AUGUST BAUMGARTNER NETO

**Extração de Casos de Teste Utilizando Redes de Petri Hierárquicas
e Validação de Resultados Utilizando OWL**

São Paulo
2015

AUGUST BAUMGARTNER NETO

**Extração de Casos de Teste Utilizando Redes de Petri Hierárquicas e
Validação de Resultados Utilizando OWL**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do título de Mestre em
Engenharia.

São Paulo
2015

AUGUST BAUMGARTNER NETO

**Extração de Casos de Teste Utilizando Redes de Petri
Hierárquicas e Validação de Resultados Utilizando OWL**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do título de Mestre em
Engenharia.

Área de Concentração:
Engenharia de Controle e Automação
Mecânica

Orientador: Prof. Dr. José Reinaldo Silva
Prof. Dr. Marcos R. Pereira
Barretto

São Paulo
2015

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

Baumgartner Neto, August

Extração de Casos de Teste Utilizando Redes de Petri Hierárquicas e Validação de Resultados Utilizando OWL / A. Baumgartner Neto -- versão corr.
- São Paulo, 2015.
95 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Redes de Petri 2.Web Semântica 3.Ontologias 4.Casos de Testes
I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos II.t.

RESUMO

Este trabalho propõe dois métodos para teste de sistemas de *software*: o primeiro extrai ideias de teste de um modelo desenvolvido em rede de Petri hierárquica e o segundo valida os resultados após a realização dos testes utilizando um modelo em OWL-S. Estes processos aumentam a qualidade do sistema desenvolvido ao reduzir o risco de uma cobertura insuficiente ou teste incompleto de uma funcionalidade. A primeira técnica apresentada consiste de cinco etapas: i) avaliação do sistema e identificação dos módulos e entidades separáveis, ii) levantamento dos estados e transições, iii) modelagem do sistema (*“bottom-up”*), iv) validação do modelo criado avaliando o fluxo de cada funcionalidade e v) extração dos casos de teste usando uma das três coberturas de teste apresentada. O segundo método deve ser aplicado após a realização dos testes e possui cinco passos: i) primeiro constrói-se um modelo em OWL (*“Web Ontology Language”*) do sistema contendo todas as informações significativas sobre as regras de negócio da aplicação, identificando as classes, propriedades e axiomas que o regem; ii) em seguida o status inicial antes da execução é representado no modelo através da inserção das instâncias (indivíduos) presentes; iii) após a execução dos casos de testes, a situação do modelo deve ser atualizada inserindo (sem apagar as instâncias já existentes) as instâncias que representam a nova situação da aplicação; iv) próximo passo consiste em utilizar um *“reasoner”* para fazer as inferências do modelo OWL verificando se o modelo mantém a consistência, ou seja, se não existem erros na aplicação; v) finalmente, as instâncias do status inicial são comparadas com as instâncias do status final, verificando se os elementos foram alterados, criados ou apagados corretamente. O processo proposto é indicado principalmente para testes funcionais de caixa-preta, mas pode ser facilmente adaptado para testes em caixa-branca. Obtiveram-se casos de testes semelhantes aos que seriam obtidos em uma análise manual mantendo a mesma cobertura do sistema. A validação provou-se condizente com os resultados esperados, bem como o modelo ontológico mostrou-se bem fácil e intuitivo para aplicar manutenções.

Palavras-chave: Teste em sistemas de software, Validação de resultados de testes, Redes de Petri, Representação de domínios em OWL, Criação de casos de testes.

ABSTRACT

This paper proposes two test methods for system software testing: the first one extracts test workflow processes from a model developed in Hierarchical Petri Nets and the other validates results after test execution using a domain model in OWL-S. Both processes increase the quality of the system developed by reducing the risk of having an insufficient coverage or an incomplete functionality test. The first technique consists of five steps: i) system evaluation and identification of separable sub modules and entities, ii) identification of states and transitions, iii) system modeling (bottom-up), iv) validation of the created model by evaluating the workflow for each functionality, and v) extraction of test cases using one of the three test coverage presented. The second method must be applied after the execution of the previous method and has also five steps: i) first a system model in OWL (Web Ontology Language) is built containing all significant information and business rules of the application; ii) then, the initial status before the test execution is represented in the model by the insertion of the instances (individuals) presented; iii) after the execution of test cases, the state model is updated by inserting (without deleting already existing instances) new instances to represent the domain state after test; iv) in the next step we use a reasoner to make OWL model checking inferences to prove model consistency, that is, if there is no error in the application; finally, the initial status instances are compared with the final status in order to verify if these instances have been changed, created or deleted correctly. The process is indicated for black-box functional tests, but can be easily adapted for white-box tests. There were obtained test cases similar to those that will be obtained in a manual analysis keeping the same test coverage. Validation has proved to be consistent compared to the expected results. Also, the ontological model has showed to be easy and intuitive for maintenance.

Keywords: System software testing, Validation of test results; Petri Nets. Domain representation in OWL; Generation of test processes.

LISTA DE ILUSTRAÇÕES

Figura 1 - Representação das etapas dos métodos aplicada dentro do processo de teste	50
Figura 2 - Exemplo de transição que é habilitada por um lugar externo	51
Figura 3 - Exemplo de situação onde a transição afetada por um lugar em outra entidade que fornece uma marca.....	52
Figura 4 - Situação onde a transição na Entidade_Transicao_Afeta afeta um lugar em outra entidade	53
Figura 5 - Transição criando duas entidades simultaneamente	54
Figura 6 - Figura apresentando uma transição que é influenciada pela morte de uma entidade	54
Figura 7 - Desenho apresentando uma transição externa afetando duas entidades	56
Figura 8 - Exemplo de mudança no axioma ao trocar o “ <i>reasoner</i> ”. (a) Para o “ <i>reasoner</i> ” FaCT++. (b) Para o “ <i>reasoner</i> ” Pellet	64
Figura 9 - 1º Nível - Rede de Petri Macro da aplicação.....	65
Figura 10 - 2º Nível - Rede de Petri representando a primeira parcela.....	66
Figura 11 - 2º Nível - Rede de Petri representando a última parcela	67
Figura 12 - 2º Nível - Rede de Petri representando o Contrato.....	67
Figura 13 - Modelo ontológico das classes da aplicação com destaque para as regras de negócio.....	68
Figura 14 - Modelo ontológico das classes da aplicação com destaque para as classes “Erro” e “Partitions”.....	69
Figura 15 - Modelo ontológico das classes da aplicação com destaque para a classe “CasosdeTestes”.....	69
Figura 16 - Axioma da classe CT003	69
Figura 17 - Axioma das classes de casos de testes. (a) Classe “CT005”. (b) Classe “CT021”. (c) Classe “CT020”	70
Figura 18 - Axioma das classes de casos de testes reversas. (a) Classe “CT005_Reverso”. (b) Classe “CT021_Reverso”. (c) Classe “CT020_Reverso”	70
Figura 19 - Regras em SWRL para as classes “CT022” e “CT022_Reverso”	70
Figura 20 - Expressões da classe ContratoCredito	71
Figura 21 - Axiomas da classe Parcela	72
Figura 22 - Axiomas de um indivíduo da classe ContratoCredito.....	73

Figura 23 - Expressões de um indivíduo da classe Parcela.....	73
Figura 24 - Representação de contratos sem uma das propriedades. (a) Contrato com valor zerado. (b) Contrato sem estado. (c) Contrato sem ID	78
Figura 25 - Classe CT022 sem o Contrato Cerrado3.....	78
Figura 26 - Classe CT022_Reverso com o Contrato Cerrado3.....	79
Figura 27 - Explicação da inconsistência da associação de um estado inexistente..	80
Figura 28 - Cerrado20 conforme é apresentado no modelo.....	80
Figura 29 - Exemplos de contratos com valor errado. (a) Contrato com uma parcela. (b) Contrato com duas parcelas	80
Figura 30 - Cerrado36 classificado como um individuo do “CT022_Reverso”	81
Figura 31 - Cerrado38 classificado como um individuo da classe “CT022_Reverso”	81
Figura 32 - Contrato C1 classificado como instância da classe “CT022”	82
Figura 33 - Detalhe do contrato C1	82
Figura 34 - Erro de inferência para um ContratoCredito sem nome.....	82
Figura 35 - Erro para uma Parcela sem nome	83
Figura 36 - Inconsistência da existência de um contrato sem parcelas	83
Figura 37 - Inconsistência de um contrato sem a declaração de uma das Parcelas.	83
Figura 38 - Exemplo positivo de execução do caso de teste 003.....	84
Figura 39 - Contrato com estado incorreto.....	84
Figura 40 - Inconsistência no estado do contrato.....	85
Figura 41 - Tentativa de inferência para uma parcela sem um estado associado	85
Figura 42 - Exemplo de parcela sem Identificador	85
Figura 43 - Incoerência da existência de uma parcela com valor igual a zero	86
Figura 44 - Contradição na existência de uma parcela com valor negativo	86
Figura 45 - Erro de uma parcela sem um contrato associado	86
Figura 46 - Comparação de uma execução com sucesso mostrando os indivíduos antes e depois da execução.....	87
Figura 47 - Comparação de uma execução com erro mostrando os indivíduos antes e depois da execução (o valores não deveriam ser modificados).....	88

LISTA DE TABELAS

Tabela 1 - Tabela de referência (Suporte, Ferramentas e Técnica).....	24
Tabela 2 - Tabela de referência (Cobertura, Tipo de Rede, Metodologia e Alvo)	25
Tabela 3 - Exemplo de mapeamento de dados.....	59
Tabela 4 - Tabela de casos de testes obtidos no 1º Nível.....	74
Tabela 5 - Tabela de casos de testes obtidos da Rede de Petri representando a primeira parcela.....	75
Tabela 6 - Tabela de casos de testes obtidos da Rede de Petri representando a última parcela.	75
Tabela 7 - Tabela de casos de testes obtidos da Rede de Petri representando o contrato.	75
Tabela 8 - Mapeamento de massa para o estudo de caso	77

SUMÁRIO

1	INTRODUÇÃO.....	10
1.1	OBJETIVO	12
1.2	ESTRUTURA DO TRABALHO.....	12
2	REVISÃO BIBLIOGRÁFICA	13
2.1	REDES DE PETRI	13
2.2	REPRESENTAÇÃO DE CONHECIMENTO.....	16
2.2.1	RDF – Resource Description Framework	16
2.2.2	OWL – Ontology Web Language	17
2.3	TESTE EM SOFTWARE	19
2.3.1	Abrangência de Teste.....	20
2.3.2	Técnicas de Teste Funcional.....	21
2.3.2.1	Técnicas Caixa-Preta (Especificação).....	21
2.3.2.2	Técnicas Caixa-Preta (Experiência).....	22
2.3.2.3	Técnicas Caixa-Branca	23
2.4	TRABALHOS CORRELADOS	23
2.4.1	Análises sobre base bibliográfica	23
2.4.2	Teste Funcional com Redes de Petri.....	26
2.4.2.1	Rede Lugar/Transição	27
2.4.2.2	Redes Predicado/Transição	27
2.4.2.3	Rede de Petri Temporizada	29
2.4.2.4	Redes de Petri Temporizada e Priorizada (PrTPN)	30
2.4.2.5	Redes de Petri Hierárquicas Tempo-Estendidas	31
2.4.2.6	Redes de Petri Coloridas	33
2.4.2.7	Redes Controladas por Evento	37
2.4.2.8	Outras Redes	38

2.4.3	Aplicações de Ontologias a Teste de Software	42
3	MODELO PROPOSTO	50
3.1	VISÃO GERAL	50
3.2	REPRESENTAÇÃO DA DINÂMICA DO SISTEMA.....	51
3.3	REPRESENTAÇÃO DO PROPÓSITO DO SISTEMA ATRAVÉS DE ONTOLOGIA.....	56
3.4	EXTRAÇÃO DE IDEIAS DE TESTES	58
3.5	VERIFICAÇÃO BASEADA EM ONTOLOGIA	59
4	ESTUDO DE CASO.....	62
4.1	CARACTERIZAÇÃO DO SISTEMA A SER TESTADO.....	62
4.2	REPRESENTAÇÃO DINÂMICA DO SISTEMA.....	65
4.3	REPRESENTAÇÃO DO NEGÓCIO ATRAVÉS DE ONTOLOGIAS.....	68
4.4	EXTRAÇÃO DOS CASOS DE TESTES	73
4.5	VERIFICAÇÃO BASEADA EM ONTOLOGIA	76
5	CONCLUSÃO.....	89
	REFERÊNCIAS.....	91

1 INTRODUÇÃO

Em um processo de desenvolvimento de sistemas *software* é inevitável o surgimento de falhas ao longo da fase inicial de desenvolvimento, o que leva a uma forte atuação em prol da qualidade destes sistemas. Quanto maior o tamanho do sistema e/ou sua complexidade maior a probabilidade de falhas, que se não devidamente tratadas, podem se converter em risco para os usuários e talvez para o próprio sistema. Com base na experiência no mercado atual, as empresas, em especial, as empresas cuja automação do negócio possui elevada dependência de sistemas de informação, como bancos, financeiras, telecomunicações e desenvolvedoras, investem cada vez mais na qualidade destes sistemas. Um *software* não devidamente homologado pode chegar à produção contendo erros graves que podem, por exemplo, bloquear o sistema que efetua transações de cartões de crédito gerando prejuízos significativos para a empresa, ou em outro exemplo, não ativar corretamente o sistema freio ABS de um carro, podendo causar acidentes (ISTQB, 2011, p. 11).

Representar um *software* através de um modelo bem estruturado melhora a compreensão dos objetivos de todas as áreas envolvidas (negócio, desenvolvimento, qualidade, usuários,...). Porém, o processo para elaboração de um modelo demanda tempo, cuidado e conhecimento específico agravando-se consideravelmente quando o *software* é extenso ou possui regras de negócio complexas. Nesse último caso, a criação de um modelo prático e inteligível demanda ferramentas apropriadas, e se possível, de fácil manutenção, permitindo a atualização do modelo conforme a evolução dos requisitos.

Porém, criar modelos práticos de sistemas complexos para depois extrair casos de testes não é trivial, por isso muitas empresas apelam para soluções práticas que consistem geralmente em utilizar uma equipe de desenvolvedores e especialistas na área de testes para analisar os requisitos e extrair os casos de testes essenciais, minimizando os riscos de testes críticos não cobertos. Essa atividade também reduz a probabilidade de erro na identificação dos casos de teste, uma vez que o foco está nos processos mais críticos do sistema, mas, por outro lado, tem a desvantagem de também consumir bastante tempo e de ter a identificação dos casos de teste

atrelada ao desempenho dos líderes da equipe, tornando o processo irregular conforme observado em diversas empresas atuais.

Além dessa questão, o processo de teste corresponde aproximadamente a metade do custo do desenvolvimento de um sistema de *software* (ELBERZHAGER, ROSBACH, *et al.*, 2012, p. 1092), sendo assim, é imperativo a busca por novos que resultem na melhoria da qualidade dos testes, aumentando sua abrangência e eficácia, reduzindo custos e/ou minimizando o tempo de execução das atividades relacionadas ao teste. Seguindo esse pensamento várias ferramentas foram criadas, modificadas ou simplesmente utilizadas de maneira diferenciada para o desenvolvimento de novos métodos, entre essas ferramentas pode-se citar Redes de Petri, Autômatos, Diagramas UML (“*Unified Modeling Language*”) e Ontologias.

As Redes de Petri vêm ganhando espaço significativo na análise e elaboração dos casos de testes devido a sua flexibilidade, eficiência e relativa facilidade de entendimento. Como, por exemplo, temos os trabalhos recentes de Thiel (THIEL e DERICHSWELER, 2011) e Sun (SUN, YE e LIU, 2011) para otimização da quantidade de casos de testes e Cai (CAI, ZHANG e LIU, 2010), Mirzaeian (MIRZAEIAN, MOJAVERI, *et al.*, 2010), Zhenyu Liu (LIU, LIU, *et al.*, 2010), Jing Liu (LIU, YE e LE, 2011) e Neto (NETO, 2013) para extração de casos de testes. Por serem baseadas em um modelo de estados e transições podem facilmente representar um processo de um sistema de *software*, de onde podem extrair com facilidade as informações necessárias para elaboração de casos de testes que, de outra forma, seriam de difícil percepção. Especificamente, as redes hierárquicas facilitam a compreensão do sistema como um todo por permitir a modelagem em subsistemas deixando de forma clara a interligação entre as partes e o funcionamento do todo.

O uso de ontologia promove a reusabilidade dos casos de testes e da massa de dados através de uma representação do domínio baseado em conhecimento. A utilização dessa ferramenta para formalizar casos de testes facilita a adaptação de testes antigos à novas modificações e até mesmo na eventual necessidade de uma simples execução de um ciclo de regressão para verificar o surgimento de novos erros após atualizações geradas pela evolução dos requisitos do sistema.

1.1 OBJETIVO

Este trabalho tem como objetivo para minimizar o esforço gasto na etapa de testes, ampliar a base de conhecimento e melhorar a reusabilidade dos artefatos de testes dentro do projeto. Para tanto são propostos dois métodos de geração de casos de teste complementares: o primeiro consiste em uma maneira de extrair o que será necessário testar baseando-se em redes de Petri hierárquicas, e o segundo, consiste de um método para verificar e validar resultados após uma bateria de testes, fazendo inferências sobre um modelo do sistema em OWL. Primeiro método fornecerá fluxos padronizados e seu modelo também servirá como método de pré-validação para a fase de execução, enquanto que o validador reduzirá o tempo gasto para verificar se os resultados da execução dos casos de testes originados do gerador e selecionados para a bateria de testes geraram erros.

1.2 ESTRUTURA DO TRABALHO

Esse trabalho está dividido em cinco capítulos: Introdução, Revisão Bibliográfica, Modelo Proposto, Estudo de Caso e Conclusão.

O capítulo de Introdução apresenta uma breve explicação do cenário atual, da motivação e dos objetivos a serem atingidos.

O capítulo de Revisão Bibliográfica exhibe os principais conceitos envolvidos no trabalho, relacionando estes conceitos e potenciais resultados com os principais artigos já realizados sobre o tema.

No capítulo Modelo Proposto apresenta-se detalhadamente os dois métodos propostos, e o relacionamento entre eles. Neste capítulo é apresentado passo a passo a aplicação desses métodos, bem como as considerações práticas assumidas.

O capítulo Estudo de Caso ilustra, com um exemplo prático, a aplicação dos métodos e os resultados esperados.

Finalmente, um capítulo de Conclusão é apresentado, contendo uma avaliação geral dos métodos, bem como melhorias e futuros estudos.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo apresentaremos os principais conceitos, estado da arte, e arcabouço teórico do trabalho proposto.

2.1 REDES DE PETRI

A Rede de Petri é uma representação matemática utilizada para modelar sistemas distribuídos. Trata-se de um grafo bipartido, onde os elementos constituintes são divididos em componentes ativos (os eventos, representados graficamente por um retângulo) e componentes passivos (os lugares, representados graficamente por um círculo) (MIYAGI, 2007, p. 74).

Arcos direcionados interligam os lugares a transições e transições a lugares, indicando as pré-condições e pós-condições de um evento. Assim, um arco nunca pode interligar duas transições ou dois lugares. Marcas são utilizadas nos lugares para indicar ativação do elemento, ou seja, que a pré-condição para uma transição associada a estes lugares está habilitada – na condição de disparo não-estrita.

Tanto os lugares como as transições independentes, habilitadas em um mesmo momento, formam estados e transições distribuídos, o que torna a Rede de Petri (RdP) uma ferramenta eficiente para modelagem de processos concorrentes (KATO e PORTO, 1999, p. 563).

A representação de sistemas complexos e/ou de grande porte em Redes de Petri convencionais gera modelos de difícil compreensão e análise, o que torna inviável o processo de extrair casos de testes. Assim, optou-se pela escolha de redes de alto nível para elaborar os modelos, minimizando a probabilidade de ocorrência de erros, tanto na construção do modelo quanto na sua validação, além de diminuir o tempo necessário para elaboração desses modelos. Dessa forma, o problema da explosão do espaço de estados com o aumento da complexidade do sistema é amenizado permitindo melhor interpretação do modelo.

Segundo a norma ISO/IEC 15.909 (ISO/IEC 15909, FINAL DRAFT V. 4.7.1, 2000) uma rede de alto nível pode ser definida como uma n-upla $P = (S, T, D, Pre, Post, M_0)$, onde S é o conjunto finito de lugares, T o conjunto finito de transições, D é um conjunto finito de assinaturas ou tipos, $Pre, Post: TRANS \rightarrow \mu PLACE$ representam as pré e pós-condições dos lugares (onde $TRANS = \{(t, m) | t \in T, m \in Type(t)\}$ e $PLACE = \{(s, g) | s \in S, g \in Type(s)\}$) e $M_0 \in \mu PLACE$ é a marcação inicial.

Uma transição está habilitada a disparar quando todos os lugares que representam sua pré-condição possuem um numero de marcas maior ou igual ao peso do arco que liga este lugar à transição em questão (condição de disparo não-estrita). O tipo do lugar determina as marcas que aquele lugar pode aceitar e o arco de conexão possui uma expressão que reflete a condição para que marcas possam fluir por este arco.

Neste trabalho optou-se pelo uso da rede GHENeSys (“*General Hierarchical Enhanced Petri Net*”), pois é uma rede unificada estendida, orientada a objeto e que satisfaz aos requisitos da norma ISO/IEC 15.909 (incluindo as definições de redes de alto nível, o formato de transferência de modelos para outros aplicativos, PNML, e normalização das extensões). Segundo Miralles (MIRALLES e SILVA, 2010), esse tipo de rede é recursivo, integrado e orientado a objetos, onde cada elemento da rede representa um objeto, além de possuir uma classe própria para representar as marcas, aumentando a flexibilidade na construção de modelos.

