Universidade Federal de Minas Gerais
Instituto de Ciência Exatas - ICEX
Departamento de Ciência da Computação

Um Verificador de Modelos Explícito-Simbólico

Umberto Souza da Costa

Tese apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Belo Horizonte, 18 de março de 2005

Agradecimentos

Deus, obrigado pelas oportunidades concedidas, pela força e perseverança nos momentos difíceis, que não foram poucos, e pelo êxito no cumprimento de meus objetivos. Meus pais e meus irmãos, vocês dividiram cada momento de minha luta. Sempre estivemos unidos para a superação dos obstáculos. Vocês foram um importante referencial e meu grande apoio nesses anos do doutorado. Meus amigos, vocês foram minha família durante este período e sempre estiveram prontos para dar o apoio necessário e para dizer que tudo daria certo. Teria sido muito difícil, senão impossível, superar todas as dificuldades sem vocês. Graças a vocês, esse período foi muito mais que uma etapa de minha vida profissional, foi um período de enriquecimento no campo das amizades, das emoções e da criação de novas perspectivas. Nos corredores do departamento, encontrei pessoas especiais como Kíssia, Silvana, Olga, Karla, Joyce, Luiz, Judson, Wesley e Rainer, amigos com os quais dividi algumas preocupações mas, principalmente, muitas alegrias. Encontrei meu grande amigo Délio, bem como Shayra, Gustavo, Paula, Josi Anne, Fafá, Larissa, Letícia, Leandro, Raphael, Gustavo, Sérgio Neves, Djalma, Cláudio, Íria, Adriana, Marcos, Danilo, Ronaldo, Sérgio Matos, Getúlio e muitas outras pessoas que fazem parte de minha vida e de minha alma. Agradeço a meu orientador, Sérgio Campos, e a meu co-orientador, David Déharbe, pelo direcionamento na travessia do labirinto de informações que é o trabalho de pesquisa de um doutorado. Obrigado, Túlia e Sheila. A gentileza, a simpatia e a presteza que vocês têm não passaram desapercebidas. Vocês estão entre os amigos dos quais sentirei saudades.

Resumo

Neste trabalho, propomos uma modelagem que combina representações explícitas e simbólicas em um modelo de verificação formal explícito-simbólico. Os modelos explícitos e simbólicos têm sido usados com sucesso na verificação de sistemas concorrentes de estados finitos, como circuitos sequenciais complexos e protocolos de comunicação. A modelagem proposta tem como objetivo combinar as técnicas explícitas e simbólicas para verificação de um mesmo modelo e permitir o emprego da técnica mais eficiente à verificação de cada aspecto do modelo. Inicialmente, apresentamos uma visão geral do processo de verificação do modelo de um sistema e discutimos como propriedades temporais podem ser especificadas para este modelo. A discussão é focalizada em lógica temporal CTL, mas também consideramos a lógica temporal LTL. Em seguida, introduzimos os modelos explícitos e seus algoritmos básicos. As principais técnicas de refinamento dos verificadores de modelos explícitos são apresentados, tais como redução de ordem parcial, *bit-state hashing*, automatos minimizados, compressão de vetores de estados e análise estática. Discutimos, também, a verificação de modelos simbólicos e os diagramas de decisão binária. Caracterizações em ponto fixo são dadas para as versões simbólicas dos algoritmos de verificação de modelos. Apresentamos a linguagem de descrição do verificador de modelos explícito-simbólico, o *Interchange Format* [Bozga et al., 1999] desenvolvido nos laboratórios Verimag. Com isso, concluímos a apresentação dos conceitos básicos necessários à introdução da modelagem explícito-simbólica. A concepção do modelo combinado explícito-simbólico e algoritmos relacionados é a base teórica de nosso trabalho. Adaptamos descrições em *Interchange Format* à modelagem teórica do modelo combinado. Após isso, discutimos a implementação do verificador de modelos explícito-simbólico, trazendo informações sobre suas estruturas de dados mais importantes e o processo geral de verificação, desde a coleta de símbolos até a construção do modelo combinado. Finalmente, mostramos resultados experimentais e sugerimos trabalhos futuros.

Resumo Estendido

Capítulo 1: Visão Geral

Neste capítulo, apresentamos a visão geral de nosso trabalho. Inicialmente, discutimos a necessidade de usarmos os métodos formais e apresentamos algumas de suas principais abordagens. O uso de métodos formais é justificado pela crescente complexidade das aplicações computacionais, que tem feito com que ferramentas de auxílio à detecção de erros sejam indispensáveis a sistemas de hardware e software. Os métodos formais incluem linguagens matemáticas, técnicas e ferramentas para especificar e verificar sistemas em hardware e software. Diferente da simulação, que examina os resultados da execução de testes sobre algumas entradas, os métodos formais executam buscas exaustivas sobre o espaço de busca do problema. Assim, os métodos formais são capazes de provar que um dado sistema atende a suas especificações. A prova de teoremas e a verificação de modelos são duas importantes abordagens da verificação formal. A prova de teoremas usa lógica matemática pra expressar um dado sistema e suas propriedades. Um provador de teoremas forma iterativamente uma definição matemática do sistema a ser verificado e, então, verifica seus invariantes sobre esta definição. Os invariantes descrevem propriedades a partir da criação e da prova de teoremas a partir dos axiomas do sistema.

A verificação de modelos, abordagem central de nossa pesquisa, é introduzida junto a seus pontos fortes e suas aplicações. A verificação de modelos consiste na representação de um dado sistema por meio de um modelo finito a ser exaustivamente analisado para a determinação de sua conformidade a certas propriedades. O modelo pode ser representado como um grafo onde cada vértice corresponde a um estado do sistema e as arestas correspondem a transições entre os estados. Temos a garantia de que o processo de verificação de um modelo sempre termina, pois seus algoritmos são baseados em transformadores de predicados monotônicos que operam sobre conjuntos de estados finitos. Outro benefício da verificação de modelos é a geração de contra-exemplos nos casos em que o modelo não atende a suas especificações. Os estados de um modelo podem ser representados explicitamente ou por meio de alguma representação simbólica, o que origina duas categorias de verificadores de modelos, os explícitos e os simbólicos.

Em seguida, apresentamos o principal objetivo de nossa pesquisa, o desenvolvimento de um modelo que combina representações explícitas e simbólicas e seus algoritmos de verifi-

cação em uma abordagem explícito-simbólica. Com este modelo combinado, pretendemos usar as técnicas explícitas e simbólicas concomitantemente para verificar um mesmo modelo e aplicar a técnica mais eficiente a cada aspecto modelado. Inicialmente, as propriedades de um sistema devem ser particionadas em subconjuntos explícito e simbólico, induzindo a geração de modelos explícito e simbólico subjacentes. O particionamento de propriedades pode ser feito de acordo com qualquer política, seguindo algum critério de eficiência. Após isso, o modelo explícito-simbólico combina e integra estes modelos subjacentes e coordena seus algoritmos de verificação. A integração baseia-se na associação entre transições explícitas e simbólicas.

Finalmente, relacionamos nosso modelo combinado explícito-simbólico a outros trabalhos, sejam aqueles relacionados à integração de outras abordagens de verificação formal, ou trabalhos voltados à verificação de modelos exclusivamente explícitos ou simbólicos. Comparamos, também, o modelo combinado ao SLAM [Ball and Rajamani, 2002], a abordagem que mais se assemelha a nossa. Apresentamos, também, nossas principais contribuições.

Capítulo 2: Verificação Lógico-Temporal de Modelos

Este capítulo apresenta a verificação de modelos como uma abordagem voltada à verificação formal de sistemas concorrentes. Inicialmente, apresentamos uma visão geral do processo de verificar um modelo de um sistema computacional. A verificação de modelos é uma abordagem segundo a qual o sistema de interesse é representado como uma máquina de estados finitos a ser verificada para a determinação de sua adequação a suas especificações. A máquina de estados finitos de um sistema concorrente pode ser vista como uma estrutura de Kripke, um grafo de transição de estados onde cada estado é definido por um conjunto de proposições. O procedimento de verificação de modelos determina se esta estrutura atende às especificações de uma fórmula particular. A especificação é atendida em um modelo se existir um caminho de execução que parte de um estado inicial onde a especificação é atendida. As especificações são dadas por alguma lógica temporal.

As lógicas temporais são usadas para descrever a ordenação temporal de eventos de um sistema, indicando como os sistemas se comportam e quais configurações eles podem assumir ao longo do tempo de execução. Uma configuração de sistema corresponde a um estado do sistema, definido pelo conjunto de valores assumidos pelas variáveis do sistema naquele momento. Em vez de especificar propriedades sobre estados individuais, as lógicas temporais especificam propriedades sobre conjuntos de estados e como o sistema passa de um estado para outro. Assim, a avaliação de uma fórmula da lógica temporal requer que todos os estados de uma seqüência de execução sejam levados em consideração. As lógicas temporais oferecem diferentes pontos de vista em relação à estrutura do tempo. As lógicas temporais de tempo ramificado, CTL, e de tempo linear, LTL, são comumente usadas na verificação de modelos.

A lógica CTL considera vários caminhos de computação alternativos em um dado momento e todos os caminhos de computação são considerados simultaneamente. Nossa discussão se concentra na lógica CTL, pois este é o formalismo usado em nossas especificações.

A lógica CTL é suficientemente expressiva para formular um importante conjunto de propriedades temporais, como *safety*, *liveness*, ausência de *deadlocks* e *fairness*. Apresentamos tanto os aspectos sintáticos como os semânticos da lógica CTL.

Adicionalmente, fornecemos uma visão geral da lógica LTL. A lógica LTL assume o tempo como sendo dado por uma seqüência de execução onde cada caminho de computação é considerado separadamente, inferindo sobre uma seqüência de execução apenas. Os poderes de expressão das lógicas CTL e LTL são diferentes. Há asserções que podem ser expressas em CTL, mas não em LTL, e vice-versa.

Capítulo 3: Verificação de Modelos Explícitos

A verificação de modelos de estados explícitos explora os estados sistematicamente, enumerando os estados possíveis de um sistema e executando verificações sobre estados individuais, um estado a cada momento. Estes verificadores têm sofrido do problema da explosão de estados, inerente à representação utilizada. O problema da explosão de estados acontece quando o tamanho da representação impede que o modelo seja verificado. Ao longo dos anos, vários aperfeiçoamentos foram desenvolvidos para minimizar este problema e estender o poder dos verificadores de modelos explícitos. Tais aperfeiçoamentos refinam a representação e os algoritmos relacionados. Atualmente, os verificadores de modelos explícitos são bastante competitivos e parecem ser a melhor escolha à verificação de alguns sistemas, como aqueles com muitas transformações de dados ou sistemas de software.

Os verificadores de estados explícitos usam um procedimento de busca em profundidade ou largura para executar buscas exaustivas no espaço de estados. Os métodos explícitos tendem a apresentar comportamentos relativamente mais previsíveis, em termos de eficiência e memória. As representações explícitas têm sido bem sucedidas no tratamento de sistemas concorrentes com poucos processos, onde o número de estados é geralmente pequeno. Os verificadores de modelos explícitos precisam de substancial abstração e redução de estados para minimizar o problema da explosão de estados e serem práticos para sistemas de tamanho industrial. Outros refinamentos são a redução de ordem parcial, *bit-state hashing* e a construção *on-the-fly* das representações.

Os modelos explícitos podem ser baseados em estruturas de Kripke ou em autômatos. Na primeira abordagem, os algoritmos rotulam recursivamente os estados do modelo com as propriedades válidas em cada um deles. Cada estado possui um conjunto inicial de rótulos que vai sendo atualizado à medida que as sub-fórmulas da especificação são avaliadas. O capítulo apresenta um conjunto básico de algoritmos de verificação desta abordagem, considerando especificações dadas em lógica CTL. Na abordagem baseada em autômatos, tanto o sistema como a especificação são descritos por autômatos. As propriedades são verificadas determinando-se a existência de ciclos de aceitação no autômato resultante do produto entre o autômato do sistema e o autômato para o complemento da especificação. Discutimos, também, o funcionamento do verificador SPIN [Holzmann, 1997b], um dos mais importantes representantes dessa categoria de verificadores. Esta discussão baseia-se na lógica LTL.

Finalmente, apresentamos algumas técnicas de otimização explícita, como a redução de ordem parcial, *bit-state hashing*, autômatos minimizados, compressão de vetores de estado e análise estática. A redução de ordem parcial é usada para reduzir o número de estados alcançáveis que são explorados durante a verificação, minimizando o problema da explosão de estados. Os métodos de redução de ordem parcial são baseados em técnicas de redução estática. *Bit-state hashing* é uma técnica comprometida com a redução do espaço voltado ao armazenamento dos estados alcançáveis, sendo uma alternativa ao uso de tabelas *hash*. Em vez de armazenar um conjunto de descritores de estados, a técnica de *bit-state hashing* representa cada estado como um endereço *hash*, via uma tabela de bits. Os autômatos minimizados, outra alternativa ao armazenamento de estados, fornecem uma estrutura dinâmica para armazenar estados por meio de um autômato finito determinístico minimizado. Técnicas de compressão de vetores de estado são comprometidas com o uso eficiente do espaço de memória e atuam reduzindo o armazenamento necessário aos vetores de estados, sem perda de informação. A análise estática é usada para aumentar a eficiência da verificação com base na remoção das partes de um sistema que são irrelevantes à verificação de uma determinada propriedade.

Capítulo 4: Verificação de Modelos Simbólicos

Os modelos simbólicos exploram regularidades no espaço de estados para produzir representações mais compactas. Em vez de explorar os estados individualmente como feito pela verificação de modelos explícitos, a verificação de modelos simbólicos usa codificações eficientes de fórmulas lógicas Booleanas para representar e explorar conjuntos de estados atomicamente. Assim, a verificação de modelos simbólicos normalmente nos permite verificar sistemas com um número de estados muito maior que o permitido pela verificação de modelos explícitos. Na verificação de modelos simbólicos, conjuntos de estados e transições são representados por suas funções características. Os estados de um modelo simbólico são descritos pelo conjunto de variáveis do sistema que são verdadeiras em cada um deles. Por outro lado, as transições podem ser definidas em termos das variáveis de estado atual e de estado futuro. As representações computacionais das funções características precisam ser eficientes e devem fornecer operações essenciais, como conjunção, disjunção, testes de igualdade, quantificação existencial e substituição. Os diagramas de decisão binária, conhecidos como BDDs [Bryant, 1986], oferecem uma representação simbólica de funções da lógica proposicional muito eficiente.

Os BDDs são uma estrutura de dados com um conjunto de algoritmos voltada à representação de funções Booleanas. As funções Booleanas são representadas por meio de grafos acíclicos dirigidos com restrições na ordenação dos argumentos das funções. Cada argumento de uma função corresponde a uma variável BDD, que pode ter um ou mais nós associados no grafo. Com exceção dos nós terminais, que são rotulados por zero ou um, cada nó no grafo têm duas arestas de saída, uma representando o caso onde o argumento correspondente é falso e o outro representando o caso onde o argumento correspondente é verdadeiro. Os BDDs têm sido usados para representar sistemas com até 10^{20} estados.

Contudo, o tamanho do grafo que representa uma função Booleana é altamente sensível à ordenação das variáveis correspondentes a seus argumentos.

A representação utilizada pelos BDDs é baseada na expansão de Shannon [Shannon, 1938], responsável por decompor uma função em outras mais simples. Dada uma função, as instâncias de seus argumentos descrevem caminhos do digrama de decisão, começando no nó raiz e tomando uma das duas arestas de cada nó, de acordo com a avaliação do argumento correspondente. Os BDDs permitem a execução de um rico conjunto de operações nos grafos que representam uma função Booleana, baseados em algoritmos básicos de manipulação e na combinação entre eles. Entre outros algoritmos, os procedimentos `apply` e `restrict` são os mais importantes para a verificação de modelos. O procedimento `apply` cria a representação de uma função de acordo com uma expressão Booleana com dois operandos e um operador genérico. Dados grafos representando os operandos e um operador genérico, o procedimento `apply` produz o grafo resultante da aplicação deste operador genérico sobre os grafos de entrada. Este procedimento atua recursivamente das raízes dos grafos representando os operandos até suas folhas, construindo o grafo de saída com base em uma derivação da expansão de Shannon. O procedimento `restrict` recebe como entrada o grafo de uma função, um índice correspondente a uma variável neste grafo e um valor específico. O procedimento produz um grafo correspondente à função de entrada após trocar cada vértice correspondente à variável selecionada pelo valor dado na entrada.

Geralmente, funções Booleanas são utilizadas para representar tanto o grafo de transição de estados como a especificação a ser verificada. Para operar sobre funções Booleanas, os algoritmos simbólicos baseiam-se em caracterizações dos operadores da lógica temporal em ponto fixo, de forma que cada operador corresponde a uma função e o conjunto de estados resultante da aplicação do operador corresponde ao ponto fixo da função. São mostrados os algoritmos para a computação de pontos fixos e as caracterizações dos operadores CTL em pontos fixos.

Capítulo 5: *Verimag Interchange Format*

O *Verimag Interchange Format* [Bozga et al., 1999], ou formato IF, é uma linguagem baseada em uma versão dinâmica de autômatos estendidos. A linguagem foi originalmente proposta para a modelagem de sistemas de tempo real com comunicação assíncrona, e tem passado por várias extensões, incluindo a habilidade para modelar e analisar sistemas dotados com a criação dinâmica de processos e tipos de dados dinâmicos. Atualmente, várias ferramentas de validação voltadas a sistemas de tempo real usam o formato IF. O conjunto de tais ferramentas é conhecido como o ambiente de validação IF.

No formato IF, os sistemas são modelados como vários autômatos estendidos que executam em paralelo e que interagem por meio da passagem de mensagens ponto-a-ponto, via *buffers* de comunicação e variáveis compartilhadas. Cada autômato estendido é descrito por meio de um processo. Como os processos podem ser dinamicamente criados ou destruídos, o número de autômatos ativos pode mudar durante a execução de um sistema. Os processos podem ter um conjunto de variáveis locais. Há um rico conjunto de variáveis,

desde aquelas com tipos predefinidos àquelas com tipos definidos pelo usuário. Os tipos predefinidos incluem o Booleano, o inteiro, o flutuante, um tipo usado pelos identificadores de processos e aquele usado para medir o tempo de execução dos processos. O usuário pode definir tipos usando mecanismos como enumerações, subfaixas e matrizes.

Descrições IF são compostas de instâncias de processos que são executadas em paralelo e interagem assíncronamente por meio de variáveis compartilhadas e passagem de mensagens, entre si ou com o ambiente externo. A execução assíncrona significa que um processo é executado enquanto os outros permanecem em estado de espera. A execução de processos é dada pelo grafo de suas execuções intercaladas. Cada processo é inequivocamente identificado por meio de um valor e possui uma fila de mensagens pendentes, isto é, mensagens recebidas e ainda não consumidas. A passagem de mensagens é realizada através de sinais, rotas de sinais e buffers de comunicação FIFO.

Os estados de controle são o principal componente de estruturação dos processos. Os estados de controle definem o comportamento de um processo por meio de ações, transições para outros estados e, possivelmente, sub-estados. Não há limite teórico para a profundidade do aninhamento de sub-estados. Uma especificação de estado pode incluir restrições temporais e um conjunto de sinais deferidos. Outros componentes da linguagem são também mostrados. Para cada componente, apresentamos sua descrição sintática e discutimos os aspectos semânticos mais importantes. A discussão é ilustrada com uma descrição IF para o protocolo de bit alternado.

Capítulo 6: A Modelagem Explícito-Simbólica

Este capítulo tem dois objetivos principais. O primeiro deles é mostrar os aspectos teóricos envolvidos com a criação do modelo combinado explícito-simbólico. Esta modelagem é totalmente independente de uma linguagem de descrição. O segundo objetivo é aplicar esta modelagem teórica e genérica a descrições no formato IF.

A modelagem teórica é dada em dois passos distintos. Inicialmente, os modelos explícitos e simbólicos são gerados como projeções induzidas pelo particionamento das variáveis de um modelo original. Cada modelo gerado cobre apenas um subconjunto das variáveis e investiga apenas parte do espaço de busca. No segundo passo, mostramos como estes modelos devem cooperar entre si. Os modelos têm que ser executados de forma combinada para viabilizar a exploração de todo o espaço de busca original e emular o comportamento do sistema original. Na modelagem teórica, a combinação dos modelos é realizada com base na rotulação das transições dos modelos gerados, para permitir o entrelaçamento dos mesmos. Transições que têm o mesmo rótulo nos modelos explícito e simbólico correspondem a transições geradas a partir da mesma transição original, e, portanto, devem acontecer simultaneamente. Esta formalização é o núcleo de nosso trabalho, pois define o comportamento do modelo combinado a partir da cooperação e sincronização dos modelos explícito e simbólico subjacentes.

A aplicação da formalização ao formato IF também é descrita em dois passos, semelhantes aos da modelagem teórica. Primeiro, nós mostramos como o formato IF é utilizado

para a geração de modelos explícito e simbólico, de acordo com o particionamento das variáveis do sistema em um subconjunto das variáveis explícitas e um subconjunto das variáveis simbólicas. A seguir, combinamos estes modelos para emular o comportamento do modelo original por meio da abordagem explícito-simbólica. No modelo combinado, a relação de transição explícita é representada como um vetor de listas ligadas, onde cada entrada do vetor corresponde a um estado e sua lista ligada define para onde o estado considerado pode realizar uma transição. Por outro lado, a relação de transição simbólica deve ser representada como uma fórmula entre proposições de estado atual e proposições de estado futuro, usando BDDs. Cada transição simbólica é definida por uma conjunção de duas fórmulas, uma definida sobre as variáveis de estado atual e a outra definida sobre as variáveis de estado futuro, cada fórmula caracterizando um estado. A relação de transição é definida pela disjunção de transições simbólicas individuais. Por uma questão de eficiência, propomos uma nova forma de associação entre as transições na modelagem de descrições IF. Em vez de usarmos as transições rotuladas do modelo teórico, os modelos explícito e simbólico são entrelaçados com base na associação de transições simbólicas a transições explícitas. Cada transição simbólica é associada à transição explícita que deve ocorrer simultaneamente a ela.

Após isso, apresentamos os algoritmos usados para a verificação do modelo combinado explícito-simbólico. Os algoritmos consideram que as especificações são dadas em CTL. Descrevemos os algoritmos em duas etapas, primeiro considerando a verificação de proposições atômicas, ou explícitas ou simbólicas, depois considerando a verificação de especificações envolvendo variáveis explícitas e simbólicas. Para cada algoritmo, descrevemos a complexidade para o pior caso. Um exemplo de verificação é mostrado para o modelo de um forno microondas.

Capítulo 7: A Implementação do Verificador

Neste capítulo, revelamos os detalhes de implementação do modelo explícito-simbólico. Apresentamos as idéias principais e as decisões tomadas durante a concepção dos modelos subjacentes e a implementação do modelo combinado. A discussão concentra-se no processo de construção dos modelos e em estruturas de dados básicas.

A implementação do verificador explícito-simbólico partiu do zero. Apesar de termos utilizado verificadores de modelos bem estabelecidos como inspiração, o uso direto deste verificadores como blocos de construção não foi adotado, pois isso provavelmente acarretaria problemas de compatibilidade, difíceis ou impossíveis de serem resolvidos. Estes problemas decorrem do uso de diferentes lógicas temporais e pela adoção de diferentes linguagens de descrição pelos verificadores, por exemplo. Além disso, a integração de verificadores de terceiros exigiria o alto custo envolvido com a compreensão do funcionamento e das técnicas de refinamento utilizadas pelos componentes integrados. Portanto, implementamos os verificadores explícito e simbólico e os integramos.

O verificador explícito foi implementado de acordo com algumas das técnicas explícitas tradicionais, mas não seguimos rigorosamente nenhum verificador. O projeto atual é in-

gênuo no sentido em que necessita de técnicas de análise estática mais robustas e redução de ordem parcial para aperfeiçoar a exploração do espaço de busca e minimizar o problema da explosão de estados. Atualmente, estados explícitos não-alcançáveis podem ser gerados durante a construção do modelo explícito. Por outro lado, nós implementamos técnicas de compressão dos vetores de estado e reduzimos o tempo gasto na identificação dos estados já visitados, através do uso de uma cache de estados explícitos.

Em grande parte, nosso verificador simbólico foi implementado de acordo com o verificador Verus [Campos, 1996]. Os algoritmos de construção e verificação foram adaptados a partir de Verus, removendo sua informação temporal quantitativa, mas mantendo os algoritmos de ajuste fino compatíveis com nossa abordagem.

