MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

TESTES DE SOFTWARE VIA MODEL CHECKING PARA SISTEMAS ESPACIAIS CRÍTICOS

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA

(PIBIC/CNPq/INPE)

Aluno: Felipe Elias Costa da Silva (UNISAL, Bolsista PIBIC/CNPq)

E-mail: felipe.eliascs@hotmail.com

Orientador: Dr. Valdivino Alexandre de Santiago Júnior
(LABAC/COCTE/INPE)

E-mail: valdivino.santiago@inpe.br

Julho de 2017

RESUMO

Testes de software e Model Checking (método de Verificação Formal) são processos/métodos diferentes para assegurar a qualidade de sistemas de software. Para sistemas críticos, tais como satélites e aplicações de balões estratosféricos que o INPE desenvolve, a questão da qualidade é ainda mais relevante, pois um defeito no software pode ocasionar grandes perdas financeiras. Dado a busca exaustiva no espaço de estados que Model Checking realiza, pesquisadores vêm propondo gerar casos de testes de software por meio de Model Checking. Nesse contexto, o raciocínio é interpretar os contraexemplos gerados pelos Model Checkers (ferramentas de software que possuem uma realização da teoria de Model Checking) como casos de teste. O principal desafio é forçar o Model Checker a criar, sistematicamente, conjuntos de tais contraexemplos. Esse projeto de pesquisa possui três objetivos específicos: a.) realizar a geração de casos de teste de software a partir de Model Checking; b.) atualizar a metodologia e a ferramenta SOLIMVA com as soluções tecnológicas desenvolvidas no projeto; e c.) aplicar a nova versão da ferramenta e da metodologia SOLIMVA a software de sistema espacial crítico em desenvolvimento no INPE. Esse relatório apresenta as atividades desenvolvidas no período de 01 de agosto de 2015 a 13 de julho de 2017.

1.) INTRODUÇÃO

Softwares precisam operar da maneira mais correta possível, pois defeitos podem trazer consequências como perdas financeiras ou até de vidas. E, para garantir um alto nível de qualidade, é necessário utilizar métodos/técnicas/processos relacionados à disciplina de Verificação e Validação (V&V) de Software, tais como Teste [Delamaro et al. 2007][Santiago Júnior 2011][Santiago Júnior e Vijaykumar 2012] e Verificação Formal de Software [Baier e Katoen 2008][Santos 2004].

Teste de software é, muito provavelmente, o processo mais adotado, na prática, entre todos relacionados à V&V. O objetivo de testar um produto de software é encontrar defeitos no código-fonte do mesmo. Inúmeras teorias, metodologias, abordagens têm sido propostas e/ou usadas para as diversas atividades do processo de Testes de Software. Uma das atividades do processo de Teste de software mais estudada, mas que ainda apresenta diversos desafios, é a **geração/seleção de casos de teste**. No fundo, dado que a execução de teste exaustivo não é viável, a idéia é utilizar de formas para selecionar, de infinitas possibilidades, um conjunto de dados de entrada de teste do domínio de entrada de um programa P, de forma a detectar o maior número possível de defeitos. Existem diversas abordagens para esse propósito, mas uma das mais interessantes é a conhecida como Testes Baseados em Modelos (TBM). TBM é uma estratégia de teste em que os casos de teste são derivados completamente, ou parcialmente, a partir de um modelo que descreve algum aspecto (funcionalidade, segurança, desempenho, etc.) de um software [Utting e Legeard 2007]. A aplicação de TBM requer que o comportamento ou estrutura do software tenham sidos descritos por meio de modelos com regras bem definidas, tais como Métodos Formais (Máquinas de Estados Finitos, Statecharts, Z, B, Sistemas de Transições) e abordagens não formais como diagramas e modelos da Unified Modeling Language (UML) [Santiago Júnior 2011].

Por sua vez, Verificação Formal é outra área de V&V extremamente relevante no contexto de desenvolvimento de sistemas/softwares críticos, e pode ser definida como a análise matemática de provar ou não provar a corretude de um sistema de hardware ou software com relação a uma certa especificação ou propriedade [Ganai e Gupta 2007]. Pelo extensivo uso de

lógica matemática, Verificação Formal possui fortes conexões com a base teórica da Ciência da Computação. Os métodos para análise são conhecidos como **Métodos de Verificação Formal**, os quais podem ser classificados geralmente como: Provas de Teorema e Model Checking [Baier e Katoen 2008][Clarke e Emerson 2008][Queille e Sifakis 2008]. Particularmente, Model Checking tem uma maior aceitação, tanto na indústria como na academia, do que Provas de Teorema pois Model Checking é uma técnica muito mais automatizada do que Provas de Teorema. Dado que exista um modelo de estados finitos (também conhecido por Sistema de Transição (ST)) de um sistema e uma propriedade formal, a idéia por trás de Model Checking é realizar, sistematicamente e de forma automatizada, a verificação que tal propriedade é satisfeita (verdadeira) pelo (por um determinado estado no) modelo [Baier e Katoen 2008].

Tradicionalmente, em Model Checking, as propriedades são geradas baseadas em documentos de requisitos e são formalizadas utilizando uma variedade de lógicas temporais tais como Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Timed Computation Tree Logic (TCTL), Probabilistic Computation Tree Logic (PCTL), entre muitas outras. Uma propriedade especifica, portanto, o comportamento desejado do sistema em consideração. Se um Sistema de Transição (ST) não satisfaz uma propriedade então um contraexemplo é gerado mostrando um traço que indica a violação. Por outro lado, o ST descreve o comportamento do sistema.

Teste e Model Checking são, portanto, técnicas diferentes para assegurar a qualidade de sistemas de software. No entanto, dado a busca exaustiva no espaço de estados que Model Checking realiza, pesquisadores vêm propondo gerar casos de testes por meio de Model Checking [Fraser et al. 2009]. Como já mencionado anteriormente, realizar Teste exaustivo de software é inviável. Por outro lado, a Verificação Formal pode provar se a propriedade é violada ou satisfeita, mas a prova mostra que um dado modelo satisfaz ou não a propriedade, enquanto a implementação real também é influenciada pelo seu ambiente, por exemplo, plataforma, compilador, etc. Embora existam Model Checkers (ferramentas de software que possuem uma realização da teoria de Model Checking) que se baseiam no próprio código-fonte ou código objeto do produto de software para realizar o Model Checking

[Pasareanu et al. 2013][Kroening et al. 2015], o espectro de defeitos encontrados por Testes pode ser diferente da classe de defeitos encontrados por Model Checking. Portanto, Testes e Verificação Formal podem ser usados de forma complementar em um processo de V&V de software. E, para o caso de gerar casos de teste de software por meio de Model Checking, o raciocínio é interpretar os contraexemplos gerados pelos Model Checkers como casos de teste. O principal desafio é forçar o Model Checker a criar, sistematicamente, conjuntos de tais contraexemplos.

A metodologia SOLIMVA [Santiago Júnior 2011] [Santiago Júnior e Vijaykumar 2012] foi desenvolvida em um trabalho de doutorado da CAP/INPE para alcançar duas metas:

a) Geração de casos de teste de sistema e aceitação baseados em modelos a partir de artefatos de requisitos elaborados em Linguagem Natural (LN). Para esse propósito, uma ferramenta, também denominada SOLIMVA, foi projetada e implementada, e tal ferramenta traduzia, automaticamente, requisitos elaborados em LN em modelos Statecharts [Harel 1987]. Uma vez gerados os Statecharts, outra ferramenta, GTSC [Santiago Júnior et al. 2012], é usada para gerar Casos de Teste Abstratos os quais depois são transformados em Casos de Teste Executáveis. Entre as teorias usadas para alcançar esse objetivo estão TBM, designs combinatoriais, e Processamento em Linguagem Natural/linguística computacional (Part Of Speech Tagging [Toutanova et al. 2003], Word Sense Disambiguation [Navigli 2009]). Essa é a versão 1.0 tanto da metodologia como da ferramenta SOLIMVA. A metodologia SOLIMVA 1.0 está mostrada na Figura 1;

b) Detecção de não completude em especificações de software. Entre as teorias usadas para alcançar esse propósito estão Model Checking combinado com arranjos simples de valores de variáveis e padrões de especificação [Dwyer et al. 1999].

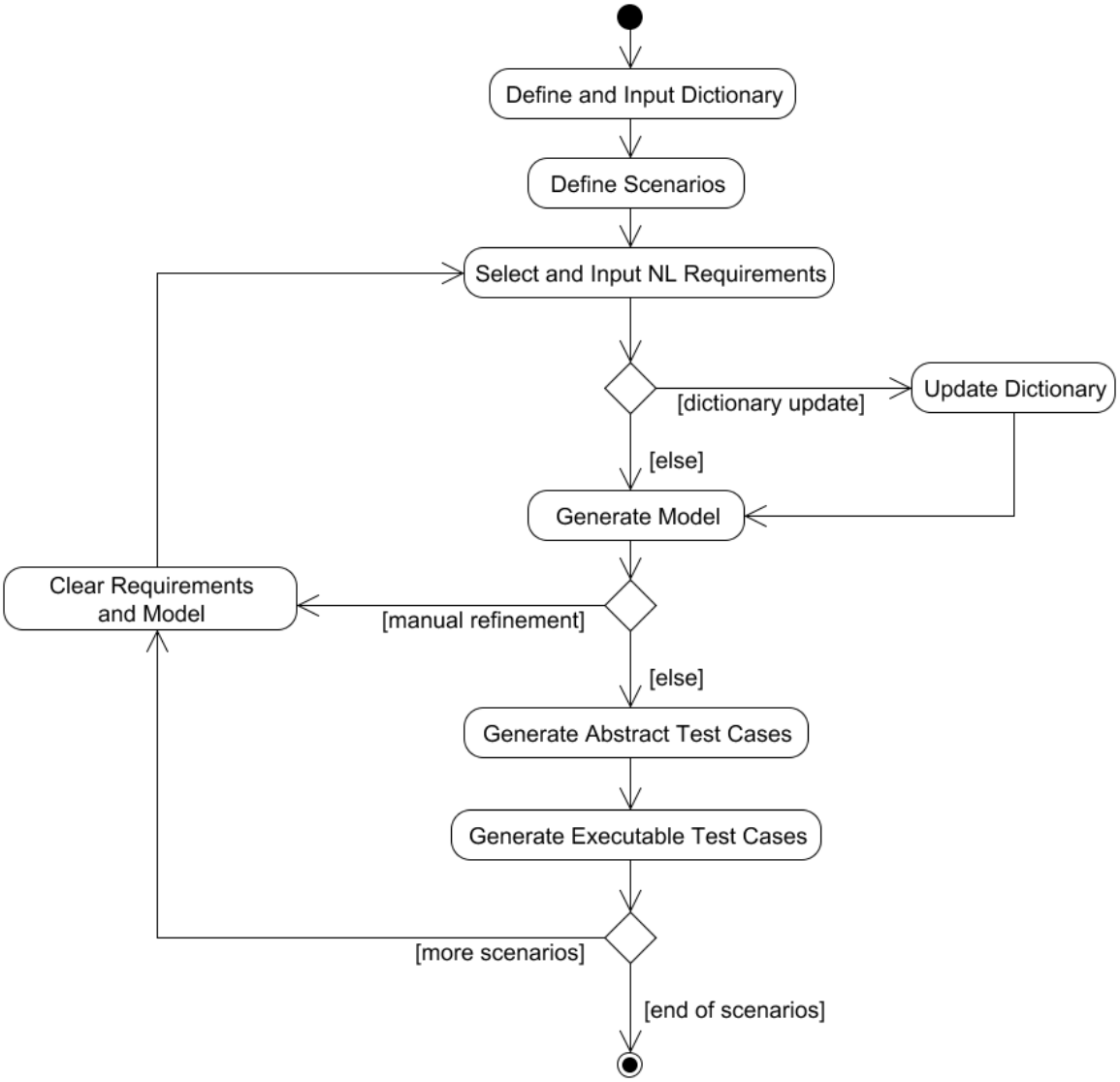


Figura 1 – Metodologia SOLIMVA 1.0

Para as duas metas citadas acima, a metodologia SOLIMVA foi aplicada a um estudo de caso da área espacial, Software for the Payload Data Handling Computer (SWPDC), desenvolvido no escopo do projeto de pesquisa, fomentado pela Financiadora de Estudos e Projetos (FINEP), denominado Qualidade do Software Embarcado em Aplicacões Espaciais (QSEE). Em relação à meta primária, também foram apresentadas diretivas de como aplicar a metodologia SOLIMVA a um segundo estudo de caso do domínio espacial, relacionado ao Segmento Solo: Satellite Control System (SATCS). O SATCS está sendo desenvolvido pela Divisão de Desenvolvimento de Sistemas de Solo (DSS/ETE).

O SWPDC está sendo, atualmente, adaptado para ser o software do computador do Subsistema de Gestão de Bordo de um outro projeto de pesquisa e desenvolvimento financiado pela FINEP, o experimento científico protoMIRAX (em desenvolvimento na DAS/CEA com parceria do LAC/CTE). Além disso, parte da metodologia SOLIMVA também já está sendo aplicada ao projeto protoMIRAX.