Por definição (FOYO, 2001) a rede GHENeSys pode ser definida como uma n-upla $P = (S, T, F, W, K, M_0)$, onde S e T são os conjuntos de lugares e transições, $F \subseteq (S \times T \rightarrow N^+) \cup (T \times S \rightarrow N^+)$ representa a relação de fluxo dos arcos, $W: F \rightarrow N^+$ é a função peso atribuída aos arcos indicando a quantidade de marcas associadas, $K: L \rightarrow N^+$ é a função capacidade de marcas associada a cada lugar e M_0 a marcação inicial da rede, onde a marcação em um lugar pode ser definido como $M: S \rightarrow N^+$ tal que $M(s) \leq K(s), \forall s \in S$.

A habilitação de uma transição $t \in T$ em uma determinada marcação M é definida pelas expressões $\forall s \in {}^*t, s \in S. M(s) \geq w[s, t]$ e $\forall s \in t^*, s \in S. M(s) \leq K(s) - w[t, s]$ onde $w \in W$, *t é o conjunto dos lugares que representam as pré-condições de t e t^* os lugares que representam as pós-condições.

Entre as extensões presentes na rede GHENeSys pode-se destacar:

- **Pseudo-Lugares** (“*Pseudo-Places*”): Servem para modelar as transferências de informações entre as diferentes partes do sistema e entre o ambiente externo e o modelo, permitindo a modularização das partes do sistema e a representação de eventos observáveis porém não controláveis. De maneira prática, esses lugares habilitam ou inibem determinadas transições na rede;

- **Hierarquia** (“*Hierarchy*”): Permite a utilização de macro elementos que representam sub-redes dentro do modelo. Esses elementos podem ser classificados como transições (ativos) que começam e terminam com transições ou lugares (passivos) que começam e terminam com lugares. No modelo, a hierarquia é representada através de macro lugares e macro transições que quando expandidas apresentam uma sub rede de Petri;

- **Tempo**: Essas redes também conseguem representar sistemas em tempo real ao assumir um intervalo de disparo para cada transição, característica original das Redes de Petri Temporal (“*Time Petri Net*”, TPN), que pode ser incorporada sem dificuldades. Conforme descrito por Foyo (FOYO e SILVA, 2011) as Redes de Petri Temporais são definidas por uma tupla $P = (S, T, B, Fo, M_o, SIM)$, onde B são as funções de incidência para trás, Fo são as funções de incidência para frente com $F = B \cup Fo$ e $SIM = T \rightarrow Q^+ \times (Q^+ \cup \{\infty\})$ o mapeamento de disparo das transições. Cada transição possui dois números racionais um representando o menor tempo de disparo possível (T_{min}) e o segundo representando o maior tempo de disparo (T_{max}), em outras palavras, a partir do momento que uma transição é habilitada em τ , ela só poderá disparar após o tempo $\tau + T_{min}$ e é obrigada a disparar antes de $\tau + T_{max}$ desde que permaneça habilitada durante esse período;

- **Gates**: Segundo Silva (SILVA e DEL FOYO, 2012), os *gates* são arcos que transportam apenas informações mantendo a marcação no lugar original. Existem dois tipos: os habilitadores (que habilitam uma determinada transição quando o lugar conectado possui marcação) e os inibidores (que mantêm a transição desabilitada enquanto o lugar conectado possuir marcas). Na rede GHENeSys os únicos lugares que podem ser conectados via *gate* são os pseudo-lugares, ou seja, apenas lugares externos ao sistema.

2.2 REPRESENTAÇÃO DE CONHECIMENTO

As ferramentas de representação de conhecimento destinam-se a expressar o conhecimento de forma que não exista duplicidade na sua interpretação, permitindo a sua utilização tanto para repassar informações entre humanos como entre máquinas. Essa modelagem fornece praticidade para a manutenção do conhecimento sobre o projeto e facilita o reuso dos dados entre projetos semelhantes.

Nos próximos parágrafos são apresentadas as duas linguagens utilizadas para representar conhecimento associado à web.

2.2.1 RDF – Resource Description Framework

O RDF (*“Resource Description Framework”*) é um framework desenvolvido pela W3C (*“World Wide Web Consortium”*) para descrever conceitualmente e modelar informações existentes em recursos *web* utilizando notações sintáticas e formatos de serialização de dados (W3C, 2004). Os recursos são definidos como uma expressão assunto-predicado-objeto, onde o assunto é o próprio recurso, o predicado denota aspectos ou peculiaridades do recurso e expressam as relações entre o assunto e o objeto. Como um exemplo simples, na expressão “O carro tem cor preta”, o assunto é “carro”, o predicado é “tem” e o objeto é “cor preta”. (W3C, 2004)

O RDF trabalha com sete conceitos: modelo de dados gráfico, vocabulário baseado em URI (*“Uniform Resource Identifier”*, trata-se de um identificador compacto para recursos) (BERNERS-LEE, FIELDING, *et al.*, 1998), tipos de dados, literais, sintaxe de serialização XML (*“Extensible Markup Language”*, linguagem de marcação utilizada para descrever o comportamento de programas) (W3C, 2000), expressões de fatos simples e vinculação (W3C, 2004).

O modelo de dados gráfico é o conjunto assunto-predicado-objeto onde o predicado é o arco que liga o nó assunto ao outro nó objeto formando a expressão. O conceito de vocabulário baseado em URI implica que todo nó deve ser definido como uma referência URI. Quando o nó é definido por uma referência ou um literal,

esse dado representa o que o nó é, e quando ele é definido em branco ele é considerado um nó único que pode ser usado em outras expressões, mas não possui um nome próprio.

Os tipos de dados são definidos como inteiros, pontos flutuantes, datas, unidades, entre outros. Literais são usados para representar valores por meios léxicos, sua representação é mais intuitiva que referências em URI, porém só podem ser usados para representar objetos.

As expressões denotam a relação entre duas coisas que podem ser representadas por um modelo de dados gráfico com o predicado nomeando a relação. Quando todos os arranjos de coisas no mundo que faz uma expressão A ser verdadeira também faz uma outra expressão B ser verdadeira, diz-se que A está vinculado a B, em outras palavras, se A é demonstrado, B pode ser inferido.

Para descrever os recursos utiliza-se a linguagem XML (*“Extensible Markup Language”*) para descrever as expressões. Essa linguagem é utilizada para definir o formato do documento através de marcações permitindo a interpretação do conteúdo. Dentro do XML, cada assunto e cada objeto definidos no framework RDF são representados como um nó enquanto o predicado é representado como um arco interligando um assunto a um objeto.

2.2.2 OWL – Ontology Web Language

Segundo Wang (WANG, BAI, *et al.*, 2007, p. 2), a ontologia é definida como “uma descrição semântica formal, uma representação conceitual do sistema através de uma teoria lógica, ou o vocabulário usado por uma teoria lógica”, ou seja, uma representação formal do conhecimento sobre determinado assunto. Pode ser utilizada tanto para identificar a estrutura quanto a semântica de modelos, regras de negócio, ou do ambiente externo ao contexto analisado.

Existem linguagens específicas baseadas em ontologia desenvolvidas para descrever processos em programas, dentre as mais utilizadas encontra-se a OWL (*“Web Ontology Language”*) utilizada nesse trabalho (W3C, 2004). Essa linguagem permite processar o conteúdo das informações nela tratada, servindo como forma de apresentação e representando o conteúdo interpretável de máquina na rede. (W3C, 2004)

Um documento OWL é composto por um cabeçalho opcional, axiomas de classes (definem a classe com base na existência de outras classes: equivalência de classes, subordinação de classes e não atrelamento de classes), axiomas de propriedades e fatos sobre indivíduos (sobre sua identidade e sobre suas propriedades) (W3C, 2004). As classes são mecanismos para agrupamento de recursos, dessa forma, todo indivíduo é uma instância de uma classe.

Cada classe é definida por “descrições de classes” que descrevem o que elas são e quais condições um objeto deve possuir para ser uma instância desta classe. Existem seis tipos de descrição de classe: identificador (único e obrigatório, exemplo: Classe “Frutas”), enumeração de indivíduos (únicos indivíduos que podem pertencer a classe, exemplo: para classe frutas só pode possuir os indivíduos “Caqui”, “Maçã” e “Banana”), restrições de propriedades (condições para o indivíduo pertencer a classe, exemplo: para o indivíduo pertencer a classe “Carro” este deve possuir uma relação de 1 para 4 com a classe “Rodas”), interseção (a classe é formada pela interseção de duas ou mais descrições de classes, exemplo: para ser considerado da classe “Vegetal”, o indivíduo deve ser da classe “Ser_Vivo” e ser da classe “Produzir_Glicose”), união (a classe é formada pela união de duas ou mais descrições de classes, exemplo: classe “Telefone” é constituída de indivíduos da classe “Celular” e da Classe “Telefone_Fixo”) e complemento (a classe é formada pela negação de outra descrição de classe, exemplo: classe “Herbívoro” é complemento da classe “Carnívoro”).

A OWL também define dois tipos de propriedades: a propriedade de objeto (associa um indivíduo a outro indivíduo) e propriedade de dados (associa um valor de dado a um indivíduo). As propriedades também podem se associar com base na subordinação entre elas, bem como possuir domínio e intervalo de operação.

Existe uma extensão da OWL chamada SWRL (“*Semantic Web Rule Language*”) que segundo a W3C (W3C, 2004) é uma combinação do OWL com a Linguagem de Marcação de Regra (“*Rule Markup Language*”) que utilizam uma sequência de predicados que implicam em uma consequência, ou seja, se todos os predicados são verdadeiros a consequência também será verdadeira formando assim um axioma. Por exemplo: $carro(x), tem_cor(x,y), branco(y) \rightarrow ambulância(y)$, se “x” é um objeto do tipo “carro” ($carro(x) = Verdadeiro$), “y” é um objeto do tipo “branco” ($branco(y) = Verdadeiro$) e o objeto “x” tem uma relação do

tipo “tem_cor” com o objeto “y” ($tem_{cor}(x,y) = Verdadeiro$), dessa forma o objeto “x” é do tipo “ambulância” ($ambulância(x) = Verdadeiro$).

Essa extensão também permite a utilização de “*Built-ins*” que permitem uma flexibilidade maior para criar sentenças e interoperabilidade com outros formalismos “*Web*” (W3C, 2004). Através desses built-ins é possível efetuar operações aritméticas (somadas, divisões, módulos, senos...), comparações mais complexas entre objetos (maior que, menor que...), operações booleanas, operações com texto (concatenar, caixa alta...), operações com data e hora (duração, subtrair tempos...), operações com URI e operações com listas (verificar se membro pertence a uma lista, subtrair listas...).

2.3 TESTE EM SOFTWARE

O teste em sistemas de *software* é o processo de qualidade que visa garantir a correta adequação dos produtos, projetos e processos conforme previamente especificado, minimizando possíveis erros durante o desenvolvimento. Este processo é contínuo e está presente em todas as fases, acompanhando o andamento e aumentando a eficácia do processo de desenvolvimento.

A etapa de testes é indispensável para se garantir qualidade em um *software*, pois permite encontrar defeitos em fases anteriores à implementação minimizando os custos de desenvolvimento.

Por definição, o teste é a investigação do *software* a fim de fornecer informações sobre sua qualidade em relação ao contexto em que ele deve operar, incluindo o processo de utilizar o produto, portanto, o sistema é operado sob condições controladas verificando se o comportamento condiz com o esperado (ISTQB, 2011, p. 13).

Por se tratar de uma atividade minuciosa e de alto custo, torna-se inviável testar um sistema abrangendo todas as possibilidades de comportamento mesmo que esse sistema seja simples, o que leva a necessidade de assumir determinados níveis de cobertura nos testes. O processo de teste também é relativamente lento, pois além de escolher a forma de cobertura é preciso desenvolver cada caso e cada cenário de teste, levantar e preparar a massa necessária de dados e montar um plano de execução, para só então executar os testes levantando as evidências das

execuções para o desenvolvedor, que por sua vez investiga as causas dos defeitos, conserta e libera o sistema para a execução de um novo ciclo de testes.

2.3.1 Abrangência de Teste

Normalmente os testes são divididos em quatro níveis: Teste de Componente, Teste de Integração, Teste de Sistema e Teste de Aceitação (ISTQB, 2011, p. 21).

Os Testes de Componentes ou Testes Unitários verificam o funcionamento de cada componente do *software* - como classes e módulos - separadamente, podendo incluir tanto teste funcionais (funcionalidades) quanto testes não-funcionais (performance, carga, memória). Normalmente são testes caixa-branca (com acesso ao código) executados diretamente pelo próprio programador no ambiente de desenvolvimento.

Os Testes de integração validam as interfaces presentes sejam elas internas ao sistema, externas, relacionadas à manipulação de arquivos ou relacionadas à comunicação com *hardwares*. Pode existir mais de um nível no teste integrado, como exemplo, só testar as interfaces de classes, só as interfaces de componentes, ou só as interfaces de sistema.

Teste de Sistema avalia o comportamento do sistema como um todo comparando com as especificações (de requisitos) definidas no início do projeto. Por esse motivo em geral são utilizados primeiramente testes caixa-preta (sem acesso ao código) e posteriormente, para verificar a eficácia do teste, alguma técnica caixa-branca. Esses testes são feitos em um ambiente próprio e devem ser realizados por uma equipe especializada em testes.

Por último, os Testes de Aceitação são normalmente realizados pelos próprios usuários do sistema de forma a estabelecer a confiança no sistema analisando disponibilidade do sistema para entrar em produção. Esse nível de teste pode ser realizado mais de uma vez e em momentos diferentes, fornecendo “*feedbacks*” antecipados para os desenvolvedores durante o avanço do projeto. As formas mais comuns são: Teste de Aceitação pelo Usuário (usuário verifica se o sistema está apto a seu uso), Teste Operacional de Aceitação (realizado pelo administrador do sistema abordando principalmente a manutenção e segurança), Teste de Aceitação do Contrato (verificando critérios de aceitação definidos em contrato), Teste de

Aceitação de Regulamento (verificando a adesão dos sistemas a outras normas vigentes), Alfa Teste (realizado pelo cliente no ambiente da empresa que desenvolveu) e Beta Teste (ou Teste de Campo, realizado pelo cliente no ambiente de operação).

Cada variação de teste possui um objetivo de validação diferente, conhecidos como os alvos de teste. O Teste Funcional tem como objetivo avaliar as funcionalidades do sistema, ou seja, foca no que o sistema faz. Os Testes Não Funcionais levam em consideração características não-funcionais do sistema tais como performance, usabilidade, segurança, *stress*, confiabilidade, manutenibilidade, portabilidade, entre outras. O Teste Estrutural auxilia na medição da eficiência dos testes, averiguando a cobertura do código atingida. Nos Testes de Confirmação o objetivo é verificar se defeitos eventualmente encontrados foram corrigidos. Por último, os Testes de Regressão têm como alvo a busca por defeitos introduzidos após uma mudança no sistema, através da repetição dos testes já realizados.

2.3.2 Técnicas de Teste Funcional

Os testes podem possuir duas abordagens: caixa-preta, que possui técnicas baseadas nas especificações, e/ou na experiência do testador, e caixa-branca, que possui técnicas baseadas na estrutura do código (ISTQB, 2011, p. 37).

2.3.2.1 Técnicas Caixa-Preta (Especificação)

As técnicas baseadas em caixa-preta utilizam modelos para especificar um problema a ser resolvido dentro da aplicação, podendo este ser utilizado para extrair casos de testes de maneira sistemática. (ISTQB, 2011, p. 38)

A técnica Partição de Equivalência consiste em dividir os dados em grupos com comportamentos semelhantes, considerando tanto dados de entrada e saída, válidos e inválidos, valores internos, relacionados com o tempo e com parâmetros de interfaces.

A Análise do Valor Limite consiste em avaliar os valores limites (máximo e mínimo) de cada partição válida e inválida, que são valores mais propícios a falharem.

Utilizadas para requisitos que possuem condições lógicas, as Tabelas de Decisão traduzem as condições de entradas e ações declaradas de forma lógica (verdadeiro ou falso). Cada coluna representa uma regra de negócio que define uma combinação de condições que resulta em um grupo de ações associada a regra.

Os Testes de Transição de Estados verificam as mudanças dos estados da aplicação conforme as condições de entradas e eventos disparados, podendo ser utilizado para fluxos específicos de estados, cobrir todos os estados, transições válidas ou transições inválidas.

A técnica de Teste de Caso de Uso, como o próprio nome informa, se baseia no fluxo e nas ramificações dos casos de uso para derivar os casos de testes, sendo muito úteis para testes de aceitação.

2.3.2.2 Técnicas Caixa-Preta (Experiência)

Nas técnicas baseadas em experiência, o conhecimento e a experiência dos envolvidos (testadores, desenvolvedores, analistas de negócio, usuários...) é utilizada para derivar os casos de testes, direcionando os esforços para a detecção de defeitos mais prováveis e mais críticos. São utilizados principalmente para gerar testes de forma prática, difíceis de serem gerados pelas técnicas formais. Infelizmente a eficiência dessa técnica está diretamente ligada à experiência do testador (ISTQB, 2011, p. 41).

A Suposição é a técnica mais utilizada nesta abordagem e consiste em estimar onde os erros são encontrados, ou seja, os testes serão direcionados para os pontos onde se supõem que os erros existem (Um módulo completamente novo, um módulo desenvolvido por um fornecedor novo, uma parte do sistema que costuma falhar com frequência...). Uma maneira mais formal, chamada de Ataque a Falha, consiste em listar os possíveis erros e executar testes de forma a atacá-los. São realizados por um tempo pré-definido, sendo utilizados onde a documentação é escassa ou o prazo de execução é curto.

2.3.2.3 Técnicas Caixa-Branca

Essas técnicas são baseadas na estrutura interna dos dados em que o sistema foi desenvolvido e são utilizadas tanto para a elaboração dos casos de testes, como para medir a cobertura dos testes. Nestes testes, a estrutura varia conforme o nível do teste (ISTQB, 2011, p. 40):

- para o nível de Componente a estrutura é código,
- para o nível de Integração a estrutura é uma árvore de chamadas e
- para o nível de Sistema uma estrutura de menu, processos ou páginas Web.

Habitualmente utilizam-se ferramentas específicas para auxiliar na execução dessas técnicas.

Teste e Cobertura de Sentença é uma técnica que confere a porcentagem de sentenças de código que foram executadas por um conjunto de casos de testes.

A técnica Teste e Cobertura de Decisão avalia a porcentagem dos resultados de decisão, como por exemplo, a quantidade de condições verdadeiras e falsas nos comandos “*IF*”. Em outras palavras, testa o controle dos fluxos possíveis internos da aplicação.

2.4 TRABALHOS CORRELADOS

2.4.1 Análises sobre base bibliográfica

As Tabelas 1 e 2 apresentam um resumo dos principais artigos existentes na área de teste de *software* que utilizam técnicas baseadas em Redes de Petri e/ou representações de conhecimento. Uma análise mais aprofundada dos artigos é apresentada nos próximos tópicos.

Referência	Tempo Suporte Requisitos	Multi-agentes	Gerador de Casos de Testes	Gerador de Massa de Dados	Redução de Casos de Testes	Ferramenta auxiliar	Técnicas de Extração
S. Thiel	X					Software EXAM; UML	Eliminação de casos de testes e passos redundantes
J. Desel			X			Tabela de casos de teste	Representação gráfica de Causa/efeito
I. Ho	X	X	X			Método do tempo variante	Matriz de incidência (representação normal)
N. Adjir	X		X			Software TINA (modelagem da rede de Petri); SSC (zonas de estados)	Otimização temporal de execução de casos de testes
S. Ramaswamy	X	X	X			Medida de complexidade ciclomática McCabe	Estratégia de escolha gananciosa; Estratégia de escolha de design; Estratégia de escolha de teste
Y. Kissoum		X	X			AUML; Ambiente JACK	Grafo de Atingibilidade
L. Cai			X			Software CPN-Tools	Espaço de Estados
E. Mirzaeian			X			Marcação Objeto; CPN-Tools	Modelo em Máquina de Estado UML
T. Sun		X	X			LBSPT	Método de seleção sequencial de teste
J. Liu			X				Baseada na relação de conformidade entrada-saída
H. Gong			X				Função de comportamento
L. Lun			X			π calculus / π ADL	π Behavior Graph (Gráfico de comportamento π)
Z. Liu	X	X	X				Análise em cima de fluxo de trabalho (casos de uso) de <i>Softwares</i> colaborativos
H. Reza	X		X				Método W
W. Dong		X			X	BPEL	Redução de Casos de Testes em cima da composição do sistema e do pareamento das entradas
Y. Wang	X		X	X		OWL-S/XML	Ontologia
C. Fan			X			SAML/XML	Ontologia (domínio-específico)
H. Zhu	X	X	X			STOWL/SOAP	Ontologia
H. Huang			X			OWL-S/BLAST	Ontologia
R. García-Castro				X		OWL DL	N/A
X. Bai		X	X	X		OWL-S/TOM	Ontologia

Tabela 1 - Tabela de referência (Suporte, Ferramentas e Técnica)

Referência	Cobertura	Tipo de Rede	Tipo de Técnica	Alvo de Teste
S. Thiel	Não especificada	P/T	Ambos	Teste Funcional de <i>Firmware</i>
J. Desel	Não especificada	Pr/T e C/E	Caixa Preta	Funcional
I. Ho	Por grupo de segmentos independentes	TPN	Caixa Branca	Teste de Sistemas em Tempo Real
N. Adjir	Por transições; Por lugares; Por marcas	PrTPN	Não especificado	Teste de Sistemas em Tempo Real
S. Ramaswamy	Conjunto de Cobertura de Transição Mínima	H-EPN	Não especificado	Funcional
Y. Kissoum	Não especificada	RCPN	Ambos	Teste de Sistemas Multi-agente
L. Cai	Por Estado; Por Transição; Por Par de Transição	CPN	Caixa Preta	Funcional
E. Mirzaeian	Cobertura de todas as Instâncias	CPN	Não especificado	<i>Softwares</i> orientados a objeto
T. Sun	Por Combinação de comportamento	CPN	Não especificado	Sistema de <i>Softwares</i> paralelos
J. Liu	Não especificada	CT-CPN	Caixa Preta	Teste de conformidade
H. Gong	Não especificada	EDPN	Não especificado	<i>Softwares</i> orientados a objeto
L. Lun	Componente Individual; Função conector individual; Componente para conector; Conector para componente; Componente direto para componente; Componente indireto; Todas conexões	Rede de alto nível não especificada	Ambos	Não especificado
Z. Liu	Coberturas condicionais: por caminho; por braço; composta	Rede de alto nível não especificada	Caixa Branca	Teste Funcional de <i>Softwares</i> colaborativos
H. Reza	Por Transições	CMPN	Caixa Preta	Não especificado
W. Dong	Não especificada	Rede de alto nível não especificada	Não especificado	Testes de <i>Web Services</i>
Y. Wang	Por Transições	P/T	Não especificado	Teste Funcional em <i>Web Services</i>
C. Fan	Por parâmetro; por caso de uso; por condição irregular; por cenário	N/A	Caixa Preta	Testes de Validação
H. Zhu	Não especificada	N/A	Caixa Preta	Teste Funcional de <i>Softwares</i> colaborativos
H. Huang	Não especificada	N/A	Caixa Preta	Teste Funcional em <i>Web Services</i> compostos
R. García-Castro	N/A	N/A	N/A	Conformidade Ferramentas OWL
X. Bai	Não especificada	N/A	Ambos	Testes de partição em <i>Web Services</i>

Tabela 2 - Tabela de referência (Cobertura, Tipo de Rede, Metodologia e Alvo)

Na Tabela 1 a coluna “Tempo” indica se o artigo faz uma abordagem envolvendo o tempo (em intervalos ou tempo real), a coluna “Suporte Requisitos” diz se há informações sobre a extração de requisitos utilizando a metodologia proposta, da mesma forma que a coluna “Suporte Multi-agentes” mostra se existe um tratamento para casos envolvendo vários agentes simultâneos. A coluna “Gerador de Casos de Teste” informa que o “*paper*” propõe um gerador, e a coluna “Gerador de massa de dados” indica se é proposto um gerador de dados para testes, enquanto a coluna “Redução de Casos de Testes” mostra se há uma proposta de redução de casos de testes. “Ferramenta Auxiliar” cita as principais ferramentas utilizadas na referência em questão e, por último, a coluna “Técnicas de Extração” informa qual foi a técnica e métodos principais utilizados no trabalho para criar o extrator de casos de testes.

