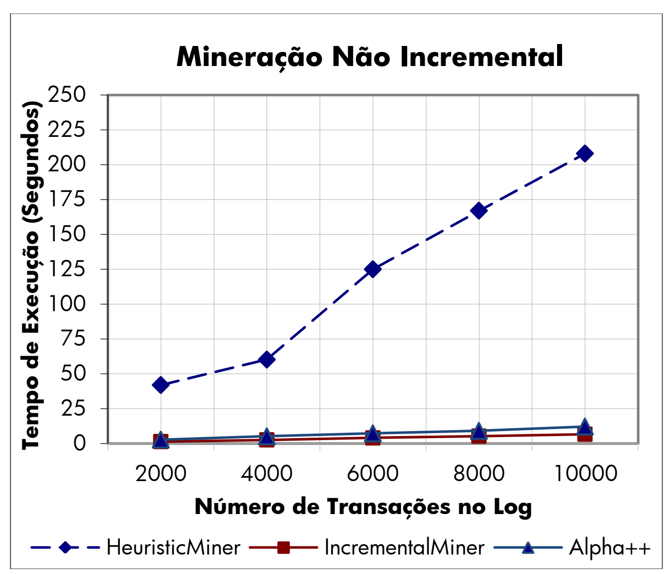


Figura 5.3: Escalabilidade dos algoritmos (sem mineração incremental)

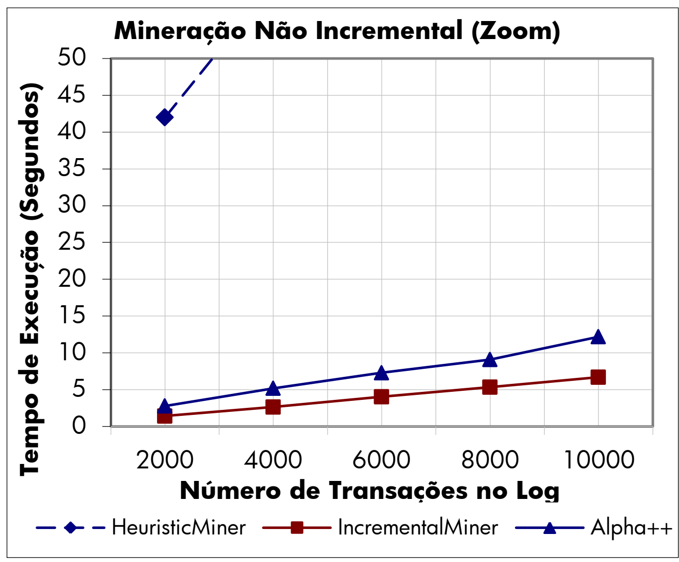


Figura 5.4: ***Zoom*** **da Figura 5.3.**

Nos experimentos realizados neste trabalho, os arquivos de *log* continham apenas *traces* gerados pela execução da aplicação de Gerenciamento Financeiro. É possível observar que as demais aplicações do sistema ERP (ver Figura 5.1) estão relacionadas a esta aplicação, o que sugere que os processos de negócio destes módulos também estão relacionados aos processos financeiros extraídos do *log*. Assim, se decidirmos minerar as demais aplicações utilizando algoritmo *IncrementalMiner,* não será necessário realizar a mineração de todo o *log* novamente. Somente as novas instâncias relacionadas à execução dos novos módulos deverão ser processadas, atualizando os modelos existentes ou gerando novos modelos quando necessário.

5.1.3 Análise da Qualidade

Esta seção descreve a análise de qualidade dos modelos de processo extraídos pela execução dos diferentes algoritmos de mineração. A comparação de qualidade foi realizada utilizando as métricas definidas em (ROZINAT; MEDEIROS; WEIJTERS; GUNTER; AALST, 2007), as quais foram revisitadas na seção 2.5 deste trabalho. Estas métricas descrevem diversas medidas para avaliar a qualidade de um modelo baseado no comportamento presente no *log* de execução do processo. A maioria destas métricas reproduz o conteúdo do *log* em uma espécie de *replay,* contra o modelo minerado para descobrir possíveis diferenças. Assim, se o *log* for reproduzido e o mesmo não reproduzir o modelo descoberto, é provável que a métrica revele um valor baixo de acurácia para o modelo.

Para executar estes experimentos, foi utilizado o plug-in *Control Flow Benchmark da* ferramenta ProM (DONGEN, 2005b). Esta ferramenta implementa as principais métricas de qualidade definidas para mineração de processos. Estas métricas são classificadas de acordo como a classificação dada na seção 2.5. Desta forma, cada modelo descoberto pelos algoritmos de mineração foi importado nesta ferramenta para medição da acurácia.

Na Tabela 5.2 é possível ver o resultado da análise. As três primeiras métricas apresentadas nas linhas 1-3 (PM, f, PF_{complete}) são responsáveis por calcular quanto do comportamento registrado no *log* foi corretamente capturado pelo algoritmo e representado no modelo (*Fitness*). Alpha++ apresentou um valor baixo para a métrica

Fitness Parsing (*PM*). Isto significa que um alto número de *traces* no *log* não pode ser gerado pelo modelo descoberto pelo algoritmo Alpha++. Outro importante valor de métrica associado ao modelo resultante do Alpha++ é *Fitness* $PF_{complete.}$ Ele considera quais problemas ocorrem durante o *replay* do *log*. Por exemplo, eventos que não podem ser reproduzidos pois a atividade correspondente não foi ativada (executada) ou foi ativada indevidamente. Em parte, esta falta de qualidade explica a diferença de tempo de execução entre o algoritmo *HeuristicMiner* e Alpha++. Embora ambos necessitem reprocessar as entradas antigas em cada novo arquivo de *log* dos experimentos apresentados aqui, Alpha++ apresentou tempo de execução significativamente mais baixo. Em contra partida o modelo resultante apresentou a pior qualidade entre todos os algoritmos. Também é possível notar que os algoritmos *IncrementalMiner* e *HeuristicMiner* possuem resultados iguais de qualidade paras os modelos de processos gerados a partir dos *logs*. Isto pode ser explicado pelo fato do algoritmo *IncrementalMiner* ter sido construído com base nas heurísticas do algoritmo *HeuristicMiner*.

Tabela 5.2: Comparativo das métricas de qualidade entre *IncrementalMiner* (IM), *HeuristicMiner* (HM), *Genetic Miner* (GM) e *Alpha++*

Métrica	IM	HM	GM	*A++*
Fitness Parsing Measure PM	1	1	1	0.006
Token Based Fitness (*f*)	1	1	1	0.926
Fitness $PF_{complete}$	1	1	1	0.728
Behavioral Appropriateness a_B	0.78	0.78	0.98	0.720
Behavioral Precision Bp	1	1	1	0.907
Behavioral Recall Br	1	1	1	0.907
Causal Footprint	0.99	0.99	1	0.96
Structural Appropriateness a_S	1	1	1	1
Structural Precision S_P	1	1	1	1
Structural Recall S_R	1	1	1	1
Duplicates Precision D_P	1	1	1	1
Duplicates Recall D_R	1	1	1	1

O segundo grupo de métricas da tabela acima quantificou a precisão e generalização do modelo. Ou seja, quanto do comportamento foi capturado a mais no modelo do que foi observado no *log*. Esta métrica considera ambas as relação sucessoras e predecessoras (ex: comportamento paralelo) para as atividades no *log* e para as atividades no modelo e realiza a comparação destas relações. Os algoritmos *HeuristicMiner* e *IncrementalMiner* apresentaram valores satisfatórios para este conjunto de métricas, diferente do algoritmo Alpha++ que apresentou um valor abaixo dos demais algoritmos para a métrica *Behavioral Appropriateness* a'_B.