A ferramenta de software SOLIMVA foi desenvolvida na linguagem de programação Java usando o paradigma de Orientação a Objetos. A despeito da metodologia/ferramenta SOLIMVA ter sido aplicada a um estudo de caso relevante da área espacial e do INPE (SWPDC), naturalmente, a ferramenta precisa de uma série de melhorias para que possa ser aplicada a outros projetos da área espacial do INPE. A metodologia SOLIMVA 1.0 (Figura 1), como um todo, necessitou de ferramentas externas para que pudesse ser aplicada. Em particular, foi necessário usar o ambiente GTSC para gerar os casos de teste de software, após os modelos Statecharts terem sido criados pela ferramenta SOLIMVA. Embora o GTSC seja um ambiente interessante, que está sendo aplicado a projetos do INPE tal como o projeto protoMIRAX, o mesmo gera casos de teste a partir de modelos Statecharts ou Máquinas de Estados Finitos. Portanto, o GTSC não permite gerar casos de teste por meio de Model Checking. Além disso, a ferramenta SOLIMVA não está integrada ao ambiente GTSC e, como o uso das 2 ferramentas se faz necessário para gerar casos de testes, então o profissional precisar fazer, manualmente, a tradução da saída da SOLIMVA para a entrada do GTSC. Do ponto de vista de uso em aplicações reais e complexas, tais como projeto de satélites e balões estratosféricos em desenvolvimento no INPE, é muito mais interessante se houvesse uma integração entre as ferramentas. Adicionalmente, esse processo de tradução manual entre duas ferramentas pode ser muito propenso a erros.

Portanto, os objetivos específicos desse projeto são:

a.) Realizar a geração de casos de teste de software a partir de Model Checking;

b.) Atualizar a metodologia e a ferramenta SOLIMVA com as soluções tecnológicas desenvolvidas no projeto;

c.) Aplicar a nova versão da ferramenta e da metodologia SOLIMVA a software de sistema espacial crítico em desenvolvimento no INPE.

Esse relatório apresenta as atividades desenvolvidas no período de **01 de agosto de 2015 a 13 de julho de 2017.** Esse é o relatório final do projeto.

2.) CRONOGRAMA DE ATIVIDADES E ETAPAS CONCLUÍDAS

Conforme mostrado no “Formulário para Solicitação de Bolsa PIBIC”, a metodologia a ser empregada para atender aos objetivos do projeto está descrita a seguir.

1. Estudar a fundamentação teórica relativa ao projeto. Especificamente, se familiarizar com os conceitos relacionados à Testes de software, Verificação Formal de software (Model Checking), Statecharts, abordagens para gerar casos de teste de software a partir de Model Checking, e a metodologia SOLIMVA juntamente com a ferramenta de software que a apóia (também denominada de SOLIMVA);

2. Analisar as ferramentas de software (Model Checkers) que são usadas para a realização de Model Checking. Selecionar 1 dessas ferramentas (Model Checker) para ser usada no projeto;

3. Adaptar algoritmo já existente, ou propor um novo algoritmo, para transformar modelos Statecharts para o Model Checker selecionado no item anterior;

4. Incorporar o algoritmo adaptado (ou novo algoritmo), para transformar modelos Statecharts para Model Checker, à ferramenta SOLIMVA realizando a interoperabilidade entre a ferramenta SOLIMVA e o Model Checker selecionado;

5. Adaptar técnica já existente, ou propor uma nova técnica, para gerar casos de testes de software a partir de Model Checking;

6. Incorporar a técnica adaptada (ou nova técnica), para gerar casos de testes de software a partir de Model Checking, à ferramenta SOLIMVA realizando a interoperabilidade entre a ferramenta SOLIMVA e o Model Checker selecionado;

7. Atualizar as interfaces gráficas com o usuário (Graphical User Interfaces - GUIs) da ferramenta SOLIMVA de acordo com novas necessidades que apareçam com o uso da ferramenta;

8. Aplicar a nova versão da ferramenta e da metodologia SOLIMVA a estudo de caso (software) crítico da área espacial;

9. Submeter artigo para conferência e/ou workshop e/ou simpósio na área de Engenharia de Software e/ou Métodos Formais, e elaborar relatório final de atividades.

O cronograma para desenvolvimento das atividades da metodologia está mostrado na Figura 2 a seguir. O número das atividades está de acordo com os números mostrados acima. Cada uma das colunas de Ano I e II representa um mês.

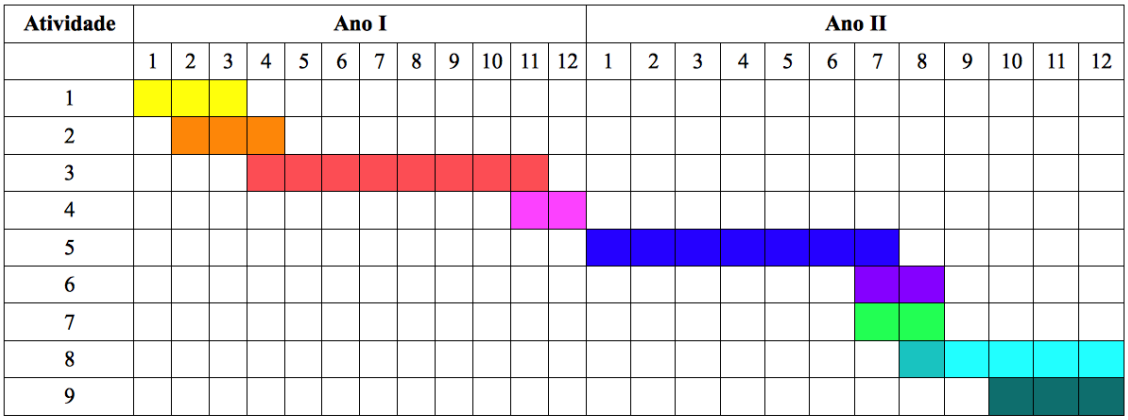


Figura 2 – Cronograma de atividades

Portanto, esse relatório compreende o **mês 1 do Ano I (agosto/2015) ao mês 12 do Ano II (julho/2017)**. Ou seja, é o relatório final do projeto. Considerando as atividades previstas para serem desenvolvidas, mostradas na Figura 2, a Tabela 1 a seguir mostra as atividades concluídas considerando o período total do projeto, a que se refere esse relatório de acompanhamento.

Tabela 1 – Etapas Concluídas

	Atividades da Metodologia	**Previsão**	**Realização**
1	Estudar a fundamentação teórica relativa ao projeto. Especificamente, se familiarizar com os conceitos relacionados à Testes de software, Verificação Formal de software (Model Checking), Statecharts, abordagens para gerar casos de teste de software a partir de Model Checking, e a metodologia SOLIMVA juntamente com a ferramenta de software que a apóia (também denominada de SOLIMVA);	100%	100%
2	Analisar as ferramentas de software (Model Checkers) que são usadas para a realização de Model Checking. Selecionar 1 dessas ferramentas (Model Checker) para ser usada no projeto;	100%	100%
3	Adaptar algoritmo já existente, ou propor um novo algoritmo, para transformar modelos Statecharts para o Model Checker selecionado no item anterior;	100%	100%
4	Incorporar o algoritmo adaptado (ou novo algoritmo), para transformar modelos Statecharts para Model Checker, à ferramenta SOLIMVA realizando a interoperabilidade entre a ferramenta SOLIMVA e o Model Checker selecionado;	100%	95%
5	Adaptar técnica já existente, ou propor uma nova técnica, para gerar casos de testes de software a partir de Model Checking;	100%	100%
6	Incorporar a técnica adaptada (ou nova técnica), para	100%	95%

	gerar casos de testes de software a partir de Model Checking, à ferramenta SOLIMVA realizando a interoperabilidade entre a ferramenta SOLIMVA e o Model Checker selecionado;		
7	Atualizar as interfaces gráficas com o usuário (Graphical User Interfaces - GUIs) da ferramenta SOLIMVA de acordo com novas necessidades que apareçam com o uso da ferramenta;	100%	95%
8	Aplicar a nova versão da ferramenta e da metodologia SOLIMVA a estudo de caso (software) crítico da área espacial;	100%	100%
9	Submeter artigo para conferência e/ou workshop e/ou simpósio na área de Engenharia de Software e/ou Métodos Formais, e elaborar relatório final de atividades.	100%	100%

Na Tabela 1 acima, a coluna **Previsão** mostra a porcentagem prevista para a realização da atividade, e a coluna **Realização** mostra a porcentagem realmente realizada da atividade, considerando o período total do projeto (01 de agosto de 2015 a 13 de julho de 2017). Desse modo, pode-se dizer que as atividades previstas para o projeto foram cumpridas de maneira plenamente satisfatórias, onde as atividades 1, 2, 3, 5, 8 e 9 foram totalmente concluídas (100%), e as atividades 4, 6 e 7 foram quase que totalmente concluídas (95%). Porém, percebe-se que as atividades 4, 6 e 7 são muito mais relevantes no contexto tecnológico do que científico. As principais atividades do ponto de vista científico foram totalmente cumpridas com um importante fato: um artigo [Santiago Júnior e Silva 2017] relacionado a esse projeto foi **aceito para publicação** no **2º Simpósio Brasileiro de Teste de Software Sistemático e Automatizado (SAST 2017) (http://www.lia.ufc.br/~cbsoft2017/ii-sast/chamada-de-trabalhos/)**, que será realizado no contexto do **CBSoft 2017 – Conferência Brasileira de Software: Teoria e Prática**. O SAST é o principal congresso na área de teste de software no Brasil, tendo alcance internacional devido a participação de pesquisadores estrangeiros no comitê de avaliação do simpósio. Portanto, o aceite desse artigo demonstra que os objetivos científicos e tecnológicos desse projeto de pesquisa foram completamente alcançados. A referência para o artigo [Santiago Júnior e Silva 2017] é:

Santiago Júnior, Valdivino Alexandre de; Silva, Felipe Elias Costa da. From Statecharts to Model Checking: A Hierarchy-based Translation and Specification Patterns Properties to Generate Test Cases. In: The 2nd Brazilian Symposium, 2017, Fortaleza, CE, Brazil. Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing - SAST (Accepted for publication).

Na atividade 1, foram estudados os fundamentos teóricos relacionados ao projeto, os conceitos relacionados à Teste de Software, Verificação Formal de software (Model Checking), Statecharts, e a metodologia SOLIMVA juntamente com a ferramenta de software que a apóia (também denominada de SOLIMVA, versão 1.0).

Na atividade 2, foi feita uma análise, sobre os mais diversos aspectos, de três ferramentas utilizadas para apoiar o processo de Model Checking: NuSMV [NuSMV 2015a, NuSMV 2015b], SPIN [Holzmann 2003] e UPPAAL [Behrmann et al. 2004]. Após a análise realizada, considerando fatores e critérios de usabilidade segundo o modelo QUIM [Seffah et al. 2006], foi selecionado o Model Checker NuSMV para alcançar os objetivos do projeto.

A atividade 3 foi uma das mais desafiadoras do projeto, devido a necessidade de realizar uma transformação, fundamentada matematicamente, de modelo Statechart para a linguagem do Model Checker selecionado, o NuSMV. Definiu-se um método, denominado *Hierarchy-based Statecharts translation into Model Checking and Specification Patterns Properties for Testing* (HiMoST; Tradução baseada em Hierarquia de Statecharts para Model Checking e Propriedades de Padrão de Especificação para apoiar o Teste de Software), para gerar casos de teste de software via Model Checking. Partindo de um modelamento comportamental elaborado nos Statecharts de Harel, o método HiMoST faz a tradução de Statecharts para uma estrutura geral baseada na linguagem do Model Checker NuSMV [Santiago Júnior Silva 2017]. Basicamente, todas as características dos Statecharts de Harel são abordadas no método HiMoST: estados XOR e AND (paralelismo), eventos, histórico raso e profundo, hierarquia. No entanto, no método HiMoST, hierarquia é o fator principal que direciona o processo de tradução de Statecharts para a estrutura geral do Model Checker NuSMV.

A atividade 4 teve seu desenvolvimento quase todo concluído. A implementação/codificação da abordagem HiMoST se deu em uma perspectiva na ferramenta SOLIMVA: BEHAVIOR (BEH). Nessa perspectiva, o usuário cria uma modelagem comportamental em Statecharts por meio de uma Interface Gráfica do Utilizador (GUI - Graphical User Interface). Então, a versão 1.2 da ferramenta SOLIMVA, automaticamente, transforma essa notação gráfica em State Chart XML (SCXML), uma recomendação do W3C. A partir do SCXML, a abordagem HiMoST transforma o modelo Statechart para uma estrutura geral baseada na linguagem do NuSMV. A implementação do método HiMoST na ferramenta SOLIMVA gera um código NuSMV que precisa de uns ajustes manuais após o processo de tradução.

Ainda em termos de codificação, a ferramenta SOLIMVA teve atualizada uma segunda perspectiva: NATURAL LANGUAGE (NL). Nessa perspectiva, a ferramenta SOLIMVA toma como entrada a notação textual da versão anterior (versão 1.0 da SOLIMVA), notação esta que representa um modelo Statechart gerado a partir de requisitos criados em Linguagem Natural. Então, a

ferramenta SOLIMVA transforma essa notação textual, que representa um modelo Statechart, para a linguagem do Model Checker NuSMV. Essa versão da ferramenta SOLIMVA é a 1.1. Entretanto, o método de transformação NL → Statecharts → NuSMV não é exatamente o método HiMoST implementado na versão 1.2 da SOLIMVA (perspectiva BEH).