As coberturas utilizadas pelos autores são mostradas na coluna “Cobertura” na Tabela 2. O tipo de rede utilizado é indicado na coluna homônima, enquanto que o tipo de técnica de teste utilizada (caixa-preta ou caixa-branca) é apresentada na coluna “Tipo de Técnica”. A última coluna “Alvo de Teste” indica qual é o foco principal do teste, o tipo de sistema ou programa ao qual o estudo proposto é aplicável. Nota-se que o uso de redes de Petri para aplicação em teste de *software* é mais comum que o uso de ontologias.

Analisando as tabelas é possível observar que só o artigo do Wang utiliza os conceitos de ontologia combinado com Rede de Petri, porém aborda outros aspectos. Nenhum dos trabalhos que envolvem ontologia as utilizam como validador de resultados após uma execução, além de existir dois artigos que utilizam o conceito de Rede de Petri Modular (Ramaswamy H-EDN (RAMASWAMY, 2000) e Reza CMPN (REZA e KERLIN, 2011)).

2.4.2 Teste Funcional com Redes de Petri

Os próximos parágrafos apresentam os pontos principais e propostas que aparecem nos trabalhos recentes envolvendo redes de Petri e geradores de casos de testes.

2.4.2.1 Rede Lugar/Transição

Thiel (THIEL e DERICHSWEILER, 2011, p. 143), propõe o uso de uma rede clássica Lugar/Transição para eliminar casos de testes dependentes e passos redundantes em sistemas eletrônicos embarcados em automóveis. Nessa proposta, os casos de testes são modelados na ferramenta de gerenciamento de teste (no caso EXAM) que os transforma em redes de Petri para avaliação. A avaliação consiste em verificar se existem módulos isomorfos ou equivalentes entre os diferentes módulos escolhendo em seguida o caso de teste que contém os demais. Outros processos de redução de redes de Petri clássicas (MURATA, 1989) de modo a obter o processo mais simplificado que corresponde também a uma rotina de testes otimizada. A simulação da rede (jogo de marcas) corresponde à execução do caso de teste.

Portanto as Redes de Petri são usadas, nesta proposta, para otimizar as sub-redes que representam os casos de teste e para dar suporte à sua execução diferente do trabalho aqui proposto que as utiliza para criar as ideias de testes, ou seja, é um passo depois. O processo de geração das ideias e a modelagem dos casos de teste (criação do passo a passo) é ainda manual e feito pelo próprio usuário na ferramenta EXAM.

2.4.2.2 Redes Predicado/Transição

A rede Predicado/Transição ("*Predicate/Transition net*", Pr/T *net*) é uma rede de alto nível baseada em predicados ao invés de estados. Esta rede pode ser definida como: $P = (S, T, F, \lambda, R, \phi, M_0)$, onde S e T representam os elementos ativos e passivos (predicados e transições), $F = (S \times T) \cup (T \times S)$ é a relação de fluxo, λ é uma estrutura que consiste em alguns conjuntos de indivíduos (constantes) combinados com algumas operações e relações, R é a função rótulo associada aos arcos e ϕ é um conjunto de inscrições para cada transição. Cada inscrição é uma fórmula lógica construída com base em indivíduos, variáveis, operações e relações da estrutura λ .

Entre os trabalhos existentes associados a teste, Desel (DESEL, OBERWEIS, *et al.*, 1997, p. 327) propõem um gerador de casos de testes com base nas causas e efeitos de um processo no *software*, utilizando uma técnica de representação gráfica em caixa-preta.

Os casos de teste são identificados construindo um grafo causa-efeito convencional e depois transformando este grafo em uma rede clássica, Condição/Evento (C/E). Uma primeira verificação pode ser feita invertendo os arcos e simulando a rede C/E de modo a garantir que cada efeito está associado com causas determinadas pela marcação “inicial”. Com base neste processo é gerada uma tabela causa-efeito. Em seguida, através de uma análise semântica da documentação, as causas e efeitos são ligados entre si utilizando as relações ‘e’ (uma causa e outra causa levam ao efeito), ‘ou exclusivo’ (ou uma causa ou a outra causa leva ao efeito), ‘simples’ (uma causa leva diretamente ao efeito) e ‘alternativo’ (uma causa leva a mais de um efeito).

Esse processo consiste em: 0) Obtenção da rede C/E; 1) Converter a direção de todos os arcos na rede e simular a rede para obter os efeitos; 2) Se um efeito ainda não foi investigado, marcar ele; 3) Gerar todas as redes de processos (rede simplificada contendo apenas o efeito, suas causas e os arcos que os interligam) para o efeito marcado de causa/efeito; 4) Para cada rede de processo identificar a relação entre um efeito e suas causas.

A etapa seguinte é construir a tabela de casos de teste, onde cada coluna representa uma relação causa/efeito identificada anteriormente, dessa forma, cada coluna representa um caso de teste. Por último é realizada uma inspeção de objetivo de teste nos casos de testes gerados, para tanto, são verificados se os efeitos são gerados quando as causas necessárias são praticadas. Para isso, para cada causa de uma coluna na tabela de casos de teste transformam-se em marcas iniciais em uma Pr/T, então o objetivo de teste é simulado e o resultado comparado com o esperado (efeito atingido).

O gerador proposto por Desel utiliza uma rede mais simples que a rede hierárquica utilizada na dissertação, como a rede C/E não possui módulos é um pouco complexo representar entidades e módulos nela tornando visualmente mais poluída. A rede também não permite representação direta de acionamentos temporais o que obriga a criar adaptações para representar determinados eventos como, por exemplo, rotinas periódicas. O foco dos casos de testes obtidos são os

resultados finais (efeitos), ou seja, o método é voltado para os resultados sem a preocupação com os possíveis caminhos alternativos que o sistema pode assumir concentrando o esforço em poucos porém importantes casos de testes.

2.4.2.3 Rede de Petri Temporizada

A rede de Petri Temporizada (*“Time Petri Net”*, TPN), são redes simples definidas apenas pela n-upla $P = (S, T, F, T_m, M_0)$ onde T_m consiste em um conjunto de intervalos de disparo que cada transição possui, sendo medido a partir do momento em que a transição é habilitada e onde o disparo pode ocorrer apenas dentro deste intervalo.

Ho (HO e LIN, 1999, p. 296) aborda também um gerador de casos de testes que engloba *softwares* de tempo real. Nessa abordagem ele utiliza uma notação própria baseada na estruturação do modelo do *“software”* construído em uma rede Petri Temporizada (RAMCHANDANI, 1973). Após o primeiro passo (a modelagem do programa em rede de Petri), uma matriz quadrada gráfica (Matriz de Incidência) é construída para representar as relações entre os lugares e as transições. Nessa matriz cada coluna e linha representam um lugar identificado do modelo e os valores de entrada correspondem as transições que interligam os lugares representados na linha e coluna. Em seguida, converte-se a matriz em uma notação representando a rede com segmentos básicos (subconjunto do gráfico que possui apenas uma entrada e uma saída).

Na quarta etapa, há uma junção dos segmentos básicos através dos operadores lógicos da notação (e; ou; ou exclusivo) seguindo algumas regras. No próximo passo, Ho decompõem a rede em segmentos independentes, ou seja, divide em conjuntos de segmentos básicos que nunca serão executados simultaneamente. Nesse ponto reduz-se o domínio temporal, diminuindo assim a complexidade. Por último expande-se o domínio temporal de forma a obter uma visão mais clara do que esta ocorrendo.

Neste trabalho, Ho foca seu gerador em identificar partes independentes do sistema de forma a obter casos de testes que conseguiram avaliar o comportamento em tempo real, tornando a análise mais complexa ao converter a relação de transições temporais em equações lógicas. Essa análise permite transformar um

sistema temporal em um discreto garantindo a cobertura dos testes, porém o fato de não utilizar redes hierárquicas acaba dificultando a modelagem de sistemas complexos. O método de extração proposto no trabalho não possui o cuidado de transformar o sistema temporal em um sistema discreto se preocupando apenas com os fluxos que o sistema deve seguir e ao utilizar redes hierárquicas simplifica a modelagem do sistema.

2.4.2.4 Redes de Petri Temporizada e Priorizada (PrTPN)

Este tipo de rede, foi proposto por Adjir (ADJIR, SAQUI-SANNES e RAHMOUNI, 2009, p. 111) contendo quatro geradores de casos de teste otimizados no tempo e um gerador de suítes de testes para sistemas de tempo real. Nesses geradores, é considerado que o sistema é modelado em duas PrTPN, uma que modela o comportamento do sistema e a outra que modela o comportamento do ambiente. Esse tipo de rede é semelhante a Rede de Petri Temporizada porém suas transições possuem uma ordem de prioridade, sendo que a transição habilitada com prioridade mais baixa não será disparada se, ao mesmo tempo existir uma transição habilitada com prioridade mais alta. São definidas como: $P = (S, T, Pre, Post, >, M_0, In_s, A_r, L)$, onde $Pre: T \rightarrow S \rightarrow \mathbb{N}^+$ são as funções de pré-condição (utilizada para facilitar a identificação de estados pré-condicionais de uma dada transição), $Post: T \rightarrow S \rightarrow \mathbb{N}^+$ são as funções de pós-condição, $>$ é a relação de prioridade, $In_s: T \rightarrow \mathbb{N}^+$ é a função de intervalo estático, A_r é conjunto finito de ações e $L: T \rightarrow A_r$ é a função que rotula cada transição com um operador.

Para o gerador mais simples de Adjir, usa-se a planta de rastreamento presente no módulo “*self*” do software comercial TINA, onde, transformando o objetivo do teste em um estado ou evento atingível no modelo, o software é capaz de gerar os casos de testes já otimizados. Os demais geradores são baseados em critérios de coberturas: por transições - satisfeito quando todas as transições que são disparadas quando a sequência de teste é executada; por lugares - satisfeito quando todos os lugares são marcados quando a sequência de teste é executada; e por marcas - satisfeito quando são geradas todas as marcas no grafo SSC, modelo de abstração de classes de estados, quando a sequência de teste é executada.

O gerador de suítes consiste em armazenar os resultados gerados por qualquer combinação dos geradores de testes anteriores, reiniciando o sistema cada vez que um deles gera uma sequência de teste. Esse gerador também é executado dentro do TINA ao se colocar marcas reiniciáveis.

O método de Adjir mantém a dificuldade para modelagem de sistemas grande ao não utilizar uma rede mais compacta para isso, além de separar fisicamente o modelo do ambiente e o modelo do sistema podendo gerar algumas dúvidas em sua interpretação. Esse problema ocorre em menor escala ao utilizar a rede de Petri hierárquica, onde todos os pontos pertinentes ao sistema são modelados e interligados entre os níveis da rede. Os geradores já fornecem os casos de testes otimizados diferente do método proposto nessa dissertação, o que simplifica a quantidade de casos de testes economizando recursos.

2.4.2.5 Redes de Petri Hierárquicas Tempo-Estendidas

A rede de Petri hierárquicas tempo-estendidas (*"Hierarchical Time-Extended Petri Net"*, H-EPN) caracterizam por mostrar tanto os processos estáticos (controle) quanto dinâmicos (fluxo) do sistema. Nessas redes os lugares são divididos em subtipos: lugares de *status* (semelhante ao local das redes de Petri anteriores), lugares de ação (denotam uma ação que não muda o estado do sistema), lugares de decisão (que precedem um conflito na rede), lugares de sub-redes (representa a macro de uma sub-rede) e locais *"source-sink"* (representa a origem e o fim de uma marca).

O artigo escrito por Ramaswamy (RAMASWAMY, 2000, p. 3087) utiliza o algoritmo Conjunto de Cobertura de Transição Mínima (*"Minimal Transition Cover Set"*, MTCS) definido sobre um modelo em rede de Petri para minimizar a quantidade de casos de testes ou requisitos do software em questão. Esse algoritmo fornece o número mínimo de transições necessárias que cobrem todos os subfluxos distintos do modelo. A primeira etapa de processo do algoritmo chama-se Seleção, é responsável por selecionar os conjuntos escolhidos que possuem os grupos de transições que são necessárias para varrer todas as partes independentes do sistema, a segunda etapa é chamada Poda, responsável por determinar o

subconjunto de transições, entre os conjuntos escolhidos na primeira etapa, que minimamente garantem a cobertura.

O primeiro passo do algoritmo de Seleção é montar uma tabela onde as linhas correspondem aos T-invariantes (diferentes ciclos fechados dentro do modelo) e as colunas correspondem às transições distintas, marcando com “1” o local onde a transição aparecer no respectivo T-invariante. A segunda ação identifica as colunas que possuem as mesmas entradas, essas transições devem ser marcadas e eliminadas com exceção de uma. Em seguida, para cada T-invariante, são selecionadas as colunas que possuem o maior numero de entradas e que possuam o respectivo T-invariante, incluindo todas essas transições no conjunto escolhido (saída do algoritmo). Para cada transição identificada no processo anterior, seleciona-se todas as colunas com o mínimo de entradas, se o T-invariante não for coberto pelas transições do conjunto, incluir essa nova transição no conjunto. Em seguida, apagam-se as colunas encontradas no passo anterior e atualiza-se a tabela resultante. Por fim, substitui-se cada transição no conjunto escolhido por transições equivalentes identificadas na segunda ação.

Considerando que o conjunto escolhido na Seleção seja composto por subconjuntos que correspondem a todas as combinações possíveis dos elementos, o primeiro passo da Poda consiste em eliminar todos esses subconjuntos que possuem transições temporais. Caso existam mais que um subconjunto, deve-se verificar se todos os subconjuntos possuem ao menos uma transição acionada apenas por um evento externo ao sistema, caso exista algum que não satisfaça elimina-se todos os subconjuntos com ao menos uma transição acionada apenas por um evento externo. Por ultimo, calcula-se a cobertura invariante total (“*Total Invariant Cover*”, TIC) para cada subconjunto. O TIC é obtido somando a cobertura comum (numero comum de entradas para o correspondente T-invariante) para cada par de transições dentro do subconjunto. O subconjunto de saída será o subconjunto que possui o menor TIC.

Ramaswamy ainda propõem três estratégias de construção do MTCS: estratégia de escolha gananciosa, estratégia de escolha de design e estratégia de escolha de teste. Na estratégia de escolha gananciosa a primeira transição que cobre maximamente os T-invariantes é escolhida para o MTCS. Apesar da simplicidade, essa estratégia depende da ordem espacial das transições na Rede de Petri, que por sua vez precisa ser montada cuidadosamente Este fato faz com que

esta não seja a estratégia mais indicada para o design ou teste de determinados agentes.

Nas estratégias de escolha de design e de teste as transições, os MTCS são divididos em dois subconjuntos, o primeiro contendo as transições que não são resultado de nenhum processo de decisão e o segundo contendo as demais (que resultam de processos de decisão). Na estratégia de design é dada a preferência aos lugares de decisão que cobrem o máximo número de T-invariantes, ou seja, às transições que alimentam esses locais com marcas. A estratégia de teste é semelhante à de design, no entanto, são as transições que cobrem o mínimo número de T-invariantes que são escolhidas para fazerem parte do MTCS.

O Método de Ramaswamy se assemelha ao método adotado no trabalho, no entanto, enquanto Ramaswamy se preocupa em reduzir a quantidade total de casos de testes já existentes de forma a manter uma cobertura utilizando os T-invariantes, o método proposto nesse trabalho foca em obter os casos de testes aplicando algumas regras no modelo do sistema, podendo os dois métodos serem utilizados complementarmente.

2.4.2.6 Redes de Petri Coloridas

No artigo de KISSOUM e SAHNOUN, 2008, p. 786), é proposto um *framework* de teste para sistemas multi-agentes que utiliza uma rede CPN (“*Colored Petri Nets*”) extraída de diagramas de sequência em AUML (Agent UML)¹. Essa aplicação é baseada no ambiente de desenvolvimento de sistemas de agentes inteligentes JACK, e foca em testes de interações entre grupos de agentes dentro do sistema. A técnica consiste em montar o grafo de atingibilidade da rede extraída da modelagem das interações, criando os casos de testes. Em seguida, aplica-se um critério de cobertura e obter os testes necessários, que são executados em uma determinada ordem, utilizando controladores e simuladores de agentes.

¹ Agent UML apareceu no início deste século fruto de uma cooperação entre a FIPA (“*Fundation of Intelligent Physical Objects*”) e a OMG (“*Object Management Group*”), com o objetivo de criar uma extensão da UML que pudesse lidar com a *interact* de vários objetos físicos independentes.

Kissoum foca na criação de um *framework* inteiro de testes ao contrário desse trabalho que cria apenas um método de geração de ideias de testes, especializando-se em testes multi-agente (vários atores atuando simultaneamente) enquanto que trabalho é focando no fluxo do processo e avaliado primeiramente agente a agente. Kissoum também opta pela rede colorida que é mais complexa, porém por outro lado mais versátil que a rede hierárquica simples utilizada.

Cai (CAI, ZHANG e LIU, 2010, p. 1) utilizou uma rede de Petri colorida para desenvolver um gerador de casos de testes para esse tipo de sistema distribuído. O método de teste baseia-se na construção de um modelo do *software* através da análise de sua especificação em uma ferramenta usada principalmente para calcular o espaço de estados (todos os estados possíveis de um modelo) automaticamente, no caso a CPN-Tools.

A partir do espaço de estados é possível obter os casos de teste de acordo com a estratégia de cobertura escolhida. O artigo propõe a aplicação de um dos três critérios de cobertura: cobertura por estado (quando todos os estados estão incluídos no caminho de teste), cobertura por transições (quando todas as transições estão incluídas no caminho de teste) e cobertura por par de transições (quando todo par de transições ligadas por um estado em comum estão incluídas no caminho de teste).

O Fato de utilizar redes coloridas, Cai aumenta o grau de dificuldade da modelagem e da análise do modelo dinâmico do sistema exigindo um conhecimento maior do usuário para aplicá-la se comparado ao gerador deste trabalho, em compensação consegue criar estruturas menos poluídas e, em conjunto com a análise do espaço de estados, consegue avaliar de forma mais completa as situações que o sistema pode atingir.

Mirzaeian (MIRZAEIAN, MOJAVERI, *et al.*, 2010, p. 251) aborda o uso das Redes de Petri Coloridas para modelar o sistema gerador de casos de testes em *softwares* orientados a objeto, focando na modelagem do sistema e não na geração dos casos de testes. A técnica consiste na modelagem do comportamento de cada objeto do sistema através de Máquinas de Estado UML ("*UML Statechart*"), demonstrando cada estado abstrato possível das instâncias de uma classe. Uma análise dos objetos é realizada classificando as instâncias de cada objeto em duas categorias: instâncias de dado sensível (uma mudança em seu valor pode ocasionar uma mudança no estado do objeto) e instâncias com método sensível (onde

existem métodos que podem alterar instâncias de dado sensitivo direta ou indiretamente).

Mirzaeian também introduz o conceito de Marcação do Objeto (“*Object Token*”), uma marcação especial que diferencia os objetos que possuem tipos diferentes. Utilizando esse conceito e os modelos anteriores, mapeia-se a Rede de Petri Colorida, onde cada lugar é um estado de um determinado objeto e cada transição um método sensitivo que pode alterar o estado daquele objeto, onde se utiliza o CPN-Tools para calcular o espaço de estados para futura geração de casos de teste.

