TESTES DE SOFTWARE VIA MODEL CHECKING PARA SISTEMAS ESPACIAIS CRÍTICOS

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA (RENOVAÇÃO)

(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 (LAC/CTE/INPE)

E-mail: valdivino.santiago@inpe.br

Junho de 2016

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 30 de junho de 2016.

.

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 relacionadas 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, provavelmente, a técnica mais adotada, na prática, entre todas relacionadas 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].

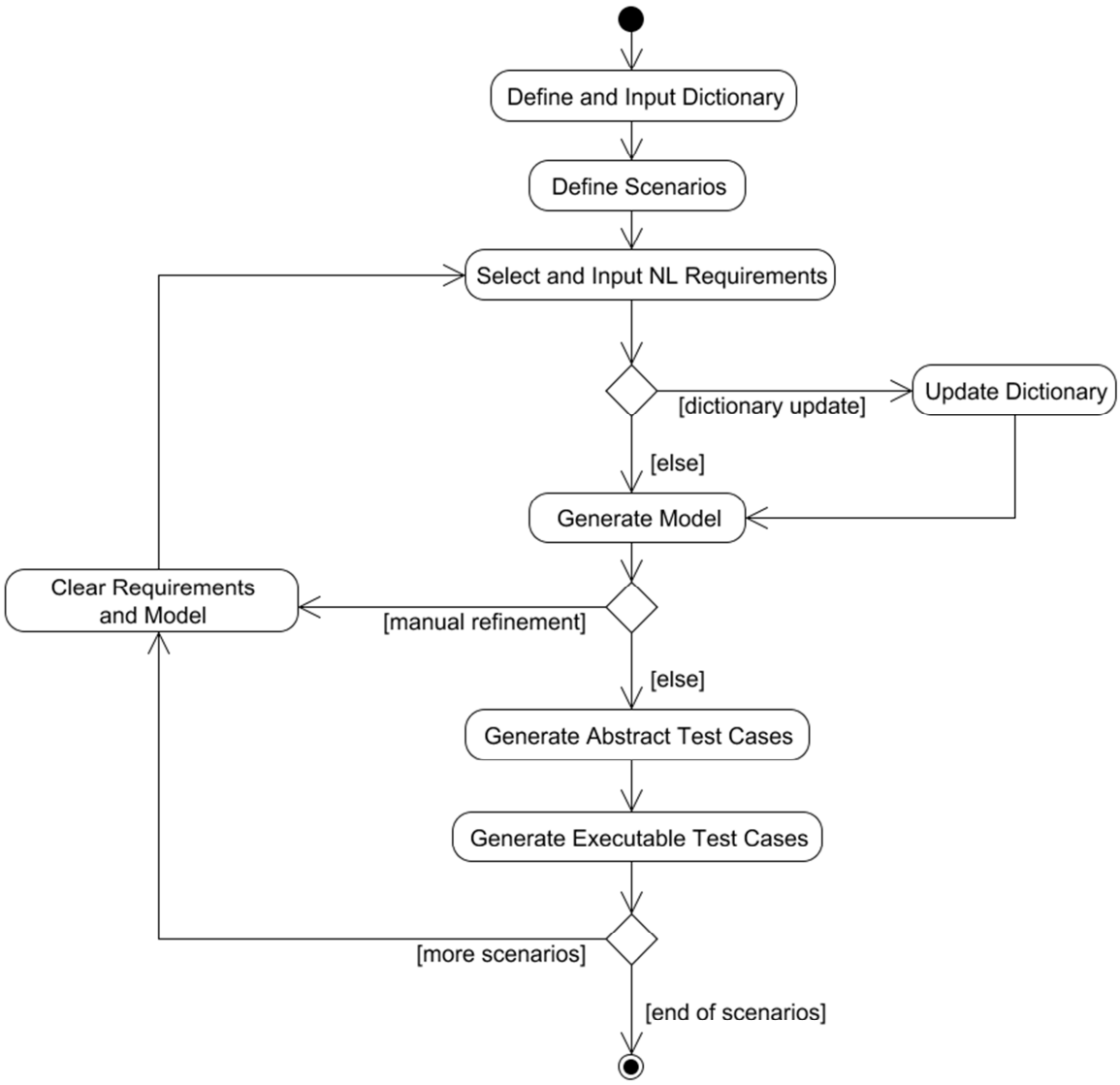

Figura 1 – Metodologia SOLIMVA 1.0

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 30 de junho de 2016.**

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.

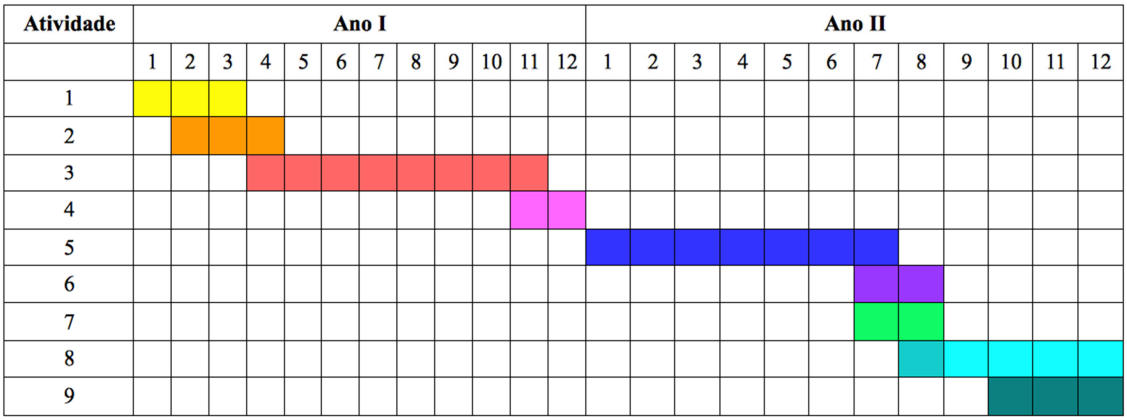

Figura 2 – Cronograma de atividades

Portanto, esse relatório compreende o **mês 1 (agosto/2015) ao mês 11 (junho/2016)**. Considerando as atividades previstas para serem desenvolvidas, mostradas na Figura 2, a Tabela 1 a seguir mostra as atividades concluídas considerando o período 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%	90%
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;	50%	45%
5	Adaptar técnica já existente, ou propor uma nova técnica, para gerar casos de testes de software a partir de Model Checking;	0%	10%

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 a que se refere esse relatório (01 de agosto de 2015 a 30 de junho de 2016). Desse modo, pode-se dizer que todas as atividades previstas para esse período da bolsa foram cumpridas adequada e satisfatoriamente: as atividades 1 e 2 foram totalmente concluídas (100%), e as atividades 3 e 4 foram também bastante desenvolvidas, e estão em fase de conclusão.