O grupo final de métricas quantifica a qualidade da estrutura do modelo. Esta estrutura consiste das atividades bem como das conexões entre estas atividades, como a semântica *split/join*. Podemos ver na Tabela 5.2 que todos os algoritmos apresentaram excelentes valores para este grupo de métricas.

Como consideração final da análise de qualidade realizada sobre a mineração do *logs* do sistema legado, é possível verificar que todos os modelos descobertos, com

exceção do modelo retornado pelo algoritmo Alpha++, reproduzem o *log* perfeitamente e possuem precisão aceitável.

5.1.4 Resultado da Mineração do Sistema de Informação

Os primeiros resultados obtidos com a extração do modelos de processo do sistema legado podem ser visto na Tabela 5.3. Ela mostra a lista completa de elementos extraídos a partir da mineração do *log* de execução do sistema de informação. Estes elementos seguem a classificação de elementos do modelo conceitual definidos na seção 4.1. Todos os elementos são parte do modelo de processo de negócio e foram minerados utilizando o algoritmo *IncrementalMiner*. Podemos verificar que a mineração incremental revelou gradualmente novos elementos após cada conjunto de instâncias mineradas, gerando modelos mais completos a cada arquivo de *log* processado. Opostamente, o último arquivo de log (ver coluna F da Tabela 5.3) revelou elementos obsoletos, como tarefas e controles de fluxo do tipo XOR, removidos do modelo de processo pelo algoritmo. Isto demonstra que foi possível extrair processos atualizados, mesmo quando o sistema legado sofreu modificações. Este comportamento é fundamental para a extração incremental de processos a partir de sistemas legados, pois a manutenção de um sistema geralmente não é congelada durante a modernização e deve considerar tais modificações durante o processo incremental.

No total, foram identificadas 16 estruturas parciais de processos (ver exemplos na Figura 5.5), pertencentes aos diversos módulos do sistema legado, como mostrado na primeira linha desta tabela. Adicionalmente, os demais elementos apresentados representam as regras de negócio (tarefas) e participantes do processo. Alguns elementos desta tabela, como *Participante Humano* e *Controle de Fluxo AND-split*, não foram extraídos a partir do código fonte do sistema legado. Isto retrata uma característica fundamental e um dos principais diferencias da mineração de processos em relação a outras abordagens de extração de processos a partir de sistemas legado (ZOU, 2006) (ZOU, 2004), (LIU, 1999). Estes elementos foram minerados diretamente a partir do comportamento dinâmico do sistema no *log* de execução, e não possuem correspondente direto no código fonte do sistema. O *Controle de Fluxo AND-split/join* representa a execução paralela de duas ou mais tarefas no processo. É possível ver na Figura 5.5-a um exemplo deste elemento, onde as tarefas *Imprimir Requisição* e *Imprimir Contrato* são executadas em paralelo dentro do processo. Ou seja, os participantes do processo (ex: *Paul* e *Mark*) podem executar paralelamente estas atividades em qualquer ordem.

Tabela 5.3: Exemplo de elementos extraídos incrementalmente a partir do sistema legado

Elemento	*Arquivos de Log*						*Total*
	A	*B*	*C*	*D*	*E*	*F*	
Estruturas de Processo de Negócio	+3	+2	+5	+2	+4	-	**16**
Participantes Humanos	+1	+1	+1	+1	+1	+1	**6**
Participantes Sistêmicos	+3	+1	+2	+3	+1	-	**10**
Regras de Negócio (Tarefas)	+28	+15	+11	+17	+18	+2 (-4)	**87**
Controle de Fluxo XOR-split/join	+10	+5	+5	+4	+3	+3(-2)	**28**
Controle de Fluxo AND-split/join	+1	-	-	-	-	-	**1**
Total	**+46**	**+24**	**+24**	**+27**	**+27**	**+6(-6)**	**148**

Outro importante comportamento extraído do *log* pelo algoritmo *IncrementalMiner* é a matriz de responsabilidades dos participantes do processo, apresentada na Tabela 5.4. Esta matriz mostra quais são as responsabilidades de um participante (humano ou sistema) dentro do processo. Ou seja, o que um participante pode ou não fazer dentro e uma organização. Por exemplo, é possível ver que *Paul* e *Mark* são responsáveis por criar a requisição de pedido financeiro (ex: tarefa *Preencher Requisição*) no processo, e que o usuário John é responsável por realizar o pagamento (ex: *Preencher Pagamento*).

A matriz de responsabilidade pode ser utilizada de várias formas, como documentação das responsabilidades dos funcionários dentro de um processo e também auditoria do processo. Na auditoria, é possível verificar por exemplo se algum usuário executou indevidamente uma atividade dentro do processo e assim realizar as devidas correções.

Tabela 5.4: Exemplo de Matriz de responsabilidades por participante extraídas dos processos da Figura 5.5.

Tarefa	*Paul*	*John*	*Mark*	*Db*	*FinancialApp*
Preencher Requisição	X		X		
Consultar Requisição				X	
Consultar Cliente				X	
Validar Data Movimento					X
Validar Data Vencto					X
Consultar Vendedor				X	
Calcular Comissão					X
Salvar Requisição				X	
Aprovar Requisição			X		
Calcular Taxas					X
Preencher Pagamento		X			
Salvar Pagamento				X	
Imprimir Requisição	X		X		
Imprimir Contrato	X		X		

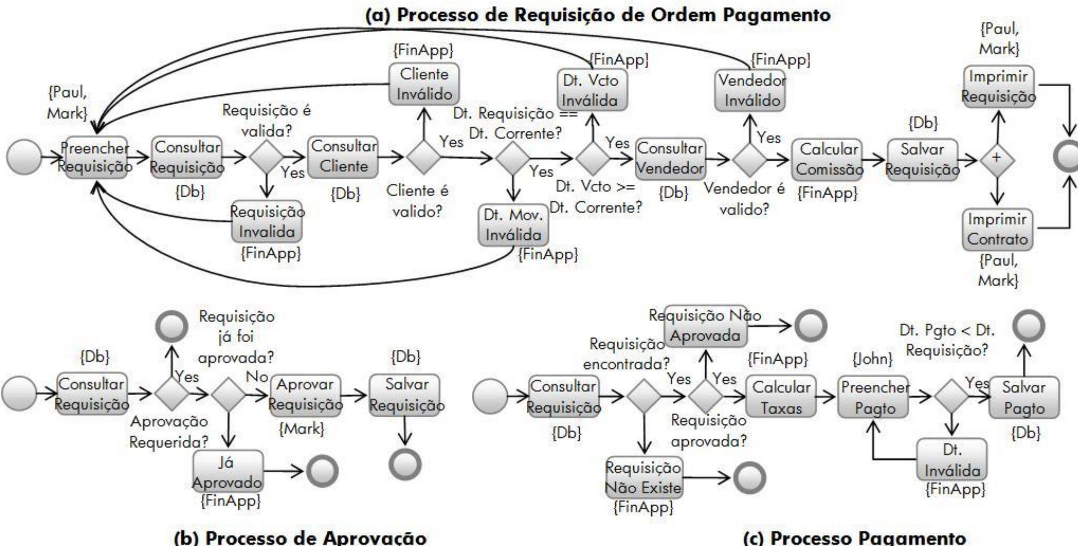


Figura 5.5: Exemplos de modelos de processos parciais extraídos do sistema legado