Na atividade 5, foi feito um estudo das abordagens para gerar casos de teste de software a partir de Model Checking, que, basicamente, podem ser divididas em duas grandes categorias: baseadas em cobertura e baseadas em mutantes [Fraser et al. 2009]. O método HiMoST [Santiago Júnior e Silva 2017] também possui um mecanismo de formalização de propriedades CTL por meio dos padrões de especificação de Dwyer et al. [Dwyer et al. 1999] e designs combinatoriais via o algoritmo T-Tuple Reallocation (TTR) [Balera e Santiago Júnior 2017].

As atividades 6 e 7 foram quase todas concluídas por meio de melhorias nas GUIs da ferramenta SOLIMVA. Na atividade 8, totalmente concluída, foi usado um modelo do SWPDC (estudo de caso da área espacial) para aplicar o método HiMoST como um todo, além de outros três estudos de caso: Controlador de Trem, Biblioteca Digital, e a implementação do TTR. Com esses quatro estudos de caso, em [Santiago Júnior e Silva 2017] foi apresentado um quasiexperimento objetivando responder a seguinte questão de pesquisa: qual é a abordagem que tem melhor custo-efetividade para geração de casos de teste considerando os seguintes padrões/escopos de padrões no contexto do método HiMoST: Ausência/Global (ABS), Resposta Encadeada (S, T responde a P)/Global (REC), Precedência Encadeada (P precede a S, T)/Global (PC1), e Precedência Encadeada (S,T precede a P)/Global (PC2)? A conclusão desse quasiexperimento é que PC1 é a melhor opção para gerar suites de teste com maior custo-efetividade. REC é melhor do que ABS somente se efetividade tiver prioridade sobre custo. Se efetividade e custo possuem a mesma prioridade, não há diferença entre REC e ABS.

Como já foi mencionado, um artigo foi submetido e aceito no Simpósio SAST 2017, demonstrando a viabilidade da pesquisa realizada nesse projeto de iniciação científica.

Os detalhamentos do desenvolvimento das atividades estão apresentados a seguir.

3.) ATIVIDADE 1: FUNDAMENTAÇÃO TEÓRICA

Verificação de modelos (Model Checking) é uma técnica de verificação automática para sistemas de estados finitos, que aplica uma busca exaustiva no espaço de estados de um determinado modelo. Foi desenvolvida na década

de 80 por Clarke, Emerson, Queille e Sifakis. E sua sintaxe é escrita em lógica temporal proposicional.

Muito utilizada para verificar exigências de sistemas críticos de tempo real e sistemas embarcados, a verificação de modelos permite fazer as verificações antes mesmo de começar a programar o software. Encontrar falhas logo no início economiza tempo e desgaste do programador, pois os requisitos que não foram atendidos podem ser consertados ante de iniciar o código, gerando economia no desenvolvimento do projeto.

Geralmente, a aplicação do Model Checking ocorre em três etapas: modelagem, especificações dos comportamentos e verificação.

1. A modelagem é a construção de um modelo formal (ST), representando os comportamentos do sistema;
2. A especificação consiste em formalizar as propriedades do sistema de acordo com os requisitos estabelecidos. Para isso, as propriedades são usualmente formalizadas em lógicas como LTL, CTL, TCTL entre outras;
3. E a verificação consiste em verificar o modelo contra as propriedades formalizadas. Se uma certa propriedade formalizada não é satisfeita pelo ST, então um contraexemplo é gerado.

Na Figura 3 podemos ver basicamente como é feita uma verificação de modelos. Nas entradas temos os grafos de transição de estados que representa todos os comportamentos possíveis do sistema e as formulas lógicas temporais que representa formalmente as propriedades a serem verificadas. Na saída o Verificador de Modelos responde sim, caso a propriedade seja verdadeira, ou não caso seja falsa. Sendo falsa, é gerado um contraexemplo que indica o traço no espaço de estados em que a propriedade é falsa

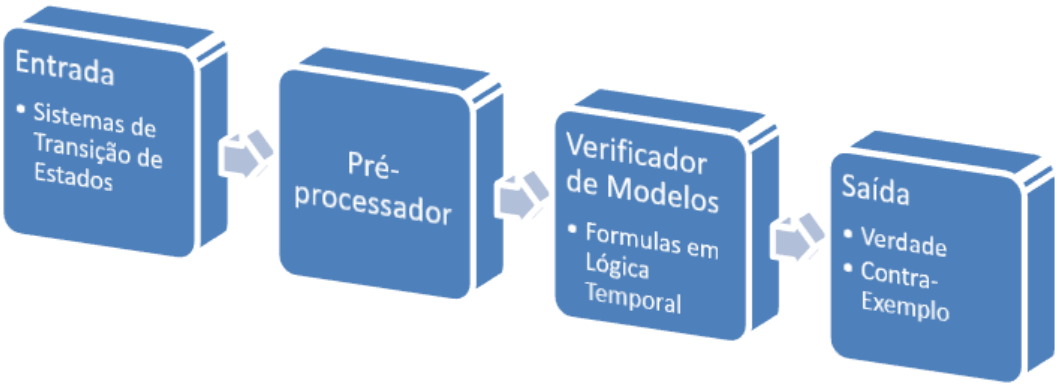


Figura 3 – Visão simplificada do Processo de Model Checking.

3.1. LÓGICA PROPOSICIONAL

A Lógica Proposicional é um sistema formal que representa afirmações formadas pela combinação de proposições atômicas usando conectivos lógicos e sistema de regras de derivação. Proposições atômicas são sentenças afirmativas declarativas que possuem a propriedade de serem ou verdadeiras (valor lógico verdadeiro ($\top$)) ou falsas (valor lógico falso ($\bot$)), mas não ambas. A tabela-verdade pode ser usada para determinar/provar a validade de argumentos. São utilizados 5 conectivos lógicos, que são apresentados a seguir e a suas respectivas tabelas-verdade estão na Figura 4:

- $\neg$ - Negação
- $\wedge$ - Conjunção
- $\vee$ - Disjunção
- $\rightarrow$ - Condicional
- $\leftrightarrow$ - Bicondicional

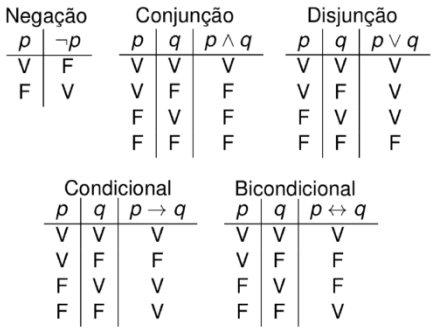


Figura 4 - Tabelas verdade elementares

A lógica proposicional é um pré-requisito para ter uma boa compreensão de lógica temporal.

3.2. LÓGICA TEMPORAL

Lógica Temporal é utilizada para formular afirmações sobre um sistema reativo quando ele evolui com o tempo. Para verificar um sistema, deve-se expressá-lo em fórmulas, que especificam os comportamentos, de uma linguagem de lógica temporal. Essa lógica encontrou uma grande importância na verificação formal de software e hardware, utilizada para declarar requisitos de sistemas.

3.2.1. LTL

A LTL, lógica temporal linear [Baier et al. 2008], é caracterizada por estender a lógica proposicional por modalidades temporais que permitem se referir ao comportamento infinito de sistemas reativos. As principais modalidades temporais definidas em LTL são:

- G – Sempre ou Globalmente;
- F – Eventualmente;
- X – Próximo;
- U – Até.

3.2.2. Verificando modelos na lógica LTL

Uma das formas de se realizar Model Checking (verificação de modelos), considerando a lógica LTL, é baseando-se em autômatos. A ideia básica é considerar variações de Autômatos Finitos Não Determinísticos (NFAs), conhecidos como Autômatos de Buchi Não Determinísticos (NBAs), que servem como aceitadores para linguagens de palavras infinitas. Então, dado um NBA *A* que especifica os traços ruins (i.e. que aceitam o complemento da propriedade em Tempo Linear *P* a ser verificada), então uma análise no produto síncrono do Transition System (*TS*) e *A* é suficiente para afirmar se o *TS* satisfaz, ou não, a propriedade *P*.

3.2.3. CTL

A CTL, Lógica de Árvore de Computação [Baier et al. 2008], é caracterizada pelo fato de sua semântica não ser baseada em uma noção de tempo linear (sequencia infinita de estados, como em LTL), mas em uma noção de tempo ramificada (árvore infinita de estados). No conceito de tempo ramificado, em cada momento podem existir muitos possíveis diferentes futuros. Além das modalidades temporais definidas em LTL (vide Seção 3.2.1), os seguintes quantificadores de caminho são usados em CTL:

- E – Para algum caminho;
- A – Para todos os caminhos.

3.2.4.Verificando modelos na lógica CTL

O Model Checking considerando requisitos formalizados em CTL pode ser realizado por um procedimento recursivo que calcula o conjunto de satisfação (*Sat*) para todas as subfórmulas de uma fórmula *P* e, então, verifica se todos os estados iniciais do *TS* pertencem ao conjunto de satisfação (*Sat*) de *P*.

3.2.5.LTL x CTL

Não pode ser feita uma comparação, em termos de expressividade, entre as duas lógicas temporais, LTL e CTL, pois, como mostrado na Figura 5, existem propriedades que podem ser formalizadas em uma lógica mas na outra não, e vice-versa. A escolha de uma lógica (LTL ou CTL) depende de uma série de fatores, desde o tipo de requisitos dos sistemas a serem avaliados, até a disponibilidade dos Model Checkers (ferramentas de software que possuem uma realização do método Model Checking) para cada lógica.

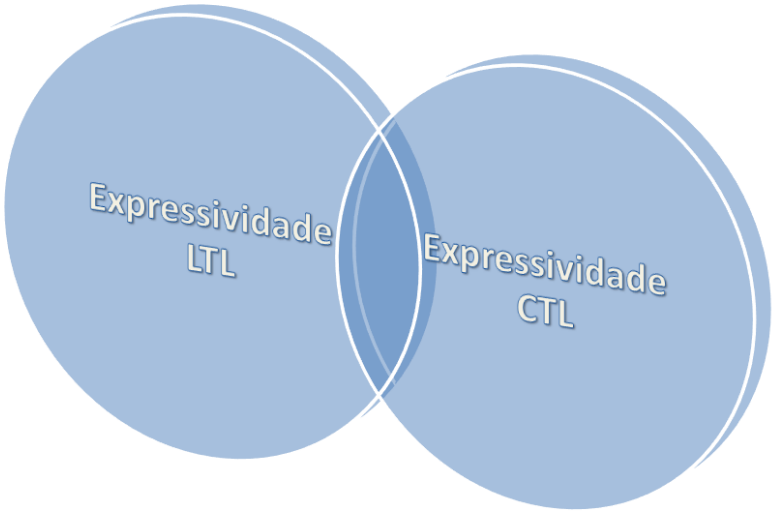


Figura 5 – LTL x CTL

3.3. Statecharts

O Statecharts é um formalismo visual criado por Harel em 1987, para especificar sistemas reativos em tempo real. Considerado uma evolução dos Diagramas de Transição de Estados por representar modelos mais complexos e claros, que suportam reações contínuas tanto externas quanto internas.

Alguns exemplos de sistemas reativos são:

- Telefone
- Rede de comunicação de dados
- Sistemas aniônicos
- Interface de usuário (Software)
- Circuitos VLSI

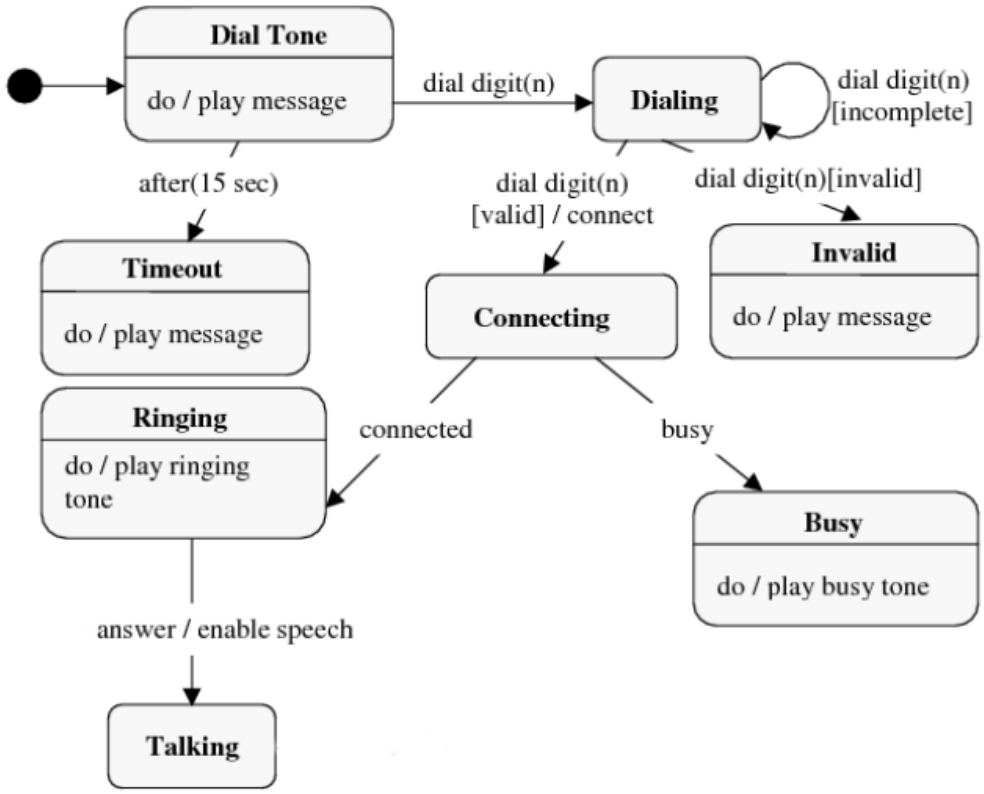


Figura 6 – Exemplo de chamada telefônica.