Mirzaeian também utiliza o espaço de estados, porém o aborda para analisar o comportamento dos objetos dentro do sistema chegando a um nível mais detalhado do comportamento do sistema, consistindo em uma etapa anterior a extração dos casos de testes. Dessa forma os resultados fornecidos pelo o método de Mizarein podem ser utilizados como dados auxiliares para a modelagem dinâmica utilizando o método descrito nessa dissertação. Deve-se ter atenção ao utilizar esse método, uma vez que este é desenvolvido apenas para sistemas orientados a objetos, enquanto que o método do trabalho pode ser utilizado para qualquer *software*.

Sun (SUN, YE e LIU, 2011, p. 164) cria um novo método de seleção sequencial de teste para sistemas de *softwares* paralelos utilizando Redes Coloridas. Nesse método, a especificação do programa é modelada por uma CPN, ao mesmo tempo em que o modelo do objetivo de teste é feito e representado pelas Sequências de Comportamento Linear Preparado para Testar (“*Linear Behavior Sequences Preparing for Testing*”, LBSPT) em uma segunda Rede Colorida que deve estar contida na CPN que especifica o programa. As LBSPT incluem funções e comportamentos relacionados ao objetivo de teste, sendo compostas de sequências de teste que representam um caso de teste.

O método consiste na modelagem do sistema em CPN, definindo em seguida projeções de operações entre os espaços de estados e criando assim sub-grafos dentro do modelo representando em LBSPT. Para tanto, define-se o estado inicial e o estado final de cada LBSPT e verifica-se a existência do estado máximo (estado sem precursores no conjunto) e do estado mínimo (estado sem sucessores no conjunto). Se existirem ambos os estados (máximo e mínimo), calcula-se todos os sub-caminhos entre os estados desse LBSPT. Por último, escolhe-se um caminho

completo na CPN que contenha o conjunto de sub-caminhos encontrados na etapa anterior obtendo assim o caso de teste.

Sun também usa o espaço de estados utilizando-o para obter quais sequências de validações serão utilizadas dentro dos casos de testes a serem obtidos. Nessa situação os casos de testes desse processo são mais amplos sendo indicados para testes de sistema onde o fluxo ou pedaço considerável dele é testado como um todo (testes fim-a-fim) diferente do método da dissertação que é mais indicado para testes de integração.

Liu (LIU, YE e LE, 2011, p. 967) elabora uma nova rede de Petri Colorida específica para testes de conformidades as Redes de Petri Coloridas Orientada a Testes de Conformidade, onde a especificação do sistema é modelada e a implementação do sistema em suas respectivas CT-CPN (*“Conformance Test-CPN”*). Aplicando a relação de conformidade entrada-saída (*“input-output conformance”*, ioco), onde a execução do teste ocorre por meio de ações de entrada iniciadas e controladas pelo testador e ações de saídas controladas pela própria implementação do programa, Liu compara a resposta do sistema implementado com a resposta esperada na especificação a uma dada ação de entrada.

O primeiro passo do algoritmo proposto consiste em inserir transições de suspensão na rede CT-CPNs para cada lugar de entrada que possa ser controlado externamente pelo testador. A segunda etapa consiste em adicionar lugares oráculo de teste para cada lugar observável, copiando as marcas dos lugares observáveis para os seus respectivos lugares oráculos. Nessa etapa também se adiciona transições de veredicto que possuem como pré-condição o lugar observável e seu lugar oráculo, e um lugar veredicto como pós-condição. Por último o algoritmo elimina os lugares e transições internas, já que não podem ser analisadas na metodologia de caixa-preta gerando assim a CT-CPN_{TS} para a execução dos testes contendo todos os casos de testes a serem avaliados.

Liu foca o estudo na avaliação das entradas e saídas do sistema olhando para os resultados obtidos após a execução não se preocupando com as etapas intermediárias, em outras palavras, há uma avaliação das causas e efeitos do *software* diferente da técnica aqui apresentada que avalia cada possível caminho, dependendo da cobertura escolhida, que o sistema pode percorrer.

2.4.2.7 Redes Controladas por Evento

Uma outra técnica proposta recentemente para geração de casos de testes, direcionada sistemas orientados a objeto e baseada em Redes de Petri envolve as redes orientadas a eventos (EDPN – “*Event Driven Petri Nets*”) e as redes EDPN mutantes (rede EDPN que possui ao menos uma transição de erro) - proposta por Gong (GONG e LI, 2008, p. 1113).

As EDPN (“*Event-Driven Petri Net*”) são definidas pela n -upla $P = (S, s_0, s_f, V, E_v, A, T, \sigma, In, out)$, onde S é um conjunto de estados, $s_0 \in S$ é o conjunto de estados iniciais, que denotam o período antes da criação do objeto, $s_f \in S$ é o conjunto de estados finais, isto é, o ponto onde o objeto é destruído, V é o conjunto de variáveis associadas aos objetos já existentes, E_v é conjunto de eventos (que invocam métodos dos objetos), A é um conjunto de ações (ações de entrada, de saída, de instanciação de uma variável, de condição de guarda), T é um conjunto de transições, $\sigma \in V$ é um objeto imaginário que armazena as ações externas ao sistema, $In \subseteq (S \cup E_v) \times T$ é conjunto de ações de entradas – sempre associadas às transições - e $out \subseteq T \times (S \cup E_v)$ é conjunto de ações de saídas, também associadas às transições.

Conforme dito anteriormente uma rede EDPN mutante é uma rede EDPN que possui ao menos uma dos três tipos de transição de erro. Sendo s um estado, f um evento de acionamento de transição e $T(s, f)$ o conjunto de transições cujo o estado inicial é s e que f foi disparada, definida por $T(s, f) = \{t \in T | s = t.source \wedge f = t.evento\}$, se $T(s, f) = \phi$, então $(s, s_e, f, true, \phi)$ é uma transição de erro da rede, ou se $T(s, f) = \phi$, e $\exists t_i (t_i.source = s \wedge t_i.f_n = f \wedge t_i.target = s_i, 1 \leq i \leq n)$, e $(t_1.guard \vee t_2.guard \vee \dots \vee t_n.guard) = true$ então $(s, s_i, f, (t_1.guard \vee t_2.guard \vee \dots \vee t_n.guard), \phi)$ também é uma transição de erro, ou se $T(s, f) = \{t_i \in T | 1 \leq i \leq n\}$, e $(t_1.guard \vee t_2.guard \vee \dots \vee t_n.guard) \neq true$ então $(s, s_e, f, \neg(t_1.guard \vee t_2.guard \vee \dots \vee t_n.guard), \phi)$ é uma transição de erro, onde $s_e \notin S$ é um estado de erro indefinido indicando que a transição de erro ocorreu e $s_i \in S$ é o estado alvo da transição t_i .

Nessa técnica, um cenário de teste é definido como sendo uma sequência de eventos externos gerados pelo teste que não podem ser reproduzidos pelas redes mutantes (implementações falhas), cada cenário tem pelo menos uma função de

comportamento que quando aplicada na transição alvo leva ao cenário. Considerando $q = t.pre \wedge t.post$, o algoritmo segue os seguintes passos: 1) para cada transição na rede mutante cuja $q_1 = t_1.pre \wedge t_1.post$ incluir a relação (t, q, t', q') da EDPN (t, q) e de sua mutante (t', q') equivalente no conjunto Q ; 2) Para cada elemento em Q , verificar se existe equivalência entre os eventos externos gerados em ambas as redes naquela transição. 2.1) Se houver intersecção entre os eventos e se os eventos externos gerados pela rede mutante estiverem contidos na rede comum e não forem os únicos, as transições da rede comum que servem de fonte para a transição avaliada devem ser incluídas em Q . 2.2) Caso os eventos externos gerados pela rede mutante estejam contidos na rede comum e forem os únicos na rede comum, as transições da rede comum que servem de fonte para a transição avaliada devem ser incluídas em Q , bem como as transições da rede mutante que servem de fonte para a equivalente mutante da transição avaliada. 2.3) Caso não haja intersecção, então o caminho correspondente do cenário gera um caso de teste passando pela transição. 3) Repetir o passo 2.

Este gerador é restrito para o uso de sistemas orientados a objetos focalizando os testes em possíveis situações de erro do sistema objetivando o encontro de falhas, ou seja, aborda as situações onde o sistema pode falhar através da criação de testes negativos (testa se o sistema faz o que não deveria fazer) oposto a técnica dessa dissertação que abrange todos os tipos de testes, incluindo os testes positivos (que testam se o sistema faz o que deveria fazer) e abrange qualquer tipo de sistema.

2.4.2.8 Outras Redes

Lun (LUN e XU, 2008, p. 1071) combina as definições de Rede de Petri e a semântica do π calculus para criar o conceito de π Behavior Graph (Grafo de comportamento π), que utilizado junto com a linguagem π -ADL (“Architecture Description Language”) produz um gerador de casos de teste. π calculus é uma ferramenta matemática rigorosa com semântica bem definida utilizada em várias áreas, especificamente em computação, e é utilizada para construção de modelos teóricos de movimentação, interação e concorrência de sistemas, através da linguagem π -ADL.

O Grafo de comportamento é definido como: $\pi BG = (Com, Con, Com_OM, Com_IM, Re)$, onde $Com = (S, T, F, OM, IM, s_0, s_f)$, OM é conjunto de informações de saída de um componente, IM é o conjunto de informações de saída de um componente, os arcos direcionados F são representados como $F \subseteq T \times (S \cup OM) \cup (S \cup IM) \times T$, Con é um conjunto de transições conectoras e $Re \subseteq (Com_OM \times Con) \cup (Con \times Com_IM)$. Lun sugere sete diferentes critérios de cobertura para serem utilizado no gerador: componente individual, função conector individual, componente para conector, conector para componente, componente direto para componente, componente indireto e todas conexões.

Lun implementa um gerador mais complexo ao utilizar o π calculus próprio para sistemas concorrentes cuja a estrutura altera com frequência, dessa maneira, a Rede de Petri proposta é mais abrangente que a apresentada nesse trabalho, podendo ser aplicada a esse tipo específico de sistema, no entanto, o nível de complexidade envolvido no conceitual dificulta sua aplicação necessitando que o arquiteto de testes conheça bem o conceito.

Liu (LIU, LIU, *et al.*, 2010, p. 214) propõem a aplicação de uma rede de Petri própria para modelar e analisar possíveis coberturas de testes de fluxos de trabalhos (casos de uso) dentro de softwares colaborativos. A rede de Petri proposta assemelha-se a uma EDPN modificada, definida por $P = (S, s_0, sf, M_0, In, Pes, Pro, B)$, onde In constitui os intervalos de disparo das transições, Pes os custos (pesos), Pro as probabilidades de ocorrência e B os valores condicionais.

O processo proposto por Liu divide o fluxo de trabalho de cada atividade em caminhos menores de acordo com sua estrutura (sequencial, paralela, seleção e repetição) e em cima dessa divisão aplica-se um dos três critérios de cobertura condicional: a cobertura por caminho (constitui na execução serial de todos os caminhos divididos ao menos uma vez seguindo uma ordem pré-fixada), a cobertura por ramo (constitui na execução de todas as condições para cada ramo dos caminhos alternativos – que precisam ser executados ao menos uma vez para cada processo mencionado no bloco) e a cobertura composta (que consiste na união dos caminhos alternativos levando em conta suas relações: serialização, englobado e sobre-sinal).

Para o caso de serialização (trechos executados em sequencia), temos o produto cartesiano entre os critérios de coberturas dos respectivos trechos ao gerar os caminhos compostos, já para o englobado um dos pedaços será um subconjunto do outro. No caso do sobre-sinal um trecho acaba funcionando como uma ramificação.

Ao incluir o fator de probabilidade de disparo das transições e suas considerações, o método de Liu acaba contribuindo com uma análise adicional para geração dos casos de testes que não é abordada nessa dissertação permitindo uma variação a mais nas ideias de testes obtidas sendo ideal para sistema que possuem fluxos alternativos importantes com grande variação probabilística.

Fazendo uso das características modulares e temporais das redes CMPN (*“Constraints-Based Modular Petri-nets”*), Reza (REZA e KERLIN, 2011, p. 568) apresentou um gerador de casos de testes alimentado por cenários correspondentes aos requisitos do *software*, mapeando os casos de usos do sistema em uma Rede de Petri. Essa rede de foi desenvolvida inicialmente como representação formal dos casos de usos do sistema apresentando características das redes TPN (*“Timed Petri Nets”*) e é uma rede modular, ou seja, pode ser dividida em componentes hierárquicos que facilitam a análise de sistemas complexos. Trata-se de um conjunto Ψ de redes de restrições $C_n = (S, T, F, W, R, M_0)$, onde $R: S \cup T \rightarrow \Psi^+$ é uma função de rótulo que associa um rótulo distinto de Ψ^+ a cada lugar ou transição. Cada rede C_i é composta por duas sub-redes: uma sub-rede que representa a estrutura interna do sistema e outra que representa as interfaces externas.

O gerador de Reza utiliza o método W (método que deriva casos de testes satisfazendo a cobertura de transições para um dado diagrama de transição de estados) que determina todos os caminhos admissíveis - dados os estados inicial e final (alvo) - utilizando todas as combinações válidas de localidades², além de gerar casos de testes conforme sua execução. Cada caminho encontrado nesse processo gera um único caso de teste.

O trabalho de Reza é a principal base para esta dissertação, assim sendo, os conceitos aplicados em ambos os estudos são praticamente os mesmos com pequenas abordagens diferentes. A principal diferença é que Reza divide o sistema

² O conceito de localidade em Redes de Petri consiste de um arranjo onde se tem pelo menos um lugar como pré-condição, e ao menos um lugar como pós-condição de uma transição, e a própria transição.

em várias Redes de Petri separando também o ambiente do sistema confundindo algumas vezes como elas se inter-relacionam diferente da abordagem adotada na dissertação apresentada onde todo o sistema e ambiente é construído em uma única rede com vários módulos representando cada entidade.

Outra diferença importante é que o método de Reza obtém os casos de testes com base nos estados inicial e final do sistema, ou seja, ele obtém um único caso de teste por aplicação do método que valida o fluxo inteiro do sistema sendo mais indicado para testes de homologação do tipo “*End-to-End*” (fim-a-fim) ao contrário do método proposto.

Dong (DONG, 2008, p. 814) utilizou uma técnica de redução do número de casos de testes através de uma rede de alto nível genérica aplicada a “*Web Services*” baseados em BPEL (“*Business Process Execution Language*”). Como a maioria dos erros ocorre no relacionamento entre si dos parâmetros de entrada, Dong utilizou a técnica de combinação de pares de entrada para obter a redução dos casos de teste.

O primeiro passo dessa proposta consiste em montar subconjuntos das relações dos parâmetros de saída (de acordo com o escopo de testes), onde cada relação colocada nesse conjunto não pode ter dois parâmetros de saída repetidos em um mesmo subconjunto. Em seguida deve-se escolher para cada subconjunto um parâmetro de entrada cujo elemento correspondente esteja dentro desse conjunto. Para cada arranjo de parâmetros de saída, constrói-se um arranjo reduzido onde os parâmetros de entrada levantados anteriormente estejam inseridos. Em seguida, basta construir a suíte de teste (conjunto de casos de testes) com os parâmetros selecionados e montar para cada caso de teste criado nessa suíte um caso de teste no conjunto total.

O estudo de Dong centralizou na redução dos casos de testes através dos parâmetros de entrada do sistema e não na obtenção dos casos de testes. Desta forma, sua técnica pode ser utilizada em complemento com a extração de casos de testes aqui apresentada para a redução dos casos de testes mantendo a cobertura desejada.

2.4.3 Aplicações de Ontologias a Teste de Software

Os próximos parágrafos apresentam alguns trabalhos recentes envolvendo ontologia.

Wang (WANG, BAI, *et al.*, 2007, p. 43) implementa duas perspectivas para gerar casos de testes em “*Web Services*”, fundindo os conceitos de Rede de Petri e OWL-S (“*Web Ontology Language for Web Services*”). Ambas as perspectivas consistem em criar uma Rede de Petri lugar-transição e especificar com base nesse modelo o serviço em OWL-S.

A primeira perspectiva, chamada pelo autor de Geração de Processo de Teste, consiste em analisar cada braço da Rede de Petri de um processo levando em consideração os tipos de controle necessário (Separação “*Split*”, Sequência “*Sequence*”, Iteração “*Iterate*”, Se-Então-Senão “*If-Then-Else*”, Escolha “*Choice*”, Qualquer-Ordem “*Any-Order*” e Separar-Juntar “*Split-Join*”) e gerar um caso de teste que cubra esse caminho. A segunda perspectiva, Geração de Dados de Teste Baseados em Raciocínio Ontológico, analisa as entradas, saídas, pré-condições e efeitos de um processo descrito em OWL-S gerando assim a massa de dados para os casos de testes levantados anteriormente.

Avaliando o método de extração de caso de teste, a aproximação de Wang se assemelha a utilizada nesse trabalho, porém aborda diretamente as tomadas de decisões existentes dentro da rede para extrair os casos de testes ao invés de avaliar processo a processo. Ele também usa uma Rede de Petri mais simples não modular o que dificulta a representação de sistemas mais complexos.

O propósito do modelo ontológico gerado no artigo de Wang é a geração de massa de testes enquanto que na dissertação é empregado para validação dos resultados após os testes.

Fan (FAN e WANG, 2012, p. 46) propõem um gerador de casos de testes de validação para sistemas digitais de instrumentação e controle na indústria nuclear. O gerador cria os casos testes baseado no documento de Necessidades de Segurança do Usuário (SAR, “*Safety Analysis Report*”), documento padrão para essa indústria envolvendo três fases: construção de um modelo ontológico do SAR, escolha de um dos quatro critérios de coberturas proposto no artigo e por último a extração dos casos de testes.

Para Fan um conjunto de ferramentas próprio foi desenvolvido, sendo que todo o estudo de caso foi realizado dentro dele. O primeiro passo contempla, como escrito anteriormente, na construção de um modelo ontológico (domínio-específico, "*domain-specific*") baseando-se no SAR demonstrando de forma clara os conceitos e as relações nesse domínio. Em seguida, escolhe-se um critério de cobertura: cobertura por parâmetro simples ("*simple parameter coverage*", todos os parâmetros existentes na planta nuclear são testados ao menos uma vez), cobertura por caso de uso ("*use case coverage*", cada caso de uso, ou no caso, cada evento possível é testado), cobertura por condição irregular ("*abnormal condition coverage*", cada situação hipotética de falha nos softwares e hardwares envolvidos são testados) ou cobertura por cenário ("*scenario coverage*", testes de novos casos de usos gerados após variação em parâmetros dentro dos casos de usos padrões).

O artigo cria no passo seguinte uma linguagem de marcação XML para representar a ontologia utilizando SAML ("*Safety Analysis Markup Language*", Linguagem de Marcação de Análise Segura). O SAR é marcado usando as etiquetas, atributos e links do SAML criado nessa etapa. Dados numéricos e informações relacionadas são retirados do documento marcado e usados para gerar os casos de testes de validação.

A interpretação de Fan é voltada para um tipo de sistema particular focando nas suas necessidades específicas. Esse trabalho se assemelha a dissertação proposta se considerar o SAR como uma especificação do sistema a ser testado contendo informações do comportamento dinâmico e das regras de negócios. Feita essa comparação, as coberturas abordadas por Fan focam em uma análise da documentação (o que sistema deve ou não deve fazer) ao invés em uma análise em cima dos fluxos obtidos ao criar o modelo.

Zhu (ZHU e ZHANG, 2010, p. 116) introduz um novo *framework* para testes colaborativos em serviços *web* utilizando STOWS ("*Software Testing Ontology Web Service*", (ZHU, 2006, p. 146)) que na verdade é uma adaptação de uma linguagem ontológica para teste baseada em UML, agora implementada em OWL. O *framework* é capaz de se integrar com sistemas em operação sem causar impactos nas operações reais, e para isso utiliza serviços de teste ("*test services*" ou T-serviços) para designar essas operações de teste sobre transações reais ou serviços funcionais ("*functional services*" ou F-serviços). Cada T-serviço deve representar

fielmente um F-serviço, porém devem possuir um *firewall* para evitar que causem efeitos no mundo real provocando distúrbios nos sistemas externos.

Para esse *framework*, também deve existir T-serviços que servem para realizar tarefas gerais de teste como planejamento e geração de casos de testes. Esses T-serviços são chamados testadores gerais ("*general testers*") para distingui-los dos demais. Especificamente, o agente de teste ("*test broker*") é importante pois controla a execução dos testes recebendo as tarefas, dividindo em sub-tarefas, criando planos de testes, encontrando testadores executáveis e apresentando os resultados aos usuários.

Para Zhu, a comunicação entre os T-serviços ocorre via o protocolo SOAP ("*Simple Object Access Protocol*", protocolo para troca de informações sobre estruturas implantadas de serviços web), guardando suas descrições de serviços no registrador Semântico de Serviços Web ("*Semantic WS*", servidores para interação máquina-máquina via rede), composto por um registrador UDDI ("*Universal Description, Discovery and Integration*", registrador e localizador de serviços web) e um associador entre os serviços solicitados e os executados, onde a linguagem escolhida para representação foi a STOWS. Por último, também existe um serviço destinado a atualizar a ontologia, o OMS ("*Ontology Management Service*") construindo sobre o programa Protégé.

O STOWS é dividido em três conceitos:

- **Elementar**: São conceitos gerais sobre *softwares* e *hardwares*, tais como tipos de *hardwares*, artefatos de *softwares* e seus formatos...;

- **Básico de teste**: Inclui um Testador (parte que participa de uma atividade e tarefas de teste com capacidade para poder ou não executar uma tarefa), Artefato (materiais de entrada ou saída de atividades com uma localização, como casos de testes e relatórios), Atividades (conjunto de ações com um objetivo, como, por exemplo, planejamento de teste, execução de teste, geração de relatório de teste, medição adequada e outros), Contexto (define o propósito das atividades de teste), Método (forma com a qual a atividade será executada) e Ambiente (configuração de hardware e software utilizada para os testes);

- **Composto de teste**: Inclui Tarefa (que possui a atividade a ser executada, o contexto, o método, o ambiente e os artefatos de entrada e saída) e Capacidade (relacionada ao testador, demonstra quais atividades, contextos, ambientes,

métodos e artefatos necessários que aquele testador pode executar uma determinada atividade).