5.1.5 Validação do Modelo Resultante

Os experimentos mostrados anteriormente tiveram como objetivo mostrar a qualidade do algoritmo medindo seus modelos através das métricas de qualidade, além de demonstrar a performance do algoritmo em relação aos demais algoritmos. Nesta seção, o objetivo é demonstrar a precisão do modelo em relação ao comportamento presente no sistema legado. Ou seja, verificar se o comportamento representado no modelo de processo é coerente com o modelo de processo executado pelo sistema legado. Para isto, foi utilizada uma medida estatística de validação da precisão dos modelos. Esta medida estatística demonstrou que existe um nível aceitável de concordância humana sobre as estruturas de processos obtidos. Para esta verificação de concordância, foi utilizado a estatística Kappa (COHEN, 1960). Kappa foi utilizado para dar uma medida quantitativa de concordância entre observadores (especialistas) do processo com relação aos modelos minerados pelo algoritmo *IncrementalMiner*. O valor de saída desta estatística pode variar de -1 (discordância completa) até +1 (concordância perfeita). A formula da estatística é apresentada pela equação (5), onde $P(A)$ é a probabilidade empírica de concordância entre dois observadores para um aspecto do processo, e $P(E)$ é a prababilidade de concordância entre dois observadores que realizam a classificação deste aspecto de forma randômica (com a frequência empírica observada de cada aspecto). Um aspecto representa uma caracterísitca da estrutura do processo a qual necessita de uma avaliação de precisão. Desta forma, a medida de concordância é aplicada sobre seis aspectos diferentes, como pode ser visto na **Tabela 5.5**.

$$K = \frac{P(A) - P(E)}{1 - P(E)} \qquad (5)$$

O primeiro aspecto verifica o mapeamento de regras de negócio no modelo. Ela definie se uma regra de negócio apresentada no modelo foi corretamente associada para uma atividade no modelo de processo (ex: tarefas e controles de fluxo). Ele também verifica se todas as regras de negócio estão contidas no modelo de processo. O segundo aspecto refere-se ao mapeamento de participantes para as tarefas do processo. Ele define se o participante associado a uma tarefa realmente executa a atividade no processo. Os próximos dois aspectos definem o mapeamento das transições de entrada e saída de uma atividade. Estes aspectos definem se todas as transições de entrada e saída associadas a uma atividade estão corretas. Os últimos dois aspectos avaliam se a semântida de entrada e saída das atividades (ex: AND/XOR-*split/join*) está correta. Assim, é possível verificar se os controles de fluxo associados ao processo estão corretos.

Nos experimentos utilizando dois analistas de negócio diferentes, a estatística Kappa apresentou valores entre 0,90 e 1,0 como mostrado na coluna Kappa da**Tabela 5.5**. Valores acima de 0,8 são geralmente considerados bons níveis de concordância (LANDIS; KOCH, 1977), o que sugere que o resultado obtido com a mineração de processos do sistema legado obteve precisão satisfatória.

Tabela 5.5: Resultado da estatística Kappa

Aspecto	*Kappa*
Mapeamento Atividades (Regras Negócio)	0,93
Mapeamento Participante	1
Transições de Entrada	0,93

Aspecto	*Kappa*
Transições de Saída	0,90
Semântica de Entrada	0,91
Semântica de Saída	0,92

Para realizar os cálculos da estatística Kappa apresentados na tabela Tabela 5.5, foi utilizado a ferramenta de cálculo *on-line* Cohen's Kappa (GEERTZEN, 2010). A ferramenta utiliza a fórmula (5), aplicando-a sobre a observação de diferentes observadores (neste caso três). Para demonstrar o funcionamento do cálculo, o aspecto Mapeamento de Regras de Negócio foi detalhado na Tabela 5.6. Na tabela, são apresentadas as observações realizadas para os processos da Figura 5.5. Na primeira coluna da tabela (Regras), são listados todas as regras de negócio que compõem os modelos de processo da Figura 5.5 (conforme classificação de regras definida em 4.1.2). A idéia aqui é calcular o nível de concordância sobre o aspecto Mapeamento de Regras de Negócio para o modelo de processo entre os três observadores (cruzamento de dados entre os observadores Analista 1, Analista 2 e *IncrementalMiner*). É importante ressaltar que este aspecto não leva em consideração a ordem parcial de execução das atividades (posição da atividade no grafo de processo) e sim apenas se a atividade está ou não presente no processo. A ordem de execução é considerada nos demais aspectos de mapeamento do processo (ex: Transições de Entrada e Saída) apresentados na Tabela 5.5 Para calcular o valor de *K,* é necessário primeiro calcular os valores de $P(A)$ e $P(E)$. Considerando inicialmente apenas a concordância entre o algoritmo *IncrementalMiner* (observador 1) e o Analista 1 (observador 2), o cálculo de P(A) indica a proporção de concordância entre os dois observadores. Neste caso, é necessário descobrir o número total de ocorrências (regras classificadas) onde ambos observadores concordam com a classificação das atividades (Regra Negócio). Neste exemplo o valor desejado é 41. Esta informação é apresentada na linha 8, coluna 8 da Tabela 5.6 (coluna IM+A1).

Para calcular o valor de P(A) é necessário dividir o valor total de concordância das atividades classificadas pelo algoritmo *IncrementalMiner* e o Analista 1 pelo total de atividades classificadas (linha 8, coluna 2), sendo P(A) = 41 / 45 = 0,91 (linha 9, coluna 3). Para calcular o valor de P(E) é necessário descobrir a probabilidade hipotética de concordância ao acaso entre os observadores. Este valor é calculado para os observadores *IncrementalMiner* e Analista 1 utilizando os valores das colunas 3 e 5 da Tabela 5.6 (colunas Proporção sobre o total). Para realizar o calculo, é necessário multiplicar os valores de ambas colunas para cada regra classificada (linhas 1 até 7) e ao final realizar o somatório destes valores. Assim P(E) = (0,07 x 0,07) + (0,18 x 0,18) + (0,16 x 0,07) + (0,24 x 0,24) + (0,31 x 0,31) + (0,00 x 0,09) + (0,04 x 0,04) = 0,20 (linha 9, coluna 5). Após calculados os valores de P(A) e P(E), é possível calcular o valor de Kappa, conforme a fórmula (5), sendo K = (0,91 – 0,20) / (1 – 0,20) = 0,89 (linha 9, coluna 7). Após calculado o valor de K para os observadores IM+A1, é necessário repetir o cálculo para o cruzamento dos demais observadores (IM+A2 e A1+A2). Ao final, o valor de K resultante representa a média Kappa de todos os observadores, onde K = (0,89 + 0,89 + 1,00) / 3 = 0,93 (valor apresentado no fim da Tabela 5.6).