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 é uma das mais desafiadoras do projeto, devido a necessidade de realizar uma transformação, fundamentada matematicamente, de modelo Statechart para Transition System (TS; Sistema de Transição) do Model Checker selecionado, o NuSMV. Já foi definida uma nova abordagem (algoritmos) composicional, onde foram identificados elementos do Statecharts para fazerem parte de tal abordagem. A abordagem composicional pode ser, formalmente descrita, como:

StatTS := XOR || Transition || AND || Hierarchy || History (Deep/Shallow) || Condition

onde:

→ StatTS = abordagem composicional para transformação de modelos Statecharts para Transition Systems do Model Checker NuSMV;

→ XOR = estados XOR;

→ Transition = transições;

→ AND = estados AND, representando paralelismo;

→ Hierarchy = hierarquia presente nos modelos Statecharts;

→ History = entrada por histórico, podendo ser rasa (shallow) e profunda (deep);

→ Condition = condições.

Essa abordagem foi aplicada a diversos modelos Statecharts e a transformação para os respectivos TSs têm se mostrado efetivas. No momento, estão sendo elaboradas as regras da semântica formal de tradução dos modelos Statecharts para o Model Checker NuSMV.

Na atividade 4, paralelamente a atividade 3, a abordagem composicional já começou a ser implementada em Java, e já está sendo incorporada à ferramenta SOLIMVA versão 1.0, de forma que será gerada uma nova versão, 1.1, da ferramenta SOLIMVA ao término dessa atividade. Perceber que a atividade 4 não era para ser totalmente concluída considerando o período desse relatório.

Em compensação, a atividade 5 não estava prevista para ser iniciada no período a que se refere esse relatório. Devido a necessidade de se fazer uma pré-análise das abordagens para gerar casos de teste de software a partir de Model Checking, já foi feito um estudo inicial de tais abordagens que, basicamente, podem ser divididas em duas grandes categorias: baseadas em cobertura e baseadas em mutantes [Fraser et al. 2009].

O detalhamento 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

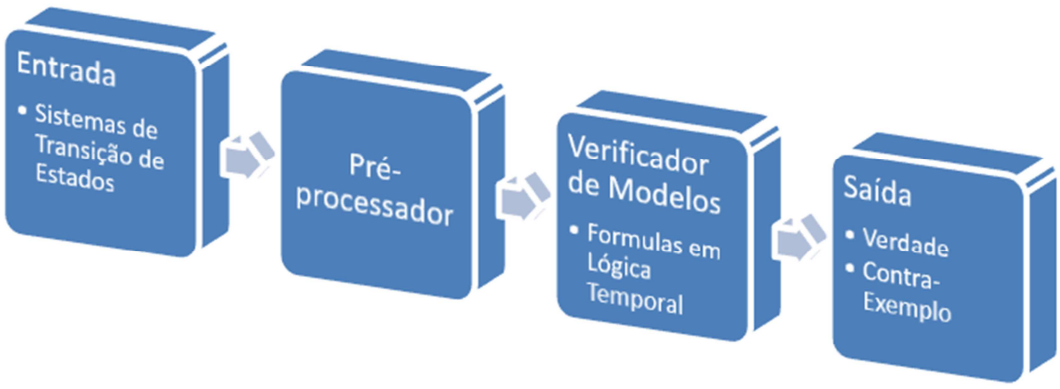

Figura 3 – Visão simplificada do Processo de Model Checking.

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:

- ¬ - Negação
- ∧ - Conjunção
- ∨ - Disjunção
- → - Condicional
- ↔ - Bicondicional

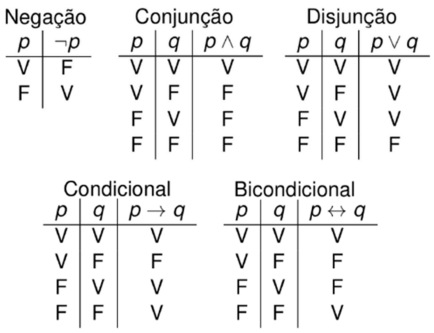

Figura 4 - Tabelas verdade elementares

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.

