VERIFICAÇÃO FORMAL DE APLICAÇÕES CONCORRENTES EM SISTEMAS ELÉTRICOS DE POTÊNCIA

Márcio Egydio da Silva Rondon

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia Elétrica.

Orientador: Aloysio de Castro Pinto Pedroza

Rio de Janeiro
Março de 2012

VERIFICAÇÃO FORMAL DE APLICAÇÕES CONCORRENTES EM SISTEMAS ELÉTRICOS DE POTÊNCIA

Márcio Egydio da Silva Rondon

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Aloysio de Castro Pinto Pedroza, Dr.

Prof. Carlos Alberto Nunes Cosenza, D.Sc.

Prof. Djalma Mosqueira Falcão, Ph.D.

Prof. Luís Henrique Maciel Kosmalski Costa, Dr.

Prof. Milton Brown do Couto Filho, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2012

Rondon, Márcio Egydio da Silva

Verificação Formal de Aplicações Concorrentes em Sistemas Elétricos de Potência/Márcio Egydio da Silva Rondon. – Rio de Janeiro: UFRJ/COPPE, 2012.

XV, 90 p.: il.; 29, 7cm.

Orientador: Aloysio de Castro Pinto Pedroza

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2012.

Referências Bibliográficas: p. 68 – 70.

1. Sistemas de Potência. 2. Proteção de Sistemas Elétricos. 3. Model Checking. 4. SPIN. 5. Verificação Formal. 6. IEC61850. I. Pedroza, Aloysio de Castro Pinto. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

A meus pais, esposa e filhos.

Agradecimentos

A Deus, o Grande Arquiteto do Universo, que sempre me deu a força necessária para continuar nos momentos de dúvidas e indecisões.

A minha família, pela compreensão das muitas ausências necessárias.

Ao professor Aloysio Pedroza - orientador e amigo, pelo incentivo constante e pela assessoria neste trabaho.

Aos colegas da Light, e principalmente aos colegas Tatiana Maria e Marco Aurélio, ambos do ONS, pela ajuda na obtenção de dados fundamentais para a elaboração desta tese.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

VERIFICAÇÃO FORMAL DE APLICAÇÕES CONCORRENTES EM SISTEMAS ELÉTRICOS DE POTÊNCIA

Márcio Egydio da Silva Rondon

Março/2012

Orientador: Aloysio de Castro Pinto Pedroza

Programa: Engenharia Elétrica

Na última década, o uso da tecnologia digital na automação de sistemas elétricos de potência experimentou uma notável evolução. Os relés de proteção se transformaram em dispositivos inteligentes, chamados Intelligent Electrical Devices (IEDs) que, além de agregarem maiores recursos às tarefas de proteção, são também capazes de participar das diversas funções de automação normalmente utilizadas em uma subestação. A norma IEC61850, através da comunicação utilizando redes LAN de alta velocidade e elevada confiabilidade, permitiu uma integração entre os IEDs. Esta integração facilita o compartilhamento das informações e a implantação de funções de supervisão, controle e automação. Também começa a ganhar importância o uso de esquemas de proteção de sistemas formado por Unidades Terminais Remotas, ou por Controladores Lógicos Programáveis, que atuando de modo isolado ou em rede tem a capacidade de proteger uma grande área ou região do sistema elétrico. Nesta tese, mostra-se, através de experimentos, o uso da técnica do *Model Checking* na verificação de alguns esquemas básicos e de um esquema real de proteção. Os resultados obtidos mostram que a utilização desta técnica em projetos aplicados a Sistemas Elétricos de Potência poderá contribuir para uma maior segurança na sua operação.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

FORMAL VERIFICATION OF CONCURRENT APPLICATIONS IN ELECTRIC POWER SYSTEMS

Márcio Egydio da Silva Rondon

March/2012

Advisor: Aloysio de Castro Pinto Pedroza

Department: Electrical Engineering

In the last decade, the use of digital technology in the automation of electric power systems experienced a noticeable evolution. Protective relays developed into smart devices called Intelligent Electrical Devices (IEDs) that, in addition to adding more resources to protective tasks, are also capable of participating in various automation functions normally used in a substation. The IEC61850 Standard, through communication using high speed and high reliability local area networks, allowed integration between IEDs. This integration facilitates the sharing of information and the implementation of supervisory, control and automation functions. The System Protection Scheme, composed by Remote Terminal Units, or Programmable Logic Controllers that operating stand alone or within a network, is getting importance. It is able to protect a large area or region of the power system. In this thesis, the use of the Model Checking technique in the verification of some basic schemes and a real protective scheme is shown through experiments. The results obtained show that the use of this technique in projects applied to Electric Power Systems may contribute to enlarge the security in its operation.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

A	em todos os *paths* futuros, p. 10
E	em algum *path* no futuro, p. 10
W	operador temporal a menos que, p. 8
$\Box$	operador temporal em sempre no futuro, p. 7
$\Leftrightarrow$	se, e somente se, p. 6
$\Rightarrow$	implica, p. 6
$\bigcirc$	operador temporal no próximo estado, p. 7
$\bigcup$	operador temporal até, p. 8
$\diamondsuit$	operador temporal em algum estado no futuro, p. 7
$\forall$	para todo, p. 12
$\in$	pertence a, p. 12
$\models$	satisfaz, p. 3
$\neg$	negação, p. 6
$\vee$	OU, p. 6
$\wedge$	E, p. 6

Lista de Abreviaturas

CEB	Centrais Elétricas de Brasília, p. 52
CLP	Controlador Lógico Programável, p. 52
ECS	Esquema de Controle de Segurança, p. 52
EPS	Esquema de Proteção de Sistemas, p. 34
GOOSE	Generic Object Oriented Substation Event, p. 33
IEC	International Electrotechnical Commission, p. 2
IED	Intelligent Electrical Device, p. 33
MAIN	Mid-America Interconnected Network, p. 36
MAPP	Mid-Continent Area Power Pool, p. 36
MMS	Manufacturing Message Specification, p. 33
NERC	Norh American Electric Reliability Council, p. 1
ONS	Operador Nacional do Sistema Elétrico, p. 52
PROMELA	Process Meta Language, p. 3
SE	Subestação, p. 52
SPIN	Simple PROMELA Interpreter, p. 3
TCP/IP	Transmission Control Protocol/Internet Protocol, p. 33
TC	Transformador de Corrente, p. 32
TP	Transformador de Potencial, p. 32
UTR	Unidade Terminal Remota, p. 34
WSCC	Western Systems Coordinating Council hoje WECC - Western Electricity Coordinating Council, p. 36

Capítulo 1

Introdução

A evolução tecnológica ocorrida principalmente nas últimas décadas provocou grandes mudanças no setor elétrico, em seus diversos níveis: geração, transmissão e distribuição de energia. O processo conhecido por globalização forçou a abertura de mercados e criou uma competitividade entre os agentes, onde a eficiência tecnológica tornou-se fundamental.

Os Sistemas Elétricos de Potência têm se tornado cada vez mais complexos. Necessitam, portanto, de tecnologias avançadas para fazer frente a uma demanda crescente no contexto de um sistema interligado que abrange praticamente todo o país. Há a necessidade de gerenciamento e controle cada vez mais sofisticados, incluindo conhecimentos modernos de diversas áreas científicas, principalmente da computação.

Todo o sistema de transmissão de Energia Elétrica se baseia em redes de alta tensão, que necessitam de segurança máxima em toda a sua atividade. Entretanto, na transmissão de grandes blocos de energia ao longo de grandes distâncias, a ocorrência de certas contingências em situações críticas pode deixar o sistema inseguro. Um dos sistemas responsáveis por garantir essa segurança é o Sistema de Proteção.

O objetivo básico de um Sistema de Proteção nos Sistemas Elétricos é proporcionar, de forma rápida e segura, o isolamento de um equipamento, área ou região em falha, deixando que o restante do Sistema Elétrico continue com o fornecimento normal de energia.

Embora na maioria das análises de confiabilidade, assume-se que os sistemas de proteção são confiáveis e operam de forma correta, sabe-se que esses sistemas podem falhar, e que a ocorrência dessas falhas tem significante influência na rede elétrica [2].

O relatório System Disturbance Report do NERC (Norh American Electric Reliability Council) mostra que entre 1986-1998, dos 30 casos que envolviam a operação de Sistemas de Especiais de Proteção, 21 foram relatados como operações com su-

cesso, enquanto nove envolveram falhas na operação[3]. As razões para essas falhas incluíam erro na lógica do projeto, erro no software, falhas de hardware, erro na configuração e falhas no disparo [4].

Vários casos de *blackouts* são devidos a falhas do Sistema de Proteção. Normalmente, são falhas que ficam adormecidas quando o sistema está operando em condição normal e que são expostas nos casos de distúrbios.

Assim, nota-se que a correta operação do sistema de proteção do Sistema Elétrico tem uma participação significativa no aumento da confiabilidade da rede elétrica.

Tradicionalmente, os engenheiros de proteção definem o esquema de proteção a ser utilizado baseado em sua experiência e conhecimentos da natureza do Sistema Elétrico.

Como os Sistemas Elétricos de Potência tem se tornado mais complexos, cobrindo grandes regiões geográficas, os requisitos para o esquema de proteção tendem a ser mais rigorosos e a maneira como estes esquemas se interligam torna a combinação entre eles mais complicada. Isto se deve ao fato de que uma falha no sistema pode provocar desligamentos em cascata em todo o Sistema Elétrico.

Com a utilização da tecnologia de relés digitais, a interligação entre os relés se dá através de comunicação em rede Ethernet via protocolo IEC 61850. Este tipo de interligação acrescenta mais um aspecto a ser avaliado durante o projeto, que é a troca de mensagens entre estes relés durante uma falha no Sistema Elétrico.

Também começa a ganhar importância o uso de esquemas de proteção de sistemas formado por Unidades Terminais Remotas, ou por Controladores Lógicos Programáveis, que atuando de modo isolado ou em rede tem a capacidade de proteger uma grande área ou região do sistema elétrico. Portanto, cada vez mais se faz necessário o desenvolvimento de ferramentas efetivas de análise para avaliação e comparação de alternativas para os esquemas de proteção.

Esta tese mostra que a técnica formal de verificação, o *Model Checking*, pode ser utilizada como ferramenta para avaliar e testar a lógica de esquemas de proteção em aplicações nos Sistemas Elétricos de Potência. Primeiramente esta demonstração é feita com o uso de exemplos básicos. Em seguida, mostra-se que, com a utilização desta técnica, foi possível a identificação de um erro em um caso real que provocou um desligamento de linhas de transmissão na região de Brasília.

O termo *Model Checking* representa um conjunto de técnicas formais de análise automática de sistemas. Através do *Model Checking*, erros de projeto podem ser (e têm sido) identificados. Esta técnica tem sido adotada como procedimento padrão para assegurar a correção de Sistemas de Controle [5]. A verificação formal, quando aplicada corretamente, produz evidências de que o projeto desenvolvido satisfaz a todos os requisitos iniciais estabelecidos para o sistema[6].

O *Model Checker* (verificador) tem como entrada a descrição do sistema a ser

analisado e um conjunto de propriedades que são esperadas para esse sistema. O verificador ou confirma que essas propriedades são alcançadas, ou informa quais as propriedades que poderão ser violadas. No último caso, o verificador apresenta o contra-exemplo: a condição que viola determinada propriedade. Através desta informação, erros no sistema podem ser identificados e corrigidos nas fases iniciais do projeto.

Para o desenvolvimento do algoritmo das ferramentas que utilizam o *Model Checking*, são empregadas as propriedades e características presentes na Lógica Temporal.

Dado um sistema $\mathcal{T}$, que pode ser representado por uma máquina de estados finita, e seja uma propriedade φ, o problema para o algoritmo do *Model Checker* (verificador) é decidir se a sentença $\mathcal{T} \models \varphi$ é, ou não, verdadeira. Se não, o verificador deve fornecer o contra-exemplo, ou seja, a exceção de $\mathcal{T}$ que viola φ.

Nos experimentos realizados mostra-se, através de um exemplo real, que o uso de técnicas formais na verificação de Sistemas de Proteção de Sistemas Elétricos de Potência poderão minimizar as falhas decorrentes de erros na lógica do projeto e na comunicação entre os relés, evitando que sejam identificadas somente quando forem expostas por uma anormalidade no sistema elétrico.

Esta tese está organizada da seguinte forma: no capítulo dois é realizada uma descrição sobre Verificação Formal, são apresentados os conceitos e operadores da Lógica Temporal e descrita a técnica Model Checking. No capítulo três apresentam-se as ferramentas utilizadas, uma descrição da linguagem PROMELA e as principais caracaterísticas do verificador SPIN. No capítulo quatro faz-se uma introdução à Proteção de Sistemas Elétricos. No capítulo cinco são apresentados dois esquemas básicos de proteção, a sua modelagem em PROMELA e mostrados os resultados de verificações de propriedades nestes dois esquemas. No capítulo seis é apresentado um caso real do sistema elétrico brasileiro, com a sua modelagem em PROMELA, e o resultado da verificação de propriedades feita utilizando-se o verificador SPIN. Nesta verificação, foi possível a identificação do erro de lógica do esquema implantado no campo, que causou desligamentos indevidos de linhas de transmissão na área do Distrito Federal. Finalmente, no capítulo sete encontram-se as conclusões da tese.

1.1 Trabalhos relacionados

As técnicas de verificação formal começam a ser aplicadas, principalmente em testes de protocolos de comunicação[5].

Atualmente estas técnicas estão, cada vez mais, sendo utilizadas em diferentes aplicações, em especial pelos principais fabricantes de microprocessadores no projeto de seus novos produtos[7]. Na indústria automobilística existe trabalho mostrando a aplicação desta técnica de verificação sendo utilizada para a identificação de erros no

sistema de controle automático de aceleração presente nos veículos, modelo Camry, da Toyota[8]. Outro trabalho utilizou o SPIN, ferramenta utilizada nesta tese, para a verificação de sistemas embarcados também na área automotiva[9]. Em Van de Molengraft *et al.*[10], existem várias descrições do uso de métodos formais e da lógica temporal em aplicações na área da robótica.

Em sistemas embutidos utilizados em grandes embarcações marítimas também encontra-se trabalho que utiliza este tipo de verificação[11] para a identificação de erros em sua rede de comunicação. Em aplicações de sistemas elétricos de potência, o trabalho que mais se aproximou do que está sendo apresentado nesta tese é a avaliação de segurança através do uso de redes de Petri[12]. Neste trabalho é apresentada uma metodologia para avaliar a segurança de um sistema elétrico de potência considerando a sua resposta frente a distúrbios produzidos por curtos-circuitos e falhas de equipamentos. A avaliação é feita através do uso de Redes de Petri Estocásticas Generalizadas. O trabalho propõe o uso deste tipo de Rede de Petri como ferramenta para o cálculo de índices de segurança de um sistema elétrico de potência.

1.2 Principais Contribuições

Cada vez mais o setor elétrico utiliza sistemas computacionais nos seus subsistemas, e a segurança de sua operação é sempre um objetivo a ser alcançado.

Como pode ser observado na seção anterior, praticamente não foram encontrados trabalhos onde métodos de verificação formal fossem utilizados em aplicações no setor elétrico. Em Ramos G. *et al.*[12], existe um estudo estatístico de erros utilizando redes de Petri, porém, o procedimento adotado e os resultados obtidos se situam bem distantes do conceito de verificação formal.

Esta tese ilustra, através de exemplos básicos e de um caso real, que a verificação formal de modelos, técnica que originalmente foi concebida para aplicações ligadas a Ciência da Computação, é mais uma ferramenta a ser explorada, de modo que venha a se tornar parte do processo de desenvolvimento de projetos de aplicações em Sistemas Elétricos de Potência. A tese também mostra que o uso de expressões com operadores da lógica temporal podem servir para descrever, de maneira formal e não ambígua, tanto os requisitos quanto as propriedades que estas aplicações devem observar.

Capítulo 2

Verificação Formal

Sistemas críticos de segurança (safety-critical systems) são exemplos onde a confiabilidade e a correção são requisitos indispensáveis ao seu funcionamento. Muitos acidentes fatais em sistemas reativos em aplicações em que a segurança é crítica têm ocorrido em situações inesperadas. Essas situações não tinham sido consideradas durante a fase de projeto e/ou de teste destes sistemas. De modo a evitar e prevenir estes tipos de acidentes, os sistemas reativos devem ser projetados de tal forma que tenham uma resposta apropriada para qualquer tipo de entrada em qualquer instante de tempo[13]. As atividades que visam garantir a correção de um projeto o mais cedo possível no ciclo de desenvolvimento têm sido um dos grandes desafios e consumido cada vez mais tempo/verbas do orçamento[14].

A prática comum para a garantia da confiabilidade e correção destes tipos de sistemas é a seguinte: o projeto se inicia com a apresentação dos requisitos estabelecidos para o sistema. Estes requisitos são analisados, as diferentes fases do projeto são definidas, e ao final destas fases, um protótipo é construído. O procedimento para constatação de que o projeto atende aos requisitos estabelecidos inicialmente é chamada de verificação de sistemas. O diagrama com o fluxo deste procedimento está apresentado na figura 2.1.

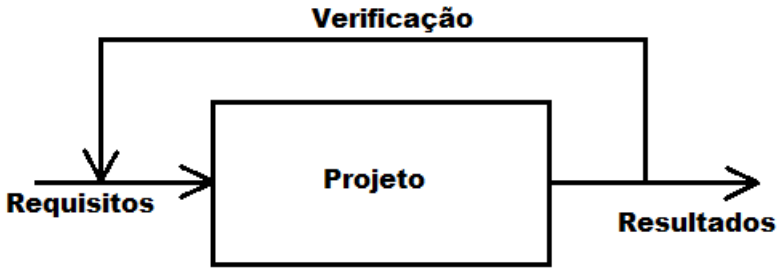

Figura 2.1: Processo de Verificação ao Final do Projeto.

Outro procedimento para identificação de erros é a verificação ao final de cada

fase do projeto. Neste caso, quando um erro é identificado, a correção se concentra somente na última fase do projeto. A Figura 2.2 representa o fluxo deste processo.

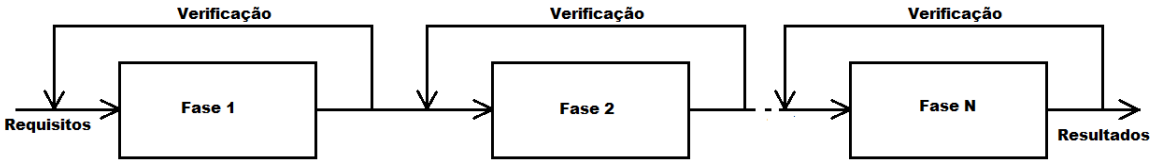

Figura 2.2: Processo de Verficação em Fases.

A identificação de falhas somente ao final de cada fase do projeto não é uma prática recomendável. Neste processo, quando ocorre a identificação de um erro, um grande esforço será necessário para a sua correção.

O teste, embora seja uma técnica de validação à correção de um sistema, normalmente é aplicado de uma maneira *ad-doc*, e realizado de forma manual.

2.1 Métodos Formais de Verificação

Atualmente, os sistemas estão cada vez mais complexos, apresentando processamento contínuo. Realizam entradas e saídas de dados durante toda a sua execução (não apenas no início ou final do processamento), e além disso, precisam também interagir com outros sistemas concorrentes. Os sistemas que possuem estas características são chamados de sistemas reativos, e, cada vez mais, busca-se o uso de técnicas avançadas e de ferramentas que dêem suporte ao seu processo de validação[15].

Sistemas concorrentes são definidos como aqueles que são executados de forma simultânea, porém de forma independente.

O uso de métodos formais de verificação exige que o sistema a ser testado seja modelado através de uma especificação precisa e não ambígua. Este modelo deve representar o possível comportamento deste sistema.

Além disso, as propriedades que definem a correção do sistema também devem ser escritas através de uma especificação formal. Nesta especificação, devem estar representadas as propriedades esperadas para o sistema, ou seja, aquelas que ele deve, e/ou, aquelas que ele não deve possuir.

Baseado nestas duas especificações, é possível verificar, através de métodos formais, se o possível comportamento está de acordo com as propriedades desejadas para o sistema.

Uma vez que este tipo de verificação é feita através de métodos matemáticos, o termo "de acordo" se torna preciso, fazendo com que seja provada a correção, ou identificado o erro do sistema.

Para que tudo isso possa ser realizado, é importante fazer com que a descrição do comportamento do sistema seja clara e sem ambiguidades.

Linguagens formais que possuem uma semântica bem definida estão sendo cada vez mais utilizadas para este objetivo. Essas linguagens não só possuem uma semântica precisa, permitindo que propriedades computacionais possam ser descritas, como também fornecem meios para que uma manipulação e análise automática possa ser realizada.

Desta forma, em testes convencionais, feitos de forma manual, um sistema pode ser avaliado com diferentes possibilidades de entrada. Porém, utilizando-se a verificação formal, que é feita de forma automática, pode-se estabelecer a correção do sistema para todas as possíveis situações de entrada.

Nas últimas décadas, a lógica temporal tem sido significativamente empregada para a especificação e verificação formal de sistemas[15].

2.2 Lógica Temporal

A lógica temporal é uma extensão da lógica clássica, sendo um tipo de lógica específica para declarações que envolvam a noção de passagem de tempo. Ela descreve a ordenação de eventos, sem que seja necessário citar o tempo explicitamente[15].

Enquanto a lógica booleana é adequada para descrever situações estáticas, a lógica temporal permite expressar situações que poderão ocorrer durante uma sequência de tempo. Seus operadores possibilitam construções linguísticas (com o uso de advérbios “sempre”, “até”, etc) de tal forma que a formalização da lógica fique bem parecida com a linguagem natural[16].

A lógica temporal é utilizada para declarar, de maneira formal, as propriedades sobre a execução de um sistema.

Os operadores utilizados pela lógica booleana também são utilizados pela lógica temporal. Ou seja, as constantes ”true”e ”false”, e os operadores apresentados a seguir:

$\neg$ - operador de negação;

$\vee$ - operador OU;

$\wedge$ - operador E;

$\Rightarrow$ - implicação;

$\Leftrightarrow$ - se, e somente se.

2.2.1 Operadores Temporais

Os operadores temporais permitem que sejam realizadas declarações sobre momentos no futuro. Os operadores da lógica temporal e seus significados estão descritos a seguir.

A expressão

$$\langle M,i\rangle \models \varphi$$

será verdadeira se, e somente se, a propriedade φ for satisfeita no tempo i do modelo M.

O operador *início*, mostrado na figura 2.3, indica início dos tempos

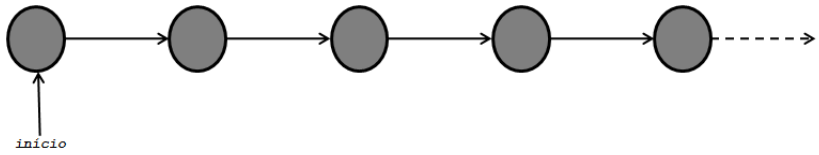

Figura 2.3: Operador Temporal *início*

$$\langle M,i\rangle \models inicio \text{ se, e somente se, } i = 0$$

O operador $\bigcirc$ indica que a propriedade φ será verdadeira no próximo estado de M. A figura 2.4 mostra um diagrama com o conceito deste operador.

$$\langle M,i\rangle \models \bigcirc\varphi \text{ se, e somente se, } \langle M,i+1\rangle \models \varphi$$

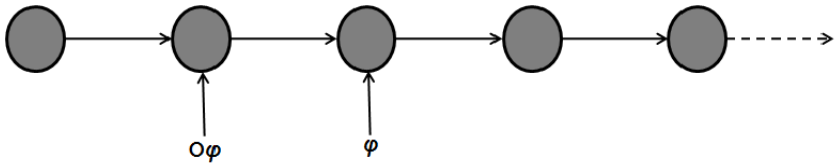

Figura 2.4: Operador Temporal próximo estado.

O operador $\diamondsuit$ indica "em algum estado no futuro". Este operador indica uma inderterminação sobre em que estado a propriedade φ será verdadeira. Porém, este operador garante que em algum no futuro, a propriedade φ será satisfeita no modelo M. Na figura 2.5 está o diagrama com o conceito.

$$\langle M,i\rangle \models \diamondsuit\varphi \text{ se, e somente se, existe } j \text{ tal que } (j \geq i) \text{ e } \langle M,j\rangle \models \varphi$$

O operador $\square$ indica que φ será verdadeiro, no próximo, e em todos os estados no futuro do modelo M. O diagrama da figura 2.6 representa o conceito do operador.

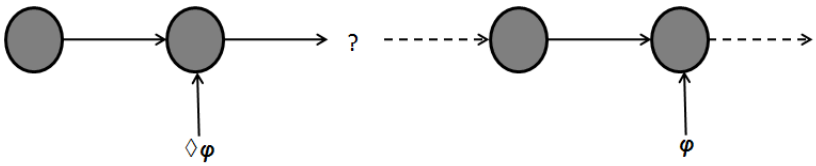

Figura 2.5: Operador Temporal em algum estado no futuro.

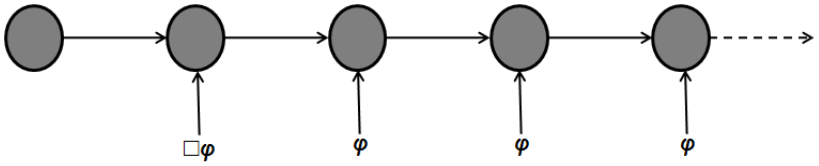

Figura 2.6: Operador Temporal Sempre.

$\langle M,i\rangle \models \Box\varphi$ se, e somente se, para todo j tal que $(j \geq i)$ e $\langle M,j\rangle \models \varphi$

Os operadores $\bigcirc$, $\Diamond$ e $\Box$ são chamados de operadores unários da lógica temporal. Em seguida, serão apresentados os operadores binários que relacionam duas sub-fórmulas temporais. O operador $\bigcup$ caracteriza, no modelo M, a situação onde uma propriedade φ é verdadeira até o momento em que outra propriedade ψ, se torne verdadeira. Esta situação pode ser visualizada no diagrama da figura 2.7.

$\langle M,i\rangle \models \varphi \bigcup \psi$ se, e somente se, existe j tal que $(j \geq i)$ e $\langle M,j\rangle \models \psi$ e, para todo k, se $(i \leq k < \mathrm{j})$, então $\langle M,k\rangle \models \varphi$

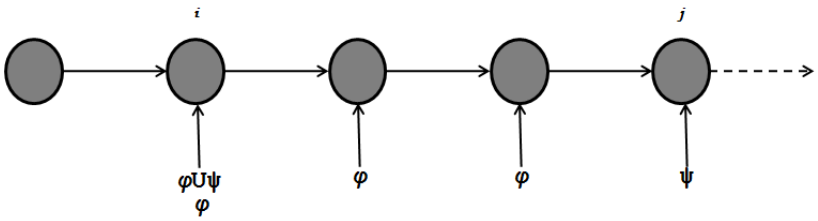

Figura 2.7: Operador Temporal Até.

O operador "a menos que" W é parecido com o operador $\bigcup$. A expressão $\varphi W \psi$, no modelo M, indica que ψ pode não ocorrer, e nesse caso, a propriedade φ será verdadeira para sempre. A figura 2.8 mostra o diagrama com o conceito deste operador.