Os algoritmos explícito-simbólicos assumem que as especificações são dadas em CTL. As especificações são dadas em uma representação estrutural por meio de uma árvore de análise sintática. Nesta árvore sintática, as folhas representam proposições atômicas enquanto os nós internos representam expressões explícito-simbólicas. Apesar de os algoritmos de verificação inicialmente propostos converterem estados explícitos e simbólicos em estados explícito-simbólicos antes da verificação da especificação, nossa implementação explora cada modelo subjacente ao máximo antes de integrar os resultados da verificação sobre estes modelos. Esta decisão reduz o custo de sincronização e torna a verificação destes modelos mais independente.

A estrutura de dados mais elementar do verificador combinado é utilizada para representar os símbolos das descrições. Há um símbolo para cada variável no sistema, incluindo as variáveis definidas pelo usuário e aquelas criadas para controlar a verificação dos modelos, como as variáveis usadas para controlar as filas de mensagens dos processos. Criamos seis categorias de símbolos, distinguidas por aplicação: símbolos para definir tipos, sejam tipos básicos ou definidos pelo usuário, símbolos para definir sinais, símbolos para definir estados, símbolos para definir variáveis locais, símbolos para definir variáveis globais e símbolos usados para definir a ativação de processos. Os símbolos usados por variáveis explícitas e simbólicas são baseados na mesma estrutura de dados.

Os vetores de estado são uma das principais estruturas de dados explícitas. Um vetor de estado representa um estado explícito e consiste em um vetor de valores inteiros, um valor para cada símbolo explícito. O número de bytes exigido por um vetor de estados é o mesmo para todos os processos de um dado sistema. Armazenamos versões comprimidas dos vetores de estado para reduzir os requisitos de memória. Em vez de armazenar o valor decimal de cada símbolo explícito em um byte, o algoritmo de compressão exige, para cada símbolo explícito, apenas o número de bits necessários para armazenar sua faixa de valores permitidos. Assim, um mesmo byte pode ser compartilhado por diversos símbolos explícitos. Esta técnica de compressão pode produzir uma economia de memória considerável, especialmente para sistemas cujas variáveis apresentam faixas de valores pequenas. Naturalmente, o algoritmo de compressão aumenta o esforço computacional, mas a análise de custo e benefício é positiva. Em termos de estruturas de dados simbólicas, os BDDs são a estrutura de dados básica. Eles são utilizados para representar tanto os estados como a relação de transição. Os modelos são integrados associando-se transições simbólicas a explícitas.

Apresentamos os processos de construção dos modelos explícito e simbólico. Transições explícitas são definidas por dois vetores de estado, um para o estado atual e outro para o estado futuro. Cada vetor de estado funciona como descritor de um estado explícito. Embora muitos estados e transições explícitos sejam necessários para modelar as diversas configurações assumidas por cada estado de controle IF, a discussão concentra-se na construção de uma transição individual. Quanto ao modelo simbólico, várias transições podem ser representadas por uma mesma fórmula Booleana. Mostramos como modelar explicita e simbolicamente os principais componentes encontrados em uma descrição IF.

A construção do modelo explícito-simbólico corresponde ao processo de criação e coordenação de representações explícitas e simbólicas a partir de descrições IF. Na primeira etapa deste processo, coletamos todos os símbolos do sistema e os armazenamos em uma tabela de símbolos com acesso *hash*. Os símbolos que definem tipos e aqueles que controlam o processo de verificação, como os símbolos que controlam a fila de mensagens dos processos, são também armazenados na tabela de símbolos. Na segunda etapa, atribuímos a representação de cada símbolo armazenado anteriormente, de acordo com um arquivo de configuração fornecido pelo usuário. Ao final destas etapas, coletamos diversas estatísticas sobre o sistema, como o número de símbolos em cada modelo subjacente. Em seguida, passamos à criação propriamente dita do modelo explícito-simbólico. Para cada estado de controle IF, diversas transições explícitas são geradas, cada uma delas com uma transição simbólica associada. A varredura dos estados de controle IF começa pela criação do conjunto dos estados explícitos atuais e pelo conjunto de estados simbólicos atuais. Estes conjuntos definem a configuração inicial, antes da execução das declarações do estado de controle. A representação simbólica permite que vários estados do modelo sejam representados por meio de uma única fórmula Booleana, mas os estados explícitos devem ser enumerados. Cada estado explícito demanda a criação de uma transição explícita particular, definida por um vetor de estado atual e outro futuro. Inicialmente, os vetores de estado atual e futuro são o mesmo. Durante a varredura das instruções do estado de controle, o vetor de estado futuro vai sendo atualizado para refletir a modelagem das instruções. Ao final da varredura de cada estado de controle, diversas transições explícitas foram criadas, cada qual com sua transição simbólica associada. Note que o algoritmo requer repetidas computações da mesma transição simbólica, uma vez para cada transição explícita computada.

Finalmente, mostramos a modelagem das filas de mensagens de cada processo e a modelagem da cache de estados explícitos.

Capítulo 8: Experimentos e Considerações Finais

Neste capítulo, mostramos os resultados experimentais do verificador explícito-simbólico e discutimos os pontos fortes e fracos de nosso trabalho. Sugerimos futuras melhorias para o trabalho atual e apresentamos nossas considerações finais.

Três sistemas foram utilizados nos experimentos, cada qual com diferentes particionamentos de variáveis. O primeiro sistema experimentado foi o sistema do forno microondas.

O segundo sistema foi um conversor de números binários para números decimais, enquanto o terceiro sistema representa uma calculadora. Os experimentos mostram o tempo total envolvido na verificação dos modelos, o tempo gasto apenas durante a verificação dos modelos e o tempo envolvido com compressões e descompressões dos vetores de estado. Nestes experimentos, os projetos foram verificados utilizando números inteiros codificados em oito bits.

A análise dos experimentos revela que a escolha das representações tem grande impacto na eficiência do verificador. Alem disso, os resultados reforçam a necessidade de aprimoramento do componente explícito do verificador combinado, especialmente no que diz respeito a técnicas de análise estática que aproximem o conjunto de estados explícitos manipulados ao conjunto dos estados alcançáveis, entre outras melhorias sugeridas. Por outro lado, quando consideramos nossa implementação como um protótipo, nosso objetivo foi alcançado ao provarmos a viabilidade da combinação explícito-simbólica. Em especial, os experimentos mostram que, para alguns problemas, a verificação de modelos explicito-simbólicos é mais eficiente do que as abordagens puramente explícita e puramente simbólica. Outros benefícios obtidos com o projeto são discutidos, como a proposta de armazenamento dos estados explícitos por meio de uma cache.

Nossas principais contribuições são a proposta de um ambiente flexível para a verificação de modelos, combinando representações explícitas e simbólicas, e a implementação computacional do modelo explícito-simbólico e seus algoritmos, considerando sistemas descritos pelo formato IF.

Abstract

In this work we propose a modeling that combines explicit and symbolic representations in an explicit-symbolic formal verification model. Both explicit and symbolic models have been successfully used in the verification of finite state concurrent systems, such as complex sequential circuits and communication protocols. The proposed model aims to use explicit and symbolic techniques together to verify the same model and to make it possible to employ the most efficient technique to handle each aspect of the model. First, we give an overview of the process of model checking a system and discuss how temporal properties can be specified for a given model. The discussion is focused on computational tree logic, but we also consider linear temporal logic. Next, we present the explicit representation and its basic algorithms. The main techniques for improving explicit model checkers are presented, such as partial order reduction, bit-state hashing, minimized automata, compression of state vectors and static analysis. Also, we discuss the symbolic model checking and the binary decision diagrams. Fixpoint characterizations are given for the symbolic versions of the model checking algorithms. Then, we present the Interchange Format [Bozga et al., 1999] developed at Verimag labs, the description language of the combined explicit-symbolic model checker. At this point, we have presented the background for the introduction of the explicit-symbolic modeling. The conception of the combined explicit-symbolic model and related algorithms is the theoretical basis of our work. Descriptions in the Interchange Format are adapted to the theoretical modeling for the combined model. After that, we discuss the implementation of the explicit-symbolic model checker, discussing important data structures and the overall process, from the collection of symbols to the construction of the combined model. Finally, we show experimental results and suggest future work.

List of Tables

List of Figures

Contents

Chapter 1

Overview

This chapter gives an overview of our work. Initially, we discuss the need for formal methods and present some of their main approaches. Model checking, the approach of interest to our research, is introduced together with its strengths and applications. Such discussion paves the way for presenting the main goal of our research, the development of a framework to combine explicit and symbolic models and its verification algorithms. After that, we compare our framework to related works and present our main contributions.

1.1 Formal Methods

Due to the ever growing complexity of computing applications, automatic tools have been used to help developers find bugs in hardware and software systems. Two approaches that have been applied to achieve this goal are simulation and formal verification. The simulation approach consists of executing tests over given inputs and examining the results. This technique can easily evaluate both control and data aspects, but generally only a subset of states is examined. In other words, simulation can state that specifications hold for some inputs, but it does not prove correctness over the whole set of states.

Formal Methods include mathematical-based languages, techniques and tools for specifying and verifying hardware and software systems [Clarke et al., 1996]. Specification consists of the description of a system and its properties. Such properties range from functional or timing behavior to performance and structural aspects. Temporal logic is one of the tools used for writing formal specifications, as are the automaton approach [Büchi, 1960] and the Z notation [Spivey, 1992]. Temporal logic deals with the behavior specification of concurrent systems. Complex systems and critical applications benefit from formal methods because they make systems more reliable. Different from simulations, formal verification techniques perform exhaustive searches over problem domains, proving conformity to specifications instead of only pointing errors for some input data. Two of the most important formal verification techniques are theorem proving and model checking.

Theorem proving uses mathematical logic to express both the system and its properties. A theorem prover iteratively forms a mathematical definition of the system to be verified and then checks the invariants against this definition. Invariants describe properties of the system, being established by phrasing and proving theorems from the axioms of the system [Clarke et al., 1996]. Theorem provers are able to handle much larger designs than model checkers can, and they provide a much wider range of reasoning techniques, as structural induction. Differently from the model checking approach, theorem proving is able to generate proofs on infinite state space. Many theorem provers use the classical higher-order logic, also called HOL. Roughly speaking, a higher-order logic is characterized for allowing sets to be quantified and for allowing sets to be elements of other sets. For more details on higher-order logic, please refer to [Church, 1940] and [Andrews, 1986]. HOL-based theorem provers include the *HOL System* [Gordon and Melham, 1993] and the *Prototype Verification System* [Owre et al., 1992] [Owre et al., 1998], also refered to as PVS. The HOL System is an interactive theorem-proving environment based on the classical higher-order logic and on a proof management meta-language, ML [Milner et al., 1997]. The meta-language is used for denoting terms and theorems of the logic, expressing and applying proof strategies and developing logical theories. Although HOL has been initially designed for the specification and verification of hardware systems, it has been applied to many other applications, as communication protocols. On the other hand, PVS has an own specification language, based on a typed higher-order logic. PVS supports a mechanized verification that exploits both theorem proving and model checking. PVS can achieve a high level of automation by using decision procedures for various theories. Unfortunately, theorem provers are more complex and difficult to use than model checkers because they typically require mathematical induction and interaction from a highly skilled user.

Model checking is one well-known and successful formal technique for verifying finite state concurrent systems. Model checking consists of representing a given system by means of a finite model to be exhaustively analyzed in order to determine its conformance to some properties. It has been successfully used to verify complex sequential circuits designs and communication protocols [Clarke et al., 1999]. A model can be represented as a graph where each vertex is a state of the system and edges are valid transitions between states. Verification is guaranteed to finish because its algorithms are based on monotonic predicate transformers that operate on finite sets of states. Thus, there exist both least and greatest fixpoint characterizations for the search space [Tarski, 1955]. This assumption guarantees that the algorithm converges in a finite number of steps. Another benefit of model checking is the generation of counter-examples in the case of the model does not conform to the specification. Search procedures check if a finite state transition system is or not a model for a given specification. Temporal logic model checking is the technique where the model is represented as finite state transition system and the specification is given in temporal logic. The main temporal logics employed are LTL [E. Allen Emerson, 1990] and CTL [E.M. Clarke and E.A. Emerson, 1981], linear and branching time logics respectively. States of a model can be represented either explicitly or by means of some symbolic representation. In the next sections we discuss basic representations and checking techniques,

namely explicit and symbolic model checking, present the main goals of our project and compare it with related works.

1.2 Research Goals

In this work, we propose a model for combining explicit and symbolic representations in an explicit-symbolic formal verification model. We intend to use explicit and symbolic techniques together to verify the same model and to make it possible to apply the most efficient technique to each aspect of the model. Given a specific system and a partitioning of its properties into explicit and symbolic subsets, we generate explicit and symbolic models. The explicit and symbolic subsets of properties can be determined accordingly to any partitioning policy. Such a partitioning can distinguish control-flow and data-flow aspects, for example. Induced by this partitioning, the explicit-symbolic model builds and integrates models for the explicit and symbolic aspects and then coordinates model checking algorithms over each one, taking advantage of their features. Note that our work is not focused on establishing partitioning policies but on supporting a given partitioning.

The explicit-symbolic model associates transitions in the explicit-state space with transitions in the symbolic space to make them consistent to the original model. Each state in the explicit-symbolic model is a pair composed of one state from the explicit model and one state from the symbolic model. So, computations on the explicit-symbolic model rely on computations on the underlying models. Traditional model checking algorithms are applied to these models as independently as possible. Whenever one of the models finishes computing some subformula, its results can be used to drive the search on the counterpart model.

1.3 Related Works

Our work is devoted to the integration of different formal methods approaches in order to confer more power to the verification of systems. Some significant efforts pursued this goal in the past years, as done by the Prototype Verification System [Owre et al., 1992] [Owre et al., 1998] and the Symbolic Model Prover [Berezin, 2002]. Such tools are concerned with expressiveness aspects and with the handling of infinite state spaces, as they put theorem proving and model checking to work in the same verification framework.

Concerning the model checking approach, the combined explicit-symbolic model relates with explicit and symbolic verifiers and their techniques as it integrates underlying explicit and symbolic models. Currently, SPIN [Holzmann, 1997b] and JPF [Visser et al., 2000] are well-known and largely used explicit-state model checkers. On the other hand, SMV [McMillan, 1993], NuSMV [Cimatti et al., 2000] and Verus [Campos, 1996] are representative and successful symbolic model checking tools. Existing algorithm optimizations from such tools, like the on-the-fly technique used by SPIN, are considered during the implementation of underlying models in order to improve model checking.

1.3.1 SLAM

The approach adopted in the SLAM project [Ball and Rajamani, 2002] is the one that more closely relates to our project, as it employs both explicit and symbolic models inside a same environment. SLAM extracts abstract models, known as Boolean programs, from C code and statically checks temporal properties of software. Also, SLAM uses predicate abstractions, symbolic reasoning and iterative refinement. Bebop is the part of SLAM liable for performing reachability analysis of Boolean programs. Boolean programs are abstraction of programs where the concrete states have been mapped to abstract states under evaluation of a finite set of predicates. The resulting program roughly corresponds to a C program with the same control-flow constructs, but where all variables have Boolean type. Because all variables have Boolean type, the state space of the program is finite. Consequently, reachability and termination are decidable for Boolean programs. More information on the predicate abstraction algorithm and its corresponding implementation can be found on [Ball et al., 2001]. Given a Boolean program, Bebop performs an inter-procedural data-flow analysis in order to determine reachability information [Ball and Rajamani, 2000]. Bebop represents control flow explicitly and sets of states implicitly using binary decision diagrams [Bryant, 1986]. The explicit representation of control-flow features is justified due to the usage of compiler optimization techniques. On the other hand, binary decision diagrams are used to symbolically represent the input and output behavior of procedures. They are used to represent sets of reachable states at a program point. A state contains the program counter and values to all the variables visible at that point.

Comparing to our explicit-symbolic model, SLAM lacks flexibility because control-flow and data-flow information must have explicit and symbolic representations, respectively. The proposed model is more general because it allows us to move variables between explicit and symbolic spaces according to any policy. Thus, we can experiment with a variety of combinations and choose the one that best fits the system needs (see section 6.5 for more details about the flexibility of the proposed model).

1.4 Main Contributions

The main contribution of our work is the development of a theoretical model and related algorithms for integrating explicit and symbolic representations into the same model-checking environment. The combined explicit-symbolic model is mainly intended to the verification of systems where both explicit-state and symbolic approaches can take advantage of individual aspects of the design. According to our work the task of explicitly model checking a system does not exclude the usage of symbolic improvements techniques anymore, and vice-versa. Explicit-state and symbolic representations have already been used together within a same environment, but our proposal makes it possible to choose representations according to any policy. The proposal of the combined explicit-symbolic model is intended to increase the efficiency of the formal verification methods and the range of problems we can deal with.

Together with the theoretical modeling, we have developed a prototype of the conceived explicit-symbolic model checker. The primary goal of our implementation is to confirm the practicability and importance of our approach. The current implementation has not been endowed with state-of-the-art improvement techniques, but it is useful as starting point in the development of a competitive explicit-symbolic model checker. By experimenting with this initial implementation we can drive efforts to promising fine-tuning of the framework and have insights about alternatives of cooperation between explicit and symbolic representations. Also, our work can be used for indicating classes of systems where either explicit or symbolic model checking produces better results.

Regarding the interconnection between the explicit and symbolic models, our work explores important aspects concerned with the partitioning of an original system and the composition and synchronization of its underlying models. Thus, it can be used to investigate the integration of other formal verification techniques and representations.

New explicit techniques have been implemented, one for compressing the state vector and another for storing the set of visited states by means of a cache of explicit states. The technique for compressing the state vector presents reasonable runtime and can produce high compression rates for cases where variable values assume short ranges. On the other hand, the cache of visited states is an alternative to the storage of visited states for designs with no tolerance for partial coverage and where the response time is critical.

Also, we discuss how descriptions in the Verimag Interchange Format [Bozga et al., 1999] can be used as input language for the explicit-symbolic model checker. The Verimag Interchange Format is able to describe systems with processes that execute in parallel and interact through point-to-point message passing, via communication buffers, and shared variables. The language also considers a rich set of typed variables, procedure calls, and dynamic processes, conferring expressiveness to the task of modeling a system. Additionally, system descriptions can be produced with the aid of graphical tools, and this can help increase the usability of the model checker. Because we use the Verimag Interchange Format as input language, our work offers another description language for explicit, symbolic and explicit-symbolic model checking.

Finally, our work promotes the comparison between the solutions and techniques offered by the explicit and symbolic approaches during the different stages of model checking a system. It is important to measure the impact that different representations have over the verification of different systems and choose the approach more competitive and efficient to each problem being checked.

The remainder of this document is organized as follows. Chapter 2 gives an overview of the process of model checking a computational system and discusses how temporal properties can be specified for a given model. The discussion is focused on computational tree logic, but we also consider linear temporal logic. Chapter 3 presents the explicit representation and its basic algorithms. The main techniques for improving explicit model checkers are presented, such as partial order reduction, bit-state hashing, minimized automata, compression of state vectors and static analysis. Chapter 4 discusses symbolic model checking and binary decision diagrams. Fixpoint characterizations are given for the symbolic

versions of the model checking algorithms. Chapter 5 presents the Interchange Format developed at Verimag labs, the description language of the combined explicit-symbolic model checker. These chapters give the background for the introduction of the explicit-symbolic modeling, in the following chapter. First, Chapter 6 focuses on the conception of the combined explicit-symbolic model and related algorithms, the theoretical basis of our work. After that, this chapter shows how the theoretical modeling for the combined model is adapted for descriptions given in the Interchange Format. Chapter 7 is dedicated to the implementation of the explicit-symbolic model checker, discussing important data structures and the overall process, from the collection of symbols to the construction of the combined model. Finally, Chapter 8 shows experimental results, final remarks and future works.

Chapter 2

Temporal Logic Model Checking

This chapter presents the model checking approach to the formal verification of concurrent systems. Initially we give an overview of the process of model checking a computational system. Next, we discuss how temporal properties can be specified for a given model. We focus on specifications expressed in the computational tree logic, a formalism used for describing temporal properties by considering a branching structure of time. Both syntactic and semantic aspects of the formalism are discussed. In addition, we give an overview of the linear-time temporal logic, another important formalism.

2.1 Introduction

Model checking is a formal verification approach where the system of interest is represented as a finite-state machine to be checked in order to determine its conformance to the given specification. Specifications are expressed by some temporal logic, a formalism that is able to infer how the system behaves along time. Usual temporal logics devoted to model checking include computational tree logic and linear time logic.

One of the main advantages of model checking is that the verification is fully automatic after the model has been constructed and the properties have been specified. The constructed model must be exhaustively examined in order to ensure that the system does not violate the requirements. The exploration of states often relies on depth-first or breadth-first searches. Another benefit of model checkers is the generation of counterexamples in the cases where the specification fails on the model. Counterexamples are very useful because they reveal the sequence of events that led the system to invalid states. Consequently, the importance of model checkers increases as the complexity of applications increases, especially for the critical ones.

The finite-state machine of the concurrent system can be viewed as a finite Kripke structure M, and the model checking procedure as an algorithm used to determine whether such a structure is a model of a particular formula f [Clarke et al.,]. Standard model checkers start from an initial state and recursively generate sequences of states by considering the

non-deterministic events of the system. This process continues either until the whole state space is explored, or until the model checker runs out of resources [Musuvathi, 2004]. The specification f holds in M if there exists a path starting at an initial state of M where f holds. The meaning of f depends on the temporal logic used for the specification. In this chapter, our discussion concentrates on computational tree logic, since the specifications on our explicit-symbolic model are expressed by using this formalism.

2.2 Temporal Logics

Temporal logics [Prior, 1957][Prior, 1967] are used to describe temporal ordering of events of a system, being used for describing how systems behave along time and for indicating the configurations that they can assume along the execution time. Each system configuration corresponds to one state of the system, being defined by the set of values assumed by the variables in the system at that time. In model checking, temporal logic properties are verified against a temporal model of the system. The model can be viewed as a state transition graph. Each state in the graph is defined by the set of propositions that are true on it. Instead of specifying properties over individual states, temporal logics are concerned with the specification of properties over sets of states and how the system changes from a state to another. Thus, the evaluation of temporal logical formulas requires all states of the sequence of execution to be considered. Consequently, the same formula can be true for some states and false for other states during the same execution. Because changes on states potentially change the evaluation of formulas, the system may present different properties at different states. So, a temporal logic formula is not statically true or false in a model, it can be true in one state of the model and false in another state.

Temporal logics offer different points of view with regard to the time structure. Branching-time temporal logics consider several alternative computation paths at a given point in time, and all the computation paths are considered simultaneously. So, branching time logics can express temporal properties involving several different branches of the state transition graph. On the other hand, linear-time temporal logics assume time as a sequence of executions of a system where each possible computation path is considered separately, reasoning about just one execution sequence. Temporal logics have been found to be very useful for describing and verifying computational systems, especially the concurrent ones, which continuously interact with the environment, such as operating systems, hardware design, medical systems, telecommunications and systems for air traffic management.

2.3 Computation Tree Logic

Computation Tree Logic [E.M. Clarke and E.A. Emerson, 1981] is a propositional temporal logic where time is assumed to have a branching, tree-like structure. Nodes in such a tree correspond to states in the system and sequences of states correspond to execution paths. Every state in the tree potentially has infinite successors and all successors are

considered by its formulas. Consequently, several states may be reached from each node in the tree by using alternative paths. At each moment, time may be split into alternate paths representing different possible futures. CTL is concerned with assertions on states and their interaction, considering past, present and future times on different computation paths. In CTL, time is not explicitly considered. Time information is introduced by temporal operators that specify formulas asserting indirect time constraints. CTL formulas express temporal information by asserting that some state is eventually reached through some computation path, or specifying that some property will always hold on the path or asserting that the next state presents some particular property.