Características do Statecharts: i) modelo apresentado através de hierarquia de MEF, que torna o modelo mais claro, dando visibilidade a aspectos de concorrência; ii) broadcasting ou reação em cadeia, permite descrever a sincronização entre os componentes ortogonais do modelo; iii) ortogonalidade, que possibilita descrever o paralelismo entre componentes do modelo especificado; e iv) história, que permita a lembrança de estados que já foram visitados (MALDONADO, 1991).

Statecharts são fundamentados nos seguintes elementos básicos: estados, eventos, condições, ações, expressões, variáveis, rótulos e transições (FRANCÊS, 2001). Estados são usados para descrever componentes de um sistema. Estados de um Statechart representam os valores das variáveis em determinado instante do sistema e dividem-se em dois grupos: simples e composto. Simples são os que não possuem subestados. Compostos são divididos em subestados e pode ser de dois tipos: OR ou AND (FRANCÊS, 2001).

Se o estado é classificado com o tipo OR, o sistema estará sempre em um único subestado em um determinado instante. Porem se for classificado do tipo AND, o estado estará em mais de um subestado.

Evento é um acontecimento que ocorre externa ou internamente e provocam transições de estados. Tal informação é representada através das setas que interligam os diferentes estados de um sistema. Ações são elementos utilizados para representar efeitos do paralelismo em statecharts. Transições é representação gráfica para realçar uma mudança de estado no sistema. Rótulos proveem algum significado adicional, no entanto são opcionais e podem ser acrescentados às setas. Condição é um predicado opcional associado a um evento que habilita o sistema a efetuar uma transição de estado. Tal informação é representada entre parênteses “( )”.

Na **Figura 6** podemos observar um sistema modelado em Statecharts. O paralelismo é representado pelo o uso de linhas pontilhadas separando os componentes.

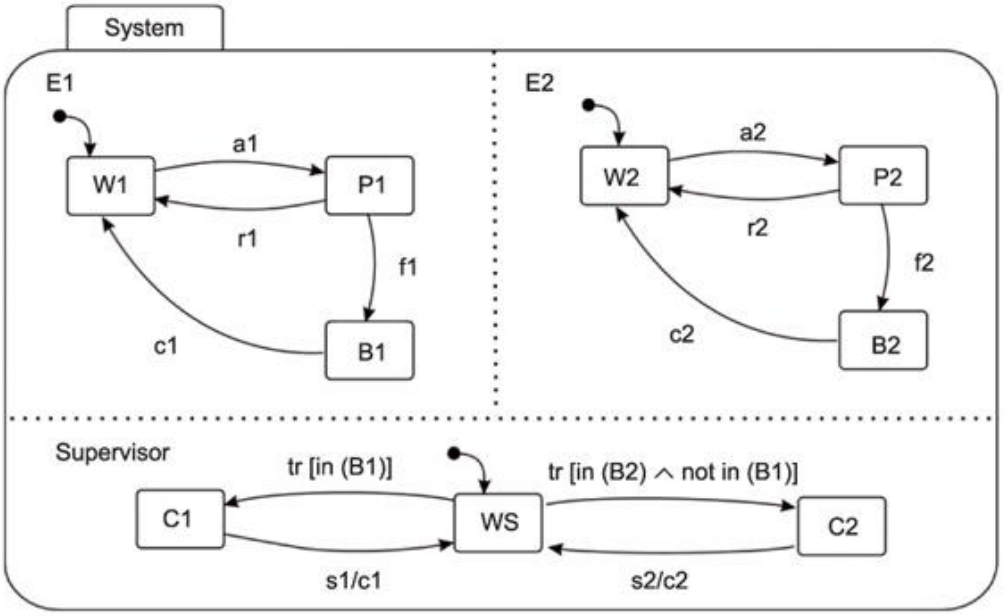


Figura 7 – Especificação em Statecharts de um sistema com duas máquinas e um reparador

3.3.1. PRINCIPAIS MECANISMOS DE STATECHARTS

Os principais mecanismos de modelagem disponibilizados por STATECHARTS são: clustering, refinamento, estado default, entrada-pela-história, concorrência e ações.

3.3.1.1. Clustering

Mecanismo que permite agrupar estados semelhantes em superestados. Permite capturar profundidade e hierarquia. A semântica do superestado criado é um XOR entre os estados internos.

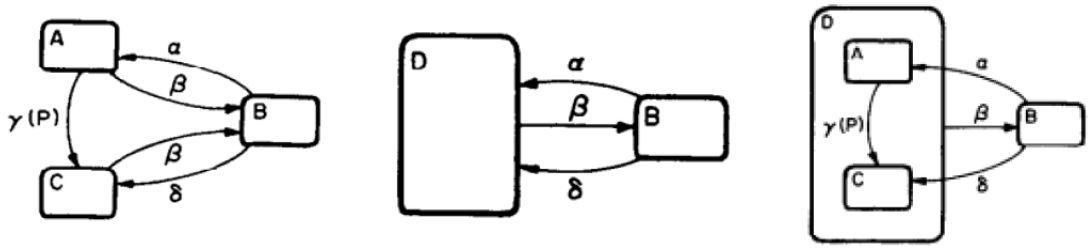


Figura 8 – Clustering.

3.3.1.2. Refinamento

Mecanismo que permite detalhar superestados, ou seja, o processo inverso do clustering. Sua função é analisar e melhorar a compreensão do funcionamento de um estado.

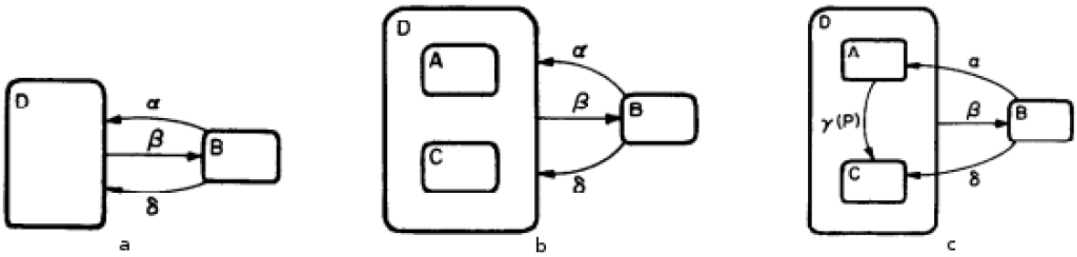


Figura 9 – Refinamento.

3.3.1.3. Estado default

Mecanismo que permite explicitar o estado inicial do sistema. Sua função é específica o estado inicial de cada superestado.

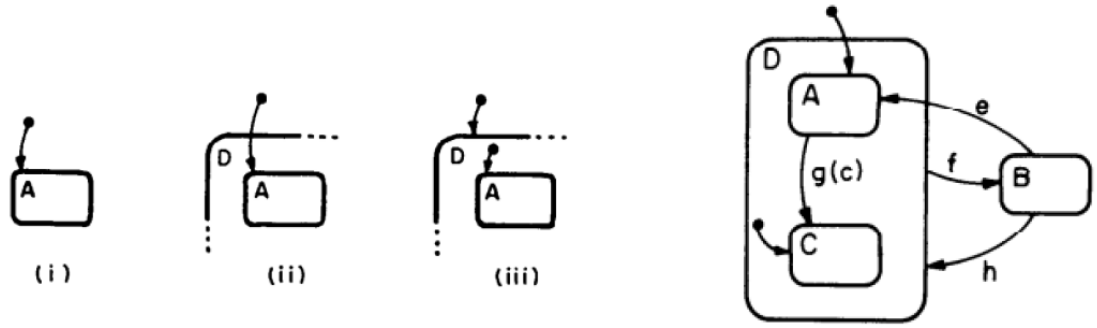


Figura 10 – Estado default.

3.3.1.4. Entrada-pela-história

Ao reentrar em um superestado, o estado mais recentemente visitado é retornado. Possibilita analisar o último estado em diferentes níveis de abstração.

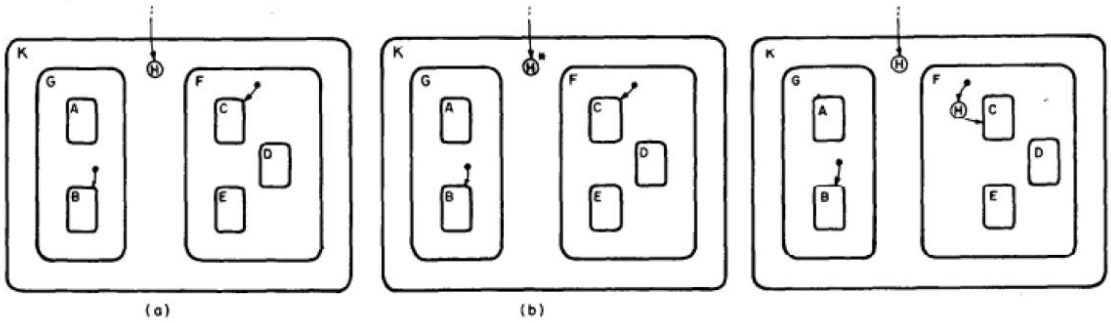


Figura 11 – Entrada pela história.

3.3.1.5. Ortogonalidade

O mecanismo de clustering descreve superestados que internamente possuem um único estado ativo. Porém, em muitos casos, o especificador precisa representar conjuntos de estados concorrentes que utilizem sincronismo ou independência (Paralelismo). Permite simplificar sistemas com estados concorrentes

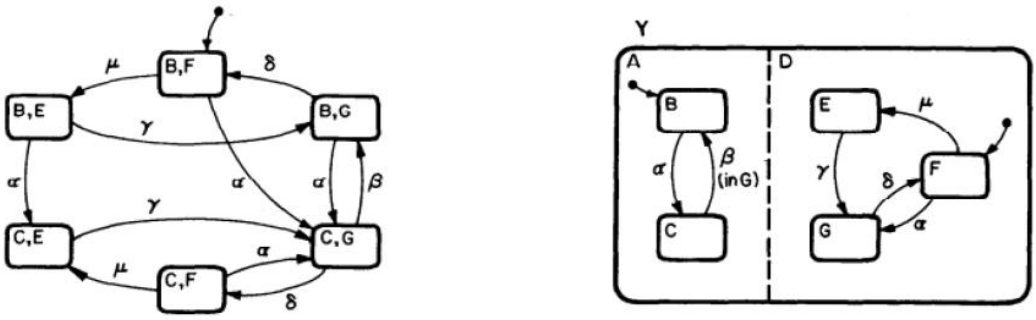


Figura 12 – Ortogonalidade/Paralelismo.

As vantagens são: (i) uma menor quantidade de estados para representar o mesmo sistema, (ii) menor quantidade de transições, (iii) agrupamento de transições semelhantes.

O evento Y representa o produto ortogonal de A e D,porem eles podem resultar em um drástico aumento da complexidade de uma especificação.

3.3.1.6. Entradas de Seleção e Condição

O mecanismo de entradas de condição visa agregar eventos semelhantes e não idênticos, com apenas condições diferentes, reduzindo a complexidade.

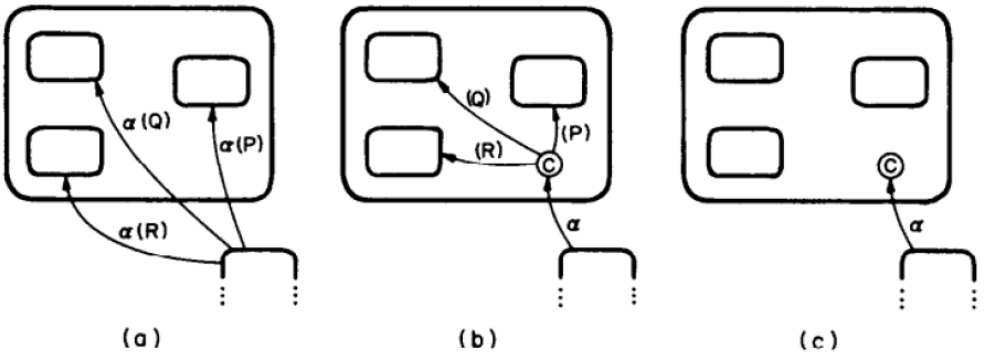


Figura 13 – Entrada de Condição.

O mecanismo de entradas de seleção tem como objetivo agrupar transições simples para a seleção de um valor.