$\langle M,i\rangle \models \varphi W \psi$ se, e somente se, $\langle M,i\rangle \models \varphi \bigcup \psi$ ou $\langle M,i\rangle \models \Box\varphi$

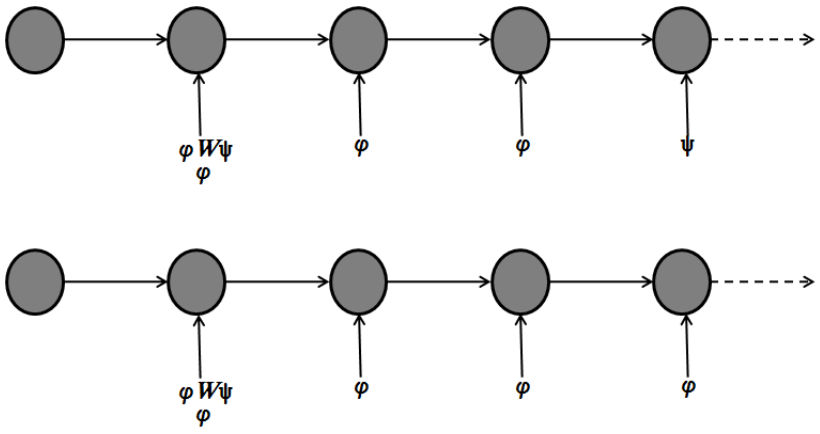

Figura 2.8: Operador Temporal A menos que.

2.2.2 Autômatos Finitos e a Lógica Temporal

Os modelos utilizados até agora para apresentar os operadores da lógica temporal foram infinitos, discretos, com sequência linear, e com um estado inicial bem identificado.

Entretanto, admitindo-se que nesses modelos existam uma quantidade finita de propriedades, é possível definir um conjunto finito de estados e utilizar para representar cada estado, um símbolo diferente.

Estando cada estado representado por um símbolo, o modelo se torna uma sequência de símbolos, ou seja, uma string. Entendendo o modelo como uma string, pode-se então aproveitar uma grande quantidade de conhecimento anterior dos chamados autômatos finitos[15].

Desta forma, uma máquina de estados finita pode ser utilizada para representar os modelos da lógica temporal. A seguir, apresenta-se uma breve descrição do conceito de autômatos finitos, muito utilizado na Ciência da Computação.

Autômatos Finitos

Um autômato de estados finitos pode ser descrito como um sistema que partindo de um estado inicial, aceita uma sequência de caracteres, e que a medida que cada caracter é aceito, muda de estado, de acordo com uma função de transição. Ele atinge o estado final ao terminar de ler toda a sequência de caracteres. Logo, um autômato de estados finitos M é representado por:

$$M = \langle A,\, S,\, \delta,\, I,\, F \rangle$$

onde:

- A é o conjunto finito de símbolos;

- S é o conjunto finito de estados;
- $\delta \subseteq S\ X\ A\ X\ S$ são as relação das transições;
- $I \subseteq S$ é o conjunto com os estados iniciais;
- $F \subseteq S$ é o conjunto com os estados finais.

Este tipo de autômato aceita uma quantidade finita de strings construída a partir do alfabeto A. Então, a sequência de símbolos é aceita começando de um estado inicial I, movendo pelos estados segundo δ e, depois de lido o último símbolo da sequência, termina em um dos estados finais F.

Como os modelos da lógica temporal aplicam-se também a sequências infinitas, utiliza-se o autômato Büchi, um tipo de automatos de estados finito que é capaz de representar um comportamento infinito.

Um autômato Büchi pode ser utilizado para modelar tanto o comportamento de programas, principalmente os sistemas reativos que têm como característica o processamento contínuo, quanto fórmulas da lógica temporal[15].

Um conceito que se apresenta neste momento é o de caminho (*path*). Num autômato A, um *path* é a sequência δ, finita ou infinita, de transições (e_i,t_i,e'_i) de A, sendo e, um estado, e t, uma transição. O estado e'_i representa o estado seguinte a e_i, ou seja, $e'_i = e_{i+1}$. O i-ésimo estado de δ, que possui a notação $\delta(i)$ é o estado e_i alcançado após i transições.

A partir do conceito de *path*, apresenta-se outros operadores da lógica temporal.

Aφ - a partir daqui, φ será verdadeiro em todos os *paths* futuros;

Eφ - a partir daqui, φ será verdadeiro em algum *path* no futuro.

Estes operadores são muito utilizados na verificação de modelos através de expressões do tipo:

A$\Box$**safe** - em todos os *paths* futuros safe será verdadeiro;

E$\bigcirc$**ativo** - existe um *path* em que no próximo estado ativo será verdadeiro;

A$\Diamond$**termina** - em todos os *paths* existe um momento onde termina será verdadeiro.

Uma vez apresentados os operadores da lógica temporal, na seção seguinte, será mostrada como esta lógica pode ser utilizada para representar o comportamento de um sistema.

2.2.3 Descrição de Comportamentos

Como tanto um programa, quanto a lógica temporal descrevem um comportamento sequencial ao longo do tempo, a lógica temporal pode ser utilizada para expressar o comportamento de um programa,

Seja um programa com a seguinte sequência

```
comando1;
comando2;
comando3;
```

O comportamento deste trecho de código pode ser expresso através dos operadores temporais como se segue:

$$start \Rightarrow \bigcirc comando1 \wedge \bigcirc\bigcirc comando2 \wedge \bigcirc\bigcirc\bigcirc comando3$$

Da mesma forma, propriedades de programas podem ser expressas utilizando-se operadores temporais. Seja o trecho a seguir:

```
x = 0;
while (x < 4)
{
   x = x + 1;
}
```

Pode-se constatar que este trecho de programa, ao final de sua execução, atende as seguintes propriedades:

$$\Box \neg(x > 4)$$

$$\Diamond (x = 4)$$

Com esses pequenos exemplos, procurou-se mostrar que uma expressão com operadores da lógica temporal pode descrever o comportamento e propriedades de um programa.

Uma vez que tem-se uma fórmula da lógica temporal que expressa características de um programa, é natural que se queira verificar as propriedades deste programa. O uso de métodos formais, particularmente a técnica chamada *Model Checking* é que tem sido utilizada para se alcançar este objetivo.

O problema do Model Checking pode ser sintetizado como a resposta para a seguinte pergunta: um modelo M satisfaz a propriedade φ?

2.3 Verificação de Modelos - *Model Checking*

Dada uma fórmula temporal, φ, que especifica uma propriedade que se deseja verificar em um sistema. Seja Σ o conjunto de todas as possíveis execuções, σ, deste sistema.

No Model Checking, como dito anteiormente, o que se deseja verificar é:

$$\Sigma \models \varphi$$

ou

$$\forall \sigma \in \Sigma, \langle \sigma, 0 \rangle \models \varphi$$

Este é o algoritmo do Model Checking[15], e está representado no diagrama da figura 2.9.

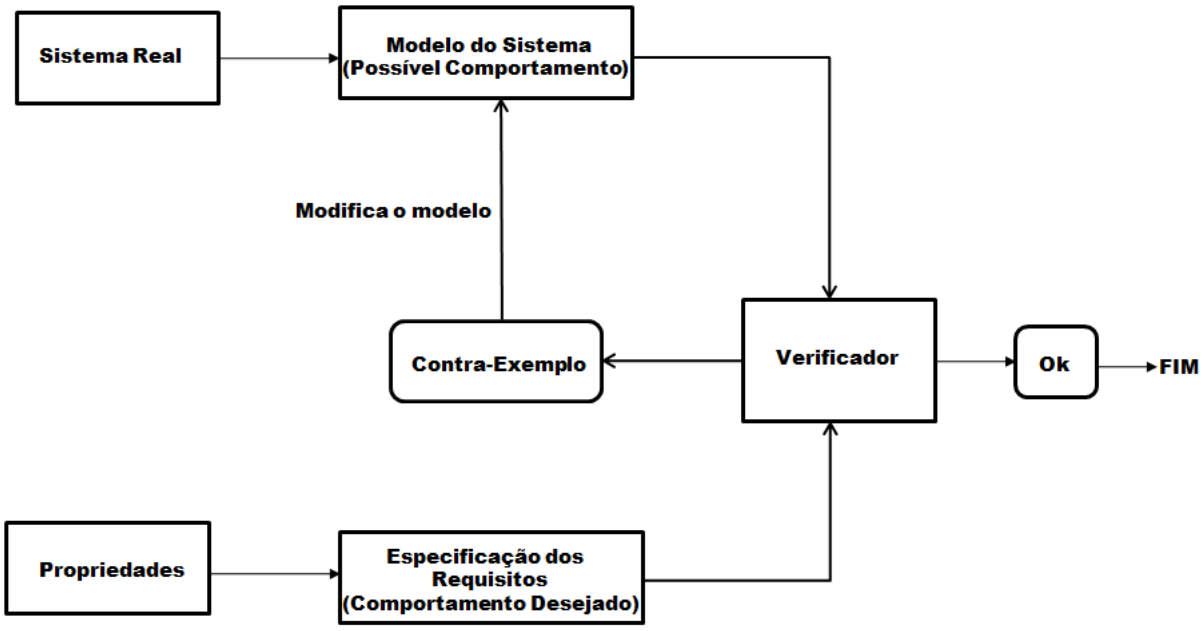

Figura 2.9: Processo do Model Checking

O Model Checking é uma técnica de verificação que faz uso de algorítmos executados em ferramentas computacionais de modo a verificar a correção de um sistema. Seu embasamento teórico é baseado na correspondência entre a Lógica Temporal, Automato Büchi e Linguagens Formais[17].

O usuário fornece como entrada a descrição do modelo do sistema (o possível comportamento) e a descrição das propriedades esperadas. A verificação é feita de forma automática pela máquina. Esta ferramenta verifica automaticamente a validade de propriedades acerca do comportamento de sistemas.

Para realizar a validação das propriedades seguem-se três passos a seguir[14]:

Modelagem É a conversão do sistema para a linguagem formal empregada pela ferramenta utilizada. A modelagem deve contemplar todas as propriedades essenciais do sistema para que se possa verificar a correção do mesmo, contudo, também deverá possuir abstrações de detalhes do sistema que não afetem a correção das propriedades a serem verificadas.

Especificação Antes da verificação, é necessário se estabelecer que propriedades o sistema deverá atender para que seja considerado correto. A especificação das propriedades, como visto anteriormente, é feita através de lógica temporal. O model checking proporciona meios para avaliar se um sistema atende a determinados requisitos, mas é impossível determinar se uma dada especificação cobre todas as propriedades que o sistema deve atender.

Verificação Neste passo, já se têm as propriedades e o modelo. Assim, aplica-se o verificador e consegue-se garantir se o modelo possui, ou não, as propriedades desejadas. Caso todas as propriedades sejam verdadeiras, então o modelo está correto. Caso não obedeça a alguma propriedade, então o verificador gera o contra-exemplo mostrando a condição da não verificação de determinada propriedade. Desta forma, pode-se detectar o erro e realizar a correção do modelo. Em tese, a verificação é totalmente automática. Entretanto, na prática necessita de auxílio humano para algumas atividades. Uma delas é a análise do resultado. Principalmente se for negativo. Porque nesse caso, deverá ser feita uma análise da condição do sistema que levou ao erro. O que, em muitos casos, pode acarretar em modificações no projeto. O erro também pode ter sido gerado por erro na modelagem ou na especificação.

Esse processo deve ser feito até que o sistema obedeça todas as propriedades. Deste modo, realiza-se um ajuste na especificação.

Existem várias razões para que métodos formais ainda não estejam sendo utilizados pela indústria, e a mais citada é a de que os métodos para a especificação dos requisitos e os recursos e ferramentas necessárias para a realização do processo, somente são conhecidos por especialistas. Além disso, considera-se difícil a identificação da abstração mais apropriada para cada sistema a ser testado[18].

2.4 Ferramentas

A seguir estão citadas seis ferramentas que estão sendo utilizadas para a verificação de sistemas. Todas elas são ferramentas acadêmicas, e de uso livre[16].

- SMV - Symbolic Model Verifier, desenvolvido na Universidade Carnegie-Merlon, Pittsburg, PA nos Estados Unidos. Possui limitações uma vez que

dispõe de poucas estruturas para simulação.

- DESIGN/CPN - Coloured Petri Nets, foi inicialmente desenvolvido pela Meta Software Corp., Cambridge, MA, nos Estados Unidos, e pelo grupo Cpn da Universidade de Arhus na Dinamarca. Hoje está sendo distribuído e mantido pelo grupo Cpn. É uma ferramenta que possui interface gráfica e utiliza o conceito de redes de Petri;

- SPIN - desenvolvido pelos Laboratórios Bell, New Jersey, nos Estados Unidos. Será melhor descrito no capítulo três;

- UPPAAL - Timed Systems, desenvolvimento conjunto entre o laboratório de Ciência da Computação da Universidade de Aalborg na Dinamarca e o departamento de Sistemas de Computação da Universidade Uppsala na Suécia. Possui poucos recursos para modelar a sincronização e não é capaz de verificar propriedades expressas em Lógica Temporal.

- KRONOS - Model Checking of Real Time Systems desenvolvido pela VERIMAG - Universidade de Grenoble. Possui uma interface pouco amigável.

- HYTECH - Linear Hybrid Systems, desenvolvido pela Universidade de Cornell, e pela Universidade da Califórnia. Não possui modo simulação, não é capaz de entender propriedades expressas em lógica temporal.

Nesta tese, optou-se por utilizar o verificador SPIN. Para que o SPIN realize sua tarefa, a especificação do modelo a ser verificado deve ser feita em uma linguagem chamada PROMELA. Esta linguagem é muito parecida com a linguagem C, fazendo com que fosse facilitada tanto a modelagem dos sistemas, quanto a representação das propriedades que se deseja verificar, uma vez que estas propriedades podem ser expressas tanto em PROMELA, quanto em lógica temporal.

2.5 Comentários

Neste capítulo foram apresentados os principais conceitos da verificação formal de modelos. Procurou-se mostrar os operadores da Lógica Temporal, que serve de base para os algoritmos utilizados pelas ferramentas de verificação. Foi mostrado que tanto a descrição do comportamento dos programas, quanto a descrição das propriedades que eles devem atender, podem ser representadas através de expressões da Lógica Temporal. Também foi apresentada a técnica do Model Checking, e as principais ferramentas, de uso livre, disponíveis no meio acadêmico.

No próximo capítulo, será feita uma descrição dos principais comandos e características da linguagem PROMELA e do uso do verificador SPIN.

Capítulo 3

O Verificador SPIN e a Linguagem PROMELA

O SPIN é uma ferramenta computacional utilizada para verificação de modelos de sistemas. Para que esta ferramenta possa ser utilizada na verificação de modelos, o sistema em análise deve ser modelado em uma linguagem de especificação chamada PROMELA (PROcess MEta LAnguage). O nome SPIN foi originalmente escolhido por ser a abreviação de Simple Promela INterpreter [19].

O verificador SPIN pode ser utilizado em dois modos distintos: simulação e verificação. No modo simulação, o SPIN é utilizado para se obter uma análise rápida sobre o comportamento do sistema. No entanto, para que seja possível a identificação mais detalhada de falhas no sistema, a ferramenta deve ser utilizada no modo verificação.

Neste modo, a verificação das propriedades estabelecidas para o sistema é executada através de uma busca exaustiva de todos os estados alcançáveis do sistema. Caso seja encontrada uma não conformidade em relação a algumas dessas propriedades, uma condição de erro é apresentada.

Uma vez encontrada uma não conformidade, será através do modo simulação que a condição que leva o sistema a não atender aquela propriedade, possa ser identificada[20]. No SPIN, a simulação e a verificação são funções complementares.

Nas próximas seções, serão descritas as principais estruturas e características da linguagem PROMELA. As estruturas e características apresentadas neste capítulo são aquelas utilizadas na modelagem dos experimentos apresentados nos capítulos posteriores.

3.1 A Linguagem PROMELA

A linguagem PROMELA é uma linguagem de especificação que tem como objetivo facilitar a abstração do projeto de um sistema. Ela não é uma linguagem para implementação de sistemas, mas sim, para a sua descrição. Isso é possível, porque a ênfase dada na linguagem é na modelagem da sincronização entre os processos.

Em PROMELA, as estruturas básicas para a construção dos modelos são os processos assíncronos, os canais de comunicação (*message channels*), os comandos de sincronização, e as estruturas de dados [19]. A linguagem não dispõe de números em ponto flutuante, e existem poucas funções computacionais.

Outro ponto a ser observado é que a linguagem PROMELA possui muitos aspectos que normalmente não são encontrados nas linguagens de programação convencionais. Estes aspectos têm como objetivo facilitar a construção de modelos de alto-nível dos sistemas.

Por exemplo, ela suporta a especificação de estruturas de controle não determinísticas e um conjunto de primitivas para comunicação entre processos. Entretanto, ela não dispõe de estruturas normalmente encontradas na maioria das linguagens de programação, tais como funções que retornam valor, ponteiros, etc.

A razão é simples: PROMELA não é uma linguagem de programação, mas uma linguagem para a construção de modelos para verificação de sistemas.

Um modelo para verificação difere em pelo menos dois importantes aspectos em relação a um programa escrito em linguagens de programação como Java, C ou C++, a saber:

- Representa uma abstração do projeto, possuindo apenas aquelas características do sistema que são relevantes para as propriedades que se deseja verificar.

- Normalmente, possui partes que não serão encontradas na implementação. Nele podem ser incluídos aspectos relativos às piores contingências possíveis que podem ser encontradas no ambiente que vai interagir com o modelo. Com isso, permite-se verificar a correção de uma determinada propriedade neste ambiente.

A seguir, apresenta-se a descrição dos tipos de dados, principais estruturas e características da linguagem PROMELA.

3.1.1 Tipos de Dados

Os tipos de dados disponíveis em PROMELA são:

- bool;

- bit;
- int;
- byte;
- short;
- chan;
- mtype.

A declaração de uma variável é feita através da informação do tipo, seguido da identificação (nome) da variável, terminando com um ponto e vírgula.

Exemplo:

```
tipo identificador(nome);
bool flag;
int state;
byte msg;
```

Dos tipos apresentados, somente os tipos *chan* e *mtype* serão descritos. Os outros tipos são similares aos encontrados na maioria das linguagens de programação.

3.1.2 Tipo *mtype*

O tipo *mtype* permite que uma variável possa armazenar valores simbólicos durante a execução do programa. A declaração:

```
mtype = {ok, nok, ack, nak, err}

mtype var;
```

indica que a variável *var* poderá receber os valores simbólicos *ok, nok, ack, nak,* e *err*.

3.1.3 Tipo chan

O tipo *chan* define um *message channel*, e é utilizado para modelar os canais de comunicação que os processos irão utilizar no modelo do sistema a ser verificado.

A declaração

```
chan rede = [16] of {short};
```

cria um *message channel* chamado *rede* que pode armazenar até 16 mensagens compostas por um valor ou variável do tipo *short*. Se a mensagem a ser enviada através do "*channel*" possuir mais de um campo, a declaração pode ser como o exemplo abaixo:

```
chan rede = [16] of {byte, int, short};
```

Nesta declaração o *message channel* armazena até 16 mensagens, cada uma delas composta por um valor ou variável do tipo byte, um int e um short.

Transmissão de Mensagem em um *message channel*

O comando:

```
rede!expr
```

envia o valor da expressão *expr* para o *message channel* rede, isto é, insere o valor da expressão no final da lista de mensagens do canal *rede*. O sinal de exclamação é o comando que deve ser utilizado por um processo para enviar uma mensagem através de um *message channel*.

Recepção de Mensagem em um *message channel*

O comando

```
rede?msg
```

recebe uma mensagem. O sinal de interrogação é o comando que deve ser utilizado por um processo para a recepção de uma mensagem em um *message channel*. A mensagem é retirada do início da fila do *message channel rede*, e armazenada na variável *msg*. As mensagens são retiradas do *message channel* através da forma *"first-in-first-out"*. A mensagem a ser enviada em um canal deve ter a mesma quantidade de parâmetros especificada na declaração do canal.

No exemplo a seguir, está sendo declarado um *message channel* chamado *net*, que pode armazenar somente uma mensagem, cada uma contendo três bytes.

```
chan net = [1] of {byte,byte,byte}
```

toda mensagem a ser transmitida através do canal *net* deverá obrigatoriamente conter três bytes.

Exemplo de uso dos comandos de transmissão e recepção de mensagens:

No processo Tx, cujo código encontra-se a seguir,

```
proctype Tx()
{
  byte var1, var2, var3;

  var1 = 01;
  var2 = 02;
  var3 = 03;

  net!var1,var2,var3
}
```

o comando *net!var1,var2,var3* envia a mensagem com os valores 01, 02, 03 através do *message channel net*.

Para que seja possível a recepção da mensagem por um outro processo, tem-se o exemplo de um outro processo, chamado de Rx, onde espera-se por uma mensagem no *message channel net*. Utilizando-se a função *eval*, a mensagem somente será retirada do canal, se possuir o primeiro byte com valor igual ao da variável *dst*, definida em Rx.

```
proctype Rx()
{
  byte dst, org, func;

  dst = 01;

  net?eval(dst),org,func
}
```

O comando *net?eval(dst),org,func* recebe a mensagem, ou seja, retira do *message channel net* se, se somente se, o valor do primeiro byte da mensagem for igual ao da variável *dst*.

Uma vez satisfeita esta condição, além da mensagem ser retirada do canal, o valor do segundo byte é armazenado na variável *org*, e o valor do terceiro byte da mensagem é armazenado na variável *func*.

No exemplo, a variável *org* receberá o valor 02 e a variável *func*, o valor 03.

Deve ser observado que o processo Rx estará bloqueado enquanto a condição avaliada pela função *eval* não for satisfeita.

A operação de envio é realizada somente quando o canal não estiver cheio. Igualmente, a operação de recepção é realizada somente quando o canal não estiver vazio.

Se um processo tentar a execução de um comando de recepção, e o canal estiver vazio, o processo será bloqueado até que uma mensagem seja inserida no canal.

Da mesma forma, se um processo tentar executar um comando de envio e o canal estiver cheio, o processo que tentou a transmissão também será bloqueado.

3.1.4 Tipo Processo

Em PROMELA, um processo é uma instância do tipo *proctype*, sendo utilizado para a definição do comportamento de uma parte do sistema.

Todo modelo deverá conter pelo menos um *proctype*. Ele é composto por declarações e um ou mais comandos. Os processos são sempre considerados como tipos globais.

No exemplo a seguir, tem-se a declaração de um processo chamado A, com uma variável local chamada *var*.

```
proctype A ( )
{
      byte var;

      var = 3;
}
```

O corpo do processo é escrito entre chaves. O processo acima tem uma variável local chamada *var* e um comando: a atribuição do valor 3 a esta variável.

O ponto e vírgula é um separador de comandos. O PROMELA aceita dois separadores de comandos: uma seta $->$ e o ponto e vírgula ';'. Ambos são equivalentes. Em alguns casos a seta é utilizada como uma forma de representar uma relação de causa. Isto pode ser observado no exemplo a seguir:

```
byte var = 6;
proctype A ( )
{
     (var == 3) -> var = 1;
}
proctype B ( )
{
     var = var - 1
}
```

Neste exemplo são declarados dois processos, A e B. A variável *var* é global, e está sendo inicializada com o valor seis. O processo A contém dois comandos separados pela seta. O processo B contém somente um comando que decrementa de um, o valor da variável *var*. Como as atribuições são sempre executadas, o processo B sempre será executado. Entretanto, o processo A somente será executado quando o valor da variável *var* atender a condição estabelecida, ou seja, quando *var* for igual a 3.

Execução de um Processo

A declaração de um *proctype* define o comportamento de um processo, porém não o executa. Na linguagem PROMELA, inicialmente, somente um processo é executado: o processo chamado *init*, que deve ser declarado explicitamente em toda especificação feita nesta linguagem.

Normalmente o processo *init* é utilizado para iniciar variáveis globais e instanciar processos. Para o exemplo acima, pode-se ter um processo *init* conforme o código a seguir:

```
init
{
   run A();
   run B();
}
```

O comando *run* é usado para se instanciar um processo.

Na declaração de um processo, ao se utilizar o prefixo *active*, este será instanciado sem que seja necessário o comando *run*. Assim, se em um modelo todos do processos tiverem o prefixo *active*, não será necessário a existencia de um processo *init*.

Se, como mostrado a seguir, no código do exemplo anterior fosse incluído o prefixo *active* nas definições dos processos

```
byte var = 6;
active proctype A ( )
{
     (var == 3) -> var = 1;
}
active proctype B ( )
{
     var = var - 1
}
```

não seria necessário a definição de um processo *init*.

Quando mais de um processo é instanciado, a execução de cada um deles é feita através de threads assíncronas. A sequência de execução dos comandos de cada processo é interrompida de forma aleatória para a execução de comandos de outros processos que também estão em execução.

Isto significa que, quando mais de um processo está sendo executado, não se pode afirmar qual deles será executado primeiro. Mesmo durante a execução de um processo, não se pode afirmar em que ponto do seu processamento, este será interrompido para a execução de um outro processo. Deste modo, a execução de processos é feita de forma aleatória.

3.1.5 Estrutura de Seleção

O exemplo utilizado para esta estrutura usa o valor das variáveis x e y para definir o que deve ser feito.

```
if
:: (x != y) -> comando1
:: (x == y) -> comando2
fi
```

No exemplo, a estrutura de seleção possui duas sequências de execução, cada uma delas está precedida por dois pontos, e somente uma delas será executada. A sequência a ser executada é a que tiver a condição satisfeita. Por isso esta condição é chamada de *guard*.

As condições são mutuamente exclusivas, mas elas podem não ser. No caso em que mais de uma condição seja satisfeita, apenas uma delas será executada de forma não determinística. Se todas as condições forem falsas, o processo é bloqueado até que pelo menos uma condição seja satisfeita. Não existem restrições quanto ao tipo de condição que pode ser usada como *guard*.

No trecho de código mostrado a seguir, é utilizado o *else* na estrutura de seleção.

```
if
:: (x == 3) -> comando1
:: (x == 5) -> comando2
:: else -> comando3
fi
```

O comando3 será executado quando nenhuma das condições anteriores forem satisfeitas, ou seja, o comando3 somente será executado quando a variável x for diferente de 3 e de 5.

Uma forma de se utilizar o comportamento não determinístico desta estrutura é fazer com que todas as opções sejam verdadeiras. Assim, durante uma execução, não se pode afirmar qual das opções será executada.

```
if
:: 1 -> comando1
:: 1 -> comando2
:: 1 -> comando3
fi
```

No exemplo anterior, como todos os *guards* são verdadeiros, somente um deles será executado, entretanto não se pode afirmar durante uma execução, qual o comando que será executado.

3.1.6 Desvio Incondicional

Outra maneira de se interromper uma estrutura de repetição é através do desvio incondicional, o comando *goto*. No exemplo a seguir, a repetição é encerrada utilizando-se o desvio incondicional. O comando *skip* ao ser executado não produz efeito algum no programa.

```
byte cont;

proctype contador ( )
{
    do
    :: (cont != 0) ->
         if
         :: cont = cont + 1
         :: cont = cont - 1
         fi
    :: (cont == 0) -> goto L1
    od;

L1: skip
}
```

3.1.7 O Label de End-State

Na verificação, o único estado considerado válido para que um processo termine é o último comando do seu código.