CTL has been found to be very useful for the verification of concurrent, reactive and non-deterministic systems. One of its main applications has been the specification of properties to be verified by model checkers, one important formal verification approach. Model checkers supporting CTL specifications include SMV, PROD and the APNN toolbox. SMV, the Symbolic Model Verifier, is a model checker used for checking finite state systems against CTL specifications [K.L. McMillan, 1992]. The input language of SMV has been developed with the main purpose of checking descriptions of transition relations of finite Kripke structures, and it is able to model finite state systems ranging from completely synchronous to completely asynchronous ones. SMV also offers counter-example generation. On the other hand, PROD is a tool for efficient reachability analysis for Predicate/Transition Nets [Varpaaniemi et al., 1997]. PROD incorporates different advanced methods for generating reduced reachability graphs in order to face the state explosion problem, including partial-order techniques and techniques for exploiting symmetries. The stubborn set method is one of the approaches used for palliating the state space explosion problem. In addition, PROD supports on-the-fly verification of LTL formulas. PROD has been developed at the Digital Systems Laboratory at Helsinki University of Technology and has been used in industrial projects. The APPN toolbox, in turn, is a collection of algorithms for the combined functional and quantitative analysis of discrete event dynamic systems (DEDS) given in the Abstract Petri Net Notation [Bause et al., 1998]. The APPN toolbox includes methods for checking classical Petri net liveness and also considers checking of more general properties given by means of CTL and LTL temporal logics. APPN applications include computer networks, communication networks and logistic systems.

2.3.1 CTL Syntax

CTL is a propositional temporal logic because its formulas are composed of atomic propositions, logical connectives, path quantifiers and temporal operators. Path quantifiers are used to describe the branching structure of time in the computation tree, that is, they indicate what paths starting from a given state have the relevant property. The universal path quantifier, `A`, asserts that all the paths starting at the state have the considered property. The existential path quantifier, `E`, indicates that at least one path starting at the state has the considered property. Path quantifiers must be immediately followed by

temporal operators that describe properties on the paths specified by the quantifiers. Basic temporal operators include X, F, G and U. Their meanings are described below.

- X: the *next time* temporal operator requires that the next state on the path have the considered property.
- F: the *eventual* or *future* operator specifies that the property will eventually hold at some state on the path.
- G: the *always* or *globally* operator asserts that the given property holds at every state on the path.
- U: the *until* operator combines two properties, by requiring the second property to hold at some state on the path and the first property to hold at every preceding state on the path.

Path quantifiers and temporal operators are used in pairs in order to compose CTL operators like AX, EX, AF, EF, AG and EG. Let γ be a CTL specification. According to the definition, γ is either given by an atomic proposition ap or it is composed of temporal-logical operators applied to CTL subformulas:

$$\gamma \ ::= \texttt{ap} \mid \texttt{False} \mid \texttt{True} \mid (\neg\gamma) \mid (\gamma \vee \gamma) \mid (\gamma \wedge \gamma) \mid (\gamma \rightarrow \gamma) \mid$$
$$\texttt{AX}\ \gamma \mid \texttt{EX}\ \gamma \mid \texttt{AF}\ \gamma \mid \texttt{EF}\ \gamma \mid \texttt{AG}\ \gamma \mid \texttt{EG}\ \gamma \mid \texttt{A}[\gamma_1\ \texttt{U}\ \gamma_2] \mid \texttt{E}[\gamma_1\ \texttt{U}\ \gamma_2]$$

AX(γ) asserts that γ holds on every immediate successor of the current state of the system. AF(γ) specifies that γ will eventually hold in the future at some state of every path starting at the current state. AG(γ) requires that γ globally holds for all the paths starting at the current state. A[γ_1Uγ_2] specifies that for all paths starting at the current state, γ_2 holds at some state and γ_1 holds at every preceding state of the path until the current state is reached. Existential path quantifiers behave analogously, except for requiring just one path to have the property. More complex formulas can be expressed by grouping simpler ones:

- AG$\neg$($\gamma_1 \wedge \gamma_2$): γ_1 and γ_2 do not simultaneously hold in the system.
- AG($\gamma_1 \rightarrow$ AFγ_2): if γ_1 is true, so γ_2 will be true somewhere on every path.
- AG($\gamma_1 \rightarrow \gamma_2$): if γ_1 is true, so is γ_2.

All CTL operators can be expressed in terms of EX, EG and EU, the basic temporal operators:

- AXγ = $\neg$EX($\neg\gamma$)
- AF γ = $\neg$EG($\neg\gamma$)
- EF γ = E[True U γ]
- AG γ = $\neg$EF($\neg\gamma$)
- A[γ_1 U γ_2] = $\neg$E[$\neg\gamma_2$ U ($\neg\gamma_1 \wedge \neg\gamma_2$)]$\wedge\neg$EG$\neg\gamma_2$

2.3.2 CTL Semantics

The semantics of CTL is defined with respect to a computation tree, a state transition graph generated from a Kripke structure, a non-deterministic model based on states. Each state on a Kripke structure is defined by a set of variable values that represents the propositions holding during some snapshot of the modeled system. From [Clarke et al., 1999], a Kripke structure M over a set of atomic properties AP is defined as the tuple $M = (S, S_0, R, L)$ where

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a transition relation that must be total, that is, every state has at least one successor. In other words, there are no deadend states.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

A path starting at state s_0 in M is an infinite sequence of states $\pi = s_0 s_1 s_2 ...$ where $R(s_i, s_{i+1})$ holds for all $i \geq 0$. Given the Kripke structure M, the semantics of CTL is defined over the computation tree $M' = (S', S'_0, R', L')$, an infinite state graph obtained from M by unfolding its transitions according to the following rules:

- S' consists of all the infinite paths in M.
- $(\pi, \pi') \in R'$ if and only if $\pi = s_0 s_1 s_2 ... s_n$, $\pi' = s_0 s_1 s_2 ... s_n s_{n+1}$ and $(s_n, s_{n+1}) \in R$.
- S'_0 is formed by all the paths in M with only one initial state.
- For all $\pi = s_0 s_1 s_2 ... s_n$ in M, $L'(\pi) = L(s_n)$.

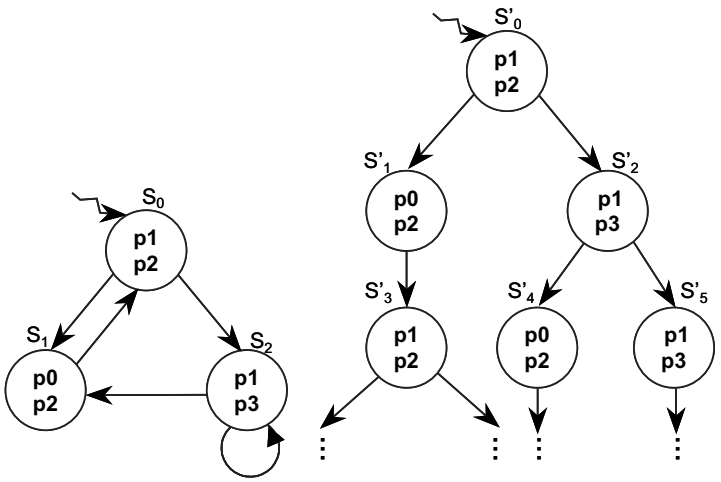
Thus, the set of states of M' is isomorphic to the set of infinite paths in M.

2.3.2.1 Example

Assume $M_e = (S, S_0, R, L)$ to be the Kripke structure defined over $AP = \{p_0, p_1, p_2\}$, a set of Boolean propositions, such that

- $S = \{s_0, s_1, s_2\}$
- $S_o = \{s_0\}$
- $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_2, s_1), (s_2, s_2)\}$
- $L = \{(s_0, \{p_1, p_2\}), (s_1, \{p_0, p_2\}), (s_2, \{p_1, p_3\})\}$

The state transition graph corresponding to M_e is depicted in figure 2.1(a). After applying the rules for converting Kripke structures into computation trees, we have the infinite state transition graph M'_e, partially depicted in figure 2.1(b).



(a) (b)

Figure 2.1: Kripke structure for M_e(a) and corresponding computation tree M'_e(b).

2.3.2.2 Defining the Semantics

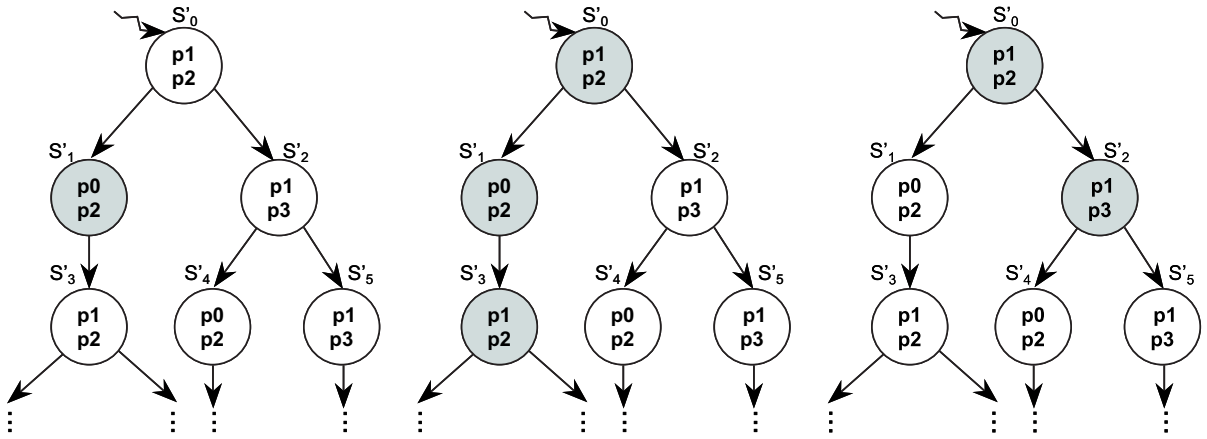
The standard notation indicates that a propositional formula γ is true at state s of a computation tree M as $M, s \vDash \gamma$. The relation $\vDash$ is inductively defined below.

$M,\ s_0 \vDash \texttt{ap} \quad \Leftrightarrow \quad \texttt{ap} \in L(s_0)$
$M,\ s_0 \vDash \neg\gamma \quad \Leftrightarrow \quad M,\ s_0 \nvDash \gamma$
$M,\ s_0 \vDash \gamma_1 \vee \gamma_2 \quad \Leftrightarrow \quad M,\ s_0 \vDash \gamma_1$ or $M,\ s_0 \vDash \gamma_2$
$M,\ s_0 \vDash \gamma_1 \wedge \gamma_2 \quad \Leftrightarrow \quad M,\ s_0 \vDash \gamma_1$ and $M,\ s_0 \vDash \gamma_2$
$M,\ s_0 \vDash \gamma_1 \rightarrow \gamma_2 \quad \Leftrightarrow \quad$ if $M,\ s_0 \vDash \gamma_1$ then $M,\ s_0 \vDash \gamma_2$
$M,\ s_0 \vDash \texttt{AX}\ \gamma \quad \Leftrightarrow \quad \forall\pi = s_0 s_1 ...,\ M, s_1 \vDash \gamma$
$M,\ s_0 \vDash \texttt{EX}\ \gamma \quad \Leftrightarrow \quad \exists\pi = s_0 s_1 ...,\ M, s_1 \vDash \gamma$
$M,\ s_0 \vDash \texttt{AF}\ \gamma \quad \Leftrightarrow \quad \forall\pi = s_0 ...,\ \exists j \geq 0,\ M, s_j \vDash \gamma$
$M,\ s_0 \vDash \texttt{EF}\ \gamma \quad \Leftrightarrow \quad \exists\pi = s_0 ...,\ \exists j \geq 0,\ M, s_j \vDash \gamma$
$M,\ s_0 \vDash \texttt{AG}\ \gamma \quad \Leftrightarrow \quad \forall\pi = s_0 ...,\ \forall i \geq 0,\ M, s_i \vDash \gamma$
$M,\ s_0 \vDash \texttt{EG}\ \gamma \quad \Leftrightarrow \quad \exists\pi = s_0 ...,\ \forall i \geq 0,\ M, s_i \vDash \gamma$
$M,\ s_0 \vDash \texttt{A}[\gamma_1 \texttt{U} \gamma_2] \quad \Leftrightarrow \quad \forall\pi = s_0 ...,\ \exists i[i \geq 0,\ M, s_i \vDash \gamma_2$ and $\forall j[0 \leq j < i \rightarrow M, s_j \vDash \gamma_1]]$
$M,\ s_0 \vDash \texttt{E}[\gamma_1 \texttt{U} \gamma_2] \quad \Leftrightarrow \quad \exists\pi = s_0 ...,\ \exists i[i \geq 0,\ M, s_i \vDash \gamma_2$ and $\forall j[0 \leq j < i \rightarrow M, s_j \vDash \gamma_1]]$

Usage of basic CTL operators is illustrated in figure 2.2, obtained by considering the computation tree previously shown in figure 2.1(b). Shaded states are those that determine the validity of the formula on each computation tree.

2.3.3 CTL Expressiveness

CTL is sufficiently expressive for the formulation of an important set of temporal properties, allowing *safety*, *liveness*, *deadlock* freedom and *fairness* to be specified in a concise



(a) M, $s'_0 \vDash \texttt{EX}(p0)$ (b) M, $s'_0 \vDash \texttt{EG}(p2)$ (c) M, $s'_0 \vDash \texttt{E}\ (p2\ \texttt{U}\ p3)$

Figure 2.2: Semantics of CTL formulas.

syntax [K.L. McMillan, 1992]. Safety properties assert that nothing bad will happen on the system. Safety properties may be used for specifying that the system will not reach undesired states, no matter what the supplied inputs were neither the executed actions. Liveness properties are used for asserting that something good will happen, that is, they specify that desired states are eventually reached or, for example, that some output is necessarily produced. On the other hand, specifications may be used for asserting that deadlocks do not occur. Deadlocks occur when the system is prevented from taking any action, that is, when no transition is possible since all activation conditions are false. Deadlock freedom specifications are used to make sure that deadend states will never be reached. Finally, fairness constraints are used to remove unwanted behavior from a system by considering correctness along fair computation paths only. Fairness constraints may be used to specify that independent processes will progress, by asserting that each process will be executed infinitely often, for example.

Fairness constraints cannot be directly expressed by CTL formulas. For expressing fairness, the original CTL semantics must be modified in order to restrict the set of states considered by model checking. The restricted set of states describes the behavior of the system when considering only fair executing paths. Each fairness condition specifies a set of states in the system and requires that these states must be traversed infinitely often in any acceptable behavior. Fair paths are those where each fairness constraint is true infinitely often. The CTL semantics only takes fair paths into account when considering fair Kripke structures. A set of fairness conditions is often called Büchi acceptance conditions or Büchi fairness constraints. Although fairness constraints are a very useful mechanism for refining the verification, they should be carefully used because some undesired properties can remain undisclosed when only part of the state space is explored during the verification. Additionally, many fairness constraints may reduce the efficiency of the verification.

2.4 Linear-Time Temporal Logic

Linear-time temporal logic [E. Allen Emerson, 1990] assumes time as a sequence of executions of a system where each possible computation path is considered separately, reasoning about just one execution sequence. Instead of being interpreted over computation trees, LTL formulas are interpreted with respect to individual computation paths. In other words, linear time temporal logics express temporal properties over a linear execution sequence of the system.

2.4.1 LTL Syntax

In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives and the temporal operators. Different from CTL, where each temporal operator must be prefixed with a path quantifier, propositional connectives and temporal operators may be nested in a different manner in LTL. An LTL formula γ is recursively defined as shown below:

$$\gamma \texttt{ ::= ap | False | True | } (\neg\gamma) \texttt{ | } (\gamma \vee \gamma) \texttt{ | } (\gamma \wedge \gamma) \texttt{ | } (\gamma \rightarrow \gamma) \texttt{ |}$$
$$(\texttt{X}\gamma) \texttt{ | } (\texttt{F}\gamma) \texttt{ | } (\texttt{G}\gamma) \texttt{ | } (\gamma_1 \texttt{ U } \gamma_2)$$

where `ap` is an atomic proposition, `X`, `F`, `G` and `U` are the temporal operators previously defined for CTL formulas. Note that LTL formulas have no explicit path quantifiers. An LTL formula is considered to be true over all computation paths, that is, LTL formulas are implicitly universally path quantified. Each LTL formula γ can be though of as `A(`γ`)`.

2.4.2 LTL Semantics

The semantics of LTL formulas is defined with respect to computation paths of a Kripke structure. Let $M = (S, S_0, R, L)$ be the Kripke structure defined over the set of atomic propositions AP. Assume $\pi = s_0 s_1 s_2 ...$ to be a computation path where $R(s_i, s_{i+1})$ holds for all $i \geq 0$ and $\pi^i = s_i s_{i+1} s_{i+2} ...$ is the suffix of π starting at s_i. The relation of satisfaction $M, \pi \vDash \gamma$ for the computation path π and LTL formula γ is inductively defined:

$$\begin{array}{lcl}
M, \pi \vDash \texttt{ap} & \Leftrightarrow & \texttt{ap} \in L(s) \text{ where } s \text{ is the first state of } \pi \\
M, \pi \vDash \neg\gamma & \Leftrightarrow & M, \pi \nvDash \gamma \\
M, \pi \vDash \gamma_1 \vee \gamma_2 & \Leftrightarrow & M, \pi \vDash \gamma_1 \text{ or } M, \pi \vDash \gamma_2 \\
M, \pi \vDash \gamma_1 \wedge \gamma_2 & \Leftrightarrow & M, \pi \vDash \gamma_1 \text{ and } M, \pi \vDash \gamma_2 \\
M, \pi \vDash \gamma_1 \rightarrow \gamma_2 & \Leftrightarrow & \text{if } M, \pi \vDash \gamma_1 \text{ then } M, \pi \vDash \gamma_2 \\
M, \pi \vDash \texttt{X } \gamma & \Leftrightarrow & \pi^1 \vDash \gamma \\
M, \pi \vDash \texttt{F } \gamma & \Leftrightarrow & \exists i \geq 0, \pi^i \vDash \gamma \\
M, \pi \vDash \texttt{G } \gamma & \Leftrightarrow & \forall i \geq 0, \pi^i \vDash \gamma \\
M, \pi \vDash \gamma_1 \texttt{ U } \gamma_2 & \Leftrightarrow & \exists i [i \geq 0, \pi^i \vDash \gamma_2 \text{ and } \forall j [0 \leq j < i, \pi^j \vDash \gamma_1]]
\end{array}$$

2.4.3 LTL Expressiveness

LTL can also express reachability, safety, liveness and deadlock freedom properties as CTL. However, the expressive powers of CTL and LTL are different. There are assertions that can be expressed in CTL but not in LTL and vice-versa. With CTL it is possible to specify that for all paths there always exists a path such that some property γ will be true in the future, that is `AG(EF`γ`)`, but we cannot express the same in LTL. On the other hand, LTL is able to express that for all paths the property γ will globally hold in the future, that is `A(FG`γ`)`, but we cannot assert the same with CTL. Both formalisms can be used for specifying properties over concurrent systems.

CTL and LTL are not capable of expressing some temporal properties, such as `E(GF`γ`)`. In such cases CTL*, a more expressive temporal logic can be used. CTL* allows path quantifiers to be followed by arbitrary linear time formulas and Boolean combinations and nested formula to be applied over path formulas. More information on CTL* can be found in [E. Allen Emerson, 1990]. Note that the choice of the temporal logic to be employed in a verification framework must consider both the kind of behavior to be checked and the efficiency of the checking procedure. Usually, more complex temporal logics require more expensive checking procedures.

Chapter 3

Explicit Model Checking

Original model checkers have been devised for operating over explicit representations by means of algorithms that operate on individual states, one state at a time. Such explicit model checkers have suffered from the inherent state explosion problem, which occurs when the size required for the representation prevents the system to be checked. Along the years, several improvements have been developed for minimizing the state explosion problem and have extended the power of the explicit model checkers. Such improvements have refined the explicit representation as well as the related algorithms. Nowadays, explicit model checkers are quite competitive and seem to be the best choice for the verification of some problems, such as those with intensive data transformations and software systems.

3.1 Introduction

Explicit-state model checking is a type of model checking in which state exploration systematically enumerates the possible states of the system, performing verifications over individual states, one state at a time. Explicit state checkers often use either a breadth-first or depth-first procedure in order to perform exhaustive state space searches. Explicit states are traditionally stored in hash tables and algorithms operate by labeling each individual state with sub-formulas which are true in that state. Originally, model-checking techniques were based on explicit-state representations, preceding symbolic model checking.

Explicit methods tend to present relatively more predictable efficiency and memory behaviors. Explicit representations have shown success in dealing with concurrent systems with few processes, where the number of states is usually small. On the other hand, explicit model checkers need substantial manual abstraction and state reduction to minimize the state explosion problem and to be practical for industrial size designs, one of its drawbacks [Chan, 1999]. The state explosion problem occurs when checkers run out of storage due to the size of the representation employed for modeling a system. Other approaches to defeat this problem include partial order reduction, bit-state hashing and the on-the-fly construction of models.

Currently, SPIN [Holzmann, 1997b] and Java Path Finder [Visser et al., 2000] are representative explicit-state model checkers. SPIN is the most well-known and widely used explicit-state model checker, endowed with several optimization algorithms that settled reference techniques for explicit model checking. More information on SPIN is given further in this chapter. Java Path Finder has been developed by NASA and checks LTL properties over Java byte-code. Both model and specification are described by means of Java statements. Java Path Finder deals with the state space explosion using techniques like symmetry reduction, partial order reduction, abstraction, slicing and partial evaluation. Java Path Finder also uses dynamic analysis techniques for data race detection and deadlock detection.

3.2 Kripke-Based Explicit Model Checking

Although Kripke structures may be used to model system to be checked against properties given both in CTL and LTL logics, this section is focused on algorithms developed for CTL specifications. Given the Kripke structure $M = (S, S_0, R, L)$, algorithms recursively label states with valid properties as shown by the following CTL-based procedures for explicit states [Clarke et al., 1999]. Assume that each state $s \in S$ has a set of labels $label(s)$ with the set of properties true on the state. Initially, each set of labels $label(s)$ contains only the labels given by the labeling function $L(s)$. Let f be the CTL formula to be checked. As subformulas of f are being visited by algorithms, the set of labels of the states in the model are updated. The following algorithms consider that f is given in terms of atomic propositions and basic CTL operators, $\neg$, $\vee$, `EX`, `EU` and `EG`. Note that the set of labels of each state s contains labels for atomic propositions, given by $L(s)$.

Algorithms for negation and disjunction are the simpler ones, being presented in procedures `checkNot(`f`)` and `checkOr(`f_1`,`f_2`)`, respectively. `checkNot(`f`)` adds the label for $\neg f$ to states that are not labeled by f.

```
01 procedure checkNot(f)
02  begin
03    T := {s|f ∉ label(s)};
04    for all t ∈ T do label(t):= label(t) ∪ {¬f};
05  end
```

On the other hand, `checkOr(`f_1`,`f_2`)` adds the label $f_1 \vee f_2$ to states that are labeled either by f_1 or f_2. Both procedures, `checkNot` and `checkOr`, require time $O(|S|)$.

```
01 procedure checkOr(f1,f2)
02  begin
03    T := {s|f1 ∈ label(s) or f2 ∈ label(s)};
```

```
04    for all t ∈ T do label(t):= label(t) ∪ {f1 ∨ f2};
05  end
```

For formulas checkEX(f), the algorithm labels with EXf states that have some successor labeled by f. checkEX(f) requires time $O(|S| + |R|)$.

```
01 procedure checkEX(f)
02  begin
03    T := {t|R(t, s) and f ∈ label(s)};
04    for all t ∈ T do label(t):= label(t) ∪ {EXf};
05  end
```

For formulas E(f_1Uf_2), the algorithm first finds all states that are labeled by f_2. After that, it uses the converse of the transition relation R in order to find states that can be reached by a path in which each state is labeled by f_1. Such states must be labeled with E(f_1U f_2). This algorithm, illustrated by procedure checkEU below, requires time $O(|S| + |R|)$.

```
01 procedure checkEU(f1, f2)
02  begin
03    T := {s|f2 ∈ label(s)};
04    for all s ∈ T do label(s):= label(s) ∪ {E[f1 U f2]};
05    while T ≠ ∅ do
06      begin
07        choose s ∈ T;
08        T := T\{s};
09        for all t such that R(t, s) do
10          if E [f1 U f2] ∉ label(t) and f1 ∈ label(t) then
11            begin
12              label(t) := label(t) ∪ {E[f1 U f2]};
13              T := T ∪ {t};
14            end
15      end
16  end
```

The algorithm for formulas checkEG(f) is based on the decomposition of the graph into nontrivial strongly connected components. A strongly connected component, SCC, is a maximal set of vertices within a graph such that there is a path between any pair of such vertices and the path is completely contained within this set of vertices. Nontrivial strongly connected components are those where either there exists more than one vertex in the set or there exists one vertex with a self-loop.

```
01 procedure checkEG(f)
02  begin
03    S' := {s|f ∈ label(s)};
04    SCC := {C|C is a nontrivial SCC of S'};
05    T := ⋃SCC;
06    for all s ∈ T do label(s) := label(s) ∪ {EGf};
07    while T ≠ ∅ do
08      begin
09        choose s ∈ T;
10        T := T\{s};
11        for all t such that t ∈ S' and R(t,s) do
12          if EGf ∉ label(t) then
13            begin
14              label(t) := label(t) ∪ {EGf};
15              T := T ∪ {t};
16            end
17      end
18  end
```

First, the procedure `checkEG(`f`)` finds the set of states labeled with f, as shown in line 03. After that, line 04 computes nontrivial strongly connected components, by considering the Kripke structure $M' = (S', S'_0, R', L')$ obtained by restricting S to states where f holds. Tarjan's strongly connected components algorithm may be used for that purpose [Tarjan, 1972]. Lines $05 - 06$ label the states within the SCCs with `EG`f. Finally, the loop spanning from lines 07 to 17 uses the converse of R' for finding predecessor of states already labeled with `EG`f where f holds. The procedure finishes after all those states have been labeled. By considering the usage of Tarjan's algorithm, the overall time required is $O(|S| + |R|)$.