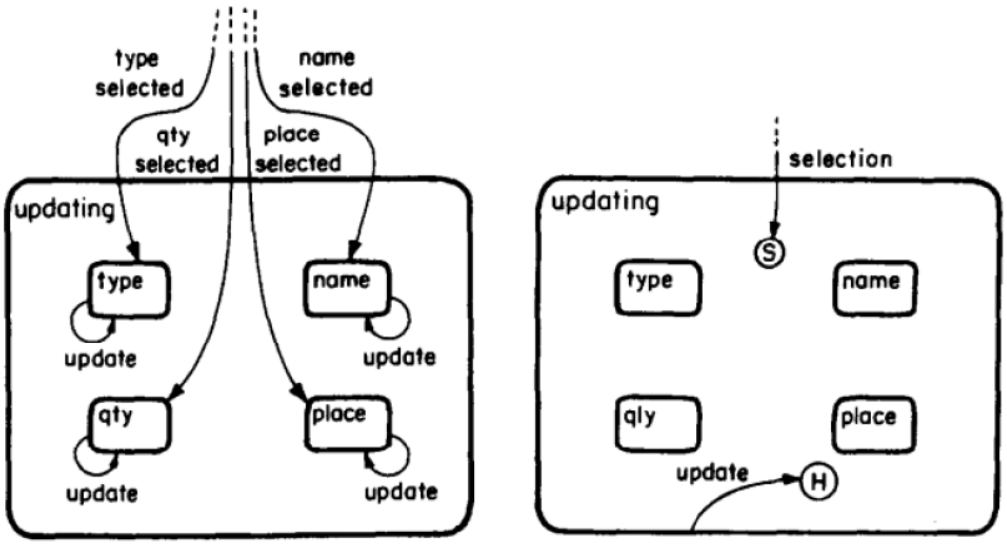


Figura 13 – Entrada de seleção.

3.3.1.7. Delays e Timeouts

É possível modelar tanto delays e timeouts explicitando isso no evento ocorrido. Entretanto, segundo Harel, isso torna a especificação de sistemas altamente complexa, visto que tais eventos ocorrem habitualmente nesses sistemas. É possível especificar tempos diferentes através da sintaxe t1 < t2.

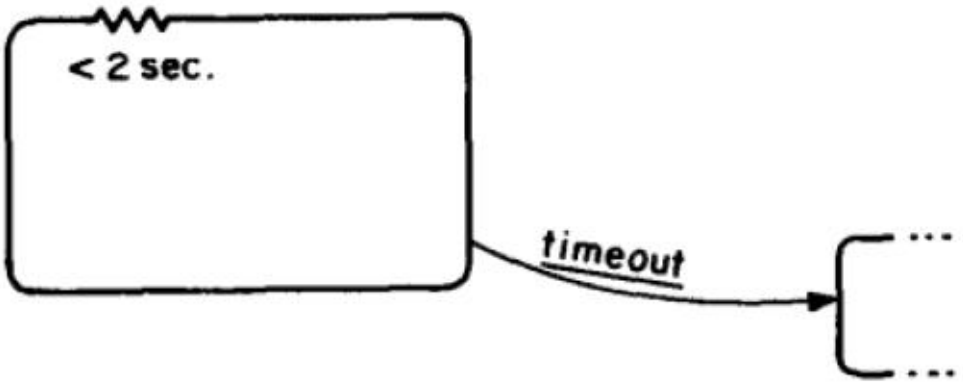


Figura 14 – Delays e Timeouts.

4.) ATIVIDADE 2: ANÁLISE DE MODEL CHECKERS

Os Model Checkers são as ferramentas utilizadas para realizar a verificação formal de modelos. O que diferencia uma ferramenta da outra é basicamente a lógica temporal suportada (CTL, LTL, TCTL, etc.), a linguagem do modelo (SMV, PROMELA, C, etc.), o contraexemplo e se há uma interface gráfica ou apenas comandos.

Dentre mais de 30 Model Checkers, foram analisados 3 ferramentas das mais populares, com o objetivo de escolher a melhor solução para pesquisa. Assim as ferramentas analisadas foram NuSMV, SPIN (SPIN)e UPPAAL. Nas analises, foram levados em consideração requisitos funcionais, tempo real, escalabilidade, usabilidade e interoperabilidade.

4.1) NuSMV

NuSMV é uma reimplementação e extensão do SMV, o primeiro verificador de modelo baseado em BDDs. NuSMV foi projetado para ser uma arquitectura aberta para verificação de modelo em CTL, que pode ser usado de forma confiável para a verificação dos desenhos industriais, como um núcleo para ferramentas de verificação de costume, como um teste para técnicas de verificação formal, e aplicado a outras áreas de pesquisa.

4.2) SPIN

A ferramenta suporta uma linguagem de alto nível para especificar descrições de sistemas chamado PROMELA. Spin tem sido usado para rastrear erros lógicos de design de sistemas distribuídos, como sistemas operacionais, protocolos de comunicação de dados, sistemas de comutação, algoritmos simultâneos, protocolos de sinalização ferroviárias, software de controle para naves espaciais, usinas nucleares, etc. A ferramenta verifica a consistência lógica de uma especificação e relatórios sobre os impasses, as condições de corrida, diferentes tipos de incompletude, e suposições injustificadas sobre as velocidades relativas dos processos.

4.3) UPPAAL

UPPAAL é um ambiente ferramenta integrada para modelagem, simulação e verificação de sistemas de tempo real. É adequado para sistemas que podem ser modelados como uma coleção de processos não-determinísticos com estrutura de controle finito e relógios de valor real, comunicando através de canais ou variáveis compartilhadas. Áreas de aplicação típicas incluem controladores em tempo real e protocolos de comunicação, em particular, aqueles aspectos onde o tempo são críticos.

4.4) Análise

Nos requisitos funcionais e na escalabilidade, o NuSMV destacou-se, pois apresenta Análise de invariante, métodos de particionamento, lógicas CTL e LTL, e pode realizar SAT-based Bounded Model Checking. Já com o requisito de tempo real, o UPPAAL levou vantagens comparado ao NuSMV.

Considerando o requisito de usabilidade, as Tabelas 2 e 3 fazem uma comparação, entre as três ferramentas e os fatores (Tabela 2) e critérios (Tabela 3) de usabilidade, segundo o modelo QUIM [Seffah et al. 2006].

Tabela 2 – Comparação de fatores de usabilidade: Modelo QUIM.

Fatores	Perguntas Relacionadas	(1 a 5)		
		Model Checkers		
		NuSMV	SPIN	Uppaal
1. Eficiência	- O software é capaz de habilitar os usuários a gastar apropriadamente uma quantidade de recursos em relação à eficácia alcançada em um contexto específico uso.	5	4	3
2. Eficácia	- O software permite a realização de tarefas com precisão e perfeição.	5	5	5
3. Produtividade	- O software permite a realização de tarefas em um tempo adequado.	4	5	4
4. Satisfação	- Sinto-me satisfeito com a utilização do software.	5	3	2
5. Capac. Aprendizado	- É fácil aprender os recursos necessários à sua plena utilização.	5	3	2
6. Segurança	- O software garante segurança necessária no que se refere aos usuários e às informações armazenadas, mesmo diante de uma condição anormal de funcionamento.	5	4	3
7. Fiabilidade	- O software é plenamente confiável.	5	5	5
	- Sinto-me confiante em indicar o software a outras pessoas.	5	3	3
8. Acessibilidade	- O software pode ser utilizado por pessoas com algum tipo de limitação (por exemplo, visual, auditiva e psicomotora).	1	1	1
9. Universalidade	- As características do software levam em conta a diversidade de usuários com diferentes origens	1	1	1

	culturais.			
10. Utilidade	- O software resolve os seus problemas de modo aceitável.	5	4	4
	Total	46	38	33

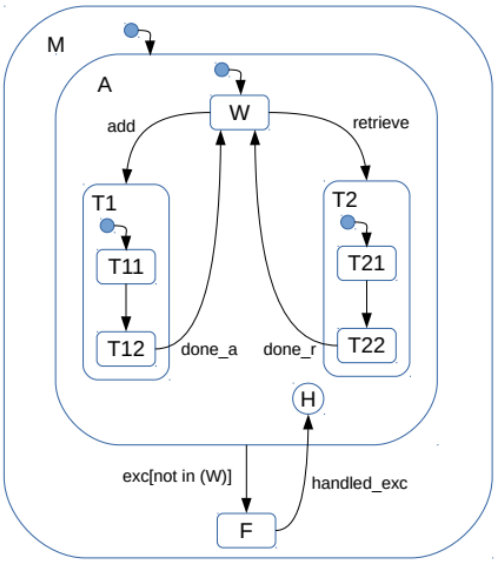
Tabela 3 – Comparação de critérios de usabilidade: Modelo QUIM.

Critérios	Perguntas Relacionadas	(1 a 5)		
		Model Checkers		
		NuSMV	SPIN	Uppaal
1. Atratividade	- Qual software é mais atrativo	5	4	3
2. Flexibilidade	- O software apresenta uma facilidade de ser modificado.	5	4	3
3. Tempo de Carregamento	- O desempenho para verificar modelos com aproximadamente 100.000 estados alcançáveis	3	5	2
	Total	13	13	8

Portando a ferramenta escolhida foi o NuSMV, pois seus contraexemplos e seus traços são mais legíveis e objetivos, e lida com modelos mais complexos apresentando uma maior robustez. Apesar de não ter uma interface gráfica como o SPIN (JSPIN), e o UPPAAL, o NuSMV apresentou fácil inteligibilidade, pois teve um melhor desempenho e aprendizagem nesta ferramenta, e escalabilidade, lidando com modelos bastante complexos. Além disso, o NuSMV também apresenta apreensibilidade e operacionabilidade segundo a ISO 9123, e é eficiente, eficaz e confiável segundo a ISO 9123 e ISO 9241.

5. ATIVIDADES 3 e 5: TRANSFORMAÇÃO DE MODELOS STATECHARTS PARA O MODEL CHECKER SELECIONADO E GERAÇÃO DE CASOS DE TESTE

O método HiMoST será apresentado nesta seção, onde detalhamos os principais algoritmos para traduzir o modelo do Statechart de Harel em uma estrutura geral baseada no idioma do NuSMV Model Checker. Tendo essa



estrutura, é possível gerar o próprio código NuSMV. Como nos referiremos mais adiante, apesar de nos focarmos em um Modelo Checker específico, esta estrutura é suficientemente geral para ser adaptada a outras ferramentas. Ao executar o NuSMV, o T S é gerado. Mostramos também como sugerimos a formalização de propriedades em CTL (Φ) através de um uso combinado do SPS proposto por Dwyer et al. E o algoritmo TTR. A Figura 1 é um exemplo em execução onde o modelo significa duas operações de banco de dados: adicionar e recuperar. Uma exceção (exc) pode ser aumentada quando este sistema está em operação e é devidamente processado (handled_exc) pelo SUT. Observe que temos dois eventos não mostrados na figura: λ1 que faz as transições de T11 para T12 e λ2 que permite a mudança de T21 para T22. Estes são eventos para processar as operações. Além disso, a história rasa (H) está presente no modelo.

Figura 15 – Modelo Statetchart: operações de banco de dados

Nossa abordagem de tradução trata basicamente de todos os recursos do Statecharts de Harel: estados XOR e AND (paralelos), eventos, história rasa e profunda, hierarquia. No entanto, no HiMoST, a principal característica que impulsiona o processo de tradução é a hierarquia. Nós escolhemos a hierarquia porque o grande potencial que Statecharts tem que modelar sistemas complexos ou sistemas de sistemas (SoS) via modularização. O algoritmo principal da tradução é mostrado no Algoritmo 1.

Algorithm 1 The HiMoST method: main algorithm

input: $\Sigma = \{\sigma_i \mid i = 1 \cdots m\}$, $\Delta = \{\delta_i \mid i = 1 \cdots n\}$
output: $P =$ NuSMV structure

1: $VAR \leftarrow createVAR(\Sigma)$
2: $INIT \leftarrow createINIT(VAR)$
3: $NEXT \leftarrow createNEXT(\Sigma, \Delta, VAR)$
4: **return** $P = \langle VAR, INIT, NEXT \rangle$

Figura 16 – Algoritmo 1

No Algoritmo 1, Σ e Δ caracterizam todos os estados e transições do modelo, respectivamente Σ é um conjunto de tuplas em que cada tupla (σi) possui vários elementos, como o nome de um estado (n), o tipo de estado (t = XOR, AND), se este for um estado com histórico relacionado (profundo ou superficial; ht), a hierarquia (h) relacionada a este estado, o conjunto de eventos de saída (E) e alguns outros. O conjunto Δ é um conjunto de tuplas devido às transições onde cada tupla (δi) é composta de elementos como os estados fonte (s) e destino (d), o evento (e) e a hierarquia relacionada à transição (H). Supõe-se que Σ e Δ são de alguma forma obtidos com base no modelo Statechart. A partir deste algoritmo, percebemos que a saída é uma estrutura baseada no NuSMV onde temos os conjuntos VAR, INIT e NEXT que significam a

declaração de variáveis, a inicialização de variáveis e as transições do código, respectivamente.

