Revisão de modelos formais de sistemas de estados finitos

Thiago Carvalho de Sousa

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO DE MESTRE
EM
CIÊNCIAS

Área de Concentração : Ciência da Computação
Orientadora : Profa. Dra. Renata Wassermann

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da CAPES

São Paulo, março de 2007

Revisão de modelos formais de sistemas de estados finitos

Este exemplar corresponde à redação
final da dissertação devidamente
corrigida e defendida por
Thiago Carvalho de Sousa
e aprovada pela Comissão Julgadora.

São Paulo, abril de 2007.

Banca examinadora:

- Prof. Dra. Renata Wassermann (Orientadora) - IME-USP
- Prof. Dr. Marcelo Finger - IME-USP
- Prof. Dr. Mario Folhadela Benevides - UFRJ

Ao querido tio Mozart
À amada tia Lulu

Agradecimentos

Inicialmente, agradeço a Deus por ter me dado a vida e por estar vivendo este momento - a minha fé me fez caminhar mesmo nos momentos de dúvidas e incertezas. Aos meus pais, Sebastião e Maria do Socorro, e minhas irmãs, Mariana e Amanda, pelo grande apoio, compreensão, amor e carinho que me ofereceram durante esta jornada. Queria também agradecer à minha enorme família pelo apoio incondicional, apesar de eu não estar presente em alguns momentos especiais. Sei que vocês torceram muito por mim. Marina, você merece um beijo especial. Sem você eu não conseguiria.

Sou eternamente grato à professora Renata Wassermann, minha estimada orientadora, pelos esclarecimentos e orientações precisas, bem como pela sua paciência e amizade. Gostaria também de agradecer aos docentes do IME pela excelente formação que me proporcionaram durante a graduação e o mestrado. Aos professores Marcelo Finger, Mário Benevides e Flávio Soares que participaram da banca de defesa e da qualificação. Aos professores Sérgio Campos (UFMG) e Roberto Cavada (ITC-IRST) pela ajuda que me deram com a implementação da ferramenta.

Meus agradecimentos aos meus irmãos de São Paulo: Eduardo, Vladimir, Christian Paz, Karina, Jesus e Cristian Bayes. Com eles, os momentos vividos foram inesquecíveis. Aos meus camaradas de graduação e pós-graduação pelo apoio nos momentos de dúvidas e desespero. Aos meus amigões de infância pela solidariedade. Aos meus companheiros de trabalho na Hyperion e na CIS pelo incentivo.

Finalmente, agradeço à sociedade brasileira que, através da USP e da CAPES, permitiu-me realizar esse sonho. Espero que este trabalho contribua de alguma forma para o desenvolvimento do nosso país.

Resumo

Neste trabalho apresentamos uma implementação de revisão de crenças baseada em comparação de modelos (estados) em uma ferramenta de verificação automática de sistemas de estados finitos. Dada uma fórmula (na lógica CTL) inconsistente com o modelo do sistema, revisamos esse modelo de tal maneira que essa fórmula temporal se torne verdadeira. Como temos oito operadores temporais (AG, AF, AX, AU, EG, EF, EX e EU), foram criados algoritmos específicos para cada um deles.

Como o modelo do sistema deriva do seu código na linguagem SMV, a sua revisão passa obrigatoriamente por mudanças na sua descrição. A nossa implementação contempla três tipos de mudanças: acréscimo de linhas, eliminação de linhas e mudança no estado inicial, sendo que as duas primeiras provocam modificações nas transições entre os estados que compõe o modelo.

Alguns testes foram aplicados para comprovar a contribuição da revisão de crenças (revisão de modelos) como ferramenta de auxílio ao usuário durante o processo de modelagem de sistemas.

Abstract

In this work we present an implementation of belief revision based on comparison of models (states) in a tool for automatic verification of finite state systems. Given a formula (in the language of CTL) inconsistent with the model of the system, we revise this model in such way that the temporal formula becomes valid. As we have eight temporal operators (AG, AF, AX, AU, EG, EF, EX and EU), specific algorithms for each one of them have been created.

As the model of the system is related with its code in SMV language, its revision forces changes in its description. Our implementation contemplates three types of change: addition of lines, elimination of lines and change in the initial state, where the first two cause modifications in the transitions between the states of the model.

Some tests were applied to prove the contribution of the belief revision (model revision) as aid-tool to the user during the process of systems modeling.

Sumário

Lista de Figuras

Capítulo 1

Introdução

A engenharia de software é um braço da ciência da computação relacionado ao desenvolvimento de software e que tem como objetivo estudar métodos e técnicas para a construção e manutenção de sistemas. Podemos comparar esses métodos e técnicas característicos da engenharia de software com métodos e técnicas similares que ocorrem em outras áreas mais tradicionais de engenharia.

As primeiras construções (edifícios e casas) não possuíam garantia alguma de que elas se sustentariam caso alguma tempestade ou um leve terremoto ocorresse. A necessidade de construir casas que garantissem a segurança de seus habitantes provocou uma evolução baseada no estudo de modelos físicos-matemáticos. Uma planta de engenharia civil nos tempos atuais, por exemplo, não só descreve o objeto a ser construído como também possui uma série de cálculos estruturais que garantem que a construção suportará tempestades e terremotos leves. O que se percebe é que essa evolução natural para um formalismo mais apurado está também ocorrendo na engenharia de software, uma vez que as exigências de confiabilidade e segurança de um software têm aumentado a cada dia.

Uma das principais sub-áreas da engenharia de software é a chamada engenharia de requisitos, que tem como objetivo levantar, modelar, analisar e validar a consistência e viabilidade dos requisitos dos usuários. A especificação desses requisitos pode ser feita através de um modelo abstrato do sistema que se deseja construir. Esse modelo descreve "o que o sistema deve fazer" e "o que o sistema não deve fazer", sem se preocupar em dizer "como fazer".

Infelizmente, em sistemas complexos, esse modelo na maioria das vezes apresenta inconsistências, não podendo ser validado com os requisitos dos usuários. Além disso, os usuários quase sempre desejam alterar ou acrescentar novas funcionalidades ao sistema, o que ocasiona uma remodelagem contínua da especificação.

A solução para esses problemas se torna uma tarefa árdua se esse modelo for descrito em linguagem natural ou em linguagens semi-formais (eg. diagramas use-cases). Uma pesquisa de 2002 realizada pelo órgão americano Department of Commerce's National

Institute of Standards and Technology (NIST) (http://www.nist.gov) indica que, em alguns sistemas, 80% dos custos de desenvolvimento são consumidos pela verificação de consistência e remodelagem da especificação dos usuários. A área de métodos formais, através de notações e linguagens com precisão lógica e matemática (eg. Z, B, VDM, ASM, etc) e verificadores automáticos de modelos (eg. NuSMV, CMU SMV, SPIN, etc), tenta prover uma maneira mais eficiente para se remodelar e validar a consistência das especificações de um sistema, uma vez que isso é pré-requisito essencial quando se deseja construir um software confiável e seguro.

A inteligência artificial é uma outra área da ciência da computação que estuda a possibilidade de se imitar comportamentos tipicamente humanos via computador. Uma das grandes dificuldades da área está relacionada à representação do conhecimento humano em uma linguagem simbólica com um grande poder de expressividade.

As primeiras modelagens da inteligência surgiram na década de 50 e tiveram como base a lógica dos predicados. Desde então, novos tipos de modelagem têm surgido com o uso de regras de produção, lógica não-monotônica, etc, o que têm agregado mais expressividade à representação do conhecimento e contribuído para o crescimento dessa área.

Uma das sub-áreas da inteligência artificial é a chamada revisão de crenças, que se utiliza da idéia de representar o conhecimento humano através de suas crenças sobre o mundo. A principal idéia da revisão de crenças é como manter a consistência de um conjunto de crenças (eg. todas as suas crenças sobre os feriados nacionais) quando se recebe uma nova crença que contradiz as suas crenças atuais (eg. você acredita que amanhã não é feriado e recebe a informação que amanhã é feriado). Essa teoria provê várias abordagens que tratam de solucionar esse problema, muitas delas aplicáveis computacionalmente.

1.1 Motivação

Vimos que um grande problema da engenharia de software é o elevado custo de verificar a consistência do modelo de especificação de um sistema, descobrir a origem dos erros e remodelá-lo de tal forma que a inconsistência seja eliminada. Vimos também que existe uma área da inteligência artificial que provê soluções para a manutenção da consistência de um conjunto de crenças quando se recebe uma nova crença que é contraditória com o conjunto original. Se utilizarmos uma pequena parte da especificação como uma crença recebida que gera inconsistência, podemos ver que a área de revisão de crenças se encaixa como possível solução para esse problema da engenharia de software.

O que se propõe é uma integração inicial entre essas duas áreas da computação como uma forma de efetiva aplicação da teoria da revisão de crenças. Especificamente, integraremos a solução proposta por esse campo da inteligência artificial ao verificador

de modelos NuSMV de tal forma que, removendo ou adicionando partes no código que descreve o modelo do sistema, o usuário consiga mantê-lo consistente.

1.2 Organização

Apresentaremos inicialmente no capítulo 2 a teoria da revisão de crenças com as suas principais construções, o que nos fornecerá a base teórica para propor uma solução ao problema descrito.

No capítulo 3 apresentaremos a verificação automática de modelos, uma das mais usadas técnicas formais de verificação de modelos de software e que servirá de base para a implementação da solução proposta. Discutiremos as lógicas e algoritmos de detecção de inconsistências utilizados por essa técnica. Mostraremos também o NuSMV, um dos mais famosos verificadores automáticos de modelos e que por possuir código aberto nos servirá de base para uma implementação que integrará essas duas áreas. Apresentaremos suas principais características, bem como o seu funcionamento.

No capítulo 4 mostraremos um algoritmo que soluciona os problemas de inconsistências encontrados pelo NuSMV usando revisão de crenças, bem como a sua simulação em um exemplo descrito na linguagem SMV.

No capítulo 5 falaremos sobre os trabalhos relacionados a essa pesquisa, bem como as conclusões, as contribuições e trabalhos futuros. O apêndice A possui a sintaxe do NuSMV. Já o apêndice B contém uma documentação técnica das principais alterações realizadas no código fonte do verificador e o apêndice C contém os exemplos e os resultados dos testes realizados.

Capítulo 2

Revisão de Crenças

A necessidade de modelar o comportamento de bases de conhecimento dinâmicas que ao receberem uma certa informação se tornam inconsistentes formou a base da teoria de revisão de crenças. Essa teoria começou a se solidificar a partir do início da década de 80 quando Alchourrón, Gärdenfors e Makinson [1] propuseram alguns postulados que descrevem as mínimas propriedades que um processo de revisão deve ter. Esses postulados ficaram conhecidos como postulados AGM em homenagem aos autores e, desde então, têm sido bastante usados na área de modelagem de sistema dinâmicos. Para maiores informações consulte os livros clássicos de revisão de crenças [2, 3].

2.1 Um exemplo

Mostraremos o seguinte exemplo informal a fim de elucidarmos melhor a teoria da revisão de crenças. Suponha que se tenha uma base de dados que contenha as seguintes informações:

α : Todo brasileiro é pobre
β : João é brasileiro
λ : Quem ganha na mega-sena é rico
ϕ : Ricardo é jogador de futebol

Suponha também que essa base de dados possua um mecanismo lógico de inferência e que portanto seja capaz de deduzir que "João é pobre". Se essa mesma base de dados receber a informação "João ganha na mega-sena", a mesma ficará inconsistente, uma vez que, intuitivamente, João não pode ser rico e pobre ao mesmo tempo. Como o que se deseja é manter a base de dados sempre consistente, é preciso revisá-la, o que significa abandonar algumas das informações (crenças) originais. Poderíamos simplesmente abandonar todas as crenças originais e considerar apenas a nova. Mas isso não é necessário e nem desejável, pois acarretaria perda de informações valiosas (no nosso caso,

por exemplo, "Ricardo é jogador de futebol"). Um dos princípios que rege a teoria de revisão de crenças é que a mudança no meu conjunto original de crenças seja a mínima possível. Esse é o Princípio da Mudança Mínima. Em outras palavras, o que se deseja é reter o máximo de informação possível. No exemplo acima, se abandonássemos a crença α, β ou λ nossa base de dados voltaria a ficar consistente.

O problema da revisão de crenças pode tornar-se mais complicado se as crenças possuírem conseqüências lógicas. Nesse caso, é preciso decidir também quais as conseqüências a serem mantidas e quais devem ser abandonadas. Um outro questionamento possível é que a informação recebida seja descartada imediatamente, mantendo-se o conjunto original de crenças, que já era consistente. Mas a teoria da revisão de crenças sempre supõe que a nova crença é aceita, o que tem provocado críticas e gerado novas abordagens sobre o tratamento da crença recebida.

2.2 Operadores

Como já mostramos, a revisão de crenças tem como papel "consertar" uma base de dados que era consistente e que se tornou inconsistente depois da adição de uma nova crença. Mas nem sempre é necessário consertá-la, uma vez que a nova informação recebida seja consistente com as crenças originais. Dessa maneira, existem três tipos de operadores de mudanças no conjunto de crenças (K). São eles:

- Expansão: uma informação consistente α, juntamente com as suas conseqüências lógicas, é adicionada ao conjunto de crenças K. A representação desse operador é $K + \alpha$;

- Contração: a informação α é abandonada pelo conjunto K. Sendo o conjunto K fechado logicamente, talvez seja necessário também abandonar outras crenças, pois elas implicariam α. A representação desse operador é $K - \alpha$;

- Revisão: uma informação α é adicionada ao conjunto K e para se manter a consistência pode ser preciso abandonar outras crenças de K. A representação desse operador é $K * \alpha$.

2.3 Representação das Crenças

A teoria da revisão de crenças se concentra nos estados epistêmicos (de conhecimento, crença) e nas mudanças ocorridas nesses estados. Já vimos na seção anterior quais operadores são responsáveis pelas mudanças. No entanto, é necessário definir uma representação para esses estados. A representação proposicional, onde uma sentença pode ser vista como uma fórmula, é uma das mais comuns.

Até o final do capítulo, utilizaremos como representação de um estado epistêmico a linguagem L que inclui a lógica clássica proposicional com seus conectivos lógicos. Além disso, se K implica logicamente α, escreveremos como $K \vdash \alpha$. Se K não implica logicamente α, escreveremos $K \nvdash \alpha$ e se K for um conjunto de sentenças, representaremos $\mathrm{Cn}(K)$ como o conjunto de todas as consequências lógicas de K, ou seja, $\mathrm{Cn}(K) = \{\alpha : K \vdash \alpha\}$. Serão utilizados também símbolos usuais da teoria dos conjuntos, tais como $\in$ e $\subseteq$.

Há várias maneiras de modelar um estado epistêmico usando a linguagem L, dentre os quais se destacam:

* Conjunto de Crenças: é a maneira mais simples de modelar um estado epistêmico. Um conjunto de crenças caracteriza-se por ser um conjunto de fórmulas logicamente fechado, ou seja, dado um conjunto K, se K implica logicamente α, então $\alpha \in K$. Ao usar essa representação, estamos propensos a trabalhar com conjuntos infinitos, o que não é apropriado do ponto de vista computacional.

* Base de Crenças: essa representação supõe que um conjunto de crenças possui uma base para inferir todas as crenças de um conjunto. B_k é uma base para um conjunto de crenças K se e somente se B_k é um subconjunto finito de K e o fecho lógico de B_k é igual ao conjunto K. Logicamente podemos ter várias bases para um mesmo conjunto K. Do ponto de vista computacional, por trabalhar com conjuntos finitos, essa representação tem sido bastante utilizada.

* Mundos Possíveis: essa representação baseia-se na idéia de um conjunto W_k de mundos possíveis. Há uma forte relação entre conjunto de crenças e a representação por mundos possíveis. Para um conjunto W_k de mundos possíveis podemos definir um conjunto de crenças K correspondente como o conjunto das fórmulas que são verdadeiras em todos os mundos de W_k. Da mesma forma, qualquer conjunto de crenças K pode ser representado pelo subconjunto W_k dos mundos possíveis em que todas as sentenças de K são verdadeiras. Essa representação é muito utilizada por lógicos e filósofos e começa a ser usada computacionalmente.

2.4 Postulados AGM

Vamos assumir, a partir deste ponto, que a representação dos estados epistêmicos será baseada no modelo de conjunto de crenças. A motivação por trás dos postulados é que o Princípio da Mudança Mínima seja atendido quando uma mudança for realizada.

2.4.1 Expansão

Dado um conjunto de crenças K e uma sentença (crença) α, a expansão é definida por $K + \alpha = \mathrm{Cn}(K \cup \{\alpha\})$

2.4.2 Revisão

Dado um conjunto de crenças K e uma sentença (crença) α, os seis postulados básicos da revisão são:

R $*$ **1** : $K * \alpha$ é um conjunto de crenças
R $*$ **2** : $\alpha \in K * \alpha$
R $*$ **3** : $K * \alpha \subseteq K + \alpha$
R $*$ **4** : Se $\neg\alpha \notin K$, então $K + \alpha \subseteq K * \alpha$
R $*$ **5** : $K * \alpha$ é inconsistente sse $\vdash \neg\alpha$
R $*$ **6** : Se $\vdash \alpha \leftrightarrow \beta$, então $K * \alpha = K * \beta$

O postulado 1 mostra que o resultado de uma revisão continua sendo um conjunto de crenças. O postulado 2 diz que a crença recebida deve pertencer ao conjunto de crenças revisado. O próximo postulado garante que nenhuma outra informação é adicionada ao conjunto de crenças a não ser a nova sentença e suas conseqüências lógicas. O postulado 4 diz que se a nova informação adicionada for consistente, a expansão está contida na revisão. Esse fato, junto com o postulado 3, mostra que a expansão é igual a revisão. O postulado 5 diz que o resultado da revisão só é inconsistente se a fórmula usada para a revisão for inconsistente. Esse é o único caso onde a revisão aplicada a um conjunto consistente leva a um inconsistente. O postulado 6 mostra que a revisão feita por sentenças logicamente equivalentes resultam no mesmo conjunto revisado. Há ainda dois postulados extras para revisões por conjunções:

R $*$ **7** : $K * (\alpha \wedge \beta) \subseteq (K * \alpha) + \beta$
R $*$ **8** : Se $\neg\beta \notin K * \alpha$, então $(K * \alpha) + \beta \subseteq K * (\alpha \wedge \beta)$

O postulado 7 diz que a revisão por uma conjunção deve estar contida no conjunto resultante da revisão de uma das fórmulas pela expansão da outra. O postulado 8 afirma que se uma das fórmulas for consistente com o conjunto revisado pela outra fórmula, então, junto com o postulado 7, o conjunto revisado pela conjunção é igual ao conjunto resultante da revisão de uma das fórmulas pela expansão da outra.

2.4.3 Contração

Dado um conjunto de crenças K e uma sentença (crença) α, os seis postulados básicos da contração são:

C - 1 : $K - \alpha$ é um conjunto de crenças
C - 2 : $K - \alpha \in K$
C - 3 : Se $\alpha \notin K$, então $K - \alpha = K$
C - 4 : Se $\nvdash \alpha$, então $\alpha \notin K - \alpha$
C - 5 : $K \subseteq (K - \alpha) + \alpha$
C - 6 : Se $\vdash \alpha \leftrightarrow \beta$, então $K - \alpha = K - \beta$

O postulado 1 mostra que o resultado de uma contração continua sendo um conjunto de crenças. O postulado 2 diz que a operação de contração não acrescenta novas informações ao conjunto de crenças original. O próximo postulado garante que se a sentença contraída já não pertencia ao conjunto de crenças, não haverá mudanças. O postulado 4 diz que a sentença contraída não fará parte do conjunto resultante. O postulado 5 diz que o conjunto original pode ser recuperado ao fazer a expansão pela sentença contraída. Esse é um dos mais polêmicos postulados. Por exemplo, se tivermos um $K = \mathrm{Cn}(p, q, q \rightarrow p)$, intuitivamente, gostaríamos de poder definir $K - p = \mathrm{Cn}(q \rightarrow p)$ e $(K - p) + p = \mathrm{Cn}(p, q \rightarrow p)$. Mas isso não é permitido pelo postulado. Já o postulado 6 mostra que contrações feitas por sentenças logicamente equivalentes resultam no mesmo conjunto. Há ainda dois postulados extras para contração também relacionados a conjunções:

C - 7 : $K - \alpha \cap K - \beta \subseteq K - (\alpha \wedge \beta)$
C - 8 : Se $\alpha \notin K - (\alpha \wedge \beta)$, então $K - (\alpha \wedge \beta) \subseteq K - \alpha$

O postulado 7 garante que as crenças comuns à contração individual de cada uma das fórmulas da conjunção estão na contração pela conjunção. O postulado 8, juntamente com o postulado 7, diz que a contração feita por uma conjunção é igual a contração feita por uma das fórmulas dessa conjunção.

2.5 Revisão versus Contração

Harper e Levi identificaram e provaram duas relações entre os operadores de revisão e contração, que depois se tornaram conhecidas como identidade de Harper e identidade de Levi:

Identidade de Levi : $K * \alpha = (K - \neg\alpha) + \alpha$

Identidade de Harper : $K - \alpha = K \cap (K * \neg\alpha)$

2.6 Construções

Agora que já vimos um exemplo ilustrativo, os operadores de mudanças, a representação das crenças e os postulados AGM, é hora de vermos as construções. Como uma revisão nada mais é do que uma contração seguida de uma expansão (Identidade de Levi), a solução dos problemas se baseia na construção de uma função de contração que siga os postulados AGM.

2.6.1 Partial Meet Contraction

Antes de analisarmos essa primeira construção [4] é preciso definir a idéia de subconjunto maximal que não implica uma dada sentença. Seja K um conjunto de crenças e α, temos que X é um subconjunto maximal de K que não implica α sse:

* $X \subseteq K$

* $X \nvdash \alpha$

* Para todo X' tal que $X \subset X' \subseteq K$, $X' \vdash \alpha$

Note que há a possibilidade de existirem vários subconjuntos maximais que se adequam às regras acima. Portanto, é preciso definir uma função de seleção que escolhe quais desses subconjuntos maximais de K que não implicam α farão parte do processo. A contração se dá pela intersecção dos subconjuntos escolhidos por essa função. Damos o nome de partial meet contraction para essa construção.

Há dois casos que são opostos e muito conhecidos em se tratando da escolha da função de seleção: Maxichoice e Full Meet. No Maxichoice, a função de seleção tem como saída um único subconjunto maximal. A operação de contração sob essa função satisfaz o postulado:

Se $\beta \in K$ e $\beta \notin K - \alpha$, então $\beta \rightarrow \alpha \in K - \alpha$

Entretanto, esse postulado produz efeitos indesejados, pois cria subconjuntos muito grandes, com acréscimo de crenças que antes da contração eram totalmente desconhecidas. Já no Full Meet, a função de seleção tem como saída todos os subconjuntos maximais. A operação de contração sob essa função satisfaz o seguinte postulado:

Para todo α e β, $K - (\alpha \wedge \beta) = K - \alpha \cap K - \beta$

Mas esse postulado possui também efeitos indesejáveis que são opostos ao problema do Maxichoice. Nesse caso, são criados subconjuntos pequenos, que abandonam crenças que intuitivamente deveriam ter sido preservadas. É importante ressaltar aqui que a função de seleção até aqui apresentada sem nenhuma restrição satisfaz apenas os postulados básicos. Para satisfazer também os outros dois postulados extras é preciso que a função faça a seleção baseando-se em uma relação de ordem transitiva entre os subconjuntos maximais.