After processing all subformulas, each state has been labeled with subformulas true on it. So, it suffices to check if f belongs to the labels of some initial state for checking the conformance of the model to the specification, that is, f holds on M if and only if $f \in label(s_0)$, where $s_0 \in S_0$.

3.3 Automaton-Based Explicit Model Checking

Usually LTL model checkers rely on operations over Büchi automata, instead of modeling the system to be checked as a Kripke structure. According to such an approach, both the system and its specification are described by means of automata. Thus, correctness

properties are checked by determining the existence of accepting cycles on the automaton resulting from a depth-first search procedure. The main benefit of such approach is the possibility of using on-the-fly techniques for the verification of partial representations.

3.3.1 Büchi Automata

Because computation models often consider infinite executions, the automata used for the explicit model checking must accept infinite strings of symbols or words, as done by Büchi automata [Büchi, 1960]. A Büchi automaton $\mathcal{B}$ is formally described by a tuple $(\Sigma, Q, \Delta, Q^0, F)$ where:

- Σ is a finite alphabet.
- Q is a finite set of states.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
- $Q^0 \subseteq Q$ is the set of initial states.
- $F \subseteq Q$ is the set of accepting states.

Given the transition $(q, a, q') \in \Delta$, it means that $\mathcal{B}$ moves from state q to state q' by consuming the input symbol a. Each string of symbols corresponds to one or more infinite paths on $\mathcal{B}$. Let $w = a_0, a_1, ... \in \Sigma^\omega$ be an infinite word such that $|w| = \omega$. The language $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$ corresponds to the set of infinite words accepted by the Büchi automaton $\mathcal{B}$. A run ρ of $\mathcal{B}$ on w is an infinite sequence $q_o, q_1, ...$ of states in Q, such that $q_0 \in Q^0$ and $(q_i, a_i, q_{i+1}) \in \Delta$ for all $i \geq 0$. Let $inf(\rho)$ denote the set of states that appear infinitely often in the run ρ. The run ρ is accepting if and only if $inf(r) \cap F \neq \emptyset$. The word $w \in \Sigma^\omega$ is accepted by $\mathcal{B}$ if and only if $\mathcal{B}$ has an accepting run on w. The language of the automaton $\mathcal{B}$ is empty whenever $\mathcal{L}(\mathcal{B}) = \emptyset$. The language of the Büchi automaton is non-empty if and only if an accepting state q' can be reached from some initial state $q_0 \in Q$, such that q' can reach itself by a non-empty sequence of transitions.

3.3.1.1 From Kripke Structures to Büchi Automata

Let $M = (S, S_0, R, L)$ be a Kripke structure over a set of atomic propositions AP. The equivalent Büchi automaton $\mathcal{B} = (\Sigma, Q, \Delta, Q^0, F)$ is defined as shown below:

- $\Sigma = 2^{AP}$
- $Q = S \cup \{\iota\}$
- $\forall q, q' \in Q$, $a \in \Sigma$, we have that $(q, a, q') \in \Delta$ if and only if:
  - $L(q') = a$ and ($q = \iota$ and $q' \in S_0$), or

- $L(q') = a$ and $((q, q') \in R)$

- $Q^0 = \{\iota\}$
- $F = S \cup \{\iota\}$

3.3.1.2 Checking Properties Over Büchi Automata

Let $M = (S, S_0, R, L)$ be a Kripke structure over a set of atomic properties AP and f be a specification for a given system. Suppose that M and f are recognized by automata $\mathcal{B}_M$ and $\mathcal{B}_f$, respectively. Assume $\mathcal{L}(M)$ and $\mathcal{L}(f)$ to be the languages accepted by $\mathcal{B}_M$ and $\mathcal{B}_f$, in the order given. M satisfies the specification f if $\mathcal{L}(M) \subseteq \mathcal{L}(f)$, case where the behavior of M is found to be among the behaviors allowed by f. Equivalently, for checking the conformance of M to f it suffices to prove that $\mathcal{L}(M) \cap \overline{\mathcal{L}(f)} = \emptyset$, where $\overline{\mathcal{L}(f)} = \Sigma^\omega - \mathcal{L}(f)$. Because the languages accepted by Büchi automata are closed under complement and intersection, for each automaton recognizing $\mathcal{L}(f)$ there is a Büchi automaton that recognizes its complement $\overline{\mathcal{L}(f)}$, and given two Büchi automata recognizing $\mathcal{L}(M)$ and $\overline{\mathcal{L}(f)}$, there is a Büchi automaton that recognizes the language $\mathcal{L}(M) \cap \overline{\mathcal{L}(f)}$. So, it is always possible to check if $\mathcal{L}(M) \cap \overline{\mathcal{L}(f)} = \emptyset$. If so, we have that M satisfies f. On the other hand, if $\mathcal{L}(M) \cap \overline{\mathcal{L}(f)} = C \neq \emptyset$, f does not hold on M and C is a counterexample for the non-satisfaction of f by M. Below, we outline the steps involved with the automaton-based model checking:

1. Convert the Kripke structure M into a Büchi automaton $\mathcal{B}_M$ over the alphabet 2^{AP}.
2. Generate $\mathcal{B}_{\overline{f}}$, the Büchi automaton for $\overline{\mathcal{L}(f)}$. $\mathcal{B}_{\overline{f}}$ can be computed in different ways:
   (a) Given $\mathcal{B}_f$, generate the complement automaton $\mathcal{B}_{\overline{f}}$.
   (b) Supply the automaton for $\mathcal{B}_{\overline{f}}$ directly.
   (c) Because every LTL formulas can be translated into equivalent Büchi automata, specify properties in an LTL formula and convert the negated formula into an equivalent Büchi automaton.
3. Generate the product Büchi automaton $\mathcal{B}_P = \mathcal{B}_M \cap \mathcal{B}_{\overline{f}}$, which accepts all the words in $\mathcal{L}(\mathcal{B}_M) \cap \mathcal{L}(\mathcal{B}_{\overline{f}})$.
4. Check whether $\mathcal{L}(\mathcal{B}_P) = \emptyset$.
   (a) Case $\mathcal{L}(\mathcal{B}_P) = \emptyset$, the specification f holds on the automaton $\mathcal{B}_M$ and on the Kripke structure M.
   (b) Case $\mathcal{L}(\mathcal{B}_P) = C \neq \emptyset$, C is counterexample for the non-satisfaction of f by M. In this case, generate a word $w = u.v^\omega$ accepted by $\mathcal{B}_P$, where u is a prefix leading to an accepting state and v is a finite loop on the path.

3.3.1.3 Intersection of Büchi Automata

Let $\mathcal{B}_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, F_1)$ and $\mathcal{B}_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2)$ be Büchi automata. The Büchi automaton accepting $\mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$, also called the product automaton, can be defined as $\mathcal{B}_1 \cap \mathcal{B}_2 = (\Sigma, Q_1 \times Q_2 \times \{0,1,2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\})$, such that $(< r_i, q_j, x > , a, < r_m, q_n, y >) \in \Delta$ if and only if we have:

- Transitions of $\mathcal{B}_1$ and $\mathcal{B}_2$ agree, that is, $(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$.
- x and y are determined according to the accepting conditions of $\mathcal{B}_1$ and $\mathcal{B}_2$:
  - If $x = 0$ and $r_m \in F_1$, then $y = 1$.
  - If $x = 1$ and $q_n \in F_2$, then $y = 2$.
  - If $x = 2$ then $y = 0$.
  - Otherwise, $y = x$.

The correspondence between components x and y makes sure that both sets of accepting states of $\mathcal{B}_1$ and $\mathcal{B}_2$ occur infinitely often. Initially, x is set to 0. It changes from 0 to 1 when an accepting state of the first automaton is reached. After that, x changes from 1 to 2 when an accepting state of the second automaton is reached. Then, x goes back to 0.

3.3.1.4 Emptiness Check for Büchi Automata

Let $\mathcal{B} = (\Sigma, Q, \Delta, Q^0, F)$ be a Büchi automaton and ρ be an accepting run. So, ρ contains infinitely many accepting states from F. Because Q is finite, there is a prefix ρ' of ρ such that every state in ρ' occurs infinitely often and every state in ρ' is reachable from every other state in ρ'. Thus, the states in ρ' form a strongly connected component of $\mathcal{B}$ that is reachable from an initial state and contains an accepting state. Consequently, $\mathcal{L}(\mathcal{B}) \neq \emptyset$ if and only if there is an accepting state $q \in F$ that is reachable from an initial state and can be reached from itself through a cycle. Therefore, the problem of determining $\mathcal{L}(\mathcal{B}) = \emptyset$ is equivalent to finding a strongly connected component within the state graph of $\mathcal{B}$.

3.3.2 SPIN

SPIN [Holzmann, 1997b] is one of the most important automaton-based model checkers. SPIN is a generic simulation and verification system focused on the design and verification of asynchronous control in software systems, rather than synchronous control in hardware systems. SPIN specifications are given in PROMELA, a language which accepts correctness properties specified in LTL. Models specified in PROMELA must be bounded and present countable distinct behaviors because there are no general decision procedures for unbounded systems. Descriptions of concurrent systems are composed of one or more process templates and at least one process instantiation. Process templates are used for

defining the behavior of different processes and can be used by a running process for instantiating other processes. In SPIN, verification is subdivided into two aspects, safety properties and liveness properties.

Process templates are translated into finite automata and the global behavior of the system is obtained by computing an asynchronous interleaving product of such automata. The resulting product is also an automaton. On the other hand, correctness properties given in LTL logic are converted into Büchi automata by means of an on-the-fly construction. This process must consider the negated original correctness properties in order to produce automata that represent erroneous behaviors, since the negated properties stand for undesirable behaviors on the system. In other words, the automaton produced from the negated specification represents infinite executions where the system presents bad behaviors. Given the automaton for the global behavior of the system and the automaton for the negated specification, SPIN performs the synchronous product between them. The resulting automaton represents erroneous behaviors found on the modeled system. If the language of the product automaton is empty, the LTL specification holds on the model. Otherwise, the product automaton represents behaviors where the original specification fails.

For determining the emptiness of the automaton which represents the combined execution of the system and the specification, SPIN looks for accepting cycles on it. Executions are considered to be accepted by a Büchi automaton if and only if the execution forces the automaton to pass through at least one accepting state infinitely often. So, it suffices to prove that there is no accepting cycle on the combined automaton. For that, SPIN uses an optimized nested depth-first search [Holzmann et al., 1996]. In cases where an accepting cycle is found on the combined automaton, the accepting cycle corresponds to a counterexample.

The asynchronous interleaving of product of individual concurrent processes, the synchronous product of the global behavior of the system and the automaton for the specification, and the detection of accepting cycles on the final automaton is accomplished by a single depth-first search procedure. Such procedure is fundamental for the verification procedure and is required to be compatible with all modes of verification in SPIN, including exhaustive search, bit-state hashing and partial-order reduction. Below, we show the on-the-fly model-checking algorithm used by SPIN.

```
01 procedure dfs()
02   begin
03     s = top(Stack);
04     if error(s)
05       reportError();
06     for all transitions t enabled in s do
07       begin
08         s' = successor(s) after executing t;
```

```
09          if s' is not in StateSpace then
10            begin
11              Enter s' into StateSpace;
12              Push s' onto Stack;
13              dfs();
14            end
15        end
16      Pop s from Stack
17    end
18
19  procedure main()
20    begin
21      Enter s0 into StateSpace;
22      Push s0 onto Stack;
23      dfs();
24    end
```

Because SPIN is an on-the-fly model checker, the construction of the model and the verification of properties occur simultaneously on the algorithm above. After each state s has been generated, the model checker verifies the current temporal logic subformula against s. Basic implementations of the algorithm consider the set *StateSpace*, which contains all states found during the search, to be stored in a hash table. The states already visited by the algorithm are stored on *Stack* in order to keep track of the current execution path. Lines 21 and 22 are used to insert the initial state s_0 both into the state space and into the stack of already visited states. When all successors of the current state s have been visited, s is removed from the stack and the previous state on the stack becomes the current state. The exploration of the state space proceeds recursively by following successors of s. Each explicit state is uniquely identified by a state vector. Usually, state vectors contain information on global variables, contents of communication channels and the local variables and process counter for each process in the system.

3.4 Explicit Model Checking Optimization Algorithms

Several reduction algorithms have been proposed to make explicit model checking more effective, such as partial order reduction, bit-state hashing, minimized automaton encoding of states, state vector compression, dataflow analysis and slicing. Such optimizations are essential for the effectiveness of model checking algorithms and are concerned with the minimization of both memory and time costs. Although some optimizations are concerned

with the automaton approach, some may also be applied to systems modeled as Kripke structures.

3.4.1 Partial Order Reduction

Partial order reduction is used to reduce the number of reachable states that must be explored to complete a verification and for minimizing the state explosion problem. Partial order reduction techniques are based on the observation that the validity of a property is often insensitive to the order in which concurrent and independent transitions are interleaved [Holzmann and Peled, 1994]. Instead of generating an exhaustive state space that include all execution sequences, the verifier can generate a reduced state space by ignoring replicated execution sequences that are indistinguishable for a given property. Also, processes that deal with local variables only can be verified apart from other processes, reducing the state space to be explored during the verification. Unfortunately it is hard to determine exclusive access to variables. Partial order reduction methods rely on static reduction techniques to decide where partial order reduction can be applied.

3.4.2 Bit-State Hashing

Bit-state hashing is an alternative to the storage of states in traditional hash tables, being a technique concerned with reductions on the space used for storing the set of reachable states [Holzmann, 1995]. Bit-state hashing views every bit in memory as a bucket in a hash table. Instead of maintaining a set of state descriptors, bit-state hashing maintains a table of bits. Initially, the bits of the table are set to zero. During the computation of the representation, each state is hashed onto a fixed number of buckets. It means that states are represented by hash addresses. Because only the fact that a bucket has become filled is recorded, there is no collision detection. Consequently, when two states hash onto the same buckets, they are considered to be the same state.

Although bit-state hashing does not guarantee an exhaustive analysis, it can perform relatively high coverage verifications within memory resources that may be orders of magnitude smaller than that required for exhaustive verifications. In comparison to classical random simulation techniques, it is always better to use bit-state hashing because the coverage is never worse than that achieved with random simulation, and it is usually much better [Holzmann, 1997b]. The accuracy of bit-state hashing can be improved by using more hash functions to hash each state onto more bits of the table of bits. The bigger the number of hash functions, the more accurate the coverage and higher the lookup cost [Dillinger and Manolios, 2004]. SPIN [Holzmann, 1997b] utilizes two hash functions as a compromise between runtime expense and coverage. Bit-state hashing has proved to achieve high coverages with low memory requirements when compared to exhaustive searches.

3.4.3 Minimized Automata

Minimized automata [Holzmann and Puri, 1999] provide a dynamic structure for storing states in a minimized deterministic finite automaton, being another alternative for storing sets of states. Instead of storing states in the hash-table, minimized automata recognize stored states. Traditional insertions and removals for updating hash tables are replaced by updates of the recognizer, changing dynamically the deterministic finite automaton. The automaton update procedure must preserve the minimality of the recognizer. Although very effective with regard to memory, minimized automata introduce sensitive computational costs. The idea behind minimized automata closely relates to binary decision diagrams [Bryant, 1986].

3.4.4 State Vector Compression

State vector compression techniques are intended for reducing the state vector size, by storing a compressed version of the state vector in the state space without losing information. The compression technique introduced by SPIN is based on the premise that processes and channels in PROMELA specifications have a small number of local states when compared to the global number of states [Holzmann, 1997a]. So, the descriptor of each component of a specification must be stored separated within the global state descriptor, with indexes for accessing the respective descriptor of the component during the verification. Naturally, the reduction on the memory requirements incur increase of execution time.

3.4.5 Static Analysis

Static analysis is used to improve the efficiency of the search by removing parts of the system that are irrelevant for checking a given property. Static analysis methods include slicing algorithms [Santiago, 2003], data dependence analysis and interprocedural flow analysis, useful for partial order reductions, elimination of dead variables and merging of statements. Formal verification tools concerned with static analysis can greatly benefit from Bandera [Corbett et al., 2000], an integrated collection of program analysis and transformation components for the automatic extraction of safe, compact finite-state models from program source code. Bandera takes as input Java code and produces models in the input language of some verifiers, such as SPIN and SMV.

Chapter 4

Symbolic Model Checking

Explicit model checking suffers from the state explosion problem, which occurs when checkers run out of storage due to the size of the representation employed for modeling a system. This problem motivated the creation of representations concerned with the exploration of regularities in the state space for producing more compact representations. Additional algorithms have been devised for dealing with such symbolic representations. Together, such representations and their algorithms originated an alternative to the verification of systems, the symbolic model checking, the approach discussed in this chapter.

4.1 Introduction

The state space explosion problem, main obstacle to the model checking techniques, motivated the creation of symbolic representations for the state transition graphs. Symbolic models explore regularity in the state space aiming to produce more compact representations [Huth and Ryan, 2000]. Instead of exploring states individually as explicit model checking does, symbolic model checking uses efficient encoding of Boolean logical formulas to represent and to explore sets of states atomically. So, symbolic model checking usually allow us to verify systems with a much higher number of states when compared to explicit model checking [McMillan, 1992]. In order to perform symbolic model checking, sets of states and transitions are represented by their characteristic functions. States of a symbolic model are described by the set of variables of the system that are true on it. On the other hand, transitions can be defined in terms of the variables holding in the current and next states. Computational representations of characteristic functions must be efficient and should provide essential operations, as conjunction, disjunction, equality tests, existential quantification and substitution. Binary Decision Diagrams [Bryant, 1986], known as BDDs, are a very efficient symbolic representation of propositional logical functions.

SMV [McMillan, 1993], NuSMV [Cimatti et al., 2000] and Verus [Campos, 1996] are representative symbolic model checkers. SMV is a finite state model checker that uses BDDs as symbolic representation and CTL as temporal logic. It computes fixpoints that

correspond to CTL properties using BDDs to represent set of states. Initially SMV was developed for hardware verification, but it has been used in verification of software specifications and protocols, among other applications. NuSMV originated from the re-engineering, reimplementation and extension of SMV. Its source code is significantly clearer and well documented compared to the SMV one. Verus is a model checker also derived from SMV and allows the computation of quantitative timing information, such as minimum and maximum time delays between given events, differently from its predecessor.

4.2 Binary Decision Diagrams

Binary Decision Diagrams, also referred to as BDDs, are a data structure with a set of algorithms devoted to the representation of Boolean functions [Bryant, 1986]. BDDs represent Boolean functions by means of directed acyclic graphs with restrictions on the ordering of the arguments of the function being represented. Each argument of the function corresponds to a BDD variable, which may have one or more associated nodes on the graph. Except for terminal nodes which are labeled either by 0 or 1, each node on the graph has two outgoing edges, one representing the case where the corresponding argument is false and the other representing the case where the argument is true.

BDDs have proved to be a very efficient representation of Boolean functions, and have been used for representing systems with up to 10^{20} states [Partridge, 1996]. Because many problems in digital logic design and testing, artificial intelligence and combinatorics can be expressed in terms of sequences of operations over Boolean functions [Bryant, 1986], BDDs arise as an alternative for reducing costs or even making Boolean representations practical in those fields. In a special way, one of the most powerful BDD applications has been symbolic model checking, used for verifying digital circuits and other finite state systems [McMillan, 1993].

However, the size of the graph representing a Boolean function is highly sensitive to the ordering of the variables of the function. The choice of the variable ordering may have significant impact on the memory and time requirements of the algorithms used for handling BDDs. Because the problem of computing the best variable ordering is an NP-Complete problem [Bollig et al., 1996], several heuristic approaches have been developed to find approximate solutions. There are both static and dynamic heuristic approaches. Such heuristics have been successfully employed to find variable orderings for complex systems at the expense of reasonable time costs.

4.2.1 BDD Representation

BDD graphs are given by a root node and a set of terminal and non-terminal vertices V. Every vertex $v \in V$ has an index $index(v)$, used for indicating the level of the corresponding variable on the graph. Indexes range from 1 to $n+1$, where the first level corresponds to the root vertex and the level $n+1$ corresponds to terminal vertices. Each non-terminal vertex

v has two child nodes, $low(v) \in V$ and $high(v) \in V$, where $index(v) < index(low(v))$ and $index(v) < index(high(v))$. The vertices $low(v)$ and $high(v)$ represent the case where v is false and the case where v is true, respectively. On the other hand, each terminal vertex v has the attribute $value(v) \in \{0, 1\}$. Along the graph, all the vertices corresponding to a BDD variable are found in a fixed order, at the same level on the graph. In terms of data structures, each BDD is represented by the following structure:

```
type vertex = record
  low, high: vertex;
  index: 1..n+1;
  value: (0,1,X);
  id: integer;
  mark: boolean;
end;
```

where `X` stands for any value assigned to `value` for non-terminal vertices, `id` represents the unique identifier used for identifying each vertex on the graph and `mark` is a field used for the handling algorithms in order to keep track of the already visited vertices during the graph traversal. Notice that vertex indexes can be changed by the creation of variables and by dynamic variable reordering methods, but the identifiers are always the same.

Given the function f and its arguments x_1,x_2,..., x_n, each instance of such arguments describes a path on the graph, starting at the root vertex and taking either $low(v_i)$, when x_i is false, or $high(v_i)$, when x_i is true, where v_i is the non-terminal vertex on the path corresponding to the argument x_i. The value corresponding to f is the value found in the terminal vertex at the end of the path. This procedure is based on the Shannon expansion [Shannon, 1938], used for decomposing a function into simpler ones. Consider the function f and the substitution of its argument x_i by the constant $c \in \{0, 1\}$. The Shannon expansion with regard to x_i corresponds to

$$f = \overline{x}_i.f|_{x_i=0} + x_i.f|_{x_i=1}$$

where $f|_{x_i=c}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, c, x_{i+1}, ..., x_n)$. The edges $low(v_i)$ and $high(v_i)$ of the vertex v_i standing for x_i correspond to $\overline{x}_i.f|_{x_i=0}$ and $x_i.f|_{x_i=1}$, respectively. Figure 4.1 represents this expansion.

4.2.2 BDD Basic Operations

BDDs rely on basic handling algorithms and on their combination to perform a rich set of operations on the graphs representing Boolean functions. Below, we reproduce the table found in [Bryant, 1986], where basic algorithms and their time complexities are summarized. In table 4.2, f stands for a Boolean function, G represents the BDD graph, S_f

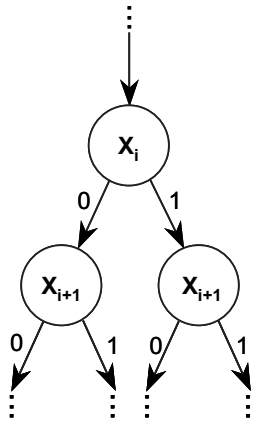


Figure 4.1: Shannon expansion.

stands for the set of BDDs satisfying f and n is the number of arguments of the function S_f, assuming a reduced graph.

Procedure	**Result**	**Time Complexity**
`reduce`	G reduced to canonical form	$O(\|G\|)$
`apply`	$f_1 <op> f_2$	$O(\|G_1\|.\|G_2\|)$
`restrict`	$f\|_{x_i=b}$	$O(\|G\|)$
`compose`	$f_1\|_{x_i=f_2}$	$O(\|G_1\|^2.\|G_2\|)$
`satisfy-one`	some element of S_f	$O(n)$
`satisfy-all`	S_f	$O(n.\|S_f\|)$
`satisfy-count`	$\|S_f\|$	$O(\|G\|)$

Table 4.2: Basic BDD operations.