Essa linguagem também possui algumas restrições na manipulação dos dados. As classes devem ser divididas em dois grupos: classes elementares (são as classes que formam o *framework*, e nenhuma delas pode ser apagada para que não haja dano na estrutura da ontologia) e classes estendidas (são classes que podem ser adicionadas e apagadas da hierarquia sem causar danos). Dentro dessa divisão foram implementados dois verificadores: o Verificador de Autoridade ("*Authority Checker*", verifica se uma classe a ser deletada é uma classe estendida) e o Verificador de Conflito ("*Conflict Checker*", que verifica se a nova classe a ser adicionada já existe na ontologia e se a classe a ser apagada não possui subclasses).

O *framework* proposto por Zhu consegue operar em paralelo com sistema em produção sem causar danos as funcionalidades do mesmo, reduzindo a necessidade de um ambiente apartado para testes e reduzindo a necessidade de criar massas fictícias para o teste através do uso de massas reais o que não seria possível fazer na tratamento realizado neste trabalho. Zhu também aborda o controle da execução através dos agentes de testes etapa não avaliada no método desta dissertação, por outro lado, o *framework* é restrito para testes de serviços *web*.

No trabalho de Huang (HUANG, TSAI, *et al.*, 2005, p. 300), é criado um validador de modelos em OWL-S ("*Web Ontology Language for Web Services*", OWL para serviços web) de serviços web compostos, ou seja, serviços que possuem pré-condições e/ou pós-condições que dependem da execução de outro serviço além de um gerador de casos de testes estendido da ferramenta BLAST. O BLAST (HUANG, TSAI, *et al.*, 2005, p. 300) é um validador e gerador de casos de testes positivos capaz de interpretar fluxos de controle e possui a propriedade predicado-limite ("*predicate-bound property*") que valida se um predicado em um determinado local possui um valor verdadeiro.

Inicialmente modela-se cada serviço em um fluxo de controle em OWL-S possibilitando monitorar, invocar, compor e descobrir serviços de forma automática, que em seguida é convertido para uma linguagem de especificação baseada em C. As entradas, saídas, pré-condições e efeitos apropriados devem ser levados em conta no modelo, caso contrário algumas propriedades não poderão ser checadas.

As operações *escolha* (ou um caminho ou outro), *separação* (de um nó parte dois caminhos), *junção* (dois caminhos se unem em um nó) e *sem-ordem* (não existe uma sequência certa para execução de determinados nós) não são reconhecidas pelo BLAST e, portanto são descritas utilizando fórmulas lógicas da linguagem PDDL ("*Planning Domain Definition Language*", (GHALLAB, HOWE, *et al.*, 1998, p. 1)) Huang também introduz um algoritmo para tratar essas operações quando há dependência entre serviços, onde os serviços independentes sequenciais são organizados em blocos básicos, e cada caminho de um conflito é varrido simultaneamente, obtendo assim todas as ordens de execução possíveis para a implementação.

No próximo passo, essa especificação é incrementada com todas as propriedades que serão verificadas, preparando-a para a ferramenta BLAST através da introdução de sentenças indicando o momento em que os erros ocorrem. Essa especificação é analisada pelo BLAST que verifica as propriedades indicando se há erros e gera casos de testes positivos para o modelo. Por último, utilizando a linguagem de especificação *C-like* e casos de testes positivos, os testes negativos são gerados. São gerados apenas os casos relacionados a valores limítrofes, partições, valores aleatórios e planos de exemplos ("*plain-samples*") através da variação de variáveis e dos valores associados a ela.

O modelo em OWL de Huang é utilizado como um gerador de casos de testes e não como uma ferramenta de validação de resultados como nessa dissertação possuindo uma limitação para a obtenção de casos de testes negativos onde é possível apenas a análise para valores limítrofes e partições. De outra forma, esse método permite avaliação ao mesmo tempo de vários serviços interligados de maneira relativamente simples, que, quando utilizada dentro do método proposto nesse trabalho, dependendo da complexidade dos serviços pode dificultar a análise dos resultados.

García-Castro (GARCÍA-CASTRO e GÓMEZ-PÉREZ, 2012, p. 1413) criou um gerador de massa de dados para testes de conformidade em ferramentas que utilizam OWL DL ("*Ontology Webservice Language Description Logic*", linguagem que suporta Lógica Descritiva possuindo todos os recursos da OWL convencional, porém com algumas restrições para garantir uma maior expressividade, mantendo a completude computacional e decidibilidade (W3C, 2004)). Os testes para essas ferramentas consistem em fazer a importação utilizando o modelo ontológico gerado

(massa de dados) e logo depois fazer a exportação comparando o modelo exportado com o original.

Para construir o gerador, assumiu-se que a conformidade da ferramenta pode ser avaliada de três pontos de vista: do modelo interno ontológico (semelhanças e diferenças); da serialização da linguagem suportada e das semânticas formais suportadas; e através de três métricas: resultado da execução (OK, falha, abortada), informações adicionadas ou perdidas e conformidade. Primeiramente os casos de testes devem ser construídos utilizando determinadas palavras-chaves que representam uma sequência de ações e definem uma atividade. As palavras-chave devem também ser acompanhadas por parâmetros específicos para aquela atividade.

Cada nova definição de palavra-chave deve ser armazenada em uma biblioteca que será usada pelo gerador de massa. A biblioteca deve abordar alternativas nas regras de produção (palavras diferentes para cada símbolo alternativos), diferenciação de símbolos opcionais, palavras representando repetições e símbolos não-terminais.

Após construir os casos de testes, eles devem ser agrupados em um *script* que representa a suíte que será executada podendo haver uso de macros internas para reaproveitamento do código. Esse *script* é ajustado por um pré-processador que substitui cada chamado de uma macro pelo código e instancia seus parâmetros adaptando o *script* para o interpretador. Por fim, o interpretador irá construir os modelos ontológicos para os testes e gerar meta-dados descrevendo cada teste presente na suíte que poderá ser usado para a automação.

A massa de dados gerada por esse método consegue cobrir os seguintes testes: descrição de classes, testes de axiomas de classes, combinação de axiomas de classes, descrição de propriedades, combinação de axiomas de propriedades, propriedades de domínio e intervalo, descrição de instâncias, combinação de axiomas de instâncias, instâncias e propriedades, intervalo de dados e propriedades de anotações.

O trabalho de García-Castro foi focado na geração de massas de testes para testar especificamente a conformidade de ferramentas que utilizam a linguagem OWL não avaliando nenhuma outra etapa e restringindo a aplicação apenas para esse tipo de ferramenta. Na dissertação aqui apresentada, não é abordado a criação

da massa, apenas um mapeamento prévio com intuito de facilitar a identificação dos defeitos.

Bai (BAI, LEE, *et al.*, 2008, p. 465) propõem um Modelo de Teste Ontológico (TOM, "*Test Ontology Model*") para especificar conceitos de testes tanto da fase de execução como da fase de arquitetura baseado em OWL-S. Essa técnica promove um gerador de artefatos de testes (casos de testes, plano de testes, massa de dados...) automático para testes de partição através da interpretação e processamento do modelo do serviço Web criado em OWL-S.

O método consiste em ter uma infra-estrutura ontológica contendo editor, analisador, *reasoner* e o domínio do serviço da aplicação e da estratégia de teste que deverá receber o modelo contendo a especificação em OWL-S.

Esse modelo é submetido a um gerador de teste, que irá criar os casos de testes e convertê-los para o modelo TOM, para um mestre de teste, que irá criar os planos de teste e coordenar a execução, e para agentes de testes, que irá mobilizar e exercitar o serviço com os casos de testes. Do ponto de vista da arquitetura, os casos de testes e as *suítes* (conjuntos de casos de testes) serão definidos levando em consideração a massa de dados necessária (incluindo partições, estratégia de uso da massa e valores), o comportamento dos testes (procedimentos para preparar o ambiente para teste) e resultados esperados. Do ponto de vista da execução, serão levantadas as informações necessárias para exercitar os casos já elaborados considerando: o plano de teste (agendamento, organização e configuração para a execução dos casos de testes) e o resultado dos testes (dados coletados após a execução para futura análise).

Bai também inclui nesse modelo, um gerador de massa de dados para testes, onde o modelo do serviço serve como entrada para geração de dados para as partições de testes. Nesse gerador, o modelo do serviço é primeiramente mapeado em um modelo de dados das partições, onde cada partição de dados é dividida em subclasses representando cada propriedade do serviço. Após o mapeamento são inseridas as restrições (como valor mínimo, enumerações, tamanho...) e eliminadas as classes redundantes criando o modelo inicial de dados para os testes. Por último um chegador de completude e de consistência ("*C&C checking*") é aplicado no modelo avaliando-o tanto contra o TOM quanto contra a especificação OWL-S.

A abordagem de BAI vai muito além da obtenção das ideias de testes e da validação dos resultados da execução tratada nessa dissertação, ele engloba praticamente todas as etapas do teste desde o planejamento até o fim da execução automatizando praticamente todo o processo de teste. Por outro lado, a técnica funciona apenas para testes de partição não envolvendo, por exemplo, testes de fluxos, testes de validação de telas e tabelas de decisões.

3 MODELO PROPOSTO

3.1 VISÃO GERAL

As principais etapas dos métodos complementares apresentados são demonstradas na Figura 1. O primeiro método aplicado neste trabalho envolve a primeira e terceira etapas, consistindo em modelar novos sistemas ou executar alterações em sistemas já existentes utilizando uma rede de Petri modular, seguindo determinadas regras e extraíndo as ideias de testes de forma a atingir uma cobertura pré-determinada. A metodologia utilizada é indicada para testes de caixa-preta (sem conhecimento do código, apenas dos requisitos e casos de uso), mas pode ser facilmente adaptada para testes de caixa-branca (conhecimento do código). Esse método visa a obtenção das ideias de teste, ou seja, indica “o que” precisa ser testado e não “como” testar.

O segundo método abrange a segunda etapa e as duas últimas, consistindo em modelar o sistema em OWL transformando as informações sobre as regras de negócio do aplicativo em axiomas e, após a execução da bateria de testes com os casos de testes obtidos a partir das ideias extraídas no primeiro método, atualizar o modelo com os dados alterados na base durante a execução e verificá-los através de um mecanismo de prova automática se as regras de negócio não foram violadas, além de verificá-los através da comparação dos estados dos indivíduos antes e depois da execução. Enquanto que a quarta etapa não é tratada neste trabalho.

Figura 1 - Representação das etapas dos métodos aplicada dentro do processo de teste

3.2 REPRESENTAÇÃO DA DINÂMICA DO SISTEMA

Nesta secção é descrito como é aplicado a primeira etapa do primeiro método que compreendem o quadro 1 na Figura 1. Segundo Neto (NETO, 2013), antes de ocorrer a extração das ideias de testes, um modelo em rede de Petri contendo os fluxos do sistema deve ser criado. Avaliar o sistema (ou os pedaços do sistema envolvidos na alteração) deve ser a primeira tarefa a ser realizada nesse método. Analisando o sistema, é possível identificar entidades e as ações dos agentes que levam a sua evolução, como, por exemplo, pequenas mudanças locais nas telas, envios ou recebimentos de informações simples, entre outros. Nessa atividade deve ser considerado o ciclo de vida dessas entidades bem como o ciclo do sistema como um todo separando em níveis quando conveniente.

A segunda atividade na modelagem é composta por levantar todos os estados possíveis de cada nível (macro e de cada entidade identificada), bem como as transições de um estado para outro. Nessa etapa, todas as regras de negócios envolvidas que influenciam nos fluxos e nas tomadas de decisões devem ser consideradas.

Levantado esses dados, modela-se cada nível de cada sub-módulo na rede de Petri modular, começando pelo nível mais baixo que representa cada elemento e subi-los até o nível macro do sistema. Nesse ponto, pode-se criar uma rede de Petri modular para cada sub-módulo.

A última atividade consiste em identificar o relacionamento entre cada entidade e unificando os módulos da rede de Petri através do uso dos macros estados e transições. Durante essa aplicação foi identificado que cada relacionamento pode ser classificado como uma das oito formas descritas abaixo:

Figura 2 - Exemplo de transição que é habilitada por um lugar externo

Figura 3 - Exemplo de situação onde a transição afetada por um lugar em outra entidade que fornece uma marca

- **Transição em uma entidade depende de um ou mais Lugares em outras entidades**: Ocorre quando o lugar habilita, inibe ou fornece marcas para a transição, como por exemplo, quando o pagamento da segunda parcela depende que a primeira parcela esteja liquidada. Para identificar essas situações é necessário verificar se as transições de uma determinada entidade possuem lugares externos a ela que influenciam em sua ativação. Em seguida para cada lugar identificado que a influência no sentido de habilitar/desabilitar a transição, deve-se criar um pseudolugar na entidade e inserir os devidos arcos habilitadores/inibidores ligando-os até as transições conforme demonstrado na Figura 2. Se o lugar identificado influenciá-la no sentido de movimentar marcas, deve-se criar um lugar intermediário externo (macro) as duas entidades com uma transição externa que recebe a marca do lugar que influencia conforme Figura 3. Na entidade que possui o lugar antigo, também será necessário interligar ao novo lugar as suas transições que possuam o lugar antigo como pré-condição, transformando o lugar novo na nova pré-condição. Para as outras entidades afetadas, interligar o novo lugar indicando quais as transições que o lugar antigo afetaria.

- **Lugar em uma entidade depende do disparo de uma transição em outra entidade**: Ocorre, por exemplo, quando o pagamento da última parcela gera a liquidação do contrato. Primeiro deve-se levantar quais as ações (manuais e automáticas) alteram o estado de cada entidade envolvida. Para cada ação, verifica-se se ela altera o estado das outras entidades: se a transição não modificar, cria-se uma transição interna a entidade incluindo a relação entre a transição e os lugares internos; se modificar, cria-se uma transição externa as entidades afetadas, incluindo os arcos que indicam as pré-condições (externas as entidades) e os arcos que indicam as pós-condições (externas as entidades), além de criar a transição

interna em cada entidade afetada indicando também as pós e pré-condições da mesma. Também é necessário criar um lugar interno a entidade que possui a transição conforme Figura 4. Por último verificar se a transição não afeta ou é afetada por outros lugares do macro modelo. Se afetar incluir os arcos correspondentes.

- **Duas entidades ou mais são criadas simultaneamente:** Nessa classificação uma transição acaba criando duas ou mais entidades no modelo, como exemplo, pode-se citar a criação de parcelas mensais automática informando apenas o valor total do contrato e numero de parcelas. Para esse caso, é preciso verificar se as transições que criam novas entidades podem gerar mais de uma entidade simultaneamente: se não existir, associar a transição através de arcos aos lugares iniciais do ciclo de vida da entidade; se existir, associar a transição através de arcos a cada entidade, lembrando de duplicar a entidade caso as entidades criadas sejam semelhantes (exemplo: todas as entidades criadas são parcelas de um mesmo tipo), a Figura 5 demonstra esse caso. Ao final, verificar os elementos externos as entidades criadas que também sofrem influências e criar as devidas conexões.

Figura 4 - Situação onde a transição na Entidade_Transicao_Afeta afeta um lugar em outra entidade

Figura 5 - Transição criando duas entidades simultaneamente

- **Duas ou mais entidades morrem simultaneamente:** Um exemplo simples é o cancelamento da operação antes do fechamento do contrato, resultando na morte do contrato e das parcelas criadas até o momento. Essa condição é semelhante a condição de criação simultânea, apenas os lugares internos das entidades afetadas que são diferentes, na criação trata-se do lugares iniciais e na morte trata-se do lugares finais.

- **Uma transição externa possui como pré-condição que uma ou mais entidades tenham concluído seu ciclo de vida:** A transição é habilitada ou inibida com a morte de uma ou mais entidades. Por exemplo, o pagamento da primeira parcela habilita o pagamento com desconto de uma segunda parcela. Para cada entidade cuja conclusão do ciclo de vida influencia o processo como um todo, associar pseudolugares que representam os lugares terminais do ciclo de vida dessa entidade as transições afetadas através de arcos habilitadores e/ou inibidores de acordo com a Figura 6. Em seguida, verificar se as transições que dependem da conclusão do ciclo de vidas de alguma entidade dependem da morte de outras entidades: se existir, criar e associar os pseudolugares que representam os lugares terminais das demais entidades que afetam a transição através de arcos inibidores e/ou habilitadores a transição.

Figura 6 - Figura apresentando uma transição que é influenciada pela morte de uma entidade

- **Criação de uma entidade depende de um ou mais Lugares em outras entidades**: Quando a transição de criação de uma nova entidade é habilitada, inibida ou precisa de marcas de lugares em entidades já existentes. Exemplo: a criação de uma nota fiscal referente a um pagamento depende da parcela esta liquidada. A modelagem dessa classificação é semelhante a modelagem da situação onde a transição de uma entidade depende de lugares em outras entidades (Figuras 2 e 3). Primeiro avalia se cada entidade depende de lugares em outras entidades para a sua criação. Para cada dependência, verifica se esta é uma dependência de inibição, habilitação ou transferência de marca: no caso da dependência de habilitação ou inibição, criar pseudolugares no macro modelo que representam os lugares das demais entidades que habilitam ou inibem a criação da nova entidade e ligar os devidos arcos a transição que é responsável criar a entidade; no caso de dependência de transferência de marcar, conectar a entidade que afeta a criação da nova à transição, incluindo os devidos acertos internamente na entidade.

- **Morte de uma entidade depende de um ou mais Lugares em outras entidades**: (habilita, inibe ou fornece marcas para a transição). Exemplo: O cancelamento da segunda parcela depende do status da primeira parcela. Nesse caso, também se aplicam as regras de quando uma transição em uma entidade depende de um ou mais Lugares em outras entidades (Figuras 2 e 3), sendo que esta transição interna a entidade é responsável por sua morte.

- **Duas ou mais entidades são afetadas simultaneamente por uma transição externa**: Exemplo: Renovação do contrato alterando as datas de pagamentos de todas as parcelas e possivelmente alterando os seus status (de vencida para aberta). Levanta-se quais as transições que alteram o estado de cada entidade envolvida. O próximo passo consiste em incluir os arcos que indicam as pré-condições (externas as entidades) e os arcos que indicam as pós-condições (externas as entidades) ligando-os as entidades afetadas, conforme apresentado na Figura 7. Por último, alteram-se internamente cada entidade afetada indicando a relação da transição externa com a mesma.

Figura 7 - Desenho apresentando uma transição externa afetando duas entidades

3.3 REPRESENTAÇÃO DO PROPÓSITO DO SISTEMA ATRAVÉS DE ONTOLOGIA

Esta secção corresponde a primeira etapa do segundo método representado pelo segundo quadro na Figura 1. Após a representação dinâmica do sistema, deve-se criar o modelo em OWL com base nas regras de negócio do sistema. Para tanto é necessário identificar os componentes-chaves no negócio da aplicação seguindo as regras abaixo, tendo-se em itálico os conceitos fundamentais da OWL:

- **Classe** (“*class*”): devem representar o componente em si, ou seja, representar cada conceito ou entidade do sistema, como por exemplo, Cartão, Empréstimo, Contrato...;
- **Propriedades de dado** (“*datatype properties*”): devem representar os atributos das entidades ou conceitos, como exemplo temos Identificador, Valor, Forma de Pagamento para parcelas... Normalmente são descrito por uma relação do tipo *has_Thing*;
- **Propriedades de objetos** (“*object properties*”): são as relações entre as entidades. As mais utilizadas serão *is_Part_Of* e *has_Part*;

Esta etapa segue as relações relativamente bem estabelecidas entre o Modelo Relacional ou o Modelo de Objetos e suas contrapartes na OWL.

Em específico um grupo especial de classes deve ser criado de forma a tornar possível algumas validações envolvendo erros estruturais de declaração, por exemplo, criação de indivíduos de uma classe inexistente no modelo, criação de indivíduos que se relacionam com outros indivíduos que não estão declarados no modelo ou criação de indivíduos sem alguma propriedade chave, por exemplo, o ID.

Essas classes devem ser disjuntas de todas as demais e qualquer indivíduo criado e classificado em uma delas deve causar um erro de inferência no modelo, ou seja, um sistema funcionando perfeitamente não deve possuir indivíduos com essas características.

Outro grupo de classe que deve ser criado deve ser os casos de testes, que são classes que representam as regras de negócio que serão testadas na situação específica do teste, por exemplo, o sistema possui uma regra em que a quantidade de peças no estoque deve ser maior ou igual que zero, para essa situação deve existir uma classe do tipo caso de teste onde sua descrição indica que a instância deve ser do tipo estoque e possui relação de 0 ou mais peças. Nesse exemplo apenas os estoques válidos serão membros dessa classe caso de teste, de forma que um teste positivo consiste em verificar se o indivíduo está presente nessa classe enquanto que um negativo consiste em verificar se o indivíduo não pertence a ela.

Em alguns casos de testes, para a validação dos casos negativos, há a necessidade de criação de classes casos de testes reversas cujo objetivo é verificar se os membros não presentes na classe caso de teste original não estão sem classificação, ou seja, verificar se o fato do modelo não ter os classificados na classe original foi por falta de dados. Essas classes só são necessárias quando a existência de uma instância fora da classe de teste é válida para o sistema. Por exemplo, classe de caso de teste valida se uma instância possui o valor 10, mas o sistema aceita valores de 0 a 20, nesse caso será necessário uma classe reversa onde todos os indivíduos que não possuam valor 10 sejam membro, se a classe caso de teste fosse a instância precisa ter valor 10 e o sistema só aceitasse instâncias com esse valor, não haveria necessidade de uma classe reversa, pois todos os indivíduos que não obedecessem a essa regra gerariam um erro de inferência no modelo dispensando a necessidade da mesma.