Vamos a um exemplo. Suponha um $K = \mathrm{Cn}(p, q)$ e desejamos contraí-lo por p usando essa construção. O primeiro passo é encontrar o conjunto $K \perp p$ com todos os subconjuntos maximais de K que não implicam p:

$$K \perp p = \{\{q, p \vee q, p \rightarrow q\}, \{p \rightarrow q, q \rightarrow p, p \leftrightarrow q\}\}$$

Além do conjunto $K \perp p$ precisamos definir um critério para a função de seleção. Para a função satisfazer os postulados **C - 7** e **C - 8** é preciso estar baseada em uma relação transitiva. Usaremos duas relações:

- Pelo menor número de fórmulas que contém p:

  A função de seleção devolve o primeiro subconjunto maximal, pois contém duas fórmulas em que p aparece, enquanto o segundo contém três. Como um único conjunto foi escolhido, a intersecção devolve o próprio. Portanto, $K - p = \{q, p \vee q, p \rightarrow q\}$. Neste caso a contração foi do tipo Maxichoice, já que um único subconjunto maximal foi selecionado pela função de seleção.

- Pelo número de fórmulas:

  Como os dois subconjuntos maximais possuem a mesma quantidade de fórmulas, ambos são escolhidos. A intersecção entre os dois é o resultado da contração. Portanto, $K - p = \{p \rightarrow q\}$. A função de seleção atuou como Full Meet ao selecionar todos os elementos do conjunto $K \perp p$.

2.6.2 Epistemic Entrenchment

Essa construção [5] é baseada na idéia de que nem todas as crenças possuem a mesma importância. É feito, portanto, uma espécie de classificação de crenças baseado em algum critério. A idéia é que as crenças em posições mais baixas no ranking (ou traduzindo ao pé da letra, menos entricheiradas) sejam abandonadas no momento da contração. Dado um conjunto de crenças K, e sendo $\alpha \leq \beta$ significando que β possui pelo menos a mesma posição de α no ranking, para que suas crenças satisfaçam a idéia do epistemic entrenchment, é preciso seguir os seguintes postulados:

EE1 : Se $\alpha \leq \beta$ e $\beta \leq \lambda$ então $\alpha \leq \lambda$
EE2 : Se $\alpha \vdash \beta$ então $\alpha \leq \beta$
EE3 : Para algum α e β, $\alpha \leq \alpha \wedge \beta$ ou $\beta \leq \alpha \wedge \beta$
EE4 : Quando $K \neq \bot$, $\alpha \notin K$ sse $\alpha \leq \beta$, para qualquer β
EE5 : Se $\alpha \leq \beta$ para qualquer α, então $\vdash \beta$

O postulado 1 diz que a ordenação deve ser transitiva. O postulado 2 diz que se uma sentença α implica alguma sentença β é melhor abandonar α e reter β. À primeira vista, esse postulado parece violar o Princípio da Mudança Mínima, porém analisando bem, se abandonarmos β o mesmo poderia ser refeito com α, o que não ocorre se decidirmos abandonar apenas a sentença α. O próximo postulado diz que se quisermos contrair $\alpha \wedge \beta$, basta contrairmos α ou β. O postulado 4 diz que qualquer sentença não pertencente ao conjunto de crenças terá o valor mínimo na ordenação. Já o postulado 5 diz que as tautologias sempre terão valor máximo na ordenação.

A contração é definida a partir da relação $\leq$ segundo a seguinte regra:

$\beta \in K - \alpha \Leftrightarrow \beta \in K$ e $(\alpha < (\alpha \vee \beta)$ ou $\vdash \alpha)$

Para elucidar melhor essa construção, voltemos ao mesmo exemplo da seção anterior. No epistemic entrenchment deve haver uma relação de ordem entre as sentenças do conjunto K que obedece aos cinco postulados citados anteriormente. Podemos definir essa relação de acordo com a figura 2.1, com $\leq$ de cima para baixo e as fórmulas no mesmo nível sendo equivalentes.

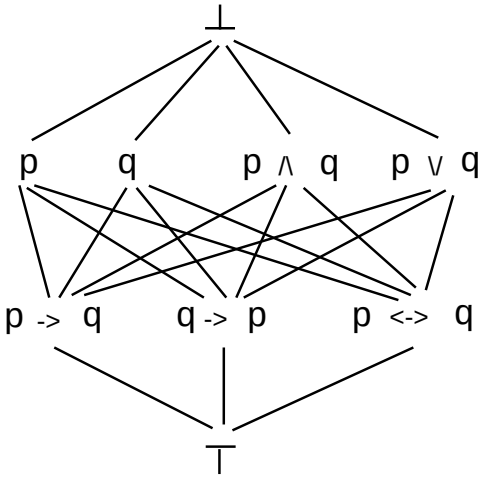

Figura 2.1: Relação de ordem

Baseado na relação anterior apresentada e na avaliação da regra de contração, temos que $K - p = \{p \rightarrow q, q \rightarrow p, p \leftrightarrow q\}$.

Essa construção foi criticada pois há a possibilidade da crença abandonada não ser a menor do ranking [6]. Uma outra crítica a essa construção, que pode ser estendida a toda a teoria dos postulados AGM, é que não há suporte a uma função iterativa, já que a ordenação é perdida logo após a primeira operação de contração.

2.6.3 Contração Segura

Antes de falarmos dessa construção [7] é preciso definirmos o que é um subconjunto minimal que implica uma dada sentença. Seja K um conjunto de crenças e α uma sentença, temos que X é um subconjunto minimal de K que implica α sse:

* $X \subseteq K$

* $X \vdash \alpha$

* Para todo X' tal que $X' \subset X \subseteq K$, $X' \nvdash \alpha$

A idéia por trás dessa construção é a seguinte: seja K um conjunto de crenças que se deseja contrair com uma sentença α e que exista uma ordem acíclica entre as sentenças de K, temos que um elemento β é seguro em relação a α se β não é o elemento mínimo (em relação à ordenação) de qualquer subconjunto minimal K' de K que implica α. Ou de outra forma, β é seguro se todo subconjunto minimal K' de K que implica α, ou não contém β ou contém algum $\lambda < \beta$.

Intuitivamente a idéia é que β é seguro se nunca puder ser responsável pela implicação de α. Note que, em contraste com as outras construções citadas, essa construção usa a idéia de subconjuntos minimais que implicam α, ao invés de subconjuntos maximais que não implicam α. A ordenação também pode ser vista como uma espécie de epistemic entrenchment. A contração segura se dá pelo conjunto de todas as sentenças que são seguras em relação a sentença que se deseja contrair.

Se voltarmos ao nosso exemplo, é preciso calcularmos o conjunto $K \asymp p$ que contém os subconjuntos minimais de K que implicam p:

$$K \asymp p = \{\{p\}, \{q, q \rightarrow p\}, \{p \wedge q\}, \{q, p \leftrightarrow q\}, \{q \rightarrow p, p \vee q\}, \{q \leftrightarrow p, p \vee q\}\}$$

Se não tivermos nenhuma ordem entre as sentenças de K, consideramos todos os elementos dos subconjuntos de $K \asymp p$ como "inseguros" em relação a p. Portanto, a única sentença segura é "$p \rightarrow q$", que é o resultado da contração.

2.6.4 Sistema de Esferas

Até o momento, as construções apresentadas se utilizavam da representação de crenças baseada em conjunto de crenças. Essas construções podem ser facilmente estendidas

para a representação das crenças através de bases. O sistema de esferas se utiliza da representação baseada em mundos possíveis.

Um sistema de esferas [8] centrado em $[K]$ é uma coleção X dos subconjuntos dos mundos possíveis M que satisfaz as seguintes condições:

- X é totalmente ordenado
- $[K]$ é o menor elemento dessa ordenação
- M é o maior elemento dessa ordenação
- Se qualquer esfera intercepta uma sentença α, então existe uma esfera em X que é a menor esfera que intercepta α

A contração através de uma dada sentença α nesse caso é feita juntando-se o $[K]$ com a intersecção de $[\neg\alpha]$ com a menor esfera de X dos mundos possíveis que tem intersecção não vazia com $[\neg\alpha]$. Já na revisão por α, os "melhores" mundos possíveis são aqueles que estão mais próximos dos mundos da esfera central, ou seja, a intersecção de $[\alpha]$ com a menor esfera. Essa construção é útil para a revisão de crenças baseada em comparação de modelos (estados, mundos possíveis).

No nosso pequeno exemplo, como estamos lidando com lógica proposicional, os mundos possíveis consistem nos possíveis valores que as variáveis podem tomar. Assim, temos quatro mundos possíveis:

W1 : p e q são verdadeiros
W2 : p é verdadeiro e q é falso
W3 : p é falso e q verdadeiro
W4 : p e q são falsos

O conjunto dos mundos possíveis de K, denotado por $[K]$, se resume ao W1, pois é o único mundo onde p e q são verdadeiros. É necessário estabelecer uma ordem para o sistema de esferas centrado em $[K]$. No segundo nível estão os mundos onde uma variável muda o seu valor (W2 e W3) e no terceiro nível estão os demais mundos (W4).

Como a contração é a união do $[K]$ com a intersecção de $[\neg\alpha]$ com a menor esfera do sistema, temos que a contração $K - p = (W1 \cup W3) = \{q, p \vee q, p \rightarrow q\}$. Podemos visualizar na figura 2.2 a contração no sistema de esferas centrado em $[K]$.

A proposta que iremos abordar é baseada nessa construção. No capítulo 4 entraremos em detalhes sobre o motivo de a escolhermos.

Note que no exemplo, com exceção do epistemic entrenchment, há coincidências nos conjuntos contraídos. O sistema de esferas devolve como resultado o mesmo conjunto contraído usando o Maxichoice e a contração segura devolve um conjunto idêntico ao

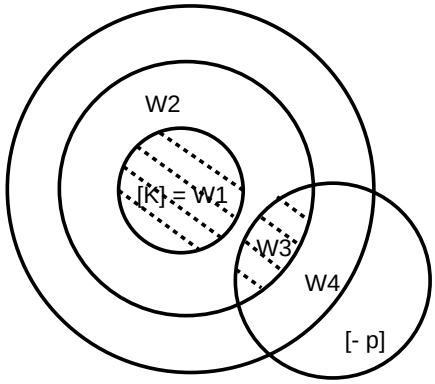

Figura 2.2: Contração no sistema de esferas centrado em $[K]$

Full Meet. Todas as construções possuem provas de equivalência e provas de que seguem os postulados AGM.

Ainda há outras construções possíveis, mas essas são as quatro mais famosas.

Capítulo 3

Verificação Automática de Modelos

Antes de entramos em detalhes sobre as idéias por trás da verificação automática de modelos é preciso expor o porquê de a escolhermos para a nossa solução. Os nossos critérios foram: método formal conhecido; ferramenta com código aberto; ferramenta com detecção automática de inconsistências. Uma vez que alguns dos mais conhecidos métodos formais de desenvolvimento de software, como B, Z, State Charts, ASM e VDM não possuem nenhuma ferramenta de detecção automática de inconsistências que possua código aberto e que já há alguns estudos que os traduzem em um problema de verificação automática de modelos [9, 10, 11], a nossa escolha foi definida.

A verificação automática de modelos [12, 13] começou a ser desenvolvida no início dos anos 80 e atualmente é uma das mais conhecidas e utilizadas técnicas de verificação de sistemas que possam ser modelados como máquinas de estados finitos. O princípio básico dessa técnica é: dada uma propriedade do sistema, descrita em alguma lógica temporal, determina-se se a máquina de estados finitos que representa o sistema descrito satisfaz tal propriedade. Em outras palavras, verifica-se se uma fórmula f é verdadeira em um grafo G de estados.

3.1 Estrutura de Kripke

A máquina de estados finitos que representa o sistema pode ser descrita como uma estrutura de Kripke específica, que é definida como: um conjunto S de estados, um conjunto R de transições entre estados (onde cada estado deve ter um sucessor), um conjunto I de estados iniciais e uma função L que associa a cada estado um conjunto de proposições verdadeiras naquele estado. Para modelar um estado em *deadlock* (estado que não possui sucessores) basta criar uma transição que aponte para si mesmo.

Estrutura de Kripke [13] Seja P um conjunto finito de proposições booleanas. Uma estrutura de Kripke sobre P é uma quádrupla $K = (S, R, I, L)$ onde:

* S é um conjunto de estados (quando S é finito, então K é uma estrutura de Kripke finita);

* $R \subseteq S \times S$ é uma relação de transição, tal que $\forall s \in S \bullet \exists s' \in S, (s, s') \in R$;

* $I \subseteq S$ é o conjunto de estados iniciais;

* $L : S \to 2^P$ é uma função que para cada estado, associa um conjunto de proposições booleanas verdadeiras naquele estado.

A figura 3.1 representa uma estrutura de Kripke onde $P = \{a, b, c\}$, $S = \{s0, s1, s2\}$, $R = \{(s0, s1), (s0, s2), (s1, s0), (s1, s2), (s2, s2)\}$, $I = \{s0\}$ com $L(s0) = \{a, b\}$,$L(s1) = \{b, c\}$ e $L(s2) = \{c\}$.

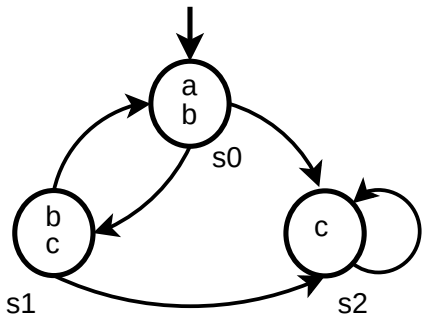

Figura 3.1: Estrutura de Kripke

Em termos de implementação, a estrutura de Kripke pode ser representada explicitamente ou através de BDDs (Binary Decision Diagrams).

3.2 Diagramas de Decisão Binária

Seja um conjunto $B = \{0, 1\}$, uma função booleana f é uma função sobre valores booleanos para um valor booleano:

$f : B^n \to B$

Considere um conjunto de variáveis booleanas $Y = \{y_1, ..., y_n\}$ e $f(y_1, ..., y_n)$ uma função booleana sobre Y com uma ordem fixa de seus argumentos. A função resultante quando um argumento y_i da função f é substituído por um valor $b \in B$ é chamada de cofator de f (também conhecido como restrição de f) e denotada por:

$f|_{y_i=b} = f(y_1, ..., y_{i-1}, b, y_{i+1}, y_n)$

Com essa notação, utilizamos a expansão de Shannon abaixo para expressar a função booleana sobre uma variável y_i:

$$f = (y_i \wedge f|_{y_i=1}) \vee (\overline{y_i} \wedge f|_{y_i=0})$$

O resultado de uma função f pode então ser encontrado através da aplicação sucessiva da expansão de Shannon, substituindo as variáveis pelos seus respectivos valores assumidos na função, tornando a computação do resultado mais fácil.

Um diagrama de decisão binária (BDD) [14] é um grafo acíclico dirigido com vértices não terminais, arcos e dois vértices terminais que permite representar e manipular funções booleanas. Os vértices não terminais possuem variáveis binárias e os arcos que saem desses vértices são rotulados com 0 ou 1. Um vértice terminal é rotulado com 0 e o outro com 1, representando os valores booleanos falso e verdadeiro.

Como cada vértice não terminal representa uma variável da função, para descobrir o valor para uma determinada entrada da função, basta percorrer o grafo a partir da raiz até um terminal. Os valores das variáveis binárias definem um caminho no grafo que pode levar ao terminal rotulado com 1 ou ao terminal rotulado com 0. Ao longo de um caminho, se uma variável de controle vale 0, deve-se seguir o arco rotulado com 0 e, se vale 1, o arco rotulado com 1. Daí a expressão "decisão binária", pois cada nó não terminal só tem dois sucessores possíveis. O valor da variável binária do nó indica, portanto, qual caminho seguir. Quando percorremos o grafo dessa maneira o que se faz na verdade é aplicar sucessivamente a expansão de Shannon para descobrir qual o valor da função booleana. Veja na figura 3.2 como $f : a \wedge b$ é representada como um BDD :

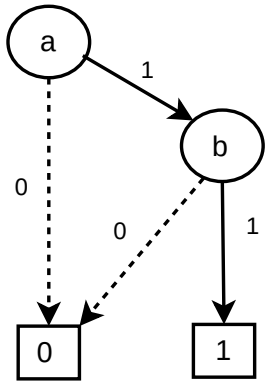

Figura 3.2: BDD de f : a $\wedge$ b

Para descobrirmos o valor de $f(1,0)$, com $a = 1$ e $b = 0$, basta percorrermos o BDD de cima para baixo, seguindo as seguintes equações:

$f(1,0) : f = (a \wedge f|_{a=1}) \vee (\overline{a} \wedge f|_{a=0})$
$f(1,0) : f = (1 \wedge f|_{a=1}) \vee (0 \wedge f|_{a=0})$
$f(1,0) : f = (1 \wedge f|_{a=1})$
$f(1,0) : f|_{a=1} = b$
$f(1,0) : f = (b \wedge f|_{b=1}) \vee (\overline{b} \wedge f|_{b=0})$
$f(1,0) : f = (0 \wedge f|_{b=1}) \vee (1 \wedge f|_{b=0})$
$f(1,0) : f|_{b=0} = 0$

Logo, o valor da função para a = 1 e b = 0 é igual a 0.

Podemos comparar um BDD a uma árvore de decisão binária com a diferença de que o primeiro é um grafo dirigido enquanto o segundo é uma árvore. Além disso, em um BDD existe uma ordem total na ocorrência das variáveis da raiz até as folhas. Bryant [14] mostrou que, para uma dada função booleana e uma dada ordem total de suas variáveis, existe um único BDD. Essa ordem das variáveis é um dos fatores críticos do uso de BDDs, pois o seu tamanho (medido em número de vértices do grafo) varia de uma maneira drástica de acordo com a ordem escolhida. Bryant também descreveu algoritmos eficientes para operações básicas de BDDs, tais como encontrar os BDDs que representam $\neg a$ (BDDNot) e $a \wedge b$ (BDDAnd) dado os BDDs que representam as funções a e b. Além disso, ele também descreveu algoritmos que computam o BDD do cofator de uma função $f(f|_{y_i=b})$, permitindo operações de quantificação e substituição das variáveis booleanas.

3.3 Estrutura de Kripke como BDDs

Burch et al [15] mostraram que um problema de verificação de modelos pode ser simbolicamente representado através de BDDs, aumentando o número de estados da estrutura de Kripke armazenáveis computacionalmente de 10^8 (na representação explícita) para 10^{20}. Com isso os verificadores atuais tendem a utilizar a representação da estrutura de Kripke através de BDDs.

Para que possamos definir um mapeamento entre estruturas de Kripke e BDDs basta descobrirmos uma maneira de expressarmos uma estrutura de Kripke através de funções booleanas. Essas funções chamaremos de funções características.

Cada estado é representado através de uma função característica. Seja $M = (S, R, I, L)$ uma estrutura de Kripke sobre o conjunto de proposições booleanas $P = \{x_i, ..., x_n\}$, a função característica de um estado s ∈ S, denotada por f_s, é definida como:

$$f_s(x_1, ..., x_n) = \left(\left(\bigwedge_{x_i \in L(s)} x_i \right) \wedge \left(\bigwedge_{x_i \notin L(s)} \neg x_i \right) \right)$$

As funções características podem ser extendidas para conjuntos de estados a partir das seguintes definições:

$$f_{\{\}}(x_1, ..., x_n) = 0$$
$$f_{\{x\} \cup X}(x_1, ..., x_n) = f_{\{x\}}(x_1, ..., x_n) \vee f_X(x_1, ..., x_n)$$

Assim, é possível expressar estados e conjuntos de estados através de suas funções características.

Essas definições se limitam a representar os estados separadamente. Para que possamos expressar as transições entre estados, utilizaremos um conjunto $P' = \{x'_1, ..., x'_n\}$, que é uma cópia das variáveis de estado e que expressa o próximo estado. A função característica de uma transição $t = (s_1, s_2) \in R$, denotada por f_t, é definida como:

$$f_t(x_1, ..., x_n, x'_1, ..., x'_n) = f_{s_1}(x_1, ..., x_n) \wedge f_{s_2}(x'_1, ..., x'_n)$$

Similarmente à representação de estados, é possível extender tal definição para representar conjuntos de transições.

Seja a estrutura de Kripke da figura 3.1, então o estado $s0$ pode ser representado pela função $f_s : (a \wedge b \wedge \neg c)$ e a transição $(s0, s2)$ pode ser representada pela função $f_t : (a \wedge b \wedge \neg c) \wedge (\neg a' \wedge \neg b' \wedge c')$.

3.4 Lógica Temporal

As propriedades do sistema são descritas em lógica temporal. A lógica temporal é um tipo de lógica modal onde é possível representar e raciocinar sobre proposições relacionadas ao tempo. Através da lógica temporal é possível expressar frases do tipo "Eu *SEMPRE* estou com fome", "Eu *FUTURAMENTE* ficarei com fome" ou "Eu ficarei com fome *ATÉ* comer em um rodízio".

As duas lógicas temporais mais utilizadas para verificação de sistemas são a LTL (Linear Temporal Logic) [16] e a CTL (Computation Tree Logic) [17]. A primeira se relaciona com os caminhos e a segunda com os estados. Outra maneira de ver a diferença é que a CTL explicitamente introduz quantificadores sobre os caminhos, enquanto na LTL o quantificador universal está implícito. Apesar de poder expressar propriedades que não são possíveis usando CTL, muitas fórmulas descritas em LTL podem ser convertidas em um problema a ser verificado com aquela lógica [18]. Portanto, vamos limitar o nosso estudo à lógica CTL. A seguir apresentaremos a sua sintaxe e semântica, bem como o algoritmo utilizado para verificação das fórmulas descritas sob essa lógica.

3.4.1 Sintaxe da CTL

A lógica temporal CTL possui a seguinte sintaxe [13] :

$$\phi ::= p|\neg\phi|\phi \wedge \phi|(AX\phi)|(AG\phi)|(AF\phi)|(EX\phi)|(EG\phi)|(EF\phi)|E(\phi\ U\phi)|A(\phi\ U\phi)$$

onde p é um átomo proposicional, $\neg$, $\wedge$ são operadores lógicos usuais e os demais são operadores modais de tempo. Cada operador temporal é composto por um quantificador de caminho (E, para algum caminho, ou A, para todos os caminhos) seguido por um quantificador de estado (X, próximo estado no caminho, U, até, G, globalmente, ou F, futuramente).

3.4.2 Semântica da CTL

A lógica temporal CTL possui a seguinte semântica [13] :

Seja M uma estrutura de Kripke e $\pi(i)$ o i-ésimo estado s_i de um caminho, então dizemos que se $M, s \models \phi$, então ϕ é verdadeiro no estado s de M. Assim temos :

1. $M, s \models p$ sse $p \in L(s_0)$
2. $M, s \models \neg\phi$ sse $M, s \not\models \phi$
3. $M, s \models \phi_1 \wedge \phi_2$ sse $M, s \models \phi_1$ e $M, s \models \phi_2$
4. $M, s \models$ EX ϕ sse existe um estado s' de M tal que $(s, s') \in R$ e $M, s' \models \phi$
5. $M, s \models$ EG ϕ sse existe um caminho π de M tal que $\pi(1) = s$ e $\forall i \geq 1 \bullet M,\pi(i) \models \phi$
6. $M, s \models$ E(ϕ_1 U ϕ_2) sse existe um caminho π de M tal que $\pi(1) = s$ e $\exists i \geq 1 \bullet (M, \pi(i) \models \phi_2 \wedge \forall j, i > j \geq 1 \bullet M,\pi(j) \models \phi_1)$

No itens 1 a 3 temos as propriedades usuais da lógica proposicional. No item 4 temos que ϕ é verdadeiro em algum próximo estado. O item 5 diz que existe algum caminho onde ϕ é verdadeiro em todos os estados, enquanto no item 6 existe um caminho no qual ϕ_1 é verdadeiro até ϕ_2 se tornar verdadeiro (nada pode ser dito a respeito de ϕ_1 quando ϕ_2 se torna verdadeiro). Os demais operadores modais podem ser derivados a partir desses três :

AX ϕ = $\neg$ EX $\neg\phi$
AG ϕ = $\neg$ EF $\neg\phi$
AF ϕ = $\neg$ EG $\neg\phi$
EF ϕ = E [true U ϕ]

$$A[\phi \text{ U } \beta] = \neg \text{ E}[\neg\beta \text{ U } \neg\phi \wedge \neg\beta] \wedge \neg \text{ EG } \neg\beta$$

Nós dizemos que uma estrutura de Kripke M satisfaz uma fórmula CTL ϕ se $M, s_0 \models \phi$, onde s_0 é um estado inicial.