Se durante uma verificação, um processo não chegar até o seu último comando, o SPIN interpretará como erro.

Este erro é apresentado como *invalid end-state*. Indica que o programa terminou, porém aquele processo não chegou ao final de seu processamento.

Na modelagem de alguns sistemas, podem existir processos que, dependendo da situação, não alcancem o seu último comando. Nestes casos, a indicação de *invalid end-state* não pode ser considerada como erro.

Para que isso não ocorra, antes do comando em que se espera que o processo possa terminar, deve ser inserido um *label* que tenha o prefixo *end*.

3.1.8 Timeout

Em PROMELA, a palavra timeout representa a modelagem de uma condição especial que permite que um processo termine se uma condição a que ele estiver aguardando nunca se torne verdadeira.

3.1.9 Verificação de Propriedades

A estrutura da linguagem PROMELA que permite a verificação de propriedades é o comando *assert*, que tem a forma geral:

```
assert (condicao)
```

O comando *assert* sempre será executado. Se, no momento em que o comando for executado, a condição especificada for verdadeira, o comando não tem efeito algum. Entretanto, se durante a simulação ou a verificação, a condição avaliada pelo *assert* não for verdadeira, um erro será identificado.

Este comando será utilizado de modo que a condição a ser avaliada seja uma propriedade que se deseja verificar naquele ponto do código. Se a propriedade definida no comando *assert* não for confirmada, será exibido um erro de *assertion violated* indicando que aquela propriedade não é válida naquele ponto do programa.

3.2 Exemplo: um comando de um Centro de Operação para uma Subestação

Esta seção tem como objetivo mostrar, através de um exemplo bem simples, as características da linguagem PROMELA e as funcionalidades do verificador SPIN.

Os modelos e resultados dos experimentos no uso destas ferramentas no setor elétrico serão descritos nos capítulos 5 e 6.

Neste exemplo será modelada uma situação fictícia de envio de um comando de abertura de um disjuntor de um Centro de Operação para uma Subestação. O centro envia o comando de *trip* para a subestação, e esta responde com uma mensagem *ack* de reconhecimento.

No modelo, o centro de controle, ao receber a mensagem de reconhecimento da subestação, assume que o disjuntor foi aberto e verifica esta condição através do comando *assert*. Entretanto, isto nem sempre é verdadeiro. O comando somente será executado se a chave local/remoto estiver na posição remoto, caso contrário, o controle não será executado.

Volta-se a ressaltar que se trata de um caso fictício, e que tem como único objetivo apresentar características da linguagem e do verificador.

No modelo, foram definidos dois processos, um chamado CO que representa o centro de controle, e outro SE, que representa a subestação.

Na figura 3.1 está apresentado diagrama do modelo para a comunicação entre os processos CO e SE.

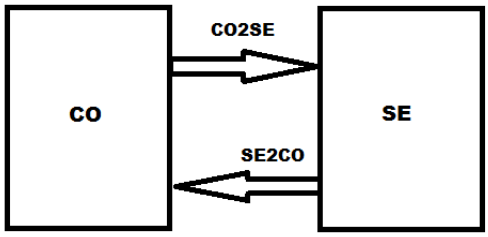

Figura 3.1: Diagrama da comunicação Centro-Subestação.

Na figura 3.2 encontra-se o código do programa em PROMELA na ferramenta utilizada neste exemplo que é o JSpin.

Entre as linhas 1 e 5, são realizadas as declarações das variáveis globais utilizadas no modelo. Elas representam o estado do disjuntor e da chave *stDj* e *stCh*, respectivamente. Assume-se o valor 0 para o estado aberto e 1 para o estado fechado. Na linha 3 tem-se a definição do tipo *mtype* com os valores simbólicos *trip* e *ack*.

Devem ser destacadas as declarações das linhas 4 e 5. São os canais CO2SE e SE2CO. Ambos podem armazenar somente uma mensagem de um tipo *mtype* definido na linha 3.

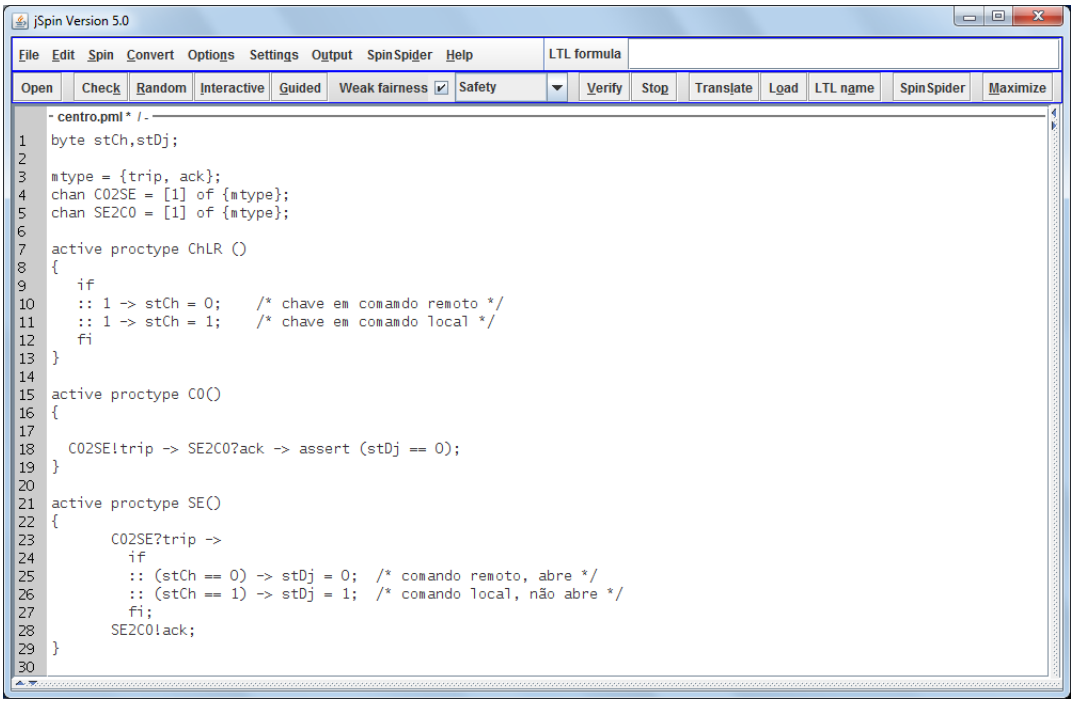

Figura 3.2: Código do Modelo em PROMELA, na ferramenta JSpin.

Da linha 7 até a linha 13 está definido o processo *ChLR*. Neste processo, se utiliza a estrutura de seleção de uma forma que qualquer uma das opções podem ser executadas de maneira não determinística. Como a condição utilizada como *guard* é o valor 1, ou seja, true, a variável *stCh* poderá receber tanto o valor 0, quanto o valor 1.

Da linha 15 até a linha 19 tem-se o código do processo CO. Este processo, através do canal CO2SE, envia um comando de *trip* para o processo SE. Em seguida aguarda a resposta de SE. Ao receber a resposta, verifica, através do *assert*, a condição de stDj = 0.

O processo SE, definido da linha 21 até a linha 29, ao ser instanciado aguarda pelo comando de *trip* no canal CO2SE. Ao recebê-lo, de acordo com estado da chave, abre ou fecha o disjuntor e envia a resposta de reconhecimento para o processo CO.

Ao se realizar a verificação deste modelo, o SPIN, depois de avaliar todos os estados do modelo, apresentará o erro através da mensagem de *assertion violated* que pode ser visualizada na primeira linha do lado direito da janela que está mostrada na figura 3.3.

Na verificação, quando o SPIN encontra um erro, ele apenas indica que existe um erro. A condição que faz com que este erro ocorra, só pode ser visualizada através de uma simulação guiada. Este tipo de simulação utiliza o arquivo *.trail*, que é gerado durante o processo de verificação, para apresentar as condições que levaram à não conformidade apresentada pelo *assertion violated* da verificação.

Ao se realizar a simulação guiada pode-se observar a condição que faz com que

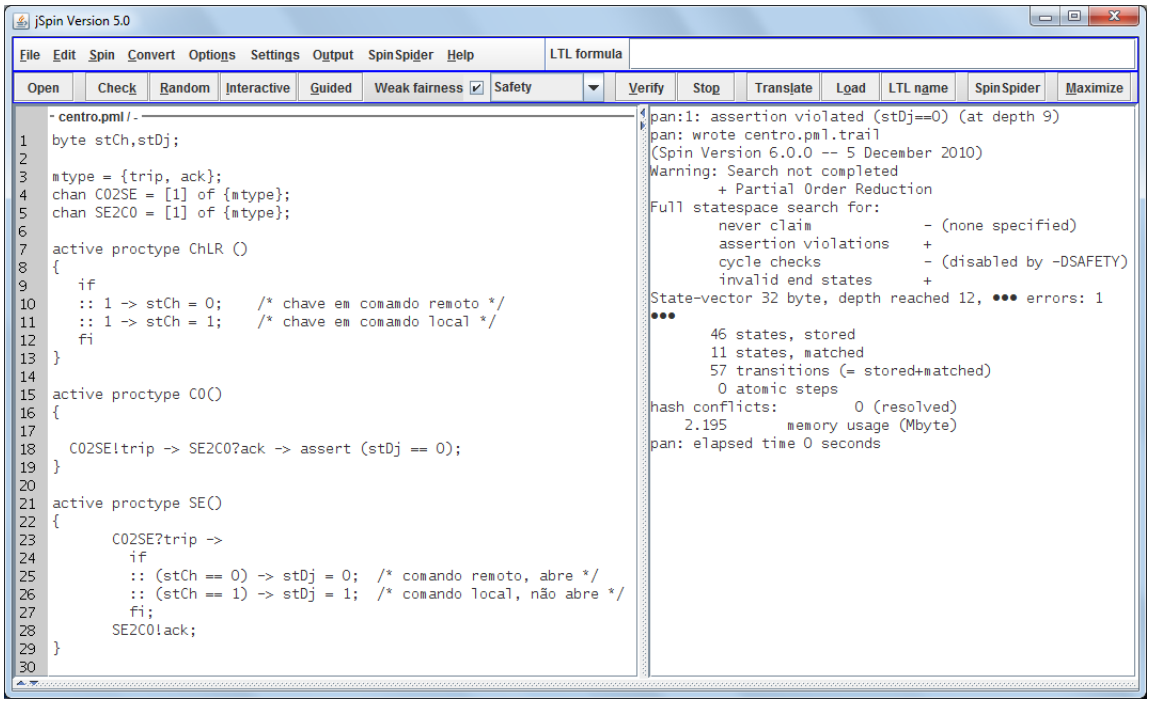

Figura 3.3: Indicação de erro na verificação do modelo.

a propriedade desejada não seja satisfeita, conforme mostrado no lado direito da figura 3.4.

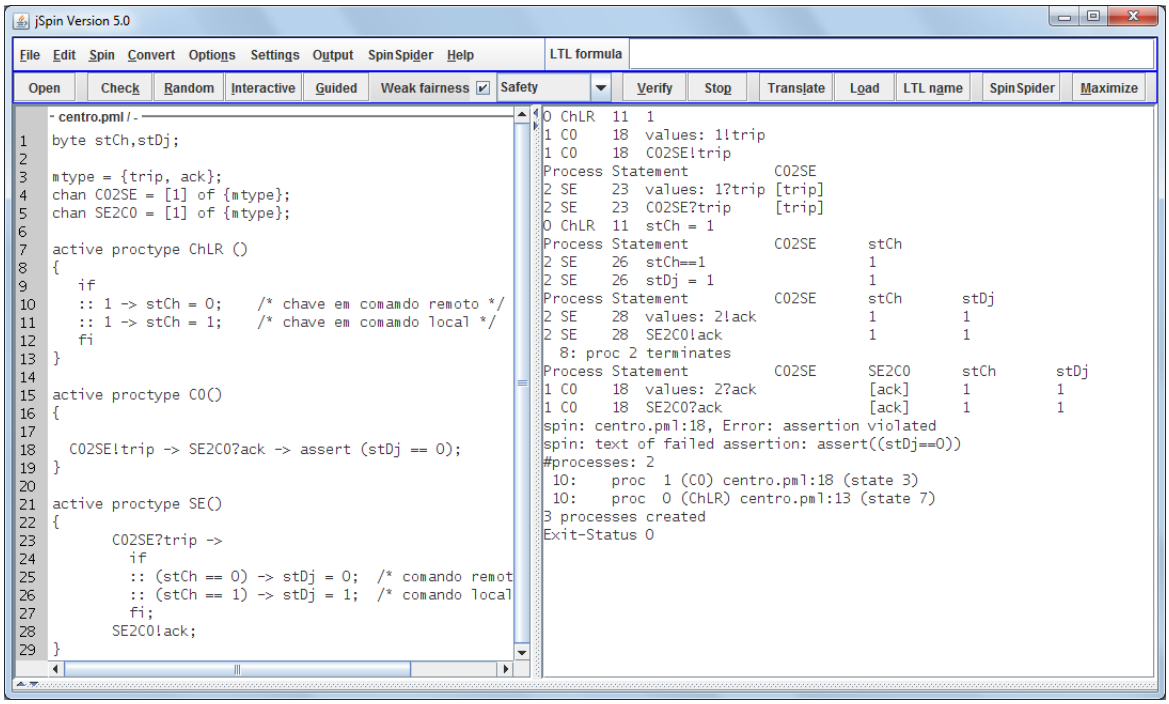

Figura 3.4: Simulação guiada pela arquivo .trail

Lá pode ser visualizada a sequência que leva ao erro. Na primeira coluna de cada linha tem-se o identificador do processo, na segunda coluna, o nome do processo, na terceira a linha que foi executada.

Desta maneira, tem-se que primeiramente o processo *ChLR* executou a linha 11, fazendo a variável *stCh* igual a 1, ou seja, comando local.

Na segunda linha da simulação guiada, tem-se a execução da linha 18 do processo *CO*. Lá o processo envia o comando *trip* para o processo *SE*. Em seguida, tem-se a recepção do comando *trip* pelo processo *SE* e como stCh é igual a 1, a execução da linha 26, ou seja, a atribuição do valor 1 a variável *stDj*, e o envio do reconhecimento na linha 28.

O erro se dá na linha 18 do processo *CO*, onde se pode observar o valor de cada variável do modelo na condição de erro. Assim, na figura, tem-se *SE2CO* com o *ack*, *stCh* com 1, chave em local, *stDj* em 1, disjuntor fechado. Desta forma, o verificador SPIN indica a condição de erro que foi provocada neste exemplo.

3.3 Comentários

Neste capítulo foi apresentada a linguagem PROMELA, com seus principais tipos e estruturas. Também foram mostradas, através de um exemplo bem simples, as principais características do verificador SPIN. Com o objetivo de demonstrar algumas das suas funcionalidades, foi realizada uma verificação e uma simulação do exemplo apresentado. Será com a utilização destas ferramentas que serão apresentados os modelos e experimentos realizados neste trabalho.

Capítulo 4

Proteção de Sistemas Elétricos

Nos Sistemas Elétricos de Potência existem um conjunto de sistemas auxiliares que ajudam na obtenção da segurança, confiabilidade e qualidade do fornecimento de energia. Um dos mais importante é o Sistema de Proteção, tipicamente, projetado para servir principalmente a três funções:

- Proteger seres humanos, tanto aqueles que trabalham nas atividades de operação e manutenção em empresas de energia elétrica como os usuários da energia em suas casas, ou que passem perto de áreas energizadas (subestações, linhas de transmissão, postes e outros locais);
- Isolar rapidamente uma falha logo após a sua ocorrência;
- Proteger os equipamentos de possíveis falhas decorrentes de valores anormais de tensão ou corrente.

Os Sistemas de Proteção têm como princípio básico a detecção de uma falha do modo mais rápido possível, conseguindo com isso, isolar a parte afetada fazendo com que o restante do sistema elétrico continue em funcionamento.

Tais sistemas podem ser divididos em duas categorias: proteção de equipamentos e proteção do sistema. A proteção de equipamentos é aquela que visa um equipamento específico tal como uma linha, um transformador, e outros[2]. A proteção do sistema é aquela que tem como função a detecção de ocorrências anormais no sistema elétrico e consequentes de ações corretivas automáticas com o objetivo de manter a sua integridade, garantindo o suprimento de energia de forma aceitável na maior parte possível do sistema.

Os relés de proteção são os principais equipamentos de um sistema de proteção. Estes equipamentos, analógicos ou digitais, são responsáveis pela análise das grandezas elétricas associadas à rede elétrica e pela implementação da lógica necessária para a atuação do sistema, caso algum distúrbio seja encontrado. A principal função

do relé é minimizar os efeitos dos curtos-circuitos e de outras condições anormais de operação.

Devido a esta importância para o sistema, os relés devem ser equipamentos extremamente confiáveis e robustos, pois suas funções somente serão exigidas em condições anormais de operação, e não fora destas.

A parte do sistema elétrico em que o relé deverá atuar no sentido de protegê-la é conhecida como zona de proteção.

4.1 Zonas de Proteção

A filosofia geral para o uso dos relés de proteção considera dividir o sistema elétrico em zonas separadas. Essas zonas são protegidas individualmente e, na ocorrência de falhas, são desconectadas de modo a permitir que o resto do sistema elétrico continue com o fornecimento de energia.

A figura 4.1 mostra um trecho com diferentes zonas de proteção. Deve se observar que essas zonas são sobrepostas, indicando que na ocorrência de uma falha, mais de um conjunto de relés irá operar.

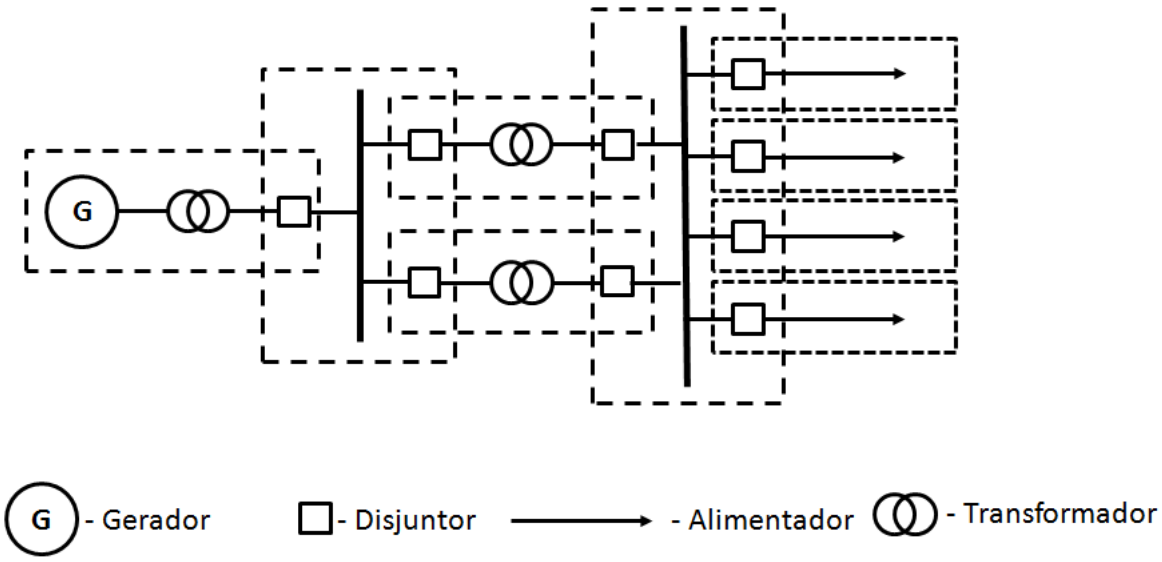

Figura 4.1: Zonas de Proteção

4.2 Proteção Primária e Backup

Na maioria dos casos, são previstas Proteções backup que têm como função cobrir possíveis falhas na proteção primária[21].

Proteção Primária - Opera toda vez que um elemento detectar falha no sistema elétrico;

Proteção Backup - Opera quando, por alguma razão, a proteção primária não funciona.

4.3 Principais Características

O sistema de proteção não é composto apenas pelo relé, mas por um conjunto de subsistemas integrados que interagem entre si com o objetivo de produzir a melhor atuação sobre o sistema elétrico, ou seja, isolar a área ou equipamento defeituoso sem que este comprometa o restante do Sistema Elétrico de Potência. Estes subsistemas são formados basicamente por relés, disjuntores, transformadores de instrumentação (Transformadores de Potencial - TP e Transformadores de Corrente - TC) e pelo sistema de suprimento de energia.

Os equipamentos utilizados nos sistemas de proteção devem observar certas características básicas, tais como[21]:

- Confiabilidade - Assegurar que a proteção atuará de forma correta quando requisitada, sendo capaz de diferenciar as situações de falha e condição normal de operação.
- Seletividade - Garantir a continuidade do fornecimento de energia, para que em condições de falha, seja desligada a menor parte possível do sistema.
- Velocidade de Operação - Minimizar o tempo de atuação de modo que a falha não venha a danificar os equipamentos da rede.
- Custo - Máxima proteção com o menor custo

Como na prática é impossível se atender a todos pontos mencionados acima. Normalmente, num sistema de proteção real, se estabelece um compromisso entre cada um deles.

Os relés de proteção são equipamentos usados há várias décadas. A partir dos anos 90, a eletrônica começou a fazer parte dos equipamentos de proteção nas empresas de energia elétrica. A primeira geração foi composta por relés eletrônicos baseados em tecnologia analógica. A segunda geração, com o desenvolvimento da eletrônica digital e dos microprocessadores, foi marcada pelo uso dos relés baseados em tecnologia digital.

Essa geração de relés representou uma mudança de paradigma na área, fazendo com que todas as empresas passassem a utilizar relés digitais em suas instalações. No final dos anos 90, com a tecnologia digital consolidada, surgem os primeiros Sistemas Digitais de Proteção e Controle. Esses sistemas são baseados nas seguintes condições:

- Predominância da eletrônica digital;
- Utilização de relés de proteção a base de microprocessadores e memórias;
- Utilização de terminais microprocessados integrando funções de controle e proteção;
- Inteligência local para execução de lógicas e automatismos;
- Utilização da tecnologia de redes de computadores para comunicação entre equipamentos na subestação;
- Utilização de microcomputadores para o controle local da subestação.
- Aquisição e Envio de dados ao Centro de Operação da empresa.

Os Sistemas de Proteção necessitam continuamente evoluir para assegurar os níveis de confiabilidade exigidos.

Com a possibilidade de comunicação entre os equipamentos dentro de uma subestação, houve o surgimento da norma IEC61850 como um padrão global para o intercambio de informações entre os equipamentos [22].

4.4 Norma IEC 61850

A norma IEC 61850 define uma arquitetura de comunicação de equipamentos dentro de uma subestação, sendo parte da arquitetura de referência para sistemas elétricos de potência do Technical Committee 57 do IEC.

O seu uso permite a integração de todas as funções de proteção, controle, medição e monitoração. Utiliza comunicação Ethernet de alta velocidade. Também oferece padronização [23]: de modelos dos IEDs, dos serviços de comunicação, dos arquivos de configuração, e ainda comunicação peer-to-peer. Com a sua utilização, observa-se uma mudança na forma de interligação de equipamentos. Os conceitos que antes eram associados a arquitetura de computadores passaram a ser utilizados em ambientes de Sistemas Elétricos de Potência [24]. A comunicação entre equipamentos passou a ser uma característica fundamental nestes sistemas, motivando a busca por novas ferramentas para a elaboração dos projetos.

O modelo de dados do IEC 61850 está mapeado para vários protocolos tais como: MMS(Manufacturing Message Specification), GOOSE, e em breve para Web Services.

O IEC 61850 pode ser utilizado em redes TCP/IP de alta velocidade dentro de uma subestação de modo a garantir baixos tempos de respostas aos relés de proteção.

4.5 EPS - Esquema de Proteção de Sistemas

Há alguns anos surgiu um conceito ligado ao aspecto da proteção sistêmica e associado à segurança global do sistema elétrico. Este conceito diz respeito ao impacto que eventuais falhas podem ter sobre o fornecimento de energia elétrica para um grande número de consumidores. Começou a ganhar importância no cenário mundial, os esquemas de proteção de sistemas - EPS. Estes esquemas têm como principal objetivo a preservação da maior parte possível do sistema, evitando os desligamentos em cascata que provocam o colapso do suprimento a grandes áreas[25].

Na proteção executada pelos EPSs a preocupação maior é com a integridade operativa da rede.

Um tipo de EPS que emprega a tecnologia baseada em microprocessadores é chamada de ECS - Esquema de Controle de Segurança.

Este tipo de esquema é composto por Controladores Lógicos Programáveis (CLPs), Unidades Terminais Remotas (UTRs), "Intelligent Electronic Devices"(IEDs), podendo operar de forma isolada ou em rede de comunicação cobrindo e protegendo grandes áreas do sistema elétrico.

Os EPSs são capazes de identificar uma grande perturbação e executar as ações automáticas de controle necessárias para manter o sistema elétrico estável. Deste modo, evita-se que os efeitos da ocorrência sejam aumentados, ou se espalhem pelo sistema.

Isto só é possível através do monitoramento contínuo de estados e grandezas em um grupo de subestações e usinas, selecionadas permitindo que o esquema atue, quando necessário, em sequência à proteção convencional, comandando, por exemplo, um corte emergencial de carga e/ou geração ou, quando for o caso, provocando uma alteração na topologia da rede. Na figura 4.2 é apresentado um diagrama com a arquitetura de um EPS.

Atualmente, em várias partes do mundo, os EPSs representam um alternativa viável para estender a capacidade do sistema de transmissão.

Embora o uso de EPS se apresente como uma alternativa mais barata que a construção de nova infra-estrutura, traz consigo alguns riscos ou incovenientes:

1. Risco de falha na ativação, ou uma ativação acidental;
2. Risco na interação com outros EPSs de um modo indesejado;
3. Maior controle na gerenciamento, coordenação e manutenção.

Além disso, há uma escassez de ferramentas para simulação e testes para a realização de estudos de confiabilidade do EPS de modo que se possa avaliar tanto a

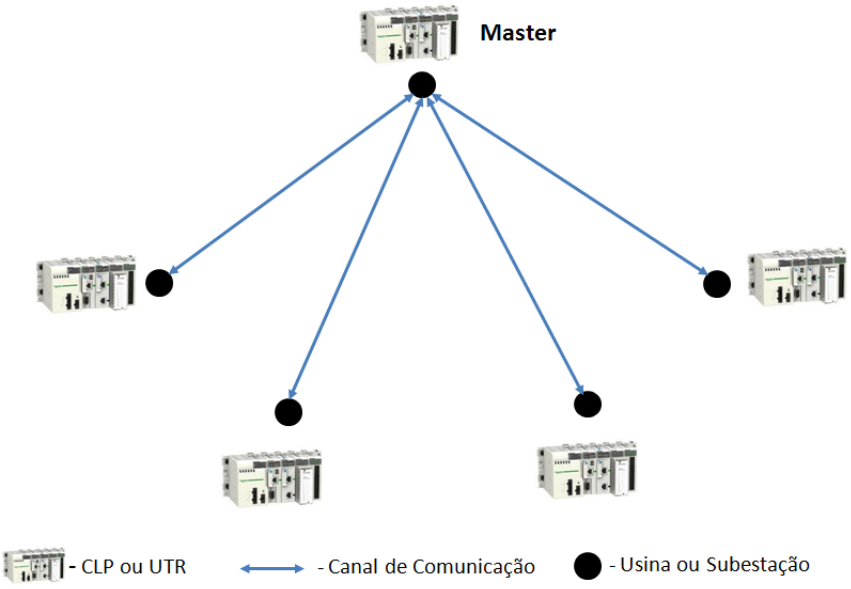

Figura 4.2: Arquitetura de um EPS

complexidade operacional que o EPS traz para o sistema, quanto as suas diversas vantagens econômicas e operacionais.