Uma vez identificado os componentes, deve-se montar os axiomas com essas entidades e construir o modelo. Os axiomas (“*axioms*”) representam as regras que definem o que são as classes, ou seja, são as formas de representação das regras de negócios. Como exemplos de axiomas pode-se citar: “Contrato possui no mínimo x Parcelas”, “Parcela deve possuir Valor máximo de Y Reais”, “Cartão é um Meio de pagamento”, entre outros. Em certo sentido, corresponde aos requisitos do sistema ou, em certos casos, o que se denomina regra de negócio.

O OWL por si só possui algumas restrições para a aplicação de alguns axiomas, por exemplo, não é possível representar regras aritméticas. Para contornar esse problema, foi utilizado o SWRL para a criação desses axiomas diferenciados. Conforme citado anteriormente o SWRL permite o uso de operações para comparação entre classes e indivíduos, comparação entre valores (maior, menor, igual e diferente), operações aritméticas (soma, subtração, multiplicação, divisão, módulo, potenciação, arredondamento, funções trigonométricas, entre outras), operações booleanas, operações com strings (comparação, concatenação, separação, caixa alta e caixa baixa, substituição, contenção, entre outras), operações com data, hora e duração, operações com URI e operações com listas que ampliam a abrangência do modelo OWL.

3.4 EXTRAÇÃO DE IDEIAS DE TESTES

A segunda e última etapa do primeiro método representado pelo quadro três da Figura 1 compreende na extração de ideias de testes a partir de um modelo bem definido de um sistema construído na primeira etapa, em outras palavras, consiste em aplicar regras bem definidas em um modelo (no caso em Rede de Petri Modular) criando uma lista do que deverá ser testado do sistema. Essas regras estão diretamente associadas a uma cobertura de teste que determinará a qualidade e o custo da execução.

Porém, antes da extração das ideias de testes deve-se validar os modelos criados comparando-o com a documentação do projeto averiguando se cada funcionalidade envolvida é representada corretamente, caso exista algum erro ou omissão de alguma informação, deve-se retornar as etapas anteriores e incluir as alterações necessárias.

Uma vez validado o modelo, uma cobertura de teste desejada é escolhida e aplicada obtendo-se os casos de testes. Entre os possíveis critérios de coberturas, é indicado o uso de uma das três coberturas sugeridas por Cai em seu artigo (CAI, ZHANG e LIU, 2010, p. 1): cobertura por estado (onde todos os estados devem ser testados ao menos uma vez), cobertura por transição (onde todas as transições devem ser testadas ao menos uma vez) e cobertura por par de transições (onde

todos os possíveis pares de transição e o estado entre elas, são testados ao menos uma vez).

O artigo de Cai não abrange lugares habilitadores ou inibidores e nem quando um sub-módulo possui mais de uma transição de entrada, para cobrir esses pontos o extrator aqui proposto abrange todas as possibilidades de habilitação e inibição nas três coberturas propostas, bem como abrange todas as transições de entrada no sub-módulo.

Ao fim da extração deve-se eliminar as ideias de testes incoerentes, como por exemplo, aqueles que envolvem lugares ou transições representadas no modelo para fins de controle e ideias de testes que quando testado isoladamente não possuem sentido. Também deve-se fazer uma avaliação verificando a presença de ideias de testes duplicadas presente entre os níveis diferentes modelados.

3.5 VERIFICAÇÃO BASEADA EM ONTOLOGIA

Após a extração deve-se escolher os testes dentre aqueles extraídos pelo extrator para formar bateria a ser executada levando em conta quais serão os resultados esperados no banco de dados (quadro quatro da Figura 1, não abordado nesse trabalho) e mapeando essa massa de dados de saída. Esse mapeamento torna-se necessário para que haja a rastreabilidade do modelo ontológico para o código quando houver detecção de inconsistências.

Tipo da Identidade	Parâmetros de Identificação	Valores dos parâmetros	Criada durante a execução	CT
Pessoa Física	RG; CPF; Endereço	99.567.433-2; 283.987.912-00; Rua Pedro IV, 90	Não	002
Pedido	Numero do Pedido; Data	00987655523; 02/05/2014	Sim	107
Pedido	Numero do Pedido; Data	00987655600; 10/05/2014	Sim	107

Tabela 3 - Exemplo de mapeamento de dados

O mapeamento da massa consiste em identificar a massa necessária que precisa estar no sistema para testar bem como identificar o que deverá ser criado durante a execução dos testes. Para tanto as entidades que serão usadas devem possuir no nome a identificação do CT que irá interagir com elas, por exemplo, cartão_A_CT001, pessoa_B_CT055. Na impossibilidade de renomear as entidades existentes deve-se criar uma tabela de suporte que relacionem CT a entidade conforme exemplo mostrado na tabela 3.

Antes da execução da bateria de testes, os indivíduos que já existiam e irão interagir com os casos de testes da bateria devem ter suas informações de interesse extraídas e inseridas no modelo em OWL. Cada componente deve ser inserido com uma instância (“*individual*”) de sua respectiva classe bem como seus atributos e relações são inseridos conforme consta nos dados extraídos da base criando o estado inicial da base. Esses indivíduos devem ser identificados com a palavra “antigo” em seus nomes. Cada individuo deve possuir uma propriedade de dados que identifique seu nome de forma a facilitar sua localização antes e depois dos testes, bem como deve possuir uma propriedade de objeto associando a um estado da base.

A próxima atividade é a execução da bateria de teste que poderá ser tanto manual quanto automática, sendo esta ultima mais eficaz para o método aqui descrito.

Após a execução, extraem-se as informações de interesse novamente da massa de dados inserindo-as no modelo em OWL efetuando a segunda etapa do segundo método representado pelo quadro cinco na Figura 1. Cada componente deve ser inserido com uma instância de sua respectiva classe bem como seus atributos e relações são inseridas conforme consta nos dados extraídos da base da mesma forma que a extração anterior, porém devem ser identificados com a palavra “novo” em seus nomes. Se algum atributo não constar na base ou se alguma relação estiver quebrada, esta não deve ser inserida mantendo o modelo fiel a situação atual da base.

Em seguida, um mecanismo de inferência (“*reasoner*”) deve ser ativado utilizando o modelo em OWL de forma que todas as relações sejam testadas concluindo o segundo método (quadro seis da Figura 1). Se o modelo continuar consistente, implica que não houve falhas na execução dos casos de testes da bateria. Caso o contrário, indica que há um defeito na aplicação, esse defeito pode

ser rastreado através do mapeamento realizado na etapa de construção da bateria facilitando a detecção do erro no código.

O último passo consiste em comparar os indivíduos no estado antigo com os indivíduos no estado novo, dessa forma é possível verificar se os testes afetaram as entidades conforme o esperado. Essa técnica permiti encontrar erros que do ponto de vista do negócio estão corretos, por exemplo, o sistema deveria alterar um valor de 5 para 10, mas no entanto alterou para 8, como o sistema aceita o valor 8 o erro não será pego nas inferências, apenas nessa segunda validação onde o usuário perceberá o valor diferente do descrito no caso de teste.

4 ESTUDO DE CASO

4.1 CARACTERIZAÇÃO DO SISTEMA A SER TESTADO

Como exemplo prático, foi selecionado uma aplicação simples que executa cadastros de contratos de créditos em uma instituição financeira. O contrato de crédito, bem como o crédito rotativo, são instrumentos de crédito de um banco, que por sua vez junto com o instrumento de financiamento e o instrumento de captação são considerados instrumentos financeiros.

O contrato é composto por uma ou mais parcelas, por um identificador único, por um estado indicando se o contrato está em dia, vencido ou concluído e um valor que corresponde a somatória dos valores de suas parcelas. O contrato poderá ser do tipo crédito consignado, um financiamento de veículo ou um crédito direto ao consumidor. As parcelas possuem também um identificador único, um valor que não poderá ser igual ou inferior a zero e um estado indicando a situação de pagamento da mesma.

Enquanto a parcela não for paga e sua data de vencimento ainda não foi atingida, esta assume o estado de “aberta”. Caso uma parcela seja paga, assume o status de “liquidada”. Se a parcela não for paga até a sua data de pagamento, passa ao estado “vencida”. Se todas as parcelas forem liquidadas, o contrato passa ao estado de “liquidado”. Se existir alguma parcela vencida, o contrato passa ao status de “vencido”; se existir ao menos uma parcela aberta e nenhuma parcela vencida, o contrato fica com o status de “aberto”.

Para simplificação do estudo de caso, são considerados apenas contratos de crédito contendo no máximo duas parcelas com valores “float”, a adição de mais parcelas ao contrato não afetaria o funcionamento do método, apenas aumentaria a quantidade de casos de testes extraída mostrando as mesmas funcionalidades e resultados. Não foi considerado as modalidades do contrato (financiamento de veículo, crédito consignado,...) e nem as regras que envolvem os campos contendo informações sobre o contrato e as parcelas que serão tratadas de forma superficial, diminuem a complexidade de níveis das Redes de Petri e das validações no OWL não afetando a veracidade dos métodos, apenas diminui a quantidade de informações apresentadas de forma a facilitar o entendimento.

Para a elaboração da rede de Petri foi utilizado o programa GHENeSys versão 1.0, a extração dos casos de testes da rede é realizada manualmente seguindo a metodologia apresentada. Como nesse estudo de caso não houve a execução dos casos de testes (simplificação), os supostos dados armazenados no banco é simulado com arquivos de textos que possuem informações sobre os contratos e parcelas inseridos na base de dados, o modelo ontológico foi construído utilizando o software Protege versão 4.3 cuja versão consegue inferir regras criadas utilizando o SWRL e a rotina utilizada para carregar as instâncias (contratos e parcelas) no modelo ontológico foi construída em Java utilizando a biblioteca OWL API versão 3.4, essa biblioteca livre permite a codificação dos axiomas, classes, propriedades e instâncias, permitindo a inserção de informações da base de dados direto no modelo em OWL.

Devido ao fato de um “*reasoner*” não atender todas as necessidades desse método optou-se pelo uso de dois em conjunto no Protege, o primeiro é o FaCT++ que fornece explicações mais completas sobre erros na inferência facilitando a rastreabilidade de eventuais defeitos além de permitir o uso de regras de comparação do tipo maior e menor direto no OWL, porém ele não consegue interpretar regras do SWRL, o segundo foi o Pellet que é um pouco mais simples porém consegue interpretar essas regras. Após a execução do “*reasoner*” FaCT++ para os erros estruturais é necessário avaliar as causas do erros apontados, direcioná-los aos devidos tratamentos, elimina-los consertando os indivíduos que os causaram para então executar o Pellet para a segunda avaliação.

Essa abordagem necessita de um cuidado, ao trocar o “*reasoner*”, não pode esquecer de eliminar as regras de comparação que estão na OWL como a representada na Figura 8, pois do contrário o “*reasoner*” Pellet irá apontar erros estruturais que não existem.

<p>Classe de Equivalência: (hasStatusBase some StatusBase) and (hasStatusParcela some StatusPartition) and (hasValor exactly 1 float[> 0.0])</p> <p>Subclasse: hasStatusParcela only (Vencido or Aberto or Liquidado) hasStatusBase only (Antes or Depois)</p> <p>(a)</p>	<p>Classe de Equivalência: (hasStatusBase some StatusBase) and (hasStatusParcela some StatusPartition) and (hasValor exactly 1 float)</p> <p>Subclasse: hasStatusParcela only (Vencido or Aberto or Liquidado) hasStatusBase only (Antes or Depois)</p> <p>(b)</p>
--	--

Figura 8 - Exemplo de mudança no axioma ao trocar o “reasoner”. (a) Para o “reasoner” FaCT++. (b) Para o “reasoner” Pellet

O fluxo de negócio da aplicação pode ser definido da seguinte forma: o usuário deve primeiro cadastrar um contrato preenchendo vários campos com informações genéricas do contrato, como por exemplo, data de vencimento do contrato, valor total do contrato, modalidade de contrato, taxa de juros mensal, entre outros.

Em seguida o usuário deve criar as parcelas intermediárias da operação indicando informações adicionais para cada parcela criada como, por exemplo, a forma de pagamento da parcela, o valor da parcela, a data de vencimento da parcela, entre outras.

Todas as parcelas devem ser criadas manualmente com exceção da última parcela, que será criada automaticamente ao incluir o contrato que deverá ter a data de vencimento do contrato e o valor igual ao valor do contrato menos os valores das demais parcelas criadas manualmente. Nesse momento de inclusão, todas as parcelas e o contrato assumem o status de “aberta”/“aberto”.

As parcelas podem ser removidas do contrato, bem como o próprio contrato pode ser cancelado a qualquer momento antes de sua inclusão.

Como a escolha dos casos de testes, bem como a execução da bateria de teste, não fazem parte do escopo deste trabalho, essas etapas não estão aqui demonstradas.

4.2 REPRESENTAÇÃO DINÂMICA DO SISTEMA

Aplicando-se o primeiro passo do método, foi possível identificar duas entidades (parcela e contrato) e dividir o sistema em dois níveis: o primeiro nível (mais alto) controla as interações entre os componentes demonstrando as ações que levam a mudança de estado desses componentes enquanto que o segundo nível controla os status de cada componente apresentando como o componente se comporta (as entidades).

No passo seguinte do método que consiste em criar os modelos, constatou-se que eram necessário 4 modelos de rede para representar o sistema divididos da seguinte maneira: um modelo para nível de interação entre os componentes (1º Nível), dois modelos para representar a entidade parcela (2º Nível, um para cada parcela) e um modelo para representar a entidade contrato (2º Nível).

A Figura 9 representa o modelo macro onde as transições representam as ações que o usuário aplica no sistema ao modificar os *status* das parcelas e do contrato ou representam ações temporais que também alteram a situação dos submódulos (passagem do tempo que causa o vencimento de parcelas) e locais dos status generalizados do sistema. Os lugares “Contrato”, “Par1” e “Ult_Par” representam as entidades e correspondem os locais onde suas sub-redes interagem com a rede macro, cada um representando uma entidade do sistema (Contrato, 1ª Parcela e Última Parcela respectivamente).

Figura 9 - 1º Nível - Rede de Petri Macro da aplicação

Nessa figura também é possível identificar o lugar de início do sistema representado como “Início” que serve de pré-condição para a transição “Criar_Contrato” e os possíveis lugares que finalizam o processo representados por “Contr_Canc” e “Fim” que representam o fim após o cancelamento do contrato e fim após o pagamento da última parcela do contrato respectivamente.

As transições “Concluir_Contr” e “Incluir_1Par” são as transições responsáveis pelo nascimento das entidades “Ult_Par” e “Par1”, ou seja, essas entidades só passam a existir após a ativação dessas transições. As transições “Adiar”, “Vencer” e “Pagar” são exemplos de transições que são afetadas por lugares presentes na entidade “Ult_Par” e geram consequências para a entidade “Contrato”, ou seja, existem lugares na entidade contrato que são influenciados por pré-condições presentes na entidade “Ult_Par”.

Ainda avaliando a Figura 9 é possível verificar o relacionamento onde duas entidades morrem simultaneamente, as entidades “Par1” e “Contrato” concluem seus ciclos de vidas juntas quando a transição Cancelar_2 é disparada. Por questão de simplificação não foi representado o relacionamento onde o disparo das transições externas “Pagar” e “Pagar_venc” dependem que a entidade “Par1” tenha atingido seu lugar terminal “Par1_liquidada” mostrado na Figura 10.

O relacionamento onde duas ou mais entidades são afetadas simultaneamente por uma transição externa é apresentado na Figura 10 através da transição “Vencer1” afetando as entidades “Contrato” e “Par1”.

As Figuras 10 e 11 representam as parcelas, indicam todos os estados possíveis da parcela, suas regras de avanço e as regras de relacionamento com ações externas.

Figura 10 - 2º Nível - Rede de Petri representando a primeira parcela

Figura 11 - 2º Nível - Rede de Petri representando a última parcela

Por último, a Figura 12 representa os estados do contrato, e assim como nas parcelas, demonstra todos seus estados possíveis e as regras de transição, além de indicar alguns locais das parcelas cujo estado influencia no processo interno do contrato, mais especificamente, o lugar que representa a ultima parcela no estado de aberta (Ult_Par_aberta) e que desabilita as transições de adiantamento da data de pagamento (Adiar1__) e pagamento de parcela vencida (Pagar_venc1__) e o lugar que representa a primeira parcela vencida (Par1_Venc) e que habilita a passagem do contrato de aberto para vencido (Vencer__).

Após a construção do modelo, este deve ser sujeito a uma validação frente a documentação do projeto, verificando se todos os requisitos e casos de usos foram abrangidos pela representação.

Figura 12 - 2º Nível - Rede de Petri representando o Contrato

4.3 REPRESENTAÇÃO DO NEGÓCIO ATRAVÉS DE ONTOLOGIAS

Aplicando-se o primeiro passo do método do validador, criou-se o modelo apresentado na Figura 13 que destaca as relações entre as entidades do sistema. Todas as modalidades de instrumento financeiro foram representadas por classes, bem como as parcelas. Uma categoria de classe do tipo partição (“*partitions*”) foi criada para representar os estados de cada indivíduo, a subclasse “*StatusPartition*” traz informações sobre o estado das parcelas (“Vencido”, “Liquidado” e “Aberto”) enquanto a subclasse “*StatusBase*” traz informações do momento em o indivíduo existiu (“Antes” ou “Depois” da execução). Essa classe é representada com detalhe na Figura 14.

A classe “Erro” foi criada disjunta de todas as demais classes para as validações de erros estruturais, contendo três subclasses: “*AssociacaoClasselNexistente*” (para indivíduos pertencentes a uma classe não declara no modelo), “*AssociacaoIndividuoInexistente*” (para indivíduos que se relacionam com indivíduo não existentes no modelo) e “*Nomezerado*” (para indivíduos sem nome). Representada com detalhe na Figura 14.

Figura 13 - Modelo ontológico das classes da aplicação com destaque para as regras de negócio.

Figura 14 - Modelo ontológico das classes da aplicação com destaque para as classes “Erro” e “Partitions”.

A classe “CasosdeTestes”, expandida na Figura 15, foi criada como repositório dos casos de testes, contendo 5 classes de casos de testes padrão (“CTXXX”) e 4 classes de casos de testes reversos (“CTXXX_Reverso”), os demais casos de testes foram omitidos para simplificar o estudo de caso. O caso de teste “CT003” valida se os contratos criados de fato possuem ao menos uma parcela conforme demonstrado na Figura 16. Esse caso de teste não necessita de um reverso pois a lógica no modelo não permite a existência de um contrato sem parcelas sem gerar um erro estrutural.

Os casos de testes “CT005”, “CT021” e “CT020” avaliam se um determinado contrato está aberto, liquidado e vencido respectivamente conforme apresentado na Figura 17 enquanto que seus reversos avaliam se um contrato não está no estado definido para seu teste original (Figura 18).

Figura 15 - Modelo ontológico das classes da aplicação com destaque para a classe “CasosdeTestes”.

Figura 16 - Axioma da classe CT003

Classe de Equivalência: hasStatusContrato some Aberto (a)	Classe de Equivalência: hasStatusContrato some Liquidado (b)	Classe de Equivalência: hasStatusContrato some Vencido (c)
--	---	---

Figura 17 - Axioma das classes de casos de testes. (a) Classe “CT005”. (b) Classe “CT021”. (c) Classe “CT020”

Classe de Equivalência: not (hasStatusContrato some Aberto) (a)	Classe de Equivalência: not (hasStatusContrato some Liquidado) (b)	Classe de Equivalência: not (hasStatusContrato some Vencido) (c)
---	--	--

Figura 18 - Axioma das classes de casos de testes reversas. (a) Classe “CT005_Reverso”. (b) Classe “CT021_Reverso”. (c) Classe “CT020_Reverso”

O caso de teste “CT022” e o seu reverso “CT022_Reverso” foram definidos utilizando axiomas do SWRL conforme Figura 19.

A primeira regra avalia se o valor de um contrato com uma parcela é igual ao valor da sua parcela enquanto a segunda regra avalia se a somatória dos valores das parcelas é igual ao valor do contrato. A terceira regra avalia se o contrato de uma parcela possui o valor diferente do que o valor de sua parcela e em caso positivo o classifica como individuo do “CT022_Reverso”, a quarta avalia que se o contrato tiver o seu valor menor que o de uma parcela sua ele já é considerado um reverso e a última regra avalia se somatória dos valores de parcelas de um contrato com duas parcelas é diferente do valor do contrato.

Regras: 1-) ContratoCredito(?c), Parcela(?p), hasParcela(?c, ?p), hasValor(?p, ?vp1), hasValorContrato(?c, ?vc), lessThan (?vc, ?vp1) -> CT022_Reverso (?c) 2-) ContratoCredito(?c), Parcela(?p1), Parcela(?p2), hasParcela(?c, ?p1), hasParcela(?c, ?p2), hasValor(?p1, ?vp1), hasValor(?p2, ?vp2), hasValorContrato(?c, ?vc), add (?soma, ?vp1, ?vp2), notEqual (?vc, ?soma), DifferentFrom (?p1, ?p2) -> CT022_Reverso (?c) 3-) ContratoCredito(?c), Parcela(?p1), Parcela(?p2), hasParcela(?c, ?p1), hasParcela(?c, ?p2), hasValor(?p1, ?vp1), hasValor(?p2, ?vp2), hasValorContrato(?c, ?vc), add (?soma, ?vp1, ?vp2), equal (?vc, ?soma), DifferentFrom (?p1, ?p2) -> CT022 (?c) 4-) ContratoCredito(?c), Parcela(?p), hasParcela(?c, ?p), hasNumeroParcela(?c, 1), hasValor(?p, ?vp1), hasValorContrato(?c, ?vc), notEqual (?vc, ?vp1) -> CT022_Reverso (?c) 5-) ContratoCredito(?c), Parcela(?p), hasParcela(?c, ?p), hasValor(?p, ?vp1), hasValorContrato(?c, ?vc), equal (?vc, ?vp1) -> CT022 (?c)
--

Figura 19 - Regras em SWRL para as classes “CT022” e “CT022_Reverso”

Foi identificada a necessidade de utilizar cinco “*built-ins*” diferentes para executar as operações de soma e comparação entre as entidades e os valores. O “*built-in*” *lessThan* verifica se o primeiro valor é inferior ao segundo, o “*built-in*” *add* soma os dois últimos parâmetros e associa o valor ao terceiro. O “*built-in*” *equal* verifica se os dois valores são iguais, enquanto que o *notEqual* faz a comparação contrária, por último tem-se o *DifferentFrom* que compara duas entidades verificando se são diferentes. As demais premissas fazem associação a classes e propriedades que serão explicadas mais adiante.