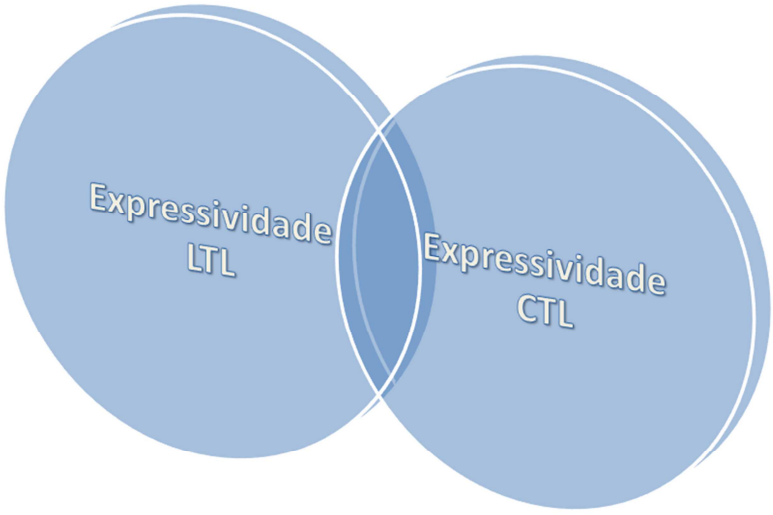

Figura 5 – LTL x CTL

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

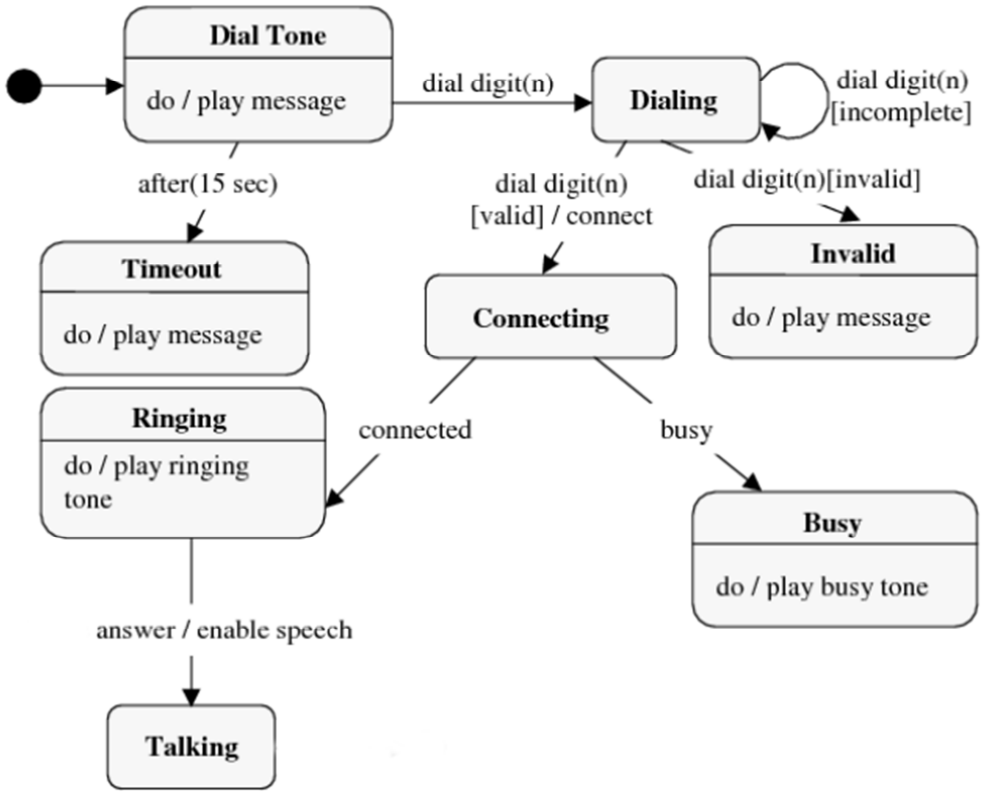

Figura 6 – Exemplo de chamada telefônica.

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.

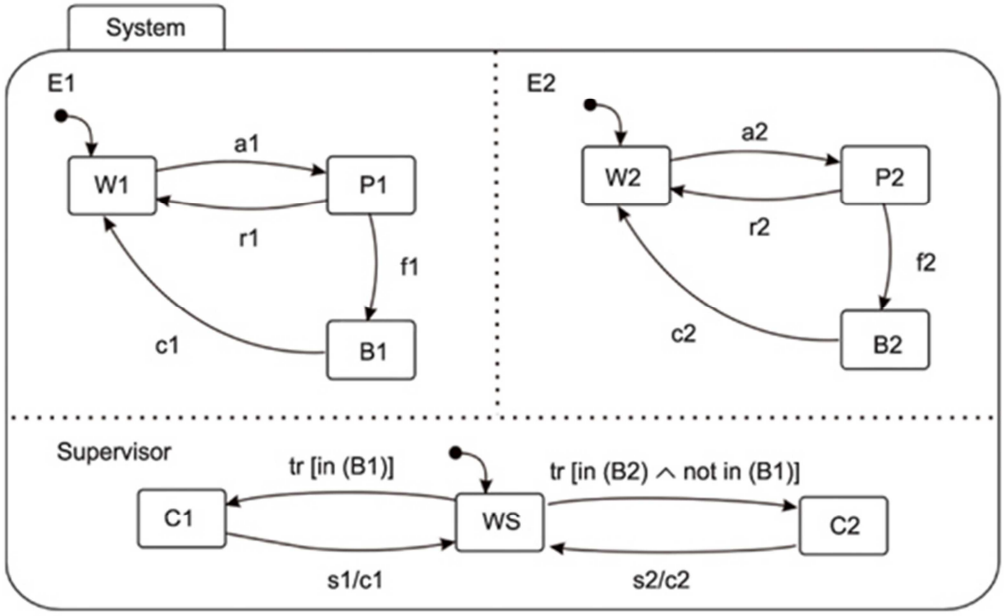

Figura 7 – Especificação em Statecharts de um sistema com duas máquinas e um reparador

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.

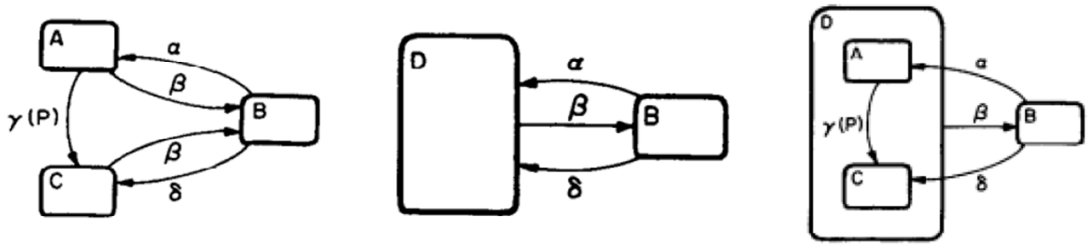

Figura 8 – Clustering.

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.

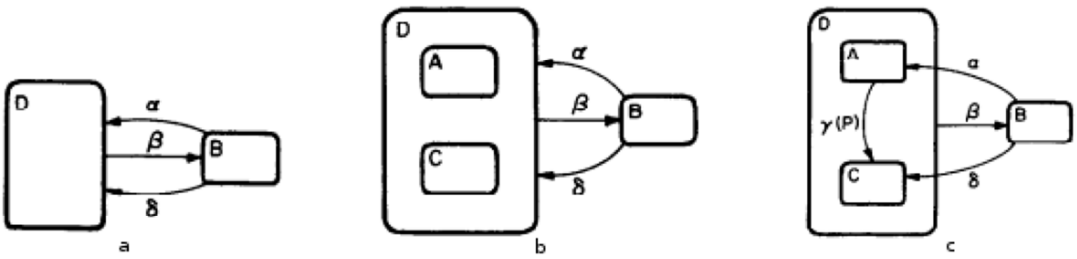

Figura 9 – Refinamento.