De qualquer forma, constata-se um aumento no uso de EPS's no mundo. A tabela 4.1 apresenta o resultado de três estudos realizados com empresas do setor elétrico nos últimos 20 anos. O resultado indica um aumento significativo no uso de EPS[3].

1989		1996		2009	
Entrevistadas	Esquemas	Entrevistadas	Esquemas	Entrevistadas	Esquemas
18	93	49	111	110	958

Tabela 4.1: Uso de EPS's

Além disso, existe um aumento na quantidade de fontes renováveis de energia, que estão sendo conectadas à rede elétrica. Por isso as aplicações de EPSs se tornaram críticas para permitir um rápida e econômica forma de se interconectar esta geração.

4.5.1 Falhas na operação de EPS's

As falhas na operação de um EPS pode ser classificada como:

- Operação desejada;
- Operação indesejada;
- Falha de Operação, quando esta é necessária.

Uma operação do EPS pode ser desejável ou indesejável, dependendo das consequências relativas a sua operação para a mesma situação. Se a consequência da operação for menos severa que a consequência para a não operação, a operação é desejável.

Se a consequência da operação do EPS for mais severa que a não operação, então ela é indesejável. Uma operação indesejável pode ser não intencional, devido a erros de hardware, software, erro humano, ou ela pode ser intencional (de acordo com o projeto), mas indesejável devido a falhas na lógica do projeto.

Quando o EPS toma uma ação desnecessária, sem que exista perturbação no sistema, vê-se um exemplo de operação indesejável e não intencional.

Um EPS pode falhar devido a diferentes razões, entre elas:

- Falha de Hardware;
- Falha no Software;
- Erro Humano;
- Falha na Lógica.

A falha de hardware ocorre quando algum problema físico envolve um ou mais componentes do sistema. A falha de software ocorre quando existe um erro no programa escrito pelo fornecedor. O erro humano pode estar associado a erros na implementação, operação ou manutenção. O erro de lógica ocorre como resultado de um estudo inadequado ou incompleto durante a fase de projeto.

Na tabela 4.2 a seguir, estão enumerados alguns casos em que ocorreram erros de lógica na operação de EPSs[3].

Data	Empresa	Esquema	Consequencia
04-abr-1988	WSCC	Separação das regiões Nordeste Sudeste	Perda de 1902 MW de geração e de 253 MW de carga
08-jan-1990	WSCC	Interligação entre Garrison e Taft em 500 kV	Perda de 119 MW de geração e de 25 MW de carga
06-nov-1997	MAPP & MAIN	Separação da interligação Leste MAPP - Oeste MAIN	Afundamento da tensão no sudoeste de Wisconsin, no leste de Iowa e oeste de Illinois

Tabela 4.2: Erros de lógica em EPSs.

Nesta tese, mostra-se que o uso de Model Checking pode ser útil no ambiente dos Sistemas Elétricos de Potência. Através de experimentos, utiliza-se a ferramenta

de verificação de modelos em alguns esquemas básicos de proteção. No primeiro exemplo, um esquema básico de proteção de circuitos, utiliza-se a comunicação entre dois relés que poderia ou não ser feita por meio de uma rede local. No segundo exemplo, um esquema de transferência automática de alimentação, acrescentam-se mais relés na comunicação e considera-se que esta passa a ser feita através de uma rede local.

4.6 Comentários

Neste capítulo foram apresentados conceitos de proteção de sistemas elétricos de potência, bem como as principais funções e características. Também construiu-se breve histórico, chegando-se aos Esquemas de Proteção de Sistemas e mostrando-se o significativo aumento de sua utilização nos últimos anos com os possíveis tipos de falhas em sua operação. Alguns casos reais foram citados, em que a falha foi causada por erros na lógica do tipo de esquema e, em cada caso, apresentadas as consequências provocadas no sistema elétrico.

Capítulo 5

Modelagem de Esquemas em PROMELA

Neste capítulo são apresentados dois exemplos de esquemas básicos de proteção. Para cada exemplo, é feita uma descrição do seu funcionamento, a respectiva da modelagem em PROMELA e, ao final, os resultados de verificações feita com o SPIN.

5.1 Esquema Básico de Proteção de Circuitos Elétricos

5.1.1 Descrição

Na figura 5.1, a corrente flui do gerador G_1 para a $Carga_1$ e para a $Carga_2$. Os relés R_1 e R_2 monitoram o sistema. Um curto-circuito ocorre, e há um aumento significativo na corrente na $Carga_2$.

Na figura 5.2, a corrente de falha é sentida pelos relés R_1 e R_2. De modo que a $Carga_1$, conectada ao disjuntor CB C, permaneça alimentada, o relé R_1 envia um comando de abertura para o disjuntor CB A, e envia um sinal de bloqueio para o relé R_2. Desta forma, somente o disjuntor CB A será aberto, e o fornecimento de energia para a $Carga_1$ continuará normalmente.

Na figura 5.3, tem-se que quando R_1 envia o sinal de abertura para o disjuntor CB A, este pode falhar, e não abrir. Então, neste caso, o relé R_1 sinaliza para que R_2 dê o comando de abertura no disjuntor CB B. Quando o relé R_2 recebe o sinal de abertura, envia o comando para abrir o disjuntor CB B. Observa-se que somente na falha de abertura do disjuntor CB A, que o disjuntor CB B será aberto.

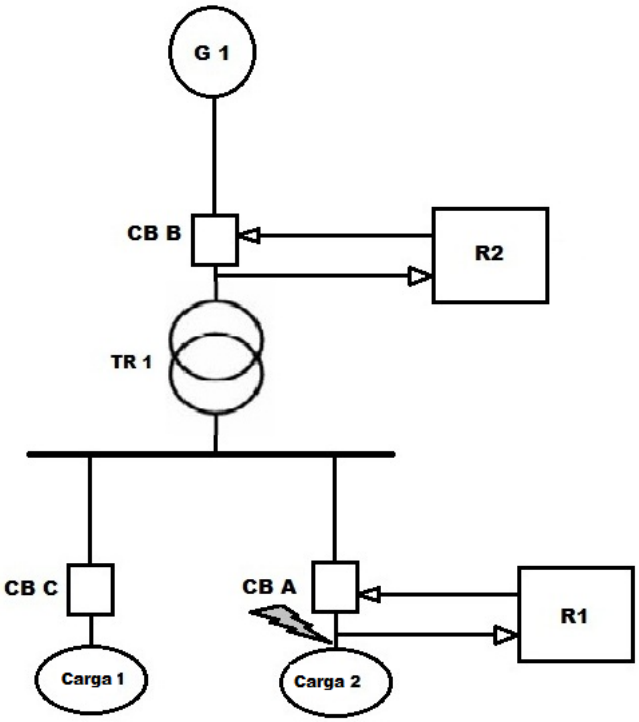

Figura 5.1: Curto-circuito na $Carga2$.

5.1.2 Modelagem

Na modelagem do esquema apresentado na seção anterior, foram modelados quatro processos, ou utilizando a terminologia PROMELA, quatro *proctypes*, cada um representando a modelagem de cada equipamento CbA, CbB, R1 e R2. Estes processos comunicam-se entre si através de cinco *message channels*, R2toCbB, CbBtoR2, R1toR2, R1toCbA e CbAtoR1. As mensagens que transitam em cada um dos canais são *trip* (camando de abertura), *bloq* (comando de bloqueio), *ack* (reconhecimento), *nak* (não reconhecimento), *reset* (reinicialização).

Cada processo foi implementado de modo a descrever exatamente o comportamento de cada equipamento durante a ocorrência da falha descrita.

No modelo implementado, a falha desejada é provocada durante a execução da função *init()*, cujo código está mostrado na figura 5.5 imediatamente após o lançamento de todos os processos. Na função *init()*, a primeira função executada em um programa na linguagem PROMELA, a falha é caracterizada através da atribuição de valores para as correntes que serão avaliadas pelos processos. Nesta tese serão mostrados somente trechos de códigos considerados significativos para o modelo.

Os valores limites estão definidos nas declarações no início do programa, conforme pode ser visto na figura 5.6.

No código do processo que modela o relé $R1$, apresentado na figura 5.7, observa-se que este ao perceber que o valor da corrente ultrapassou o valor limite, envia uma

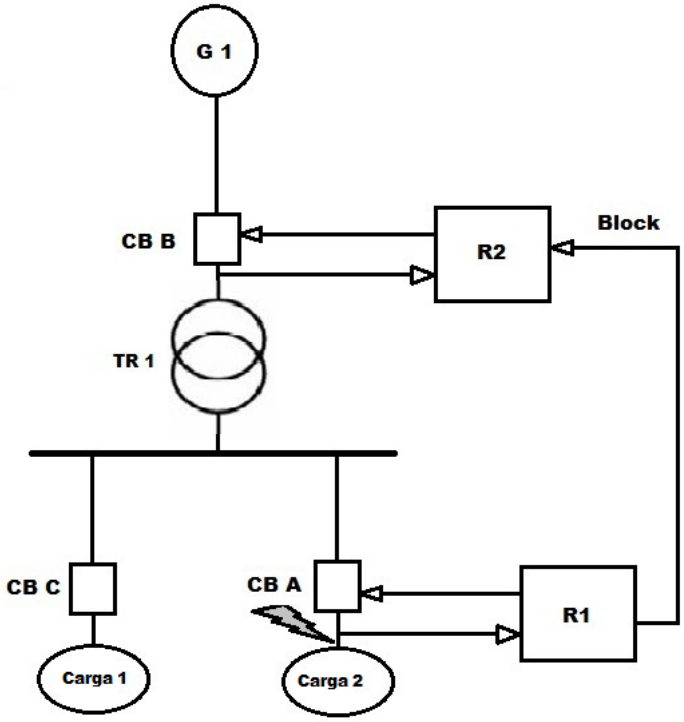

Figura 5.2: R1 percebe o curto-circuito e envia sinal de bloqueio para R2.

mensagem *trip* para que o disjuntor CbA abra, e uma mensagem de *bloq* para o relé $R2$ impedindo que este emita o comando *trip* para o disjuntor CbB. Em seguida, aguarda pela resposta com o resultado da operação do disjuntor CbA. O processo aguarda até que uma das condições, que neste caso são mutuamente exclusivas, seja satisfeita. Se houver uma resposta positiva do disjuntor CbA, um *ack*, o estado do disjuntor é alterado, um *reset* é enviado para $R2$, e a verificação da propriedade que representa a condição final é realizada através do comando *assert*. Caso a resposta do disjuntor CbA seja negativa, *nak*, indicando que este não conseguiu abrir, um comando de *trip* é enviado ao relé $R2$ para que este abra o disjuntor CbB.

Na modelagem do disjuntor CbA, apresentado na figura 5.8, mais uma vez, utilizou-se a característica não determinística da linguagem PROMELA através do uso da estrutura *if*. Nessa estrutura, as duas condições a serem verificadas são idênticas, ou seja, o recebimento da mensagem de *trip* vinda do relé $R1$. Entretanto, devido ao não determinismo da linguagem, somente uma delas, de modo aleatório, será executada em cada execução. No modelo, esta estrutura foi utilizada de modo que o disjuntor CbA, ao receber um comando de *trip* pode, ou não, abrir. Caso abra, o estado do disjuntor é alterado, com a consequente interrupção da passagem de corrente e o envio da mensagem de *ack* para o relé $R1$. Toda essa sequencia é feita utilizando o modo sem interrupção, dentro do bloco *atomic*. Existe a possibilidade de que a estrutura *if* seja executada onde está representada uma possível falha no disjuntor CbA, quando então é enviado para $R1$ a mensagem, *nak*.

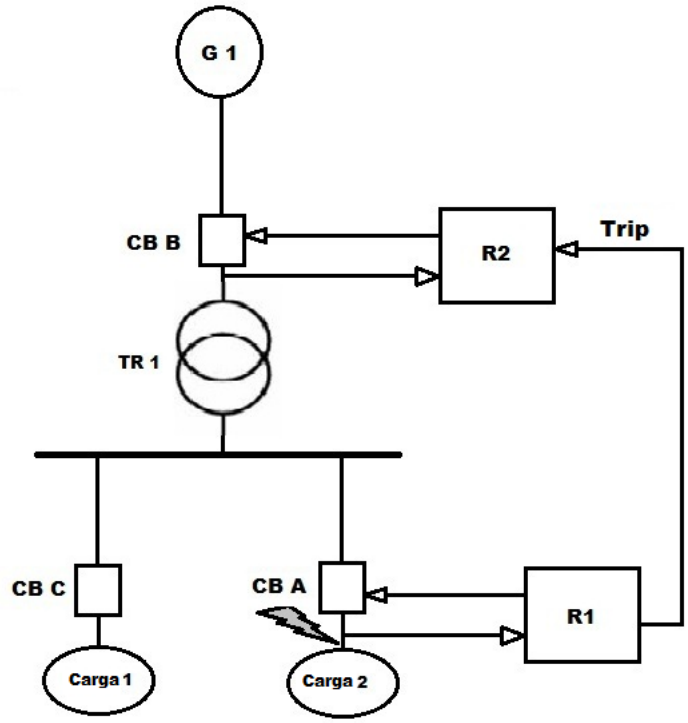

Figura 5.3: CB A não abre, R1 envia comando de abertura para R2.

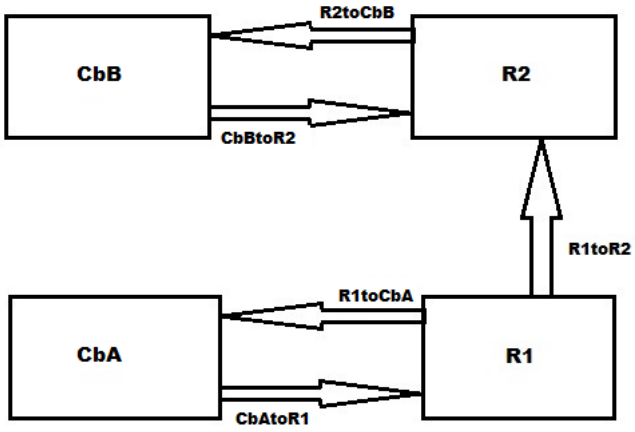

Figura 5.4: Modelo do Esquema Básico Proteção de Circuito.

Da mesma forma que o $R1$, o início do processo $R2$, apresentado na figura 5.9 é a observação do valor da corrente que passa através do disjuntor CbB. Quando o valor desta corrente ultrapassar um determinado limite, o processo $R2$ entra em uma estrutura *if* aguardando duas condições. A primeira é o recebimento de um comando de bloqueio, *bloq*, vindo do relé $R1$. A segunda é a situação em que não recebe mensagem alguma e, consequentemente ocorre um *timeout*. Na condição de recebimento de um bloqueio, o processo aguarda, ou pelo recebimento de um

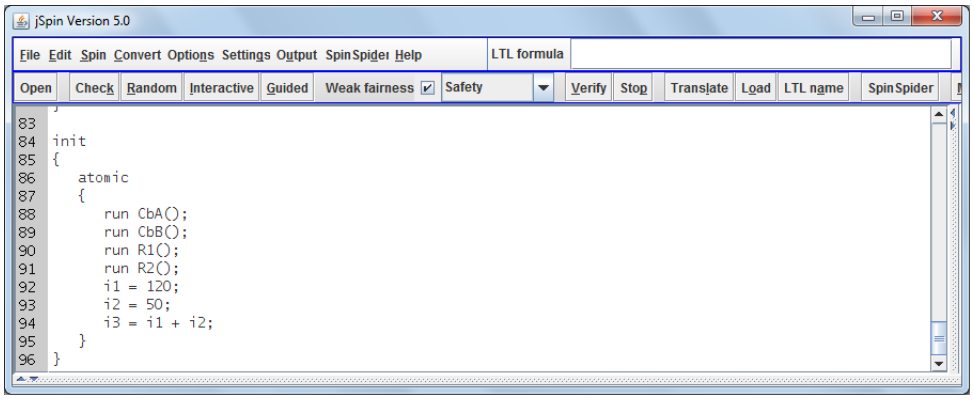

Figura 5.5: Processo init.

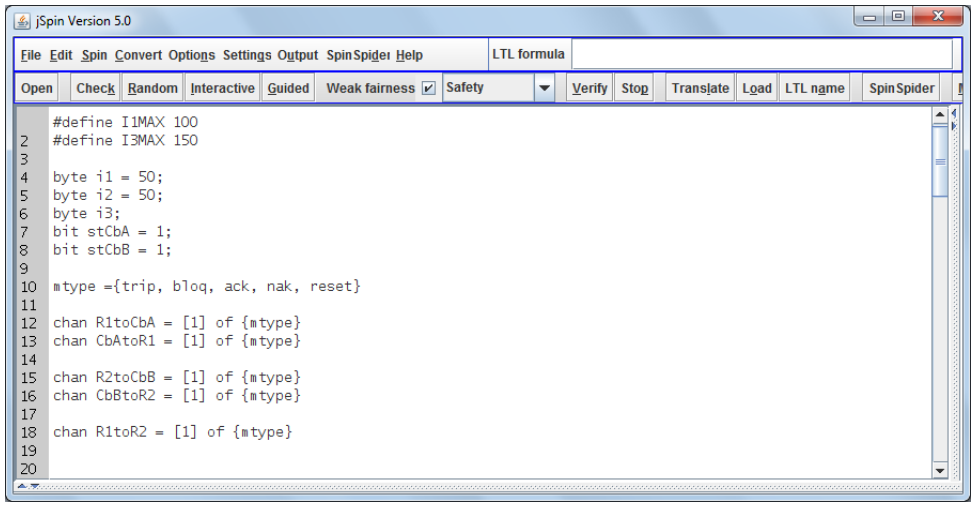

Figura 5.6: Declarações.

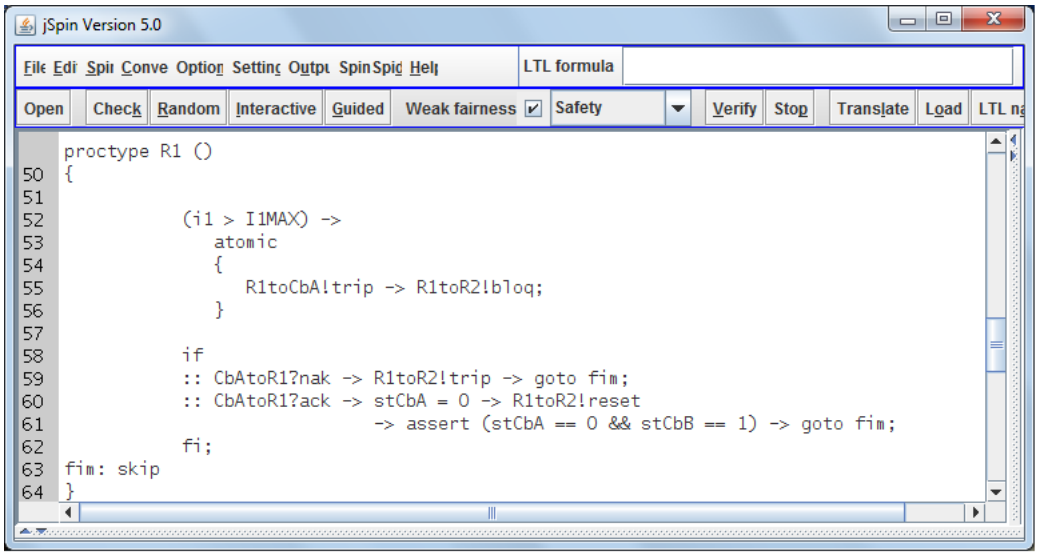

Figura 5.7: Processo R1.

comando *trip*, que nada mais é do que a autorização para a abertura do disjuntor CbB, ou pelo comando de *reset* que é a informação de que o relé $R1$ realizou com

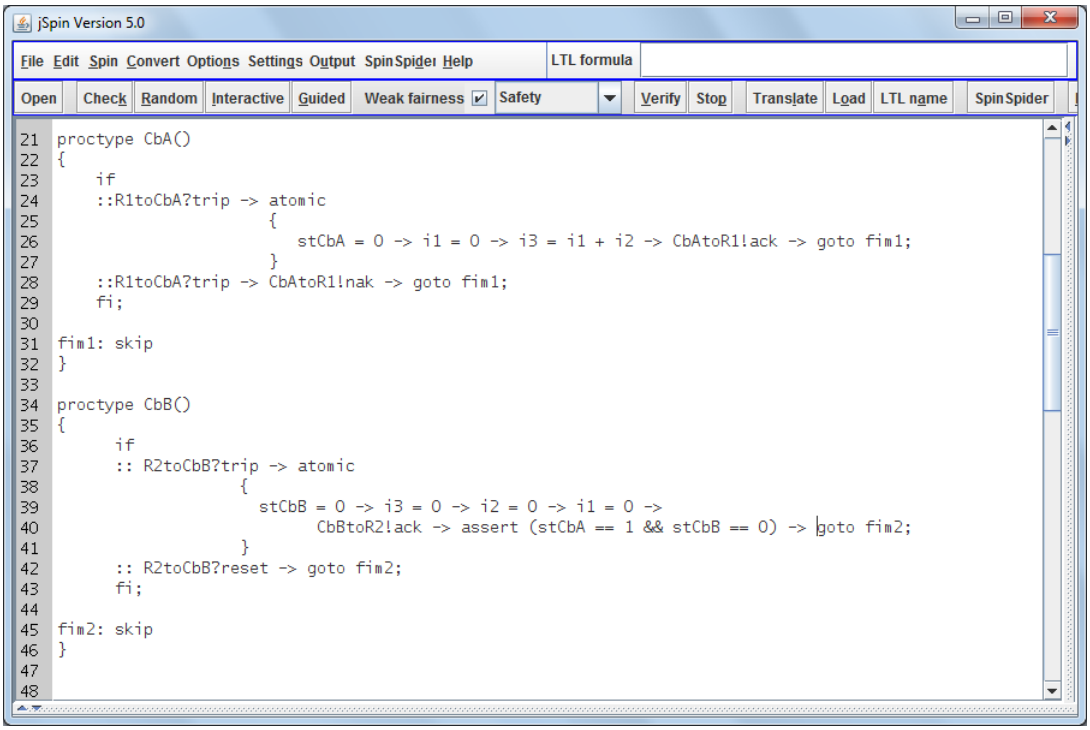

Figura 5.8: Processos CbA e CbB.

sucesso sua função. Na situação de *timeout*, o processo assume que a função do relé $R1$ não foi realizada e então o relé $R2$ realiza a sua através do envio do comando *trip* para o disjuntor CbB. Neste processo, a verificação de propriedade é feita todas as vezes que um comando para a abertura do disjuntor é enviado. E neste caso, como o comando somente será enviado devido a um problema ocorrido durante o fechamento de CbA, a propriedade a ser verificada é CbA fechado e CbB aberto.

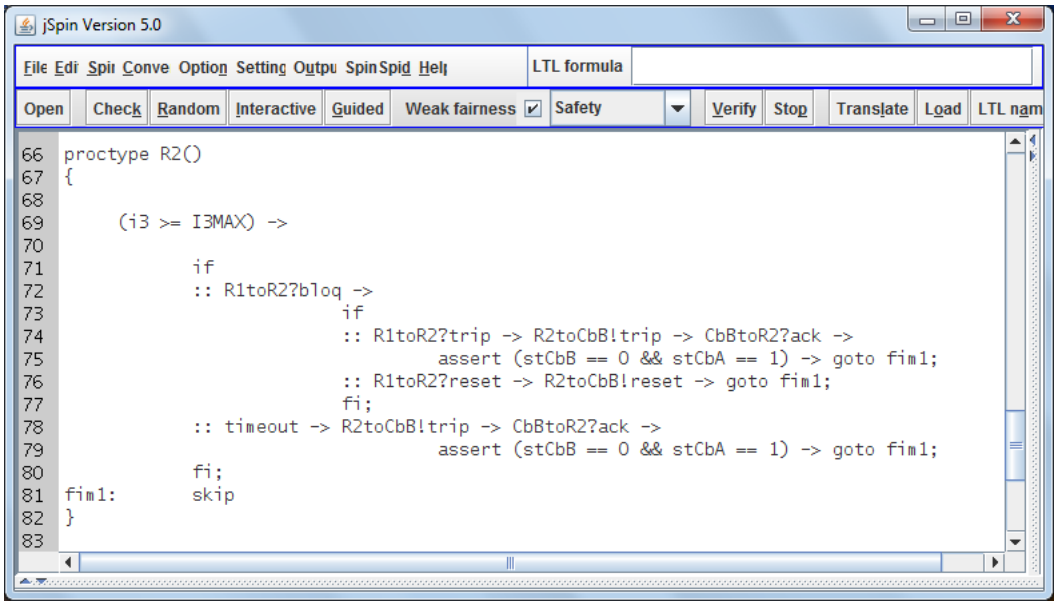

Figura 5.9: Processos R2.

5.1.3 Verificação

A verificação realizada com o modelo sem erros produziu a saída mostrada na figura 5.10. Como nesta verificação o modelo não continha erros, a saída produzida pelo SPIN indicou quantidade de erros igual a zero.

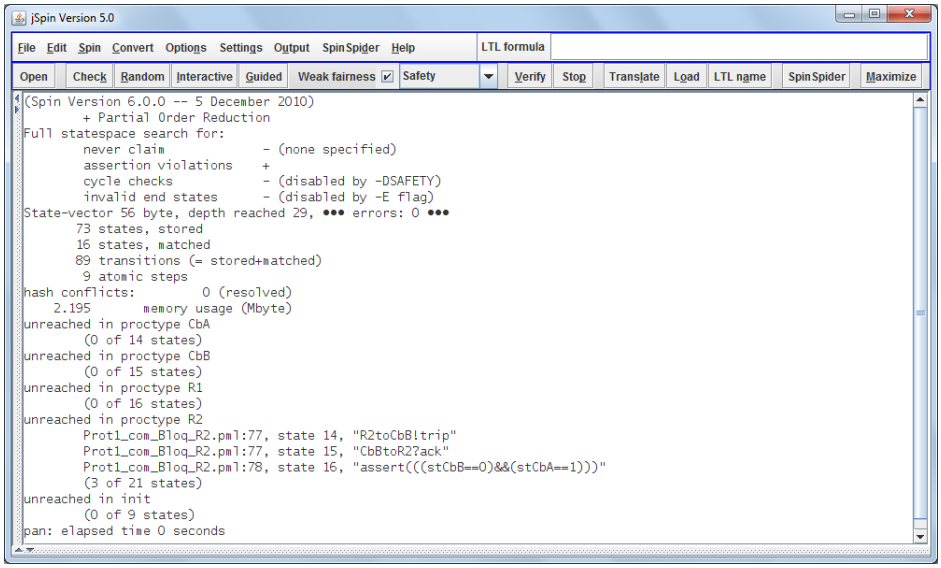

Figura 5.10: Verificação do Esquema Básico sem Erros.