Tabela 5.6: Resultado do cálculo para o aspecto Mapeamento de Atividades (Regras de Negócio)

	IncrementalMiner (IM)		Analista 1 (A1)		Analista 2 (A2)		Ocor. concordância entre		
Regras	Ocor. nos processos	Proporção sobre total	Ocor. nos Processos	Proporção sobre total	Ocor. nos Processos	Proporção sobre total	IM+A1	IM+A2	A1+A2
Start	3	0,07	3	0,07	3	0,07	3	3	3
DP	8	0,18	8	0,18	8	0,18	8	8	8
End	7	0,16	3	0,07	3	0,07	3	3	3
CF	11	0,24	11	0,24	11	0,24	11	11	11
UI	14	0,31	14	0,31	14	0,31	14	14	14
N/D	0	0,00	4	0,09	4	0,09	0	0	4
Math	2	0,04	2	0,04	2	0,04	2	2	2
Total	45		45		45		41	41	45
IM+A1	P(A) =	0,91	P(E) =	0,20	K =	0,89	Nível de concordância (K) médio entre observadores		0,93
IM+A2	P(A) =	0,91	P(E) =	0,20	K =	0,89			
A1+A2	P(A) =	1,00	P(E) =	0,21	K =	1,00			

Grande parte da razão do valor Kappa calculado acima não alcançar a concordância perfeita está relacionada a falta de concordância dos observadores com relação aos eventos de fim do modelo de processo (a) e (b) da Figura 5.5 (ver valores destacados nas três colunas *Ocor. nos processos* da linha 3 na Tabela 5.6). Segundo os Analistas 1 e 2, estes eventos não deveriam estar nestes modelos, divergindo neste ponto do modelo de processo gerado pelo *IncrementalMiner*. Adicionalmente, os analistas confirmaram que o modelos de processo da Figura 5.5 deveriam estar interligados, formando um único modelo de processo, e não 3 modelos parciais independentes, como foi gerado pelo algoritmo *IncrementalMiner*. Esta limitação está associada ao identificador de processo (*trace id*) utilizado durante a geração do *log*. Conforme descrito anteriormente na seção 4.1.2, este identificador é composto de duas formas diferentes, i) a partir do identificador da instância da aplicação ou ii) através de um variável comun dentro da aplicação. Para a aplicação utilizada no estudo de caso apresentado aqui, não havia a existência desta variável comun capaz de ser utilizada como identificador do *trace* do processo durante a geração do *log*. Desta forma, não foi possível interligar os trechos de processos executados por usuários diferentes. Este tipo de processo é geralmente tratado como um processo colaborativo e atualmente não é suportado de forma total pela abordagem de mineração deste trabalho. Desta forma, a limitação não está associada diretamente ao algoritmo *IncrementalMiner*, e sim a como o trace de execução do processo é gerado.

5.2 Experimentos Utilizando Dados Simulados

Os experimentos foram divididos e executados em dois grupos. O primeiro grupo demostrou a qualidade dos modelos durante a mineração de logs sem a ocorrência de instâncias modificadas. Já o segundo grupo avaliou a qualidade dos modelos minerados durante a mineração incremental com a incidência de modificações nas instâncias de processos (elementos adicionados e removidos do modelo).

5.2.1 Geração dos *Logs* de Eventos

Obter dados reais sobre processos não é uma tarefa trivial. Desta forma, foi necessário utilizar um simulador (BURATTIN; SPERDUTI, 2010) para gerar dados sobre a definição de modelos de processo e também os logs de execução destes modelos. Os modelos foram gerados de forma recursiva. Primeiramente, n partes foram geradas. Cada parte era formada por uma atividade (com probabilidade de 50%), uma estrutura paralela (20%) e uma estrutura alternativa (20%) ou uma iteração (10%). Também foram incluídos 5% de ruídos no *log* de todos os traces. Adicionalmente, cada tarefa tem um participante que representa um ator do processo. Para uma estrutura paralela ou alternativa a simulação gerou randomicamente b ramos (*branches*). Um modelo de processo geralmente possui não mais do que 100 tarefas em sua estrutura (AALST; MARUSTER, 2004c), desta forma foi definido que $n = 4$ e $2 \leq b \leq 4$ para limitar a escala da geração do modelo. Desta forma, cada modelo tem ao menos uma iteração, no máximo três estruturas alternativas e no máximo três estruturas paralelas. O simulador randomicamente gera o tempo de execução e espera de uma atividade dentro do processo durante a geração do *log*, o que permitiria também avaliar no log informações como o tempo decorrido da execução de uma atividade. Por fim, cada rota de decisão encontrada (*gateway* XOR/AND-*split*), cada ramificação é escolhida e seguida com a mesma probabilidade.

Outro ponto importante da geração dos modelos é que cada modelo de processo gerado pelo simulador possui também uma versão modificada. Ela foi utilizada para simular a evolução do modelo de processo e realizar a mineração incremental com instâncias de processos modificadas.

Ao final, foram gerados 400 modelos de processos, sendo 200 modelos iniciais, e 200 modelos representando versões modificadas dos modelos iniciais. Além disso, foram gerados 200 arquivos de logs, com uma média de 47,6 tarefas. Em cada conjunto de dados de *log* pertencente a um modelo de processo, foram geradas 500 instâncias simuladas de processos, formadas por 300 novas instâncias, geradas pela execução do modelo inicial, e mais 200 instâncias modificadas, geradas a partir da versão modificada do mesmo processo.

5.2.2 Experimentos com Mineração sem Modificações (Não Incremental)

Para medir a acurácia do método proposto neste trabalho em um cenário não incremental e ainda com dados empíricos, foram utilizadas as mesmas métricas de conformidade da seção 5.1.3. Para realizar este experimento, foram utilizados apenas os *logs* gerados a partir dos modelos de processo iniciais, os quais geraram apenas novas instâncias de processo no *log* (sem instâncias modificadas). O resultado gerado a partir da mineração dos 200 conjuntos de logs pode ser visto na Tabela 5.7. A métrica *Token Fitness* e *Structural Fitness* sugerem que o algoritmo proposto aqui tem precisão um pouco superior ao do algoritmo *α-algorithm* e do algoritmo proposto por Ma et al. Já a métrica *Behavioral Appropriateness* apresentou valores similares aos dos demais algoritmos.

Tabela 5.7: Avaliação de qualidade para *IncrementalMiner*, α-*algorithm* e o método de Ma et al.