The procedure `reduce` is used to convert an input graph into a canonical representation denoting the same function, being the core procedure with regard to the compactness of the representation. It removes redundant subgraphs and assigns unique identifiers to the vertices. Procedures `apply` and `restrict` are the more important ones for the model checking technique, they will be discussed later in this section. Procedure `compose` constructs the graph for the function obtained by composing two functions. Although it can be expressed by the combination of procedures `apply` and `restrict`, it is accomplished by a ternary Boolean operation `ITE` due to efficiency reasons. Procedures `satisfy-one` and `satisfy-all` are used to enumerate some element or all the elements of a satisfying set, respectively. Procedure `satisfy-one` is based on a classic depth-first search with backtracking and `satisfy-all` relies on an exhaustive search of the graph. For efficiency

reasons, both procedures must be applied on reduced graphs. Finally, `satisfy-count` is used for computing the number of elements in a satisfying set. It traverses the graph by recursively visiting every vertex and its subgraphs.

The `apply` Procedure

The procedure `apply` creates the representation of a function according to a Boolean expression with two operands and a generic operator. Given the graphs representing the input functions f_1 and f_2, and the generic operator `<op>`, `apply` produces a reduced graph representing the function f_1`<op>`f_2, such that:

$$[f_1\texttt{<op>}f_2](x_1,\ldots,x_n) = f_1(x_1,\ldots,x_n) \texttt{ <op> } f_2(x_1,\ldots,x_n)$$

The `apply` algorithm recursively proceeds from the roots of the input functions to leaves of the graph, building the output function by using a derivation of the Shannon expansion:

$$[f_1\texttt{<op>}f_2](x_1,\ldots,x_n) = \overline{x}_i.(f_1|_{x_i=0} \texttt{ <op> } f_2|_{x_i=0}) \texttt{ + } x_i.(f_1|_{x_i=1} \texttt{ <op> } f_2|_{x_i=1})$$

During the construction of the output function the algorithm considers different cases by distinguishing terminal from non-terminal vertices and by considering vertex indexes. In order to improve the computation, the implementation also uses a cache of already computed operations and short-circuit evaluation. Both improvements are concerned with the reduction of the number of recursive operations performed on the subgraphs of the function of interest.

The graphs produced by the traditional procedure `apply` are not necessarily reduced. So, the reduction algorithm must be applied before returning the produced graphs. On the other hand, some BDD packages are able to directly build already reduced graphs, without the need of a distinct reduction step. In [Brace et al., 1990], we found a BDD package based on an efficient implementation of the if-then-else operator which has been found to be faster and more memory-efficient than traditional implementations.

The `restrict` Procedure

Given a graph for the input function f, an index i corresponding to a variable on the graph and a value b, the procedure `restrict` produces a reduced graph corresponding to the input function after replacing each vertex at level i on the graph by b, that is, `restrict` receives f and produces $f|_{x_i=b}$. For each vertex v at level i, the algorithm must replace v either by $low(v)$, in the case where b is false, or $high(v)$, in the case where b is true. After that, the algorithm must invoke procedure `reduce`, for reducing the generated graph and for assigning unique identifiers to the vertices. Because the computation required for restricting each vertex is constant, the time complexity of this procedure is dominated by procedure `reduce`. So, `restrict` also presents time complexity $O(|G|)$.

For many Boolean operations, the execution time required by BDDs is polynomial on the number of variables in the expression [Partridge, 1996]. Because equivalent Boolean sub-expressions are represented just once on the BDD graph, they are very efficient as the size of the graph is not so big. Another important characteristic of BDDs is the canonical representation of functions. The canonicity of BDDs allows a unique representation of Boolean function to be used for a variable ordering. Thus, equivalence tests are reduced to the task of verifying if two graphs are the same. Also, the satisfiability of a function is reduced to the problem of comparing the graph of the given function and the graph for the constant false [Bryant, 1986].

4.2.3 BDD Example

In order to exemplify the operation of some BDD operators, consider the creation of the graph for the function $f = (a \wedge b) \vee \neg(c \wedge d)$, assuming the ordering a, b, c, d for the variables, from the root to the leaves of the graph. Initially, we have the following graph:

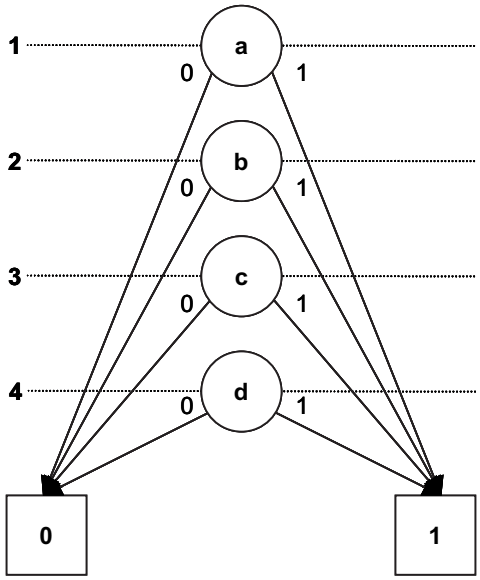


Figure 4.2: BDDs for `a`, `b`, `c`, `d`.

Computing Conjunctions

According to the derivation of the Shannon expansion for the `apply` procedure, the BDD for $f_{(a \wedge b)} = (a \wedge b)$ is given by:

$$\begin{aligned}[f_a \wedge f_b](a,b) &= \overline{a}.(f_a|_{a=0} \wedge f_b) + a.(f_a|_{a=1} \wedge f_b) \\ &= \overline{a}.(F \wedge f_b) + a.(T \wedge f_b)\end{aligned}$$

$$= \overline{a}.(\overline{b}.(F \wedge f_b|_{b=0}) + b.(F \wedge f_b|_{b=1})) + a.(\overline{b}.(T \wedge f_b|_{b=0}) + b.(T \wedge f_b|_{b=1}))$$
$$= \overline{a}.(\overline{b}.(F \wedge F) + b.(F \wedge T)) + a.(\overline{b}.(T \wedge F) + b.(T \wedge T))$$
$$= \overline{a}.(\overline{b}.(F) + b.(F)) + a.(\overline{b}.(F) + b.(T))$$

where T represents the terminal node labeled by 1, F represents the terminal node labeled by 0, f_a and f_b stand for a and b, respectively. The computation of the graph for $f_{(c \wedge d)} = (c \wedge d)$ is similar. In figure 4.3, we show the reduced graphs for $(a \wedge b)$(a) and $(c \wedge d)$(b).

Complementing Functions

The graph for $f_{\neg(c \wedge d)} = \neg(c \wedge d)$ can also be generated according to the `apply` procedure, by using the complement operator. However, complementing a function is the same as complementing the values of the terminal vertices of its graph. Figure 4.4(a) depicts the reduced graph for $f_{\neg(c \wedge d)}$, after complementing the terminal nodes of figure 4.3(b).

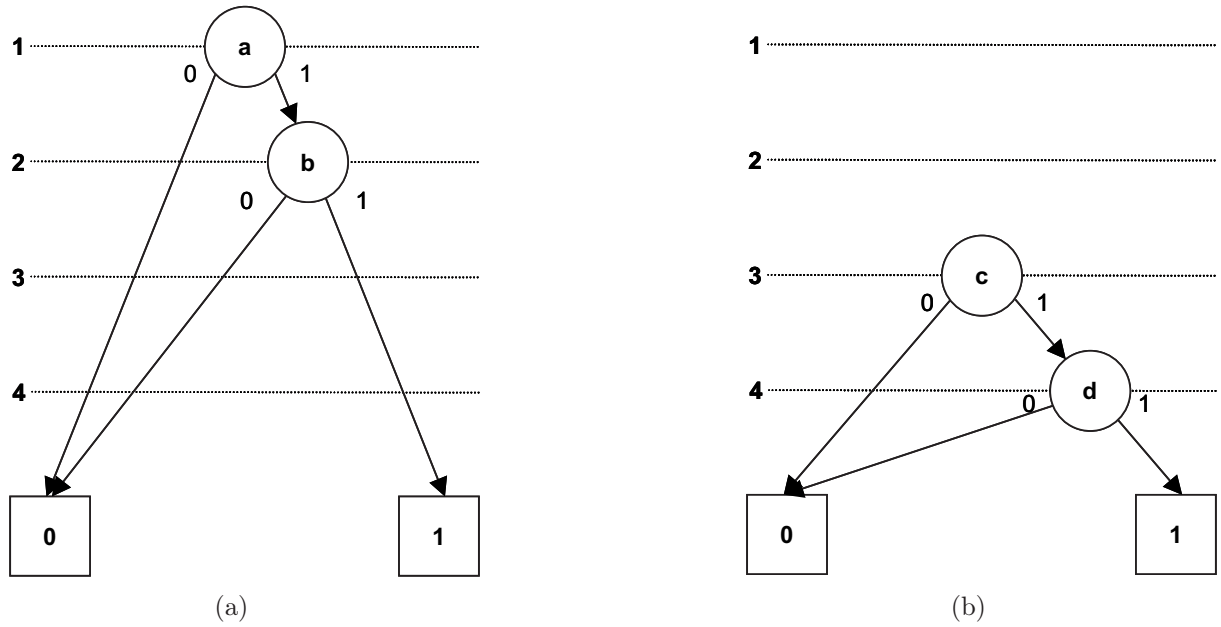


Figure 4.3: BDDs for $(a \wedge b)$(a) and $(c \wedge d)$(b).

Computing Disjunctions

By using the reduced graphs for $f_{(a \wedge b)} = (a \wedge b)$ and $f_{\neg(c \wedge d)} = \neg(c \wedge d)$, the derivation of the Shannon expansion for the `apply` procedure gives the graph for $f = (a \wedge b) \vee \neg(c \wedge d)$:

$$[f_{a \wedge b} \vee f_{\neg(c \wedge d)}](a, b, c, d) = \overline{a}.(f_{a \wedge b}|_{a=0} \vee f_{\neg(c \wedge d)}) + a.(f_{a \wedge b}|_{a=1} \vee f_{\neg(c \wedge d)})$$
$$= \overline{a}.(F \vee f_{\neg(c \wedge d)}) + a.(f_b \vee f_{\neg(c \wedge d)})$$

$$
\begin{aligned}
&= \overline{a}.(\overline{c}.(F \vee f_{\neg(c\wedge d)}|_{c=0}) + c.(F \vee f_{\neg(c\wedge d)}|_{c=1}))+ \\
&\quad a.(\overline{b}.(f_b|_{b=0} \vee f_{\neg(c\wedge d)}) + b.(f_b|_{b=1} \vee f_{\neg(c\wedge d)})) \\
&= \overline{a}.(\overline{c}.(F \vee T) + c.(F \vee f_d))+ \\
&\quad a.(\overline{b}.(F \vee f_{\neg(c\wedge d)}) + b.(T \vee f_{\neg(c\wedge d)})) \\
&= \overline{a}.(\overline{c}.(T) + c.(\overline{d}.(F \vee f_d|_{d=0}) + d.(F \vee f_d|_{d=1})))+ \\
&\quad a.(\overline{b}.(\overline{c}.(F \vee f_{\neg(c\wedge d)}|_{c=0}) + c.(F \vee f_{\neg(c\wedge d)}|_{c=1})) + b.(T)) \\
&= \overline{a}.(\overline{c}.(T) + c.(\overline{d}.(F \vee T) + d.(F \vee F)))+ \\
&\quad a.(\overline{b}.(\overline{c}.(F \vee T) + c.(F \vee f_d)) + b.(T)) \\
&= \overline{a}.(\overline{c}.(T) + c.(\overline{d}.(T) + d.(F)))+ \\
&\quad a.(\overline{b}.(\overline{c}.(T)+c.(\overline{d}.(F\vee f_d|_{d=0})+d.(F\vee f_d|_{d=1})))+b.(T)) \\
&= \overline{a}.(\overline{c}.(T) + c.(\overline{d}.(T) + d.(F)))+ \\
&\quad a.(\overline{b}.(\overline{c}.(T) + c.(\overline{d}.(F \vee T) + d.(F \vee F))) + b.(T)) \\
&= \overline{a}.(\overline{c}.(T) + c.(\overline{d}.(T) + d.(F)))+ \\
&\quad a.(\overline{b}.(\overline{c}.(T) + c.(\overline{d}.(T) + d.(F))) + b.(T))
\end{aligned}
$$

Once again, T represents the terminal node labeled by 1, F represents the terminal node labeled by 0, f_a, f_b and f_d stand for a, b and d, respectively. Figure 4.4(b) presents the final and reduced graph for our example function $f = (a \wedge b) \vee \neg(c \wedge d)$. Note that all the graphs depicted in this section have already been reduced after their generation.

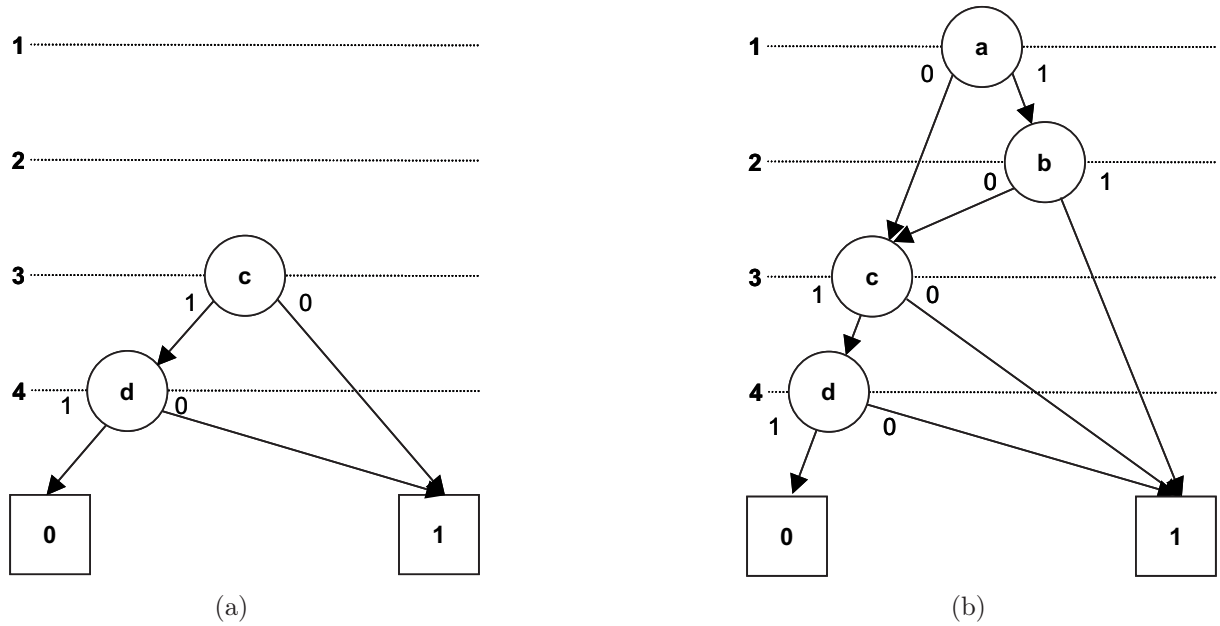


Figure 4.4: BDDs for $\neg(c \wedge d)$(a) and $(a \wedge b) \vee \neg(c \wedge d)$(b).

4.3 Symbolic Computation

Instead of operating over individual states, symbolic model checking handles a set of states at a time. Usually, Boolean functions are used to represent both the set of states and the specification to be checked. For operating over Boolean functions, symbolic algorithms rely on fixpoint characterizations for the regular temporal logic operators, such that each temporal logic operator corresponds to a function, and the set of states resulting from the application of the operator corresponds to the fixpoint of the function.

4.3.1 Fixpoint Characterization of CTL Operators

Let $M = (S, S_0, R, L)$ be a Kripke structure for a given system and $\mathcal{P}(S)$ the set of all subsets of S. $\mathcal{P}(S)$ forms a lattice under the set inclusion ordering, where the empty set, also referred to as $false$, and S, also referred to as $true$, are the least and the greatest subsets, respectively. Each subset corresponds to a predicate on S, in the sense that the predicate evaluates to $true$ in all states in the subset. Given a Boolean function $f : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, we have that $S' \in \mathcal{P}(S)$ is a fixpoint of f if $f(S') = S'$. Note that f is a predicate transformer that maps elements from $\mathcal{P}(S)$ to $\mathcal{P}(S)$. Provided that f is a monotonic predicate transformer, there are a least fixpoint, $\mu Z.f(Z)$, and a greatest fixpoint, $vZ.f(Z)$ [Tarski, 1955]. Because S is finite and f is monotonic, $\mu Z.f(Z) = \cap\{Z \mid f(Z) \subseteq Z\}$ and $vZ.f(Z) = \cup\{Z \mid f(Z) \supseteq Z\}$. The algorithms for computing the least and greatest fixpoints are shown below [Clarke et al., 1999], where f is a predicate transformer, Q and Q' are predicates:

```
function leastfp(f)
 begin
   Q := false;
   Q' := f(Q);
   while (Q ≠ Q') do
    begin
      Q := Q';
      Q' := f(Q');
    end
   return Q;
 end
```

```
function greatestfp(f)
 begin
   Q := true;
   Q' := f(Q);
   while (Q ≠ Q') do
    begin
      Q := Q';
      Q' := f(Q');
    end
   return Q;
 end
```

According to [Emerson and Clarke, 1980], we have the following fixpoint characterizations for the functions representing the CTL temporal logic operators:

- AF $f = \mu Z.f \vee$ AXZ
- EF $f = \mu Z.f \vee$ EXZ

- AG $f = \nu Z.f \wedge$ AXZ
- EG $f = \nu Z.f \wedge$ EXZ
- A[f_1 U f_2] = $\mu Z.f_2 \vee (f_1 \wedge$ AX$Z)$
- E[f_1 U f_2] = $\mu Z.f_2 \vee (f_1 \wedge$ EX$Z)$

4.3.2 Checking Algorithms

Symbolic model checking algorithms take into account that the transition relation is represented by means of BDDs. Each variable in the system is associated with two BDDs, one representing its current state and another representing its next state. So, the transition relation is composed of transitions $R(\overline{v}, \overline{v}')$, where v and v' stand for the BDDs representing the set of variables in the current and in the next state of the transition. We consider that specifications are expressed in CTL logic, by using its regular syntax. Let `check` be the main model checking procedure and f the formula to be checked. Procedure `check` receives f as input and produces the BDD corresponding to states where the expression holds. If ap is an atomic proposition, `check(`ap`)` returns the BDD representing the states where ap holds, based on the BDD operation `apply`. Given the BDDs for subformulas f_1 and f_2, the operation `apply` is also used for computing `check(`$f_1 \wedge f_2$`)` and `check(`$f_1 \vee f_2$`)`. Although `check(`$\neg f$`)` can be computed by means of the `apply` procedure, such an operation can be done by simply exchanging the terminal nodes of the graph for f. Below, we use the routine `bdd_not` for computing $\neg f$, without concerns about the internal procedure used.

- `check(`ap`) = apply[`$ap \wedge True$`]`
- `check(`$f_1 \wedge f_2$`) = apply[`$f_1 \wedge f_2$`]`
- `check(`$f_1 \vee f_2$`) = apply[`$f_1 \vee f_2$`]`
- `check(`$\neg f$`) = bdd_not(`f`)`

formulas of the form EXf, E[f_1Uf_2] and EGf are dealt by intermediate procedures:

- `check(EX`f`) = checkEX(check(`f`))`
- `check(E[`f_1`U`f_2`]) = checkEU(check(`f_1`), check(`f_2`))`
- `check(EG`f`) = checkEG(check(`f`))`

The procedure `checkEX(`f`)` returns the set of states with successors where f holds. Let $f(\overline{v})$ be the set of variables where f holds, represented as a BDD:

$$\texttt{checkEX(}f(\overline{v})\texttt{)} = \exists \overline{v'}\texttt{[}f(\overline{v'}) \wedge R(\overline{v}, \overline{v}')\texttt{]}$$

where $R(\overline{v}, \overline{v}')$ is the BDD representing the transition relation. The existential quantification is used to quantify out $\overline{v'}$, in order to find the states where `EX`f holds by removing the next state BDDs. The quantification operators are usually implemented by the BDD library as combinations of `restrict` and `apply` operators. However, instead of performing separate operations for computing the logical conjunction between two Boolean functions and for computing the existential quantification, those two operations are usually performed together by means of a relational product routine. Usually, the relational product is much more efficient than doing those operations separately.

The procedure `checkEU(`f_1`,`f_2`)` first determines all states where f_2 holds and then go backwards through the converse of the transition relation looking for states that can be reached by a path in which f_1 holds in each state. The procedure is based on the least fixpoint computation of the function corresponding to operator `EU`:

$$\texttt{E[}f_1\ \texttt{U}\ f_2\texttt{]} = \mu Z.f_2 \vee (f_1 \wedge \texttt{EX}Z)$$

The function `leastfp` is used to compute a sequence of approximations until the set of states where `E[`f_1`U`f_2`]` holds has converged.

The procedure `checkEG(`f`)` is based on the greatest fixpoint characterization for the operator `EG` to find states where `EG` f holds:

$$\texttt{EG}\ f = \nu Z.f \wedge \texttt{EX}Z$$

The greatest fixpoint computation is accomplished by the function `greatestfp`.

4.4 Drawbacks of the Symbolic Approach

Unfortunately, sometimes it is difficult to express satisfactorily data information by means of symbolic expressions [Sebastiani, 2001]. Although symbolic representations have shown to be practical for many industrial hardware circuits, they are not suitable for certain arithmetic operations and generally do not work well for software [Chan, 1999]. According to [Cheung and Kramer, 1999] the symbolic approach usually does not achieve sensitive reductions in software specifications, whose logical structures are less regular than those of hardware designs. Generally, integers and arithmetic operations are not efficiently represented by means of simple Boolean expressions, due to the size of the produced representations [Chan et al., 1997]. Additionally, systems with continuous variables ranging over non-countable domains, like real-time systems, are not suitable for symbolic model checking [Chan, 1999]. Usually, symbolic methods are less efficient than explicit methods when the state space presents a less regular structure.

Chapter 5

The Verimag Interchange Format

The Verimag Interchange Format [Bozga et al., 1999], also called IF from now on in this work, is a language based on a dynamic version of extended automata. The language has been originally proposed for modeling asynchronous communicating real-time systems and has suffered several extensions, including the ability to model and to analyze systems endowed with dynamic process creation and dynamic data types. Currently, several validation tools focused on real-time systems use the Interchange Format. The set of such tools is known as the IF validation environment.

5.1 Extended Automata

In the Interchange Format, systems are modeled as several extended automata that execute in parallel and interact through point-to-point message-passing via communication buffers and shared variables. Each extended automaton is described by means of a process. Because processes can be dynamically created and destroyed, the number of active automata may change during the execution of a system. Processes are allowed to have local memory, that is, sets of local variables. There is a rich set of typed variables, from predefined types to user-defined ones. Predefined types include Boolean, integer, float, pid and clock. The user is supplied with classical type definition mechanisms, as enumerations, ranges and arrays in order to create additional types. Clocks are used to measure time during the execution of processes. Control states are the main structuring component of processes, as discussed later. Each process is unequivocally identified by means of a pid value and has a queue of pending messages.

IF models are composed from process instances, running in parallel and interacting asynchronously through shared variables and message-passing. Message-passing is accomplished by means of signals, instances of signal routes and fifo communication buffers.

The execution of extended automata is given by the graph of their interleaved executions. Automata execute asynchronously, that is, one automaton is executing while the

other ones are waiting. Automata can interact with each other and with the environment, through signal exchange.

5.2 The Description Language

In this section, we show the main components of the IF language. For each component, we give its BNF description and present the more relevant semantic aspects. Along our discussion, the alternate bit protocol is employed to illustrate the usage of constructions.