De modo a observar o comportamento do verificador SPIN na ocorrência de um erro, foi realizada uma alteração na modelagem do processo R1. Nessa alteração, foi retirado o envio do comando para o bloqueio de R1 para R2. Com essa alteração, o processo R1 envia o comando de trip para o processo CB A, mas não envia o comando de bloqueio para R2. Ao observar, na figura 5.11 o resultado apresentado pela verificação, uma violação de propriedade é identificada, uma vez que haverá o desligamento indesejável dos dois disjuntores CB A e CB B.

Para a identificação da condição que levou ao erro, se faz necessário realizar a chamada simulação guiada. Este tipo de simulação utiliza o arquivo com extensão *trail*. Este arquivo somente é gerado, quando, durante uma verificação, um erro é identificado no modelo. Ele permite o acompanhamento da sequência de execução até o ponto onde o erro foi encontrado.

A figura 5.12 mostra a saída da simulação guiada. Nela pode ser observado, através das variáveis stCbA e stCbB, que os dois disjuntores estão abertos, e ainda a linha do código e o processo onde ocorreu o erro, que neste exemplo foi na linha 40 no processo CbB. Também pode ser observado por esta simulação, a ocorrência do timeout que provocou a abertura do disjuntor CB B.

Outra forma de se verificar o comportamento do sistema é estabelecer uma propriedade que o sistema deve atender, representá-la em uma expressão utilizando operadores da lógica temporal, e submetê-la ao verificador. Neste caso, foi realizada

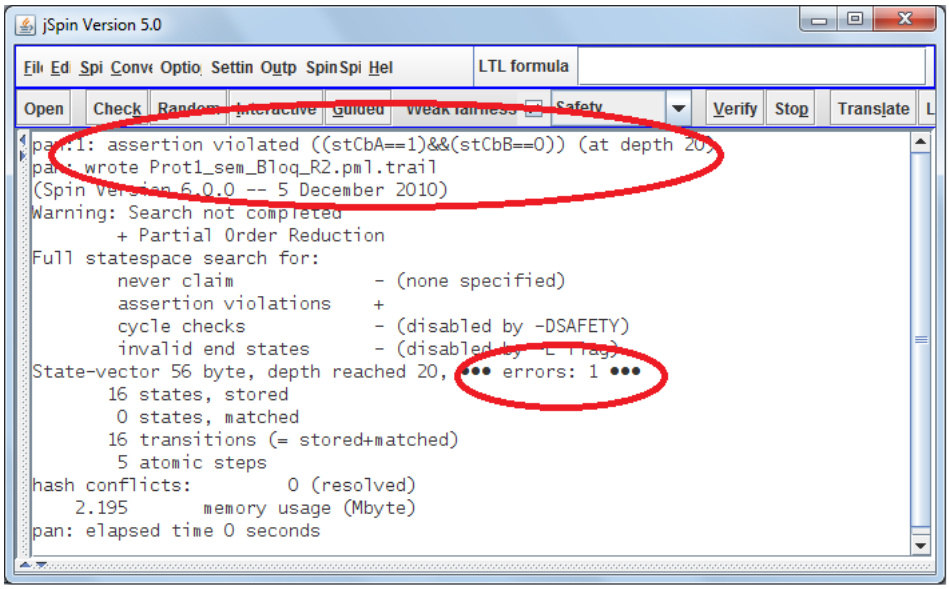

Figura 5.11: Verificação do Esquema Básico com Erros.

a verificação da sentença:

$$\Box\ (\text{stCbB} == 1)$$

ou seja, se em todos os momentos no futuro, o disjuntor Cb B permenecerá fechado. O resultado desta verificação pode ser visualizada na figura 5.13 e a simulação guiada na figura 5.14.

Pela simulação, observa-se que a falha na abertura do disjuntor CbA, causa a abertura do disjuntor CbB, fazendo com que a sentença seja falsa. Ou seja, existe pelo menos um caso em que o disjuntor CbB abre, o que é o resultado esperado para o exemplo.

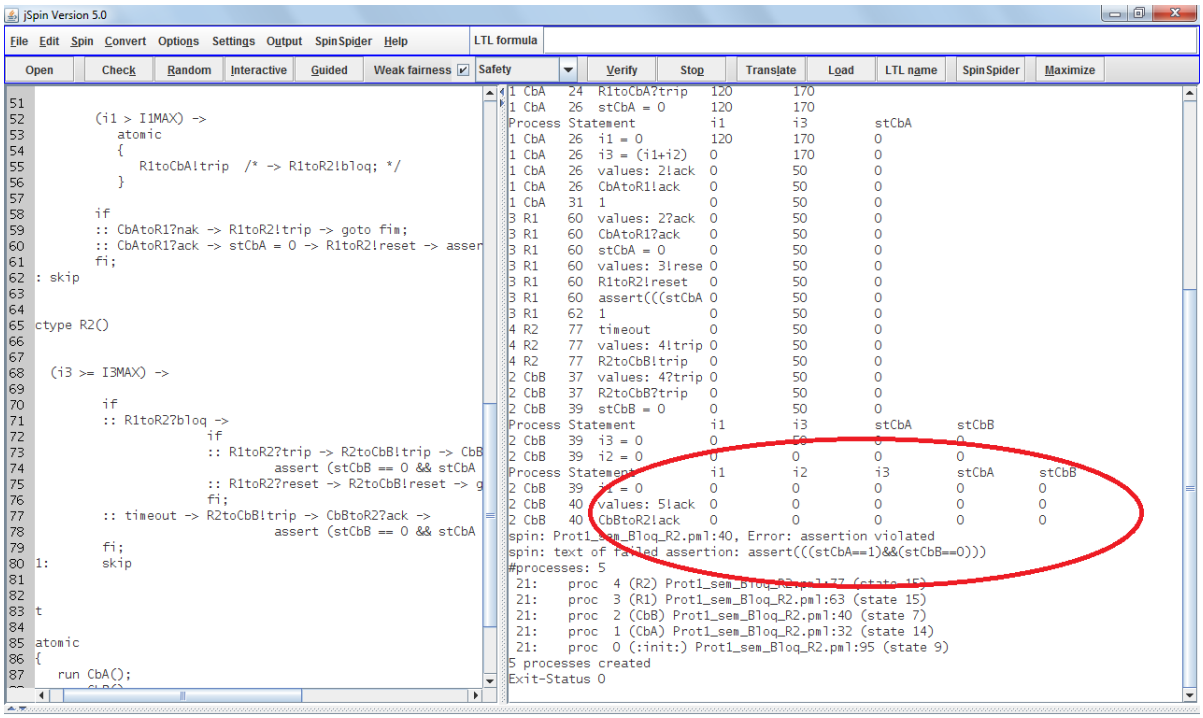

Figura 5.12: Simulação Guiada do Esquema Básico com Erros.

5.2 Transferência Automática de Alimentação

5.2.1 Descrição

Para que sejam atendidos os indicadores de qualidade para o fornecimento de energia, tipicamente as subestações de energia elétrica são alimentadas por duas linhas de transmissão, uma linha sendo considerada como principal e outra considerada como reserva.

Em caso de falha no fornecimento na linha principal, automaticamente haverá a transferência, de modo que o fornecimento de energia passe a ser realizado pela linha considerada reserva. Na figura 5.15 está representado o diagrama de uma subestação típica alimentada pelas linhas 1 e 2. Normalmente, o disjuntor DJ_1 está fechado, e o disjuntor DJ_2 está aberto, indicando que o fornecimento de energia à subestação está sendo feito pela Linha 1. Neste exemplo está presente uma característica que tem se tornado muito comum nos últimos tempos, que é um gerador ligado na barra da subestação. Como se sabe, hoje as empresas de energia, em alguns casos, possuem clientes com geração própria. Este tipo de geração é realizada por algum agente externo à empresa, capaz de produzir sua própria energia, e que comercializa o seu excedente. No exemplo, considera-se que a geração está sendo fornecida e que, portanto, o disjuntor DJ_g está fechado fazendo com que um bloco de energia esteja sendo injetado na barra $B1$ pelo gerador G_1.

Caso ocorra uma falha na Linha 1, um curto-circuito por exemplo, o relé $R1$ ao

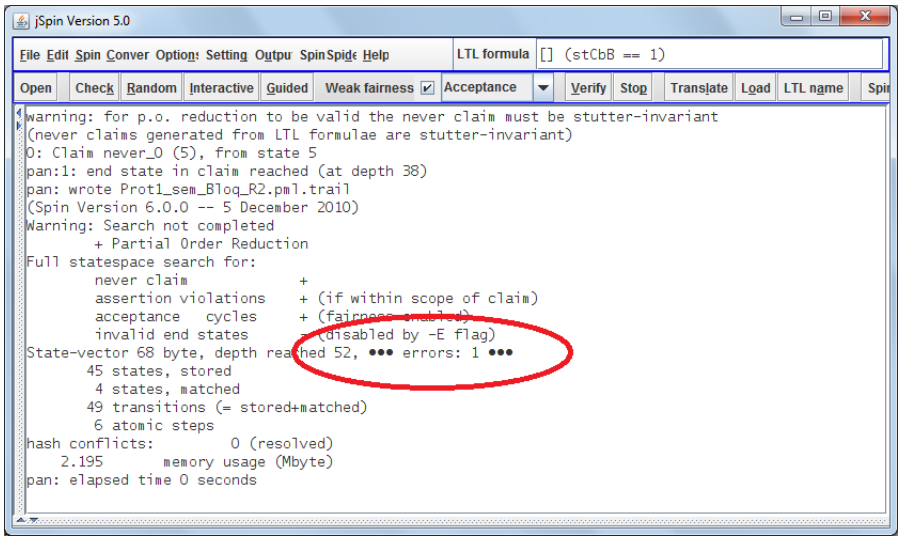

Figura 5.13: Verificação de fórmula temporal do Esquema Básico.

sentir a queda de tensão na linha, imediatamente, deverá enviar o comando para a abertura do disjuntor DJ_1, e antes de providenciar o fechamento de DJ_2, para a transferência de alimentação, deve solicitar que o disjuntor DJ_g seja aberto. Somente quando a geração não estiver ligada à barra $B1$, o comando de fechamento para o disjuntor DJ_2 deverá ser enviado. Uma vez que esta sequencia esteja terminada, a subestação voltará a estar alimentada pela linha 2 considerada reserva, e o gerador G_1 poderá ser novamente ligado ao sistema.

Neste exemplo, diferentemente do anterior, foi considerado que os relés se comunicam através de uma rede local. Portanto, o modelo será implementado através de troca de mensagens entre os relés passando por um único canal de comunicação, a rede local.

5.2.2 Modelagem

Na modelagem deste segundo exemplo foi utilizado outro tipo de arquitetura. Neste modelo, apresentado na figura 5.16, os processos se comunicam através de um único meio que é a rede local. Desta forma, foi definida uma estrutura básica para as mensagens. Cada mensagem é composta por três bytes, o primeiro com o endereço do destino, o segundo com o endereço da origem, e o terceiro com a função. O byte com a função pode assumir os seguintes valores: *trip* (comando de abertura), *close* (comando de fechamento), *gout* (comando para retirada do gerador), *gin* (comando para conectar o gerador), *busok* (informação de que o barramento está ok), *busnok* (informação de que o barramento não está ok).

Da mesma forma que no modelo anterior, a falha que se deseja estudar também será provocada durante a execução da função *init()*, antes do lançamento de todos os outros processos. Neste modelo, pela característica do tipo de comportamento

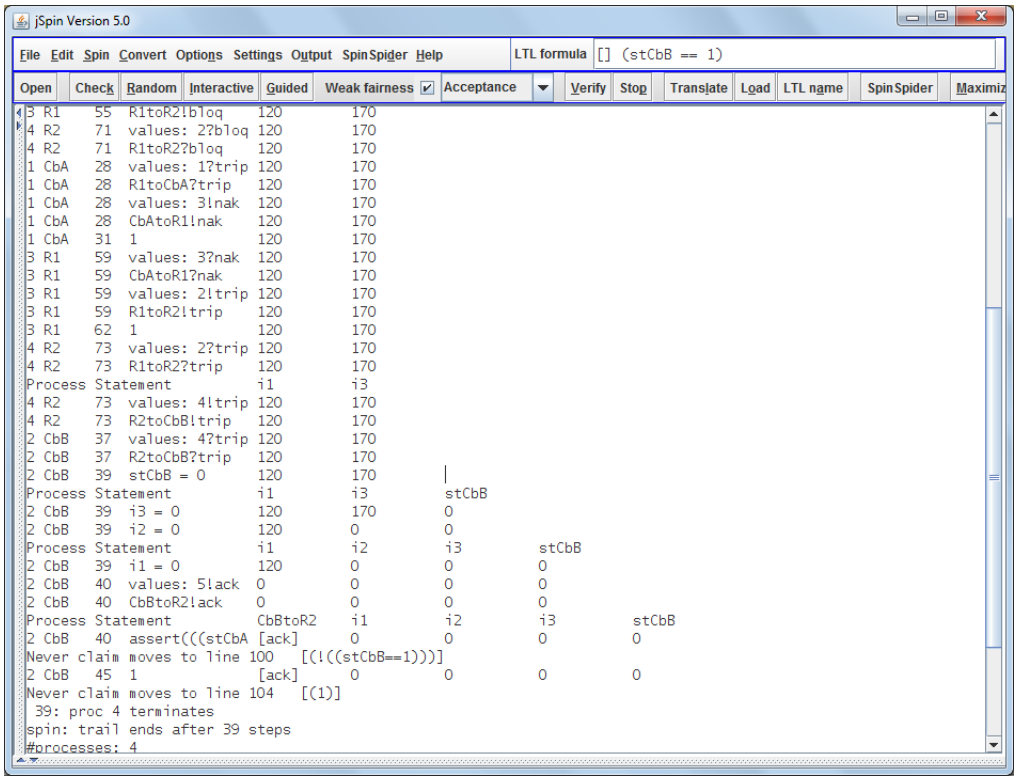

Figura 5.14: Simulação Guiada da verificação do fórmula temporal do Esquema Básico.

que se deseja verificar, serão definidos estados para as linhas e valor de tensão para a barra. Nos próximos parágrafos estarão descritos cada um dos processos, com apresentação dos trechos de código mais significativos.

Inicialmente, o processo *init()*, conforme descrito no parágrafo anterior, foram definidos os estados dos disjuntores. O disjuntor $DJ_1 = 1$, fechado, o disjuntor $DJ_2 = 0$, aberto, e o disjuntor $DJ_g = 1$ fechado. Além disso, foram definidos valores para as tensões nas linhas e na barra. No exemplo, o valor da tensão da $Linha1$ foi abaixo de 70% do valor nominal e na barra, o valor nominal. Somente depois destas definições, os outros processos são lançados. O código do processo *init()* pode ser observado na figura 5.17.

Na figura 5.18 encontra-se o código do processo $Rl1$, no início de sua execução aguarda por duas ocorrências: ou pelo problema na $Linha1$, ou por uma solicitação de atuação vinda da rede de comunicação. Quando o processo percebe que a linha está com problema, um comando de abertura é enviado para o disjuntor DJ_1. Estando o disjuntor DJ_1 aberto, o processo $R1$ envia uma mensagem interrogando o relé Rb, que observa a barra, se pode ser enviada a solicitação para que o relé $R2$ realize o comando de fechamento do disjuntor DJ_2. Como o gerador G_1 está ligado na barra $B1$, o disjuntor DJ_2 não pode ser fechado, quando G_1 estiver conectado. Estando o gerador G_1 desconectado da barra $B1$, então, o processo $Rl1$ solicita ao processo $Rl2$ para comandar o fechamento do disjuntor DJ_2.

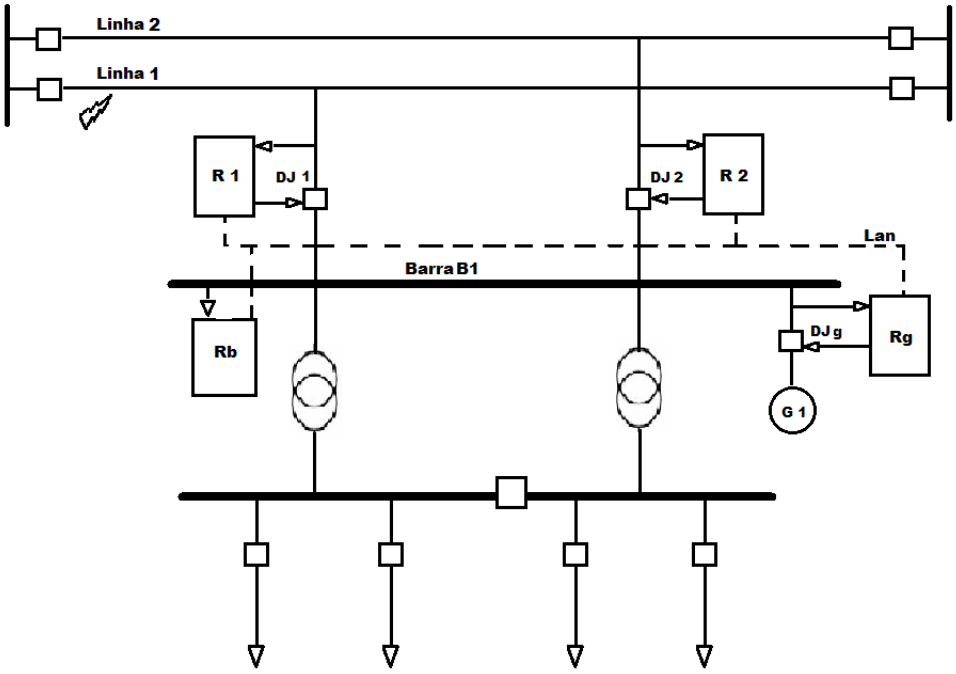

Figura 5.15: Curto-circuito na Linha 1.

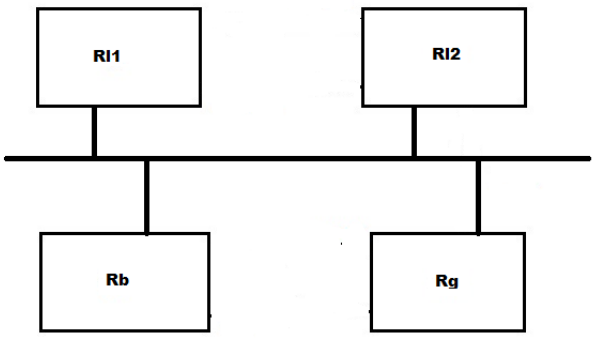

Figura 5.16: Modelo da Transferência Automática da Alimentação.

A outra ocorrência que este processo aguarda é o recebimento de mensagem com o comando para o fechamento do disjuntor DJ_1, vinda do processo $Rl2$. Esta ocorrência acontece quando a alimentação da subestação está sendo feito pela $Linha2$, e o relé $Rl2$ detecta uma queda na tensão. É a situação inversa do que foi descrito para a $Linha1$. Neste caso, o processo $Rl1$ verifica se a tensão na linha é adequada, e então envia o comando para fechar o disjuntor DJ_1. Neste processo são incluídas algumas verificações de propriedades, todas elas referentes aos estados que os disjuntores devem assumir naquela situação. Como exemplo, no caso em que a barra $B1$ não esteja pronta para a transferência, os disjuntores DJ_1 e DJ_2 devem estar abertos. Deve ser observado que o processo $Rl2$ segue característica análoga,

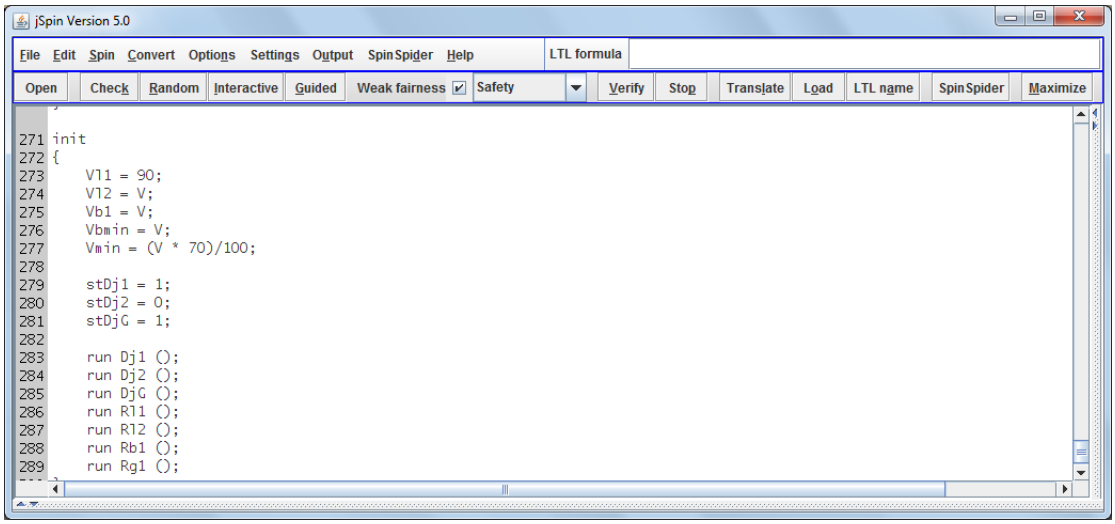

Figura 5.17: Processo init.

e, portanto não terá seu código apresentado.

O processo $Rb1$, cujo código encontra-se na figura 5.19 que observa o estado da barra $B1$, tem como principal característica a espera pela mensagem de interrogação se a barra está pronta para a transferência. Ao receber a mensagem, o processo verifica a tensão, se o valor da tensão indicar que o gerador está conectado, envia o comando solicitando a desconexão do gerador G_1. Ao chegar a resposta de que o gerador foi desconectado, o processo envia a resposta de que a barra está pronta para a transferência de alimentação.

O processo $Rg1$, apresentado na figura 5.20 aguarda por mensagem vinda do relé que observa a barra com o comando para conectar ou desconectar o gerador. Os processos que modelam os disjuntores são idênticos, eles recebem o comando dos respectivos relés e, de forma não determinística, realizam ou não o comando solicitado.

5.2.3 Verificação

Para a verificação deste modelo, o estado inicial era a alimentação da subestação pela $Linha1$, com o gerador G_1 conectado à barra $B1$. Ou seja, DJ_1 fechado, DJ_2 aberto e DJ_g fechado.

Uma vez que para um modelo sem erros, o SPIN não apresenta saída com informações relevantes, optou-se por apresentar as saídas de verificações que apresentaram erros. Desta maneira, com o objetivo de se provocar um erro, no código do processo $Rl1$, ao ser detectada a queda de tensão na linha, deixou-se de observar a conexão do gerador na barra $B1$ do modelo, fazendo com que este enviasse imediatamente o comando para abrir o DJ_2.

Como pode ser observado na figura 5.21, a verificação apontou uma violação de

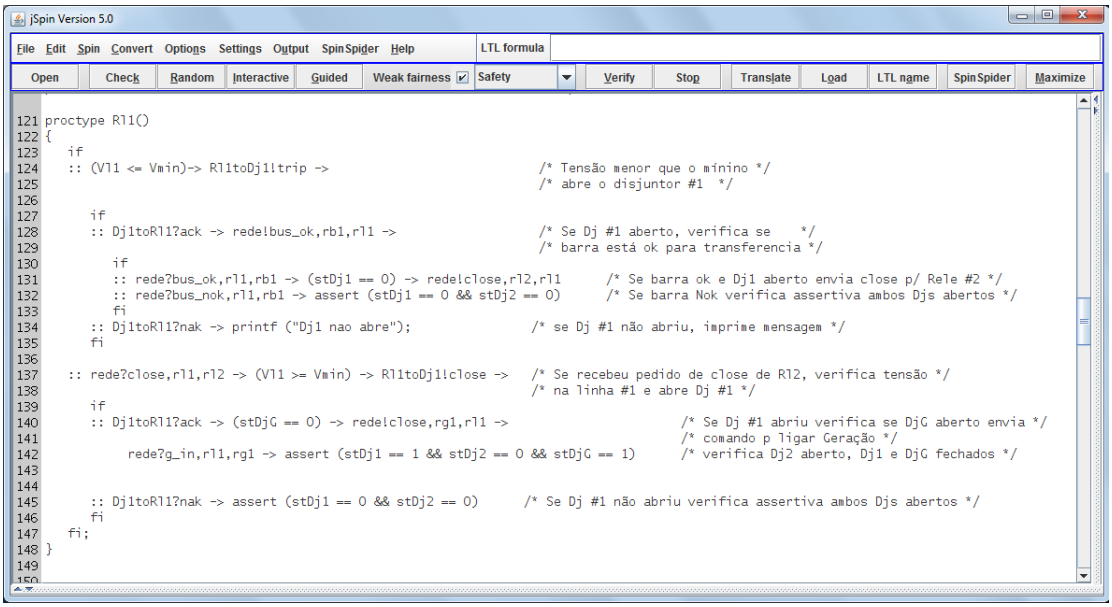

Figura 5.18: Processo Rl1.

propriedade, com a consequente geração de um arquivo *.trail*. Através da simulação guiada, mostrada na figura 5.22, é possível observar que o estado do DJ_g permaneceu fechado, ou seja, o gerador estava conectado à barra durante a transferência. O que diante do erro provocado, era o resultado esperado.

Outro experimento realizado foi a avaliação da seguinte expressão da lógica temporal.

$$(\text{stDj1} = 0) \rightarrow \Diamond\, (\text{stDj2} = 1)$$

O significado desta sentença é: uma vez que o disjuntor 1 abra, em algum momento no futuro, o disjuntor 2 vai fechar. Ou seja, sempre haverá a transferência de alimentação. Entretanto, como no modelo existe a previsão de que pode acontecer uma falha na abertura de qualquer um dos disjuntores, a propriedade expressa pela sentença em lógica temporal não será atendida pelo modelo. A simulação guiada da figura 5.23 mostra que o verificador invalidou a sentença, identificando a condição que provoca o erro.