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.

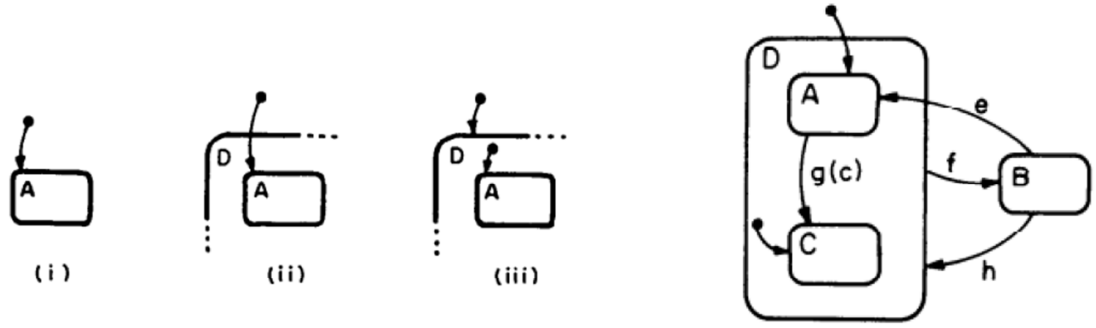

Figura 10 – Estado default.

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.

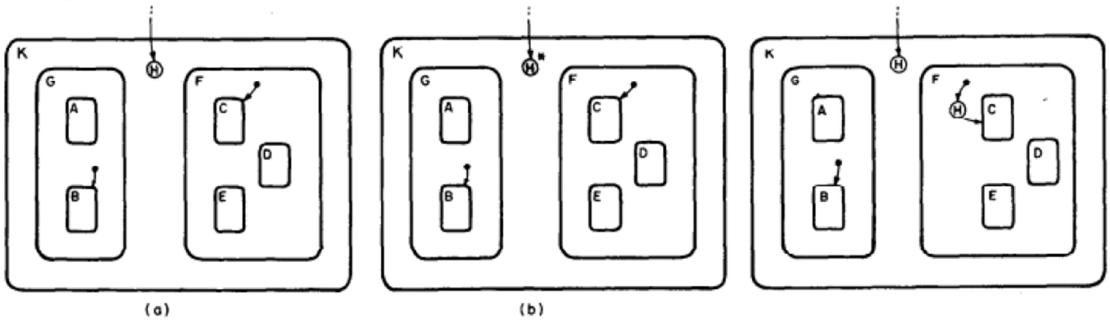

Figura 11 – Entrada pela história.

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

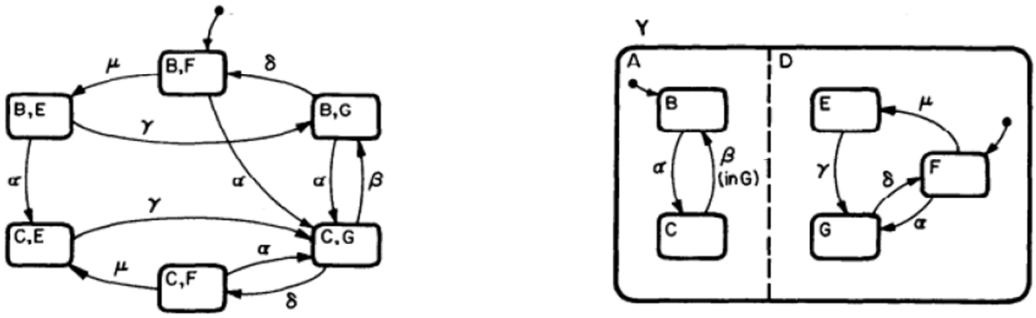

Figura 12 – Ortogonalidade/Paralelismo.

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.

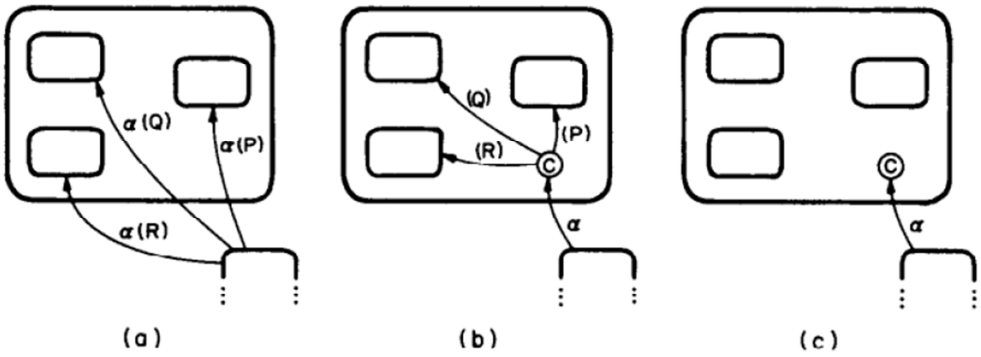

Figura 13 – Entrada de Condição.

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

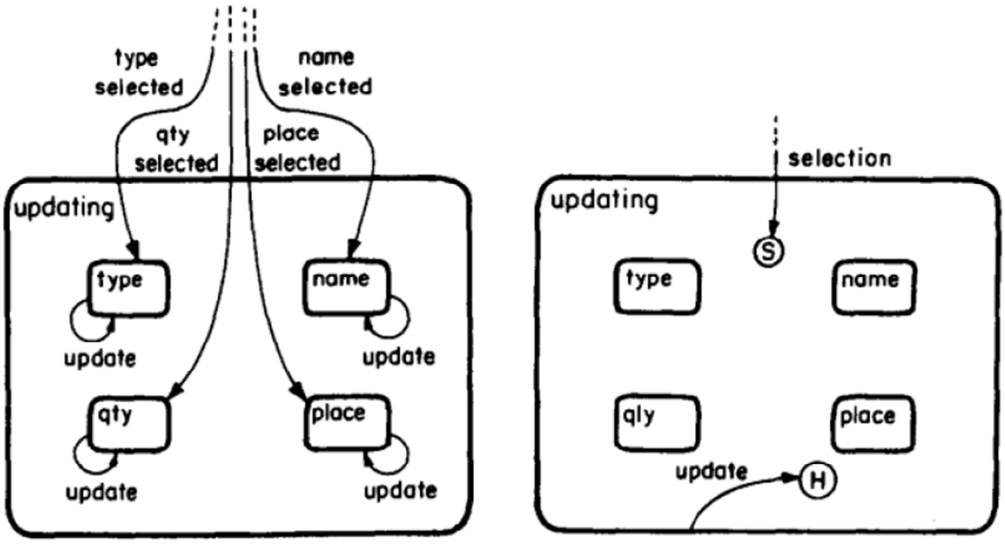

Figura 13 – Entrada de seleção.