Métrica	***IncrementalMiner***	**α-algorithm**	**Ma et al**
Token Fitness	0.998	0.882	0.953
Behavioral Appropriateness	0.863	0.865	0.854
Structural Fitness	1.000	1.000	0.901

5.2.3 Experimentos com Mineração Incremental com Modificações nos Processos

Devido a falta de técnicas para medir a conformidade de modelos durante a mineração incremental, principalmente na ocorrência de traces modificados registrados no *log*, foi necessário a utilização de uma ténica alternativa. Desta forma, para realizar a análise da qualidade dos modelos de processos durante a mineração incremental, foi utilizado a estatística Kappa. O método estatístico foi aplicado similarmente ao apresentado na seção 5.1.5. Kappa deu uma medida quantitativa de concorância entre os modelos de entrada (modelo de processo original e modelo de processo modificado), os quais geraram os registros no *log* (observador 1), e o modelo de processo minerado pelo algoritmo *IncrementalMiner* (observador 2) a partir destes *logs*. Aqui, a medida de concordância define o nível de similaridade entre as estruturas de ambos os modelos de processos (versão final do processo e o processo minerado). Esta medida de concordância foi aplicada aos mesmos aspectos estruturais e semânticos definidos anteriormente na **Tabela 5.5**.

A razão por utilizar Kappa para verificar a similaridade do grafo do processo ao invés de outras técnicas como (DIJKMAN; DUMAS; GARCÍA-BAÑUELOS, 2004) esteve no fato de que foi necessário considerar também aspectos organizacionais do processo, como verificação dos participantes mapeados à atividades do modelo, o qual não faz parte da estrutura do grafo.

Tabela 5.8: Similaridade média entre os modelos resultantes da mineração incremental.

Aspecto	**Precisão**
Mapeamento Atividades	1
Mapeamento Participante	**0,71**
Transições de Entrada	1
Transições de Saída	1
Semântica de Entrada	1
Semântica de Saída	1

O processo de verificação da precisão dos modelos foi realizada conforme apresentado na Figura 5.6. Primeiramente, foi realizado a mineração dos traces iniciais disponibilizados no log (3). Estes traces foram gerados no *log* (2) a partir de modelos originais de processo (1). Como resultado da mineração inicial, foram descobertos modelos de processo que refletem o modelo de entrada (4). Em seguida, foram submetidos ao algoritmo (7) os *logs* com as instâncias modificadas (6) geradas a partir das versões modificadas dos processos (5). O resultado final obtido da mineração foram modelos de processos atualizados (8), onde elementos foram adicionados e removidos. Ao final, os modelos atualizados (resultado final da mineração) e os modelos modifcados (processo modificado) são analisados do ponto de vista da similaridade (9). Na Tabela 5.8, é possível verificar os resultados obtidos. Foram obtidos valores altos para Kappa na maioria dos aspectos, o que sugere que os modelos descobertos e os modelos que geraram os *logs* são similares. Isto significa que a mineração incremental

dos *logs* foi conduzida de forma eficiente para a maioria dos aspectos. É importante observar que o aspecto *Mapeamento de Participantes* foi o único que apresentou valores mais baixos de precisão (média de 0,71). A razão para este valor é que o algoritmo *IncrementalMiner* ainda não suporta a remoção de participantes obsoletos do modelo minerado, como é feito para tarefas e *gateways*.

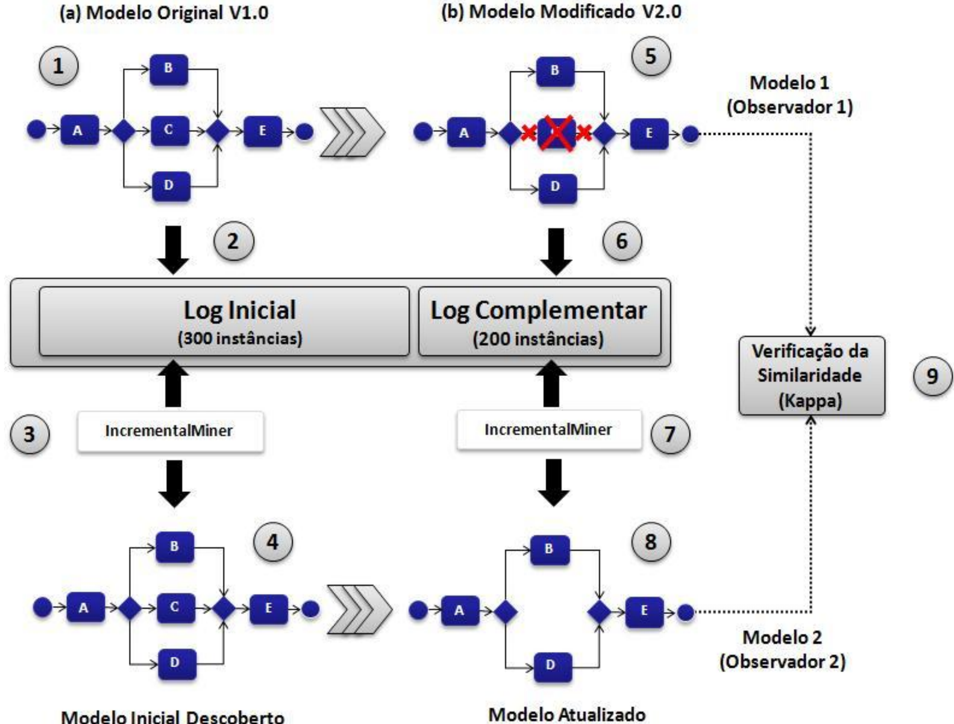


Figura 5.6: Processo de verificação do modelo minerado durante a mineração incremental.

6 CONCLUSÃO

Neste trabalho foi proposta uma abordagem de mineração incremental para extração de modelos de processos de negócio a partir de arquivos de *log* de eventos de um sistema de informação. Esta abordagem utiliza como base, o algoritmo *IncrementalMiner*, um algoritmo que extrai estruturas do processo, de forma incremental, a partir do *log* de execução.

6.1 Principais Contribuições

Na avaliação de performance, o algoritmo *IncrementalMiner* mostrou ser superior para extrair modelos de processos a partir de *logs* de eventos com grande volume de *traces.* O tempo total de processamento dos *logs* pode ser reduzido em 64% durante a mineração incremental e foi quatro vezes (400%) mais rápido que o algoritmo Alpha++ e oitenta vezes (8000%) superior que o algoritmo *HeuristicMiner* durante a mineração incremental. No processamento de *logs* completos (sem mineração incremental), o algoritmo *IncrementalMiner* mostrou-se (80%) mais rápido que o algoritmo Alpha++ e vinte e oito vezes (2800%) mais rápido que o algoritmo *HeuristicMiner,* isto foi possível devido a utilização de estruturas de dados mais otimizadas, como árvores binárias balanceadas AVL.

Na avaliação da qualidade, os modelos extraídos com o *IncrementalMiner* mostraram boa acurácia (igual ou superior) quando comparados aos algoritmos mais tradicionais de mineração de processo. Já em relação aos algoritmos incrementais, o algoritmo *IncrementalMiner* mostrou ser superior nos aspectos envolvendo modificações no processo (ex: remoção de atividades). Isto se deve ao fato de somente o algoritmo *IncrementalMiner* ser capaz de minerar este tipo de modificação no processo durante a mineração incremental dos *logs*. Além disso, foi demonstrado que a mineração incremental pode ser útil durante a identificação de modelos de processos em sistemas legados, permitindo a extração inicial de resultados parciais (ex: análise por módulo) e incrementando-os à medida que novos processos são necessários ou à medida que estes processos são alterados. Além disso, neste contexto é possível reduzir significamente o tempo total de processamento, pois apenas novos eventos são submetidos ao algoritmo ao invés de todo o conteúdo do arquivo de *log*.