Visualmente temos na figura 3.3 abaixo a representação dos operadores temporais da CTL.

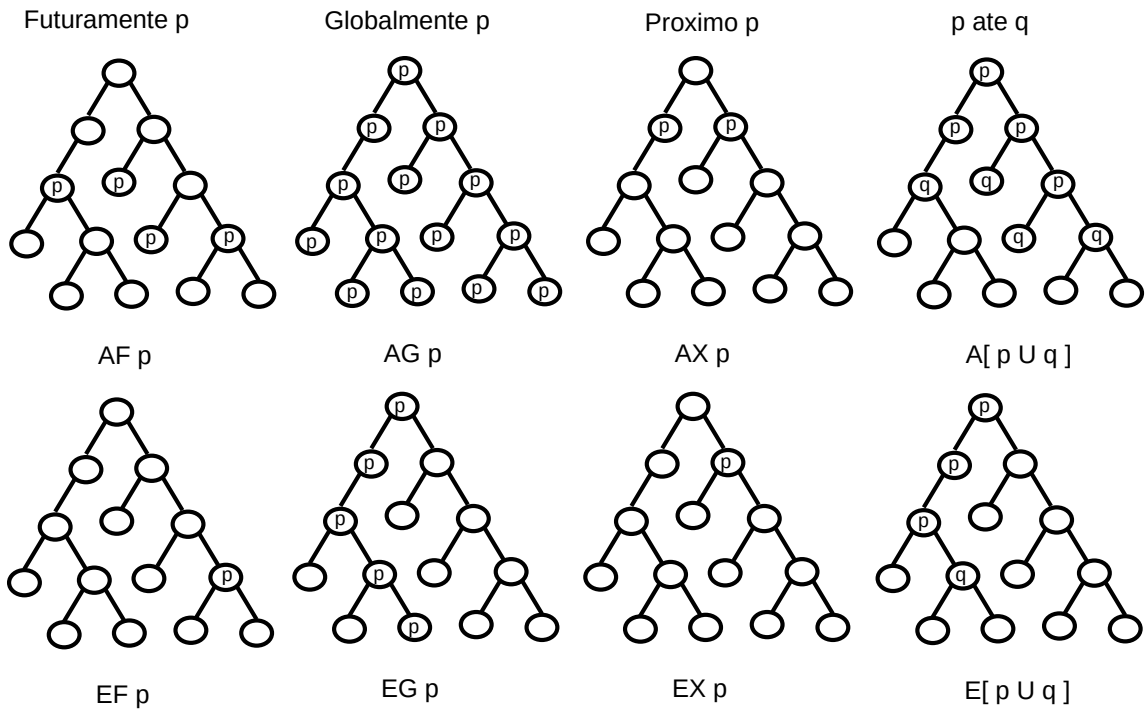

Figura 3.3: Operadores temporais da CTL

Na figura 3.3 fica claro a idéia de cada operador temporal. No AF p, para todos os caminhos a partir da raiz, p é verdadeiro futuramente. No AG p, em todos os caminhos a partir da raiz, p é verdadeiro globalmente, ou seja, p é verdadeiro em todos os estados. No AX p, em todos os caminhos partindo da raiz, o próximo estado possui p verdadeiro. Em A[p U q], p é verdadeiro em todos os caminhos a partir da raiz até q ser verdadeiro. De forma similar podemos fazer a mesma leitura da figura com o quantificador existencial.

Note que a figura 3.3 é meramente ilustrativa, pois na prática os caminhos possuem tamanho infinito. Isso acontece porque na estrutura de Kripke definida anteriormente todo estado possui uma transição, mesmo aqueles que estão em *deadlock*. Veja na figura 3.4 a interpretação em termos de caminhos da estrutura de Kripke da figura 3.1.

Nessa interpretação podemos visualmente constatar que as fórmulas EG b e E [b U c] são verdadeiras a partir do estado inicial s0.

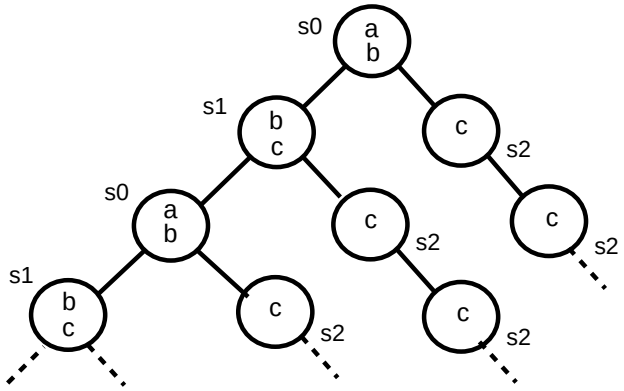

Figura 3.4: Interpretação em caminhos da figura 3.1

3.4.3 Algoritmos de Verificação para CTL

A principal abordagem para a verificação automática de fórmulas na lógica CTL é baseada na teoria de ponto fixo [13] para caracterizar seus operadores.

Um reticulado completo nada mais é que um conjunto que possui uma ordem parcial em seus elementos, um maior elemento $\top$ (verdadeiro) e um menor elemento $\bot$ (falso). Seja S o conjunto dos estados de uma estrutura de Kripke finita, podemos formar um reticulado completo com os subconjuntos de S (2^S), utilizando o operador está contido ($\subseteq$) para a ordenação, o conjunto vazio ($\{\}$) como o menor elemento e o conjunto S como maior elemento. Como é possível representar cada subconjunto de S através de suas funções características, esse mesmo reticulado também pode ser representado usando a implicação booleana ($\rightarrow$) como ordenação, "falso" como menor elemento e a função característica de S como maior elemento. Um exemplo de reticulado dos subconjuntos dos estados da estrutura de Kripke da figura 3.1 pode ser visto na figura 3.5.

Um ponto fixo de uma função f é encontrado quando $f(x) = x$. O menor ponto fixo de uma função f (lfp(f)) é encontrado quando esta é aplicada sucessivamente a partir do menor elemento ($\bot$) do reticulado até encontrar o ponto fixo. O maior ponto fixo de f (gfp(f)) é encontrado quando f é aplicada sucessivamente a partir do maior elemento ($\top$) até encontrar o ponto fixo. Tarski demonstrou que se f é uma função monotônica (se A $\subseteq$ B, então f(A) $\subseteq$ f(B)), então existe um único menor ponto fixo e um único maior ponto fixo.

Pela definição, a fórmula EG ϕ é verdadeira em um estado $s \in S$, se :

- ϕ é verdadeira em s, e

- um sucessor de s satisfaz EG ϕ

Logo, EG $\phi \equiv (\phi \wedge \text{EX}(\text{EG } \phi))$.

Uma vez que essa função é monotônica, calcular EG ϕ corresponde a calcular o maior ponto fixo dessa função aplicada sob o reticulado já descrito [13]. Mais precisamente,

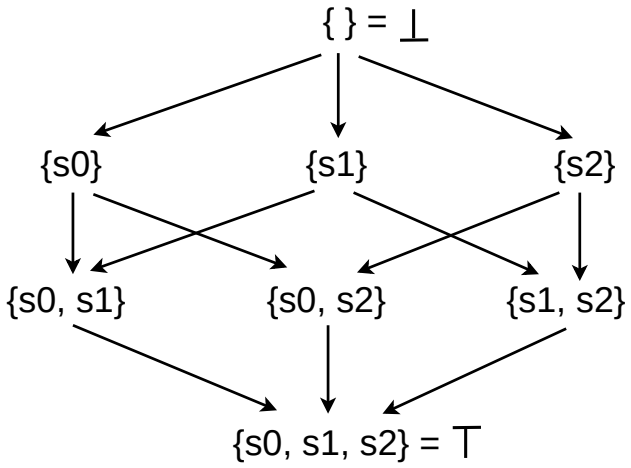

Figura 3.5: Reticulado dos subconjuntos de S ordenado por $\subseteq$

EG ϕ é o menor ponto fixo da função $f(z) : [[\phi]] \cap \{s \in S \mid \exists s' \in R(s) \cap Z\}$, onde $R(s) = \{s' \in S \mid (s, s') \in R\}$ e $[[\phi]] = \{s \in S \mid M, s \models \phi\}$. Podemos verificar que os estados $s0$ e $s1$ da figura 3.1 são EG b aplicando sucessivamente essa função a partir do maior elemento do reticulado.

$f(S) = [[b]] \cap S = \{s0, s1\}$
$f(\{s0, s1\}) = [[b]] \cap \{s0, s1\} = \{s0, s1\}$ (ponto fixo)

Pela definição, a fórmula E[ϕ U ψ] é verdadeira em um estado $s \in S$, se :

- ou ψ é verdadeira em s

- ou ϕ é verdadeira em s e um sucessor de s satisfaz E[ϕ U ψ]

Logo, E[ϕ U ψ] $\equiv (\psi \vee (\phi \wedge \text{EX}(\text{E}[\phi \text{ U } \psi])))$.

Uma vez que é uma função monotônica, calcular E[ϕ U ψ] corresponde a calcular o menor ponto fixo dessa função aplicada sob o reticulado descrito anteriormente [13]. Mais precisamente, EU [ϕ U ψ] é o menor ponto fixo da função $f(z) : [[\psi]] \cup ([[\phi]] \cap \{s \in S \mid \exists s' \in R(s) \cap Z\})$. Podemos verificar que todos os estados da figura 3.1 são E [b U c] aplicando sucessivamente essa função a partir do menor elemento do reticulado.

$f(\{\}) = [[c]] \cup \{\} = \{s1, s2\}$
$f(\{s1, s2\}) = [[c]] \cup ([[b]] \cap \{s0, s1, s2\}) = \{s0, s1, s2\}$
$f(\{s0, s1, s2\}) = [[c]] \cup ([[b]] \cap \{s0, s1, s2\}) = \{s0, s1, s2\}$ (ponto fixo)

Como os demais operadores temporais são derivados do EG, EU e EX, basta agora apenas calcularmos EX, uma tarefa fácil e eficiente quando usamos a representação baseada em BDDs.

A idéia principal do algoritmo é "etiquetar" cada estado $s \in S$ com as sub-fórmulas de ϕ que são verdadeiras em s. O algoritmo inicia com fórmulas de tamanho 1 (proposição atômicas, verdadeiro e falso). As demais iterações abordam fórmulas de tamanho $i+1$, e levam em consideração fórmulas já tratadas em iterações passadas. Por exemplo: para $\phi = \phi_1 \vee \phi_2$, s será etiquetado com ϕ caso s seja etiquetado com ϕ_1 ou com ϕ_2. O algoritmo termina quando ϕ é tratada. $M, s_0 \models \phi$ será verdadeiro se s_0 for etiquetado com ϕ.

Os pseudo-códigos do algoritmo principal **Label** e das demais funções utilizadas por ela estão descritas a seguir. O algoritmo vai desmembrando a fórmula temporal através de chamadas recursivas até se chegar a fórmulas de tamanho 1. Quando for necessário se calcular os estados que alcançam estados onde uma dada sub-fórmula é verdadeira, então é evocada a função **Pre**. Na função **Label_EG** a variável Y é iniciada com o maior elemento do reticulado, sendo atualizada por Label(F) ∩ Pre(Y) na linha 6 até se encontrar o ponto fixo (linha 4). Já na função **Label_EU**, a variável Y é iniciada com o menor elemento do reticulado, sendo atualizada por Label(G) ∪ (Label(F) ∩ Pre(Y)) na linha 6 até se encontrar o ponto fixo (linha 4). Note que Label(F) ∩ Pre(Y) corresponde a calcular $f(z) : [[\phi]] \cap \{s \in S \mid \exists s' \in R(s) \cap Z\}$, enquanto Label(G) ∪ (Label(F) ∩ Pre(Y)) corresponde a calcular $f(z) : [[\psi]] \cup ([[\phi]] \cap \{s \in S \mid \exists s' \in R(s) \cap Z\})$.

FUNCTION Label
Entrada: fórmula CTL F
Saída: estados onde a fórmula F é verdadeira

```
1: if F == true then
2:    return S; {conjunto de todos os estados}
3: else if F == false then
4:    return ∅; {conjunto vazio}
5: else if F == um átomo p then
6:    return estados onde p é verdadeiro
7: else if F == ¬ f1 then
8:    return S \ Label(f1);
9: else if F == f1 ∧ f2 then
10:   return Label(f1) ∩ Label(f2);
11: else if F == f1 ∨ f2 then
12:   return Label(f1) ∪ Label(f2);
13: else if F == EX f1 then
14:   return Pre(f1);
15: else if F == E[f1 U f2] then
16:   return Label_EU(f1,f2);
17: else if F == EG f1 then
18:   return Label_EG(f1);
19: end if
```

FUNCTION Pre
Entrada: fórmula CTL F
Saída: estados que alcançam estados onde a fórmula F é verdadeira

```
1: var X;
2: X ⇐ ∅;
3: for all s' ∈ Label(F) do
4:    for all s ∈ S do
5:       if <s,s'> ∈ R then
6:          X ⇐ X + {s};
7:       end if
8:    end for
9: end for
10: return  X;
```

FUNCTION Label_EG
Entrada: fórmula CTL F
Saída: estados onde a fórmula EG F é verdadeira

```
1: var X,Y;
2: Y ⇐ S;
3: X ⇐ Label(F) ∩ Pre(Y);
4: while X != Y do
5:    Y ⇐ X;
6:    X ⇐ Label(F) ∩ Pre(Y);
7: end while
8: return  X;
```

FUNCTION Label_EU
Entrada: fórmula CTL F e fórmula CTL G
Saída: estados onde a fórmula E[F U G] é verdadeira

```
1: var X,Y;
2: Y ⇐ ∅;
3: X ⇐ Label(G) ∪ (Label(F) ∩ Pre(Y));
4: while X != Y do
5:    Y ⇐ X;
6:    X ⇐ Label(G) ∪ (Label(F) ∩ Pre(Y));
7: end while
8: return  X;
```

A arquitetura do verificador de modelos está descrita na figura abaixo :

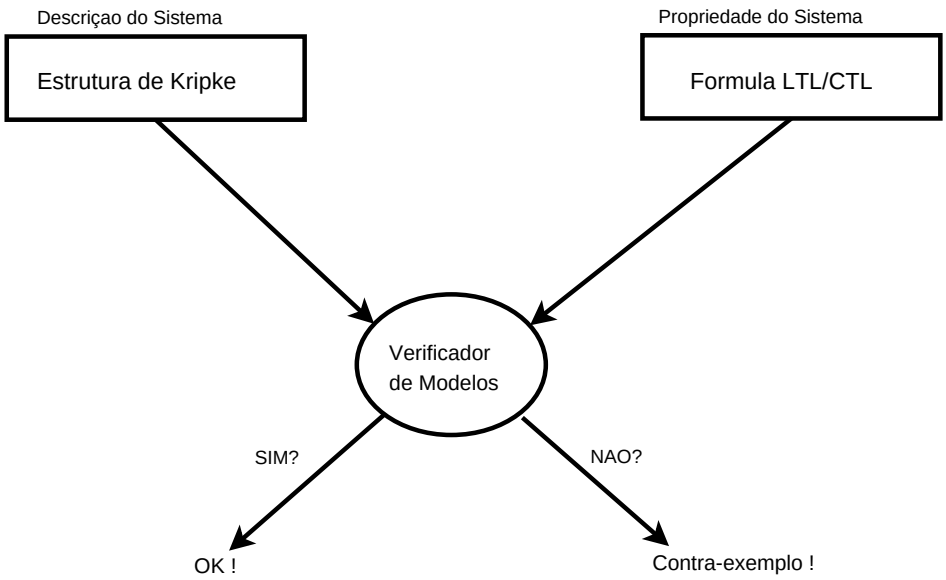

Figura 3.6: Arquitetura do Model Checking

3.5 NuSMV

NuSMV [19] é uma das mais famosas e usadas ferramentas de verificação formal de sistemas de estados finitos. Sua primeira versão foi desenvolvida no final da década de 90 conjuntamente pelo centro de pesquisa italiano ITC-IRST e pela Universidade de Carnegie-Mellon como uma reimplementação do verificador modelo de SMV criado por McMillan [20] durante seu PhD.

Essa ferramenta possui grandes vantagens em relação as outras, pois, além de possuir código livre (em C, com licença LGPL 2.1) e representação através de BDDs (evitando assim a explosão de estados), permite verificar sistemas de estados finitos a partir de especificações nas lógicas temporais CTL e LTL. Além disso, possui um algoritmo específico para a verificação de invariantes que mostra o menor caminho até o estado inconsistente. Atualmente está na versão 2.4 e possui uma lista de discussão bastante ativa. A grande desvantagem é que ainda não possui interface gráfica.

A linguagem do NuSMV foi projetada para permitir a descrição de sistemas tanto síncronos como assíncronos. Essa linguagem, baseada no antecessor SMV, permite também a modularização de componentes, facilitando assim a manutenção e a reutilização dos mesmos. O NuSMV se utiliza do cálculo proposicional para descrever as transições de uma estrutura finita de Kripke, fornecendo muita flexibilidade, mas ao mesmo tempo

podendo introduzir inconsistências. Como o NuSMV descreve máquinas de estados finitos, a sua linguagem só permite tipos de dados finitos, como booleanos, escalares e vetores de tipos de dados básicos.

3.5.1 Sintaxe

As principais construções da sintaxe do NuSMV são: Expressões Simples, Expressões Case, Expressões Set, Expressões Next e Expressões CTL. Com as expressões simples é possível descrever variáveis booleanas, variáveis numéricas, bem como operadores lógicos e numéricos no sistema. Já as expressões case geralmente são utilizadas para relacionar as pré-condições com as pós-condições. As expressões set são utilizadas para descrever a teoria dos conjuntos, como definição de conjunto, teste de inclusão e união de conjuntos. As expressões next, por sua vez, são utilizadas para expressar as transições relacionada a uma dada variável na máquina de estados finitos, sendo freqüentemente utilizada em conjunto com as expressões case. As expressões CTL são usadas para definir a propriedade a ser verificada.

A seguir apresentaremos um pequeno exemplo de um sistema de exclusão mútua (mutex), retirado da documentação oficial do NuSMV, a fim de elucidar melhor a sintaxe. Esse sistema é modelado com dois estados (representando processos) que estão, ou em uma região crítica (c1,c2), ou esperando a vez para entrar nessa região (t1,t2) ou em uma região não crítica (n1,n2). O sistema ainda possui uma variável de bloqueio (turn) que indica qual o estado tem a prioridade para entrar na região crítica. O usuário deseja verificar se em algum momento os dois estados entram na região crítica (EF((state1 = c1) & (state2 = c2))). Se essa propriedade for falsa, então a exclusão mútua está garantida.

```
MODULE main
VAR
state1: {n1, t1, c1}; state2: {n2, t2, c2}; turn: {1, 2};

ASSIGN
init(state1) := n1; init(state2) := n2; init(turn) := 1;

next(state1) := case
  (state1 = n1) & (state2 = t2): t1;
  (state1 = n1) & (state2 = n2): t1;
  (state1 = n1) & (state2 = c2): t1;
  (state1 = t1) & (state2 = n2): c1;
  (state1 = t1) & (state2 = t2) & (turn = 1):  c1;
  (state1 = c1): n1;
  1: state1;
esac;
```

```
next(state2) := case
  (state2 = n2) & (state1 = t1): t2;
  (state2 = n2) & (state1 = n1): t2;
  (state2 = n2) & (state1 = c1): t2;
  (state2 = t2) & (state1 = n1): c2;
  (state2 = t2) & (state1 = t1) & (turn = 2):  c2;
  (state2 = c2): n2;
  1: state2;
esac;
next(turn) := case
  (state1 = n1) & (state2 = t2): 2;
  (state2 = n2) & (state1 = t1): 1;
  1: turn;
esac;

SPEC
EF((state1 = c1) & (state2 = c2))
```

O espaço de estados da máquina de estados finitos é determinado pela declaração das variáveis de estado (no exemplo acima, state1, state2 e turn). A variável state1 é declarada como sendo do tipo escalar, assumindo simbolicamente os valores n1, t1 e c1. O mesmo acontece com as variáveis state2 e turn, que assumem os valores n2, t2, c2 e 1,2, respectivamente.

Com a declaração ASSIGN é possível definir atribuições às variáveis. A primeira atribuição se refere ao valor inicial das variáveis (init). No exemplo acima a variável state1 irá começar com valor n1 e a variável state2 com n2. Já a variável turn se inicia com valor 1. As atribuições definidas pela declaração next expressam a relação de transição das variáveis. No exemplo, temos que, no próximo estado, o valor da variável turn será 2 caso no estado atual a expressão (state1 = n1) & (state2 = t2) seja verdadeira. Se por outro lado, a expressão (state2 = n2) & (state1 = t1) for verdadeira no estado atual, o próximo valor da variável turn será 1. Se nenhuma das duas expressões forem verdadeiras, então a variável turn continua com o mesmo valor no próximo estado. Nesse exemplo é possível ver claramente como as expressões case atuam em conjunto com as expressões next e são usadas para relacionar as pré-condições com as pós condições (se (state1 = n1) & (state2 = t2) [pré-condição], então turn := 2 [pós-condição]).

Com a declaração SPEC ou INVARSPEC é possível ao usuário definir qual propriedade temporal se deseja verificar no sistema. No exemplo acima, o que se quer verificar é se existe um caminho que em um dos seus estados futuros a expressão (state1 = c1) & (state2 = c2) seja verdadeira. A declaração INVARSPEC é usada apenas para verificar invariantes (operador AG) no sistema. Na próxima seção explicaremos a diferença de funcionamento dessas duas declarações.

Essas construções da linguagem do NuSMV são as mais usuais e conhecidas. Há

ainda a possibilidade de expressar transições de uma outra maneira, modularizar a descrição do sistema, bem como modelá-lo como um sistema assíncrono. Mas como não os usaremos, não vamos detalhá-los. Para maiores detalhes da sintaxe do NuSMV que vamos utilizar em nossa proposta, consulte o apêndice A.

3.5.2 Verificação

O NuSMV verifica propriedades na lógica CTL utilizando a abordagem da teoria do ponto fixo através da declaração SPEC. A sua grande vantagem é que, ao fazer uso de BDDs, a computação se torna mais fácil e ágil. No NuSMV, cada relação de transição definida através da declaração next se torna um BDD, assim como cada estado.

A ferramenta utiliza tais estruturas para gerar um novo BDD *CTLStates* (através do algoritmo principal *Label*) com os estados onde a propriedade descrita na lógica CTL é verdadeira. A partir desse BDD gerado, pegamos o BDD que representa a sua negação ($\neg CTLStates$) aplicando a operação BDDNot descrita por Bryant. Esse novo BDD contém os estados onde a propriedade não é verdadeira.