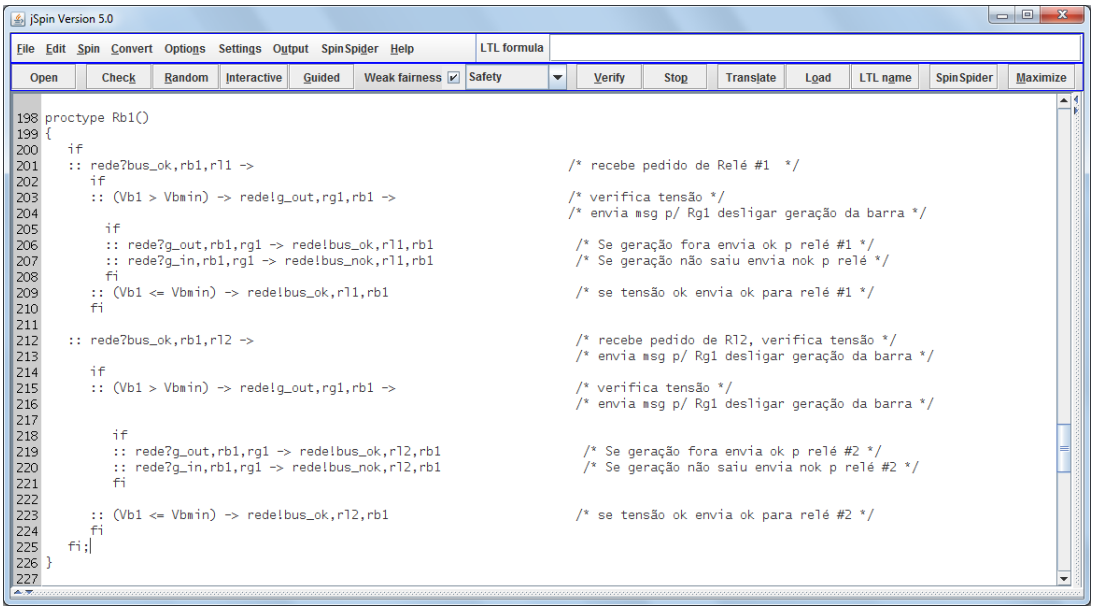

Figura 5.19: Processo Rb1.

5.3 Comentários

Neste capítulo foram apresentados exemplos teóricos e realizadas modelagens e verificações destes exemplos. De modo a observar o comportamento do verificador, erros foram inseridos intencionalmente em cada modelo. Foi possível constatar que todos eles foram identificados pelo verificador SPIN. No capítulo seguinte será realizada a modelagem e verificação de um caso real.

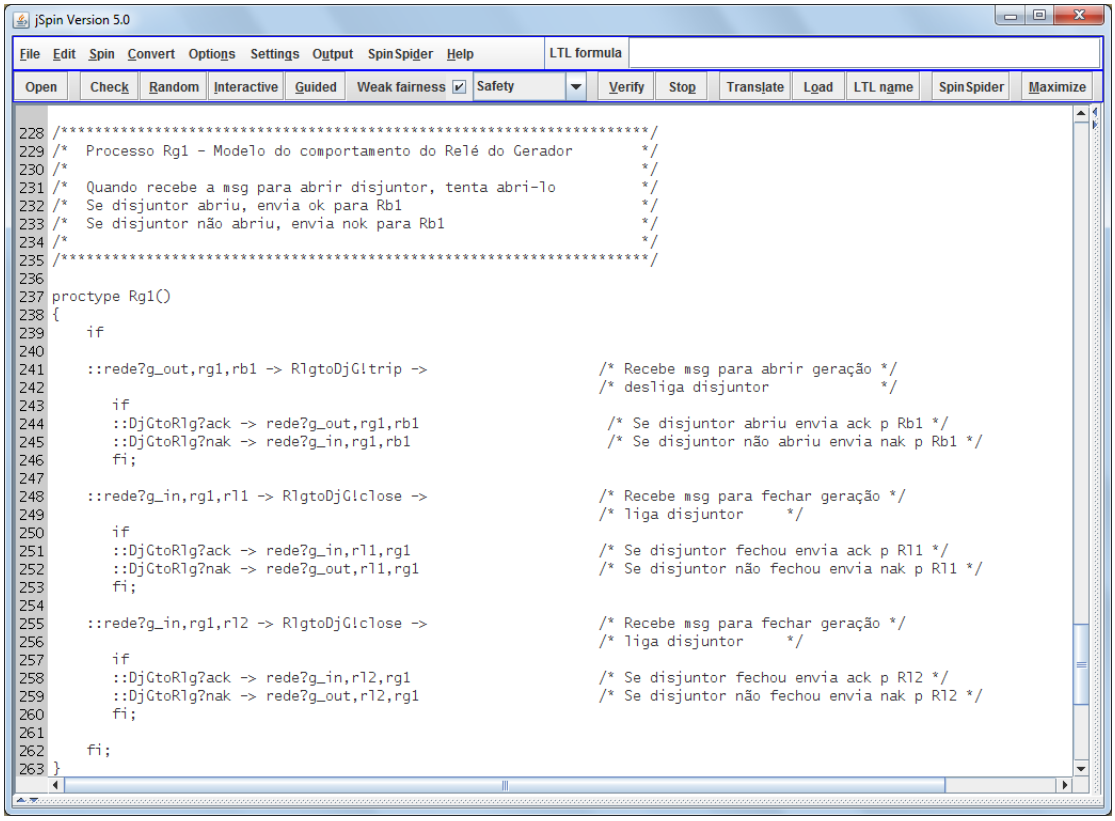

Figura 5.20: Processo Rg1.

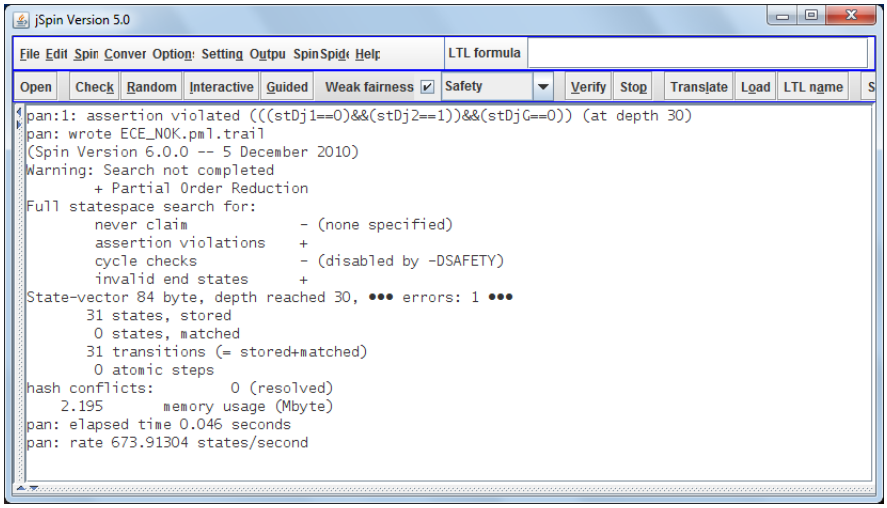

Figura 5.21: Resultado da Verificação de Transferência Automática.

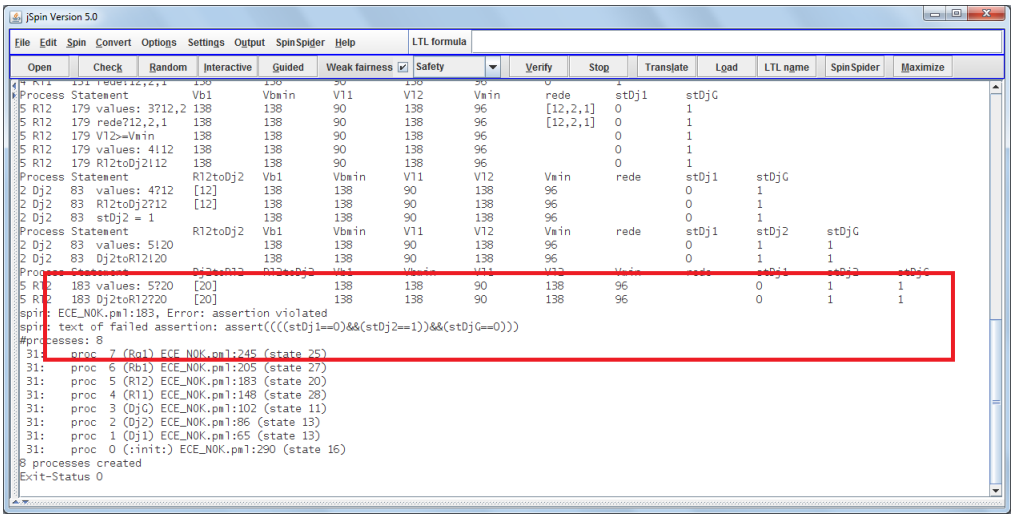

Figura 5.22: Simulação Guiada de Transferência Automática.

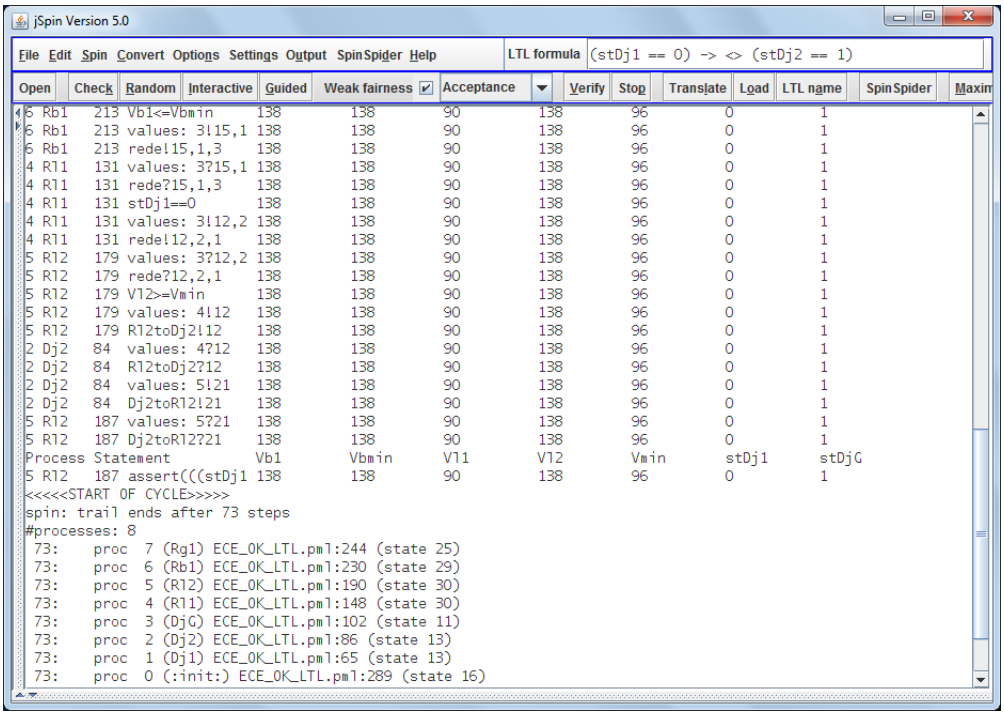

Figura 5.23: Simulação Guiada do resultado da Lógica Temporal.

Capítulo 6

Caso Real: Esquema de Controle de Segurança para corte seletivo de carga

Neste capítulo será apresentado um exemplo da utilização da Verificação Formal em um caso real do Sistema Elétrico Brasileiro, ocorrido na região do Distrito Federal. No exemplo que será mostrado, um erro na lógica de um Esquema de Controle de Segurança - ECS provocou um desligamento indevido da linha de transmissão Brasília Sul - Santa Maria, na subestação - SE Brasília Sul das Centrais Elétricas de Brasília - CEB.

Este esquema é utilizado pelo Operador Nacional do Sistema Elétrico - ONS, e tem como objetivo preservar parte da carga considerada prioritária do Distrito Federal, quando da ocorrência da perda da Linha de Transmissão de circuito duplo Brasília Sul - Samambaia de 345 kV, associada à perda de uma das seções do barramento de 345 kV da SE Brasília Sul [1].

A figura 6.1 apresenta o diagrama com o sistema da área de influência do ECS de Brasília

6.1 Descrição do funcionamento do Esquema

O esquema é implementado através de uma rede de Controladores Lógicos Programáveis - CLPs remotos, situados nas instalações de FURNAS Centrais Elétricas e da CEB. Esta rede é conectada a um outro CLP concentrador (Master) localizado na subestação Brasília Sul de FURNAS.

A monitoração dos estados, da configuração do sistema e o comando para o corte de carga são feitos pelos CLP's remotos e o processamento da lógica para que sejam decididas as ações nos equipamentos da rede elétrica é feito pelo CLP Master

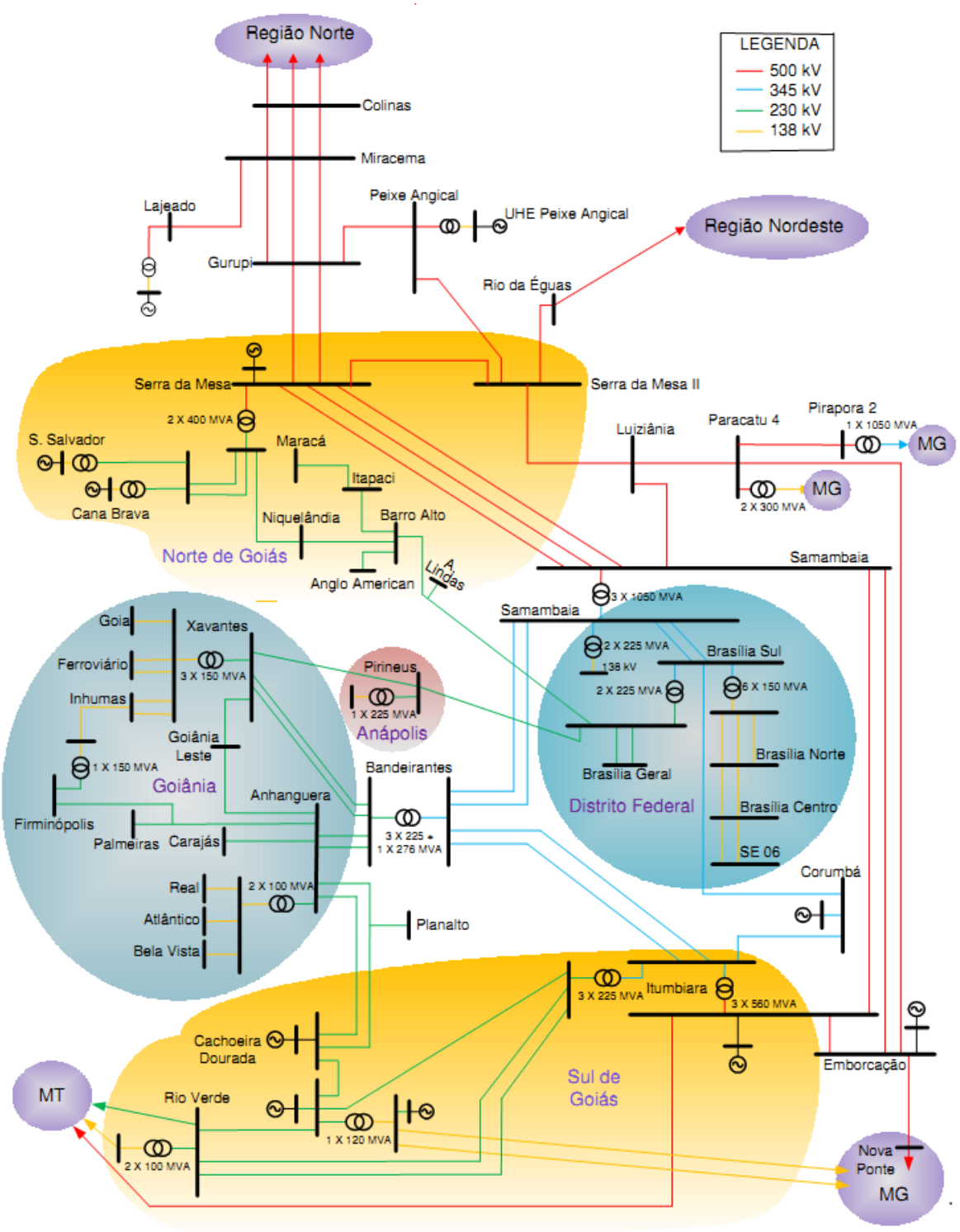

Figura 6.1: Sistema da área de influência do ECS de Brasília[1].

da SE Brasília Sul. Ao término do processamento da lógica, o CLP Master envia suas decisões de volta aos CLPs remotos, para que estes realizem os respectivos comandos.

O esquema visa a eliminação da possibilidade de perda total da subestação Brasília Sul devido a sobrecarga nos transformadores 345/138 kV remanescentes,

e a consequente interrupção total no fornecimento de energia ao Distrito Federal.

Desta forma, o esquema realiza:

- A abertura dos circuitos de saída de 138 kV da SE Brasília Sul, e
- a inserção de bancos de capacitores nas subestações Brasília Sul, Brasília Centro, SE 06 e um corte seletivo de cargas nas subestações da CEB: Brasília Norte, Brasília Centro e subestação 06.

Para que a parte da carga considerada como prioritária do Distrito Federal seja preservada, as lógicas do CLP Master, localizado na SE Brasília Sul, são identificadas como Lógica de Perda Dupla e Lógica de Perda de Barra. Elas detectam as perdas dos circuitos e de seções do barramento, monitoram as condições de tensão e enviam os comandos para que os CLP's remotos realizem os cortes de carga.

Nas seções seguintes serão descritas as duas lógicas do esquema: a Lógica de Perda Dupla e a Lógica de Perda de Barra.

6.1.1 Lógica de Perda Dupla

Esta lógica monitora os dois circuitos Samambaia - Brasília Sul de 345 kV. Na ocorrência da perda de ambos, e sem que os relés de proteção da barra de 345 kV na subestação Brasília Sul tenham atuado, o esquema executa os seguintes comandos:

- Se a carga da CEB estiver nas condições Média ou Pesada
  - Fecha os disjuntores e chaves dos bancos de capacitores das subestações Brasília Sul, 06 e Brasília Centro;
  - Abre a linha de transmissão Brasília Sul - Ceilândia C1 e C2;
  - Abre a linha de transmissão Brasília Sul - Taguatinga C1 e C2;
  - Abre a linha de transmissão Brasília Sul - Santa Maria, com retardo de 2 segundos, se a tensão na barra da subestação Brasília Sul 345 kV estiver inferior a 90%.
- Se a carga da CEB estiver na condição Leve
  - Abre a linha de tranmissão Brasília Sul - Taguatinga C1 e C2;
  - Abre a linha de tranmissão Brasília Sul - Ceilândia C1 e C2, com retardo de 2 segundos, se a tensão na barra da subestação Brasília Sul 345 kV estiver inferior a 90%.

Na figura 6.2 é apresentado o diagrama com a lógica de Perda Dupla sem a atuação da proteção da barra de 345 kV da subestação Brasília Sul. Deve ser observado que, nos diagramas, procura-se delimitar as lógicas, a monitoração e os comandos dentro do CLP onde cada uma dessas ações são realizadas.

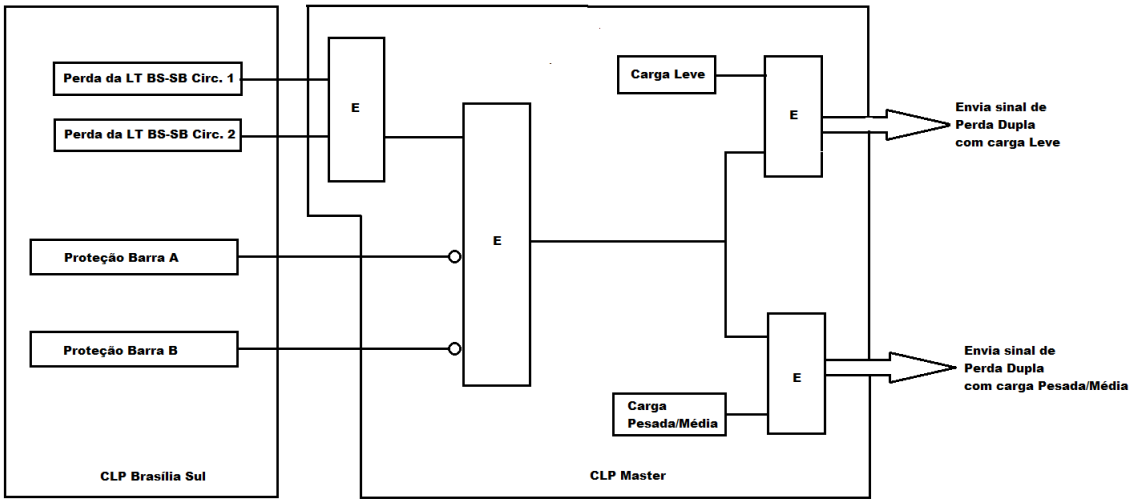

Figura 6.2: Diagrama com a Lógica de Perda Dupla.

6.1.2 Lógica de Perda Barra

Da mesma maneira que a descrita na seção anterior, esta lógica também monitora os dois circuitos Samambaia - Brasília Sul de 345 kV. Na ocorrência da perda de ambos, porém, com a atuação dos relés de proteção da barra de 345 kV na subestação Brasília Sul, o esquema executa os seguintes comandos:

- Fecha os disjuntores e chaves dos bancos de capacitores das subestações Brasília Sul, 06 e Brasília Centro;
- Abre a linha de transmissão Brasília Sul - Ceilândia C1 e C2;
- Abre a linha de transmissão Brasília Sul - Taguatinga C1 e C2;
- Abre a linha de transmissão Brasília Sul - Santa Maria;
- Corte adicional das cargas da CEB, caso as tensões nas barras da SE Brasília Sul 138 kV, Brasília Norte 13,8 kV, Brasília Centro 13,8 kV e SE 06 13,8 kV permanecerem inferiores a 90%, o corte adicional das cargas da CEB em função da barra da SE Brasília Sul 345 kV (A ou B) que foi desligada e do período de carga (pesada, leve ou média).

Na figura 6.3 é apresentado o diagrama com a Lógica de Perda de Barra. Como o diagrama da seção anterior, nos diagramas, procurou-se delimitar as lógicas, a monitoração e os comandos dentro do CLP onde cada uma dessas ações são realizadas.

6.1.3 A descrição da Ocorrência

No dia 05 de junho de 2011, houve a ocorrência que fez com que o erro na lógica do ECS da área Goiás/Brasília fosse identificado[26]. Às 12:15 hs, o esquema detectou, em situação de carga leve, a perda dos dois circuitos da linha de transmissão Brasília Sul - Samambaia. Entretanto, neste momento, uma das barras de 345 kV da SE Brasília Sul estava impedida.

Na lógica, em operação desde 10 de fevereiro de 2010, não havia previsão para a condição em que uma das barras estivesse impedida. O impedimento de uma das barras foi interpretado como condição de tensão degradada. A ocorrência, que era uma perda dupla, foi considerada pela lógica como perda de barra. Este erro provocou o envio de comandos de desligamentos indevidos, com o corte de 64 MW de carga, e com duração de 151 minutos.

Os detalhes desta ocorrência foram obtidos junto à Gerência de Estudos Especiais, Proteção e Controle do ONS.

6.2 Modelagem do ECS da área Goiás/Brasília em PROMELA

Na modelagem do Esquema de Controle e Segurança da área Goiás/Brasília, cada CLP utilizado no esquema foi representado por um processo. Um aspecto a ser observado no modelo implementado, quando um único CLP realiza simultaneamente a função de supervisão e controle, ele foi dividido em dois processos, um para cada uma das funções.

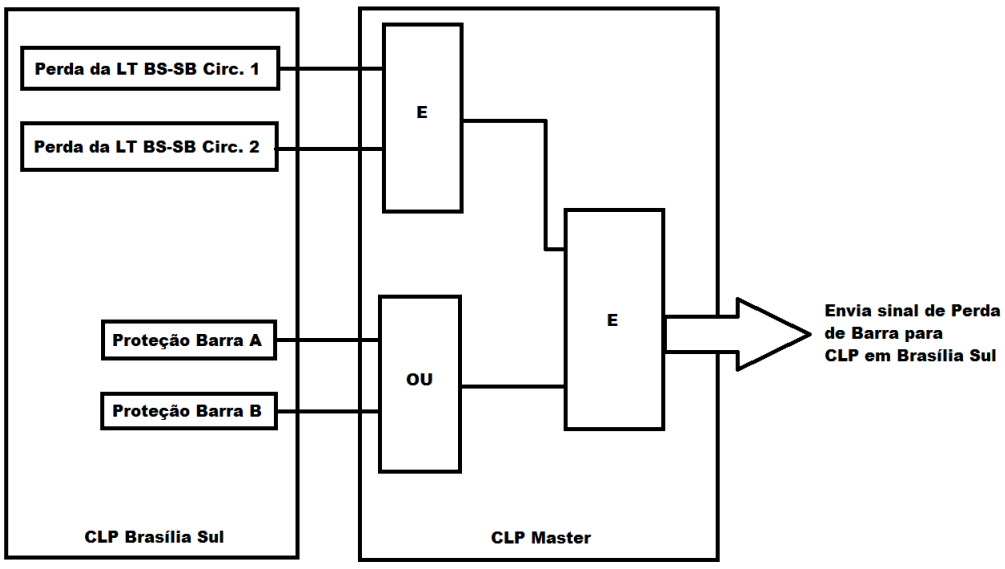

Figura 6.3: Diagrama com a Lógica de Perda de Barra.

Na figura 6.4 está representado o diagrama com a modelagem dos processos e canais de comunicação. Cada CLP foi modelado com um processo, exceto o de Brasília Sul, que foi divido em dois processos conforme a função a ser desempenhada.

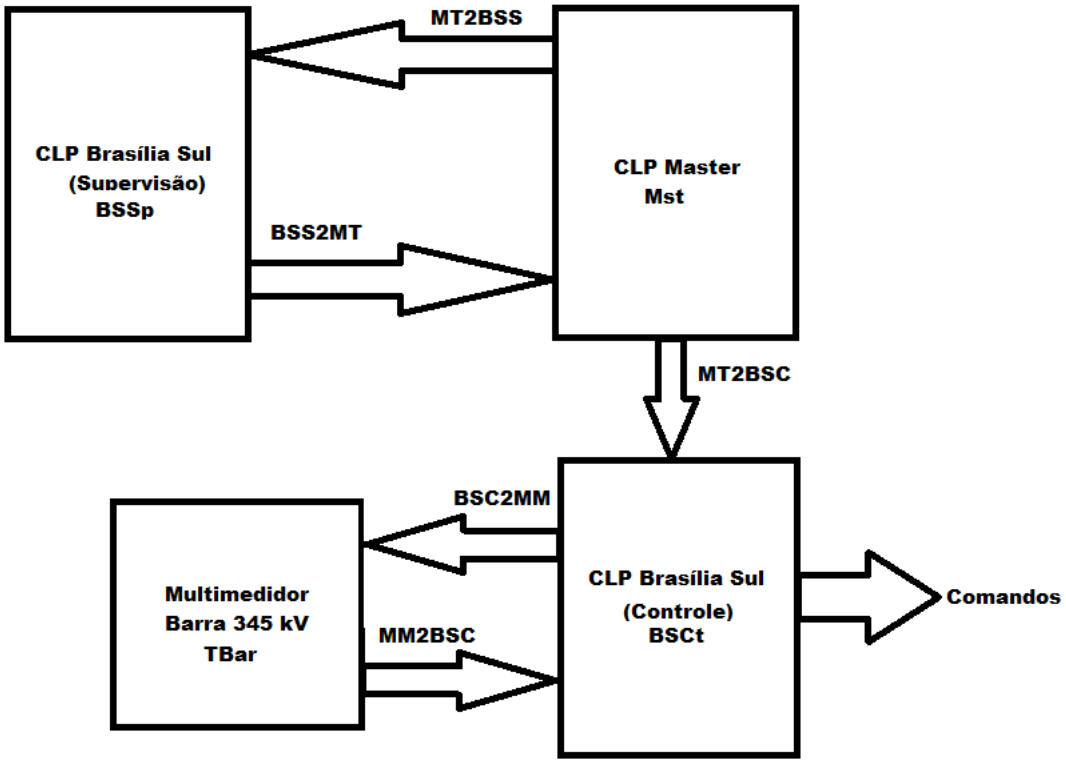

Figura 6.4: Representação do Modelo em PROMELA.