Desta forma, a principal contribuição deste trabalho foi a criação de um mecanismo incremental de mineração capaz de suportar os principais aspectos da aprendizagem incremental de modelos de processos. O uso de mineração de processos permitiu também a extração de estruturas de processo importantes, tais como participantes e atividades concorrentes, as quais não eram possíveis de se extrair em outras abordagens de extração de processos a partir de sistemas de informação.

6.2 Trabalhos Futuros

É possível citar cinco melhorias futuras que podem ser trabalhadas sobre a abordagem de mineração incremental de processos proposta neste trabalho. A primeira esta relacionada à possibilidade de melhoria na correlação de estruturas de processo parciais que fazem parte de um mesmo processo colaborativo, extraídas a partir de um sistema de informação. Atualmente, a variável *traceId* (variável que realiza a ligação de todas as atividades que pertencem à mesma instância de execução ou *trace* de um processo), variável esta utilizada na instrumentação do código fonte do sistema, depende de outras variáveis que estão presentes durante toda a execução do sistema para realizar a correlação dos eventos de um *trace* como, por exemplo, um identificador de pedido, paciente, um número de contrato, etc. Porém, muitas vezes não há a existência de tais variáveis, ou as mesmas não estão disponíveis durante toda a execução do processo, gerando pequenas estruturas parciais de processo ao invés de estruturais maiores, como pode ser visto na validação do modelo de processo extraído do sistema de informação do estudo de caso, descrito na seção 5.1.5. Esta melhoria poderia ser realizada gravando-se informações complementares no *log* como, por exemplo, os parâmetros de entrada e saída das regras de negócio. Com estas informações, poderia-se transformar os dados das regras (atividades) e processos em ontologias e por fim aplicar algoritmos de matching de ontologias a estas estruturas como, por exemplo, o algoritmo Similarity Flooding (MELNIK et al, 2002), para então unificar as ontologias similares em estruturas maiores e mais completas.

A segunda melhoria está na extração e integração de condições de decisão (ex: regras de negócio contendo expressões condicionais de controle de fluxo) ao modelo de processo final. Estes dados podem ser obtidos utilizando técnicas de mineração de decisões (ROZINAT; AALST, 2006a), descritas na Seção 2.4. Com a utilização destas técnicas é possível enriquecer o modelo de processo, adicionando regras condicionais a cada controle de fluxo do processo.

A terceira melhoria aborda o suporte a remoção de participantes do processo que não executam mais determinadas atividades (ver item *vi)* da seção 4.4.1). Atualmente, o algoritmo *IncrementalMiner* não suporta este tipo de atualização durante a mineração incremental, realizando apenas a remoção de elementos obsoletos como tarefas, controles de fluxo e transições.

A quarta melhoria está relacionada a integração do algoritmo incremental à ferramenta ProM (van DONGEN et al, 2005b). A ferramenta ProM reúne em um ambiente integrado e constantemente atualizado por plug-ins, os principais algoritmos e técnicas de mineração de processos, desenvolvidas por diferentes pesquisadores da área. Atualmente a ferramenta ProM não possui muitas técnicas incrementais implementas. Desta forma, a implementação na ferramenta das técnicas apresentadas aqui pode agregar valor à esta pesquisa, além de poder propagar o conhecimento gerado aqui a outras.

A quinta e última melhoria está associada a geração automática dos cenários de execução utilizados para coordenar a execução do sistema legado e geração do *log* (ver seção 4.2). Neste trabalho, os cenários foram criados manualmente com a ajuda do usuário, necessitando de um tempo considerável para isto. Assim, este processo pode ser significamente melhorado com a utilização de abordagens de geração automática de dados de teste (KHAMIS et al, 2012), (GIRGIS et al, 2005). Nestas abordagens, os dados de entrada dos cenários de testes são gerados automaticamente utilizando a

combinação de análise estática, dinâmica e algoritmos genéticos. Com isto, não haveria a necessidade de interação humana nesta parte do processo, o que pode aumetar drasticamente a agilidade e precisão na criação dos cenários que serão executados pelo sistema para geração do *log* de execução.

REFERÊNCIAS

ABRAMS, C. SCHULTE, R. W. **Service-Oriented Architecture Overview and Guide to SOA Research**. Technical report G00154463, Gartner Research, jan. 2008.

ADELSON-VELSKII, G.; LANDIS, E. M. **An algorithm for the organization of information.** Proceedings of the USSR Academy of Sciences 146: p. 263-266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, v.3, p. 1259-1263, 1962.

AGRAWAL, R. GUNOPULOS, D. LEYMANN, F. **Mining Process Models from Workflow Logs**. In: SIXTH INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, p. 469-483, [S.l.], 1998.

BISBAL, J. LAWLESS, D. WU, B. GRIMSON, J. **Legacy Information Systems: Issues and Directions.** IEEE SOFTWARE, set, [S.l.], 1999.

BONDI, A. B. **Characteristics of scalability and their impact on performance.** Proceedings of the 2ND INTERNATIONAL WORKSHOP ON SOFTWARE AND PERFORMANCE, Ottawa, Ontario, Canada, 2000. p. 195-203.

BOSE, R. P. van der AALST, W. M. P. ZLIOBAITÉ, I. PECHENIZKIY, M. **Handling concept drift in process mining.** CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, CAISE'11**.** Londres, Reino Unido, 2011. pp 391-405.

BREIMAN, L. FRIEDMAN, J. H. OLSHEN, R. A. STONE, C. J. **Classification and regression trees.** Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, Califórnia, EUA, 1984.

BURATTIN, A. SPERDUTI, A. **PLG: a Framework for the Generation of Business Process Models and their Execution Logs.** In: PROCEEDING BPI WORKSHOP 2010, Stevens Institute of Technology, Hoboken, New Jersey, EUA, 2010.

CHIANG, C. C. **Extracting business rules from legacy systems into reusable components**. IEEE/SMC INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEMS ENGINEERING, Los Angeles, Califórnia, EUA, 2006. p. 6.

COHEN, J. **A coefficient of agreement for nominal scales, Educational and Psychological Measurement.** v.20, n.1, p.37-46, [S.l.], 1960.

COOK, J. E. WOLF, A. L. **Discovering Models of Software Processes from Event-Based Data.** ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY, New York, EUA v.7, n.3, p. 215-249, 1998.

DESEL, J. ESPARZA, J. Free **Choice Petri Nets**. v.40 of Cambridge Tracts in Theoretical Computer Science. Cambridge, UK, Cambridge University Press, 1995.

DIJKMAN, R. DUMAS, M. GARCÍA-BAÑUELOS, L. **Graph Matching Algorithms for Business Process Model Similarity Search.** BUSINESS PROCESS MANAGEMENT JOURNAL (BPM'09), Berlin, p. 48-63, Springer-Verlag, 2009.