Assim, aplicando a operação BDDAnd (também descrita por Bryant e que nesse caso funciona como uma intersecção entre conjuntos de estados) no BDD que representa os estados iniciais em conjunto com o BDD $\neg CTLStates$, podemos verificar se há intersecção ou não entre esses conjuntos de estados, ou seja, se os estados iniciais fazem ou não parte do conjunto dos estados onde a propriedade não é verdadeira. Se fizer parte desse conjunto, a ferramenta escolhe um estado qualquer do BDD gerado pela operação BDDAnd e traça um caminho do estado inicial até o mesmo, devolvendo ao usuário um contra-exemplo, que nem sempre é o de menor caminho.

O NuSMV possui também um algoritmo específico para verificação de uma propriedade de segurança conhecida como invariante. Um invariante de um sistema corresponde a uma propriedade que deve ser verdadeira em todos os estados do sistema, ou mais formalmente, dada uma propriedade ϕ, se desejarmos que a mesma seja um invariante, basta verificarmos AG ϕ. O algoritmo provido pelo NuSMV só pode ser utilizado quando ϕ não possui operadores temporais.

Ao executar a declaração INVARSPEC ϕ, o NuSMV gera um BDD *InvarStates* com os estados onde ϕ é verdadeiro e calcula a sua negação ($\neg InvarStates$)(através da operação BDDNot), encontrando um BDD com os estados onde o invariante não é verdadeiro.

A partir dos estados iniciais, o NuSMV calcula um BDD com os todos os próximos estados de acordo com as relações de transição. Assim, aplicando a esse BDD a operação BDDAnd em conjunto com o BDD $\neg InvarStates$, podemos verificar se os próximos estados fazem ou não parte do conjunto dos estados onde o invariante não é verdadeiro. Se fizer parte desse conjunto, a ferramenta escolhe um estado qualquer

do BDD gerado pela operação BDDAnd e traça um caminho do estado inicial até o mesmo, devolvendo ao usuário sempre o menor contra-exemplo. Se não fizer, repete-se o algoritmo até encontrar todos os estados alcançáveis. Se todos os estados alcançáveis já foram calculados e a operação BDDAnd devolver um BDD vazio, então o invariante é verdadeiro no sistema.

Esse algoritmo funciona como uma espécie de busca em largura e é mais eficiente que o outro porque devolve uma solução assim que encontra um estado inconsistente com o invariante, ou seja, devolve uma solução em tempo de execução ("on-the-fly"). Além disso, essa solução é o menor caminho do estado inicial até o estado inconsistente, o que torna o contra-exemplo mais fácil de ser analisado.

Apesar de tal eficiência, não utilizaremos esse algoritmo (e portanto a declaração INVARSPEC) no nosso trabalho por uma simples razão: o mesmo funciona apenas para o operador AG. Como nossa proposta é abranger os oito operadores temporais e a declaração SPEC funciona para todos eles, o uso de tal algoritmo fica descartado.

Capítulo 4

Revisão de Modelos

Antes de detalharmos as idéias por trás da revisão de modelos, vamos começar com um exemplo que ilustra os problemas de modelagem e inconsistência que estão sujeitas as pessoas que trabalham com desenvolvimento de software.

4.1 Um exemplo

Suponha que um engenheiro de software deseja modelar um sistema de controle de iluminação de um prédio baseado na adaptação das especificações LCS (*The Light Control System*) feitas por Rodrigues et al [21]:

- O sistema possui 3 tipos de iluminação: iluminação padrão (d), iluminação escolhida pelo usuário (c) e sem iluminação (n).

- O sistema também controla o tempo que o prédio está desocupado e o tempo que o alarme está disparando.

- Quando a luz externa for muito forte a iluminação é suspensa. Se a luz externa é razoável para o ambiente e a iluminação foi escolhida pelo usuário, a iluminação padrão é estabelecida.

- Sempre que um usuário entrar no prédio, a iluminação padrão é estabelecida. Entretanto, se um usuário entrar no prédio antes de um tempo mínimo para reocupação, a iluminação é estabelecida de acordo com a escolha do usuário anterior. Se o tempo mínimo para reocupação expirar, a iluminação é suspensa.

- Quando o alarme é disparado e já se passou um determinado tempo, a iluminação é suspensa. Caso contrário, a iluminação padrão é estabelecida.

De acordo com esses requisitos o engenheiro de software decidiu modelar o sistema da seguinte forma:

- uma variável para representar os 3 tipos de iluminação (light).
- três variáveis de relógios (timeun para contar o tempo de desocupação, timeal para contar há quanto tempo o alarme está disparando e timer que será utilizado para simular eventos).
- duas variáveis para simulação de eventos (user para detectar a presença de pessoas e alarm para detectar que há fogo).
- uma variável sensorial (lux para verificar a luminosidade externa).

Podemos modelar o sistema com a seguinte descrição na linguagem do NuSMV:

```
MODULE main
VAR
light : {n,d,c};    -- state variable
timeun : 0..3;      -- timer variables
timeal : 0..3;
timer : 0..6;
user : boolean;     -- simulation variables
alarm : boolean;

IVAR
lux : {1,2};        -- input variable

ASSIGN
init(light) := d; init(timeun) := 3; init(timeal) := 0;
init(timer) := 0; init(user) := 1; init(alarm) := 0;

next(light) := case
  lux = 2 : n;
  lux = 1 & light = c : d;
  timeun < 3 & user : c;
  user : d;
  timeun = 3 : n;
  timeal < 3 & alarm : d;
  timeal = 3 & alarm : n;
  1 : light;
esac;
next(timeun) := case
  user : 0;
  timeun = 3 : 3;
  !user : timeun + 1;
  1 : timeun;
esac;
```

```
next(timeal) := case
  !alarm : 0;
  timeal = 3 : 3;
  alarm : timeal + 1;
  1 : timeal;
esac;
next(timer) := case
  timer = 6 : 0;
  timer < 6 : timer + 1;
  1 : timer;
esac;
next(user) := case
  timer < 2: 1;
  timer <= 6 : 0;
  1 : user;
esac;
next(alarm) := case
  timer < 1 : 0;
  timer <= 6 : 1;
  1 : alarm;
esac;
```

A definição dos estados iniciais serve para simular cenários. No caso da descrição acima a iluminação é padrão (light = d), o tempo estabelecido para reocupação já expirou (timeun = 3) e o usuário acabou de entrar no prédio (user = 1). A expressão case da variável user funciona como simulador da entrada do usuário (user = 1) a cada cinco instantes de tempo, pois entre $2 \leq$ timer ≤ 6 o mesmo encontra-se fora (user = 0). O usuário permanece dentro do prédio por dois instantes de tempo ($0 \leq$ timer < 2). A expressão case da variável alarm é bastante semelhante ao da variável user. O alarme pára (alarm = 0) a cada seis instantes de tempo, pois entre $1 \leq$ timer ≤ 6 o mesmo está disparando (alarm = 1). O alarme permanece desligado por apenas um instante de tempo ($0 \leq$ timer < 1). A variável timer funciona como um relógio para as simulações das variáveis user e alarm.

A variável timeal controla o tempo estabelecido pelo administrador do sistema para o alarme estar disparando (timeal = 3) e a variável timeun controla o tempo máximo estabelecido para a reocupação (timeun = 3). A variável lux representa a luminosidade externa, que pode ser alta (lux =2) ou média (lux = 1). Por se tratar de uma variável de entrada, a variável lux não possui controle de valor. A expressão case da variável light é o núcleo do sistema, pois controla a iluminação do estado corrente. As sete linhas internas na expressão são baseadas, de cima para baixo respectivamente, nas especificações e_2, e_1, r_6, r_7, r_5, s_1 e s_2 de Rodrigues et al [21].

Essa modelagem poderia gerar até 1344 estados. Mas devido às restrições de pré-condições, o máximo de estados alcançáveis se resume a 16, como podemos ver na figura 4.1.

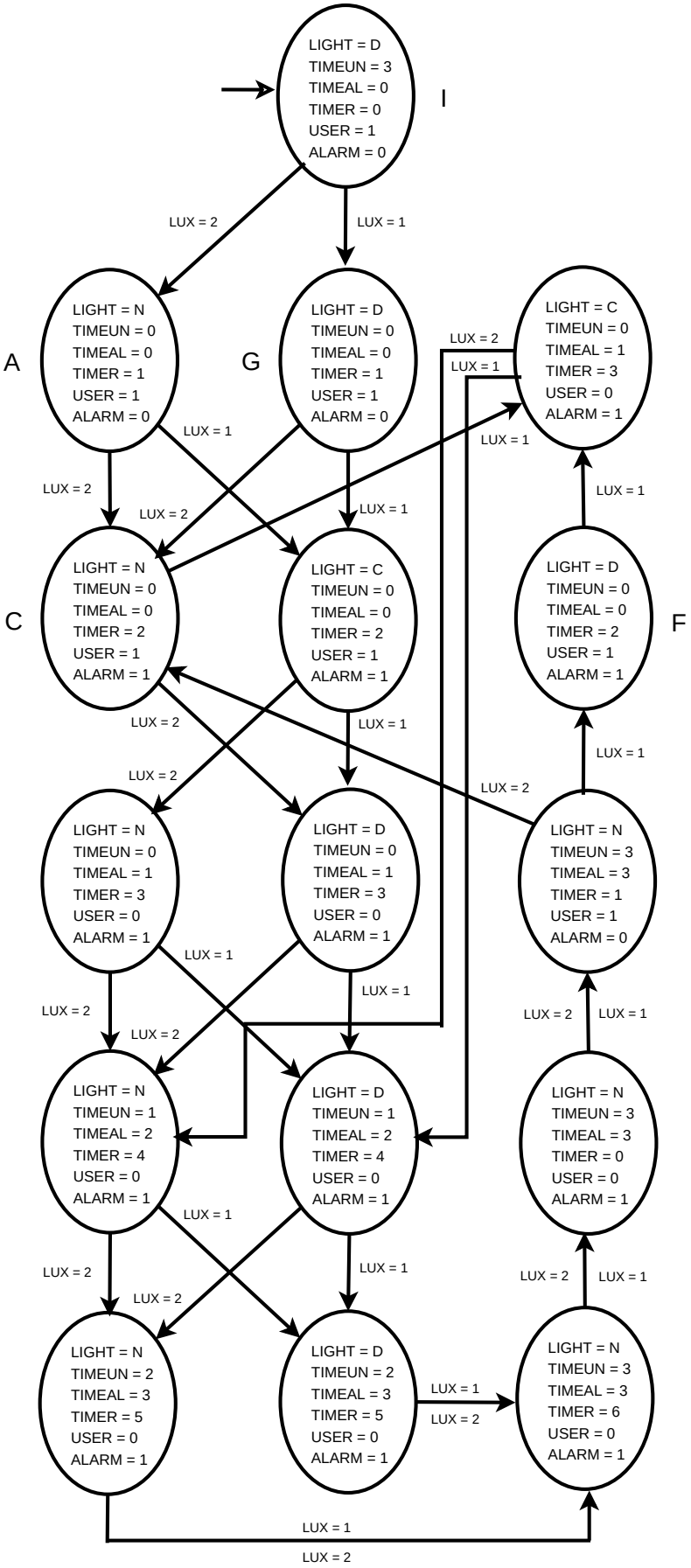

Figura 4.1: Modelo visual do exemplo

Suponha que o engenheiro deseja que a propriedade do controle de reocupação funcione em todo o sistema e por isso a modelou como um invariante. Esse invariante foi descrito na linguagem do NuSMV da seguinte maneira:

```
SPEC
AG(((timeun < 3 & user) -> light = c) & (timeun = 3 -> light = n))
```

A saída da verificação realizada pelo verificador de modelos NuSMV é um caminho a partir do estado inicial para o estado inconsistente. Se ativarmos o NuSMV se comprovará que essas propriedades não são verdadeiras, mostrando um estado (inicial) onde o tempo para a reocupação expirou e a iluminação não está suspensa. Segue abaixo a saída do NuSMV para esse exemplo.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- specification  AG (((timeun < 3 & user) -> light = c) &
(timeun = 3 -> light = n))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
 light = d
 timeun = 3
 timeal = 0
 timer = 0
 user = 1
 alarm = 0
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Se existirem muitas variáveis no estado e/ou a fórmula temporal for complexa, fica difícil identificar onde está o problema. Além disso, só é mostrado um estado inconsistente por vez e, mesmo descobrindo a causa e corrigindo o problema daquele estado, poderão existir outros gargalos de inconsistência. Por exemplo, a modificação do nosso estado inicial para {light = n, timeun = 3, timeal = 0, timer = 0, user = 1, alarm = 0} não torna a fórmula verdadeira, apesar do estado inicial ser consistente com a mesma. O que pretendemos é fazer algo que já revise todos os estados inconsistentes com a fórmula temporal e dê uma solução para o usuário modificar a descrição do sistema removendo/acrescentando transições.

4.2 Estado como Modelo

O primeiro passo quando se deseja aplicar revisão de crenças é definir como será a representação das crenças. Como estamos lidando com máquinas de estados finitos, cada estado possui informação completa sobre o mundo, ou seja, cada estado pode ser

comparado a um modelo completo das variáveis do sistema. Assim, um estado {light = n, timeun = 3, timeal = 3, timer = 1, user = 1, alarm = 0} do nosso exemplo é um possível modelo do conjunto {light, timeun, timeal, timer, user, alarm}. Portanto, iremos adotar para as crenças a representação baseada em modelos (estados, mundos possíveis). A partir deste ponto, sempre que nos referirmos a um estado estaremos nos referindo a um modelo completo das variáveis do sistema.

4.3 Mudança Mínima

No capítulo 2 introduzimos as idéias mais relevantes na área de revisão de crenças. Uma das idéias chaves é o Princípio da Mudança Mínima, que afirma que a mudança no conjunto original de crenças deva ser a mínima possível. Mas como definir uma mudança mínima? Rodrigues [22] definiu uma função d que mede quantitativamente quão próximo um modelo (estado) está de outro através do número de variáveis proposicionais que possuem valores diferentes nos modelos (estados). A nossa idéia de mudança mínima será baseada nessa função, mas com uma abrangência maior. Não ficaremos restritos a variáveis proposicionais. Qualquer diferença de valor entre variáveis influenciará no cálculo. Assim a nossa função distância d é definida como:

Função *d* Sejam A e B modelos (estados). A distância entre A e B é o número de variáveis p_i tal que $A(p_i) \neq B(p_i)$, onde $A(p_i)$ é o valor da variável p_i no modelo (estado) A e $B(p_i)$ é o valor da variável p_i no modelo (estado) B.

Por exemplo, se

A = {light = n, timeun = 3, timeal = 3, timer = 1, user = 1, alarm = 0} e

B = {light = d, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1},

então $d(A, B) = 5$.

Assim como propôs Rodrigues, podemos também extender a função d. A diferença é que nossa extensão calcula a menor distância entre um modelo (estado) e um conjunto de modelos (estados), ao invés de calcular entre dois conjuntos. Na próxima seção explicaremos o porquê disso.

Função *D* Seja C um modelo (estado), Θ um conjunto de modelos (estados) e *min* uma função que devolve o menor valor de um conjunto de números naturais. A distância

entre C e Θ é definida como :

$$D(C,\Theta) = \begin{cases} min\{d(C,E) \mid E \in \Theta\} & \text{se existe } C \text{ e } \Theta \text{ não é vazio} \\ \infty & \text{caso contrário} \end{cases}$$

4.4 Revisão no Estado

Um vez que já definimos que a nossa representação de crenças será baseada em mundos possíveis, para a estrutura da revisão ficar completa, precisamos definir: como será o nosso sistema de esferas, mais especificamente em qual conjunto de crenças o mesmo será centrado (base de conhecimento atual) e qual o critério de ordenação das esferas; e como será a crença recebida.

O nosso sistema de esferas será centrado em um estado, ou seja, será centrado em um único mundo possível. A ordem das esferas será baseada no valor da função d descrita anteriormente. No segundo nível estão os mundos que possuem distância igual a 1 em relação ao mundo do centro (estado). No terceiro nível estão os mundos com distância igual a 2, no quarto com distância igual a 3, e assim por diante, como podemos ver na figura 4.2

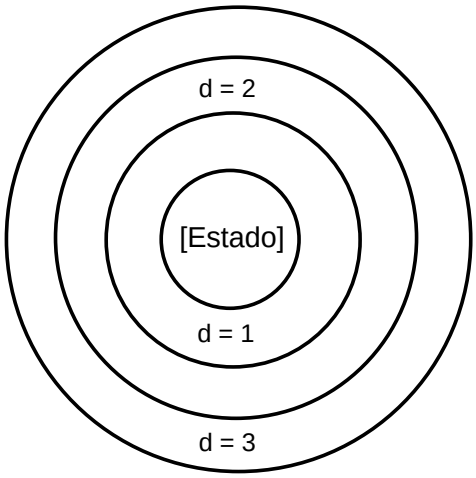

Figura 4.2: Sistema de esferas centrado em um estado

Vimos no capítulo 2 que a idéia de revisão de crenças baseada em comparação de modelos (estados) é encontrar os modelos (estados) da sentença recebida que são mais próximos dos modelos (estados) da base de conhecimento. Se pensarmos em termos de um sistema de esferas, a revisão é a intersecção dos mundos possíveis da sentença recebida com a esfera mais próxima da esfera central.

Como a nossa base de conhecimento é um único modelo (estado), não há necessidade de compararmos conjuntos de modelos (estados). Precisamos apenas comparar o modelo

(estado) da base com os modelos (estados) da sentença recebida. Em razão disso, a nossa definição da função D pode ser mais específica que a definição de Rodrigues.

Mas como garantir que a sentença recebida possua pelo menos um modelo? Para isso é preciso garantir que a sentença seja consistente e que seja descrita em alguma lógica que não sofra interferência com as mudanças propostas. Isso exclui a lógica temporal. Por outro lado, a lógica proposicional é imutável em relação às mudanças no espaço de estados do sistema. Assim, se a sentença recebida for uma fórmula proposicional consistente, então garantimos que sempre existirá pelo menos um modelo. No NuSMV qualquer expressão simples de sua sintaxe (ver anexo A) pode ser adaptada à lógica proposicional. Portanto, a crença recebida será uma expressão simples cuja adaptação para essa lógica produza um fórmula consistente.

Assim, mais uma vez baseado na definição de Rodrigues, a nossa revisão pode ser definida como:

Revisão Seja F um modelo (estado) que representa minhas crenças atuais e Ψ um conjunto de modelos de uma sentença α, então o resultado da revisão $F * \alpha$ será um conjunto Δ tal que:

$$\Delta = \{G \in \Psi \mid d(F, G) = D(F, \Psi)\}$$

Se pegarmos um modelo (estado) do nosso exemplo

F = {light = d, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}

e o revisarmos por uma sentença

α = {((timeun $<$ 3) & user $\rightarrow$ light = c) & (timeun = 3 $\rightarrow$ light = n)}

o resultado será o seguinte conjunto:

Δ = {{light = c, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1},
{light = d, timeun = 0, timeal = 0, timer = 2, user = 0, alarm = 1}}

A sentença α possui 784 modelos, mas apenas dois possuem a menor distância (1) para o modelo (estado) representado por F.

4.5 Mudanças na Descrição

Uma vez que já comparamos os estados a modelos e já mostramos como fazer a revisão baseada em comparação de modelos (estados), a próxima etapa é definir as mudanças a serem realizadas na descrição do sistema para se alcançar o estado revisado.

Como o sistema é representado como um grafo de transições de estados, basta nos concentrarmos em modificar as transições entre um estado e seu estado sucessor inconsistente. Não iremos modificar a propriedade temporal a ser verificada, nem a quantidade e nem o tipo das variáveis.

Vimos anteriormente que as transições são definidas no NuSMV através das expressões next em conjunto com as expressões case. Assim, a partir de agora consideraremos como uma transição cada linha (pré-condição : pós-condição) de uma expressão case. A idéia é acrescentar/eliminar transições na descrição do sistema de tal forma que o estado revisado seja alcançado a partir de seu estado antecessor.

A figura 4.3 representa uma transição do nosso exemplo que leva a um estado inconsistente com a fórmula $\alpha = \{((\text{timeun} < 3)\ \&\ \text{user} \rightarrow \text{light} = \text{c})\ \&\ (\text{timeun} = 3 \rightarrow \text{light} = \text{n})\}$ e uma possível revisão desse estado.

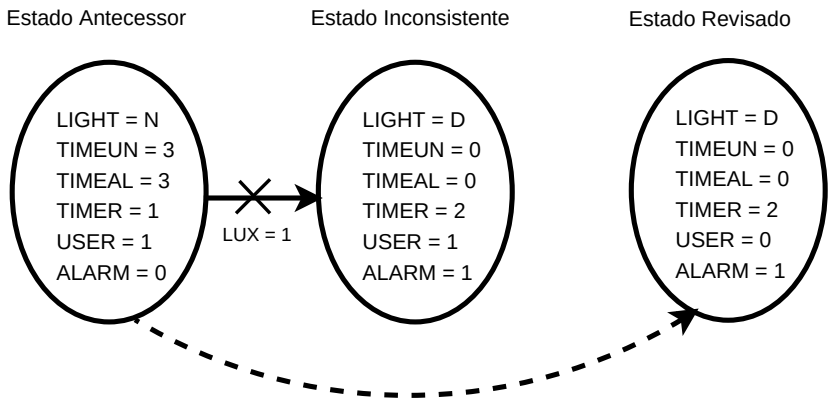

Figura 4.3: O objetivo é forçar a transição pontilhada e evitar a outra

Para descobrir quais expressões next devemos modificar, precisamos apenas comparar o estado original inconsistente com o estado revisado e encontrar as variáveis que possuem valores diferentes. Na figura 4.3 só há uma variável : user.

Para alcançar o estado revisado a partir do antecessor temos duas opções. Uma é acrescentar uma transição que modifique ou mantenha o valor da variável de acordo com o estado revisado. Para definir a transição a ser acrescentada é preciso garantir que ela seja aplicada unicamente no estado antecessor a fim de evitar interferências em outras transições do sistema. A única maneira de garantir isso é que tal transição possua como pré-condição a conjunção formada pelas variáveis comuns e pela variáveis de entrada do estado antecessor. A pós-condição é o valor da variável no estado revisado.

No exemplo da figura 4.3, acrescentaríamos a transição "light = n & timeun = 3 & timeal = 3 & timer = 1 & user = 1 & alarm = 0 & (lux = 1) : 0;" na expressão case que está internamente na expressão next da variável user. Essa transição é de modificação, pois há mudança do valor da variável do estado antecessor para o alcançado. Mas em outros casos poderíamos acrescentar uma transição de manutenção, onde não há mudança do valor da variável do estado antecessor para o alcançado. Ainda é preciso definir em qual "posição" das expressões case iremos adicionar as transições. Como as pré-condições são analisadas de cima para baixo e desejamos que essas transições sejam obrigatoriamente efetivadas, então as acrescentamos no início das expressões case.

A outra opção é eliminar transições. Primeiro é preciso verificar de cima para baixo se há alguma transição na expressão case da variável que modifique ou mantenha o seu valor de acordo com o estado revisado. Se essa transição existir, então podemos eliminar as demais transições que estão acima dela na expressão case e que podem ser efetivadas no estado antecessor. No nosso exemplo, podemos simplesmente eliminar a transição "timer $<$ 2 : 1;" da expressão case da variável user.