A classe “ContratoCredito” é definida com as expressões presentes na Figura 20, indicando as restrições de quantidade de parcela e de estados, já as expressões da classe “Parcela” são apresentadas na Figura 21 demonstrando as restrições de valor e de estados.

```

Classe de Equivalência:
(hasStatusBase some StatusBase) and
(hasStatusContrato some StatusPartition)
and (hasParcela min 1 Parcela)
Subclasse:
(((hasParcela some (Parcela
  and (hasStatusParcela some Aberto)))
and (hasParcela only (Parcela
  and (hasStatusParcela only (
not (Vencido))))))
and (hasStatusContrato some Aberto))
or ((hasParcela some (Parcela
  and (hasStatusParcela some Vencido)))
and (hasStatusContrato some Vencido))
or ((hasStatusContrato some Liquidado)
and (hasParcela only (Parcela
  and (hasStatusParcela only
Liquidado))))
hasStatusBase only (Antes
or Depois)
hasParcela only Parcela
hasParcela some Parcela
hasStatusContrato only (Aberto or
Liquidado or Vencido)

```

Figura 20 - Expressões da classe ContratoCredito

```

Classe de Equivalência:
(hasStatusBase some StatusBase) and
(hasStatusParcela some StatusPartition)
and (hasValor exactly 1 float[> 0.0])
isParcelaOf some ContratoCredito
Subclasse:
hasStatusParcela only (Vencido
or Aberto
or Liquidado)
hasStatusBase only (Antes
or Depois)

```

Figura 21 - Axiomas da classe Parcela

Nota-se que a classe “Parcela” possui um axioma de comparação “*hasValor exactly 1 float[>0.0]*”, esse axioma deve ser editado ao se trocar o “*reasoner*”, para o FaCT++ não há necessidade de alterá-lo, já para o Pellet precisa eliminar a parte “[>0.0]”.

Com relação a propriedades de dados foram definidos sete: *hasIDContrato* (*string*), *hasIDParcela* (*string*), *hasValor* (*float*), *hasValorContrato* (*float*), *hasNumeroParcela* (*int*), *hasNomeContrato* (*string*) e *hasNomeParcela* (*string*). Os dois primeiros trazem informações com relação ao identificador dos indivíduos, enquanto o *hasValor* traz informação do valor da parcela e o *hasValorContrato* o valor do contrato. *hasNumeroParcela* traz a informação de quantas parcela o contrato possui e os dois últimos trazem informações sobre o nome do individuo. Todas as setes são propriedades de dados funcionais, ou seja, cada individuo possui no máximo uma propriedade de cada tipo.

Também foram definidas cinco propriedades de objetos: *hasParcela*, *isParcelaOf*, *hasStatusContrato*, *hasStatusParcela* e *hasStatusBase*. *hasParcela* é uma propriedade assimétrica (se individuo se relaciona através dessa propriedade com outro individuo, então esse último individuo não pode se relacionar com o primeiro através dessa mesma propriedade) e inflexível (não pode ser utilizada para se autorelacionar) utilizada para relacionar o contrato com suas parcelas. A propriedade funcional *isParcelaOf* é a inversa da *hasParcela*. As propriedades *hasStatusContrato* e *hasStatusParcela* são propriedades funcionais, assimétricas e inflexíveis que relacionam o individuo (“ContratoCredito” ou “Parcela”) ao seu estado. Por último a propriedade *hasStatusBase* é uma propriedade funcional, assimétrica e inflexível utilizada para relacionar qualquer individuo a uma estado da base (“StatusBase”).

Figura 22 - Axiomas de um indivíduo da classe ContratoCredito

Figura 23 - Expressões de um indivíduo da classe Parcela

Tendo montado o modelo em OWL e após extrair os casos de testes, deve-se selecionar os casos de testes para bateria e executá-la. Após a execução uma rotina deve extrair os dados necessários da base e inseri-los no modelo ontológico. As Figuras 22 e 23 mostram os axiomas de indivíduos do tipo contrato e do tipo parcela inseridos no modelo.

4.4 EXTRAÇÃO DOS CASOS DE TESTES

Após a validação dos modelos, deve-se escolher uma cobertura de teste para a extração dos casos de testes. É aconselhado utilizar no mínimo a cobertura por transição pelo fato de cobrir todos os fluxos possíveis dentro do modelo. Para efeito de demonstração foi utilizado a cobertura por estado por ser a mais simples.

Modelo de controle da operação (Figura 9)							
CT	L1	T1	L2	T2	L3	T3	L4
001	Inicio	Criar_Contrato	Per_Criar_1Par				
002	Inicio	Criar_Contrato	Contrato				
003			Per_Criar_1Par	Concluir_Contr	Ult_Par		
004					Ult_Par	Pagar_venc	Fim
005			Per_Criar_1Par	Incluir_1Par	Contr_1Par		
006					Contr_1Par	Cancelar_2	Contr_Canc
007					Contr_1Par	Concluir_Contr_2	Par1

Tabela 4 - Tabela de casos de testes obtidos no 1º Nível

Extraíndo os casos de testes do primeiro modelo e utilizando a cobertura por estado obtemos a Tabela 4. É esperado que os casos de testes do primeiro nível sejam mais abrangentes o que os tornam ideais para a execução de *smoke tests* (bateria de testes iniciais em um sistema que tem por objetivo verificar a existência de defeitos impeditivos).

Avaliando os casos de testes obtidos desse nível, visualiza-se que eles são relativamente simples, por exemplo, o CT005 representa a inclusão de uma parcela para a operação, do ponto de vista gráfico, o lugar “Per_Criar_1Par” representa o estado do sistema pronto para receber novas parcelas, o lugar “Contr_1Par” representa o contrato já com uma parcela criada e a transição “Incluir_1Par” a ação do usuário em incluir a parcela.

Os casos de testes do segundo nível são apresentados nas Tabelas 5 e 6 (para a primeira e última parcela respectivamente) e 7 (para o contrato). Esses casos de testes validam o ciclo de vida das entidades, testando sua evolução. Nesse nível, podem-se destacar os casos de testes CT009 que consiste no cancelamento do contrato, o CT021 que representa o pagamento da primeira parcela e o CT023 que representa o vencimento da última parcela.

Os casos de testes CT011 e 012 na Tabela 7 exploram as possibilidades diferentes do lugar “Par1_Venc”, enquanto o CT011 é um teste positivo verificando se contrato realmente passa do estado “Aberto” para “Vencido” quando a primeira parcela vence, o CT012 é um teste negativo verificando se o contrato não muda de estado se a parcela não mudar para o estado de vencida. Os casos de testes CT022 e CT025 representam as duas formas de iniciar o sub-módulo “Ult_Par”, o primeiro representa sua criação quando o contrato só possui uma parcela enquanto que o segundo representa sua criação quando o contrato possui 2 parcelas.

Modelo de controle da Primeira Parcela (Figura 10)						
CT	T1	L1	T2	L2	T3	L3
017	Incluir_1Par_	Par1_criada				
018		Par1_criada	Cancelar_2_	Cancelado1		
019		Par1_criada	Concluir_Contr_2_	Par1_aberta		
020				Par1_aberta	Vencer1_	Par1_vencida
021				Par1_aberta	Pagar1_	Par1_liquidada

Tabela 5 - Tabela de casos de testes obtidos da Rede de Petri representando a primeira parcela.

Modelo de Controle da Última Parcela (Figura 11)				
CT	T1	L1	T2	L2
022	Concluir_Contr_	Ult_Par_aberta		
023		Ult_Par_aberta	Vencer_	Ult_par_vencida
024		Ult_Par_aberta	Pagar_	Ult_par_liquidada
025	Concluir_Contr_2_	Ult_Par_aberta		

Tabela 6 - Tabela de casos de testes obtidos da Rede de Petri representando a última parcela.

Modelo de controle do Contrato (Figura 12)						
CT	T1	L1	T2	L2	T3	L3
008	Criar_Contrato_	Contr_aberto				
009		Contr_aberto	cancelar_	Contr_cancelado		
010		Contr_aberto	Pagar_	Contr_liquidado		
011		Contr_aberto	Vencer__(Hab.)	Contr_vencido		
012		Contr_aberto	Vencer__(Des.)	Contr_aberto		
013				Contr_vencido	Pagar_venc1__(Hab.)	Contr_aberto
014				Contr_vencido	Pagar_venc1__(Des.)	Contr_vencido
015				Contr_vencido	Adiar1__(Hab.)	Contr_aberto
016				Contr_vencido	Adiar1__(Des.)	Contr_vencido

Tabela 7 - Tabela de casos de testes obtidos da Rede de Petri representando o contrato.

Como última tarefa, o extrator precisa verificar a existência de casos de testes duplicados e de casos de testes incoerentes (como, por exemplo, teste de estruturas internas do software onde não há como validar). Neste estudo de caso não é possível identificar casos de testes incoerentes, no entanto, é possível notar alguns casos de testes duplicados que podem ser eliminados da bateria de execução, por exemplo, o CT005 e o CT017 que podem ser validados em um único caso de teste.

Nota-se nessa extração que não há detalhamento de como o software opera dentro dos sub-modelos (CT003) ou como as condições de um estado é atingida (CT011), esses cuidados devem ser avaliados no momento da elaboração do passo-a-passo através da análise mais aprofundada do próprio modelo, uma vez que o

método apenas indica o que deverá ser testado. Outro ponto de atenção que deve ser considerado na execução é que alguns casos de testes dependem da execução de outros como, por exemplo, o CT023 que depende do CT022, esse fato sugere a criação de plano de execução de teste.

4.5 VERIFICAÇÃO BASEADA EM ONTOLOGIA

Para o exemplo, foi mapeada a massa apenas para alguns casos de testes para efeitos de demonstração conforme apresentado na Tabela 8. Assume-se que não seja possível indicar o número do caso de teste no nome da entidade.

Tipo da Identidade	Parâmetros de Identificação	Valores dos parâmetros	Criada durante a execução	CT
Contrato	Nome	C1	Sim	003/022
Parcela	Nome	Par1_C1	Sim	003/022
Contrato	Nome	Cerrado6	Sim	003
Parcela	Nome	Par1_Cerrado6	Sim	003
Contrato	Nome	Cerrado31	Sim	005
Parcela	Nome	Par1_Cerrado31	Sim	005
Contrato	Nome	Cerrado40	Sim	007
Parcela	Nome	Par1_Cerrado40	Sim	007
Parcela	Nome	(Sem Parcela)	Sim	007
Contrato	Nome	Cerrado20	Sim	008
Parcela	Nome	Par1_Cerrado20	Sim	008
Contrato	Nome	Cerrado27	Sim	008
Parcela	Nome	Par1_Cerrado27	Sim	008
Contrato	Nome	Cerrado16	Sim	010
Parcela	Nome	Par1_Cerrado16	Sim	010
Contrato	Nome	Cerrado21	Sim	017
Parcela	Nome	(Sem Nome)	Sim	017
Contrato	Nome	Cerrado2	Sim	019
Parcela	Nome	Par1_Cerrado2	Sim	019
Contrato	Nome	Cerrado4	Sim	019
Parcela	Nome	Par1_Cerrado4	Sim	019
Contrato	Nome	Cerrado25	Sim	019
Parcela	Nome	Par1_Cerrado25	Sim	019
Contrato	Nome	Cerrado33	Sim	020
Parcela	Nome	Par1_Cerrado33	Sim	020

Parcela	Nome	Par2_Cerrado33	Sim	020
Contrato	Nome	Cerrado3	Sim	022
Parcela	Nome	Par1_Cerrado3	Sim	022
Contrato	Nome	(Sem Nome)	Sim	022
Parcela	Nome	Par1_Cerrado12	Sim	022
Contrato	Nome	Cerrado36	Sim	022
Parcela	Nome	Par1_Cerrado36	Sim	022
Contrato	Nome	Cerrado38	Sim	022
Parcela	Nome	Par1_Cerrado38	Sim	022
Parcela	Nome	Par2_Cerrado38	Sim	022
Contrato	Nome	Cerrado9	Não	023
Parcela	Nome	Par1_Cerrado9	Não	023
Contrato	Nome	C10	Não	024
Parcela	Nome	Par1_C10	Não	024

Tabela 8 - Mapeamento de massa para o estudo de caso

Após realizar todas as preparações, finalmente, aciona-se o primeiro mecanismo de inferência do Protege para verificar a consistência da base após a adição dos indivíduos ao modelo (FaCT++). Para o primeiro arquivo de texto (dados.txt) que possui apenas entidades que obedecem todas as regras, o mecanismo de inferência não apresentou nenhuma inconsistência conforme o esperado, todos os contratos e parcelas foram inseridos com sucesso no modelo. Em seguida elimina-se a regra de comparação da classe “Parcela” e aciona-se o segundo mecanismo de inferência (Pellet) que também não apresentou inconsistência.

A Figura 24 apresenta possíveis situações onde ha ausência de informações nas propriedades ou com a propriedade zerada.

No caso (a), o método deve acusar erro ao executar o caso de teste “CT022” (avaliado pelo Pellet) verificando que a soma dos valores das parcelas é diferente que a soma do contrato, ou seja, individuo não será classificado como um indivíduo da classe “CT022” e será classificado com um individuo da classe “CT022_Reverso” conforme comparação entre as figuras 25 e 26.

Contrato	Cerrado3	Contrato	Cerrado16	Contrato	Cerrado20
Estado	Aberto	Estado		Estado	Vencido
NParcelas	1	NParcelas	1	NParcelas	1
IDContrato	1008	IDContrato	1015		
ValorContrato	0	ValorContrato	350.00	ValorContrato	35.12
Parcela	Par1_Cerrado3	Parcela	Par1_Cerrado16	Parcela	Par1_Cerrado20
Estado	Aberto	Estado	Aberto	Estado	Vencido
IDParcela	10080001	IDParcela	10150001	IDParcela	10190001
ValorParcela	200000.00	ValorParcela	350.00	ValorParcela	35.12
isParcelaOf	Cerrado3	isParcelaOf	Cerrado16	isParcelaOf	Cerrado20
(a)		(b)		(c)	

Figura 24 - Representação de contratos sem uma das propriedades. (a) Contrato com valor zero. (b) Contrato sem estado. (c) Contrato sem ID

Figura 25 - Classe CT022 sem o Contrato Cerrado3

Figura 26 - Classe CT022_Reverso com o Contrato Cerrado3

Pelas regras de negócio um contrato deve possuir um estado e este deve ser um dos três estados possíveis: “Aberto”, “Vencido” ou “Liquidado”. Portanto ao executar um caso de teste que tenha alteração no status do contrato (por exemplo, o CT010), caso o sistema não esteja corretamente implementado a situação da Figura 24 caso (b) poderá ocorrer. Nesse caso, ao ser carregado o contrato no modelo ontológico e ativado o “Reasoner” FaCT++ a inconsistência da Figura 27 será apontada. Nessa inconsistência, o programa não encontrou um estado (vazio) possível dentro das subclasses de estados disponíveis e associou o erro estrutural de associação de classe inexistente que por sua vez possui a regra de permiti que um individuo pertença a esta classe e ao mesmo tempo seja um “ContratoCredito”. Uma validação para verificar se o caso de teste CT010 foi executado corretamente seria verificar se o contrato especificado no mapa de massas aparece como membro da classe “CT021”, essa classe por verificar se o contrato esta liquidado pode ser usada tanto nos testes do CT010 como nos testes CT021.

Para o caso (c) da Figura 24, nenhum dos “reasoner” apontou erros na execução do caso de teste CT008, pois pela simplificação do exemplo não há regra implementada que restrinja a propriedade de dados *hasIDContrato*, em outras palavras, essa propriedade pode assumir qualquer valor, ou até mesmo não assumir valor algum. No modelo esse contrato será introduzido conforme a Figura 28.

A Figura 29 demonstra dois exemplos de contratos cujo valor do contrato é diferente da soma de suas parcelas. O caso (a) mostra um contrato com apenas uma parcela e com valor do contrato superior ao dessa parcela enquanto que o caso (b) mostra um contrato com duas parcelas com valor inferior ao somatório. Para ambos os casos o “*reasoner*” Pellet os classificaram como indivíduos da classe “CT022_Reverso” conforme as Figuras 30 e 31.

- Explanation for: Thing SubClassOf Nothing
- 1) Cerrado16_Novo Type ContratoCredito
 - 2) Erro DisjointWith Partitions
 - 3) StatusPartition SubClassOf Partitions
 - 4) Cerrado16_Novo Type hasStatusContrato only AssociacaoClasseInexistente
 - 5) AssociacaoClasseInexistente SubClassOf Erro
 - 6) ContratoCredito SubClassOf hasStatusContrato some StatusPartition

Figura 27 - Explicação da inconsistência da associação de um estado inexistente

Cerrado20_Novo	
Tipo:	
ContratoCredito	
and (hasStatusBase only Depois)	
and (hasStatusContrato only Vencido)	
Associações de Propriedades de Dados:	
hasNumero Parcela 1	
hasIDContrato ""^string	
hasValorContrato 35.12f	
hasNomeContrato "Cerrado20"^string	

Figura 28 - Cerrado20 conforme é apresentado no modelo

Contrato	Cerrado36	Contrato	Cerrado38	
Estado	Vencido	Estado	Aberto	
NParcelas	1	NParcelas	2	
IDContrato	1035	IDContrato	1037	
ValorContrato	800.00	ValorContrato	1802.00	
Parcela	Par1_Cerrado36	Parcela	Par1_Cerrado38	Par2_Cerrado38
Estado	Vencido	Estado	Aberto	Aberto
IDParcela	10350001	IDParcela	10370001	10370002
ValorParcela	400.00	ValorParcela	8400.00	8400.00
isParcelaOf	Cerrado36	isParcelaOf	Cerrado38	Cerrado38
(a)		(b)		

Figura 29 - Exemplos de contratos com valor errado. (a) Contrato com uma parcela. (b) Contrato com duas parcelas

Figura 30 - Cerrado36 classificado como um individuo do “CT022_Reverso”

Figura 31 - Cerrado38 classificado como um individuo da classe “CT022_Reverso”

Para efeito de demonstração o CT022 foi executado mais de uma vez com massas diferentes para que fosse possível exemplificar todos os tipos de erros passíveis de ocorrer. A princípio, na cobertura de testes adotada no estudo de caso ele só seria executado uma vez e apenas um dos erros seria identificado (dependendo da massa adotada), no entanto, o outro erro seria pego na execução de outros casos de testes, pois na classe “CT022_Reverso” da OWL seriam apresentados todos os Contratos com erro no cálculo não apenas aquele utilizado para o teste em si. Caso o resultado do teste fosse positivo o contrato seria apresentado dentro da classe “CT022” como é o caso do ContratoCredito C1 apresentado na Figura 32 e 33.

Figura 32 - Contrato C1 classificado como instância da classe “CT022”

Contrato	C1
Estado	Aberto
NParcelas	1
IDContrato	0020
ValorContrato	10000.00
Parcela	Par1_C1
Estado	Aberto
IDParcela	00200001
ValorParcela	10000.00
isParcelaOf	C1

Figura 33 - Detalhe do contrato C1

- 1) Nomezerado **SubClassOf** Erro
- 2) ContratoCredito **SubClassOf** InstrumentoCredito
- 3) Erro **DisjointWith** InstrumentoFinanceiro
- 4) _Novo **Type** Nomezerado
- 5) _Novo **Type** ContratoCredito **and** (hasStatusBase **only** Depois) **and** (hasStatusContrato **only** Aberto) **and** (hasParcela **exactly** 1 Parcela)
- 6) InstrumentoCredito **SubClassOf** InstrumentoFinanceiro

Figura 34 - Erro de inferência para um ContratoCredito sem nome

Na Figura 34 é mostrado um exemplo de contrato inserido no modelo sem nome. Nessa figura é possível enxergar que o indivíduo “_Novo” resultado de uma terceira tentativa de execução do CT022, que não possui um nome, é associado a classe de erro “Nomezerado” gerando o erro de inferência uma vez que para o modelo ser valido nenhum individuo deve compartilhar dessa classe.

De forma semelhante, o modelo também é capaz de apontar através do mesmo erro de inferência uma Parcela sem nome “0_Novo” ao executar o CT017, conforme apresentado na Figura 35. Caso o a execução deste teste tivesse ocorrida sem erros, o individuo da classe “Parcela” teria sido criado com um nome válido e

poderia ser valido utilizando uma classe parecida com a “CT003” porém para parcelas.

As Figuras 36 e 37 apresentam erros de inferência para contratos com quantidade de parcelas erradas. A primeira figura mostra um contrato que foi inserido sem parcelas indicado pelo axioma “hasParcela max 0 Parcela”, como a regra diz que o “ContratoCredito” deve possuir alguma “Parcela”, o Protege aponta o erro, o que seria validado pela classe “CT003”. Na segunda figura, a segunda parcela não foi declarada, nessa situação o programa fez uma associação do contrato com um individuo da classe de erro “AssociacaoIndividuoInexistente” causando o erro na inferência na execução do caso de testes CT007. Já a Figura 38 demonstra um exemplo de uma execução OK do “CT003”, onde é possível vê o contrato “C1_Novo” classificado como membro da classe “CT003”.