O procedimento de criação VAR é apresentado no Algoritmo 2. Observe que a saída é um conjunto, VAR, onde cada elemento (V ari) é de fato um conjunto de três outros conjuntos. EvDisph é um conjunto que controla o disparo (valor ativado) ou não dispara (valor desativado) de eventos dentro de cada nível de hierarquia. Statesh é, de fato, outro conjunto com todos os estados dentro de um determinado nível de hierarquia (nota S na linha 7). Por outro lado, Eventsh relaciona-se aos eventos dentro de um determinado nível de hierarquia (nota S na linha 8). Note-se que, se tivermos paralelismo (estados AND), o HiMoST cria conjuntos EvDisph, Statesh e Eventsh para cada estado do estado AND dentro de um nível hierárquico. Na Figura 1, o conjunto de Hierarquia, Olá, contém os seguintes elementos: {M, A, T1, T2}. Então, para todo σi ⊂ Σ, o algoritmo verifica se a hierarquia em σi corresponde a alguma hierarquia h ∈ Hi (veja a linha 6; o script superior h significa o elemento hierárquico da tupla σi). Se for assim, define Statesh e Eventsh são criados com os respectivos estados e eventos. Por exemplo, quando h = A ∈ Hi, então StatesA = {W, T1, T2}, porque estes são os estados dentro do nível de hierarquia A. Por outro lado, EventsA = {add, retrieve, exc}. Aqui, não devemos que exc, também se relaciona com a hierarquia A, porque é possível sair do estado A se uma exceção é gerada quando A está em T1 ou T2.

Algorithm 2 The $createVAR$ procedure

input: $\Sigma = \{\sigma_i \quad | \quad i = 1 \cdots m\}$
output: $VAR = \{Var_i \quad | \quad i = 1 \cdots h\}$

```
1: initializeAllSets()
2: Hi ← identifyHierarchy(Σ)
3: for all h ∈ Hi do
4:     EvDisp_h ← {on ∪ off}
5:     for all σ_i ⊂ Σ do
6:         if σ_i^h = h then
7:             States_h ← ⋃ σ_i^s
8:             Events_h ← ⋃ σ_i^E
9:         end if
10:    end for
11:    Var_i ← {EvDisp_h ∪ States_h ∪ Events_h}
12:    VAR ← VAR ∪ Var_i
13: end for
14: return VAR
```

Figura 17 – Algoritmo 2

O Algoritmo 3 apresenta o procedimento createINIT do nosso método. O objetivo é inicializar todos os conjuntos (variáveis). No início, createINIT

identifica o estado (s1) que tem um evento que pode ser disparado em primeiro lugar (linha 2). Na Figura 1, s1 = W, porque este é o estado inicial em A que possui eventos que podem ser disparados. Assim, o distribuidor de eventos correspondente (EvDisph) é inicializado e todos os outros são inicialmente definidos como desativados (linhas 4-8). Assim, inicialmente, InitEvDispM 1 = desligado, InitEvDispA 2 = ligado, InitEvDispT 1 3 = desligado e InitEvDispT 2 4 = desligado. Os valores iniciais de estados (Statesh) e eventos (Eventsh) são selecionados aleatoriamente. Observe que | INIT | = | VAR | Mas cada elemento InitEvDisph, InitStatesh e InitEventsh é realmente um único elemento e não um conjunto como estava no VAR.

Algorithm 3 The $createINIT$ procedure

input: $VAR = \{Var_i \mid i = 1 \cdots h\}$
output: $INIT = \{Init_i \mid i = 1 \cdots h\}$

1: $initializeAllSets()$
2: $s_1 \leftarrow identifyFirstState(VAR)$
3: **for all** $Var_i \subset VAR$ **do**
4: **if** $Var_i^{States_h} = s_1$ **then**
5: $Init_i^{EvDisp_h} \leftarrow on$
6: **else**
7: $Init_i^{EvDisp_h} \leftarrow off$
8: **end if**
9: $Init_i^{States_h} \leftarrow random(Var_i^{States_h})$
10: $Init_i^{Events_h} \leftarrow random(Var_i^{Events_h})$
11: $INIT \leftarrow INIT \cup \{Init_i^{EvDisp_h} \cup Init_i^{States_h} \cup Init_i^{Events_h}\}$
12: **end for**
13: **return** $INIT$

Figura 17 – Algoritmo 3

O último passo do nosso método refere-se à definição das transições (set NEXT) na estrutura NuSMV. Algoritmo 4 apresenta o procedimento createNEXT. A entrada deste procedimento são os conjuntos Σ, Δ e V AR já definidos. Note que trabalhamos basicamente com o mesmo raciocínio já apresentado para a criação do V AR. Em outras palavras, definimos conjuntos de tuplas para cada distribuidor de eventos dentro de uma hierarquia (EV Dh), transições de estado (STh) e mudanças de eventos (EVh). Cada elemento em EV Dh, STh e EVh é uma tupla da forma hevd, s, e, vi, onde evd é um expedidor de eventos, s é um estado, e é um evento e v é o próximo valor de um certo variável. Todos esses elementos se referem ao mesmo nível hierárquico, h. A idéia básica de interpretar cada uma das tuple hevd, s, e, vi ao

gerar as próximas declarações no código NuSMV é a seguinte: em outras palavras, criamos uma próxima declaração em que temos um predicado evd ∧ s ∧ e que irá Resulte no próximo valor de uma variável (v) quando é avaliado como verdadeiro. Devemos observar que, então, criamos um código NuSMV padronizado dessa maneira. O procedimento identif yActivateDeactiveDisp (linha 4) visa identificar quais eventos ativar ou desativar um determinado expedidor dentro de um nível de hierarquia. Assim, eles mostram quais eventos, e, permitem entrar ou voltar a um determinado nível de hierarquia para que eventos de tal hierarquia possam ser desencadeados nas etapas consequentes. O resultado são dois conjuntos, um com os eventos para ativar (Da) e outro com eventos para desativar (Dd) os despachantes. Consideremos a Figura 1 e a hierarquia T1 (h = T1). Por isso, DaT 1 = add, handle exc. Porque, a partir do estado W, o sistema passa para T1 através do evento e volta para T1 quando retorna do estado F depois de manusear uma exceção (ex manipulado) se o sistema estiver em T11 Ou T12 quando ocorreu a exceção. Isso ocorre porque o histórico superficial (H) no modelo. Precisamente, o sistema retorna ao T11, o estado inicial de T1. Do mesmo modo, DdT 1 = {done a, exc} porque estes são os eventos que fazem o sistema sair da hierarquia T1. Depois que Da e Dd são preenchidos, cada tupla em EV Dh pode então ser criada. Observe que as tuplas estão na forma hV arEvDisph = on, -, da, on ou hVarEvDisph = on, -, dd, offi. Importante mencionar que todos os valores no conjunto EvDisph são considerados dentro dos predicados. O traço "-" significa que a contribuição do estado para o (s) predicado (s) não está presente. Na verdade, os estados são apenas considerados em alguns predicados da próxima declaração, a fim de lidar com situações muito específicas, e. História (profunda, rasa). A forma de predicado mais comum inclui apenas evd e e. Devido às restrições de espaço, decidimos não mostrar o procedimento createNEXT com muito detalhe escolhendo descrevê-lo em sua forma mais comum. Considere h = M na Figura 1. Um elemento em EV DM é hV arEvDispT 1 = em 3, -, exc, oni. Traduzindo esta tupla para um predicado de declaração seguinte para a variável V arEvDispM, isso significa que: se o agente de eventos ativo atual é T1 e o evento a ser disparado dentro de T1 é exc, o próximo valor de VarEvDispM está ativado. Mais especificamente, se o sistema estiver em T1 (T11 ou T12) e uma exceção acontece (exc), o modelo deixa hierarquias T1, A e a próxima hierarquia ativa será M (on). O procedimento de recuperação de identificação e estado (linha 11) leva o conjunto de transições (Δ) como entrada e apenas pesquisa esse conjunto para perceber sobre valores atualizados para variáveis do tipo V arStatesh. O conjunto St consiste em mapeamentos e 7 → v dizendo qual será o próximo estado (v) se e for disparado. Assim, o novo valor, v, da variável é δ d i ∈ δi ⊂ Δ, isto é, é o estado de destino do conjunto δi. Considere h = T1 na Figura 1. Um elemento no STT 1 é hV arEvDispT 1 = em 3, -, λ1, T12. Traduzindo esta tupla para um predicado de declaração seguinte para a variável V arStatesT 1 significa a transição T11 7 → λ1 7 → T12 no modelo de Statechart original. Para identificar mudanças de eventos (procedimento de identificação de

eventos na linha 15), o conjunto de transições (Δ) também é tomado como entrada. Cada elemento em Ec é um mapeamento ei 7 → ej indicando qual será o próximo evento a ser acionado (ej) considerando o evento atual (ei). Neste caso, é possível que o ej seja um conjunto porque dois eventos podem ser disparados em um determinado estado s. Por exemplo, se h = A, um elemento em EVA é hV arEvDispT 1 = em 3, -, feito a, {add, retrieve} i. Depois de processar uma operação de adicionar (feito a), os eventos adicionados ou recuperados podem ser disparados no estado W. O Modelo de Verificador escolhe-o de forma não determinista. Considerando o modelo do Statechart na Figura 1, o conjunto de variáveis (V AR) é composto de 4 conjuntos (V ari) que, por sua vez, cada V ari é a união de 3 outros conjuntos: EvDisph, Statesh e Eventsh. No total, temos 12 conjuntos, que serão 12 variáveis no código NuSMV (após a etapa de implementação). Cada uma dessas 12 variáveis terá uma próxima declaração associada e, portanto, | NEXT | = 12.

Algorithm 4 The $createNEXT$ procedure

input: Σ, Δ, VAR
output: $NEXT = \{Next_i \quad | \quad i = 1 \cdots h\}$

1: $initializeAllSets()$
2: $Hi \leftarrow identifyHierarchy(\Sigma)$
3: **for all** $h \in Hi$ **do**
4: $\quad [Da, Dd] \leftarrow identifyActivateDeactivateDisp(VAR)$
5: $\quad$ **for all** $da \in Da$ **do**
6: $\quad\quad EVD_h \leftarrow EVD_h \cup \langle Var_i^{EvDisp_h=on}, -, da, on \rangle$
7: $\quad$ **end for**
8: $\quad$ **for all** $dd \in Dd$ **do**
9: $\quad\quad EVD_h \leftarrow EVD_h \cup \langle Var_i^{EvDisp_h=on}, -, dd, off \rangle$
10: $\quad$ **end for**
11: $\quad St \leftarrow identifyStateTransitions(\Delta)$
12: $\quad$ **for all** $st \in St$ **do**
13: $\quad\quad ST_h \leftarrow ST_h \cup \langle Var_i^{EvDisp_h=on}, -, st, \delta_i^d \rangle$
14: $\quad$ **end for**
15: $\quad Ec \leftarrow identifyEventChanges(\Delta)$
16: $\quad$ **for all** $ec \in Ec$ **do**
17: $\quad\quad EV_h \leftarrow EV_h \cup \langle Var_i^{EvDisp_h=on}, -, ec, next(\delta_i^e) \rangle$
18: $\quad$ **end for**
19: $\quad Next_i \leftarrow \{EVD_h \cup ST_h \cup EV_h\}$
20: $\quad NEXT \leftarrow NEXT \cup Next_i$
21: **end for**
22: **return** $NEXT$

Figura 18 – Algoritmo 4

A outra contribuição de nosso método é uma maneira sistemática de formalizar propriedades através de um SPS [8] e projetos combinatórios baseados no algoritmo TTR [5, 6]. Acreditamos que este é um passo importante para fins práticos, porque os profissionais exigem diretrizes para aplicar uma teoria adequada aos seus projetos de software. A forma sistemática para formalizar propriedades CTL é apresentada no Algoritmo 5, incluído nas etapas para gerar o conjunto de teste, T.

Algorithm 5 Properties formalization and test case generation

input: P = NuSMV structure, C = NuSMV code
output: T = Test Suite

1: Consider all events, e, in the original Statechart model by inspecting $Var_i^{Events_h} \subset VAR \subset P$
2: Split equally all such events in three sets: S_1, S_2, S_3
3: Run the TTR algorithm considering sets S_1, S_2 and S_3 and $strength = 2$. A Mixed-level Covering Array, $M_{|m|\times 3}$, is the TTR's output
4: Define the SPS's Pattern and Scope set, Ψ
5: **for all** $m \subset M$ **do**
6: **for all** $\psi \in \Psi$ **do**
7: Formalize a CTL property, Φ, according to ψ by replacing the formulas with the elements of m
8: **end for**
9: **end for**
10: Run the Model Checker with all Φ against C
11: Realize about the counterexamples whose size are greater than a threshold, τ
12: Generate an initial test suite, T_i, based on such counterexamples
13: Eliminate possible redundant test cases in T_i, and generate the final test suite, T
14: **return** T

O primeiro ponto a comentar é que propomos apenas os eventos (e 2 V arEventsh i) do modelo Statechart original para formalizar as propriedades. Outros estudos geram propriedades levadas em consideração uma combinação de diferentes tipos de variáveis ou mesmo conjuntos criados devido a certos critérios de cobertura [17]. Como os eventos são os elementos básicos para estimular o SUT, decidimos considerar apenas eventos em nossas fórmulas sem uso correspondente de outras variáveis (conjuntos). Acreditamos que esta solução simplifica a geração de fórmulas porque precisamos nos concentrar apenas em uma combinação de eventos para gerar fórmulas para forçar a derivação de contra-exemplos. Mesmo assim, a quantidade de eventos pode resultar em um grande número de combinações e isso pode ser inviável na prática. Assim, desenhos combinatórios como na

imagem para evitar um número excessivo de combinação de eventos. Esses métodos provaram ser úteis para o teste de software, gerando conjuntos de teste de menor custo [6]. Então, dividimos todos os eventos em 3 conjuntos, S1, S2 e S3 (linha 2), e executamos o algoritmo TTR considerando esses conjuntos e o grau de interação (força) é igual a 2 (linha 3). O resultado da TTR é uma matriz de cobertura de nível misto, Mjmj? 3. Consideremos a Figura 1 novamente. Portanto, temos S1 = fexc; Manipulado exc; Addg, S2 = fretrieve; ? 1; Feito ag, S3 = f? 2; Feito rg. Correndo TTR, temos os resultados apresentados na Tabela 1.