GEISSER, S. **Predictive Inference.** New York: Chapman and Hall, New York, EUA, 1993.

GIRGIS, M. R. **Automatic test data generation for data flow testing using a genetic algorithm.** JOURNAL OF UNIVERSAL COMPUTER SCIENCE, v.11, n.6, p. 898-915, 2005.

GUNTHER, C.W. RINDERLE-MA, S. REICHERT, M. van Der AALST W.M.P. **Using process mining to learn from process changes in evolutionary systems.** INTERNATIONAL JOURNAL OF BUSINESS PROCESS INTEGRATION AND MANAGEMENT, v.3, n.1, p. 61-78, Inderscience, [S.l.], 2008.

HALILI, E. H. **Apache JMeter.** Packt Publishing Ltd., Birmingham, UK, 2008.

HAN, J. PEI, J. YIN, Y. **Mining frequent patterns without candidate generation.** INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD'00**.** Dallas, Texas, EUA, Vol. 29, Number 2, pp 1-12, 2000.

HERNÁNDES-LEÓN, R. HERNÁNDES-PALANCAR, J. CARRASCO-OCHOA, J.A. MARTÍNEZ-TRINIDAD, J, F. **A Novel Incremental Algorithm for Frequent Itemsets Mining in Dynamic Datasets.** PROGRESS IN PATTERN RECOGNITION, IMAGE ANALYSIS AND APPLICATIONS**,** Havana, Cuba, CIARP'08, pp 145-152, 2008.

HOPCROFT, J. TARJAN, R. **Algorithm 447: Efficient Algorithms for Graph Manipulation.** *Comm. ACM*, New York, EUA, v.16, p. 372-378, 1973.

HUANG, H. TSAI, W. T. BHATTACHARYA, S. CHEN, X. P. WANG, Y. SUN, J. **Business rule extraction from legacy code**. COMPSAC '96, Seoul, Coréia do Sul, p. 162-167, 1996.

JABLONSKI, S. BUSSLER, C. **Workflow Management: Modeling Concepts, Architecture, and Implementation.** INTERNATIONAL THOMSON COMPUTER PRESS, London, UK, 1996.

KALSING, A. C. IOCHPE, C. THOM, L. H. **An Incremental Process Mining Algorithm.** INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS - ICEIS'10, Ilha da Madeira, Portugal, v.1, p. 263-268, jun. 2010a.

KALSING, A. C. NASCIMENTO, G.S. IOCHPE, C. THOM, L.H. **An Incremental Process Mining Approach to Extract Knowledge from Legacy Systems.** ENTERPRISE DISTRIBUTED OBJECT COMPUTING CONFERENCE – EDOC'10, 14th IEEE International, Vitória, ES, Brasil, p 79-88, out. 2010b.

KALSING, A. C. IOCHPE, C. THOM, L.H. NASCIMENTO, G.S. **Evolutionary Learning of Business Process Models from Legacy Systems using Incremental**

Process Mining. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSM'12), Trento, Itália, 2012 (em avaliação).

KHAMIS, A. M. GIRGIS, M. R. GHIDUK, A. S. **Automatic software test data generation for spanning sets coverage using genetic algorithms.** COMPUTING AND INFORMATICS (CAI'12), v.26, n.4, p. 383-401, 2012.

KINDLER, E. RUBIN, V. SCHAFER, W. **Incremental Workflow mining based on Document Versioning Information**. UNIFYING THE SOFTWARE PROCESS SPECTRUM JOURNAL, p 287-301, Springer, 2006.

KNUTH, D. **The Art of Computer Programming.** Sorting and Searching. v.3, p. 506-542, 1973.

LANDIS, J. R. KOCH, G. G. **The measurement of observer agreement for categorical data.** INTERNATIONAL BIOMETRIC SOCIETY, v.33. p. 159-174, 1977.

LEYMANN, F. ROLLER, D. **Production Workflow: Concepts and Techniques.** Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

LIU, K. ALDERSON, A. QURESHI, Z. **Requirements recovery from legacy systems by analysing and modelling behavior**. IEEE INTERNATIONAL CONFERENCE ON SOFT. MAINTENANCE, p. 3-12, 1999.

MA, H. TANG, Y. WU, L. **Incremental Mining of Processes with Loops.** INTERNATIONAL JOURNAL ON ARTIFICIAL INTELLIGENCE TOOLS, v.20, n.1, p. 221-235, 2011.

MEDEIROS, A. K. WEIJTERS, A. J. M. M. van der AALST, W.M.P. **Genetic process mining: an experimental evaluation.** DATA MINING AND KNOWLEDGE DISCOVERY JOURNAL, v.14, n.2, p. 245-304, Springer, 2007.

MELNIK, S. GARCIA-MOLINA, H. RAHM, E. **Similarity flooding: A versatile graph matching algorithm and its application to schema matching.** 18TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, p. 117-128, IEEE, 2002.

MITCHELL, T. M. **Machine Learning and Data Mining.** MAGAZINE COMMUNICATIONS OF THE ACM, v.42, 1999.

MURATA, T. **Petri Nets: Properties, Analysis and Applications.** in: Proceedings of the IEEE, v.77, n.4, abr. 1989.

NASCIMENTO, G. S. IOCHPE, C. THOM, L. H. REICHERT, M. **A Method for Rewriting Legacy Systems using Business Process Management Technology.** ICEIS, v.3, p. 57-62, 2009.

NASCIMENTO, G. S. IOCHPE, C. KALSING, A. C. THOM, L. MOREIRA, A. F. **Identifying Business Rules to Legacy Systems Reengineering based on BPM and SOA**. LECTURE NOTES IN COMPUTER SCIENCE, 2012.

NATIS, Y. V. PEZZINI, M. SCHULTE, R. W. IIJIMA, K. **Predicts 2007: SOA Advances**. Technical report G00144445, GARTNER RESEARCH, nov. 2006.

PUTRYCZ, E. KARK, A.W. **Recovering Business Rules from Legacy Source Code for System Modernization**. LECTURE NOTES IN COMPUTER SCIENCE, v.4824, p. 107-118, Springer, 2007.

QUINLAN, J. R. **C4.5: Programs for Machine Learning.** Morgan Kaufmann Publishers, 1993.

RADAELLI JUNIOR, W. A. NASCIMENTO, G. S. IOCHPE, C. **Survey and Proposal of a Method for Business Rules Identification in Legacy Systems Source Code and Execution Logs**. 13TH INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, 2011, Beijing. ICEIS 2011 - Proceedings of the 13th International Conference on Enterprise Information Systems, v. 3. p. 207-213, 2011.

ROZENBURG, G. ENGELFRIET, J. **Elementary Net Systems.** in: W. Reisig, G. Rozenberg (Eds.), Lectures on Petri Nets I: Basic Models - Advances in Petri Nets, v.1491 of Lecture Notes in Computer Science, Springer , p. 12-121, 1998.