5.2.1 Systems

Systems are composed of dynamic components, such as processes, signal routes and signals, and static components, such as variables, types, constants and external procedures. Syntactically, systems are described according to the following BNF rules:

```
system-decl ::=
  system system-id ;
    {system-component}*
  endsystem ;
```

```
system-component ::=
  process-decl |
  signalroute-decl |
  signal-decl |
  procedure-decl |
  var-decl |
  type-decl |
  const-decl
```

Semantically, a system is given as a tuple (D, S, P), where D is the set of data types including Boolean, integer, clock, pid and user-defined ones, S is the set of typed signals and P is the set of process types. Below, we show some of those elements composing a specification for the alternate bit protocol in the Interchange Format:

```
01  system bitalt;
02  type data = range 0 .. 1;
03  signal sdt(data, boolean);
04  signalroute tr(1) #unicast #lossy
05    from transmitter to receiver with sdt;
06  process transmitter(1);
07    ...
08  endprocess;
09  process receiver(1);
```

```
10    ...
11  endprocess;
12  endsystem;
```

The schematic description above presents the user-defined type `data`, line 02, the signal `sdt,` line 03, the signal route `tr`, lines $04-05$, and processes `transmitter` and `receiver`, lines $06-08$ and $09-11$, respectively. The numbers within parentheses represent the number of initial instances for the signal route and processes. Also, note the signal route options, `#unicast` and `#lossy`, that establish the delivering policy and the reliability of the communication channel, respectively.

5.2.2 Processes

Processes are defined as extended finite-state machines. Each process instance has a unique identifier number and an input fifo buffer storing all the incoming messages. Process specifications include the name of the process, the number of initial instances, the list of formal parameters, the set of local variables, which correspond to the local memory, and the set of control states. Data types, constants and procedure definitions may also be included. Syntactically, processes are given by the following BNF rules:

```
process-decl ::=
  process process-id(const) ;
    [fpar fpar-decl{,fpar-decl}*;]
    {process-component}*
  endprocess ;

process-component ::=
    state |
    procedure-decl |
    var-decl |
    type-decl |
    const-decl
```

Semantically, each process is viewed as a tuple $(Q, X, T, q_0) \in P$, where Q is the set of control states, X is the set of typed variables, T is the set of transitions and $q_0 \in Q$ is the initial state. Note that X includes both local and global variables. The expansion of process `transmitter` shown in a previous specification reveals some details:

```
01  process transmitter(1);
02  var t clock;
03  var b boolean;
04  var c boolean;
05  var m data;
06  state start #start ;
```

```
07    ...
08  endstate;
09  state idle;
10    ...
11  endstate;
12  state busy;
13    ...
14  endstate;
15  state q8 #unstable ;
16    ...
17  endstate;
18  endprocess;
```

Note the control states inside the process `transmitter`, namely `start`, lines $06-08$, `idle`, lines $09-11$, `busy`, lines $12-14$, and `q8`, lines $15-17$. Also, lines 06 and 15 show state options, `#start` and `#unstable`, that define `start` as the initial state of the process and `q8` as an unstable state.

5.2.3 Control States

Control states define the behavior of a process by means of actions, transitions to other states and, possibly, substates. There is no theoretical limit for the number of nested substates. State specifications include temporal constraints and the set of deferred signals.

state ::=
  state *state-id*{*state-option*}*;
    [tpc *constraint*]
    [save *signal-id* {,*signal-id*}*]
    {*state-component*}*
  endstate ;

state-component ::=
  transition |
  state

The initial state and the stability of states are set during the state definition, by options. The stability controls process execution steps. Transitions between stable states are atomic, uninterrupted. Formally, we have the following options:

state-option ::=
  #start |
  #stable |
  #unstable

Below, we detail state q8, from process transmitter. Lines $02-05$ and $06-07$ define transitions to other states of transmitter. The conditions shown in lines 02 and 06 are transition guards and must be true for the execution of the nested statements.

```
01  state q8 #unstable ;
02    provided c = b;
03      task b := not b;
04      reset t;
05        nextstate idle;
06    provided c <> b;
07        nextstate busy;
08  endstate;
```

5.2.4 Transitions

Transitions between states are controlled by condition guards. So, transitions can be viewed as process reactions in response to stimuli. Transition can be triggered by the activation of some untimed guard, activation of some timed guard or the presence of some signal in the input buffer of the process instance. When a condition guard is satisfied, some actions may be required before passing the control to another state. Actions include variable assignments, clock setting, clock and variable resetting, signal sending, procedure calls, creation and destruction of processes and signal routes. Transitions can also be used with a stop action, which means the destruction of the process instance. Untimed guards are implemented through provided clauses, composed of constants, variables and Boolean, arithmetic and relational operators. Timed guards are implemented through when clauses, which control time constraints on clock variables. Finally, the presence of signals in the input buffer is implemented through input clauses.

```
transition ::=
  [deadline {eager | delayable | lazy};]
  [provided expression ;]
  [when constraint ;]
  [input signal-id([expression {,expression}*]);]
    {statement}*
    terminator
```

```
statement ::=
  action |
  if expression then {statement}* [else {statement}*] endif |
  while expression do {statement}* endwhile

terminator ::=
  nextstate state-id; |
  stop;
```

Some guards are presented in the states shown below. The `input` and `when` clauses, lines 02 and 04 in state `busy`, establish the conditions for the execution of transitions from state `busy` to states `q8` and `busy`, respectively. The `provided` clauses, lines 02 and 06 in state `q8`, establishes the condition for the execution of the transition from state `q8` to state `idle` and from state `q8` to state `busy`, respectively. Note that there can be some statements between the guard condition and the transition itself. Because more than one transition can be enabled at some control state, and all the situations have to be considered at execution, process may have non-deterministic behavior.

```
01  state busy;
02    input ack(c);
03      nextstate q8;
04    when t = 1;
05      output sdt(m, b) via {tr}0;
06      set t := 0;
07      nextstate busy;
08  endstate;
```

```
01  state q8 #unstable ;
02    provided c = b;
03      task b := not b;
04      reset t;
05      nextstate idle;
06    provided c <> b;
07      nextstate busy;
08  endstate;
```

Semantically, transitions are given by $q \xrightarrow[u]{g\ a} q' \in T$ of $(Q, X, T, q_0) \in P$, where $q \in Q$ is the source state, $g \in 2^X$ is a Boolean guard defined over system variables, a is an action, $q' \in Q$ is the target state and $u \in \{eager, delayable, lazy\}$ is an urgency type.

5.2.5 Signals

Signals are used to communicate between processes. Processes communicate asynchronously through point-to-point signal exchange, directly or via signalroutes. For sending signals one process must know the identity of the receiver process. Because the language currently deals with asynchronous communication, the sender process is never blocked by the avail-

ability of the receiver process. Messages are immediately delivered to the receiver process, being stored in its input message queue. All signals share one queue per process instance and, therefore, `input` clauses get blocked when the expected message is not in the buffer or it is not the message in the head of the queue. Signal specification include the signal identifier and the parameter types carried by the signal:

```
signal-decl ::=
  signal signal-id([type-id {, type-id}* ]);
```

Signalroutes can be dynamically created and destroyed in order to configure communication paths among processes. Signalroute specifications include the identifier of the signalroute, the initial number of instances, source and destination endpoints and the signals allowed to transit on the signalroute. Each signalroute instance possesses a unique identifier number.

```
signalroute-decl ::=
  signalroute signalroute-id (const)
{signalroute-option}*
from {process-id | env} to {process-id | env}
with signal-id {, signal-id}*;
```

Additionally, the behavior of the signalroute may be refined by using options to determine the queuing policy, the reliability, the delivering policy and the delaying policy:

```
signalroute-option ::=
  #fifo | #multiset |
  #reliable | #lossy |
  #peer | #multicast | #unicast |
  #urgent | #delay[l,u] | #rate[l,u]
```

Below, we show more details on the set of signal and signalroutes used by the transmitter process, the receiver process and the environment for the alternate bit protocol:

```
01  signal get(data);
02  signal put(data);
03  signal ack(boolean);
04  signal sdt(data, boolean);
05  signalroute et(1)
06    from env to transmitter
07    with put;
08  signalroute tr(1) #unicast #lossy
09    from transmitter to receiver
10    with sdt;
11  signalroute rt(1) #unicast #lossy
12    from receiver to transmitter
13    with ack;
14  signalroute re(1)
15    from receiver to env
16    with get;
```

Signal declarations embrace lines from 01 to 04. Signalroutes are declared from lines 05 to 16. The keyword `env`, line 06, denotes the environment.

5.2.6 Actions

Actions are used inside transitions in order to change variable values. The language considers internal actions, informal actions, variable assignments, clock setting, variable and clock resetting, signal sending, procedure calls, creation and destruction of process and signalroute instances, as shown below.

```
action ::=
 skip ; |
 informal string ; |
 task expression := expression ; |
 set expression := expression ; |
 reset expression ; |
 output signal-id([expression {,expression}*])
   [via signalroute-id] [to expression] ; |
 [expression :=] call procedure-id([expression {,expression}*]); |
```

```
[expression :=] fork process-id([expression {,expression}*]); |
[expression :=] fork signalroute-id([expression {,expression}*]); |
kill expression;
```

5.2.7 Expressions

Expressions are made of constant values, variables and operators. The language offers several operators, including boolean operators, arithmetic operators, comparison operators, cast operators and conditional operators. Struct field selectors and array indexes are also available.

```
expression ::=
  const |
  variable-id | fpar-id |
  expression.field-id | expression[expression] |
  function-id([expression {, expression}*]) |
  un-op expression |
  expression bin-op expression |
  cast-op expression |
  expression ? expression : expression |
  (expression)

un-op ::=
  not | active |
  + | -

bin-op ::=
  or | and |
  = | <> | < | <= | >= | > |
  + | - | * | / | %

cast-op ::=
  {type-id | process-id | signalroute-id}
```

Constraints are particular expressions composed of conditions on clocks. They are conjunctions of atomic constraints, made of comparisons between clocks or clock differences and integer expressions.

```
constraint ::=
  atomic-constraint {and atomic-constraint}*

atomic-constraint ::=
  expression comp-op expression |
  expression - expression comp-op expression

comp-op ::=
  = | < | <= | >= | >
```

5.2.8 Variables

Global variables, declared outside processes and accessible in the whole system, and local variables, declared and accessible inside a specific process, have name and type. Naturally, formal parameters are similar to local variables.

```
var-decl ::=
  var var-id type-id {private | public};

fpar-decl ::= [in | inout | out] fpar-id type-id
```

Basic types include boolean, integer, float, pid and clock. Boolean, integer and real domains have their usual interpretation. Process identifier values belongs to countable, infinite set of values. Clocks values are real numbers. Besides, a rich set of type constructors is available for creating user-defined types, such as enumerations, ranges, records and arrays. In addition, it is possible to define abstract types for supporting complex types by using only the signature of the abstract type being defined. External implementations must be supplied before using the abstract type.

```
type-decl ::=
  type type-id = type;
```

```
type ::=
  enum const-id {, const-id}* endenum |
  range const .. const |
  array [const .. const] of type-id |
  record { field-decl} * endrecord |
  string [const] of type-id |
  abstract {function-decl}* endabstract |
  type-id

field-decl ::=
  field-id type-id;

function-decl ::=
  type-id function-id ([type-id {, type-id}*]);
```

The legibility of descriptions can be improved by naming important values with constants. Constants are used in expressions, type definitions and system configurations. Specifically, the constant `self` denotes the pid of a process or signalroute instance, while the constant `nil` denotes a null pid.

```
const-decl ::=
  const const-id [= const];

const ::=
  true | false |
  integer | float |
  self | nil |
  const-id
```

5.2.9 External Procedures

External procedures describe intensive data transformations, executed without interaction in some instance. They are the mechanism to import and to use code of an external language into specifications. Procedure declarations include the procedure identifier, formal parameters, the return type and the code imported from the external language.

```
procedure-decl ::=
  procedure procedure-id ;
    [fpar fpar-decl {, fpar-decl}* ; ]
    [returns type-id ; ]
    [{# external code #}]
  endprocedure ;
```

Chapter 6

The Explicit-Symbolic Modeling

This chapter has two main purposes. First, we show the theoretical aspects of the modeling of the combined explicit-symbolic model from an original model. Such formalization is totally independent of a description language. Next, we apply the general formalization for generating explicit-symbolic models from descriptions in the Verimag Interchange Format.

6.1 General Formalization

The general formalization is done in two distinct steps. Initially, both explicit and symbolic models are generated as projections induced by the partitioning of variables of the original model. After that, the explicit and symbolic models are combined to compose the explicit-symbolic model. Although the combined model can be generated directly from a labeled state-transition model, we decided to present two distinct steps due to didactic reasons.

6.1.1 Decomposition of the Original Model

Let $M = (S, R, L)$ be the model for a given system, where S represents the set of states, $R \subseteq S \times S$ represents the transition relation between states, $L : S \rightarrow 2^{AP}$ represents the labeling function which assigns a set of atomic propositions to each state. Assume AP as the set of atomic propositions of M. Suppose that the set AP is decomposed into subsets of explicit and symbolic propositions AP_e and AP_s, respectively, where $AP = AP_e \cup AP_s$ and $AP_e \cap AP_s = \emptyset$. Using this proposition partitioning, two equivalence relations are generated, $\approx_e$ and $\approx_s$, such that

$$\forall\, s,\, s' \in S,\, s \approx_e s' \longleftrightarrow L(s)|_{AP_e} = L(s')|_{AP_e}$$
$$\forall\, s,\, s' \in S,\, s \approx_s s' \longleftrightarrow L(s)|_{AP_s} = L(s')|_{AP_s}$$

Each induced equivalence class $[s]_e$ is an explicit state and each equivalence class $[s]_s$ is a symbolic state. In other words, each state $s \in S$ corresponds to an explicit state $[s]_e$

and a symbolic state $[s]_s$. Consequently, the set of states S is projected into explicit and symbolic sets of states, S_e and S_s, as defined below:

$$S_e = \{[s]_e \,|\, s \in S\} \qquad S_s = \{[s]_s \,|\, s \in S\}$$

The $\approx_e$ and $\approx_s$ relations define two labeling functions L_e and L_s for explicit and symbolic states, respectively. Labels for an explicit state $[s]_e$ and a symbolic state $[s]_s$ are shown below:

$$L_e([s]_e) = L(s)|_{AP_e} \qquad L_s([s]_s) = L(s)|_{AP_s}$$

Therefore, $L(s) = L_e([s]_e) \cup L_s([s]_s)$ because $L(s) = L(s)|_{AP_e} \cup L(s)|_{AP_s}$.

The projection of the original system requires the original transition relation R to be split into a transition relation for the explicit and another for the symbolic part of the model, R_e and R_s, respectively. In order to maintain the correspondence between explicit and symbolic transitions, the original transitions must be labeled. One explicit transition and one symbolic transition will be assigned the same label $id \in \mathbb{N}$ only if they come from the same original transition (s, id, s'), where s, $s' \in S$. So, labels associate explicit and symbolic transitions, indicating which occur together on the models. Given one original transition relation R, the following two transition relations are obtained:

$$R_e = \{([s]_e, id, [s']_e) \,|\, (s, id, s') \in R\} \qquad R_s = \{([s]_s, id, [s']_s) \,|\, (s, id, s') \in R\}$$

that is, if $(s, id, s') \in R$ then $([s]_e, id, [s']_e) \in R_e$ and $([s]_s, id, [s']_s) \in R_s$.

The original model can be restored using the explicit and symbolic models together. Each two transitions $([s]_e, id_e, [s']_e) \in R_e$ and $([s]_s, id_s, [s']_s) \in R_s$, where $id_e = id_s$, define one original transition $(s, id, s') \in R$ such that $id = id_e = id_s$, $L(s) = L_e([s]_e) \cup L_s([s]_s)$ and $L(s') = L_e([s']_e) \cup L_s([s']_s)$.

6.1.2 Composition of the Explicit-Symbolic Model

This section presents the theoretical basis of the combined explicit-symbolic model, as it defines the behavior of the combined model from the cooperation and synchronization between the underlying models. Given $M_e = (S_e, R_e, L_e)$ and $M_s = (S_s, R_s, L_s)$, we obtain the explicit-symbolic model $M_h = (S_h, R_h, L_h)$ by the composition of such explicit and symbolic models. Each explicit-symbolic state $s_h \in S_h$ is given by a pair (s_e, s_s), where $s_e \in S_e$, $s_s \in S_s$ and there exist explicit and symbolic transitions $(s_e, id_e, s'_e) \in R_e$ and $(s_s, id_s, s'_s) \in R_s$ such that $id_e = id_s$. For each $s_h \in S_h$, we have $L_h(s_h) = L_h(s_h)|_{AP_e} \cup L_h(s_h)|_{AP_s}$, where $L_h(s_h)|_{AP_e} = L_e(s_e)$ and $L_h(s_h)|_{AP_s} = L_s(s_s)$. If $(s_e, id, s'_e) \in R_e$ and $(s_s, id, s'_s) \in R_s$ then $(s_h, id, s'_h) \in R_h$, where s_h, $s'_h \in S_h$. Due to the definition of R_h, models M_e and M_s must be explored in an interleaved and synchronized fashion, based on transition labels. States of S_h are visited during the exploration of states in S_e and S_s.

Taking an explicit-symbolic model, we can restore the original explicit and symbolic models in a straightforward way. Given $M_h = (S_h, R_h, L_h)$, we generate $M_e = (S_e, R_e, L_e)$

such that $S_e = \{s_e | (s_e, s_s) \in S_h\}$, $R_e = \{(s_e, id, s'_e) | ((s_e, s_s), id, (s'_e, s'_s)) \in R_h\}$ and $L_e(s_e) = L_h((s_e, s_s))|_{AP_e}$. The model M_s can be analogously obtained.

The construction of the combined model is based on the fact that pairs of explicit and symbolic transitions are considered to occur together if and only if they are labeled with the same identifier. Labels guide both the construction of the combined model and the model checking, being used for coordinating the underlying models.

6.2 Formalization for the Interchange Format

First, we show how Interchange Format descriptions are employed for generating both explicit and symbolic models. Next, we combine explicit and symbolic models in order to emulate the behavior of the original model by means of the explicit-symbolic model.

6.2.1 The Explicit and Symbolic Models

Let $M = (D, S, P)$ be a system specified in the Intermediate Format. Two independent models, $M_e = (D, S, P_e)$ and $M_s = (D, S, P_s)$, must be generated according to the partitioning of M variables into explicit and symbolic classes, X_e and X_s, respectively. Considering each process $(Q, X, T, q_0) \in P$, the variable partitioning induces the explicit process $(Q_e, X_e, T_e, q_{0e}) \in P_e$ and the symbolic process $(Q_s, X_s, T_s, q_{0s}) \in P_s$. Given each process $(Q, X, T, q_0) \in P$, its control states and respective guards must be analyzed. Guards, actions and transitions performed over X_e variables are projected in the model M_e, while those over X_s variables are projected in the model M_s. Let $q \xrightarrow{g\ a} q' \in T$ be a transition of $(Q, X, T, q_0) \in P$, where $q \in Q$ is the source state, $g \in 2^X$ is a Boolean guard, a is an action, $q' \in Q$ is the target state. Such transition affects variables both in X_e and X_s, where those variables represent the program counter, condition guards and actions. Because both models are affected, the original transition induces one explicit transition and one symbolic transition:

$$q_e \xrightarrow{g_e\ a_e} q'_e \in T_e \quad \text{and} \quad q_s \xrightarrow{g_s\ a_s} q'_s \in T_s$$

such that $g_e = g|_{X_e}$, $a_e = a|_{X_e}$, q_e and $q'_e \in Q_e$, $g_s = g|_{X_s}$, $a_s = a|_{X_s}$, q_s e $q'_s \in Q_s$, where $Q_e = Q|_{X_e}$, $Q_s = Q|_{X_s}$. The initial states in M_e and M_s are those projected by the original initial state q_0, that is $q_{0e} = q_0|_{X_e}$ and $q_{0s} = q_0|_{X_s}$. Remember that such explicit and symbolic transitions must be associated with each other.

6.2.1.1 The Explicit Transition Relation

The explicit transition relation should be represented as an array of linked lists where one array entry corresponds to one state and the linked list defines to which states the current state can transition to:

$$T_e(Q_e, Q'_e) = (q_{e_1}, L_1),\ (q_{e_2}, L_2), ...$$

where $L_i = q'_{e_{i1}}, q'_{e_{i2}}, q'_{e_{i3}}, ...$ The first component of each pair of the transition relation stands for an explicit state where transitions come from and the second component of the pair represents the linked list of states where those transitions go to. For example, (q_{e_1}, L_1) defines transitions between states $(q_{e_1} \rightarrow q'_{e_{11}})$, $(q_{e_1} \rightarrow q'_{e_{12}})$ and so on.

6.2.1.2 The Symbolic Transition Relation

The symbolic transition relation should be represented as a formula between propositions in the current and in the next state, $T_s(Q_s, Q'_s)$, to be implemented using BDDs. Note that Q_s and Q'_s represent the current and the next state variables from which the Boolean formula that represents the transition relation is built from, respectively. Each symbolic transition is defined by a conjunction of formulas over Q_s and Q'_s. On the other hand, $T_s(Q_s, Q'_s)$ is defined by the disjunction of such individual symbolic transitions.

6.2.2 The Explicit-Symbolic Model

The partitioning of system variables induces the generation of distinct explicit and symbolic models. Each generated model covers just a subset of the variables and investigates only part of the search space. Such models have to be executed in an interleaved fashion in order to allow us to explore the whole search space and emulate the behavior of the original system. Due to efficiency reasons, we propose a new way of associating transitions. Instead of using the transition labels defined in the modeling, the explicit and symbolic models are interleaved by associating symbolic transitions with explicit transitions. Given systems $M_e = (D, S, P_e)$ and $M_s = (D, S, P_s)$, where D is the set of data types, S is the set of typed signals, P_e and P_s are respectively sets of explicit and symbolic process types, the explicit-symbolic model $M_h = (D, S, P_h)$ is obtained as shown below. For each process $(Q_h, X_h, T_h, q_{0h}) \in P_h$, where Q_h is the set of explicit-symbolic control states, X_h is the set of explicit-symbolic variables, T_h is the set of explicit-symbolic transitions and $q_{0h} \in Q_h$ is the initial state of the explicit-symbolic process, we combine explicit and symbolic representations by introducing a symbolic transition for each explicit transition:

$$T_h(Q_h, Q'_h) = (q_{e_1}, L_1),\ (q_{e_2}, L_2), ...$$

where $L_i = (q'_{e_{i1}}, T_{s_{i1}}), (q'_{e_{i2}}, T_{s_{i2}}), (q'_{e_{i3}}, T_{s_{i3}}), ...$ and $T_{s_{ij}} = T_{s_{ij}}(Q_s, Q'_s)$. Graphically each $(q_{e_i}, L_i) \in T_h(Q_h, Q'_h)$ can be represented as shown in figure 6.1. According to this explicit-symbolic model, first we explore states in the explicit model and then states in the symbolic model. Intuitively $T_{s_{ij}}$ represents the symbolic transition from q_{s_i} to $q'_{s_{ij}}$ associated with the explicit transition from q_{e_i} to $q'_{e_{ij}}$. Such transitions define the explicit-symbolic transition from (q_{e_i}, q_{s_i}) to $(q'_{e_{ij}}, q'_{s_{ij}})$, both in Q_h. So, Q_h corresponds to the set of pairs (q_e, q_s) associated by the explicit-symbolic transition relation T_h, where $q_e \in Q_e$, $q_s \in Q_s$. Consequently, $X_h = X_e \cup X_s$ and $q_{0h} = (q_{0e}, q_{0s})$.

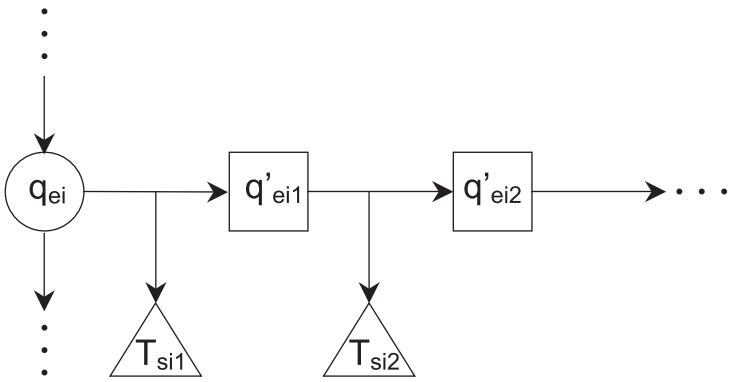


Figure 6.1: Explicit-symbolic transitions.

The explicit and symbolic models can be obtained from the explicit-symbolic model in a straightforward way. Because each $q_h \in Q_h$ corresponds to one pair (q_e, q_s), where $q_e \in Q_e$ and $q_s \in Q_s$, one explicit-symbolic state is associated with exactly one explicit and one symbolic state. Regarding the transition relation $T_h(Q_h, Q'_h) = (q_{e_1}, L_1), (q_{e_2}, L_2), ..., (q_{e_n}, L_n)$, each pair (q_{e_i}, L_i) corresponds to explicit transitions from q_{e_i} to every explicit state $q'_{e_{ij}}$ in the pairs of L_i, where $L_i = (q'_{e_{i1}}, T_{s_{i1}}), (q'_{e_{i2}}, T_{s_{i2}}), (q'_{e_{i3}}, T_{s_{i3}}), ...$ as above. Similarly, the symbolic transition relation can be obtained from $\cup T_{s_{ij}}$, where $(q'_{e_{ij}}, T_{s_{ij}}) \in L_i$.