Note que nem sempre é possível alcançar o estado revisado apenas eliminando transições, pois às vezes não há na descrição uma transição que modifique a variável para seu valor no estado revisado. No nosso exemplo, se não tivéssemos a transição "timer $<=$ 6 : 0;" nunca alcançaríamos um estado onde "user = 0" utilizando apenas a opção de eliminar transições.

Vale ressaltar que essa abordagem só é possível porque as transições dependem apenas dos valores das variáveis e não há estados duplicados no modelo do sistema.

Assim, podemos ter duas variações possíveis para a nova descrição da expressão next da variável user de tal forma que o estado antecessor alcance o novo estado revisado em vez do estado original inconsistente.

```
Opção 1 :   /* Acrescenta uma transição */
next(user) := case
  light = n & timeun = 3 & timeal = 3 & timer = 1 & user = 1 &
  alarm = 0 & (lux = 1) : 0;
  timer < 2 : 1;
  timer <= 6 : 0;
  1 : user;
esac;

Opção 2 :   /* Elimina uma transição */
next(user) := case
  timer <= 6 : 0;
  1 : user;
esac;
```

Mas há um porém quando se pensa em eliminar uma transição. Quando estamos acrescentando, não modificamos o espaço de estados gerado anteriormente, o que garante

que o estado revisado seja alcançado. Já quando eliminamos um transição, podemos modificar o espaço de estados já gerado e analisado anteriormente, sem garantia alguma de alcançar o estado revisado. Para solucionar esse problema é preciso armazenar as transições que já foram utilizadas no espaço de estados já gerado. Se uma dada transição nunca foi utilizada, então podemos eliminá-la. Assim, garantimos que tal eliminação não vai afetar os estados antecessores e o estado revisado será alcançado. Caso contrário, não podemos eliminá-la.

Há ainda alguns casos onde precisaremos alterar o estado inicial. Como não há estado antecessor, basta modificarmos na descrição os valores das variáveis nas declarações init de acordo com o estado revisado.

4.6 O Algoritmo

Já sabemos como realizar a revisão de um estado inconsistente com uma dada fórmula e como modificar a descrição do sistema para que o estado antecessor alcance o estado revisado. A nossa próxima etapa é definir um algoritmo que encontre todas as mudanças a serem feitas na descrição do sistema de tal maneira que a fórmula temporal se torne verdadeira no novo modelo.

Dada uma máquina de estados finitos e uma fórmula temporal $T\alpha$, onde T é um dos oito operadores temporais e α uma expressão simples e consistente, a idéia principal é considerar cada estado como o conjunto original de crenças e α como a crença recebida. Uma vez que a verificação de uma fórmula temporal passa pela determinação de um estado e o NuSMV sempre escolhe o estado inicial, o nosso algoritmo se inicia considerando o estado inicial como o conjunto atual de crenças.

O algoritmo vai percorrendo os caminhos do modelo do sistema a partir do estado inicial via busca em profundidade e, assim que encontra uma inconsistência com a fórmula temporal, um estado é revisado. Para se alcançar esse estado revisado primeiro tentamos remover transições. Caso não seja possível, são adicionadas transições na descrição do sistema que garantam que esse novo estado seja alcançado a partir do seu estado antecessor.

O processo de "parada" do algoritmo varia de acordo com o tipo de operador temporal que estamos lidando. Um único critério será utilizado: escolhido um caminho, retardamos o máximo possível o processo de revisão, garantindo, para esse caminho, o maior número de estados idênticos em relação ao modelo original. No final do algoritmo temos todas as transições que devem ser removidas e/ou adicionadas na descrição, bem como se é necessário modificar o estado inicial, para que a fórmula temporal seja verdadeira no sistema.

É importante frisar que a modificação do estado inicial só ocorrerá em casos estri-

tamente necessários, ou seja, apenas com os operadores AG, EG, AU e EU.

A seguir temos um pseudo-código do nosso algoritmo, que recebe como entrada o estado inicial e uma fórmula temporal.

FUNCTION Revisador_Modelos
Entrada: estado Inicial e fórmula CTL F
Saída: sugestões de mudanças na descrição do sistema e efeitos colaterais

```
1: if F == AG f1 then
2:    return Revisa_AG(Inicial,f1);
3: else if F == AX f1 then
4:    return Revisa_AX(Inicial,f1);
5: else if F == AF f1 then
6:    return Revisa_AF(Inicial,f1);
7: else if F == AU[f1 U f2] then
8:    return Revisa_AU(Inicial,f1,f2);
9: else if F == EG f1 then
10:   return Revisa_EG(Inicial,f1);
11: else if F == EX f1 then
12:   return Revisa_EX(Inicial,f1);
13: else if F == EF f1 then
14:   return Revisa_EF(Inicial,f1);
15: else if F == EU[f1 U f2] then
16:   return Revisa_EU(Inicial,f1,f2);
17: end if
```

A função principal funciona como um parser. Apenas determina o tipo de fórmula temporal que estamos lidando. É importante ressaltar que cada f1 e f2 dessa função é uma expressão simples consistente, ou seja, uma expressão que pode ser mapeada para uma fórmula proposicional consistente.

Antes de detalharmos cada uma das oito funções específicas a cada tipo de operador temporal, iremos apresentar algumas funções comuns a elas. Por questões de praticidade omitiremos as definições das variáveis.

A função **Reconstroi_Trans** refaz as transições de tal maneira que o estado antecessor alcance o revisado. Essa função recebe como entrada o estado antecessor, o atual e a sua revisão, devolvendo as modificações a serem realizadas pelo usuário. Primeiro tenta-se remover uma transição (função **Pode_Remover**). Caso não se obtenha sucesso, então adiciona-se uma transição através da função **Adiciona_Trans**. O algoritmo para remoção ou adição de transições é baseado nas idéias da seção anterior. Há ainda as funções **Imprime_Remocao** e **Imprime_Adicao** para imprimir respectivamente as transições removidas e adicionadas.

FUNCTION Reconstroi_Trans
Entrada: estados Antecessor, Atual e Revisado
Saída: sugestões de remoção/adição na descrição do sistema para o estado Antecessor alcançar o estado Revisado

```
1: for all variável de Atual diferente de Revisado do
2:    if Pode_Remover(Antecessor,Atual,Revisado) then
3:       Imprime_Remocao();
4:    else
5:       Adiciona_Trans(Antecessor,Atual,Revisado);
6:       Imprime_Adicao();
7:    end if
8: end for
9: return ;
```

Como a função **Adiciona_Trans** é linear, a complexidade da função **Reconstroi_Trans** se situa no laço (linha 1) versus a função **Pode_Remover** (linha 2). A complexidade de **Pode_Remover** se dá pelo número de linhas (multiplicado por 2, pois percorremos duas vezes) da expressão case da variável analisada. Assim, seja T o número de variáveis de um estado e L a maior quantidade de linhas dentre as expressões case, temos que a complexidade da função **Reconstroi_Trans** é $O(T \times (2 \times L))$.

Temos também a função **Revisao**, que devolve um estado revisado de acordo com as definições já apresentadas, realiza as modificações necessárias nas transições e as imprime em tempo de execução. A função **Inconsistente** verifica se um modelo (estado) é consistente ou não com uma fórmula.

FUNCTION Revisao
Entrada: estados Antecessor e Atual e fórmula F
Saída: estado Revisado ou Atual

```
1: if Inconsistente(Atual,F) then
2:    Calcula conjunto Δ da definição de Revisão;
3:    Revisado ⇐ escolha aleatória de um elemento de Δ;
4:    Reconstroi_Trans (Antecessor,Atual, Revisado);
5:    return Revisado;
6: else
7:    return Atual;
8: end if
```

Na linha 1 verificamos se o estado atual é inconsistente com a fórmula, pois em alguns casos chamaremos a função **Revisao** sem saber se isso ocorre. Nas linhas 2 e 3 definimos o estado revisado e na linha 4 chamamos a função **Reconstroi_Trans** que refaz e imprime as transições de acordo com as possibilidades encontradas.

O gargalo da complexidade da função **Revisao** se concentra quando calculamos o conjunto Δ (linha 2) e quando chamamos a função **Reconstroi_Trans** (linha 4). Liberatore et al [23] mostram que, na lógica proposicional, a revisão de crenças baseada

em comparação de modelos (estados) é exponencial no número de variáveis. No nosso caso, a complexidade é definida quando comparamos os modelos da fórmula com o modelo (estado) para se calcular a distância mínima. Como estamos lidando com variáveis booleanas e escalares (número finito de valores), podemos fazer uma aproximação para a complexidade da lógica proposicional, ou seja, a nossa revisão pode ser $O(2^T \times T)$, onde T é o número de variáveis. Portanto, a complexidade final da função **Revisao** é $O(2^T \times T + T \times (2 \times L))$, onde L é a maior quantidade de linhas dentre as expressões cases.

A função **Revisao_Inicial** é específica para se realizar revisão no estado inicial, devolvendo um estado revisado e imprimindo as mudanças necessárias (**Imprime_Inicial**).

FUNCTION Revisao_Inicial
Entrada: estado Atual e fórmula F
Saída: mudança sugerida para estado inicial com o estado Revisado ou somente o estado Atual

```
1: if Inconsistente(Atual,F) then
2:    Calcula conjunto Δ da definição de Revisão;
3:    Revisado ⇐ escolha aleatória de um elemento de Δ;
4:    Imprime_Inicial();
5:    return Revisado;
6: else
7:    return Atual;
8: end if
```

A função **Revisao_Inicial** é semelhante a função **Revisao**. A diferença é que, ao invés de reconstruir transições (**Reconstroi_Trans**), ela apenas imprime as modificações a serem feitas no estado inicial. Como a função **Imprime_Inicial** é linear, a complexidade dessa função é $O(2^T \times T)$, onde T representa o número de variáveis de um estado.

No nosso algoritmo há também o uso intensivo de uma pilha global **P** e uma lista de adjacências **AdjList** que armazena para cada estado os seus estados sucessores. Essa lista e essa pilha são importantes, pois, para percorrermos o grafo, utilizaremos a abordagem de busca em profundidade. A pilha é manipulada com as funções usuais: **Empilha**, **Topo_Pilha** e **Desempilha**. A função **Remove** retira aleatoriamente um estado de uma dada lista de adjacências. A função **Recria_AdjList** refaz a lista de adjacências e é útil quando há revisão, pois nesse caso há mudanças nas transições de estados. A função **Trans_Usadas** armazena as transições já efetivadas até o momento, que são usadas pela função **Pode_Remover**.

Agora iremos apresentar as funções específicas para cada operador. Começaremos com a função responsável por revisar o modelo para que uma fórmula temporal AG se torne verdadeira.

FUNCTION Revisa_AG
Entrada: estado Inicial e fórmula F
Saída: mudanças sugeridas na descrição do sistema para a fórmula AG F se tornar verdadeira

```
1:  Inicial ⇐ Revisao_Inicial(Inicial,F);
2:  Recria_AdjList();
3:  Empilha(Inicial,P);
4:  while P != ∅ do
5:     S ⇐ Topo_Pilha(P);
6:     if AdjList(S) != Nil then
7:        W ⇐ Remove(AdjList(S));
8:        if Inconsistente(W,F) then
9:           W ⇐ Revisao(S,W,F);
10:          Recria_AdjList();
11:       end if
12:       if W ∉ P then
13:          Empilha(W,P);
14:       end if
15:       Trans_Usadas(S,W);
16:    else
17:       Desempilha(P);
18:    end if
19: end while
20: return  ;
```

A função **Revisa_AG** recebe como entrada um estado inicial e uma fórmula e ao final devolve as mudanças a serem feitas para o usuário. Nessa função, primeiro tratamos de deixar o estado inicial consistente com a fórmula (linha 1). Se a mudança no estado inicial foi necessária, então precisamos também refazer a lista de adjacências (linha 2). A linha 7 remove um estado W sucessor do estado S da lista de adjacências. Se esse estado W for inconsistente (linha 8), então o revisamos e recriamos a lista.

Na linha 13 há a garantia de que todo estado empilhado é consistente e não está na pilha **P**. Na linha 15 armazenamos as transições já utilizadas. Como estamos lidando com uma busca exaustiva, podemos afirmar com certeza que o modelo é verdadeiro para AG no estado inicial. A complexidade dessa função se concentra nas linhas 4 e 9. Seja M a quantidade de estados no modelo do sistema (exponencial no número de variáveis), temos que a linha 4 consome $O(M)$. Seja N a complexidade da função **Revisao** já mostrada anteriormente, temos que a complexidade da função **Revisa_AG** é $O(M \times N)$, ou seja, no pior caso precisamos revisar todos os estados do modelo.

A função **Revisa_AX** também recebe como entrada um estado inicial e uma fórmula e imprime as mudanças para o usuário, sendo parecida com a anterior. A diferença é que não revisamos o estado inicial e não empilhamos estados.

FUNCTION Revisa_AX
Entrada: estado Inicial e fórmula F
Saída: mudanças sugeridas na descrição do sistema para a fórmula AX F se tornar verdadeira

```
1:  Empilha(Inicial,P);
2:  while P != ∅ do
3:     S ⇐ Topo_Pilha(P);
4:     if AdjList(S) != Nil then
5:        W ⇐ Remove(AdjList(S));
6:        if Inconsistente(W,F) then
7:           W ⇐ Revisao(S,W,F);
8:           Recria_AdjList();
9:        end if
10:       Trans_Usadas(S,W);
11:    else
12:       Desempilha(P);
13:    end if
14: end while
15: return  ;
```

Para cada sucessor W do estado S (linha 5), se o mesmo for inconsistente com a fórmula F, fazemos a revisão (linha 7) e recriamos a lista (linha 8). Nesse caso, como se analisa todos os estados sucessores do estado inicial, podemos afirmar com certeza que o modelo é verdadeiro para AX no estado inicial. A complexidade dessa função se concentra nas linhas 5 e 7. Seja R a quantidade de estados que sucedem o estado inicial (também exponencial no número de variáveis) e seja N a complexidade da função **Revisao**, temos que a complexidade da função **Revisa_AX** é $O(R \times N)$.

A função **Revisa_AF** também recebe como entrada um estado inicial e uma fórmula e devolve as mudanças. Como em todas as funções anteriores, essa função começa empilhando o estado inicial. Na linha seguinte começamos uma busca em profundidade. A linha 6 funciona para eliminar a busca por caminhos que já são consistentes. Quando existir um caminho (linha 7) e esse caminho até aquele momento não possui nenhum estado onde a fórmula F seja verdadeira, então o revisamos (linha 8) e recriamos a lista (linha 9). As linhas seguintes garantem que enquanto não houver um ciclo e, portanto, o "fim" de um caminho, a busca continua empilhando estados inconsistentes (linha 11). Como estamos lidando com uma busca exaustiva, então podemos afirmar com certeza que em todos os caminhos a partir do inicial futuramente a fórmula F será verdadeira.

A complexidade dessa função se concentra nas linhas 2 e 8 e é semelhante a da função **Revisa_AG**. Seja M a quantidade de estados no modelo do sistema, temos que a linha 2 consome $O(M)$. Seja N a complexidade da função **Revisao**, temos que a complexidade da função **Revisa_AF** é $O(M \times N)$.

A função **Revisa_AU** recebe como entrada um estado inicial e duas fórmulas e devolve as mudanças para o usuário. A linha 8 funciona para eliminar a busca por

FUNCTION Revisa_AF
Entrada: estado Inicial e fórmula F
Saída: mudanças sugeridas na descrição do sistema para a fórmula AF F se tornar verdadeira

```
1:  Empilha(Inicial,P);
2:  while P != ∅ do
3:     S ⇐ Topo_Pilha(P);
4:     if AdjList(S) != Nil then
5:        W ⇐ Remove(AdjList(S));
6:        if Inconsistente(W,F) then
7:           if W ∈ P then
8:              W ⇐ Revisao(S,W,F);
9:              Recria_AdjList();
10:          else
11:             Empilha(W,P);
12:          end if
13:       end if
14:       Trans_Usadas(S,W);
15:    else
16:       Desempilha(P);
17:    end if
18: end while
19: return  ;
```

caminhos já consistentes. Quando o "fim" de um caminho é encontrado (linha 9), então revisamos o estado (linha 10) e recriamos a lista de adjacências (linha 11). A função **Revisa_AU** é bastante semelhante a anterior (**Revisa_AF**). A principal diferença é que só empilha estados que são consistentes com a fórmula F1 (linha 15). Como estamos lidando com uma busca exaustiva, isso garante que em todos os caminhos, a fórmula F1 vai ser verdadeira até se encontrar um estado verdadeiro com a fórmula F2. A outra diferença está no tratamento do estado inicial. Nesse caso precisamos garantir que o estado inicial seja consistente com a fórmula F1.

A complexidade dessa função se concentra nas linhas 4, 10 e 13 e se assemelha as funções **Revisa_AF** e **Revisa_AG**. Isso já era esperado, pois essas funções exploram exaustivamente o espaço de estados para encontrar todos os caminhos. Seja M a quantidade de estados no modelo do sistema, temos que a linha 4 consome $O(M)$. Seja N a complexidade da função **Revisao** já mostrada, então a complexidade da função **Revisa_AU** é $O(M \times N)$.

Com os operadores existenciais, vamos modificar a estrutura do algoritmo, pois não há a necessidade de se fazer uma busca completa pelo espaço de estados. A pilha e a lista de adjacências funcionam apenas para guiar o caminho. A função **Revisa_EG** recebe como entrada um estado inicial e uma fórmula e devolve as mudanças necessárias para o usuário.

FUNCTION Revisa_AU
Entrada: estado Inicial e fórmulas F1 e F2
Saída: mudanças sugeridas na descrição do sistema para a fórmula A[F1 U F2] se tornar verdadeira

```
1:  Inicial ⇐ Revisao_Inicial(Inicial,F1);
2:  Recria_AdjList();
3:  Empilha(Inicial,P);
4:  while P != ∅ do
5:     S ⇐ Topo_Pilha(P);
6:     if AdjList(S) != Nil then
7:        W ⇐ Remove(AdjList(S));
8:        if Inconsistente(W,F2) then
9:           if W ∈ P then
10:             W ⇐ Revisao(S,W,F2);
11:             Recria_AdjList();
12:          else
13:             W ⇐ Revisao(S,W,F1);
14:             Recria_AdjList();
15:             Empilha(W,P);
16:          end if
17:       end if
18:       Trans_Usadas(S,W);
19:    else
20:       Desempilha(P);
21:    end if
22: end while
23: return  ;
```

A função **Check_EG** é uma função nativa do NuSMV que recebe um estado e verifica se EG vale nele. Primeiro tratamos de deixar o estado inicial consistente com a fórmula (linha 1). Se a nossa alteração deixou o modelo consistente, então devolvemos a modificação apenas no estado inicial (linha 4). Caso contrário, vamos percorrer o grafo aleatoriamente (linha 6), revisando os estados inconsistentes (linha 7) até se garantir que a verificação é verdadeira a partir do estado inicial. Como todos os estados do caminho (pilha) são consistentes com a fórmula até o momento da saída do laço, podemos garantir que a fórmula F é verdadeira em um caminho do modelo a partir do estado inicial.

A complexidade dessa função se concentra nas linhas 4 e 7. Como pode existir um caminho que percorre todos os estados do grafo, então a linha 4 pode ser executada até M vezes, onde M é a quantidade de estados no modelo do sistema. Como a verificação realizada na linha 4 também consome O(M), então, seja N a complexidade da função **Revisao** já mostrada anteriormente, temos que a complexidade da função **Revisa_EG** é O($M^2 \times N$).

A função **Revisa_EX** recebe como entrada um estado inicial e uma fórmula e devolve as mudanças necessárias a serem feitas pelo usuário.

FUNCTION Revisa_EG
Entrada: estado Inicial e fórmula F
Saída: mudanças sugeridas na descrição do sistema para a fórmula EG F se tornar verdadeira

```
1: Inicial ⇐ Revisao_Inicial(Inicial,F);
2: Recria_AdjList();
3: Empilha(Inicial,P);
4: while Check_EG(Inicial,F) == false do
5:    S ⇐ Topo_Pilha(P);
6:    W ⇐ Remove(AdjList(S));
7:    W ⇐ Revisao(S,W,F);
8:    Recria_AdjList();
9:    Empilha(W,P);
10:   Trans_Usadas(S,W);
11: end while
12: return ;
```

FUNCTION Revisa_EX
Entrada: estado Inicial e fórmula F
Saída: mudanças sugeridas na descrição do sistema para a fórmula EX F se tornar verdadeira

```
1: Empilha(Inicial,P);
2: while Check_EX(Inicial,F) == false do
3:    S ⇐ Topo_Pilha(P);
4:    W ⇐ Remove(AdjList(S));
5:    W ⇐ Revisao(S,W,F);
6:    Recria_AdjList();
7: end while
8: return ;
```

Essa função é sem dúvida a mais simples de todas: escolhe-se aleatoriamente um estado sucessor do inicial e o revisa. A função **Check_EX** é uma função nativa do NuSMV que recebe um estado e verifica EX nele.

A estrutura dessa função é idêntica a da função **Revisa_EG**. A diferença é que não existe a necessidade de empilhar e nem de armazenar as transições realizadas. Pela simplicidade podemos garantir que EX é verdadeiro no estado inicial. Como temos certeza que o laço é executado uma única vez, essa função possui claramente complexidade $O(2 \times M + N)$, onde N representa a complexidade da função **Revisao** e M a quantidade de estados no modelo.

A função **Revisa_EF** também recebe como entrada um estado inicial e uma fórmula e devolve as mudanças para o usuário. A função **Check_EF** é uma função nativa do NuSMV que recebe um estado e verifica a validade de EF nele.

A função vai percorrendo o grafo (linha 4) até encontrar um ciclo e portanto um caminho (linha 5). Quando um caminho é encontrado, então é feita a revisão (linha 6). Note que esse algoritmo também segue o critério de revisão definido anteriormente:

FUNCTION Revisa_EF
Entrada: estado Inicial e fórmula F
Saída: mudanças sugeridas na descrição do sistema para a fórmula EF F se tornar verdadeira

```
1:  Empilha(Inicial,P);
2:  while Check_EF(Inicial,F) == false do
3:     S ⇐ Topo_Pilha(P);
4:     W ⇐ Remove(AdjList(S));
5:     if W ∈ P then
6:        W ⇐ Revisao(S,W,F);
7:        Recria_AdjList();
8:     else
9:        Empilha(W,P);
10:    end if
11:    Trans_Usadas(S,W);
12: end while
13: return  ;
```

escolhido um caminho, retardamos o quanto possível o processo de revisão. Como estamos lidando com um quantificador existencial, basta garantir que um único estado do grafo seja consistente com a fórmula F para garantirmos também que EF(F) será verdadeira.

A complexidade dessa função se concentra nas linhas 2 e 6. Seja M a quantidade de estados no modelo do sistema e seja N a complexidade da função **Revisao**, temos que a complexidade da função **Revisa_EF** é $O(M^2 + N)$. Isso devido ao fato da certeza que a linha 6 é executada uma única vez.

A função **Revisa_EU** recebe como entrada um estado inicial e duas fórmulas e devolve as mudanças para o usuário. A função **Check_EU** é uma função nativa do NuSMV que recebe um estado e verifica se EU vale nele.

Essa função possui bastante semelhança com as funções **Revisa_EG** e **Revisa_EF**, pois verifica se o estado inicial está consistente (linha 1) e verifica a formação de ciclos (linha 7). A grande diferença é que se força todos os estados do caminho a serem consistentes com a fórmula F1 (linha 11) antes de fazer a revisão pela fórmula F2 (linha 8). Com o mesmo argumento das funções semelhantes, podemos garantir que EU é verdadeiro no estado inicial.