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.

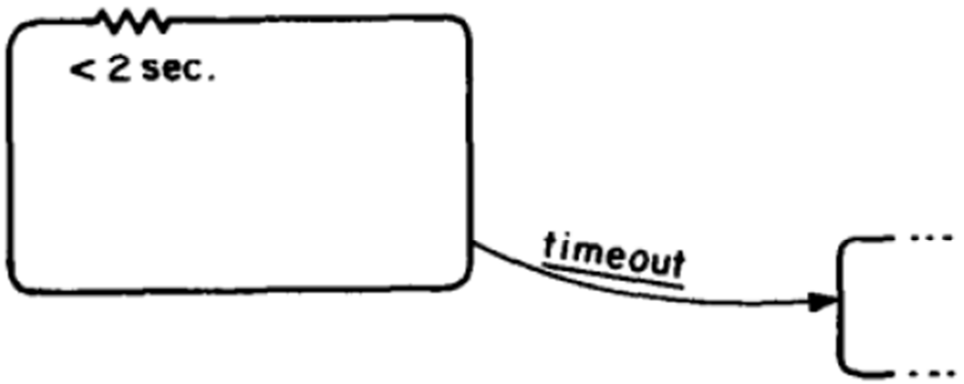

Figura 14 – Delays e Timeouts.

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. ATIVIDADE 3: TRANSFORMAÇÃO DE MODELOS STATECHARTS PARA O MODEL CHECKER SELECIONADO

Dado que o Model Checker NuSMV foi o selecionado, a próxima etapa se refere a realização de transformação de modelos Statecharts, gerados pela ferramenta SOLIMVA, para o NuSMV. Conforme mencionado previamente, essa atividade é bastante complexa pois, além de precisar realizar a transformação em si, será preciso fazer uma análise matemática da transformação do modelo original, Statecharts, para o modelo traduzido para o NuSMV.

Essa atividade é subdividida em várias etapas, conforme mencionado a seguir:

1.) Instalar e executar a ferramenta SOLIMVA 1.0, no intuito de se familiarizar com a ferramenta. **Essa etapa já foi 100% concluída;**

2.) Propor uma solução para controle de versões da ferramenta SOLIMVA. Diversos softwares foram estudados (SVN, Git/GitHub, Mercurial,

Bitbucket). A solução Bitbucket foi a selecionada, pois os repositórios ficam em nuvem privada, podendo ser acessado por todos envolvidos no projeto, e por ser uma solução gratuita. **Essa etapa também já foi 100% concluída;**

3.) Estudar artigos para transformação de modelos Statecharts para o Model Checker NuSMV. Diversos artigos foram selecionados e estudados. **Essa etapa também já foi 100% concluída**;

4.) Adaptar ou desenvolver novo algoritmo (abordagem) para transformar modelos Statecharts para TSs do Model Checker NuSMV. Essa é a etapa mais complexa dessa atividade, devido a necessidade de realizar uma transformação, fundamentada matematicamente, de modelo Statechart para o TS do Model Checker selecionado, o NuSMV. Já foi definida uma nova **abordagem (algoritmos) composicional,** onde foram identificados elementos do Statecharts para fazerem parte de tal abordagem. A abordagem composicional pode ser, formalmente, descrita como:

StatTS := XOR || Transition || AND || Hierarchy || History (Deep/Shallow) || Condition

onde:

→ StatTS = abordagem composicional para transformação de modelos Statecharts para Transition Systems do Model Checker NuSMV;

→ XOR = estados XOR;

→ AND = estados AND, representando paralelismo;

→ Transition = transições;

→ Hierarchy = hierarquia presente nos modelos Statecharts;

→ History = entrada por histórico, podendo ser rasa (shallow) e profunda (deep);

→ Conditions = condições.

Como exemplo, considere o modelo Statecharts mostrado na Figura 15.

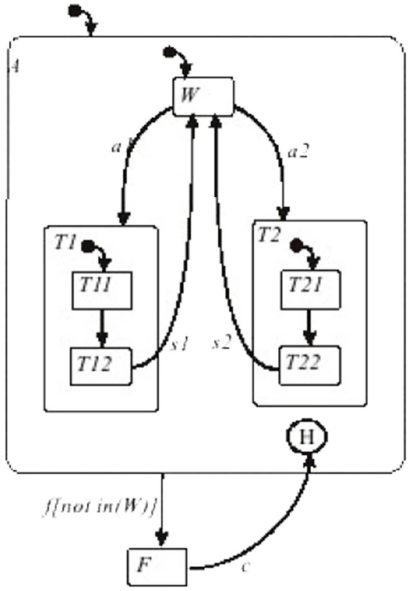

Figura 15 – Statecharts: exemplo.

Podemos identificar no modelo alguns mecanismos de Statecharts citados anteriormente, como: clustering, entrada pela história e ortogonalidade.

A nova abordagem proposta compõe o modelo Statechart da **Figura 15** basicamente separando-o por níveis de hierarquia e atividade paralelas. Cada ortogonalidade e nível hierárquico deste modelo será composto por: ATIVADOR_DE_EVENTOS, ESTADOS_DO_MESMO_NÍVEL e EVENTOS.

Podemos identificar os níveis hierárquicos com mais clareza na Figura 16.