6.3 Construction of the Explicit-Symbolic Model

This section presents the algorithms used for checking the combined explicit-symbolic model. For each algorithm, we describe the worst-case running-time by means of the O-notation [Cormen et al., 1990]. The algorithms consider specifications to be given in computation tree logic, the temporal logic used on the combined model.

6.3.1 Atomic Propositions

During the parsing of atomic propositions, we must consider two different situations. First, the atomic proposition ap can be explicit, being represented in an explicitly coded state. Second, ap can be symbolic, being represented by a BDD. Independently of the situation, inputs must produce a set of explicit-symbolic states S.

6.3.1.1 Explicit Atomic Propositions

Consider the proposition $ap \in X_e$. First, the algorithm determines the set of explicit states E where ap holds. Next, for each $q_{e_i} \in E$ it determines the set of symbolic states, represented by q_{s_i}, that compose a valid explicit-symbolic state (q_{e_i}, q_{s_i}). The set of symbolic states q_{s_i} is composed of the states where the symbolic transitions of interest come from.

Given a symbolic transition $T_{s_{ij}}$, line 07 existentially quantifies out the next state symbolic variables, $\exists X_s^{'}(T_{s_{ij}})$, in order to compute the symbolic states where $T_{s_{ij}}$ comes from.

```
01 S ← ∅
02 E ← {q_e ∈ Q_e | q_e ⊨ ap}
03 for each q_ei ∈ E do
04   begin
05     q_si ← False
06     for each T_sij where (q_ei, L_i) ∈ T_h and (q_ej, T_sij) ∈ L_i do
07       q_si ← q_si ∨ ∃X'_s(T_sij)
08     add (q_ei, q_si) to S
09   end
```

In the worst case, the entire computation requires time $O(|T_e|.|Q_s|^2)$. Together, the loops dominate the computation. They are used to investigate the symbolic transitions associated with the explicit transitions of interest. Note that each $T_{s_{ij}}$ represents a set of symbolic transitions. Because each explicit transition is exclusively associated with a set of symbolic transitions, line 07 is executed $|T_e|$ times. The computation of the set of current symbolic states where transitions of $T_{s_{ij}}$ come from, $\exists X_s^{'}(T_{s_{ij}})$, and the disjunction require time $O(|Q_s|^2)$.

6.3.1.2 Symbolic Atomic Propositions

Consider the proposition $ap \in X_s$. For each explicit state q_{e_i}, the algorithm investigates all the symbolic transitions associated with the explicit transitions that come from q_{e_i} in order to determine the existence of symbolic transitions that originate from states where ap holds. The set of states q_{s_i} where those symbolic transitions come from composes an explicit-symbolic state together with q_{e_i}, (q_{e_i}, q_{s_i}). Given a symbolic transition of interest $T_{s_{ij}}$, line 07 existentially quantifies out the next state symbolic variables, $\exists X_s^{'}(T_{s_{ij}})$, in order to compute the symbolic states where $T_{s_{ij}}$ comes from.

```
01 S ← ∅
02 for each q_ei ∈ Q_e do
03   begin
04     q_si ← False
05     for each T_sij where (q_ei, L_i) ∈ T_h and (q_ej, T_sij) ∈ L_i do
06       if (T_sij ∧ ap) ≠ False then
07         q_si ← q_si ∨ ∃X'_s(T_sij)
08     if (q_si ≠ False) then
```

```
09        add (q_{e_i}, q_{s_i}) to S
10    end
```

This algorithm requires time $O(|T_e|.|Q_s|^2)$ in the worst case. Loops are used to investigate the symbolic transitions associated with the explicit transitions of interest, as done by the previous algorithm. Because each explicit transition is exclusively associated with a set of symbolic transitions $T_{s_{ij}}$, line 07 is executed $|T_e|$ times. The computation of the set of current symbolic states where transitions of $T_{s_{ij}}$ come from, $\exists X'_s(T_{s_{ij}})$, and the disjunction are $O(|Q_s|^2)$.

Both algorithms for computing explicit-symbolic states from atomic propositions only include an explicit-symbolic state (q_{e_i}, q_{s_i}) into results if q_{e_i} and q_{s_i} hold simultaneously in the explicit and symbolic models, respectively. Specifically, the inner loops ensure that states q_{e_i} and q_{s_i} are associated with each other. Conceptually, all the atomic propositions must be converted into explicit-symbolic states before checking a given specification. However, the implementation of such algorithms should explore each underlying model as much as possible before computing explicit-symbolic states from atomic propositions, due to efficiency reasons. This decision can reduce the cost of combining the explicit and the symbolic models and can improve the performance of the search on the combined model.

6.3.2 Explicit-Symbolic Expressions

The following algorithms assume that initial expressions have been used to produce explicit-symbolic states. Each algorithm takes sets of input explicit-symbolic states I_1 and I_2 corresponding to the evaluation of subexpressions and produces another set of explicit-symbolic states S. Because temporal-logical operators $\neg$, $\vee$, EX, EU and EG can be used to define the remaining CTL operators, we restrict our discussion to such basic operators.

6.3.2.1 Negation

Given an expression φ represented by the set of input states I_1, the algorithm produces the set of output explicit-symbolic states S where $\neg\varphi$ holds. For each explicit state $q_{e_i} \in Q_e$, if (q_{e_i}, q_{s_i}) is in I_1 for some $q_{s_i} \in Q_s$, the algorithm produces the complement for the symbolic state, $\neg q_{s_i}$, and checks if it composes a valid pair with the explicit state q_{e_i}. If so, the pair $(q_{e_i}, \neg q_{s_i})$ is added into S. On the other hand, if (q_{e_i}, q_{s_i}) is not in I_1 for some $q_{s_i} \in Q_s$, the algorithm adds all the valid pairs (q_{e_i}, q_{s_i}) to S. In other words, the algorithm replaces the current set of valid explicit-symbolic states by its complement.

```
01 S ← ∅
02 sort I_1
03 for each q_{e_i} ∈ Q_e do
04   if (q_{e_i}, q_{s_i}) is in I_1 for some q_{s_i} ∈ Q_s then
```

```
05     begin
06       if (T_{s_ij} ∧ ¬q_{s_i}) ≠ false and (q_{e_i}, ¬q_{s_i}) ∉ I_1 then
07         add (q_{e_i}, ¬q_{s_i}) to S where (q_{e_i}, L_i) ∈ T_h, (q_{e_j}, T_{s_ij}) ∈ L_i
08     end
09   else
10     begin
11       q_{s_i} ← False
12       for each T_{s_ij} where (q_{e_i}, L_i) ∈ T_h and (q_{e_j}, T_{s_ij}) ∈ L_i do
13         q_{s_i} ← q_{s_i} ∨ ∃X'_s(T_{s_ij})
14       add (q_{e_i}, q_{s_i}) to S
15     end
```

The set of input states I_1 must be sorted, as shown at line 02. By using an efficient algorithm as *quicksort* [Cormen et al., 1990], the sorting can be performed in time $O(|I_1|.lg|I_1|)$. The sorting of I_1 makes it possible to use an efficient search procedure at lines 04 and 06, as *binary search* [Cormen et al., 1990]. The main loop coordinates the exploration of the symbolic transitions in order to determine valid symbolic states. The computation of valid symbolic states itself is done at lines 06 and 13, in time $O(|Q_s|^2)$. Because each explicit transition is exclusively associated with a set of symbolic transitions $T_{s_{ij}}$, such an operation is computed at most $|T_e|$ times. Also, line 06 requires a search on the set of input explicit-symbolic state $|I_1|$, what requires time $O(lg|I_1|)$. So, the entire computation requires time $O(|T_e|.(|Q_s|^2 + lg|I_1|))$. Note that the sorting of I_1 asymptotically requires less time than the main loop, that is $O(|I_1|.lg|I_1|) \leq O(|T_e|.(|Q_s|^2 + lg|I_1|))$, because $|I_1|$ is bounded by $|Q_e|$ and $|Q_e| \leq |T_e|$.

6.3.2.2 Disjunction

Given that ψ and γ are represented by sets of input explicit-symbolic states I_1 and I_2 respectively, the algorithm produces the set of output explicit-symbolic states S where $\psi \vee \gamma$ holds. First, the explicit-symbolic states in I_1 are stored in S. Next, explicit-symbolic states in I_2 are also copied to S. Explicit-symbolic states with shared explicit states are merged by performing the disjunction of their symbolic states.

```
01 S ← ∅
02 sort I_1
03 for each pair (q_e, q_s) of I_1 do
04   add (q_e, q_s) to S
05 for each pair (q_e, q_{s_i}) of I_2 do
06   if (q_e, q_{s_j}) ∈ I_1 for some q_{s_j} ∈ Q_s then
```

```
07      replace (q_e, q_{s_j}) with (q_e, q_{s_j} ∨ q_{s_i}) in S
08    else
09      add (q_e, q_{s_i}) to S
```

Once again, the set of input states I_1 must be sorted, as shown at line 02. By using an efficient algorithm as *quicksort* [Cormen et al., 1990], the sorting can be performed in time $O(|I_1|.lg|I_1|)$. The sorting of I_1 makes it possible to use an efficient search procedure at line 06, as *binary search* [Cormen et al., 1990], in time $O(lg|I_1|)$. Note that the replacement that takes place at line 07 can be based on the position already computed at line 06, because the loop spanning lines $03 - 04$ maintains the order between the first $|I_1| - th$ states in S, when considering the explicit component as the sorting descriptor. On the other hand, the disjuction of symbolic states shown at line 07 requires time $O(|Q_s|^2)$. So, the loop spanning lines $05 - 09$ requires time $O(|I_2|.(lg|I_1| + |Q_s|^2))$. Consequently, the entire computation requires time $O((|I_1|.lg|I_1|) + (|I_2|.(lg|I_1| + |Q_s|^2)))$.

6.3.2.3 $EX(\varphi)$

Given that φ is represented by the set of input explicit-symbolic states I_1, the algorithm produces the set of output explicit-symbolic states S where $EX(\varphi)$ holds. For each explicit-symbolic state (q_{e_j}, q_{s_j}) in I_1, first the algorithm computes the set of predecessors E for the explicit component q_{e_j}. After that, for each $q_{e_i} \in E$ it computes the predecessor q_{s_i} for the symbolic state q_{s_j}, restricted to $T_{s_{ij}}$, and adds (q_{e_i}, q_{s_i}) to S.

```
01 S ← ∅
02 for each pair (q_{e_j}, q_{s_j}) of I_1 do
03   begin
04     E ← {q_e | ∃(q_e, q_{e_j}) ∈ T_e}
05     for each q_{e_i} ∈ E do
06       begin
07         q_{s_i} ← EX_{symb}(T_{s_{ij}}, q_{s_j})
08        if (q_{s_i} ≠ false) then
09          begin
10             if (q_{e_i}, q_{s_k}) is in S for some q_{s_k} ∈ Q_s then
11               replace (q_{e_i}, q_{s_k}) with (q_{e_i}, q_{s_k} ∨ q_{s_i}) in S
12             else
13               add (q_{e_i}, q_{s_i}) to S
14          end
15       end
16   end
```

Note that $EX_{symb}(T_{s_{ij}}, q_{s_j})$ produces the predecessors of q_{s_j} over the symbolic transition $T_{s_{ij}}$, where $T_{s_{ij}}$ is the symbolic transition associated with the explicit transition from q_{e_i} to q_{e_j}. This restriction guarantees that the explicit transition (q_{e_i}, q_{e_j}) and the symbolic transition (q_{s_i}, q_{s_j}) occur simultaneously in the explicit and symbolic model, respectively.

The entire computation requires time $O(|I_1|.(|T_e| + |Q_e|.(|Q_s|^2 + lg|Q_e|)))$. The main loop is executed $|I_1|$ times, one iteration per input explicit-symbolic state. For each input state (q_{e_j}, q_{s_j}), the executing time is dominated either by the computation of the predecessor states of q_{e_j} at line 04, what demands time $O(|T_e|)$, or by the inner loop. The inner loop is executed at most $|Q_e|$ times, one time per each predecessor q_{e_i} of q_{e_j}. Initially, each iteration of the inner loop computes the symbolic predecessor q_{si} of q_{s_j} over the symbolic transition $T_{s_{ij}}$, what requires time $O(|Q_s|^2)$. After that, according to the condition at line 08, the algorithm searches for the state (q_{e_i}, q_{s_i}) in S, what demands time $O(lg|Q_e|)$, and either computes a disjunction, in time $O(|Q_s|^2)$, or insert the pair (q_{e_i}, q_{s_i}) into the ordered set S, in time $O(lg|Q_e|)$.

6.3.2.4 $E(\psi\, U\, \gamma)$

Given that ψ and γ are represented by sets of input explicit-symbolic states I_1 and I_2 respectively, the algorithm produces the set of output explicit-symbolic states S where $E(\psi\, U\, \gamma)$ holds. The algorithm computes $E(\psi\, U\, \gamma)$ determining states where γ holds and looking backwards for states where ψ holds, until converging to the greatest set where $E(\psi\, U\, \gamma)$ holds.

```
01 S ← ∅
02 for each pair (q_e, q_s) of I_2 do
03   add (q_e, q_s) to S
04 do
05    continue ← false
06    Aux ← EX(S)
07    for each pair (q_e, q_s) of Aux do
08       if ((q_e, q_s) is not in S and (q_e, q_s) is in I_1) do
09          begin
10             add (q_e, q_s) to S
11             continue ← true
12          end
13 while (continue)
```

Because $E(\psi\, U\, \gamma)$ holds in states where γ holds, first the algorithm adds states of I_2 to S, in time $O(|I_2|)$. After that, it computes the predecessors of states in S, named Aux, using the previously EX algorithm defined over explicit-symbolic states. Note that such a

computation only need to consider the frontier at each iteration, that is, the states that were added to S during the last iteration. This step, shown at line 06, requires time $O(|Q_e|.(|T_e|+|Q_e|.(|Q_s|^2+lg|Q_e|)))$. During the loop spanning lines $07-12$, the algorithm looks for states of Aux included in I_1 and not yet in S. Those states are added into S. This loop requires time $O(|Q_e|.lg|Q_e|)$, because it is executed $|Q_e|$ times and each iteration demands a search $O(lg|Q_e|)$. Note that the computation of the explicit-symbolic predecessors dominates the executing cost of the `while` loop. Because the `while` loop is executed at most $|Q_e|$ times until no state is included in S, it is $O(|Q_e|^2.(|T_e|+|Q_e|.(|Q_s|^2+lg|Q_e|)))$ and, therefore, determines the executing time of the entire computation, since $|I_2| \leq |Q_e|$.

6.3.2.5 $EG(\varphi)$

The implementation of the explicit-symbolic algorithm for $EG(\varphi)$ is adapted from the algorithm used for computing the explicit EG. The explicit $EG(\varphi)$ is computed over a modified state graph where all states at which φ does not hold are deleted and the transition relation is restricted accordingly. After that, the explicit version of the algorithm is based on the computation of strongly connected components (SCCs). In the explicit-symbolic version, the original state graph must be modified so that the relevant states are those where both explicit and symbolic components of the state hold. Because formulas are given in a structural representation being computed from elementary sub-formulas to the more complex ones in bottom-up fashion, the set of relevant states for the computation of EG over the combined model corresponds to the set I_1 of input explicit-states given by φ. So, initially we have to eliminate from the combined state graph all the states not found in I_1. In the combined state graph, the information about symbolic states is recorded by means of the symbolic transitions associated with explicit transitions, so the determination of the symbolic states require additional computation to existentially quantify out the next state variables from the symbolic transition. The algorithm below is used for computing the explicit-symbolic EG. Note that $X_s^{'}$ stands for the set of next state symbolic variables.

```
01 for each (q_e, q_s) ∈ I_1 do
02   for each T_s where (q_e, L) ∈ T_h, (q'_e, T_s) ∈ L do
03     if q_s ∈ ∃X'_s(T_s) then
04       mark q_e as a candidate state
05 for each q_e ∈ Q_e do
06   if q_e is not marked
07     eliminate q_e from the state graph
08 SCC ← {C|C is a nontrivial SCC of T'_h}
09 S ← {(q_e, q_s)|(q_e, L) ∈ ⋃SCC, (q'_e, T_s) ∈ L, q_s = ∃X'_s(T_s)}
10 S' ← ∅
11 while (S' ≠ S) do
```

```
12  begin
13    S' ← S
14    for each (q_e, q_s) where (q_e, L) ∈ T'_h, (q'_e, T_s) ∈ L, q_s = ∃X'_s(T_s) do
15      if (q_e, q_s) ∉ S then
16        add (q_e, q_s) to S
17  end
```

In the algorithm above, lines from 01 to 07 generate the modified state graph $T_h^{'}$ where φ holds. First, the loop spanning lines $01-04$ marks the explicit states that are in accordance with the set of input states I_1 as candidate states. Candidate states are those that can compose at least one valid explicit-symbolic state that belongs to I_1. So, for each $(q_e, q_s) \in I_1$, the algorithm investigates each set of symbolic transitions T_s associated with explicit transitions starting from q_e. Each T_s is existentially quantified out, in time $O(|Q_s|^2)$, in order to determine if q_s belongs to the set of states where transitions of T_s come from. Because each explicit transition is exclusively associated with a set of symbolic transition T_s, such an operation is performed at most $|T_e|$ times. Next, the loop spanning lines $05-07$ eliminates all the non-marked states, in time $O(|Q_e|)$. Because $|Q_e| \leq |T_e|$, the computation described by lines from 01 to 07 is $O(|T_e|.|Q_s|^2)$. After that, line 08 computes the nontrivial strongly connected components considering only the explicit transitions of $T_h^{'}$. Such a computation can be adapted from the algorithm of Tarjan [Tarjan, 1972], which presents time $O(|Q_e^{'}|+|T_e^{'}|)$. Next, line 09 adds all the explicit-symbolic states belonging to the strongly connected components into the solution S, in time $O(|T_e^{'}|.|Q_s|^2)$. Finally, the `while` loop described at lines from 10 to 17 is used for finding all of those states that lead to states in S. The `while` iterates at most $|Q_e^{'}|$ times, where this upper bound corresponds to the case where only one state is added to S during each iteration and all the candidate states belong to the solution. In this worst case, the existential quantification found at line 14 and the search over S found at line 15 are executed $|T_e^{'}|$ times. So, the `while` loop takes time $O(|T_e^{'}|.(|Q_s|^2 + lg|Q_e^{'}|))$. Therefore, the entire computation requires time $O((|T_e|.|Q_s|^2) + |Q_e^{'}| + (|T_e^{'}|.(|Q_s|^2 + lg|Q_e^{'}|)))$.

6.4 The Microwave Oven Model

In this section, we present an example in order to clarify the generation of the explicit-symbolic model and the application of related techniques and algorithms applied to IF descriptions. Although we have mentioned the alternate bit protocol in our previous discussion about the modeling language, for didactic reasons we illustrate the model checking on a small example that describes the behavior of a microwave oven, taken from [Clarke et al., 1999].

6.4.1 Decomposition of the Microwave Oven Model

Assume that the behavior of the microwave oven is modeled by the process (Q, X, T, q_0), given by the Kripke structure shown in figure 6.2, where the set of states Q is represented by ellipses, the set of variables X is represented by propositions shown within ellipses, that is $X = \{start, close, heat, error\}$, the transition relation T is represented by arcs and $q_o = S1$. For clarity, each state is labeled with both atomic propositions that are true in the state and negations of the propositions that are false in the state. Labels on the arcs indicate the actions that cause transitions but are not part of the Kripke structure.

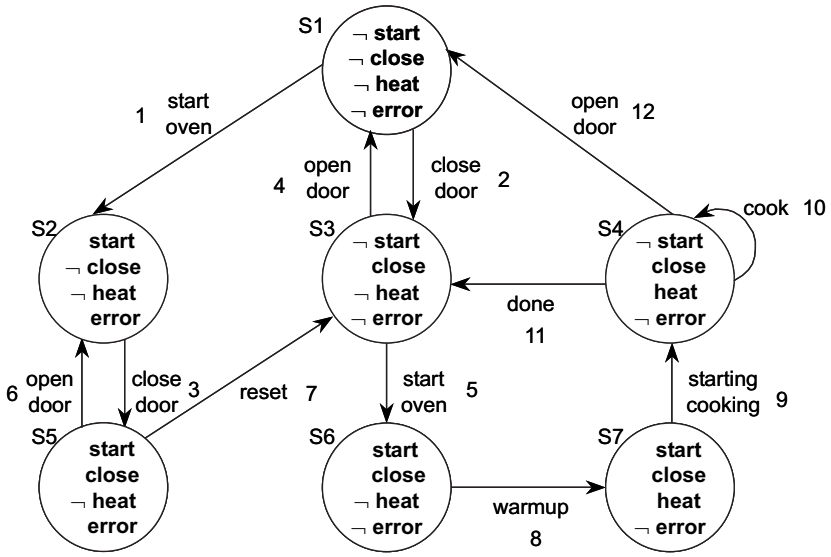


Figure 6.2: The microwave oven model.

Suppose that the set X is decomposed into subsets of explicit and symbolic propositions $X_e = \{start, close\}$ and $X_s = \{heat, error\}$, respectively. Now, consider the set of states under a special point of view where only explicit propositions matter. Because states with the same set of explicit propositions are not distinct under such explicit point of view, such states are considered the same and are projected as one explicit state. Figure 6.3 shows the grouping of states considering explicit propositions (a) and the corresponding explicit process (Q_e, X_e, T_e, q_{0e}) generated (b). Note that we have another set of states, Q_e, and another set of propositions composed only of explicit propositions, X_e. The arcs on figure 6.3(b) correspond to the explicit transition relation T_e and have been labeled in order to reveal the original transitions they were projected from. Naturally, $q_{oe} = ES1$.

The symbolic process (Q_s, X_s, T_s, q_{0s}) is generated in a similar manner, by considering the original model under the symbolic point of view. Figure 6.4 presents the grouping of symbolic states (a) and the corresponding symbolic process (Q_s, X_s, T_s, q_{0s})(b).

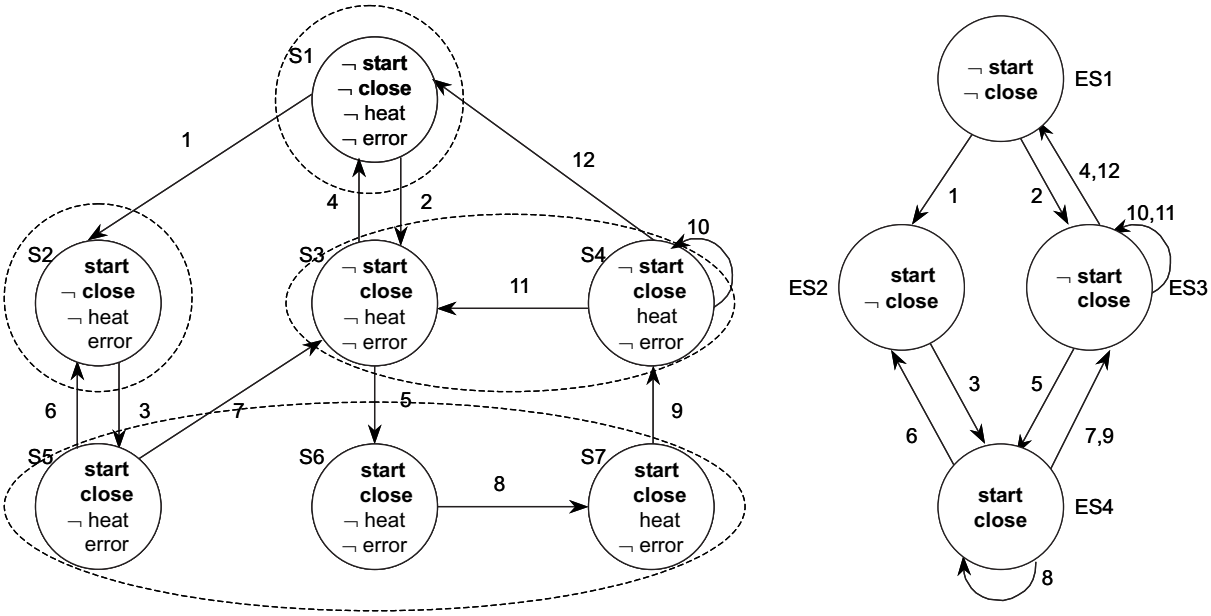


Figure 6.3: Grouping of explicit states (a) and corresponding explicit model (b).