A complexidade dessa função se concentra nas linhas 4, 8 e 11 e se assemelha a função **Revisa_EG**. Isso já era esperado, pois essas funções exploram um caminho no grafo, que pode ter no máximo o tamanho do espaço de estados. Seja M a quantidade de estados no modelo do sistema e seja N a complexidade da função **Revisao** já mostrada anteriormente, temos que a complexidade da função **Revisa_EU** é $O(M^2 \times N)$.

Note que esses algoritmos representam apenas uma das possibilidades de fazermos a revisão para que a fórmula temporal se torne verdadeira. Se a fórmula for do tipo E[α

FUNCTION Revisa_EU
Entrada: estado Inicial e fórmulas F1 e F2
Saída: mudanças sugeridas na descrição do sistema para a fórmula E[F1 U F2] se tornar verdadeira

```
1:  Inicial ⇐ Revisao_Inicial(Inicial,F);
2:  Recria_AdjList();
3:  Empilha(Inicial,P);
4:  while Check_EU(Inicial,F1,F2) == false do
5:     S ⇐ Topo_Pilha(P);
6:     W ⇐ Remove(AdjList(S));
7:     if W ∈ P then
8:        W ⇐ Revisao(S,W,F2);
9:        Recria_AdjList();
10:    else
11:       W ⇐ Revisao(S,W,F1);
12:       Recria_AdjList();
13:       Empilha(W,P);
14:    end if
15:    Trans_Usadas(S,W);
16: end while
17: return  ;
```

U β], AF β ou A[α U β], bastaria simplesmente revisarmos o estado inicial com algum modelo (estado) que fosse consistente com β para a fórmula ser tornar verdadeira. Mas como o estado inicial é tão claro para o usuário, não nos parece razoável que o mesmo já esteja errado. Outra possibilidade seria revisarmos o estado inicial com α e seus sucessores com β. Mas quanto mais próximo um estado está do estado inicial, mais certeza o usuário tem da sua validade. Por isso, tentamos adiar ao máximo a revisão quando um caminho é escolhido.

4.7 Efeitos Colaterais

O NuSMV possui a capacidade de analisar em uma mesma execução várias fórmulas temporais. Como a máquina de estados finitos é fixa, ou seja, possui sempre o mesmo estado inicial, os mesmos estados e as mesmas transições, a ferramenta pode verificar cada uma das fórmulas utilizando o algoritmo do capítulo 3 sem se preocupar se a análise de uma interfere na outra.

Como o nosso algoritmo foi incorporado ao NuSMV, a revisão do modelo do sistema pode ser feita também para cada uma das fórmulas. Mas no nosso caso, temos uma espécie de efeito colateral. Uma vez que estamos modificando a máquina de estados finitos, pode haver interferência da revisão de uma fórmula sobre a análise de outra, isto é, uma fórmula que era verdadeira pode se tornar falsa após as mudanças sugeridas

pela revisão de outra. E isso é um efeito indesejado, pois, para o usuário, a especificação pode deixar de fazer sentido, uma vez que não adianta "estragar" toda a parte que estava de acordo com os seus requisitos por causa de uma revisão.

Mas o inverso também pode acontecer: a revisão por uma dada fórmula pode tornar verdadeiras outras fórmulas que antes eram falsas, dispensando a necessidade de revisá-las (e portanto diminuindo a quantidade de mudanças) e melhorando de alguma forma a visão do usuário sobre o seu sistema.

Incorporamos ao nosso algoritmo um aviso sobre as fórmulas que eram verdadeiras e deixaram de ser e as que eram falsas e se tornaram verdadeiras, assim que um processo de revisão é realizado. Essa extensão foi de fácil implementação, pois ao final de uma revisão temos um novo modelo do sistema, bastando apenas confrontá-lo com as outras fórmulas através dos algoritmos de verificação nativos do NuSMV. Os detalhes da implementação desse e dos algoritmos da seção anterior podem ser encontrados no apêndice B.

4.8 Simulação

Vamos agora aplicar esse algoritmo ao nosso exemplo do início do capítulo. Para um melhor entendimento sugerimos a visualização pela figura 4.1 e pela figura 4.4, uma vez que as letras que dão nomes aos estados estão assinaladas nas mesmas.

1. A função principal **Revisador_Modelos** chama a função **Revisa_AG**
2. O estado inicial I é inconsistente com a fórmula
3. A revisão do estado inicial devolve o estado
   I' = {light = n, timeun = 3, timeal = 0, timer = 0, user = 1, alarm = 0}
4. O estado I' é empilhado
5. Início do laço
   (a) Pega-se o estado I' do topo da pilha
   (b) Um estado é retirado da lista de adjacências de I'. Esse estado é
   A = {light = n, timeun = 0, timeal = 0, timer = 1, user = 1, alarm = 0}
   (c) Como a fórmula é falsa, então o estado A é revisado, dando origem a
   B = {light = c, timeun = 0, timeal = 0, timer = 1, user = 1, alarm = 0}
   através da adição da transição
   (light = n) & (timeun = 3) & (timeal = 0) & (timer = 0) & (user = 1) & (alarm = 0) & (lux = 2) : c; na expressão case da variável light

(d) O estado B é empilhado, pois não está na pilha

```
----  Repetimos o laço pois a pilha não está vazia  ----
```

(a) Pega-se o estado B do topo da pilha

(b) Um estado é retirado da lista de adjacências de B. Esse estado é
C = {light = n, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}

(c) Como a fórmula é falsa, então o estado C é revisado, dando origem a
D = {light = c, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}
através da remoção da transição
(lux = 2) : n; na expressão case da variável light

(d) O estado D é empilhado, pois não está na pilha

```
----  Repetimos o laço até chegarmos a essa situação  ----
```

(a) Pega-se o estado E do topo da pilha

(b) Um estado é retirado da lista de adjacências de E. Esse estado é
F = {light = d, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}

(c) Como a fórmula é falsa, então o estado F é revisado, dando origem a
D = {light = c, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}
através da adição da transição
(light = n) & (timeun = 3) & (timeal = 3) & (timer = 1) & (user = 1) &
(alarm = 0) & ((lux = 1) | (lux = 2)) : c; na expressão case da variável light

(d) O estado D não é empilhado, pois já está na pilha

```
----  Repetimos o laço até chegarmos a essa situação  ----
```

(a) Pega-se o estado B do topo da pilha

(b) Um estado é retirado da lista de adjacências de B. Esse estado é
F = {light = d, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}

(c) Como a fórmula é falsa, então o estado F é revisado, dando origem a
D = {light = c, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}
através da adição da transição
(light = c) & (timeun = 0) & (timeal = 0) & (timer = 1) & (user = 1) &
(alarm = 0) & (lux = 1) : c; na expressão case da variável light

(d) O estado D é empilhado, pois não está na pilha

```
----  Repetimos o laço até chegarmos ao estado inicial ----
```

(a) Pega-se o estado I' do topo da pilha

(b) Um estado é retirado da lista de adjacências de I'. Esse estado é G = {light = d, timeun = 0, timeal = 0, timer = 1, user = 1, alarm = 0}

(c) Como a fórmula é falsa, então o estado G é revisado, dando origem a B = {light = c, timeun = 0, timeal = 0, timer = 1, user = 1, alarm = 0} através da adição da transição (light = n) & (timeun = 3) & (timeal = 0) & (timer = 0) & (user = 1) & (alarm = 0) & (lux = 1) : c; na expressão case da variável light

(d) O estado B é empilhado, pois não está na pilha

```
----  Repetimos o laço até a pilha se esvaziar ----
```

6. Fim do algoritmo.

É possível conhecer todas as alterações propostas para tornar a fórmula verdadeira através das impressões em tempo de execução. Esse exemplo conseguiu abranger os três tipos de mudança que o algoritmo é capaz de sugerir: revisão do estado inicial, adição de transições e remoção de transições. A seguir podemos visualizar a saída proveniente quando executamos o algoritmo. Como estamos verificando apenas uma fórmula, os efeitos colaterais da revisão não importam na especificação do usuário.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- specification  AG (((timeun < 3 & user) -> light = c) &
(timeun = 3 -> light = n))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample ->
State: 1.1 <-
  light = d
  timeun = 3
  timeal = 0
  timer = 0
  user = 1
  alarm = 0

-- but this specification can be true if you make the following
changes:

1) Change your initial state to (light = n) & (timeun = 3) & (timeal
= 0) & (timer = 0) & (user = 1) & (alarm = 0)
```

```
2) Add the transition (light = n) & (timeun = 3) & (timeal = 0) &
(timer = 0) & (user = 1) & (alarm = 0) & (((lux = 2))) : c; into
case expression's first position of variable light

3) Remove the transition lux = 2 : n; into case expression's of
variable light

4) Add the transition (light = n) & (timeun = 3) & (timeal = 3) &
(timer = 1) & (user = 1) & (alarm = 0) & (((lux = 1)) | ((lux = 2)))
: c; into case expression's first position of variable light

5) Add the transition (light = c) & (timeun = 0) & (timeal = 0) &
(timer = 1) & (user = 1) & (alarm = 0) & (((lux = 1))) : c; into
case expression's first position of variable light

6) Add the transition (light = n) & (timeun = 3) & (timeal = 0) &
(timer = 0) & (user = 1) & (alarm = 0) & (((lux = 1))) : c; into
case expression's first position of variable light

-- with the following side effects:

-> there is no side effects!
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Podemos ver na figura 4.4 como o novo modelo ficou. O espaço de estados foi reduzido a apenas dez. Oito estados foram mantidos, oito eliminados e dois novos foram criados. Note que cinco estados inconsistentes no modelo original (I, A, C, F, G) foram os responsáveis pelas seis mudanças propostas, cada um por uma, com exceção do estado F, pois o mesmo foi alcançado duas vezes durante a execução do algoritmo. É importante ressaltar que essa é apenas uma das possibilidades de revisão, pois há a escolha aleatória de modelos (estados) quando se evoca a função **Revisao**.

Podemos verificar abaixo como o código na linguagem do NuSMV ficaria, caso o usuário optasse pelas modificações sugeridas através no nosso algoritmo.

```
...

ASSIGN
init(light) := n; init(timeun) := 3; init(timeal) := 0;
init(timer) := 0; init(user) := 1; init(alarm) := 0;

next(light) := case
  (light = n) & (timeun = 3) & (timeal = 0) &
  (timer = 0) & (user = 1) & (alarm = 0) & (((lux = 1))) : c;
  (light = c) & (timeun = 0) & (timeal = 0) &
  (timer = 1) & (user = 1) & (alarm = 0) & (((lux = 1))) : c;
```

```
    (light = n) & (timeun = 3) & (timeal = 3) &
    (timer = 1) & (user = 1) & (alarm = 0) &
    (((lux = 1)) | ((lux = 2))) : c;
    (light = n) & (timeun = 3) & (timeal = 0) &
    (timer = 0) & (user = 1) & (alarm = 0) & (((lux = 2))) : c;
    lux = 1 & light = c : d;
    timeun < 3 & user : c;
    user : d;
    timeun = 3 : n;
    timeal < 3 & alarm : d;
    timeal = 3 & alarm : n;
    1 : light;
  esac;

  ...
```

A mudança no estado inicial e a remoção de transições possuem pouca influência sobre a legibilidade do código. O principal gargalo ocorre quando há acréscimo de transições. Se um sistema possuir muitas variáveis, a pré-condição (formada pela conjunção dessas variáveis) pode se tornar grande demais, tornando ilegível o novo código. Mas na maioria das vezes, essa pré-condição pode ser reduzida, pois alguns valores das variáveis ocorrem em um único estado. Além disso, podemos fundir transições (linhas) no código, melhorando bastante a sua legibilidade.

Manualmente podemos reduzir o já revisado código do nosso exemplo. Note que é possível fundir a nossa quarta transição com a primeira, acrescentando apenas uma disjunção na variável lux e criando uma nova transição "(light = n) & (timeun = 3) & (timeal = 0) & (timer = 0) & (user = 1) & (alarm = 0) & (lux = 1 | lux = 2) : c;". Além disso, é possível reduzir o tamanho das pré-condições, uma vez que no novo modelo do sistema só no estado I' temos (light = n) & (timeal = 0), somente no estado B temos (light = c) & (timer = 1) e apenas no estado E temos (light = n) & (timer = 1). O código revisado ficaria bem razoável com essas reduções, como podemos ver a seguir.

```
  ...

  ASSIGN
  init(light) := n; init(timeun) := 3; init(timeal) := 0;
  init(timer) := 0; init(user) := 1; init(alarm) := 0;

  next(light) := case
    (light = n) & (timeal = 0) & (((lux = 1)) | ((lux = 2))) : c;
    (light = c) & (timer = 1) & (((lux = 1))) : c;
    (light = n) & (timer = 1) & (((lux = 1)) | ((lux = 2))) : c;
    lux = 1 & light = c : d;
```

```
    timeun < 3 & user : c;
    user : d;
    timeun = 3 : n;
    timeal < 3 & alarm : d;
    timeal = 3 & alarm : n;
    1 : light;
  esac;

  ...
```

4.9 Restrições

Como a linguagem do NuSMV é bastante expressiva, para um perfeito funcionamento do nosso algoritmo, é necessário impormos algumas restrições em relação à descrição do sistema. São elas :

1. As variáveis do sistema só podem ser booleanas e escalares.
2. Todas as variáveis devem possuir a definição de sua transição, ou seja, todas devem possuir a expressão next.
3. Todas as expressões next devem possuir internamente uma expressão case, como por exemplo no formato abaixo:

```
next(x) := case
  y : 1;
  z : 0;
  1 : x;
esac;
```

4. Todas as transições devem ser determinísticas.
5. O sistema deve possuir um único estado inicial.
6. A propriedade a ser verificada dever ser uma fórmula temporal simples, com a parte interna sendo uma expressão simples e consistente.
7. O sistema modelado deve possuir apenas o processo principal.

No item 1 optamos por trabalhar com variáveis booleanas e escalares, pois as mesmas podem facilmente ser adaptadas para a lógica proposicional clássica, facilitando a nossa abordagem de revisão baseada em comparação de modelos (estados). Além disso, são os tipos de variáveis mais utilizadas.

Impomos as restrições dos itens 2 e 3 porque se uma variável não está associada a uma expressão next e ela for referenciada no processo de revisão, não há como saber

o que remover ou onde a transição deve ser acrescentada. Decidimos pelas expressões case por serem as mais usadas no interior das expressões next.

Na linguagem do NuSMV transições não-determinísticas ocorrem se não definirmos a transição de alguma variável e/ou tivermos várias pós-condições para uma mesma pré-condição. Em virtude da imposição do item 2, aliado ao fato de podermos eliminar estados consistentes se eliminarmos uma transição do tipo [pré-condição:pós-condição1 ou pós-condição2], resolvemos colocar a restrição do item 4.

Se tivermos dois ou mais estados iniciais o processo de revisão pode gerar mudanças distintas para cada um deles, muitas vezes incompatíveis entre si. Por isso a restrição do item 5.

O item 6 é necessário pois vamos sempre precisar ter um modelo, que será a nossa crença recebida, para se fazer a revisão baseada em comparação de modelos (estados). A única maneira de garantir que exista sempre um modelo para ser comparado é restringindo a expressão interna ao operador temporal a ser uma fórmula proposicional consistente. Se tivermos fórmulas temporais encadeadas não podemos garantir que sempre existirá um modelo e, mesmo quando existir, esse modelo pode ser extinto quando fizermos as alterações na descrição.

A imposição do item 7 se deve ao fato da revisão poder propor diferentes mudanças para processos semelhantes. Se tivermos dois processos com a mesma descrição, é possível que o algoritmo sugira duas mudanças distintas para o estado inicial, o que é incompatível. Essa restrição ocorre pelo mesmo problema relatado no item 5.

4.10 Análise Comparativa

Para finalizar, faremos uma breve análise comparativa entre o arcabouço teórico e o nosso algoritmo de revisão de crenças. A principal diferença ocorre em relação ao conjunto de crenças a ser revisado. Os postulados AGM se referem a um único conjunto de crenças com vários mundos possíveis. No nosso caso, temos vários conjuntos de crenças com um único mundo possível.

Para mostrarmos que a revisão proposta sob um estado segue a teoria clássica da revisão de crenças, basta a encaixarmos em alguma construção descrita no capítulo 2. Na seção 4.4 utilizamos um sistema de esferas que pode ser comparado a um sistema de esferas que segue os postulados.

Primeiro é preciso verificar se um estado pode ser a esfera central do sistema. Como não há restrições em relação ao número de mundos possíveis do conjunto K, então não há problema em $[K]$ se restringir a um único modelo (estado). É preciso também garantir que exista uma ordem total entre as esferas. Na nossa implementação utilizamos a função d como critério. Sejam B e C modelos (estados) e E o modelo (estado) da

esfera central, podemos verificar que se $d(E, B) < d(E, C)$, então a esfera com o modelo (estado) E está contida na esfera que possui os modelos (estados) B, que por sua vez está contida na esfera com os modelos (estados) C, ou seja, quanto menor a distância d, mais informações sobre o estado são preservadas. Como a função d devolve um número natural e a relação de ordem entre os números naturais é total, então esse critério pode ser usado na ordenação das esferas.

Como estamos lidando com um número finito de variáveis, a nossa maior esfera da ordenação (M) abrange todos os possíveis estados. Isso garante que os modelos (estados) que estão entre a penúltima e a última esfera sejam aqueles que não possuem nenhuma informação sobre o estado central. Por fim, basta verificar que se uma esfera intercepta o sistema de esferas, então há uma menor esfera no sistema que a intercepta. Novamente, como estamos lidando com um número finito de variáveis, qualquer esfera que representa um conjunto de estados (mundos possíveis) está contida na esfera M e portanto intercepta pelo menos uma esfera do sistema. Como existe uma ordenação, então existe uma menor esfera que a intercepta.

Para concluírmos, é importante ressaltar duas peculiaridades da nossa implementação em relação ao sistema de esferas. A primeira é que não permitimos a revisão por uma crença consistente com o conjunto original. Ainda se permitíssemos, haveria diferença. Como no sistema de esferas geralmente o conjunto de crenças original possui vários mundos possíveis, uma revisão desse tipo leva a uma refinamento desses mundos. No nosso caso, como há apenas um modelo (estado), uma revisão assim não produziria efeito algum. A segunda diferença está no tratamento da crença recebida. Na nossa implementação impomos que a crença recebida tenha pelo menos um modelo, ou seja, não fazemos revisão por crenças inconsistentes.

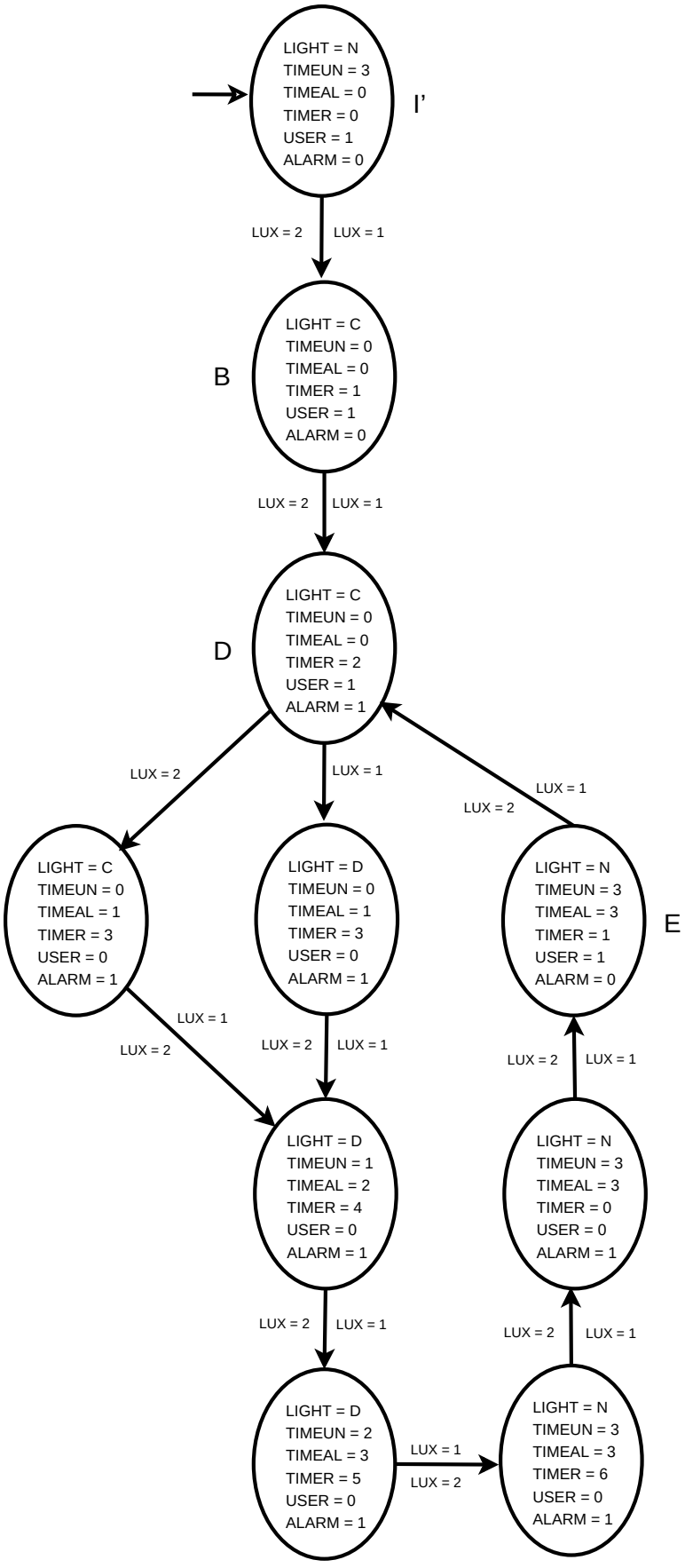

Figura 4.4: Modelo revisado do exemplo

Capítulo 5

Conclusões

Um dos maiores gargalos da engenharia de software é, sem dúvida, a revisão e manutenção da especificação do modelo do sistema. O nosso estudo teve como objetivo mostrar que a revisão de crenças pode auxiliar os engenheiros nessa difícil tarefa.

Mostramos que através de uma implementação de revisão de crenças no verificador de modelos NuSMV é possível eliminar as inconsistências dos modelos de softwares. O nosso algoritmo provê uma alternativa ao usuário sobre o que deve ser feito para um dado modelo se torne consistente com a fórmula temporal. Essa alternativa proposta não tem a pretensão de ser a melhor ou mais simples alteração na descrição do sistema, mas sim dar dicas ao desenvolvedor sobre como eliminar os erros.

A área de revisão de crenças possui poucas aplicações práticas implementadas. A nossa contribuição nessa área aumenta esse número. Até onde temos conhecimento, essa é a primeira implementação de revisão de crenças em uma ferramenta utilizada para modelagem formal. Nos testes realizados pudemos comprovar a sua eficácia para a manutenção da consistência dos modelos de softwares.

A contribuição do nosso trabalho na área de métodos formais para modelagem de sistemas se dá também na parte prática. Acreditamos que até o presente momento não exista nenhuma ferramenta de verificação formal de modelos que possua suporte para tratamento automático de inconsistências. Mais especificamente, acreditamos que não há nenhuma funcionalidade nas ferramentas conhecidas para tratar automaticamente inconsistências de fórmulas temporais descritas em CTL com modelos de softwares.

Esperamos que o nosso trabalho seja uma primeira integração entre essas duas áreas da computação e que, apesar do processo de modelagem ser algo intrínseco a mente humana, a área de revisão de crenças seja cada vez mais útil no auxílio ao desenvolvimento de sistemas.