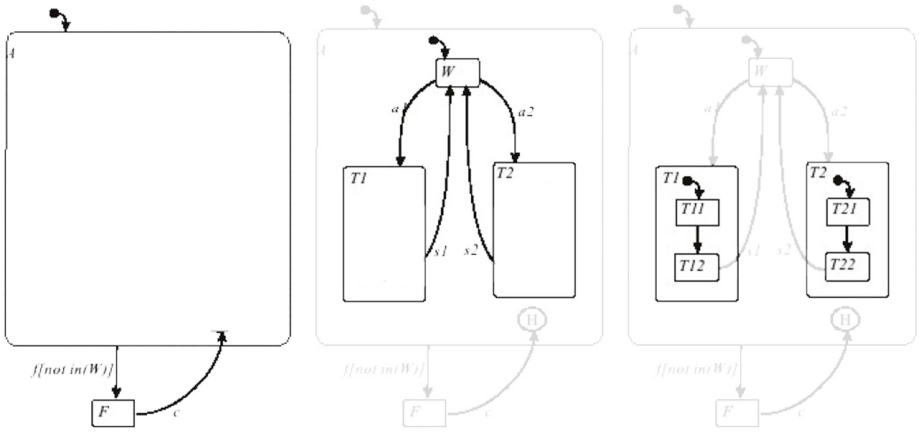

Figura 16 – Níveis de Hierarquia.

No primeiro nível hierárquico, podemos identificar que os estados compostos pelo estado global são ***A*** e ***F***. O segundo nível é composto apenas pelos estados ***W***, ***T1*** e ***T2***. Já o terceiro apresenta ortogonalidade, assim comporemos as atividades paralelas separadamente. Os estados que compõem ***T1*** são ***T11*** e ***T12*** e os que compõem ***T2*** são ***T21*** e ***T22***. Assim a tradução dos estados e eventos para a linguagem do NuSMV ficará como a seguir:

```
VAR
	event_dispatch_m : {on,off};
	states_m : {a,f};
	events_m : {f,c};

	event_dispatch_a : {on,off};
	states_a : {w,t1,t2};
	events_a : {a1,a2};

	event_dispatch_t1 : {on,off};
	states_t1 : {t11,t12};
	events_t1 : {null,s1,f};

	event_dispatch_t2 : {on,off};
	states_t2 : {t21,t22};
	events_t2 : {null,s2,f};

	event_dispatch_s : {on,off};
	states_s : {u,w,r};
	events_s : {u,a,null,r};
```

O ativador de eventos é responsável por ativar ou não os eventos de um determinado nível hierárquico.

Para demonstrarmos como foi feita a tradução completa de um modelo Statecharts para a linguagem do NuSMV, utilizaremos um modelo mais simples, mostrado na Figura 17.

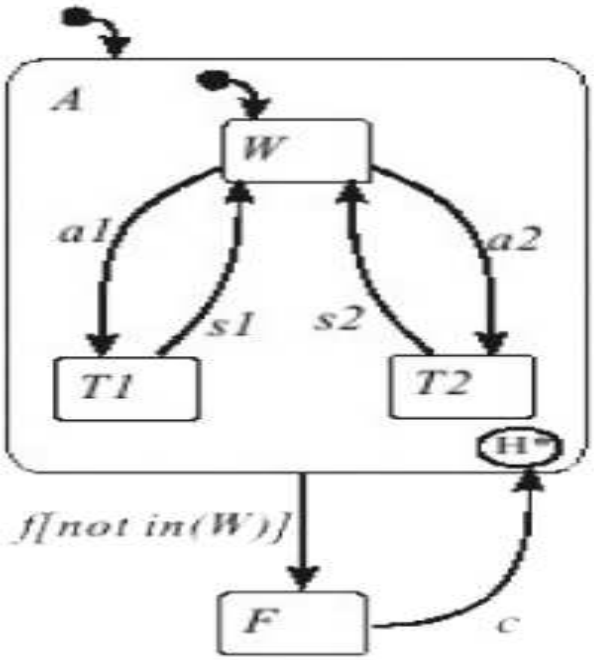

Figura 17 – Statecharts: Hierarquia e Histórico.

Seguindo a mesma lógica descrita anteriormente, para esse modelo teremos dois níveis hierárquicos, do estado ***M*** e do estado ***A***, sendo ***M*** o estado global.

Assim, temos:

```
MODULE main
VAR
	event_dispatch_m : {on,off};
	states_m : {a,f};
	events_m : {f,c};

	event_dispatch_a : {on,off};
	states_a : {w,t1,t2};
	events_a : {a1,a2,s1,s2,f};
```

Os disparadores de eventos servem para controlar os estados e eventos, a partir da hierarquia.

Em seguida, temos os estados e eventos default, onde cada subestado e eventos são iniciados.

```
ASSIGN
	init (event_dispatch_m) := off;
	init (states_m) := a;
	init (events_m) := f;

	init (event_dispatch_a) := on;
	init (states_a) := w;
	init (events_a) := a1;
```

No código abaixo, temos as lógicas do estado ***global***.

```
//Logica do disparador de eventos.
next (event_dispatch_m) := case
	event_dispatch_a = on & events_a = f : on;
	event_dispatch_m = on & events_m = c : off;
	TRUE : event_dispatch_m ;
esac;

//Lógica das transições de estados.
next (states_m) := case
	event_dispatch_a = on & events_a = f : f;
	event_dispatch_m = on & events_m = c : a;
	event_dispatch_m = on & events_m = f : a;
	TRUE : states_m ;
esac;

//Lógica das transições de estados
next (events_m) := case
	event_dispatch_a = on & events_a = f : c;
	event_dispatch_m = on & events_m = c : f;
	TRUE : events_m ;
esac;
```

E no código abaixo, temos as lógicas do estado ***A***.

```
//Logica do disparador de eventos.
next (event_dispatch_a) := case
	event_dispatch_a = on & events_a = f : off;
	event_dispatch_m = on & events_m = c : on;
	TRUE : event_dispatch_a ;
esac;

//Lógica das transições de estados.
next (states_a) := case
	event_dispatch_a = on & events_a = a1 : t1;
	event_dispatch_a = on & events_a = a2 : t2;
	event_dispatch_a = on & events_a = s1 : w;
	event_dispatch_a = on & events_a = s2 : w;

	TRUE : states_a ;
esac;

//Logica das transições de eventos.
next (events_a) := case
	event_dispatch_a = on & events_a = s1 : a1;
	event_dispatch_a = on & events_a = s2 : a2;
```

```
        event_dispatch_a = on & events_a = a1 : {s1,f};
        event_dispatch_a = on & events_a = a2 : {s2,f};
        event_dispatch_a = on & states_a = t1 & events_a = f : s1;
        event_dispatch_a = on & states_a = t2 & events_a = f : s2;

        TRUE : events_a ;
esac;
```

Como a transição de estados e de eventos são controlados pelo disparador de eventos, quando o mesmo não está em operação, ocorre um congelamento do último estado em que foi passado, naquela hierarquia. Assim podemos controlar o histórico do modelo com auxilio do disparador de eventos e das lógicas de cada hierarquia.