Na figura 6.5 tem-se as declarações das variáveis e *message channels* utilizados no modelo.

Nele, inicialmente podem ser observadas todas as definições das constantes simbólicas. Em seguida, estão declaradas as variáveis globais. Estas variáveis serão utilizadas nas verificações das propriedades. Optou-se por utilizar variáveis globais para esta finalidade porque deste modo é possível a inserção de assertivas em qualquer ponto do modelo.

Para a declaração dos *message channels*, foi definido, arbitrariamente, um formato de mensagem para troca de dados entre os processos. Em todas as mensagens, o primeiro byte é o endereço destino, o segundo o endereço de origem, seguido pelos bytes com a informação. Ao lado de cada declaração está o comentário com o formato da mensagem utilizada em cada canal.

O código do processo BSSp que modela o função supervisão do CLP de Brasília Sul está apresentado na figura 6.6.

Neste código, o processo aguarda, na linha 90, a interrogação do CLP Master

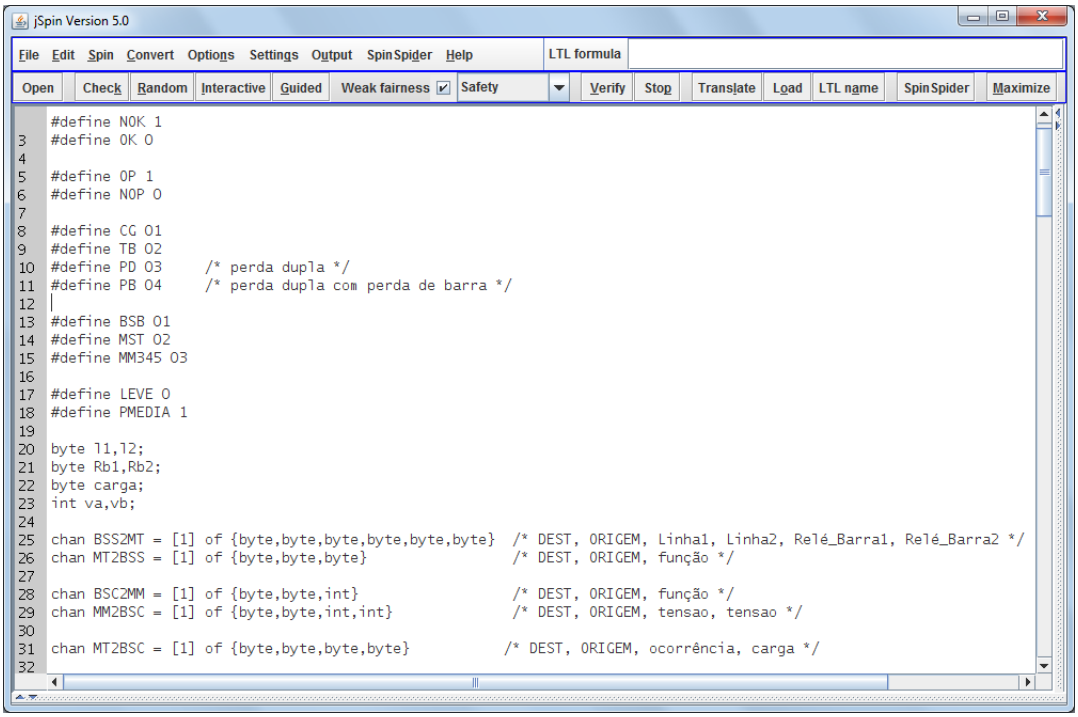

Figura 6.5: Declarações das variáveis e *message channels*.

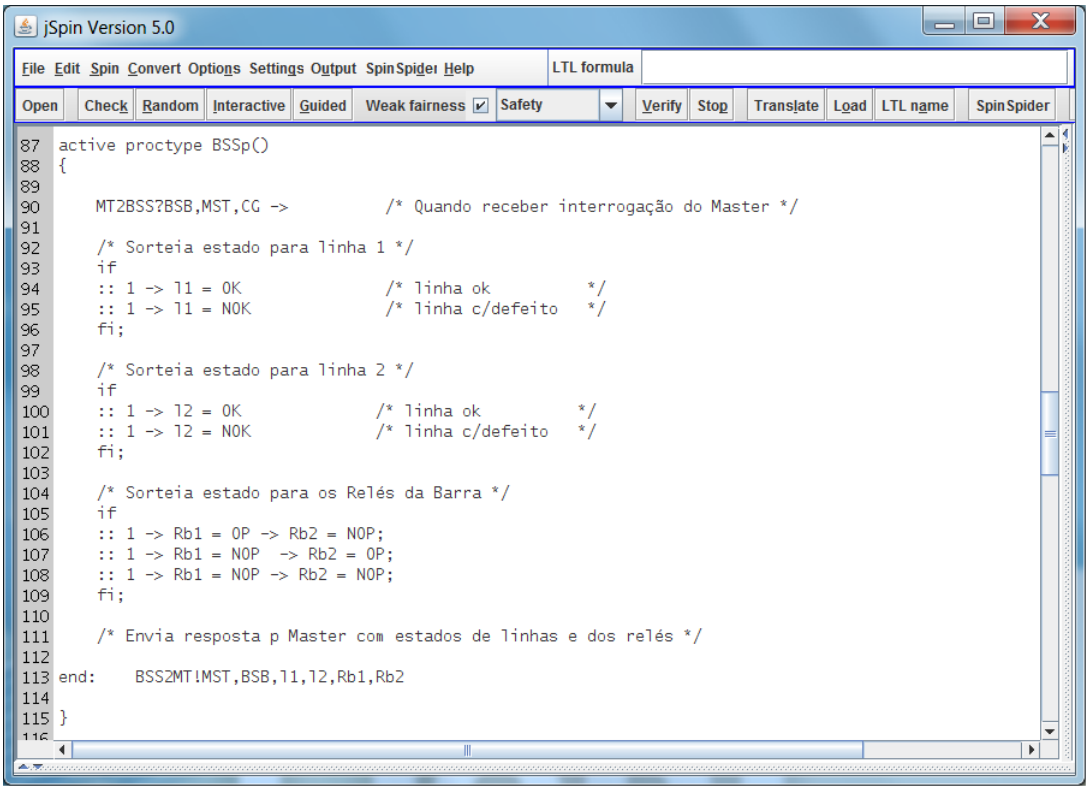

Figura 6.6: Código do processo Brasília Sul Supervisor.

sobre o estado das linhas e dos relés de proteção das barras. Quando recebe a mensagem com esta solicitação, através da estrutura não deterministica if, "sorteia" os estados possíveis para o estado dos dois circuitos da linha de transmissão, linha

93 a 102, e os estados dos relés de proteção de cada uma das barras de 345 kV da SE Brasília Sul, linhas 105 a 109.

Ao final de sua execução, envia, na linha 113, a resposta ao processo chamado Mst, que modela o CLP Master, e que o código está mostrado na figura 6.7.

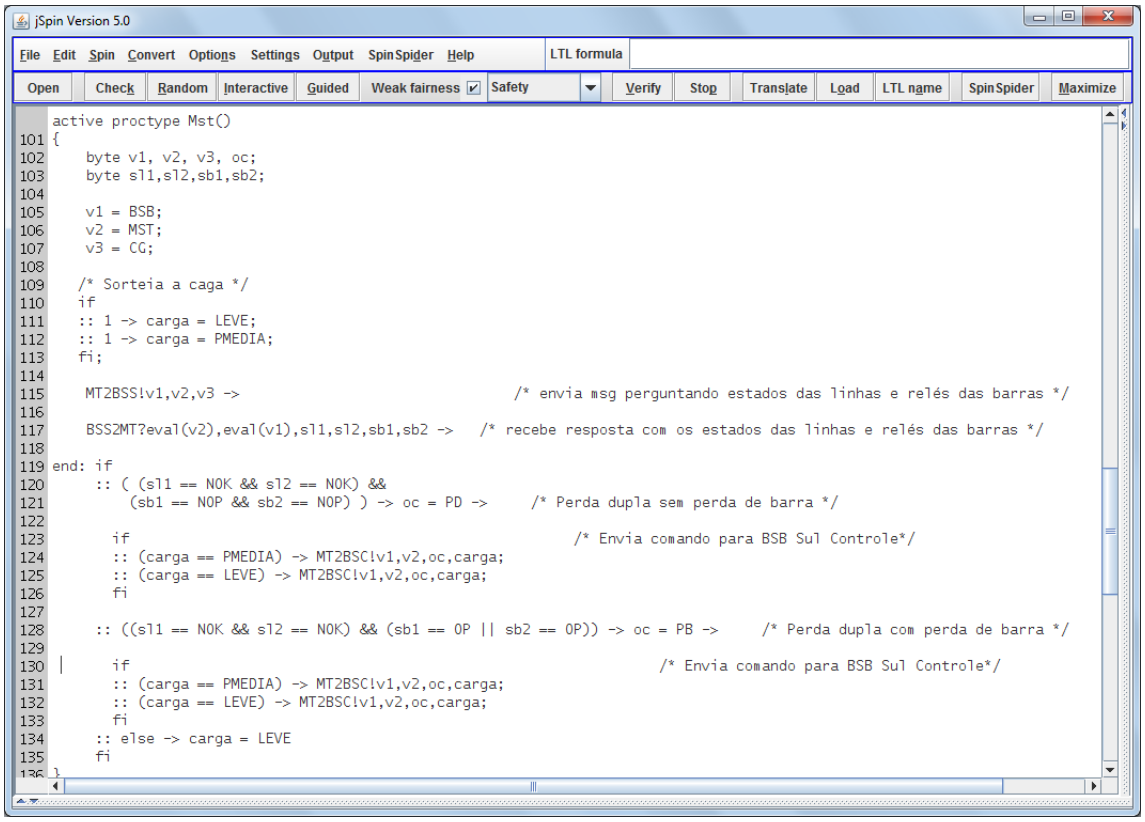

Figura 6.7: Código do processo Master.

Ao ser lançado, o processo Mst, na linha 110 a 113, "sorteia" o tipo de carga, leve ou média/pesada, que deverá ser considerada na verificação. Em seguida, envia, na linha 115, uma interrogação ao CLP supervisor de Brasília Sul, sobre os estados dos dois circuitos da linha de transmissão Brasília Sul - Samambaia e sobre os estados dos relés de proteção das barras de 345 kV da SE Brasília Sul.

Quando recebe, na linha 117, a mensagem com os estados solicitados, identifica o tipo de ocorrência: perda dupla ou perda de barra. De acordo com o tipo de carga, envia mensagem para o processo CLP Brasília Sul que modela as funções de controle com o tipo de ocorrência, e o tipo de carga, que pode ser observado entre as linhas 119 e 135.

Na figura 6.8 está apresentado o código do processo *BSCt* que modela a lógica do CLP de Brasília Sul responsável pelo envio dos comandos.

Seguindo exatamente a lógica implementada no CLP utilizado no sistema real, o processo *BSCt* ao ser lançado, aguarda, na linha 149, pela mensagem vinda do CLP Master com as informações com o tipo evento, e com a situação da carga a ser

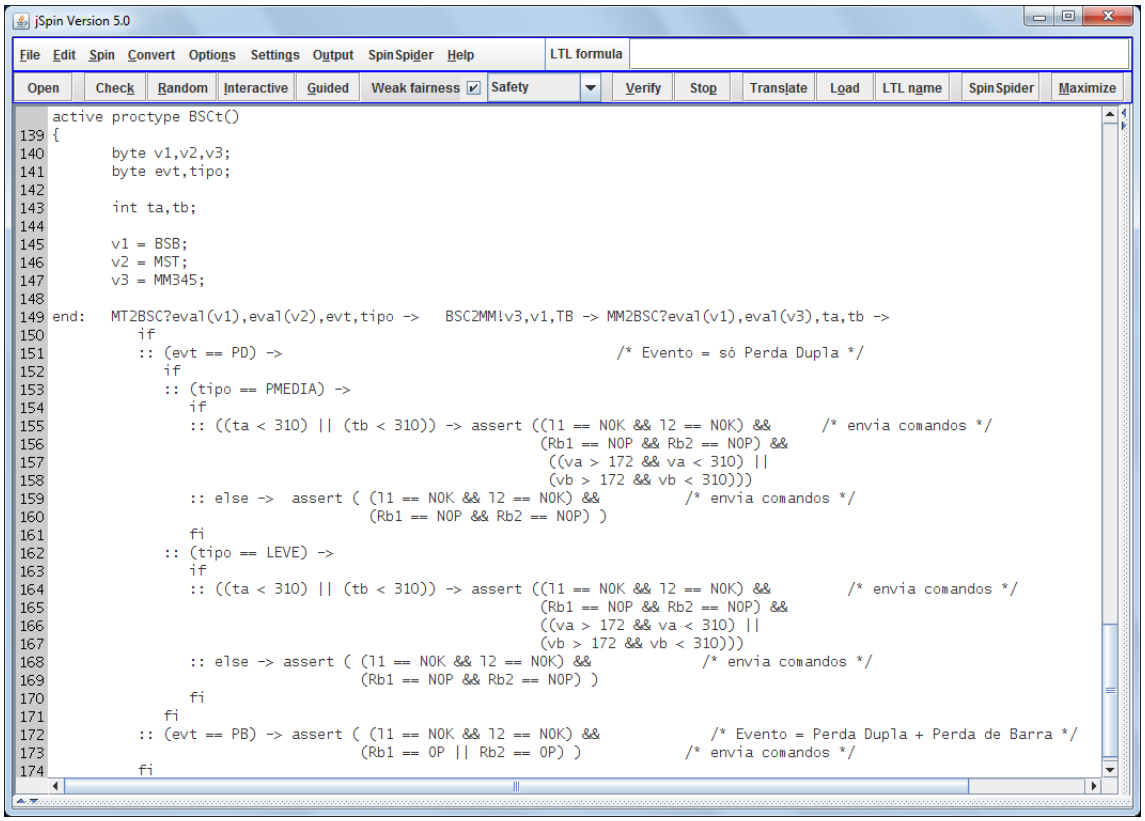

Figura 6.8: Código do processo Brasília Sul Controle.

considerada. Ao receber esta mensagem, o processo consulta o processo $TBar$, que modela o multimedidor, para obter o valor das tensões das barras de 345 kV da SE Brasília Sul. De posse de todas as informações necessárias, o processo então toma a decisão sobre os comandos a serem realizados.

Na figura 6.9 está o código do processo $Tbar$. Este processo simplesmente aguarda pela interrogação do processo $BSCt$ sobre os valores da tensões, entre as linhas 47 e 53, "sorteia" os valores e envia, na linha 65, a resposta com os valores sorteados. Deve ser observado que são atribuídas todas as combinações possíveis para os valores de tensões de cada uma das barras.

Voltando ao processo $BSCt$ da figura 6.8. Este ao receber a resposta com as tensões das barras, identifica o tipo de ocorrência, perda dupla ou perda de barra, o tipo de carga e por último, de acordo com os valores das tensões recebidas, envia os comandos para os outros CLP's.

Em uma modelagem real do ECS, deveriam existir um processo para cada CLP que recebesse o comando e em cada um deles seria feita a respectiva verificação.

Neste ponto, com único objetivo de diminuir a quantidade de processos, optou-se por uma simplificação no modelo, com a inserção das verificações das propriedades no processo $BSCt$.

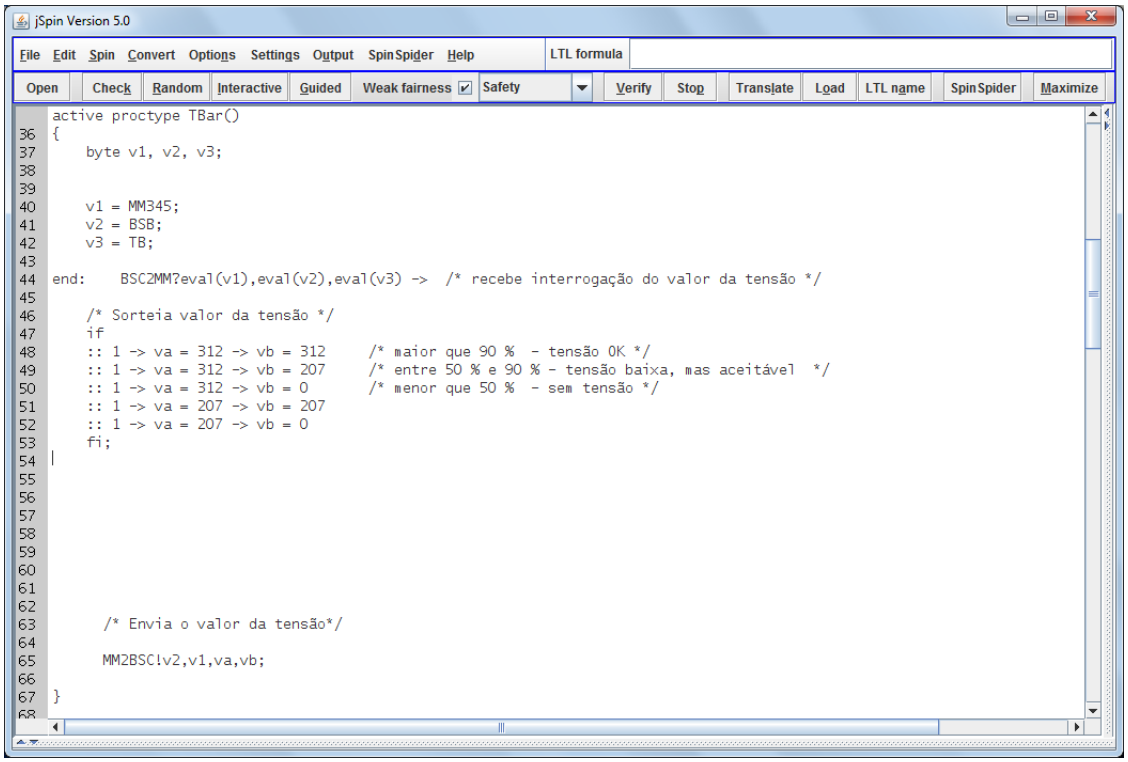

Figura 6.9: Código do processo TensaoBarra.

6.3 Verificação

Para a verificação deste caso, o modelo, que retratava exatamente sua implementação real no campo, foi submetido ao verificador SPIN. Na figura 6.10 está o resultado da verificação. Nele pode ser observado, através da mensagem *assertion violated* que um erro foi identificado.

Conforme mostrado no capítulo três, uma vez identificado um erro durante o processo de verificação, será através de uma simulação guiada, onde o SPIN utiliza o arquivo com extensão *.trail*, gerado na verificação, que a condição que levou ao erro poderá ser identificada.

Na figura 6.11, está apresentado o resultado da simulação guiada. Nele se pode ver exatamente a condição da ocorrência daquele dia. Onde na perda de ambas as linhas, $l1 = 1$ e $l2 = 1$, sem que tenha ocorrido a atuação dos relés de proteção, $Rb1 = 0$ e $Rb2 = 0$, ou seja, uma perda dupla, com uma das barras com tensão maior que 90% e a outra desligada, o ECS envia um comando indevido de desligamento.

Ao se corrigir o modelo, em uma nova verificação, o SPIN não produziu identificação de erros.

6.4 Comentários

Neste capítulo foi mostrado um caso real: um erro na lógica de um ECS da área Goiás - Brasília que provocou o desligamento indevido de linhas de transmissão da CEB. O esquema teve seu modelo representado em PROMELA, e ao ser validado, com uso do verificador SPIN, foi capaz de identificar um erro na ocorrência de uma perda dulpla com uma barra desligada. Desta forma, mostra-se a importância que o uso de ferramentas que utilizem técnicas de verificação formal podem vir a ter no projeto de aplicações em Sistemas Elétricos de Potência.

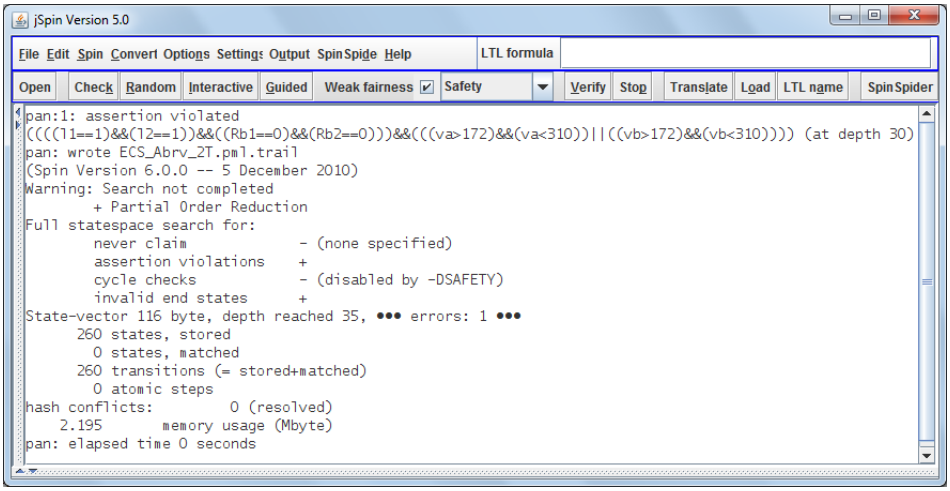

Figura 6.10: Resultado de Verficação.

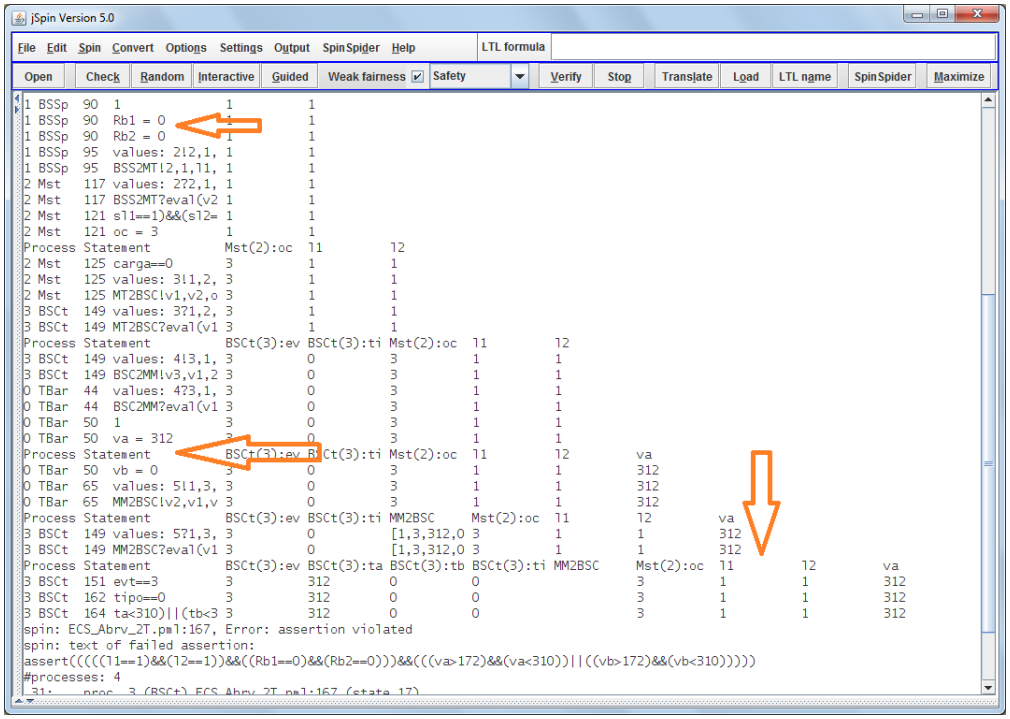

Figura 6.11: Simulação Guiada.

Capítulo 7

Conclusões

Com a evolução das técnicas para verificação formal de sistemas concorrentes, observa-se que seu uso poderá fazer uma grande contribuição para a melhoria na elaboração de projetos de aplicações para o setor elétrico.

Como os experimentos apresentados, mostrou-se que a área de projetos de sistemas elétricos de potência é mais uma das que pode colher os benefícios do uso da técnica de verificação formal de sistemas. Atualmente, com a utilização de relés microprocessados, que se comunicam através de rede de dados, os conceitos que antes eram associados à arquitetura de computadores passaram a ser utilizados em ambientes de Sistemas Elétricos de Potência, tanto em sistemas digitais de proteção e controle dentro de uma subestação, como em sistemas com uma abrangência geográfica maior. A comunicação entre equipamentos passou a ser uma característica fundamental nestes sistemas.

Esquemas de proteção de sistemas, formados por UTRs ou CLPs, que têm a capacidade de proteger uma grande área ou região do sistema elétrico tornam-se cada vez mais complexos. Portanto, se faz necessário o desenvolvimento de ferramentas efetivas de análise para avaliação e comparação de alternativas que venham aprimorar a elaboração do seu projeto.

O Model Checking provou ser uma ferramenta madura em diversas áreas. Nesta tese, o experimento apresentado no capítulo seis foi capaz de identificar um erro presente no ECS da área Goiás-Brasília, e que acabou por provocar um desligamento indevido em linhas de transmissão na área de concessão da CEB.

Assim, mostra-se ser de grande importância a investigação para que se encontre uma ferramenta madura para o projeto de aplicações para o setor elétrico. Sabe-se que um erro de software em sistemas críticos neste setor pode ter consequências catastróficas. E, quanto mais cedo puderem ser encontrados, menor sua consequência e o custo para resolvê-lo.

No caso real relatado, apresenta questões quanto a erros semelhantes que podem estar adormecidos, e quanto as consequências e custo para corrigi-los. Portanto o

uso da técnica de verificação formal de sistemas se torna ainda mais promissora.

Nesta tese, as ferramentas utilizadas na modelagem foram a linguagem PROMELA, juntamente com o verificador SPIN, que não foram desenvolvidas para aplicações do setor elétrico. Em trabalhos futuros, poderão ser realizadas alterações nas ferramentas atuais, de modo a se desenvolver uma específica, com uma interface amigável, onde tanto a modelagem quanto as propriedades possam ser expressas de forma mais intuitiva por especialistas da área.

Programas que realizam o controle automático para ligar e/ou desligar bancos de capacitores na rede, programas de reconfiguração automática da rede em caso de falha (*self-healing*) são exemplos de aplicações onde a verificação formal pode ser empregada.

No futuro, quando as ferramentas para a verificação formal de modelos forem capazes de avaliar sistemas com grande quantidade de variáveis, incluindo as do tipo real, talvez seja possível verificar se um determinado sistema elétrico, diante de uma perturbação, sempre voltará a sua condição normal de operação. Além dessas, também poderão ser identificadas outras áreas do Sistema Elétrico de Potência, em que a verificação formal possa ser empregada de modo a aumentar a confiabilidade de sua operação.

Referências Bibliográficas

[1] ONS. *ECS da Área Goiás - Brasília - Relatório de implantação.* Relatório técnico, Operador Nacional do Sistema Elétrico, sep 2009.