ROZINAT, A. MEDEIROS, A. K. GUNTHER, C.W. WEIJTERS, A. J. M. M. van der AALST, W.M.P. **Towards an evaluation framework for process mining algorithms.** BPM Center Report BPM-07-06, BPMcenter. Org, 2007

ROZINAT, A. van der AALST, W. M. P. **Conformance Testing: measuring the fit and appropriateness of event logs and process models**. BPM 2005 WORKSHOPS (WORKSHOP ON BUSINESS PROCESS INTELLIGENCE), Berlin, v.3812, p. 163-176, Springer-Verlag, 2005.

ROZINAT, A. van der AALST, W. M. P. **Decision Mining in Business Process**. INTERNATIONAL CONFERENCE ON BUSINESS PROCESS MANAGEMENT (BPM 2006a).

ROZINAT, A. van der AALST, W.M.P. **Conformance testing: Measuring the fit and appropriateness of event logs and process models.** BUSINESS PROCESS MANAGEMENT WORKSHOPS, p. 63-176, Springer, 2006b.

SILVA, R., ZHANG, J. SHANAHAN, J.G. **Probabilistic workflow mining**. In: KDD, p. 275-284, 2005.

SNEED, H. M. ERDOS, K. **Extracting business rules from source code**. PROCEEDING OF THE FOURTH IEEE WORKSHOP ON PROGRAM COMPREHENSION, p. 240-247, 1996.

SPIRTES, P. GLYMOUR, C. SCHEINES, R. **Causation, Prediction and Search.** Cambridge University Press, 2000.

SUN, W. LI, T. PENG, W. SUN, T. **Incremental Workflow Mining with Optional Patterns and Its Application to Production Printing Process.** INTERNATIONAL

JOURNAL OF INTELLIGENT CONTROL AND SYSTEMS, v.12, n.1, p. 45-55, 2007.

THORBURN, W. M. **Occam's razor,** Mind, 24, p. 287288, 1915.

van der AALST, W. M. P. **Business Alignment: using process mining as a tool for delta analysis**. PROCEEDINGS OF THE 5TH WORKSHOP ON BUSINESS PROCESS MODELING. Development and Support (BPMDS04). v.2 of Caise04 Workshops. Riga Technical University. Latvia. 2004a.

van der AALST, W. M. P. MARUSTER, L. **Workflow Mining: discovering process models from event logs**. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, vol.16 n.9, p.1128-1142, 2004c.

van der AALST, W. M. P. van DONGEN, B. HERBST, J. MARUSTER, L. SCHIMM, G. WEIJTERS, A. **Workflow Mining: a survey of issues and approaches**. DATA AND KNOWLEDGE ENGINEERING, 2003.

van der AALST, W. M. P. WEIJTERS, A. **Process Mining: a research agenda.** Department of Technology Management, Eindhoven University of Technology P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands, 2004b.

van der AALST, W.M.P. DESEL, J. OBERWEIS, A. (Eds.), **Business Process Management: Models, Techniques, and Empirical Studies.** LECTURE NOTES IN COMPUTER SCIENCE, v.1806, Springer-Verlag, Berlin, 2000.

van der AALST, W.M.P. van HEE, K. M. **Workflow Management: Models, Methods, and Systems.** MIT press, Cambridge, MA, 2002.

van DONGEN, B. F. van der AALST W. M. P. **A Meta Model for Process Mining Data.** Proceedings of the CAiSE, v.5, p. 309-320, 2005a.

van DONGEN, B. MEDEIROS, A. K. VERBEEK, H. WEIJTERS, A.J.M.M. van der AALST, W. M. P. **The ProM framework: a new era in process mining tool support**. APPLICATIONS AND THEORY OF PETRI NETS JOURNAL. p. 444-454, Springer, 2005b.

van GLABBEEK, R.J. WEIJLAND, W.P. **Branching time and abstraction in bisimulation semantics.** JOURNAL OF THE ACM vol.43, n.3, p. 555-600, 1996.

VERBEEK, H.M.W. BASTEN, T. van der AALST, W.M.P. **Diagnozing workflow processes using Woflan.** THE COMPUTER JOURNAL, vol.44, n.4, p. 246-279, 2001.

WANG, C. ZHOU, Y. CHEN, J. **Extracting Prime Business Rules from Large Legacy System**. INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFT. ENGINEERING, v.2, p. 19-23, 2008.

WEIJTERS, A. van der AALST, W.M.P. MEDEIROS, A. K. **Process Mining with the HeuristicsMiner Algorithm.** Technische Universiteit Eindhoven, Tech. Rep. WP, v.166, 2006.

WEN, L. van der AALST, W.M.P. WANG, J. SUN, J. **Mining process models with non-free-choice constructs.** DATA MINING AND KNOWLEDGE DISCOVERY JOURNAL, v.15, n.2, p. 145-180, Springer, 2007.

WEN, L. WANG, J. van der AALST, W.M.P. HUANG, B., SUN J. **A novel approach for process mining based on event types.** JOURNAL OF INTELLIGENT INFORMATION SYSTEMS, v.32, n.2, p. 163-190, Springer, 2009.

WITTEN, I. H. FRANK, E. **Data Mining: Practical Machine Learning Tools and Techniques**. 3d Edition, Morgan Kaufmann, 2011.

ZOU, Y. HUNG, M. An **Approach for Extracting Workflows from E-Commerce Applications.** 14th IEEE ICPC 2006, p. 127-136, 2006.

ZOU, Y. LAU, T. C. KONTOGIANNIS, K., TONG, T. MCKEGNEY, R. **Model-driven business process recovery.** 11TH WORKING CONFERENCE ON REVERSE ENGINEERING, p. 224-233, 2004.

Em meio eletrônico

BUSINESS RULE GROUP (BRG). **Defining Business Rules – What Are They Really?,** Disponível em: http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf, 2000.

GEERTZEN, J. **Cohen's Kappa tool for more than two annotators with multiple classes.** Disponível em: http://cosmion.net/jeroen/software/kappa/, 2010

KRILL, P. **The future's bright ... the future's Cobol**. Acessado em 30 de 2009, Disponível em: http://www.computerworlduk.com/technology/applications/ enterprise/ indepth/index.cfm?articleid=95, 2006.

OASIS. **Reference Architecture for Service Oriented Architecture Version 1.0.** 2008. Disponível em: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-pr-01.pdf.

OMG. **Business Process Management Notation**. Version 1.2, Acessado em Fev. 2009. Disponível em: <http://www.omg.org/docs/ formal/09-01-03.pdf>.

WORKFLOW MANAGEMENT COALITION (WfMC), **Process Definition Interface,** WFMC-TC-1025, 2008. Disponível em:http://www.wfmc.org/index.php?option=com_docman&task=doc_download&Itemid=72&gid=132

WORKFLOW MANAGEMENT COLATION (WfMC). **Terminology & Glossary.** 1999. Disponível em: http://www.wfmc.org/index.php?option=com_docman&task=doc_download&gid=93&Itemid=72

WEIDEN, M., HERMANS, L. SCHREIBER, G. van der ZEE, S. **Classification and Representation of Business Rules.** Acessado em Set. 2009. Disponível em: http://www.omg.org/docs/ad/02-12-18.pdf , 2002.