5.1 Trabalhos Relacionados

A idéia de se usar revisão de crenças como uma ferramenta de revisão e manutenção da especificação de um sistema não é nova. Zowghi et al [24] apresentam um arcabouço para modelar e raciocinar sobre a evolução da especificação. Esse arcabouço vê a especificação como uma teoria de alguma lógica não-monotônica, cuja evolução envolve o mapeamento de uma teoria a outra. Eles implementaram operadores da revisão de crenças no sistema THEORIST (sistema de raciocínio não-monotônico) e mostram com exemplos como se faz a evolução e manutenção da especificação.

Gervasi et al [25] apresentam um protótipo chamado CARL que transforma os requisitos (em linguagem natural controlada) em lógica proposicional e aplica revisão de crenças. O resultado é transformado novamente em linguagem natural. A idéia é baseada no trabalho de Zowghi citado no parágrafo anterior, com o diferencial que a lógica fica completamente escondida do usuário.

Já Macnish et al [26] mostram o uso de uma proposta de revisão de crenças iterativa, chamada de maxi-adjustment, para a manutenção do modelo de especificação. Elas usam um arcabouço chamado *Goal Structured Analysis* que se baseia na decomposição da meta principal em sub-metas. Elas mostram que esse arcabouço é o ideal para se aplicar revisão de crenças baseado no epistemic entrenchment, uma vez que a meta é mais importante que as sub-metas, não violando o postulado **EE2**. Apesar de não possuir uma implementação, elas mostram como deveria funcionar a aplicação através de um exemplo de uma especificação para um sistema de freios de avião.

Rodrigues et al [21] apresentam a idéia de revisão de crenças clusterizadas para solucionar o problema. Basicamente, as crenças são agrupadas de acordo com algum critério e esses grupos ordenados parcialmente ou totalmente. A revisão é feita abandonando-se as crenças dos grupos que estão nas primeiras posições da ordenação. A idéia é derivada do epistemic entrenchment, só que com uma abstração maior. Eles implementaram um protótipo que usa lógica clássica proposicional (DNF) como representação da especificação e o testaram com um exemplo de especificação de um sistema de controle de iluminação (LCS).

Apesar de serem propostas bem interessantes e aplicáveis, ainda estão distantes da área prática de engenharia de software, pois usam arcabouços e linguagens matemáticas que são desconhecidas até mesmo para pessoas de desenvolvimento formal de software. Além disso, essas propostas se focam apenas nos requisitos, sem nenhuma preocupação em relação ao modelo de funcionamento do sistema.

Os estudos mais recentes se focam na manutenção de modelos de especificações em linguagens e ferramentas que são usadas para desenvolvimento formal de software, em especial os verificadores automáticos de modelos.

Em sua tese de doutorado, Gorogiannis [27] propõe um algoritmo que gera um autômato a partir de fórmulas ACTL (sub-conjunto da CTL), transforma esse autômato em código NuSMV e o junta ao código original do sistema. A composição síncrona realizada pelo NuSMV se encarrega de fazer a intersecção dos dois autômatos, tornando a fórmula verdadeira. Essa abordagem se assemelha a nossa devido ao fato de propor também alterações no código SMV que descreve o sistema. A diferença é que essa abordagem foca exclusivamente na fórmula temporal, ignorando completamente a descrição do modelo feita pelo usuário. Além disso, a lógica ACTL é mais restrita, não permitindo por exemplo expressar que no próximo estado uma variável p é verdadeira (EX p). Infelizmente não tivemos acesso à implementação para realizar um teste comparativo. Mas no exemplo apresentado por Gorogiannis com a fórmula AG ($p \rightarrow AF\ q$), podemos verificar que são criadas 3 novas variáveis, 44 novas transições e 7 novos estados iniciais, o que torna totalmente ilegível o novo código.

Já Ding [28] propõe um atualizador de modelos, onde as variáveis nos estados são atualizadas minimamente para se satisfazer uma fórmula CTL. Essa abordagem é semelhante a nossa no sentido de propor mudanças no modelo definido pelo usuário. Mas, além de trabalhar apenas com variáveis booleanas, essa abordagem peca por não ter sido implementada em nenhuma linguagem da área de desenvolvimento formal de software. Ding desenvolveu um protótipo com uma linguagem própria para descrever o estado inicial, as transições entre estados e a fórmula temporal. Infelizmente também não tivemos acesso a esse protótipo, mas o estudo de caso apresentado mostra claramente que as transições não são pautadas na relação de pré-condição com pós-condição, algo bastante comum em sistemas de estados finitos.

O trabalho de Harris [29] mostra que a área de *feature integration* (acréscimo de novas características a um sistema) pode ser comparada a área de atualização de crenças. Em sua abordagem, Harris transforma a descrição do sistema na linguagem SMV em lógica proposicional (se tornando a base de conhecimento), bem como a nova característica a ser adicionada (também descrita em SMV), e mostra que o modelo gerado pela atualização corresponde ao modelo do sistema com a característica incorporada. Essa é uma abordagem diferente da nossa, pois a crença recebida é uma parte da descrição, enquanto que propomos que a crença recebida seja a fórmula temporal. Na verdade, Harris faz apenas uma comparação entre essas duas áreas e relata possíveis trabalhos futuros, não havendo uma proposta direta sobre modificações no modelo do sistema, na nova característica ou na fórmula temporal para que a consistência seja estabelecida.

5.2 Trabalhos Futuros

O NuSMV e sua linguagem SMV é apenas um dos vários métodos/ferramentas de modelagem formal de sistemas. A disseminação da idéia de revisão de crenças em outros produtos e comunidades de desenvolvimento de software deve produzir bons frutos. Em especial, seria interessante e desafiador uma implementação desse tipo em ferramentas ligadas aos métodos B e Z, devido ao fato de serem mais conhecidas e de possuírem uma lógica bem mais expressiva.

Em relação a implementação atual, podemos destacar algumas melhorias que podem ser feitas no futuro. Uma delas é estender o algoritmo para realizar a revisão a partir de fórmulas temporais mais complexas. Acreditamos que para isso a revisão deva ser feita na abordagem *bottom-up*, primeiro revisando as sub-fórmulas mais simples até se chegar a fórmula original, talvez aproveitando o próprio algoritmo de verificação baseado na teoria do ponto fixo.

Um outra possível melhoria seria a definição de uma mudança mínima no modelo. Seria preciso definir qual o melhor critério para a mudança, se o número de estados diferentes entre os modelos ou o número de mudanças na descrição do sistema. Para isso seria necessário encontrar todas as alternativas possíveis de revisão, o que pode ser bastante ineficiente. A definição desse critério ainda passa pelo impacto das mudanças na visão do usuário, se é algo que faz sentido ou não. Definido o critério, ainda seria preciso garantir que os algoritmos de revisão o seguissem, o que a princípio não parece ser uma tarefa das mais fáceis. Ainda nessa frente, um grande desafio seria encontrar um meio da revisão provocar a menor quantidade possível de efeitos colaterais indesejáveis e a maior quantidade possível de efeitos desejáveis.

A função distância d usada no cálculo dos estados mais próximos do estado inconsistente pode ser alvo de melhorias também, uma vez que há a escolha aleatória quando dois ou mais estados possuem a mesma distância. Por exemplo, suponha que a revisão revele dois estados como possíveis alternativas a um estado inconsistente cuja variável timer possui valoração zero: um com a variável timer com valor 1 e outro com valor 5. A escolha mais intuitiva é pelo primeiro devido à proximidade numérica, mas a nossa implementação escolhe aleatoriamente um dos dois estados.

Seria interessante também analisar a possibilidade da expansão da revisão de crenças para lidar com informações incompletas, isto é, revisar vários estados simultaneamente. Além disso, poderíamos expandir o algoritmo para incorporar atualização de crenças. Katsuno e Mendelzon [30] formalizaram a diferença entre atualizar e revisar uma base de conhecimento. Se pensarmos em termos de mundos possíveis (como no sistema de esferas), na atualização, cada modelo da base de conhecimento representa um possível estado do mundo, sendo portanto necessário considerá-los individualmente. A atualização é utilizada para descrever mudanças sobre um mundo dinâmico onde o estado

atual não é totalmente conhecido. Por outro lado, a revisão descreve mudanças quando se descobre algo novo ou um erro das crenças sobre um mundo estático. No modelo do sistema poderíamos aplicar atualização nos estados iniciais, uma vez que cada um deles representa um possível modelo (estado) do mundo, sendo necessário analisá-los separadamente.

O nosso algoritmo foca nas mudanças das linhas das expressões case de cada uma das variáveis e na mudança do estado inicial. Seria interessante se analisar a possibilidade de outros tipos de mudanças, como por exemplo mudar apenas as pré-condições ou a pós-condição de uma dada linha. Uma importante expansão seria eliminar a restrição na qual todas as variáveis devem possuir transições (abrindo espaço para sistemas não-determinísticos), assim como abolir o uso apenas de transições com expressões case. Poderíamos também estender o nosso algoritmo para aceitar mais de um estado inicial. Para isso, primeiro precisaríamos encontrar um meio de reduzir o tamanho das pré-condições das transições acrescentadas, que nesse caso seria as disjunções das conjunções dos valores das variáveis em cada estado.

Ao invés de nos concentrarmos apenas em mudanças na descrição, poderíamos sugerir o enfraquecimento da fórmula temporal a ser verificada. Por exemplo, suponha que o NuSMV descubra que uma dada fórmula AF α é falsa. Uma extensão do nosso algoritmo poderia encontrar e sugerir que EF α ou AX α são verdadeiras. Uma implementação trivial seria testar α com todos os operadores temporais e encontrar aquelas fórmulas que são verdadeiras.

A redução automática do código NuSMV revisado seria também bastante útil para o processo, uma vez que deixaria o código bem mais legível. A opção de refatorar o código para prepará-lo mais adequadamente ao processo de revisão seria uma outra linha de pesquisa futura. O suporte a modelos com dois ou mais processos é uma outra extensão desejável, bem como a introdução de uma simulação onde o desenvolvedor pudesse passo-a-passo escolher o modelo revisado.

Apêndice A

Sintaxe do NuSMV

As principais construções da sintaxe do NuSMV que utilizamos são detalhadas aqui. Para maiores informações consulte a documentação oficial no site abaixo

`http://nusmv.irst.itc.it/NuSMV/userman/v23/nusmv.pdf`

Expressões Simples

```
simple_expr ::
  atom                              ;; a symbolic constant
  | number                          ;; numeric constant
  | TRUE                            ;; the boolean constant
  | FALSE                           ;; the boolean constant
  | var_id                          ;; a variable identifier
  | ( simple_expr )
  | ! simple_expr                   ;; logical not
  | simple_expr & simple_expr       ;; logical and
  | simple_expr | simple_expr       ;; logical or
  | simple_expr xor simple_expr     ;; logical exclusive or
  | simple_expr -> simple_expr      ;; logical implication
  | simple_expr <-> simple_expr     ;; logical equivalence
  | simple_expr = simple_expr       ;; equality
  | simple_expr != simple_expr      ;; inequality
  | simple_expr < simple_expr       ;; less than
  | simple_expr > simple_expr       ;; greater than
  | simple_expr <= simple_expr      ;; less than or equal
  | simple_expr >= simple_expr      ;; greater than or equal
  | simple_expr + simple_expr       ;; integer addition
  | simple_expr - simple_expr       ;; integer subtraction
  | simple_expr * simple_expr       ;; integer multiplication
  | simple_expr / simple_expr       ;; integer division
  | simple_expr mod simple_expr     ;; integer remainder
```

```
  | set_simple_expr                ;; a set simple_expression
  | case_simple_expr               ;; a case expression
```

Expressões Cases

As expressões cases geralmente são utilizadas para relacionar as pré-condições com as pós-condições

```
case_simple_expr ::
  case
    simple_expr : simple_expr
    simple_expr : simple_expr
    ...
  esac
```

Expressões Set

```
set_expr ::
  { set_elem , ... , set_elem }     ;; set definition
  | simple_expr in simple_expr      ;; set inclusion test
  | simple_expr union simple_expr   ;; set union
set_elem :: simple_expr
```

Expressões Next

As expressões next são utilizadas para expressar as transições entre estados

```
next_expr ::
  simple_expr
  | next ( simple_expr )            ;; next value of an expression
```

Expressões CTL

As expressões CTL são utilizadas para definir as propriedades a serem verificadas

```
ctl_expr ::
  simple_expr                       ;; a simple boolean expression
  | (ctl_expr )
  | ! ctl_expr                      ;; logical not
  | ctl_expr & ctl_expr             ;; logical and
  | ctl_expr | ctl_expr             ;; logical or
  | ctl_expr xor ctl_expr           ;; logical exclusive or
  | ctl_expr ->  ctl_expr           ;; logical implies
  | ctl_expr <-> ctl_expr           ;; logical equivalence
  | EG ctl_expr                     ;; exists globally
  | EX ctl_expr                     ;; exists next state
  | EF ctl_expr                     ;; exists finally
  | AG ctl_expr                     ;; forall globally
```

```
| AX ctl_expr                    ;; forall next state
| AF ctl_expr                    ;; forall finally
| E [ ctl_expr U ctl_expr ]      ;; exists until
| A [ ctl_expr U ctl_expr ]      ;; forall until
```

Apêndice B

Implementação

A implementação dos algoritmos descritos nesse trabalho foi realizada sob a versão 2.3.1 do NuSMV e pode ser encontrada no endereço abaixo com todas as instruções para instalação e uso. Para facilitar a integração com o NuSMV, utilizamos também o **C** como linguagem de programação. A ferramenta de desenvolvimento usada foi a IDE Eclipse 3.2 com o plugin CDT (`http://www.eclipse.org/cdt/`). Toda a implementação produziu cerca de 2 mil linhas de código.

`http://www.ime.usp.br/~thiago/mestrado/Br-NuSMV-2.3.1.tgz`

Apesar de lançada recentemente, a versão 2.4.0 possui vários relatos de bugs e queda de performance feitas pelos seus usuários, o que fez a equipe de desenvolvimento prometer um patch para o mais breve possível. Além disso, quando começamos a nossa implementação a nova versão ainda não tinha sido lançada, o que nos fez continuar na versão anterior. Mas acreditamos que, assim que estiver estabilizada, uma adaptação dos algoritmos para essa nova versão não será muito complicada.

O uso ou não dos nossos algoritmos depende da escolha do usuário. Essa escolha é definida pela opção "-br" (belief revision), passada como um dos possíveis argumentos ao NuSMV via linha de comando. Como os algoritmos não permitem interação com o usuário, o suporte a revisão de crenças só pode ser executado em modo automático (batch mode). Resolvemos colocar a revisão como mais uma opção porque assim o usuário, caso não tenha interesse em utilizá-la, pode usufruir normalmente de todas as outras opções originais do NuSMV.

A análise da opção pela revisão é feita no final do algoritmo de verificação das fórmulas CTL e só acontece se uma dada fórmula é inconsistente com o modelo. Se a fórmula é falsa e o usuário optou pelo suporte a revisão, então o nosso algoritmo principal (**Revisador_Modelos**) é evocado.

Antes de relatarmos outros detalhes da implementação, é importante destacarmos uma das principais estruturas de dados usada pelo NuSMV e que nos foi bastante útil. Essa estrutura é o *node* (que a partir daqui chamaremos apenas de nó), que é uma implementação semelhante a uma s-expressão do Lisp (expressões simbólicas a serem avaliadas). Podemos compará-lo a uma árvore binária, com um ponteiro para a esquerda, outro para a direita e com uma variável de conteúdo que armazena o tipo do nó. Cada um dos seus ponteiros pode apontar para uma variável inteira, para uma cadeia de caracteres, para um outro nó ou para qualquer outra estrutura. Veja abaixo a estrutura de um nó, bem como as estruturas usadas por ele.

```
struct string_ {
  struct string_ *link;
  char *text;
};

union node_val {
  int inttype;
  struct node *nodetype;
  struct string_ *strtype;
  void *bddtype;
};

struct node {
  struct node *link;
  short int type;
  short int lineno;
  node_val left;
  node_val right;
};
```

A estrutura de nó tem como principal utilidade armazenar em formato cadeia de caracteres (strings) o resultado proveniente do parser realizado no arquivo de entrada. Além disso, um nó armazena o tipo relacionado a cada string. Com isso podemos descobrir facilmente se um certo string é uma declaração de transição, se é a definição de um estado inicial, se é uma fórmula CTL, se é apenas o valor de uma variável, etc. Como descrito no capítulo 3, o NuSMV usa a abordagem simbólica para representar estados e transições através de BDDs e para isso utiliza um algoritmo que transforma um nó em um BDD. Não entraremos em detalhes sobre esse algoritmo pois está fora do nosso escopo de estudo.

Voltemos agora ao nosso algoritmo principal. Como temos a fórmula CTL armazenada em um nó, conseguimos descobrir exatamente com qual tipo de fórmula estamos lidando. Assim a implementação do nosso algoritmo **Revisador_Modelos** se tornou trivial.

Os algoritmos utilizam uma pilha (**P**) e uma lista de adjacências (**AdjList**) para a busca em profundidade. Como no NuSMV todos os estados estão representados através de BDDs, a nossa pilha é um vetor onde cada posição armazena um BDD e a nossa lista de adjacências é um vetor onde cada posição armazena uma lista de BDDs. Essa lista de BDDs faz parte de nossa implementação, bem como os operadores mais usuais de listas. Sua estrutura pode ser visualizada a seguir.

```
typedef struct BddList_TAG {
  bdd_ptr bdd;
  struct BddList_TAG *prox;
} BddList;
```

O tamanho da pilha é o número de estados possíveis, enquanto o tamanho da lista de adjacências é calculado de acordo com o número de estados alcançáveis a partir do inicial. O NuSMV possui funções que calculam esses números.

Uma função que é utilizada em todos os algoritmos e é sem dúvida o núcleo da nossa implementação é a função **Revisao**. Vamos agora analisar as suas sub-funções e as estruturas de dados que são utilizadas por elas. Dada uma fórmula proposicional (expressão simples), o NuSMV possui uma função que calcula um BDD que representa todos os modelos possíveis dessa fórmula. Assim, para se saber se uma fórmula proposicional é inconsistente com um dado modelo (estado), basta aplicarmos a operação BDDAnd para descobrir se esse estado faz parte do conjunto dos modelos possíveis da fórmula. Se não fizer, há a inconsistência. A nossa função **Inconsistente** faz exatamente isso.

O NuSMV também permite comparar diretamente BDDs e nos casos onde um BDD representa um estado é possível comparar os valores das variáveis. Assim, conseguimos rapidamente saber se dois estados são idênticos. E caso não sejam, podemos descobrir quais as variáveis possuem valores distintos. Devido a essa facilidade, aliada ao fato do NuSMV também computar os modelos possíveis de uma dada fórmula, optamos por utilizar revisão de crenças baseada em comparação de modelos. O cálculo do conjunto Δ da definição de revisão no capítulo 4 se torna simples. Para cada um dos possíveis modelos de uma dada fórmula calculamos a quantidade de variáveis com valores distintos em relação ao estado a ser revisado. Os modelos que possuem o menor valor fazem parte do conjunto Δ.

Uma outra função é a **Reconstroi_Trans**. Para cada valor de variável diferente entre dois estados tenta-se primeiro remover transições, usando a função **Pode_Remover**. Caso não seja possível, adicionamos transições com a função **Adiciona_Trans**. Antes de descrevermos cada uma dessas duas funções é preciso apresentarmos as estruturas utilizadas por ela e as dificuldades encontradas.

Um ponto crítico nessa etapa foi a manipulação das transições. Como o NuSMV transforma todas as transições em BDDs logo após a leitura do arquivo de entrada, te-

ríamos que encontrar um meio de modificar os BDDs de tal forma que essa modificação representasse a remoção ou adição das transições desejadas. Relatado esse problema na lista de discussão da ferramenta, a equipe de desenvolvimento do NuSMV nos recomendou fortemente que não tentássemos modificar os BDDs. A alternativa sugerida foi modificar o nó que representa as transições e recriar os BDDs a cada mudança na descrição.

Ao manipularmos as transições no nó que as armazena, tivemos que lidar com um outro problema: como encontrar a posição exata da modificação se esse nó armazena todas as transições de todas as variáveis? Para solucionar isso criamos um vetor (**VarTrans**), do tamanho da quantidade de variáveis, que armazena em cada posição o sub-nó responsável pelas transições de uma variável. Esse vetor é preenchido antes da transformação do nó em BDDs. Assim, para cada variável, temos o mapeamento das suas transições, garantindo que as funções **Pode_Remover** e **Adiciona_Trans** manipulem as transições corretas.

Na função **Pode_Remover** é preciso verificar se há alguma, dentre as transições da variável selecionada, que modifique ou mantenha o seu valor de acordo com o estado revisado. Para isso percorremos o sub-nó e, para cada transição, transformamos em BDD a sua pré-condição e verificamos através de uma operação BDDAnd se a mesma é aplicável no estado antecessor. Se tal transição foi encontrada, então percorremos novamente o sub-nó até essa transição e realizamos o mesmo procedimento anterior a fim de eliminarmos as transições anteriores que são aplicáveis no estado antecessor.

Como relatamos no capítulo 4, antes da eliminação de uma transição ainda é necessário analisar se a mesma já não foi utilizada anteriormente. Para isso criamos um vetor (**UsadasTrans**), do tamanho da quantidade de variáveis, que armazena em cada posição, através de uma lista de nós, as transições já utilizadas por uma dada variável. Esse armazenamento é feito em tempo de execução e é de responsabilidade da função **Trans_Usadas**.

Se existe uma transição que mantém ou modifica o valor da variável de acordo com o seu valor no estado revisado e essa transição nunca foi usada, então a removemos do sub-nó armazenado no vetor **VarTrans** e recriamos os BDDs que representam as transições. Caso contrário a função **Adiciona_Trans** é evocada.

A princípio, o acréscimo de transições no sub-nó armazenado no vetor **VarTrans** nos parecia mais simples por não haver a necessidade de percorrê-lo, bastando apenas inserir a transição no seu início. Definimos na seção 4.5 que a transição a ser inserida deve ter como pré-condição a conjunção formada pelas variáveis comuns e variáveis de entrada do estado antecessor. Portanto, precisaríamos apenas transformar o estado antecessor e suas entradas, que são BDDs, em nós. Nesse ponto tivemos mais um problema. O NuSMV possui uma função interna que transforma nós em BDDs, mas nenhuma que transforme BDDs em nós. A equipe de desenvolvimento nos socorreu mais uma vez.

Eles criaram uma função que faz essa transformação e a disponibilizaram através de um patch. Com esse patch o acréscimo de transições se tornou uma tarefa simples.

As demais funções são: **Recria_AdjList**, **Imprime_Inicial**, **Imprime_Remocao** e **Imprime_Adicao**. A primeira recebe a nova máquina de estados finitos proveniente da revisão e calcula mais uma vez, para cada estado, os seus estados sucessores através de funções nativas do NuSMV. As outras três se utilizam das funções padrões para impressão da linguagem C e funções internas do NuSMV para a impressão de BDDs e nós.

Por fim, uma outra função importante é a **Efeitos_Colaterais**, que verifica a interferência de uma revisão em outras fórmulas, sendo evocada ao final de todo processo de revisão. O NuSMV armazena em um vetor de nó todas fórmulas a serem verificadas. Essa função percorre esse vetor e, para cada fórmula (com exceção da que acabou de realizar a revisão), analisa o seu valor (verdadeiro ou falso) utilizando os algoritmos de verificação do NuSMV, em relação ao modelo original do sistema e em relação ao modelo revisado. Se os valores divergirem, então imprime ao usuário um aviso sobre o efeito colateral.