[2] ALESAADI, A. R., HAGHIFAM, M., FIROUZABAD, M., et al. "An approach for modeling the protection system in transmission network reliability evaluation". In: *Transmission and Distribution Conference and Exhibition: Asia and Pacific, 2005 IEEE/PES*, pp. 1–6, Dalian, dec 2005.

[3] MCCALLEY, J., OLUWASEYI, O., KRISHNAN, V., et al. *System Protection Schemes: Limitations, Risks, and Management.* Relatório técnico, Power Systems Engineering Research Center - PSERC, dec 2010.

[4] HSIAO, T.-Y., LU, C.-N. "Special Protection System Reliability Assessment". In: *Industrial & Commercial Power Systems Technical Conference, ICPS 2007. IEEE/IAS*, pp. 1–7, Edmonton, Alta, may 2007.

[5] BAIER, C., KATOEN, J.-P. *Principles of Model Checking.* Cambridge, Massachusetts, The MIT Press, 2007.

[6] ILIASOV, A. "Generation of certifiably correct programs from formal models". In: *First International Workshop on Software Certification*, pp. 43 – 48, nov 2011.

[7] SLOBODOVÁ, A., DAVIS, J., SWORDS, S., et al. "A Flexible Formal Verification Framework for Industrial Scale Validation". In: *9th IEEE-ACM International Conference on Formal Methods for Codesign (MEMOCODE)*, pp. 89 – 97, jul 2011.

[8] RAS, J., CHENG, A. M. K. "On Formal Verification of Toyota's Electronic Throttle Controller". In: *IEEE International System Conference*, pp. 552 – 555, apr 2011.

[9] FENG, L., CHEN, D., LONN, H., et al. "Verifying System Behaviors in EAST-ADL2 with the SPIN Model Checker". In: *IEEE International Conference on Mechatronics and Automation*, pp. 144 – 149, aug 2010.

[10] VAN DE MOLENGRAFT, R., BEETZ, M., FUKUDA, T. "Robot Challenges: Towards Development of Verification and Synthesis Techniques", *IEEE Robotics & Automation Magazine*, pp. 22–23, sep 2011.

[11] KEATING, D., MCINNES, A., HAYES, M. "Model Checking a TTCAN implementation". In: *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 387 – 396, mar 2011.

[12] RAMOS, G., SÁNCHES, J. L., TORRES, A., et al. "Power Systems Security Evaluation Using Petri Nets", *IEEE Transactions on Power Delivery*, v. 25, pp. 316–322, jan 2010.

[13] HAGIHARA, S., KITAMURA, Y., SHIMAKAWA, M., et al. "Extracting Environmental Constraints to Make Reactive System Specifications Realizable". In: *16th Asia-Pacific Software Engineering Conference*, pp. 61 – 68, dec 2009.

[14] CLARK, E. M., GRUMBERG, O., PELED, D. A. *Model Checking*. Cambridge, Massachusetts, The MIT Press, 2000.

[15] FISCHER, M. *An Introduction to Pratical Formal Methods Using Temporal Logic*. United Kingdom, John Wiley & Sons, Ltd., 2011.

[16] BÈRARD, B., OTHERS. *Systems and Software Verification: Model Checking Techniques and Tools*. Berlin - Germany, Springer-Verlag, 2001.

[17] CICHON, J., CZUBAK, A., JASINSKI, A. "Minimal Buchi Automata for Certain Classes of LTL Formulas". In: *Fourth International Conference on Dependability of Computer Systems. DepCos-RELCOMEX*, pp. 17 – 24, Brunow, sep 2009.

[18] FERREIRA, A. P. L. "Model Checking". In: *Workshop-School on Theorical Computer Science (WEIT)*, pp. 9 – 14, Pelotas, aug 2011.

[19] HOLZMANN, G. J. *The Spin Model Checker: Primer and Reference Manual*. Boston, Addison Wesley, 2003.

[20] HOLZMANN, G. J. "The Model Checker SPIN", *IEEE Transactions on Software Engineering*, v. 23, pp. 279–295, may 1997.

[21] GERS, J. M., HOLMES, E. J. *Protection of Electricity Distribution Networks*. 3 ed. London, United Kingdom, Institution of Engineering and Technology, 2011.

[22] LLORET, P., VELÁSQUEZ, J. L., MOLAS-BALADA, L., et al. "IEC 61850 as a flexible tool for electrical systems monitoring". In: *9th Internacional Conference Electric Power Quality and Utilisation*, Barcelona, oct 2007.

[23] MCCALLEY, J. D., FU, W. "Reliability of Special Protection Systems", *IEEE Transactions on Power Systems*, v. 14, pp. 1400–1406, nov 1999.

[24] SUN, X., REDFERN, M. A. "An Investigation into the Design of an IEC61850 based Protection Relay". In: *The 44th International Universities Power Engineering Conference (UPEC)*, pp. 1 – 5, Glasgow, sep 2009.

[25] DE ALMEIDA, P. C. *Esquemas de Proteção de Sistemas de Energia Elétrica*. Tese de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro, 2002.

[26] "Ata da 98^a reunião do Comitê de Monitoramento do Setor Elétrico do Ministério das Minas e Energia". jun 2011.

Apêndice A

Esquema Básico de Proteção de Circuitos

```
001#define I1MAX 100
002#define I3MAX 150
003
004byte i1 = 50;
005byte i2 = 50;
006byte i3;
007bit stCbA = 1;
008bit stCbB = 1;
009
010mtype ={trip, bloq, ack, nak, reset}
011
012chan R1toCbA = [1] of {mtype}
013chan CbAtoR1 = [1] of {mtype}
014
015chan R2toCbB = [1] of {mtype}
016chan CbBtoR2 = [1] of {mtype}
017
018chan R1toR2 = [1] of {mtype}
019
020
021proctype CbA()
022{
023    if
024    ::R1toCbA?trip -> atomic
025                      {
026                         stCbA = 0 -> i1 = 0 ->
```

```
027                                 i3 = i1 + i2 ->
028                                     CbAtoR1!ack -> goto fim1;
029                       }
030    ::R1toCbA?trip -> CbAtoR1!nak
031 -> goto fim1;
032    fi;
033
034fim1: skip
035}
036
037proctype CbB()
038{
039      if
040      :: R2toCbB?trip -> atomic
041                    {
042                      stCbB = 0 -> i3 = 0 ->
043                                i2 = 0 -> i1 = 0 -> CbBtoR2!ack
044                                    -> assert (stCbA == 1 && stCbB == 0)
045                                       -> goto fim2;
046                    }
047      :: R2toCbB?reset -> goto fim2;
048      fi;
049
050fim2: skip
051}
052
053
054proctype R1 ()
055{
056
057           (i1 > I1MAX) ->
058              atomic
059              {
060                 R1toCbA!trip -> R1toR2!bloq;
061              }
062
063           if
064           :: CbAtoR1?nak -> R1toR2!trip -> goto fim;
065           :: CbAtoR1?ack -> stCbA = 0 ->
```

```
066                     R1toR2!reset -> assert (stCbA == 0 && stCbB == 1)
067                           -> goto fim;
068           fi;
069fim: skip
070}
071
072
073proctype R2()
074{
075
076     (i3 >= I3MAX) ->
077
078            if
079            :: R1toR2?bloq ->
080                           if
081                           :: R1toR2?trip -> R2toCbB!trip ->
082                                 CbBtoR2?ack ->
083                                    assert (stCbB == 0 && stCbA == 1)
084                                       -> goto fim1;
085
086                           :: R1toR2?reset -> R2toCbB!reset -> goto fim1;
087                           fi;
088
089            :: timeout -> R2toCbB!trip -> CbBtoR2?ack ->
090                                    assert (stCbB == 0 && stCbA == 1)
091                                       -> goto fim1;
092            fi;
093fim1:       skip
094}
095
096init
097{
098   atomic
099   {
100      run CbA();
101      run CbB();
102      run R1();
103      run R2();
104      i1 = 120;
```

```
105       i2 = 50;
106       i3 = i1 + i2;
107    }
```

Apêndice B

Transferência Automática de Alimentação

```
/* definicoes dos enderecos e tipos de funcoes das mensagens */

001#define rl1 01
002#define rl2 02
003#define rb1 03
004#define rg1 04
005
006#define trip 11
007#define close 12
008#define g_out 13
009#define g_in 14
010#define bus_ok 15
011#define bus_nok 16
012
013#define ack 20
014#define nak 21
015
016/* declaracoes das variaveis */
017
018byte stl1;                    /* tensao na linha 1 */
019byte stl2;                    /* tensao na linha 2 */
020byte V = 138;                /* nivel da tensao */
021byte Vb1;                    /* tensao na barra */
022byte Vbmin;                  /* tensao minima na barra */
023
024bit stDj1;              /* status do Disjuntor #1 */
```

```
025bit stDj2;              /* status do Disjuntor #2 */
026bit stDjG;              /* status do Disjuntor da Geracao */
027
028
029chan rede = [1] of {byte,byte,byte}  /* estrutura da mensagem do channel rede
030                                      tipo,
031                                      endereco destino,
032                                      endereco origem */
033
034                                  /* channels entre reles e DJs */
035                                  /* Rele ---> Dj - trip/close  */
036                                  /* Dj ---> Rele - ack/nak     */
037chan Rl1toDj1 = [1] of {byte}
038chan Dj1toRl1 = [1] of {byte}
039chan Rl2toDj2 = [1] of {byte}
040chan Dj2toRl2 = [1] of {byte}
041chan RlgtoDjG = [1] of {byte}
042chan DjGtoRlg = [1] of {byte}
043
044/*****************************************************************/
045/*  Processo Dj1 - Modelo do comportamento do Disjuntor #1       */
046/*                                                               */
047/*  Recebe um comando de trip ou um comando de close             */
048/*  Em ambos os casos pode ou nao ter o comando executado        */
049/*  comando executado - altera o estado e responde ack           */
050/*  comando nao executado - responde nak                         */
051/*                                                               */
052/*****************************************************************/
053
054proctype Dj1()
055{
056    if
057       /* recebe trip, abre o Dj e envia ack */
058    ::Rl1toDj1?trip -> stDj1 = 0 -> Dj1toRl1!ack
059
060       /* recebe trip, nao abre o Dj e envia nak */
061    ::Rl1toDj1?trip -> Dj1toRl1!nak
062
063       /* recebe close, fecha o Dj e envia ack */
```

```
064    ::Rl1toDj1?close -> stDj1 = 1 -> Dj1toRl1!ack
065
066       /* recebe close, nao fecha o Dj e envia nak */
067    ::Rl1toDj1?close -> Dj1toRl1!nak
068    fi;
069}
070
071/*****************************************************************/
072/*  Processo Dj2 - Modelo do comportamento do Disjuntor #2       */
073/*                                                               */
074/*  Recebe um comando de trip ou um comando de close             */
075/*  Em ambos os casos pode ou nao ter o comando executado        */
076/*  comando executado - altera o estado e responde ack           */
077/*  comando nao executado - responde nak                         */
078/*                                                               */
079/*****************************************************************/
080
081
082proctype Dj2()
083{
084    if
085       /* recebe trip, abre o Dj e envia ack */
086    ::Rl2toDj2?trip -> stDj2 = 0 -> Dj2toRl2!ack
087
088       /* recebe trip, nao abre o Dj e envia nak */
089    ::Rl2toDj2?trip -> Dj2toRl2!nak
090
091       /* recebe close, fecha o Dj e envia ack */
092    ::Rl2toDj2?close -> stDj2 = 1 -> Dj2toRl2!ack
093
094       /* recebe close, nao fecha o Dj e envia nak */
095    ::Rl2toDj2?close -> Dj2toRl2!nak
096    fi;
097}
098
099
100/*********************************************************************/
101/*  Processo DjG - Modelo do comportamento do Disjuntor do Gerador   */
102/*                                                                   */
```

```
103/*  Recebe um comando de trip ou um comando de close                 */
104/*  Em ambos os casos pode ou nao ter o comando executado            */
105/*  comando executado - altera o estado e responde ack               */
106/*  comando nao executado - responde nak                             */
107/*                                                                   */
108/*******************************************************************/
109
110
111proctype DjG()
112{
113    if
114       /* recebe trip, abre o Dj e envia ack */
115    ::RlgtoDjG?trip -> stDjG = 0 -> DjGtoRlg!ack
116
117       /* recebe trip, nao abre o Dj e envia nak */
118    ::RlgtoDjG?trip -> DjGtoRlg!nak
119
120       /* recebe close, fecha o Dj e envia ack */
121    ::RlgtoDjG?close -> stDjG = 1 -> DjGtoRlg!ack
122
123       /* recebe close, nao fecha o Dj e envia nak */
124    ::RlgtoDjG?close -> DjGtoRlg!nak
125    fi;
126}
127
128/*******************************************************************/
129/*  Processo Rl1 - Modelo do comportamento do Rele #1                */
130/*                                                                   */
131/*  Quando a tensao na linha for menor que o minimo, abre o Dj #1    */
132/*  Se o Dj #1 abriu, pergunta ao Rb #1 se barra ok para entao       */
133/*  fechar o disjuntor #2.                                           */
134/*                                                                   */
135/*******************************************************************/
136
137proctype Rl1()
138{
139   if
140       /* Problema na linha #1 abre o disjuntor */
141   :: (stl1 == 0)-> Rl1toDj1!trip ->
```

```
142
143      if
144         /* Se Dj #1 aberto, verifica se barra esta ok para transferencia  */
145      :: Dj1toRl1?ack -> rede!bus_ok,rb1,rl1 ->
146
147         if
148
149            /* Se barra ok e Dj1 aberto envia close p/ Rele #2 */
150         :: rede?bus_ok,rl1,rb1 -> (stDj1 == 0) -> rede!close,rl2,rl1
151
152            /* Se barra Nok verifica assertiva ambos Djs abertos */
153         :: rede?bus_nok,rl1,rb1 -> assert (stDj1 == 0 && stDj2 == 0)
154         fi
155
156         /* se Dj #1 nao abriu, imprime mensagem */
157      :: Dj1toRl1?nak -> printf ("Dj1 nao abre");
158      fi
159
160      /* Se recebeu pedido de close de Rl2, verifica linha #1 e abre Dj #1 */
161   :: rede?close,rl1,rl2 -> (stl1 == 1) -> Rl1toDj1!close ->
162
163      if
164         /* Se Dj #1 abriu verifica se DjG aberto envia comando */
165      :: Dj1toRl1?ack -> (stDjG == 0) -> rede!close,rg1,rl1 ->
166
167           /* verifica Dj2 aberto, Dj1 e DjG fechados */
168           rede?g_in,rl1,rg1 -> assert (stDj1 == 1 && stDj2 == 0 && stDjG == 1)
169
170         /* se Dj #1 nao abriu, imprime mensagem */
171      :: Dj1toRl1?nak -> printf ("Dj1 nao abre");
172      fi
173   fi;
174}
175
176
177/*****************************************************************/
178/*  Processo Rl2 - Modelo do comportamento do Rele #2             */
179/*                                                               */
180/*  Quando a tensao na linha for menor que o minimo, abre o Dj #2  */
```

```
181/*  Se o Dj #2 abriu, pergunta ao Rb #1 se barra ok para entao       */
182/*  fechar o disjuntor #1.                                           */
183/*                                                                   */
184/*******************************************************************/
185
186
187proctype Rl2()
188{
189   if
190      /* Problema na linha #2 abre o disjuntor #2  */
191   :: (stl2 == 0)-> Rl2toDj2!trip ->
192
193      if
194         /* Se Dj #2 aberto, verifica se a barra esta ok para transferencia */
195      :: Dj2toRl2?ack -> rede!bus_ok,rb1,rl2 ->
196
197         if
198            /* Se barra ok e Dj2 aberto envia close p/ Rele #1 */
199         :: rede?bus_ok,rl2,rb1 -> (stDj2 == 0) -> rede!close,rl1,rl2
200
201            /* Se barra Nok verifica assertiva ambos Djs abertos */
202         :: rede?bus_nok,rl2,rb1 -> assert (stDj1 == 0 && stDj2 == 0)
203         fi
204
205         /* se Dj #2 nao abriu, imprime mensagem */
206      :: Dj2toRl2?nak -> printf ("Dj2 nao abre");
207      fi
208
209      /* Se recebeu pedido de close de Rl1, verifica linha #2 e abre Dj #2 */
210   :: rede?close,rl2,rl1 -> (stl2 == 1) -> Rl2toDj2!close ->
211
212
213end:  if
214
215         /* Se Dj #2 abriu verifica se DjG aberto envia comando p ligar Geracao
216      :: Dj2toRl2?ack -> (stDjG == 0) -> rede!close,rg1,rl2 ->
217
218            /* verifica Dj1 aberto, Dj2 e DjG fechados */
219           rede?g_in,rl2,rg1 -> assert (stDj1 == 0 && stDj2 == 1 && stDjG == 1)
```

```
220
221          /* se Dj #2 nao abriu, imprime mensagem */
222      :: Dj2toRl2?nak -> printf ("Dj2 nao abre");
223
224      fi
225   fi;
226}
227
228/*********************************************************************/
229/*  Processo Rb1 - Modelo do comportamento do Rele da Barra          */
230/*                                                                   */
231/*  Quando recebe a pergunta se a barra ta ok, verifca se tensao     */
232/*  eh maior que o limite e entao envia msg p/ Rg1 abrir o DjG       */
233/*  Se geracao desligada envia ok p/ rele da linha                   */
234/*                                                                   */
235/*********************************************************************/
236
237
238proctype Rb1()
239{
240   if
241      /* recebe pedido de Rele #1  */
242   :: rede?bus_ok,rb1,rl1 ->
243      if
244
245         /* verifica tensao envia msg p/ Rg1 desligar geracao da barra */
246      :: (Vb1 > Vbmin) -> rede!g_out,rg1,rb1 ->
247        if
248           /* Se geracao fora envia ok p rele #1 */
249        :: rede?g_out,rb1,rg1 -> rede!bus_ok,rl1,rb1
250
251           /* Se geracao nao saiu envia nok p rele */
252        :: rede?g_in,rb1,rg1 -> rede!bus_nok,rl1,rb1
253        fi
254
255         /* se tensao ok envia ok para rele #1 */
256      :: (Vb1 <= Vbmin) -> rede!bus_ok,rl1,rb1
257      fi
258
```

```
259      /* recebe pedido de Rl2, verifica tensao */
260      /* envia msg p/ Rg1 desligar geracao da barra */
261   :: rede?bus_ok,rb1,rl2 ->
262      if
263
264         /* verifica tensao envia msg p/ Rg1 desligar geracao da barra */
265      :: (Vb1 > Vbmin) -> rede!g_out,rg1,rb1 ->
266
267         if
268            /* Se geracao fora envia ok p rele #2 */
269         :: rede?g_out,rb1,rg1 -> rede!bus_ok,rl2,rb1
270
271            /* Se geracao nao saiu envia nok p rele #2 */
272         :: rede?g_in,rb1,rg1 -> rede!bus_nok,rl2,rb1
273         fi
274
275          /* se tensao ok envia ok para rele #2 */
276      :: (Vb1 <= Vbmin) -> rede!bus_ok,rl2,rb1
277      fi
278   fi;
279}
280
281/*********************************************************************/
282/*  Processo Rg1 - Modelo do comportamento do Rele do Gerador        */
283/*                                                                   */
284/*  Quando recebe a msg para abrir disjuntor, tenta abri-lo          */
285/*  Se disjuntor abriu, envia ok para Rb1                            */
286/*  Se disjuntor nao abriu, envia nok para Rb1                       */
287/*                                                                   */
288/*********************************************************************/
289
290proctype Rg1()
291{
292    if
293      /* Recebe msg para abrir geracao desliga disjuntor */
294    ::rede?g_out,rg1,rb1 -> RlgtoDjG!trip ->
295       if
296         /* Se disjuntor abriu envia ack p Rb1 */
297       ::DjGtoRlg?ack -> rede?g_out,rg1,rb1
```

```
298
299         /* Se disjuntor nao abriu envia nak p Rb1 */
300       ::DjGtoRlg?nak -> rede?g_in,rg1,rb1
301       fi;
302      /* Recebe msg para fechar geracao liga disjuntor */
303    ::rede?g_in,rg1,rl1 -> RlgtoDjG!close ->
304       if
305         /* Se disjuntor fechou envia ack p Rl1 */
306       ::DjGtoRlg?ack -> rede?g_in,rl1,rg1
307
308         /* Se disjuntor nao fechou envia nak p Rl1 */
309       ::DjGtoRlg?nak -> rede?g_out,rl1,rg1
310       fi;
311
312      /* Recebe msg para fechar geracao liga disjuntor     */
313    ::rede?g_in,rg1,rl2 -> RlgtoDjG!close ->
314       if
315         /* Se disjuntor fechou envia ack p Rl2 */
316       ::DjGtoRlg?ack -> rede?g_in,rl2,rg1
317
318         /* Se disjuntor nao fechou envia nak p Rl2 */
319       ::DjGtoRlg?nak -> rede?g_out,rl2,rg1
320       fi;
321
322    fi;
323}
324
325init
326{
327
328
329     if
330     :: 1 -> stl1 = 0 -> stl2 = 1 -> Vb1 = V -> Vbmin = V;
331     :: 1 -> stl1 = 1 -> stl2 = 0 -> Vb1 = V -> Vbmin = V;
332     fi;
333
334
335    if
336    :: 1 -> stDj1 = 1 -> stDj2 = 0 -> stDjG = 1;
```

```
337     :: 1 -> stDj1 = 0 -> stDj2 = 1 -> stDjG = 1;
338     fi;
339
340     run Dj1 ();
341     run Dj2 ();
342     run DjG ();
343     run Rl1 ();
344     run Rl2 ();
345     run Rb1 ();
346     run Rg1 ();
347}
```

Apêndice C

Esquema de Controle de Segurança

```
#define NOK 1
#define OK 0

#define OP 1
#define NOP 0

#define CG 01
#define TB 02
#define PD 03     /* perda dupla */
#define PB 04     /* perda dupla com */
                  /* perda de barra */


#define BSB 01
#define MST 02
#define MM345 03

#define LEVE 0
#define PMEDIA 1

byte l1;
byte l2;

byte Rb1;
byte Rb2;
```

```
byte carga;

int vb;

/* DEST, ORIGEM, Linha1, Linha2, Rele_Barra1, Rele_Barra2 */
chan BSS2MT = [1] of {byte,byte,byte,byte,byte,byte}


chan MT2BSS = [1] of {byte,byte,byte}      /* DEST, ORIGEM, funcao */

chan BSC2MM = [1] of {byte,byte,int}       /* DEST, ORIGEM, funcao */
chan MM2BSC = [1] of {byte,byte,int}       /* DEST, ORIGEM, estado */

chan MT2BSC = [1] of {byte,byte,byte,byte} /* DEST, ORIGEM, ocorrencia, carga */



active proctype TBar()
{
    byte v1, v2, v3;


    v1 = MM345;
    v2 = BSB;
    v3 = TB;

        /* recebe interrogacao do valor da tensao */
end:    BSC2MM?eval(v1),eval(v2),eval(v3) ->

    /* Sorteia valor da tensao */
    if
    :: 1 -> vb = 312     /* maior que 90 %  - tensao OK */
    :: 1 -> vb = 207     /* entre 50 % e 90 % - tensao baixa, mas aceitavel  */
    :: 1 -> vb = 0       /* menor que 50 %  - sem tensao */
    fi;
```

```
      /* Envia o valor da tensao*/

      MM2BSC!v2,v1,vb;

}

active proctype BSSp()
{

    /* Quando receber interrogacao do Master */
    MT2BSS?BSB,MST,CG ->

    /* Sorteia estado para linha 1 */
    if
    :: 1 -> l1 = OK      /* linha ok          */
    :: 1 -> l1 = NOK     /* linha c/defeito   */
    fi;

    /* Sorteia estado para linha 2 */
    if
    :: 1 -> l2 = OK      /* linha ok          */
    :: 1 -> l2 = NOK     /* linha c/defeito   */
    fi;

    /* Sorteia estado para os Reles da Barra */
    if
    :: 1 -> Rb1 = OP -> Rb2 = NOP;
    :: 1 -> Rb1 = NOP  -> Rb2 = OP;
    :: 1 -> Rb1 = NOP -> Rb2 = NOP;
    fi;

    /* Envia resposta p Master com estados de linhas e dos reles */
```

```
end:    BSS2MT!MST,BSB,l1,l2,Rb1,Rb2

}


active proctype Mst()
{

    byte v1, v2, v3, oc;
    byte sl1,sl2,sb1,sb2;

    v1 = BSB;
    v2 = MST;
    v3 = CG;


   /* Sorteia a caga */
   if
   :: 1 -> carga = LEVE;
   :: 1 -> carga = PMEDIA;
   fi;

    /* envia msg perguntando estados das linhas e reles das barras */
    MT2BSS!v1,v2,v3 ->

   /* recebe resposta com os estados das linhas e reles das barras */
    BSS2MT?eval(v2),eval(v1),sl1,sl2,sb1,sb2 ->

end: if
        /* Perda dupla sem perda de barra */
     :: ( (sl1 == NOK && sl2 == NOK) &&
          (sb1 == NOP && sb2 == NOP) ) -> oc = PD ->

       if
           /* Envia comando para BSB Sul Controle*/
       :: (carga == PMEDIA) -> MT2BSC!v1,v2,oc,carga;
       :: (carga == LEVE) -> MT2BSC!v1,v2,oc,carga;
       fi
```

```
          /* Perda dupla com perda de barra */
      :: ((sl1 == NOK && sl2 == NOK) &&
          (sb1 == OP || sb2 == OP)) -> oc = PB ->

        if
            /* Envia comando para BSB Sul Controle*/
        :: (carga == PMEDIA) -> MT2BSC!v1,v2,oc,carga;
        :: (carga == LEVE) -> MT2BSC!v1,v2,oc,carga;
        fi
      :: else -> carga = LEVE
      fi
}

active proctype BSCt()
{
        byte v1,v2,v3;

        byte evt,tipo;

        int tb;

        v1 = BSB;
        v2 = MST;
        v3 = MM345;

end:    MT2BSC?eval(v1),eval(v2),evt,tipo ->

        BSC2MM!v3,v1,TB ->


        MM2BSC?eval(v1),eval(v3),tb ->

           if
               /* Evento = so Perda Dupla */
           :: (evt == PD) ->
              if
              :: (tipo == PMEDIA) ->
                 if
                     /* envia comando */
```

```
                :: (tb < 310) -> assert ((l1 == NOK && l2 == NOK) &&
                                         (Rb1 == NOP && Rb2 == NOP) &&
                                         (vb > 172 && vb < 310) )

                   /* envia comando */
                :: else ->  assert ( (l1 == NOK && l2 == NOK) &&
                                     (Rb1 == NOP && Rb2 == NOP) )
                fi
             :: (tipo == LEVE) ->
                if
                :: (tb < 310) -> assert ((l1 == NOK && l2 == NOK) &&
                                         (Rb1 == NOP && Rb2 == NOP) &&
                                         (vb > 172 && vb < 310) )

                :: else -> assert ( (l1 == NOK && l2 == NOK) &&
                                    (Rb1 == NOP && Rb2 == NOP) )
                fi
             fi

              /* Evento = Perda de Barra */
          :: (evt == PB) -> assert ( (l1 == NOK && l2 == NOK) &&
                                     (Rb1 == OP || Rb2 == OP) )
          fi
}
```