- 1) Nomezerado **SubClassOf** Erro
- 2) 0_Novo **Type** Nomezerado
- 3) Erro **DisjointWith** Parcela
- 4) 0_Novo **Type** Parcela

Figura 35 - Erro para uma Parcela sem nome

- 1) Cerrado6_Novo **Type** hasParcela **max** 0 Parcela
- 2) ContratoCredito **SubClassOf** hasParcela **some** Parcela
- 3) Cerrado6_Novo **Type** ContratoCredito

Figura 36 - Inconsistência da existência de um contrato sem parcelas

- 1) IndInexistente_0_Cerrado40_Novo **Type** Parcela
- 2) Erro **DisjointWith** Partitions
- 3) StatusPartition **SubClassOf** Partitions
- 4) IndInexistente_0_Cerrado40_Novo **Type** hasStatusParcela **only** AssociacaoIndividuoInexistente
- 5) AssociacaoIndividuoInexistente **SubClassOf** Erro
- 6) Parcela **SubClassOf** hasStatusParcela **some** StatusPartition

Figura 37 - Inconsistência de um contrato sem a declaração de uma das Parcelas

Figura 38 - Exemplo positivo de execução do caso de teste 003

Contrato	Cerrado33
Estado	Aberto
NParcelas	2
IDContrato	1032
ValorContrato	250.00
Parcela	Par1_Cerrado33
Estado	Vencido
IDParcela	10320001
ValorParcela	125.00
isParcelaOf	Cerrado33
Parcela	Par2_Cerrado33
Estado	Aberto
IDParcela	10320002
ValorParcela	125.00
isParcelaOf	Cerrado33

Figura 39 - Contrato com estado incorreto

A figura acima (Figura 39) mostra o exemplo de um contrato que teve seu estado classificado de forma errada, pela regra de negócio o contrato deveria ter seu estado igual a “Vencido” por possuir uma parcela vencida, no entanto, seu estado é “Aberto” (CT020). Esse erro apresentou a inconsistência mostrada na Figura 40, onde é possível observar na linha 4 a regra para os estados do contrato, especificamente no trecho “...or ((hasParcela some (Parcela and (hasStatusParcela some Vencido))) and (hasStatusContrato some Vencido))...” nas linhas 5 e 6 que diz que se o contrato tiver uma parcela no estado vencido, esse contrato só poderá assumir o estado vencido.

Para as parcelas, o comportamento do modelo quando um dos parâmetros está vazio é semelhante aos dos contratos de créditos já exemplificados. O erro apresentado na Figura 41 na execução do CT019 para uma parcela que não possui um estado é semelhante ao demonstrado na Figura 27, da mesma forma que no contrato, não é apontado um erro quando a parcela não possui o ID, pois essa regra não foi implementada. A Figura 42 mostra um exemplo inserido no OWL semelhante ao que ocorreu na Figura 28. Tanto na execução do CT020 quanto na execução do CT019 seria possível vê o respectivo contrato mapeado na massa classificado como membro da classe de mesmo nome se o teste resultasse em sucesso.

```
hasParcela InverseOf isParcelaOf
Par1_Cerrado33_Novo isParcelaOf Cerrado33_Novo
Parcela EquivalentTo (hasStatusBase some StatusBase) and (hasStatusParcela some StatusPartition) and (hasValor exactly 1 float[> 0.0])
ContratoCredito SubClassOf (((hasParcela some (Parcela and (hasStatusParcela some Aberto))) and (hasParcela only (Parcela and (hasStatusParcela only (not (Vencido)))))) and (hasStatusContrato some Aberto)) or ((hasParcela some (Parcela and (hasStatusParcela some Vencido))) and (hasStatusContrato some Vencido)) or ((hasStatusContrato some Liquidado) and (hasParcela only (Parcela and (hasStatusParcela only Liquidado))))
DisjointClasses: Aberto, Liquidado, Vencido
Cerrado33_Novo Type ContratoCredito and (hasStatusBase only Depois) and (hasStatusContrato only Aberto) and (hasParcela exactly 2 Parcela)
Par1_Cerrado33_Novo Type Parcela and (hasStatusBase only Depois) and (hasStatusParcela only Vencido)
```

Figura 40 - Inconsistência no estado do contrato

- 1) Par1_Cerrado25_Novo Type hasStatusParcela only AssociacaoClasseInexistente
- 2) Par1_Cerrado25_Novo Type Parcela
- 3) Erro DisjointWith Partitions
- 4) StatusPartition SubClassOf Partitions
- 5) AssociacaoClasseInexistente SubClassOf Erro
- 6) Parcela SubClassOf hasStatusParcela some StatusPartition

Figura 41 - Tentativa de inferência para uma parcela sem um estado associado

Par1_Cerrado27_Novo

Tipo:
Parcela
and (hasStatusBase only Depois)
and (hasStatusParcela only Aberto)

Associações de Propriedades de Objetos:
isParcelaOf Cerrado27_Novo

Associações de Propriedades de Dados:
hasNomeParcela "Par1_Cerrado27"~string
hasValor 305.1f
hasIDParcela ""~string

Figura 42 - Exemplo de parcela sem Identificador

Explanation for: Thing SubClassOf Nothing

- 1) Par1_Cerrado2_Novo hasValor 0.0f
- 2) Parcela SubClassOf hasValor some float[> 0.0]
- 3) Functional: hasValor
- 4) Par1_Cerrado2_Novo Type Parcela

Figura 43 - Incoerência da existência de uma parcela com valor igual a zero

Explanation for: Thing SubClassOf Nothing

- 1) Par1_Cerrado4_Novo hasValor -100.0f
- 2) Parcela SubClassOf hasValor some float[> 0.0]
- 3) Functional: hasValor
- 4) Par1_Cerrado4_Novo Type Parcela

Figura 44 - Contradição na existência de uma parcela com valor negativo

- 1) Par1_Cerrado31_Novo Type Parcela
- 2) Erro DisjointWith Parcela
- 3) Par1_Cerrado31_Novo Type AssociacaoIndividuoInexistente
- 4) AssociacaoIndividuoInexistente SubClassOf Erro

Figura 45 - Erro de uma parcela sem um contrato associado

Para o caso da parcela com valor igual a zero a inconsistência encontrada é demonstrada na Figura 43 também pega na execução do CT019, enquanto que para o caso da parcela com valor negativo a Figura 44 mostra a contradição.

Quando uma parcela não possui um contrato, o modelo reage de forma semelhante a Figura 37 que demonstra um contrato sem a declaração de uma parcela na execução do caso de teste CT005. A Figura 45 demonstra essa situação.

Após a verificação pela inferência deve-se comparar cada individuo no estado “novo” com seu respectivo individuo no estado “antigo” verificando se as alterações foram realizadas com sucesso. A Figura 46 mostra a comparação dos estados do contrato “C10” que estava aberto (C10_Antigo) e teve sua parcela paga indo para o status liquidado (C10_Novo) conforme o caso teste indicava (CT024). Pelo mapeamento de massa concluímos que o CT10 foi executado com sucesso.

C10_Antigo	C10_Novo
Tipo: ContratoCredito and (hasStatusBase only Antes) and (hasStatusContrato only Aberto) and (hasParcela exactly 1 Parcela)	Tipo: ContratoCredito and (hasStatusBase only Depois) and (hasStatusContrato only Liquidado) and (hasParcela exactly 1 Parcela)
Associações de Propriedades de Dados: hasNumeroParcela 1 hasIDContrato "0029"^^string hasValorContrato 10500.0f hasNomeContrato "C10"^^string	Associações de Propriedades de Dados: hasNumeroParcela 1 hasIDContrato "0029"^^string hasValorContrato 10500.0f hasNomeContrato "C10"^^string
Par1_C10_Antigo	Par1_C10_Novo
Tipo: Parcela and (hasStatusBase only Antes) and (hasStatusParcela only Aberto)	Tipo: Parcela and (hasStatusBase only Depois) and (hasStatusParcela only Liquidado)
Associações de Propriedades de Objetos: isParcelaOf C10_Antigo	Associações de Propriedades de Objetos: isParcelaOf C10_Novo
Associações de Propriedades de Dados: hasNomeParcela "Par1_C10"^^string hasValor 10500.0f hasIDParcela "00290001"^^string	Associações de Propriedades de Dados: hasNomeParcela "Par1_C10"^^string hasValor 10500.0f hasIDParcela "00290001"^^string

Figura 46 - Comparação de uma execução com sucesso mostrando os indivíduos antes e depois da execução

Para demonstrar um erro pego através da comparação dos status do sistema foi utilizada a Figura 47. O erro acontece quando o sistema atualiza os status do contrato "Cerrado9_Novo" que teve uma parcela vencida após a execução, o contrato assume o status de "Vencido" com sucesso, mas por algum defeito o valor de sua parcela é alterado indevidamente da mesma forma que o valor do contrato (CT023). Como não há regra de negócio que limite o valor da parcela (desde que ele seja acima de "0" e soma das parcelas seja igual ao valor do contrato) o modelo não aponta o erro.

<p style="text-align: center;">Cerrado9_Antigo</p> <p>Tipo: ContratoCredito and (hasStatusBase only Depois) and (hasStatusContrato only Aberto) and (hasParcela exactly 1 Parcela)</p> <p>Associações de Propriedades de Dados: hasNumeroParcela 1 hasNomeContrato "Cerrado9"^^string hasValorContrato 200000.0f hasIDContrato "1007"^^string</p>	<p style="text-align: center;">Cerrado9_Novo</p> <p>Tipo: ContratoCredito and (hasStatusBase only Depois) and (hasStatusContrato only Vencido) and (hasParcela exactly 1 Parcela)</p> <p>Associações de Propriedades de Dados: hasNumeroParcela 1 hasNomeContrato "Cerrado9"^^string hasValorContrato 100000.0f hasIDContrato "1007"^^string</p>
<p style="text-align: center;">Par1_Cerrado9_Antigo</p> <p>Tipo: Parcela and (hasStatusBase only Depois) and (hasStatusParcela only Aberto)</p> <p>Associações de Propriedades de Objetos: isParcelaOf Cerrado9_Antigo</p> <p>Associações de Propriedades de Dados: hasIDParcela "10070001"^^string hasValor 200000.0f hasNomeParcela "Par1_Cerrado9"^^string</p>	<p style="text-align: center;">Par1_Cerrado9_Novo</p> <p>Tipo: Parcela and (hasStatusBase only Depois) and (hasStatusParcela only Vencido)</p> <p>Associações de Propriedades de Objetos: isParcelaOf Cerrado9_Novo</p> <p>Associações de Propriedades de Dados: hasIDParcela "10070001"^^string hasValor 100000.0f hasNomeParcela "Par1_Cerrado9"^^string</p>

Figura 47 - Comparação de uma execução com erro mostrando os indivíduos antes e depois da execução (o valores não deveriam ser modificados)

5 CONCLUSÃO

O método de extração, quando utilizado a cobertura por transição, consegue abranger todos os caminhos principais e alternativos possíveis do programa diminuindo os riscos de condições não executadas no projeto. Também diminui os riscos que dependem da “inspiração” dos arquitetos de teste para levantar as idéias de testes que serão aplicadas padronizando parcialmente o processo. A técnica facilita a identificação da massa de dados necessária para execução de cada caso de teste e a dependência existente entre eles proporcionando a elaboração do plano de execução de teste. Pode se utilizar os modelos construídos para explicar a novos integrantes da equipe de teste sobre o fluxo interno do sistema contribuindo para uma adaptação mais rápida.

Como pontos negativos o método possui uma curva de aprendizado elevada exigindo algum tempo para a adaptação do arquiteto de teste, além de exigir pleno conhecimento do sistema e demandar tempo para a análise e construção dos modelos em Rede de Petri e Ontologia. Devido a esses problemas é necessário um teste em algum ambiente de trabalho real para confirmar sua viabilidade prática.

Um conversor de diagramas UML para Rede de Petri é uma área de melhoria plausível, pois diminuiria o tempo gasto para a elaboração dos modelos que é a atividade que mais exige recursos. Outra situação para desenvolvimento futuro é a melhoria do modelo, sincronizando melhor os sub-modelos paralelos, eliminando “*dead-locks*” e permitindo assim a utilização da simulação desse modelo para ajudar na avaliação de possíveis erros funcionais presentes no escopo do projeto.

Com relação a validação da execução, uma vez preparado o ambiente, o método é relativamente simples e rápido de ser aplicado obtendo resultados úteis em pouco tempo. Há facilidade de compreender os conceitos de ontologia e da linguagem OWL, o que proporciona um incentivo para a implementação em empresas.

O procedimento também favorece a reusabilidade e a sua manutenção, pois mesmo que a aplicação mude, não afetará a forma de validar do modelo a não ser que haja mudanças nas regras do negócio. Mesmo que existam mudanças nas regras, a alteração no modelo é simples, qualquer alteração pode ser facilmente introduzida no modelo e no programa responsável por fazer a integração.

Como pontos negativos o método necessita de um ambiente bem estruturado interligando o banco de dados da aplicação e o modelo, o que demandará muitos recursos para a preparação. Inconvenientemente, a técnica só funciona para casos de testes que interagem com a base de dados, testes de “*layout*” de telas e de funcionalidades simples de botões não podem ser avaliados sem adaptações.

Um padrão de integração entre aplicações e o modelo ontológico é um ponto de melhoria, uma vez que um padrão exista, facilitaria a construção de modelos e até na construção da arquitetura do banco de dados. Um ambiente que una todos os programas envolvidos (modelo ontológico, extrator de caso de testes, modelo dos fluxos da aplicação, aplicação e banco de dados) de forma automática facilitaria consideravelmente a manutenção ganhando tempo e diminuindo custos do projeto.

REFERÊNCIAS

ADJIR, N.; SAQUI-SANNES, P. D.; RAHMOUNI, M. K. Time-optimal Real-Time Test Case Generation Using Prioritized Time Petri Nets. **First International Conference on Advances in System Testing and Validation Lifecycle, 2009. VALID '09**, Porto, 20 Setembro 2009. 110-116.

BAI, X. et al. Ontology-Based Test Modeling and Partition Testing of Web Services. **Web Services, 2008. ICWS '08. IEEE International Conference on**, Beijing, 26 Setembro 2008. 465-472.

BERNERS-LEE, T. et al. Uniform Resource Identifiers (URI): Generic Syntax. **USC Viterbi School of Engineering**, 1998. Disponível em: <<http://www.isi.edu/in-notes/rfc2396.txt>>. Acesso em: 10 Maio 2015.

CAI, L.; ZHANG, J.; LIU, Z. Generating Test Cases Using Colored Petri Net. **2nd International Symposium on Information Engineering and Electronic Commerce (IEEC), 2010**, Ternopil, 23 Julho 2010. 1-5.

DESEL, J. et al. A Test Case Generator for the Validation of High-Level Petri Nets. **6th International Conference on Emerging Technologies and Factory Automation Proceedings, 1997. ETFA '97., 1997**, Los Angeles, 9 Setembro 1997. 327-332.

DONG, W. Test Case Reduction for BPEL-based Testing. **International Symposium on Electronic Commerce and Security, 2008**, Guangzhou, 22 Agosto 2008. 814-817.

ELBERZHAGER, F. et al. Reducing test effort: A systematic mapping study on existing approaches. **Information and Software Technology**, Outubro 2012. 1092-1106.

FAN, C.-F.; WANG, W.-S. Validation test case generation based on safety analysis ontology. **Annals of Nuclear Energy**, 22 Março 2012. 46-58.

FOYO, P. M. G. D. GHENeSys: Uma rede estendida orientada a objetos para projeto de sistemas discretos. **Master's thesis, Escola Politécnica da Universidade de São Paulo**, São Paulo, 2001.

FOYO, P. M. G. D.; SILVA, J. R. Some Issues in Real-Time Systems Verification Using Time Petri Nets. **Journal of the Brazilian Society of Mechanical Sciences & Engineering**, Dezembro 2011. 467-474.

GARCÍA-CASTRO, R.; GÓMEZ-PÉREZ, A. A keyword-driven approach for generating OWL DL conformance test data. **Engineering Applications of Artificial Intelligence**, 9 Agosto 2012. 1413-1420.

GHALLAB, M. et al. PDDL – The Planning Domain Definition Language V. 2. **Technical Report, report CVC TR-98-003 / DCS TR-1165, Yale Center for Computational Vision and Control**, New Haven, Outubro 1998. 1-26.

GONG, H.; LI, J. Generating Test Cases of Object-Oriented Software Based on EDPN and Its Mutant. **The 9th International Conference for Young Computer Scientists, 2008. ICYCS 2008**, Hunan, 18 Novembro 2008. 1112 - 1119.

HO, I.; LIN, J.-C. Generating Test Cases for Real-time Software by Time Petri Nets Model. **Eighth Asian Test Symposium, 1999. (ATS '99) Proceedings.**, Shanghai, 16 Novembro 1999. 295-300.

HUANG, H. et al. Automated Model Checking and Testing for Composite Web Services. **Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on**, 20 Maio 2005. 300-307.

ISO/IEC 15909, FINAL DRAFT V. 4.7.1. High-level Petri Nets - Concepts, Definitions and Graphical Notation, Outubro 2000.

ISTQB. Certificação em Teste Foundation Level Syllabus, Agosto 2011.

KATO, E. R. R.; PORTO, A. J. V. Associação de Redes de Petri e Linguagens de Programação CLP para o Projeto do Controle de Sistemas de Manufatura Discreta. **4º SBAI - Simpósio Brasileiro de Automação Inteligente**, São Paulo, 10 Setembro 1999. 563-568.

KISSOUM, Y.; SAHNOUN, Z. A Recursive Colored Petri Nets Semantics for AUML as Base of Test Case Generation. **IEEE/ACS International Conference on Computer Systems and Applications, 2008. AICCSA 2008**, Doha, 22 Abril 2008. 785-792.

LIU, J.; YE, X.; LE, J. Colored Petri nets model based conformance test generation. **2011 IEEE Symposium on Computers and Communications (ISCC)**, Kerkyra, 18 Agosto 2011. 967-970.

LIU, Z. et al. Test Coverage for Collaborative Workflow Application based on Petri Net. **14th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 2010**, Shanghai, 14 Abril 2010. 213-218.

LUN, L.; XU, H. An Approach to Software Architecture Testing. **The 9th International Conference for Young Computer Scientists, 2008**, Washington, 2008. 1070-1075.

MIRALLES, J. A. S.; SILVA, J. R. Enquadrando GHENeSys dentro do Petri Net Standard. **In: Congresso Brasileiro de Engenharia Mecânica**, Bonito, 2010. 735-741.

MIRZAEIAN, E. et al. An Optimized Approach to Generate Object Oriented Software Test Case by Colored Petri Net. **2nd International Conference on Software Technology and Engineering (ICSTE), 2010**, San Juan, 3 Outubro 2010. 251-255.

MIYAGI, P. E. **Controle Programável**. 3ª Edição. ed. São Paulo: Edgard Blücher, 2007.

MURATA, T. Petri Nets: Properties, Analysis and Applications. **Proceedings of the IEEE, v. 77, no. 4**, 1989.

NETO, A. B. Método de Validação e Teste em Software Utilizando Redes de Petri Modulares. **10th International Conference on Information Systems and Technology Management**, São Paulo, jun. 2013.

RAMASWAMY, S. A Petri Net based Approach for Establishing Necessary Software Design and Testing Requirements. **IEEE International Conference on Systems, Man, and Cybernetics, 2000**, Nashville, 8 Outubro 2000. 3087-3092.

RAMCHANDANI, C. Analysis of asynchronous concurrent systems by timed Petri nets. **Ph. D. diss.**, Cambridge, Setembro 1973.

REZA, H.; KERLIN, S. D. A Model-Based Testing Using Scenarios and Constraints-based Modular Petri Nets. **Eighth International Conference on Information Technology: New Generations (ITNG), 2011**, Las Vegas, 11 Abril 2011. 568-573.

SILVA, J. R.; DEL FOYO, P. M. G. Timed Petri Net. **InTech**, 29 Agosto 2012. 359-378.

SUN, T.; YE, X.; LIU, J. A test sequence selection method for parallel software systems. **Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on**, Tianjin, Dezembro 2011. 163-167.

THIEL, S.; DERICHSEWILER, F. Petri Net Based Verification of Causal Dependencies in Electronic Control Unit Test Cases. **IEEE 35th Annual on**

Computer Software and Applications Conference Workshops (COMPSACW), 2011, Munique, 18 Julho 2011. 143-148.

W3C. Extensible Markup Language (XML) 1.0 (Second Edition). **W3C Recommendation**, 2000. Disponivel em: <<http://www.w3.org/TR/2000/REC-xml-20001006>>. Acesso em: 10 Maio 2015.

W3C. OWL Web Ontology Language Guide. **W3C Recommendation**, 2004. Disponivel em: <<http://www.w3.org/TR/owl-guide/>>. Acesso em: 1 Setembro 2013.

W3C. OWL Web Ontology Language Overview. **W3C Recommendation**, 2004. Disponivel em: <<http://www.w3.org/TR/owl-features/>>. Acesso em: 4 junho 2013.

W3C. OWL Web Ontology Language Reference. **W3C Recommendation**, 2004. Disponivel em: <<http://www.w3.org/TR/owl-ref/>>. Acesso em: 28 Julho 2013.

W3C. RDF Primer. **W3C Recommendation**, 2004. Disponivel em: <<http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>>. Acesso em: 28 Julho 2013.

W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. **W3C Recommendation**, 2004. Disponivel em: <<http://www.w3.org/TR/rdf-concepts/>>. Acesso em: 29 Setembro 2013.

W3C. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. **W3C Member Submission**, 2004. Disponivel em: <<http://www.w3.org/Submission/SWRL/>>. Acesso em: 05 Outubro 2014.

WANG, Y. et al. Ontology-Based Test Case Generation for Testing Web Services. **Eighth International Symposium on Autonomous Decentralized Systems, 2007. ISADS '07**, Sedona, 21 Março 2007. 43-50.

ZHU, H. A Framework for Service-Oriented Testing of Web Services. **Proc. 30th Ann. Int'l Computer Software and Applications Conf. (COMPSAC '06)**, 17 Setembro 2006. 145-150.

ZHU, H.; ZHANG, Y. Collaborative Testing of Web Services. **IEEE Transactions on Services Computing**, vol. 5, no. 1. 116-130.