Portanto, os exemplos acima mostram que a nova abordagem composicional proposta tem sido efetiva na tradução de modelos Statecharts, considerando vários aspectos da linguagem Statecharts, para a linguagem do Model Checker NuSMV. Portanto, essa atividade está em processo de finalização onde estão sendo elaboradas as regras da semântica formal de tradução dos modelos Statecharts para o Model Checker NuSMV.

6. ATIVIDADE 4: INCORPORAÇÃO DA NOVA ABORDAGEM À FERRAMENTA SOLIMVA

Paralelamente a atividade 3, a abordagem composicional já começou a ser implementada em Java, e já está sendo incorporada à ferramenta SOLIMVA versão 1.0, de forma que será gerada uma nova versão, 1.1, da ferramenta SOLIMVA ao término dessa atividade. A Figura 18 mostra parte da estrutura de pacotes do código-fonte da SOLIMVA 1.1, 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.

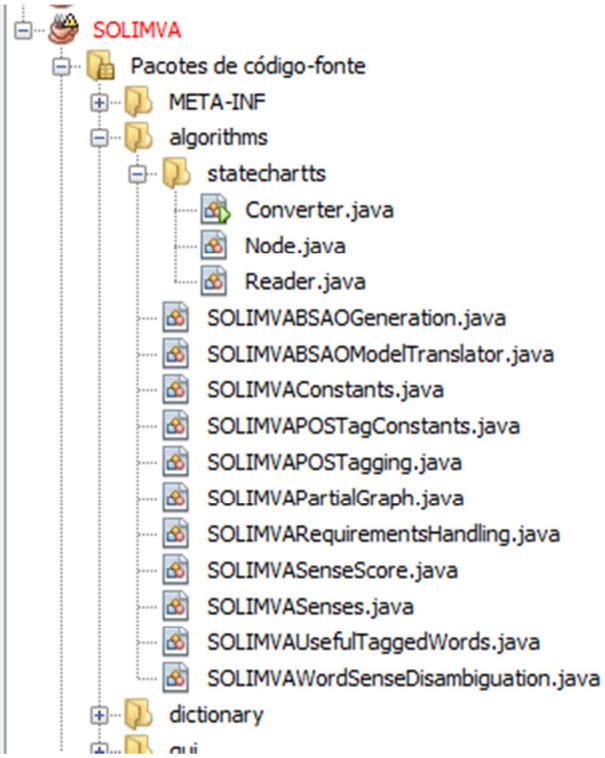

Figura 18 – SOLIMVA 1.1: parte da estrutura de pacotes do código-fonte.

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

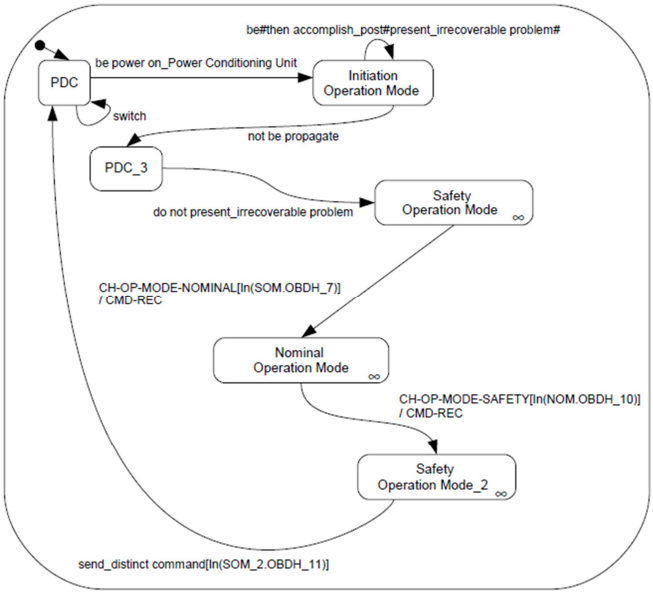

Figura 19 – Statechart gerado pela SOLIMVA. Adaptado de [Santiago Júnior 2011]

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.

No intuito de tornar mais amigável o uso da SOLIMVA 1.1, no que se refere a essa transformação de Statecharts para o NuSMV, uma nova aba, denominada StatSMV, foi criada para que seja possível visualizar o código gerado a partir do modelo Statechart para a linguagem NuSMV, executar o NuSMV diretamente a partir da SOLIMVA (cobrindo a questão de

interoperabilidade entre as ferramentas), permitir que sejam gerados os estados alcançáveis e os traços do TS, tudo isso de forma automática. A Figura 20 mostra a aba StatSMV, incorporada a SOLIMVA 1.1, e a Figura 21 mostra a interface gráfica relacionada a aba StatSMV.