Table 1: TTR's results: M

m	S_1	S_2	S_3
1	*exc*	*retrieve*	λ_2
2	*exc*	λ_1	λ_2
3	*exc*	*done_a*	*done_r*
4	*handled_exc*	*retrieve*	λ_2
5	*handled_exc*	λ_1	*done_r*
6	*handled_exc*	*done_a*	λ_2
7	*add*	*retrieve*	*done_r*
8	*add*	λ_1	λ_2
9	*add*	*done_a*	λ_2

SPSs são muito importantes para fins práticos. Eles fornecem modelos de fórmula uma vez que um padrão e o escopo do padrão são identificados por um profissional com base nos requisitos do SUT. Atualmente, existem 9 padrões e o padrão de 5 padrões para CTL. Sugerimos a seguinte combinação de padrão de patentes / patern, conforme mostrado abaixo:

1. Absence/Global (ABS). CTL: $\forall\Box(\neg P)$;
2. Response Chain (S, T responds to P)/Global (REC). CTL: $\forall\Box(P \rightarrow \forall\Diamond(S \wedge \forall \bigcirc (\forall\Diamond(T))))$;
3. Precedence Chain (P precedes S, T)/Global (PC1). CTL: $\neg\exists[\neg P \cup (S \wedge \neg P \wedge \exists \bigcirc (\exists\Diamond(T)))]$;
4. Precedence Chain (S, T precedes P)/Global (PC2). CTL: $\neg\exists[\neg S \cup P] \wedge \neg\exists[\neg P \cup (S \wedge \neg P \wedge \exists \bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))]$.

Nas fórmulas CTL acima, temos os quantos de caminho para todos os caminhos e para algum caminho. Além disso, há as modalidades temporais sempre, eventualmente, próximas e até. Naturalmente, um designer de teste pode escolher apenas um ou todo esse escopo padrão / padrão ou mesmo usar outra combinação. Mas, acreditamos que essas quatro sugestões são adequadas devido à cadeia de eventos que eles relacionaram.

No início, nosso conjunto de testes possui 9 casos de teste (m = 9). Mas, na verdade, nem todas as fórmulas CTL produzirão um contra-exemplo. Além disso, alguns contraexemplos podem ter apenas um único (o estado inicial) ou poucos estados. Por isso, o limiar? (Linha 11) serve para descartar contra-exemplos muito curtos que, na prática, não tem sentido executar. Finalmente, é provável que alguns contraexemplos sejam precisamente iguais. Então, desejei eliminar possíveis casos de teste redundantes para obter o conjunto de teste nal, T. Devemos enfatizar que existe uma certa aleatoriedade na definição dos conjuntos S1, S2 e S3 e, conseqüentemente, gerar as fórmulas de CTL . Assim, algumas propriedades CTL podem parecer inadequadas porque exigem uma certa sequência de estímulos para o SUT que não estão de acordo com os requisitos do sistema. Mas precisamos lembrar que a idéia é forçar o Model Checker a gerar contra-exemplos. Por isso, esta aleatoriedade é interessante, precisamente, para mostrar que o SUT não apresenta alguns comportamentos \ nenhum sentido \. A conclusão é que um conjunto de testes gerado dessa maneira pode ser útil não apenas para testes de conformidade, mas também para testes de robustez. Considerando os 4 padrões / padrões que nós propomos,? = 2 (ou seja, o tamanho dos contra-exemplos mais de 2 para considerar como um caso de teste), a eliminação de casos de teste redundantes e a Fase 1, o conjunto de teste T de acordo com o método HiMoST é como mostrado abaixo:

$$
\begin{aligned}
T = \{&\{add, \lambda_1, done\_a, exc\},\\
&\{add, \lambda_1, exc, handled\_exc, \lambda_1, exc, handled\_exc\},\\
&\{add, \lambda_1, done\_a, add\},\\
&\{add, \lambda_1, exc, handled\_exc, \lambda_1\},\\
&\{add, \lambda_1, done\_a, retrieve, \lambda_2\},\\
&\{add, \lambda_1, done\_a, retrieve, \lambda_2, done\_r\},\\
&\{add, \lambda_1, done\_a, retrieve\}\}.
\end{aligned}
$$

Observe que foram gerados 7 casos de teste (cada caso de teste é incluído por {...}). Neste trabalho, descrevemos um caso de teste (tc) como: tc = {tsi | i E N\{0}}, onde tc = caso de teste e tsi = passo de teste i. Um passo de teste, tsi, é uma atividade atômica para preparar ou estimular o SUT. O estímulo contém os dados de entrada de teste e, possivelmente, os resultados esperados (em alguns casos, não há resultados esperados explícitos). Em outras palavras, um caso de teste é uma série de etapas de teste. É importante ressaltar que os primeiros 4 casos de teste foram devidos à Cadeia de Resposta (S, T responde a P) / Global (REC) e os 3 últimos devido à Cadeia de Precedência (P precede S, T) / Global (PC1) . Os outros dois padrões geraram nenhum caso de teste.

6. ATIVIDADES 4, 6 e 7: INCORPORAÇÃO DO MÉTODO HiMoST À FERRAMENTA SOLIMVA E ATUALIZAÇÃO DE GUIs

Paralelamente a atividade 3, a abordagem composicional StatNuSMV está sendo implementada/incorporada à ferramenta SOLIMVA. A implementação/codificação da abordagem composicional StatNuSMV resultou em duas perspectivas na ferramenta SOLIMVA:

a) NATURAL LANGUAGE (NL). Nessa perspectiva, a nova versão da ferramenta SOLIMVA, versão 1.1, toma como entrada a notação textual da versão anterior (1.0), notação esta que representa um modelo Statechart gerado a partir de requisitos criados em Linguagem Natural. Então, a abordagem composicional transforma essa notação textual, que representa um modelo Statechart, para o código-fonte do Model Checker NuSMV. Porém, os elementos AND e History (Deep/Shallow) não são contemplados nessa perspectiva NL. O Model Checker NuSMV é executado de dentro da própria ferramenta SOLIMVA. Portanto, a versão 1.1 da ferramenta SOLIMVA possui a implementação da perspectiva NL;

b) BEHAVIOR (BEH). Nessa perspectiva, o usuário cria uma modelagem comportamental em Statecharts por meio de uma Interface Gráfica do Utilizador (GUI - Graphical User Interface). Então, a versão 1.2 da ferramenta SOLIMVA, automaticamente, transforma essa notação gráfica em State Chart XML (SCXML), uma recomendação do W3C. A partir do SCXML, a abordagem composicional transforma o modelo Statechart para o código-fonte do NuSMV. Todos os elementos da abordagem composicional são contemplados. O Model Checker NuSMV é executado de dentro da própria ferramenta SOLIMVA. A versão 1.2 da ferramenta SOLIMVA, portanto, possui a implementação das perspectivas NL e BEH.

A Figura 18 mostra parte da estrutura de pacotes do código-fonte da SOLIMVA 1.2, onde pode-se ver o novo pacote, **statechartts**, onde está sendo implementada a abordagem composicional.

A classe Reader lê o arquivo *model_hierarchy-final.bhm*, saída da ferramenta SOLIMVA 1.0 e onde o modelo Statechart está representado, e cria nós de estados interligados, passando os parâmetros para a classe Converter, onde as propriedades do modelo Statechart são identificadas. Na classe Converter também é feita a tradução para a linguagem do Model Checker NuSMV.

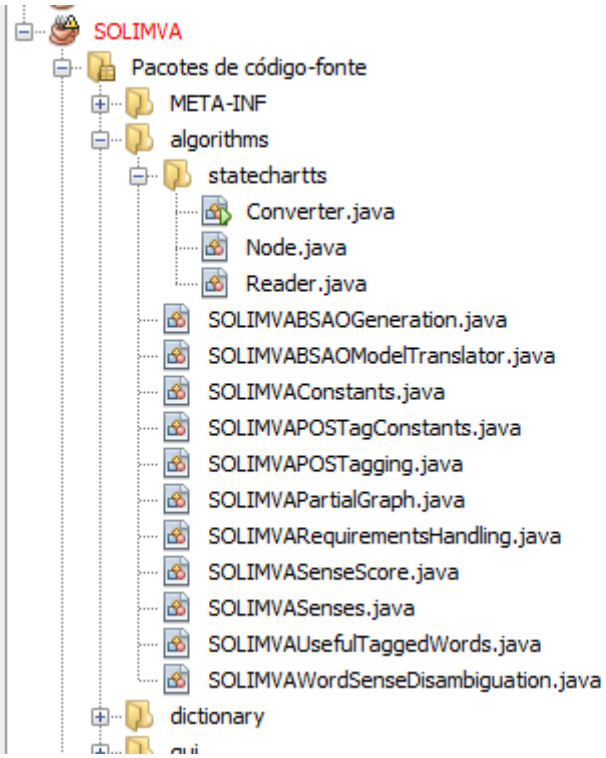


Figura 18 – SOLIMVA 1.2: parte da estrutura de pacotes do código-fonte.

A título de exemplificação, considere o modelo Statechart apresentado na Figura 19.

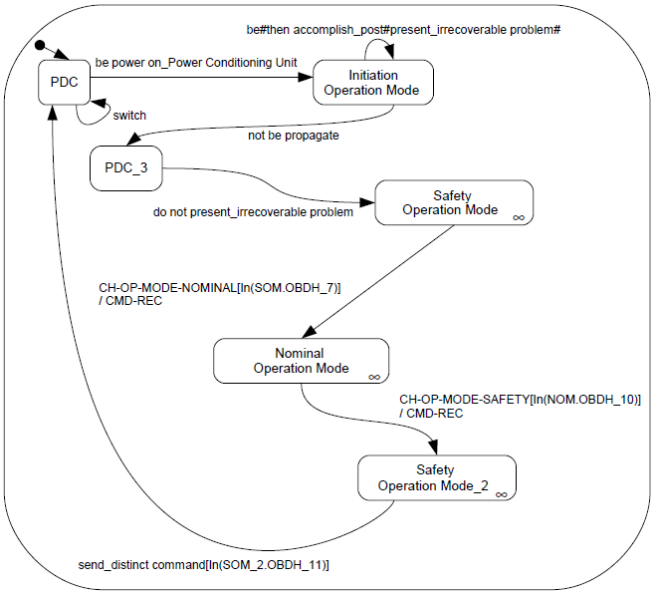


Figura 19 – Statechart gerado pela SOLIMVA. Adaptado de [Santiago Júnior 2011]

Parte do arquivo *model_hierarchy-final.bhm*, que representa esse modelo Statechart, está mostrado a seguir.

```
Src State: PDC - Inp Ev Trans: 1-be 2-power 2-on_Power Conditioning Unit - Out Ev Trans: null - Dest State: Initiation Operation Mode

      Src State: Initiation Operation Mode - Inp Ev Trans: 3-be#6-then 7-accomplish_post#9-present_irrecoverable problem# - Out Ev Trans: null - Dest State: Initiation Operation Mode

      Src State: Initiation Operation Mode - Inp Ev Trans: 15-not 16-be 17-propagate - Out Ev Trans: null - Dest State: PDC_3

      Src State: PDC_3 - Inp Ev Trans: 18-do 19-not 20-present_irrecoverable problem - Out Ev Trans: null - Dest State: Safety Operation Mode
```

Portanto, é baseado nessa representação que a classe Reader cria os nós de estados interligados.

As atividades 6 e 7 têm sido feitas concomitantemente. A atividade 6, que se refere a incorporação de técnica para gerar de casos de teste via Model Checking à ferramenta SOLIMVA, está sendo realizada via melhoria na GUI da SOLIMVA de forma a contemplar todos os padrões e escopos de padrões definidos nos padrões de especificação [Dwyer et al. 1999]. Ao mesmo tempo, a atividade 7 está sendo realizada pela atualização da GUI da SOLIMVA.

A Figura 20 mostra a aba StatSMV, incorporada à versão 1.2 da ferramenta SOLIMVA, onde podem ser vistos o código-fonte do NuSMV gerado a partir de SCXML (perspectiva BEH), e os padrões e escopos de padrão mapeados na GUI. A Figura 21 mostra, com mais clareza, o mapeamento de padrões e escopos de padrão.