A seguir temos uma lista dos arquivos usados, com uma breve explicação sobre as modificações realizadas:

- compile/compileFsmMgr.c : esse arquivo possui as funções nativas do NuSMV para a criação da máquina de estados finitos. Incorporamos uma função que separa e armazena no vetor **VarTrans** os sub-nós responsáveis pelas transições de cada variável.

- compile/compileStruct.c : esse arquivo possui funções que evitam a reprocessamento da máquina de estados finitos "sujando" algumas variáveis. Incorporamos algumas funções que "limpam" essas variáveis a fim de nos permitir tal reprocessamento.

- bdd/enc/BddEnc.c : esse arquivo contém as funções de manipulação de BDDs. As principais funções e estruturas descritas no capítulo 4 foram criadas nele.

- bdd/enc/BddEncCache.c : aqui temos funções que otimizam operações relacionadas aos BDDs. Uma modificação foi feita nesse arquivo para evitar que os BDDs das antigas transições fossem utilizados indevidamente.

- fsm/sexp/SexpFsm.c : esse arquivo possui funções que manipulam nós. Acrescentamos uma função que modifica o nó representante do estado inicial para posterior transformação em BDD.

- mc/mcMc.c : o arquivo contém as rotinas nativas do NuSMV responsáveis pela verificação de fórmulas CTL. A nossa função principal **Revisador_Modelos** foi acrescentada aqui.

- opt/optCmd.c : contém funções que controlam as opções passadas pelo usuário ao NuSMV via linha de comando. Acrescentamos aqui uma função com a opção de suporte a revisão de crenças.

- prop/propProp.c : esse arquivo possui rotinas para o tratamento de fórmulas temporais. Eliminamos uma função que evita a análise duplicada de uma fórmula, o que é necessário quando verificamos os efeitos colaterais.

- sm/smMain.c : função principal do NuSMV. Chamamos aqui a função incorporada ao arquivo optCmc.c para a realização da revisão.

- sm/smMisc.c : possui funções auxiliares para inicialização correta do NuSMV. Incorporamos as funções que inicializam as estruturas que foram criadas.

- utils/NodeList.c : esse arquivo possui funções para manipulação de listas. A lista de BDDs da nossa implementação foi criada aqui.

Apêndice C

Resultados dos Testes

Além do teste no modelo descrito no capítulo 4, outros foram realizados a fim de verificar a validade da revisão em todos os operadores temporais, bem como nos dois tipos possíveis de modelo (com ou sem variáveis de entrada). Como se tratam apenas de testes não nos preocuparemos se a descrição ou a verificação da fórmula temporal fazem ou não sentido. Os modelos testados estão no seguinte endereço:

`http://www.ime.usp.br/~thiago/mestrado/Br-NuSMV-2.3.1.tgz`

O modelo criado com variáveis de entrada se encontra no arquivo model-withinput.smv localizado na pasta /examples/br/ da implementação e sua descrição pode ser visualizada a seguir. Esse modelo é uma criação nossa que tenta simular um sistema de controle de incêndio.

```
MODULE fcs

VAR
light : boolean;
water : boolean;
alarm : boolean;
co2   : boolean;
exfan : boolean;

IVAR
user  : boolean; -- user sensor
temp  : boolean; -- thermal sensor
smoke : boolean; -- smoke sensor
```

```
INIT
!light & !water & !alarm & !co2 & !exfan
-----------------------------------------------------------------------
MODULE flip_light (s)

ASSIGN
next(s.light) := case
  s.smoke & s.user : 1 ;
  !s.temp | !s.user : 0;
  1 : s.light;
esac;
-----------------------------------------------------------------------
MODULE  flip_water(s)

ASSIGN
next(s.water) := case
  s.temp & !s.user : 1;
  s.user | !s.temp: 0;
  1 : s.water;
esac;
-----------------------------------------------------------------------
MODULE flip_alarm (s)

ASSIGN
next(s.alarm) := case
  s.temp : 1;
  !s.temp | !s.smoke | !s.user : 0;
  1 : s.alarm;
esac;
-----------------------------------------------------------------------
MODULE flip_co2 (s)

ASSIGN
next(s.co2) := case
  s.smoke & s.user : 1;
  !s.temp : 0;
  1 : s.co2;
esac;
-----------------------------------------------------------------------
MODULE flip_exfan (s)

ASSIGN
next(s.exfan) := case
  s.smoke & s.user : 1;
  !s.smoke : 0;
  1 : s.exfan;
esac;
```

```
-----------------------------------------------------------------------
MODULE main VAR

sta : fcs;
fl : flip_light(sta);
fw : flip_water(sta);
fa : flip_alarm(sta);
fc : flip_co2(sta);
fe : flip_exfan(sta);
-----------------------------------------------------------------------

SPEC
EG (!sta.co2 | sta.water)
SPEC
EX (sta.water & !sta.light)
SPEC
EF (sta.water)
SPEC
E [!sta.water U sta.alarm]
SPEC
AG ((sta.water -> sta.light) & (sta.water -> sta.co2) &
    (sta.co2 <-> sta.exfan))
SPEC
AX (sta.water -> sta.exfan)
SPEC
AF (sta.light)
SPEC
A [!sta.water U  sta.co2]
SPEC
EG ((sta.co2 <-> sta.exfan ) & sta.water)
SPEC
EX (sta.co2 & !sta.exfan)
SPEC
EF (sta.water & sta.co2 & !sta.alarm)
SPEC
E [!sta.light U  (sta.co2 & !sta.exfan)]
```

Testamos oito fórmulas temporais que são inconsistentes com o modelo e quatro fórmulas que são consistentes. O algoritmo propôs as seguintes sugestões de mudanças e avisos para cada uma das fórmulas. Modificamos manualmente o arquivo que contém a descrição e observamos realmente que as mudanças sugeridas tornaram as fórmulas verdadeiras e que os avisos sobre os efeitos colaterais estão corretos.

```
-- specification  EG (!sta.co2 | sta.water)  is true
-- specification  EX (sta.water & !sta.light)  is true
-- specification  EF sta.water  is true
-- specification  E [ (!sta.water) U sta.alarm ]   is true
```

```
-- specification  AG (((sta.water -> sta.light) &
(sta.water -> sta.co2)) & (sta.co2 <-> sta.exfan))  is false

-- but this specification can be true if you make the following
changes:

1) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 0) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 1) &
(sta.temp = 1) & (sta.smoke = 0))) : 0; into case expression's first
position of variable sta.co2

2) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 1) &
(sta.temp = 1) & (sta.smoke = 0))) : 0; into case expression's first
position of variable sta.co2

3) Remove the transition (sta.temp & !sta.user) : 1; into case
expression's of variable sta.water

4) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 0) &
(sta.temp = 1) & (sta.smoke = 0))) : 0; into case expression's first
position of variable sta.co2

5) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 1) &
(sta.temp = 1) & (sta.smoke = 0)) | ((sta.user = 0) & (sta.temp = 1)
& (sta.smoke = 0))) : 0; into case expression's first position of
variable sta.co2

6) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1))) : 0; into case expression's first
position of variable sta.exfan

7) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1))) : 0; into case expression's first
position of variable sta.exfan

8) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 0) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 0) &
(sta.temp = 1) & (sta.smoke = 0))) : 0; into case expression's first
position of variable sta.co2

9) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 0) & (sta.co2 = 1) & (sta.exfan = 1) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1))) : 0; into case expression's first
```

```
position of variable sta.exfan

-- with the following side effects:

-> specification  EX (sta.water & !sta.light)  will change from true
to false!

-> specification  EF sta.water  will change from true to false!

-> specification  AX (sta.water -> sta.exfan)  will change from
false to true!

 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  AX (sta.water -> sta.exfan)  is false

-- but this specification can be true if you make the following
changes:

1) Remove the transition (sta.temp & !sta.user) : 1; into case
expression's of variable sta.water

-- with the following side effects:

-> specification  EX (sta.water & !sta.light)  will change from true
to false!

-> specification  EF sta.water  will change from true to false!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  AF sta.light  is false

-- but this specification can be true if you make the following
changes:

1) Add the transition (sta.light = 0) & (sta.water = 1) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 1) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 1) &
(sta.smoke = 0))) : 1; into case expression's first position of
variable sta.light

2) Add the transition (sta.light = 0) & (sta.water = 1) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 0) &
(sta.smoke = 0)) | ((sta.user = 1) & (sta.temp = 0) & (sta.smoke = 0))) :
1; into case expression's first position of variable sta.light
```

```
3) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 1) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 1) &
(sta.smoke = 0))) : 1; into case expression's first position of
variable sta.light

4) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 0) &
(sta.smoke = 0)) | ((sta.user = 1) & (sta.temp = 0) & (sta.smoke = 0))) :
1; into case expression's first position of variable sta.light

5) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 1) &
(sta.temp = 1) & (sta.smoke = 0))) : 1; into case expression's first
position of variable sta.light

6) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 0) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 0) &
(sta.smoke = 0)) | ((sta.user = 1) & (sta.temp = 0) & (sta.smoke = 0))) :
 1; into case expression's first position of variable sta.light

-- with the following side effects:

-> there is no side effects!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  A [ (!sta.water) U sta.co2 ]   is false

-- but this specification can be true if you make the following
changes:

1) Remove the transition (sta.temp & !sta.user) : 1; into case
expression's of variable sta.water

2) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 0) &
(sta.smoke = 0)) | ((sta.user = 1) & (sta.temp = 0) & (sta.smoke =0))) :
1; into case expression's first position of variable sta.co2

3) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 1) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 1) &
(sta.smoke = 0)) | ((sta.user = 1) & (sta.temp = 1) & (sta.smoke = 0))) :
1; into case expression's first position of variable sta.co2
```

```
4) Add the transition (sta.light = 0) & (sta.water = 0) &
(sta.alarm = 0) & (sta.co2 = 0) & (sta.exfan = 0) & (((sta.user = 0) &
(sta.temp = 0) & (sta.smoke = 1)) | ((sta.user = 0) & (sta.temp = 0) &
(sta.smoke = 0)) | ((sta.user = 1) & (sta.temp = 0) & (sta.smoke = 0))) :
 1; into case expression's first position of variable sta.co2

-- with the following side effects:

-> specification  EG (!sta.co2 | sta.water)  will change from true
to false!

-> specification  EX (sta.water & !sta.light)  will change from true
to false!

-> specification  EF sta.water  will change from true to false!

-> specification  AX (sta.water -> sta.exfan)  will change from
false to true!
-> specification  EX (sta.co2 & !sta.exfan)  will change from false
to true!

-> specification  E [ (!sta.light) U (sta.co2 & !sta.exfan) ]   will
change from false to true!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  EG ((sta.co2 <-> sta.exfan) & sta.water)  is false

-- but this specification can be true if you make the following
changes:

1) Change your initial state to (sta.light = 0) & (sta.water = 1) &
(sta.alarm = 0) & (sta.co2 = 0) & (sta.exfan = 0)

-- with the following side effects:

-> specification  E [ (!sta.water) U sta.alarm ]   will change from
true to false!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  EX (sta.co2 & !sta.exfan)  is false

-- but this specification can be true if you make the following
changes:

1) Remove the transition (sta.smoke & sta.user) : 1; into case
```

```
expression's of variable sta.exfan

-- with the following side effects:

-> specification  E [ (!sta.light) U (sta.co2 & !sta.exfan) ]   will
change from false to true!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  EF ((sta.water & sta.co2) & !sta.alarm)  is false

-- but this specification can be true if you make the following
changes:

1) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 0) & (((sta.user = 1) &
(sta.temp = 1) & (sta.smoke = 0))) : 1; into case expression's first
position of variable sta.water


2) Add the transition (sta.light = 1) & (sta.water = 0) &
(sta.alarm = 1) & (sta.co2 = 1) & (sta.exfan = 0) & (((sta.user = 1) &
(sta.temp = 1) & (sta.smoke = 0))) : 0; into case expression's first
position of variable sta.alarm

-- with the following side effects:

-> there is no side effects!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  E [ (!sta.light) U (sta.co2 & !sta.exfan) ]   is
false

-- but this specification can be true if you make the following
changes:

1) Remove the transition (sta.smoke & sta.user) : 1; into case
expression's of variable sta.light

-- with the following side effects:

-> there is no side effects!
```

O modelo criado para teste sem variáveis de entrada se encontra no arquivo model-noinput.smv localizado na pasta /examples/br/ da implementação e sua descrição pode ser visualizada abaixo. Esse modelo é uma variação do modelo descrito no capítulo 4. A principal diferença é que a variável lux se torna uma variável comum.

```
MODULE main

VAR
unocc : boolean; alarm : boolean; light : {n,d,c};
timeun : 0..2; timeal : 0..3; timer : 0..7;
user : boolean; lux : {1,2}; fire : boolean;

ASSIGN
init(unocc) := 0; init(alarm) := 0; init(light) := n;
init(timeun) := 0; init(timeal) := 0; init(timer) := 0;
init(user) := 0; init(lux) := 1; init(fire) := 0;

next(unocc) := case
  !user & timeun < 2 : 1;
  user | timeun = 2  : 0;
  1 : unocc;
esac;
next(alarm) := case
  fire : 1;
  !fire : 0;
  1 : alarm;
esac;
next(light) := case
  unocc & user : c;
  user : d;
  !user : n;
  lux = 2 : n;
  lux = 1 & (light = c | light = d) : n;
  1 : light;
esac;
next(timeun) := case
  user : 0;
  timeun = 2 : 2;
  !user : timeun + 1;
  1 : timeun;
esac;
next(timeal) := case
  !alarm : 0;
  timeal = 3 : 3;
  alarm : timeal + 1;
  1 : timeal;
esac;
next(timer) := case
  timer = 7 : 0;
  timer < 7 : timer + 1;
  1 : timer;
esac;
```

```
next(user) := case
  timer < 2: 1;
  timer <=7 : 0;
  1 : user;
esac;
next(lux) := case
  timer < 1 : 2;
  timer <=7 : 1;
  1 : lux;
esac;
next(fire) := case
  timer < 3 : 0;
  timer <=7 : 1;
  1 : fire;
esac;

SPEC
AG (!user | !alarm | !fire)
SPEC
AX (light = n & !alarm)
SPEC
AF (timeun = 0 -> light = c)
SPEC
A [!alarm U unocc]
SPEC
AG (light = n & !fire)
SPEC
AX((user) -> (light = d & lux = 1))
SPEC
AF (user & fire)
SPEC
A [!fire  U (timer = 6 & timeun = 1)]
SPEC
EG(user)
SPEC
EX (alarm & timer = 2)
SPEC
EF(user & fire)
SPEC
E [light = n U timeal = 1]
```

Novamente testamos oito fórmulas temporais que são inconsistentes com o modelo e quatro que são consistentes. O algoritmo propôs as seguintes sugestões de mudanças e avisos para cada uma das fórmulas. Modificamos manualmente o arquivo que contém a descrição e observamos mais uma vez que as mudanças sugeridas tornaram as fórmulas verdadeiras e que os avisos sobre os efeitos colaterais estão corretos.

```
-- specification  AG ((!user | !alarm) | !fire)  is true
-- specification  AX (light = n & !alarm)  is true
-- specification AF (timeun = 0 -> light = c)  is true
-- specification  A [ (!alarm) U unocc ]   is true
-- specification AG (light = n & !fire)  is false

-- but this specification can be true if you make the following
changes:

1) Remove the transition (unocc & user) : c; into case expression's
of variable light

2) Remove the transition user : d; into case expression's of
variable light

3) Remove the transition timer <= 7 : 1; into case expression's of
variable fire

-- with the following side effects:

-> there is no side effects!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  AX (user -> (light = d & lux = 1))  is false

-- but this specification can be true if you make the following
changes:

1) Remove the transition timer < 2 : 1; into case expression's of
variable user

-- with the following side effects:

-> specification  E [ light = n U timeal = 1 ]   will change from
false to true!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  AF (user & fire)  is false

-- but this specification can be true if you make the following
changes:

1) Add the transition (unocc = 0) & (alarm = 0) & (light = d) &
(timeun = 0) & (timeal = 3) & (timer = 2) & (user = 1) & (lux = 1) &
(fire = 0) : 1; into case expression's first position of variable
 user
```

```
2) Add the transition (unocc = 0) & (alarm = 0) & (light = d) &
(timeun = 0) & (timeal = 3) & (timer = 2) & (user = 1) & (lux = 1) &
(fire = 0) : 1; into case expression's first position of variable
fire

-- with the following side effects:

-> specification  EF (user & fire)  will change from false to true!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  A [ (!fire) U (timer = 6 & timeun = 1) ]   is
false

-- but this specification can be true if you make the following
changes:

1) Remove the transition timer <= 7 : 1; into case expression's of
variable fire

2) Add the transition (unocc = 0) & (alarm = 0) & (light = d) &
(timeun = 0) & (timeal = 0) & (timer = 2) & (user = 1) & (lux = 1) &
(fire = 0) : 1; into case expression's first position of variable
timeun

3) Add the transition (unocc = 0) & (alarm = 0) & (light = d) &
(timeun = 0) & (timeal = 0) & (timer = 2) & (user = 1) & (lux = 1) &
(fire = 0) : 6; into case expression's first position of variable
timer

-- with the following side effects:

-> there is no side effects!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  EG user  is false

-- but this specification can be true if you make the following
changes:

1) Change your initial state to (unocc = 0) & (alarm = 0) &
(light = n) & (timeun = 0) & (timeal = 0) & (timer = 0) &
(user = 1) & (lux = 1) & (fire = 0)

2) Remove the transition timer <= 7 : 0; into case expression's of
variable user
```

```
-- with the following side effects:

-> specification  AG ((!user | !alarm) | !fire)  will change from
true to false!

-> specification  AX (light = n & !alarm)  will change from true to
false!

-> specification  AF (timeun = 0 -> light = c)  will change from
true to false!

-> specification  A [ (!alarm) U unocc ]   will change from true to
false!

-> specification  AF (user & fire)  will change from false to true!

-> specification  EF (user & fire)  will change from false to true!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  EX (alarm & timer = 2)  is false

-- but this specification can be true if you make the following
changes:

1) Add the transition (unocc = 0) & (alarm = 0) & (light = n) &
(timeun = 0) & (timeal = 0) & (timer = 0) & (user = 0) & (lux = 1) &
(fire = 0) : 1; into case expression's first position of variable
alarm

2) Add the transition (unocc = 0) & (alarm = 0) & (light = n) &
(timeun = 0) & (timeal = 0) & (timer = 0) & (user = 0) & (lux = 1) &
(fire = 0) : 2; into case expression's first position of variable
timer

-- with the following side effects:

-> specification  AX (light = n & !alarm)  will change from true to
false!

-> specification  E [ light = n U timeal = 1 ]   will change from
false to true!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  EF (user & fire)  is false
```

```
-- but this specification can be true if you make the following
changes:

1) Add the transition (unocc = 0) & (alarm = 0) & (light = d) &
(timeun = 0) & (timeal = 3) & (timer = 2) & (user = 1) & (lux = 1) &
(fire = 0) : 1; into case expression's first position of variable
user

2) Add the transition (unocc = 0) & (alarm = 0) & (light = d) &
(timeun = 0) & (timeal = 3) & (timer = 2) & (user = 1) & (lux = 1) &
(fire = 0) : 1; into case expression's first position of variable
fire

-- with the following side effects:

-> there is no side effects!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- specification  E [ light = n U timeal = 1 ]   is false

-- but this specification can be true if you make the following
changes:

1) Remove the transition (unocc & user) : c; into case expression's
of variable light

2) Remove the transition user : d; into case expression's of
variable light

-- with the following side effects:

-> there is no side effects!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Referências Bibliográficas

[1] C.E. Alchourron, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510–530, 1985.

[2] Peter Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. MIT Press, 1988.

[3] Sven-Ove Hansson. *A Textbook of Belief Dynamics*. Kluwer Academic Publishers, 1997.

[4] C.E. Alchourron and D. Makinson. On the logic of theory change: Contraction functions and their associated revision functions. *Theoria*, 48:14–37, 1982.

[5] P. Gärdenfors and D. Makinson. Revisions of knowledge systems using epistemic entrenchment. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge Conference*, pages 83–95. Morgan Kaufmann, 1988.

[6] Hans Rott. Two dogmas of belief revision. *The Journal of Philosophy*, 97(9):503–522, 2000.

[7] C.E. Alchourron and D. Makinson. On the logic of theory change: Safe contraction. *Studia Logica*, 44(4):405–422, 1985.

[8] Adam Grove. Two modellings for theory change. *Journal of Philosophical Logic*, 17(2):157–170, 1988.

[9] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[10] Robert Büessow. *Model Checking Combined Z and Statechart Specifications*. PhD thesis, Technical University of Berlin, Faculty of Computer Science, November 2003. `http://edocs.tu-berlin.de/diss/2003/buessow_robert.pdf`.

[11] Kirsten Winter. Model checking for abstract state machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[13] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[14] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[15] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[16] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1980.

[17] E. M. Clark and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[18] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification CAV*, LNCS 818, pages 415–427. Springer-Verlag, 1994.

[19] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An Opensource Tool for Symbolic Model Checking. In E. Brinksma and K. Larsen, editors, *Proceedings of Computer Aided Verification (CAV 02)*, LNCS 2404, pages 359–364. Springer-Verlag, 2002.

[20] Ken McMillan. *Symbolic Model Checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, School of Computer Science, May 1992. `http://www.kenmcmil.com/pubs/thesis.pdf`.

[21] O. Rodrigues, A. Garcez, and A. Russo. Reasoning about requirements evolution using cluster belief revision. In A. Bazzan and S. Labidi, editors, *Proceedings of 17th Brazilian Symposium on Artificial Intelligence (SBIA 2004)*, LNCS 3171, pages 41–51. Spring-Verlag, 2004.

[22] Odinaldo Rodrigues. *A Methodology for Iterated Information Change*. PhD thesis, Imperial College, Department of Computing, December 1997. `http://www.dcs.kcl.ac.uk/staff/odinaldo/pdf/phd_thesis.pdf`.

[23] P. Liberatore and M. Schaerf. The complexity of model checking for belief revision and update. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 556–561. AAAI Press/The MIT Press, 1996.

[24] D. Zowghi, A. Ghose, and P. Peppas. A Framework for Reasoning about Requirement Evolution. In N. Foo and R. Goebel, editors, *Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence*, LNCS 1114, pages 157–168. Springer-Verlag, 1996.

[25] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering and Methodology*, 14(3):277–330, 2005.

[26] C.K. MacNish and M-A. Williams. From belief revision to design revision: Applying theory change to changing requirements. In G. Antoniou and M. Truszczynski, editors, *Learning and Reasoning with Complex Representations*, LNCS 1359, pages 207–222. Springer-Verlag, 1998.

[27] Nikos Gorogiannis. *Computing Minimal Changes of Models of Systems*. PhD thesis, University of Birmingham, School of Computer Science, June 2003. `http://www.cems.uwe.ac.uk/~ngkorogi/pubs/thesis.ps.gz`.

[28] Y. Ding and Y. Zhang. CTL Model Update: Semantics, Computations and Implementation. In *Proceedings of the 17th Europen Conference on Artificial Intelligence (ECAI 2006)*, pages 362–366. IOS Press, 2006.

[29] H. Harris and M. D. Ryan. Theoretical Foundations of Updating Systems. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 291–294. IEEE Computer Society Press, 2003.

[30] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge database and revising it. In P. Gärdenfors, editor, *Belief Revision*, Cambridge Tracts in Theoretical Computer Science, pages 183–203. Cambridge University Press, 1992.