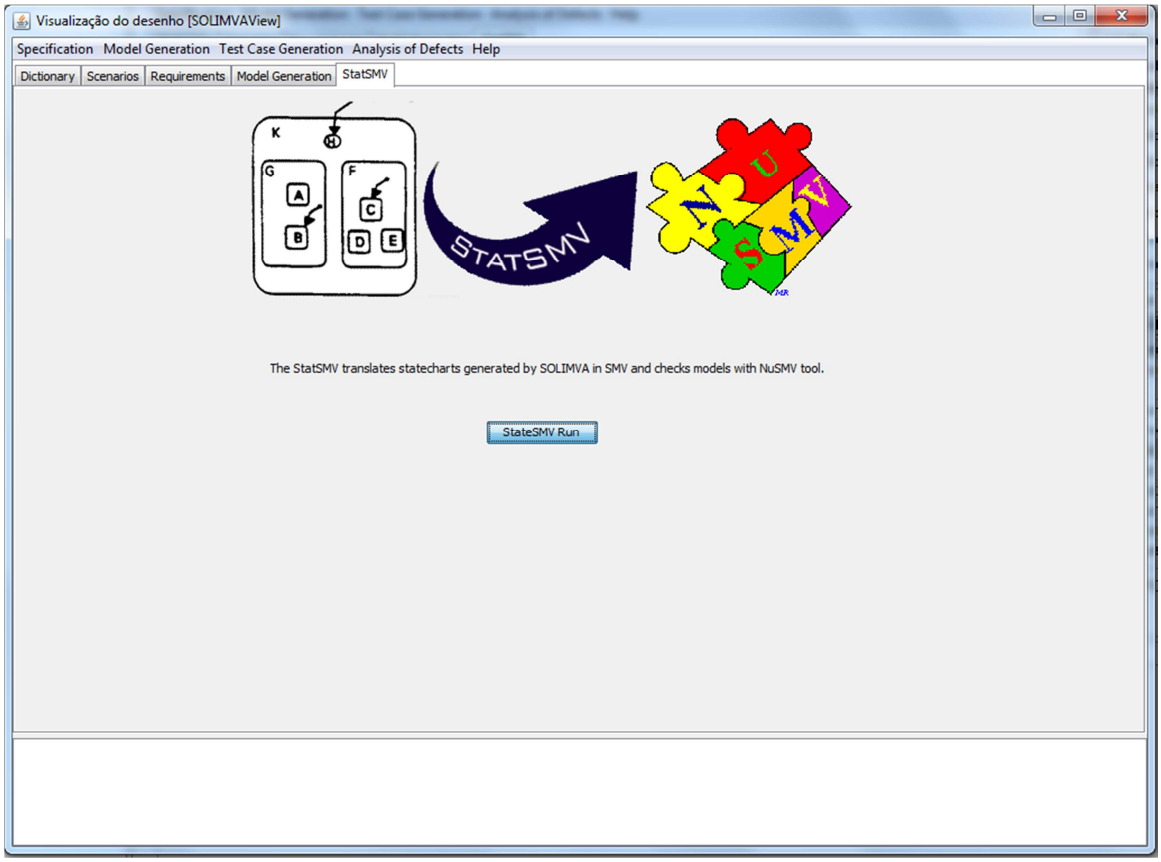

Figura 20 – SOLIMVA 1.1: Nova aba StatSMV

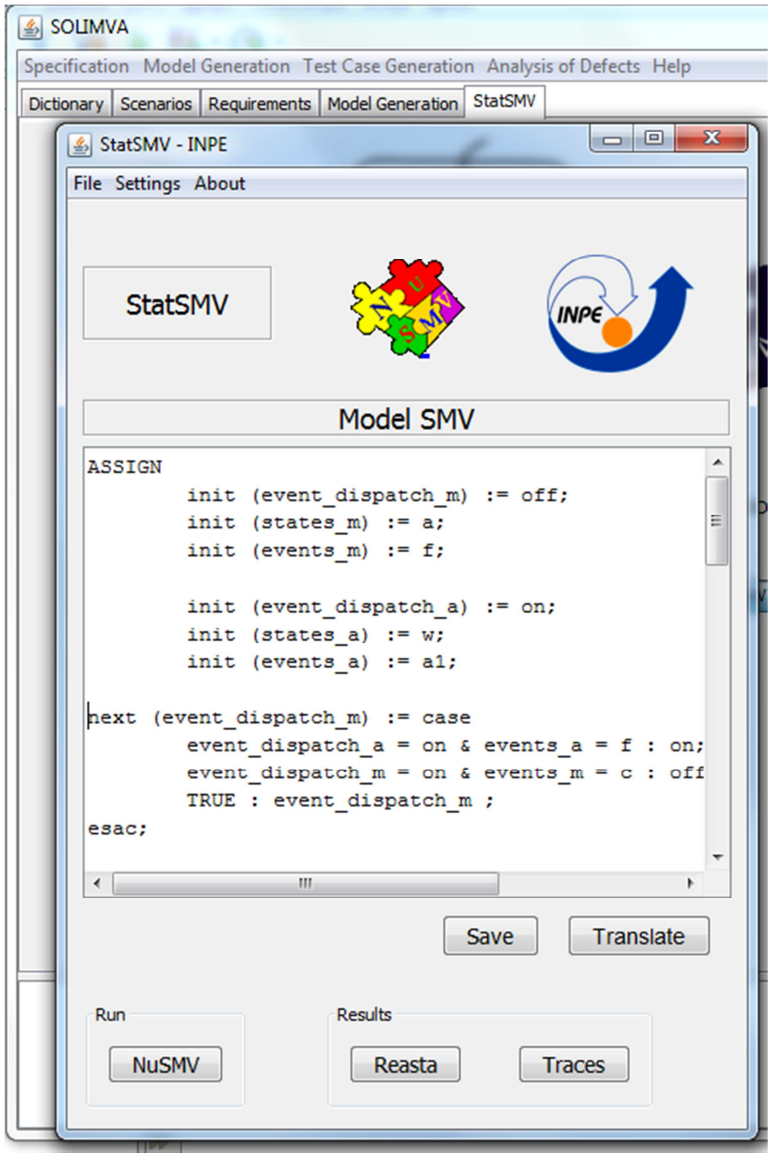

Figura 21 – Interface Gráfica relacionada a aba StatSMV.

7. CONCLUSÕES

Esse relatório apresentou as atividades desenvolvidas, no período de **01 de agosto de 2015 a 30 de junho de 2016**, relacionadas ao projeto *Testes de Software via Model Checking para Sistemas Espaciais Críticos.* Conforme descrito nesse relatório, as atividades previstas para esse período foram cumpridas adequada e satisfatoriamente. Detalhadamente, as seguintes atividades foram desenvolvidas: estudo de fundamentação teórica relacionada aos conceitos da pesquisa; análise e seleção de Model Checker para ser usado no projeto (o Model Checker selecionado foi o NuSMV); desenvolvimento de novos algoritmos (abordagem composicional) para transformar modelos Statecharts para o Model Checker NuSMV; implementação parcial, e incorporação à ferramenta SOLIMVA, dos novos algoritmos (abordagem composicional) para transformar modelos Statecharts para o Model Checker NuSMV. Além disso, uma atividade não prevista para ser iniciada no período a que se refere esse relatório, a atividade para gerar casos de testes de software a partir de Model Checking, já foi antecipadamente iniciada.

8. REFERÊNCIAS

[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.