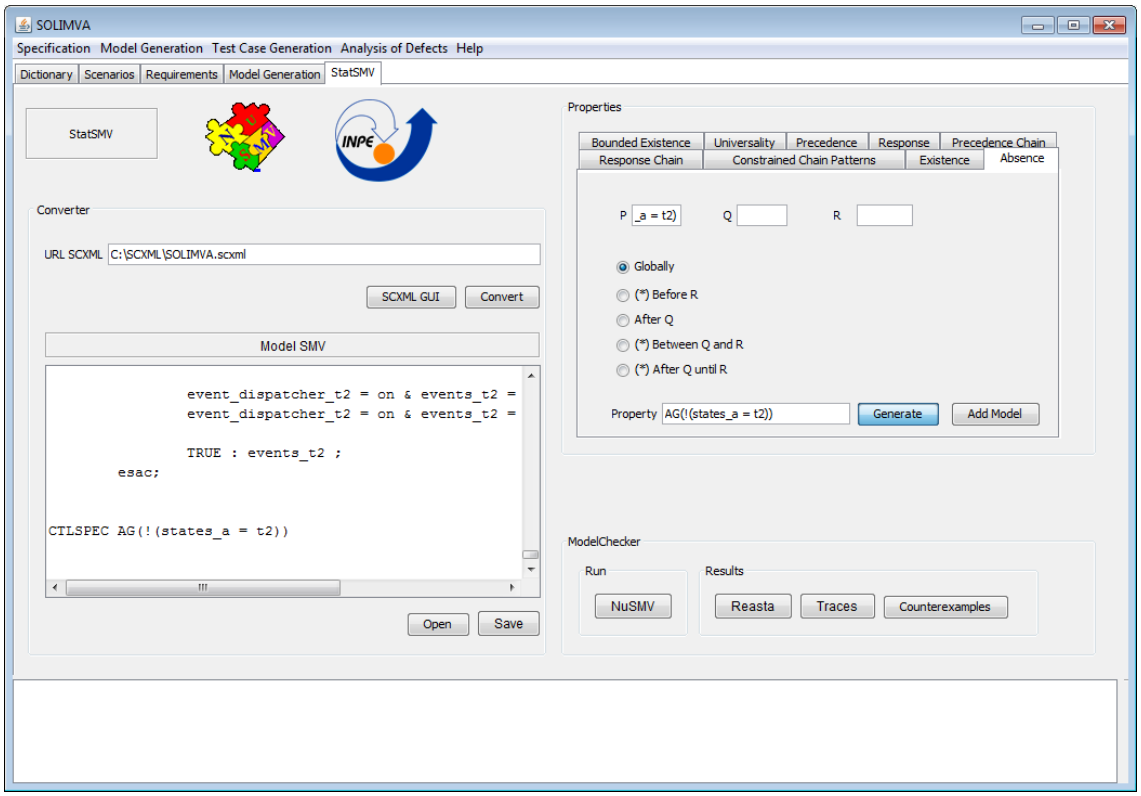


Figura 20 – SOLIMVA 1.2: Nova aba StatSMV

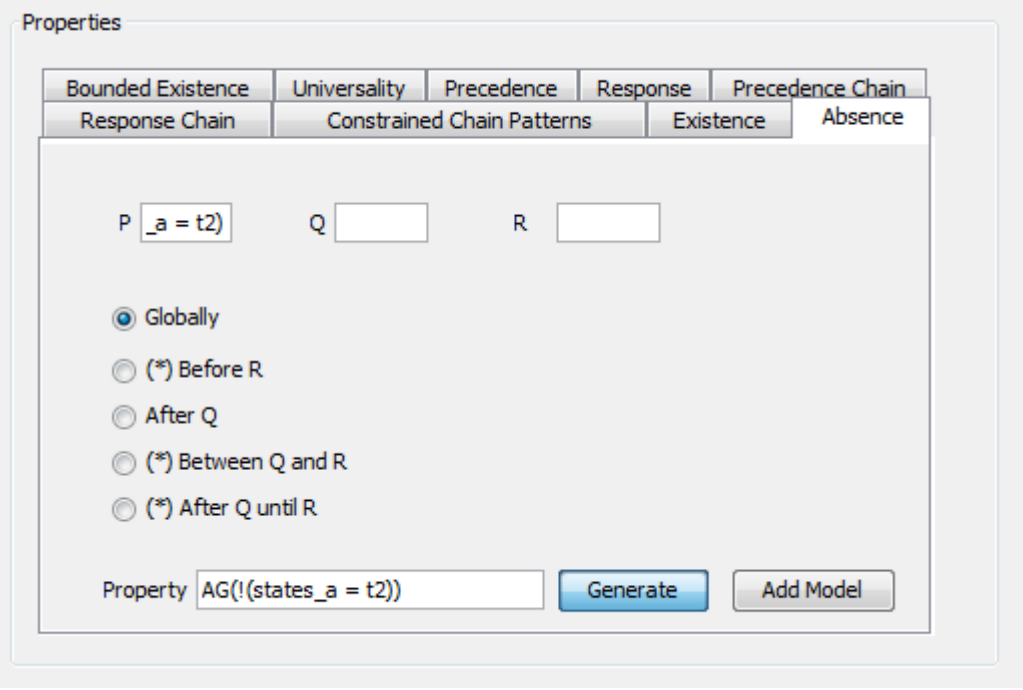


Figura 21 – SOLIMVA 1.2: padrões e escopos de padrão

7. ATIVIDADE 8: AVALIAÇÃO EXPERIMENTAL

Esta avaliação empírica multiobjectivo visa avaliar juntas duas características, custo e eficiência, da suíte de teste geradas através de cada um dos padrões.

Por isso, ao invés de usar todos esses padrões para derivar um conjunto de teste exclusivo, queremos avaliar quais desses padrões têm melhor desempenho, eficiência, se os considerarmos sozinhos para gerar um conjunto de testes.

O custo é definido como a quantidade total de etapas de teste (#ts) de todos os casos de teste de um conjunto de testes. Acreditamos que esta é uma definição de custo mais realista porque um caso de teste, tc1, poderia ter associado algumas etapas, digamos 5 e, por exemplo, um segundo caso de teste, tc2, pode ser composto de 100 etapas de teste. Definimos eficiência de acordo com três Perspectivas. Primeiro, examinamos a proporção de transições coberto no Statecharts original desenvolvido para uma amostra (estudo de caso). Em segundo lugar, examinamos a cobertura das instruções e, finalmente, a cobertura das filiais ao iniciar o conjunto de testes contra a amostra.

8. CONCLUSÃO

Esse relatório apresentou as atividades desenvolvidas, no período de 01 de agosto de 2015 a 13 de julho de 2017, relacionadas ao projeto “Testes de Software via Model Checking para Sistemas Espaciais Críticos”. Esse, portanto, é o relatório final do projeto. Conforme descrito nesse relatório, os objetivos dessa pesquisa foram completamente alcançados, inclusive com publicação de artigo em renomada conferência na área de teste de software.

9. REFERÊNCIAS

[Santiago Júnior e Silva 2017] Santiago Júnior, Valdivino Alexandre de; Silva, Felipe Elias Costa da. From Statecharts to Model Checking: A Hierarchy-based Translation and Specification Patterns Properties to Generate Test Cases. In: The 2nd Brazilian Symposium, 2017, Fortaleza, CE, Brazil. Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing - SAST (Accepted for publication).

[Balera e Santiago Júnior 2016] Balera, Juliana Marino ; Santiago Júnior, Valdivino Alexandre de. A Controlled Experiment for Combinatorial Testing. In: The 1st Brazilian Symposium, 2016, Maringa. Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing - SAST. p. 1-10.

[Delamaro et al. 2007] M. E. Delamaro, J. C. Maldonado, and M. Jino. Introdução ao teste de software. Campus-Elsevier, 2007. 408 p.

[Baier e Katoen 2008] BAIER, C.; KATOEN, J.-P. Principles of model checking. Cambridge, MA, USA: The MIT Press, 2008.

[Clarke e Emerson 2008] CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In: GRUMBERG, O.; VEITH, H. (Ed.). 25 years of model checking. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2008. v. 5000, p. 196-215. Lecture Notes in Computer Science (LNCS).

[Dwyer et al. 1999] DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specications for finite-state verication. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 21., 1999, Los Angeles, CA, 1999. p. 411-420.

[Fraser et al. 2009] FRASER, G.; WOTAWA, F.; AMMANN, P. E. Testing with model checkers: a survey. Software Testing, Verification and Reliability, v. 19, p. 215–261, 2009.

[Ganai e Gupta 2007] GANAI, M.; GUPTA, A. SAT-Based scalable formal verification solutions. New York, NY, USA: Springer Science+Business Media, 2007.

[Kroening et al. 2015] Kroening, D.; Lewis, M.; Weissenbacher, G. Proving Safety with Trace Automata and Bounded Model Checking. FM 2015: Formal Methods, v. 9109, p. 325-341, 2015, Lecture Notes in Computer Science (LNCS).

[Navigli 2009] NAVIGLI, R. Word sense disambiguation: A survey. ACM Computing Surveys, v. 41, n. 2, p. 1-69, 2009.

[NASA 2008] NATIONAL AERONAUTICS AND SPACE ADMINISTRATION. NASA/SP-2008-565: Columbia Crew Survival Investigation Report. USA: NASA, 2008.

[Pasareanu et al. 2013] PASAREANU, C.; VISSER, W.; BUSHNELL, D.; GELDENHUYS, J.; MEHLITZ, P.; RUNGTA, N. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering, v. 20, p. 391–425, 2013.

[Queille e Sifakis 2008] QUEILLE, J.-P.; SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In: GRUMBERG, O.; VEITH, H. (Ed.). 25 years of model checking. Berlin/Heidelberg, Germany: Springer Berlin/Heidelberg, 2008. v. 5000, p. 216-230. Lecture Notes in Computer Science (LNCS).

[Santiago Júnior 2011] SANTIAGO JÚNIOR, V. A. SOLIMVA: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications. 2011. 264 p. Thesis (Doctorate at Post Graduation Course in Applied Computing) - Instituto

Nacional de Pesquisas Espaciais, São José dos Campos, SP, Brazil, 2011. Avaliable from: <http://urlib.net/8JMKD3MGP7W/3AP764B>. Access in: Feb. 06, 2014.

[Santiago Júnior e Vijaykumar 2012] SANTIAGO JÚNIOR, V. A.; VIJAYKUMAR, N. L. Generating model-based test cases from natural language requirements for space application software. Software Quality Journal, v. 20, n. 1, p. 77-143, 2012. DOI: 10.1007/s11219-011-9155-6.

[Santiago Júnior et al. 2012] SANTIAGO JÚNIOR, V. A.; VIJAYKUMAR, N. L.; FERREIRA, E.; GUIMARÃES, D.; COSTA, R. C. GTSC: Automated Model-Based Test Case Generation from Statecharts and Finite State Machines. In: Sessão de Ferramentas do III Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2012, Natal-RN. Anais do III Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2012. p. 25-30.

[Toutanova et al. 2003] TOUTANOVA, K.; KLEIN, D.; MANNING, C. D.; SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In: CONFERENCE OF THE NORTH AMERICAN CHAPTER OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS ON HUMAN LANGUAGE TECHNOLOGY, 2003, Edmonton, Canada. 2003. p. 173-180.

[Utting e Legeard 2007] UTTING, M.; LEGEARD, B. Practical Model-Based Testing: A tools Approach. Waltham, MA, USA: Morgan Kaufmann Publishers, 2007.

[Harel 1987] Harel, D. Statecharts: a visual formalism for complex systems. Science of Computer Programming, v. 8, p. 231-274, 1987.

[Rumbaugh et al. 1991] RUMBAUGH, J. et al. Object-oriented modeling and design. Englewood Cliffs: Prentice-Hall, 1991.

[Santos 2004] Santos, O. M. Verificação Formal de Sistemas Distribuídos Modelados na Gramática de Grafos Baseada em Objetos. Faculdade de Informática, PPGCC, 2004.

[NuSMV 2015a] Cavada, R.; Cimatti, A.; Jochim, C. A.; Keighren, G.; Olivetti, E.; Pistore, M.; Roveri, M.; Tchaltsev, A. “NuSMV 2.5 User Manual”, http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf, 2015.

[NuSMV 2015b] Cavada, R.; Cimatti, A.; Keighren, G.; Olivetti, E.; Pistore, M.; Roveri, M. “NuSMV 2.5 Tutorial”, http://nusmv.fbk.eu/NuSMV/tutorial/v25/tutorial.pdf, 2015.

[Holzmann 2003] G. J. Holzmann. The SPIN model checker. Addison-Wesley Professional, USA, 2003.608 p.

[Behrmann et al. 2004] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, Formal methods for the

design of real-time systems, volume 3185, pages 200-236. Springer Berlin/Heidelberg, Berlin/Heidelberg, Germany, 2004. Lecture Notes in Computer Science (LNCS).

[Seffah et al. 2006] A. Seffah, M. Donyaee, R. B. Kline, and H. K. Padda. Usability measurement and metrics: A consolidated model. Software Quality Journal, (2006) 14: 159–178.