6.4.2 Verification on the Explicit-Symbolic Microwave Oven Model

Given the explicit and symbolic models shown in figures 6.3(b) and 6.4(b), respectively, explicit and symbolic transitions with labels in common should be linked together in order to establish their interdependence during model checking. This subsection explores the operation of such combined model for checking CTL expressions. The expression $EX(\neg close \vee error)$ is the expression of interest, because it allows us to explore some of the most important CTL operators. According to the previous partitioning, $close \in X_e$ and $error \in X_s$. Note that all the algorithms produce explicit-symbolic states as output.

Finding States Where *close* Holds

By using the algorithm for processing explicit atomic propositions (subsection 6.3.1.1), line 02, we generate the explicit states where *close* holds, $E = \{ES3, ES4\}$. The loop spanning lines 03-05 generates the set S of explicit-symbolic states where *close* holds, such that $S = \{(ES3, SS2), (ES3, SS3), (ES4, SS1), (ES4, SS2), (ES4, SS3)\}$.

Finding States Where ¬*close* Holds

By using the negation algorithm (subsection 6.3.2.1) over the set of explicit-symbolic states where *close* holds, specifically lines 04-07, we generate $S = \{(ES1, SS2), (ES2, SS1)\}$.

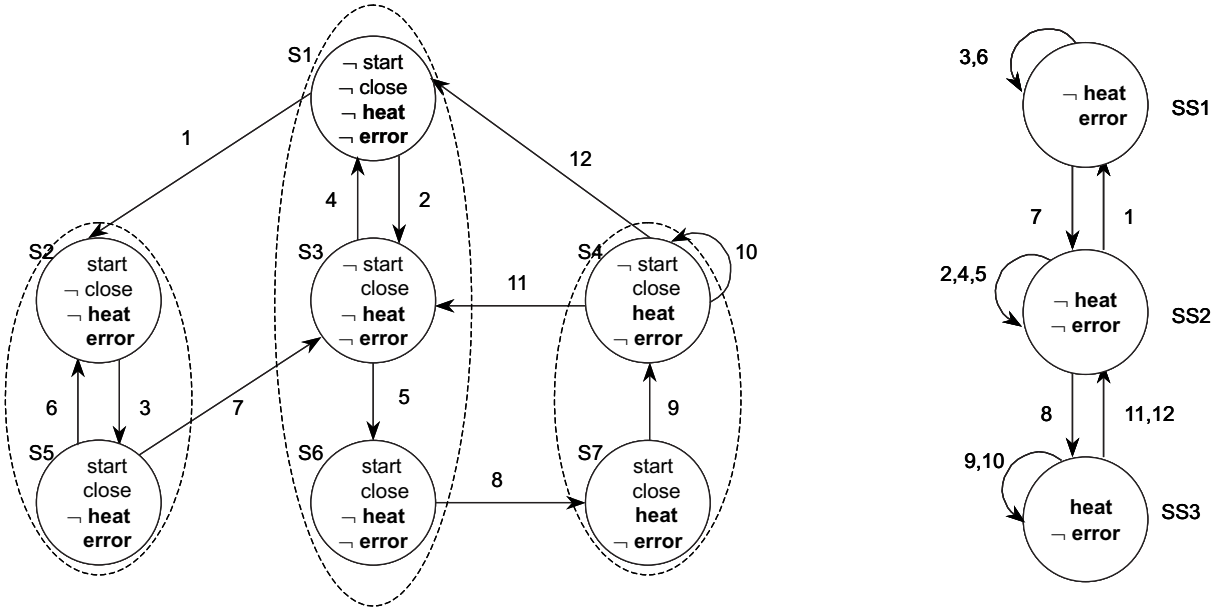


Figure 6.4: Grouping of symbolic states (a) and corresponding symbolic model (b).

Finding States Where *error* Holds

By using the algorithm for processing symbolic atomic propositions (subsection 6.3.1.2), line 02, we have that q_{s_i} represents the set $\{SS1\}$. Lines 03-09 produce $S = \{(ES2, SS1), (ES4, SS1)\}$.

Finding States Where $\neg close \vee error$ Holds

By using the disjunction algorithm (subsection 6.3.2.2), lines 02-03, we add every state where $\neg close$ holds into the output set. After that, lines 04-08 include states where *error* holds into the output set, producing $S = \{(ES1, SS2), (ES2, SS1), (ES4, SS1)\}$.

Finding States Where $EX(\neg close \vee error)$ Holds

Given the input state set $I_1 = \{(ES1, SS2), (ES2, SS1), (ES4, SS1)\}$, consider that it is processed in the order presented. Each iteration of the outer loop of the EX algorithm (subsection 6.3.2.3) considers a specific input explicit-symbolic state. For the first state, $(ES1, SS2)$, the algorithm initially computes the set of predecessors E_1 for its explicit component, $ES1$, by using line 04. After that, the loop spanning lines 05-15 computes predecessors of the symbolic component, $SS2$, by considering only symbolic transitions associated with relevant explicit transitions. The relevant explicit transitions are those from states of E_1 to $ES1$. Each symbolic predecessor found and its corresponding explicit predecessor compose an explicit-symbolic state of S_1. Below, we show results from the first to the third and last iteration of the outer loop of the EX algorithm. The set of explicit-

symbolic states of the third iteration, S_3, are those where $EX(\neg close \vee error)$ holds. Such explicit-symbolic states correspond to the original states $\{S4, S3, S1, S5, S2\}$, as expected.

$E_1 = \{ES3\}$
$S_1 = \{(ES3, SS3), (ES3, SS2)\}$.

$E_2 = \{ES1, ES4\}$
$S_2 = \{(ES3, SS3), (ES3, SS2), (ES1, SS2), (ES4, SS1)\}$.

$E_3 = \{ES2, ES3, ES4\}$
$S_3 = \{(ES3, SS3), (ES3, SS2), (ES1, SS2), (ES4, SS1), (ES2, SS1)\}$.

6.5 The Explicit-Symbolic Model Flexibility

Our explicit-symbolic model has important differences with regard to the SLAM project. Instead of dealing with a fixed approach, where control-flow and data-flow information must have explicit and symbolic representations, respectively, our approach is more general. The explicit-symbolic model allows us to move variables between explicit and symbolic spaces according to any policy. Thus, we can experiment with a variety of combinations and choose the one that best fits the system needs. Consider, for example, systems with data dependent control, that is, systems where the control-flow depends on evaluations of symbolically represented variables. In such cases, we have the opportunity of improving the overall performance by moving those symbolic variables to the explicit model, reducing interaction between explicit and symbolic representations. Naturally there are other questions involved, as the dependence between those symbolic variables and the remaining ones, but the proposed explicit-symbolic model can support such configuration whenever it is advantageous. As another example of the generality of the flexibility achieved, our explicit-symbolic model also supports the partitioning of variables accomplished by SLAM. Because the program counter and variables involved in control guards of IF systems determine control-flow information, it suffices to move such variables to the explicit model in order to achieve a SLAM-like partitioning. Therefore, the explicit-symbolic model offers the possibility of using a more flexible environment to evaluate different representations and choose the one that improves the model-checking procedure.

Chapter 7

Checker Implementation

This chapter reveals the details of implementing the combined explicit-symbolic model checker. It presents the main ideas and decisions taken during the conception of the explicit and symbolic underlying models and the implementation of the combined model. Because the checking algorithms have been introduced in a previous chapter, the current discussion focuses on the construction of models and on the basic data structures involved in this procedure.

7.1 Introduction

The implementation of the explicit-symbolic checker started from scratch. Although we have used well-established model checkers as inspiration, the direct usage of those checkers as building blocks would probably lead us to compatibility problems, hard or even impossible to be solved. Such inconveniences arise from the different specification logics and the different modeling languages used by the checkers, for example. Also, the integration of third-part explicit and symbolic checkers would require a high comprehension about the operation and improvement techniques used by each component to be integrated. Thus, the explicit and symbolic components have been implemented individually from scratch and their algorithms and data structures have been designed to be compatible to each other.

The explicit component has been implemented according to some of the traditional explicit techniques but it does not follow any checker rigorously. Instead of expressing properties as automata and performing the model checking by means of automaton operations, as done by SPIN [Holzmann, 1997b], the developed algorithms virtually label each state with the properties true on that state. The current project is naive in the sense that it lacks more robust static analysis techniques and partial order reduction to improve the exploration of the state space and to minimize the state explosion problem. Currently, non-reachable explicit states may be generated during the construction of the explicit model.

On the other hand, we employ state vector compression for saving memory and introduce a technique to reduce the time spent in explicit state lookup.

The symbolic component we have implemented is largely based on Verus [Campos, 1996]. Both model checking and construction algorithms have been adapted from Verus. The quantitative timing information has been removed from the modules taken from Verus, but all the fine-tuning has been inherited.

The explicit-symbolic algorithms assume the system properties are given in CTL. CTL formulas are given in a structural representation by means of a parsing tree. In such a parsing tree, the leaves stand for atomic propositions, represented either in the explicit model or in the symbolic model, while internal nodes stand for explicit-symbolic expressions, represented in the explicit-symbolic model. In spite of the proposed model checking algorithms initially convert both explicit and symbolic states into corresponding explicit-symbolic states before checking formulas, the real implementation explores each underlying model as much as possible before integrating the explicit and symbolic models. This decision reduces the synchronization overhead and makes the checking of the underlying models more independent. The following sections discuss the implementation details involving both explicit and symbolic components of the combined explicit-symbolic checker.

7.2 Basic Data Structures

The most elementary data structure are the symbols. There is a symbol per each variable in the system, including the user-defined variables and those created for controlling the model checking, as the variables used for the queue writing and reading positions. Symbols used for storing explicit variables are not distinct from symbols used for storing symbolic variables. In fact, a symbol can potentially store both aspects, although usually just one of them is activated. Syntactically, symbols are described by the type `t_symbol`:

```
type t_symbol = record
  id: string;
  class: integer;
  nbits: integer;
  signal_info: t_signal_info *;
  storage: t_storage *;
  state_info: t_state_info *;
  index: integer;
  lower_index: integer;
  upper_index: integer;
end;
```

Symbol Classes	Application
TYPE_CLASS	Defines basic and user-defined types.
SIGNAL_CLASS	Defines signals, by means of the field signal_info.
STATE_CLASS	Defines states, by means of the field state_info.
VARIABLE_CLASS	Defines local variables.
GLOBAL_CLASS	Defines global variables, as the queue variables.
ENABLE_CLASS	Defines which process is enabled at the current time.

Table 7.1: Symbol Classes.

Each symbol has a unique field id, dedicated to its identification. Symbols are stored into a hash table and the hash function is based on such a field. The field class is used to create symbol categories according to symbol applications. There are six symbol categories, as shown in table 7.1. The field nbits stores the number of bits required by a type, being concerned with type definitions (TYPE_CLASS). Fields signal_info and state_info are used to store information regarding the parameter types for a signal (SIGNAL_CLASS) and the identification of states (STATE_CLASS), respectively. The explicit or symbolic representation of each variable (VARIABLE_CLASS, GLOBAL_CLASS, ENABLE_CLASS) is maintained by means of the field storage, discussed below. Although local variables, global variables and symbols used to enable processes have the same structure, we distinguish them due to semantic reasons. The three last fields, index, lower_index and upper_index, have to do with the explicit representation and are going to be discussed later in this chapter.

The construction of the combined explicit-symbolic model is mainly based on the symbols corresponding to variables, both local and global ones, and the symbols used for enabling processes. Specifically, the construction of models depends on the storage associated with those symbols. Syntactically, storages are given by means of t_storage structures:

```
type t_storage = record
  base_type: t_symbol *;
  representation: integer;
  current_state: [bdd];
  swap_bdds: [bdd];
  next_state: [bdd];
  value: integer;
  seen_bits: integer;
  explicit_value: integer;
end;
```

The field base_type gives the symbol which represents the type of the storage and representation indicates if the symbol is explicit, symbolic or hybrid. Initially, while the symbols of a sys-

tem are being collected, the representation of each symbol is undefined. After that, the configuration file is used for determining their representations. Because each signal can carry several values, one value per parameter, each signal is associated with several storages, one storage per parameter. This is accomplished by creating a symbol per parameter and setting the type of each parameter in accordance with the `t_signal_info` structure of the signal, as discussed above. The other fields of a `t_storage` structure are explained in the next subsections.

7.2.1 Explicit Data Structures

Within the storage of each symbol there is an integer field named `explicit_value`. The `explicit_value` is determined by reading the value of the corresponding variable from the state vector. A state vector represents an explicit state and consists of an array of integer values, one value per explicit symbol. During the evaluation of an expression, its subexpressions are recursively broken into simpler subexpressions, until only subexpressions composed exclusively of symbols for variables have been found. At that moment, the algorithms update the `explicit_value` of the symbols and the evaluation continues. The evaluation of an expression produces an explicit symbol whose `explicit_value` corresponds to the result of the evaluation of the expression. Note that symbols are not able to represent explicit finite state machines. Symbols just store values of individual explicit symbols for a specific state vector. Our implementation uses another data structure to represent the explicit states and the transitions among them:

```
type t_explicit_state = record
  state_vector: [unsigned char];
  destination_states: t_explicit_destination *;
  visited: boolean;
  mark: boolean;
end;
```

where `state_vector` is an array of integer values and `destination_states` is the linked list of the explicit states that are reached from the current state by executing exactly one transition. In other words, `state_vector` gives the configuration assumed by an explicit state and `destination_states` describes the transitions starting from that state. In subsection 7.2.3 we discuss `destination_states` with more details. Fields `visited` and `mark` are used by model checking algorithms: `visited` controls the depth-first search, indicating states already explored, `mark` indicates states where a specific property holds.

The number *nbytes* of bytes required by the state vector is the same for all the processes of a given system. *nbytes* is computed by considering a compressed version of the state vector, according to the following rule:

$$nbytes = \left\lceil \sum_{i=0}^{n-1} symbol_i.nbits/8 \right\rceil$$

where $symbol_i.nbits$ stands for the number of bits required for representing the $i-th$ explicit symbol. Instead of storing the decimal value of each explicit symbol into a byte, the compression algorithm requires exactly $symbol_i.nbits$ bits for storing the equivalent binary value of $symbol_i$ within the state vector. This compression method presents significant memory savings for symbols where $nbits < 8$. In order to determine the sequence of bits of a given symbol, we store the `lower_index` and `upper_index` numbers within each symbol. Together, `lower_index` and `upper_index` indicate the positions where the sequence of bits for the symbol begins and ends, respectively. This compression technique can produce sensitive memory savings, especially for systems where variables have short value ranges, which seems to be the general case. Naturally, the memory saving is obtained at the expense of increasing the computational efforts, but the trade-off has shown to be positive.

7.2.2 Symbolic Data Structures

Binary decision diagrams are the basic data structure of the symbolic model. Both states and transition relation are represented by means of BDDs. Each symbolic state $q_s \in Q_s$ is represented by the set of values assigned to symbolic variables, such that $q_s \in 2^{X_s}$. The assignment of values is given by means of Boolean formulas, constructed over BDD operations. Transitions are represented by two sets of variables, a set of variables for the current state, X_s, and a set of variables for the next state of the transition, $X_s^{'}$. Each symbolic transition $t_s \in T_s$ is given by formulas involving those current and next state variables, that is, $T_s \subseteq (2^{X_s} \times 2^{X_s^{'}})$. The global transition relation T_s corresponds to the disjunction of such individual transitions. In order to support this modeling, each symbol has an array of BDDs for representing X_s, `current_state`, and another array of BDDs for representing $X_s^{'}$, `next_state`. Additionally, every symbol has a third array of BDDs, `swap_bdds`, used during the construction of the model to compute symbolic assignments.

Our implementation uses the BDD library developed by David E. Long, because we are very familiar with this package. Since the default options supplied by the library for printing BDDs is hard to read, we have taken the method used by Verus for printing BDDs. The implementation of this printing method requires, for each symbol, an integer value to store its cumulative decimal value, `value`, and maintains the number of bits already considered for such a computation, `seen_bits`.

7.2.3 Synchronization Data Structures

Given an origin explicit state, its set of next explicit states is represented by the field `destination_states`, a linked list of destination states. Together with each destination state this field stores the related symbolic transition, computed by considering only the

symbolic variables of the system over the same execution path. That is the way how explicit and symbolic transitions are synchronized in our implementation. Below, we present `t_explicit_destination`, the data structure used for representing the set of destination states and for linking explicit and symbolic transitions together:

```
type t_explicit_destination = record
  state: t_explicit_state *;
  symb_transition: bdd;
  next: t_explicit_destination *;
end;
```

where `state` stands for the destination state, `symb_transition` represents the symbolic transition that occurs together with this explicit transition and `next` points to the next destination state for the origin state of interest. It is used by the checking algorithms of the combined model to establish the integration and synchronization of the underlying models.

7.3 Construction of the Explicit Representation

Explicit variables are represented by an array of integer values usually called state vector. Within this array there is a specific index for each explicit variable. In our project, state vectors are implemented as arrays of unsigned chars in order to reduce the size required for storing them. Although unsigned chars are useful to enlarge the range of values of each explicit variable, they do not offer support to designs that deal with negative values. Fortunately, this decision does not affect the generality of our solution and it can be changed without significant efforts. Explicit states are identified by means of state vectors.

Explicit transitions are defined by two state vectors, one for the origin state and another for the destination state. Usually, many states and transitions are necessary to model the configurations assumed by each control state of an IF description and its transitions. In the following discussion, we focus on the processing of an individual explicit transition from state q to $q^{'}$. Assume that q and $q^{'}$ are described by state vectors X and $X^{'}$, respectively. For an explicit variable x, its values in the current state and in the next state are given by $X[index(x)]$ and $X^{'}[index(x)]$, respectively, where $index(x)$ returns the index of x within the state vectors. The index is retrieved by using the field `index` of the symbol standing for x. The global explicit state graph is obtained by linking adjacent transitions to each other. Given two explicit transitions $t_{e_{ij}}$, from q_i to q_j, and $t_{e_{kl}}$, from q_k to q_l, they must be adjacent if and only if q_j and q_k are described by the same state vector and, consequently, they are the same state. In our work, the set of visited states is maintained by means of a cache of explicit states, as described in section 7.7. Given a state vector, the cache looks for the existence of its corresponding state. If the state is already there, the cache simply returns a pointer to it. Otherwise, first the appropriate state is created and inserted into the cache. Next, the cache returns a pointer to that state. Because the creation of each

explicit transition demands the creation of its source and destination states, cache queries are used to identify adjacent transitions.

7.3.1 `state` Statements

In order to control the execution flow, our implementation uses an additional variable per process. This variable, called state counter from now on, assumes a different value for each control state of the process and is used for unequivocally identifying the current state and the next state of transitions. Each transition is identified by the clock number of the origin control state and the clock number of the destination state, according to `state` and `nextstate` statements. `state` statements are used to initialize the transition when a new control state is found in the description of a system. They reflect the fact that a new transition will be computed by determining the state counter of the current state and by establishing that the values of variables do not change across transitions. For establishing that $\texttt{state}_i$ is the origin state of the transition being created, we assign the state counter variable (sc) the identifier of this state in the origin state vector:

$$\texttt{state state}_i\ \texttt{endstate} \Leftrightarrow X[index(sc)] \leftarrow i$$

Because the state vector for the destination state is initialized to be the same as the state vector for the origin state, we guarantee that variables values are maintained unless that they are assigned new values.

7.3.2 `nextstate` Statements

`nextstate` statements are intended to transfer the control from one state to another at the end of transitions. These statements update the state vector with the value corresponding to the destination state. Let sc represent the state counter variable of the process being modeled. The following expression shows how the state vector is updated:

$$\texttt{nextstate state}_i \Leftrightarrow X'[index(sc)] \leftarrow i$$

Notice that `nextstate` statements can take place in an implicit manner, during the execution of `input` or `output` statements. Because these two statements can block the model, when the required signal is not at the head of the queue of pending messages and when the queue is full, respectively, dummy transitions are used to model the blockage. Dummy transitions are self-loop transitions where all the variables, except the clock ones, have their values maintained. So, these statements also employ this function to model transitions.

7.3.3 Expressions

In the explicit model, expressions are evaluated by replacing each variable occurrence with its corresponding value and by executing regular operations over such values. The value of

a variable is determined according to the state vector of the destination explicit state. In a state vector, Boolean variables are represented by integer values, 0 standing for $False$ and 1 standing for $True$, as shown by the function E below:

$$E[\texttt{true}] = 1$$
$$E[\texttt{false}] = 0$$

The representation of integer variables is straightforward:

$$E[\texttt{x}] = X^{'}[index(x)]$$

References to IF variables denotes values in the state vector for the destination state, it does not matter if the variable occurs on the left-hand side or on the right-hand side of an assignment. The state vector for the destination state contains the most recent variable values, being the one used when reading or writing variables in expressions. Unlike symbolic operations, explicit operations over integer values have direct implementation, being computed by usual operators of the programming language. Before starting the scanning of a control state, the state vector for the origin and destination states are the same, that is, $X[index(x)] = X^{'}[index(x)]$ for all the explicit variables x. The origin state vector is maintained throughout the computation of the transition, but the destination state vector is updated according to the statements found within the control state.

7.3.4 Assignments

Explicit assignments correspond to trivial assignments of programming languages:

$$\texttt{task x := expr} \Leftrightarrow X^{'}[index(x)] \leftarrow E[\texttt{expr}]$$

7.3.5 Sequential Execution

The sequential execution corresponds to consecutive updates of the state vector of the destination state, according to the statements found within an IF transition:

$$\texttt{stmt}_1; \texttt{stmt}_2 \Leftrightarrow E[\texttt{stmt}_1]; E[\texttt{stmt}_2]$$

The expression above considers statements of an individual transition statement.

7.3.6 Nondeterministic Execution

Each explicit transition is defined by a pair of origin and destination state vectors. Initially, the explicit transition is updated with general information about the control state being modeled. The initial explicit transition is supplied for each transition of the control state, and the updates accomplished by one transition do not affect the modeling of another transition. Because we have to consider the nondeterministic execution of all transitions, all the active transitions found at the end of a transition scanning must be represented into the explicit state-transition graph. Active transitions are those whose state vector does not configure blocking conditions.

7.3.7 Conditionals

Conditionals are modeled by evaluating the appropriate conditional according to the values found in the state vector for the destination state, X'. If the condition is true, the `then` statements are modeled. Otherwise, the `else` statements are modeled.

7.3.8 Loops

The modeling of explicit loops considers that the loop is unrolled into nested `if` statements, that is `while expr do stmt endwhile` $\equiv$ `if expr then {stmt; if expr then {stmt;...} endif} endif`. So, `while` loops could be modeled simply as successive updates of the state vector by considering the whole set of statements produced by unrolling the loop. However, because the construction of the explicit model requires the simulation of the statements of the system being modeled in order to update the state vector for the destination explicit state, the occurrence of endless loops in the system descriptions would lead the construction algorithm to execute an endless loop too. Consider the loop below:

```
while (true) do
  stmt
endwhile
```

During the generation of the explicit model the state vector for the destination state should be successively updated until the evaluation of the `while` condition be false with regard to the values found in the state vector. Because the `while` condition for the example will never be false, the construction algorithm would simulate the `while` statement forever. So, the modeling of endless loops requires new conditions to be created to make sure that the modeling finishes. This new condition is called *halt* condition in our discussion. Regarding the halt condition, we maintain a stack with the state vectors produced by the previous iterations of the loop for detecting the occurrence of cycles. After modeling each iteration, we detect the occurrence of cycles. The halt condition is true when the loop condition is false or when a cycle has been found. In both cases the `while` modeling finishes.

Additionally, it is necessary to distinguish individual iterations of the loop in order to ensure that even non-terminating `while` loops are always observable. So, we model each iteration as a transition, by inserting a `nextstate` statement followed by a `state` statement at the beginning of the body of every loop:

```
while (true) do
  nextstate qi
  state qi
  stmt
endwhile
```

where $\mathtt{q}_i$ represents the state being modeled at the $i-th$ iteration. Therefore, there are transitions between the blocks of statements obtained by unrolling consecutive iterations of the loop and endless loops are modeled as states with self-loops, as shown in figure 7.1. Because the creation of an explicit state $\mathtt{q}_i$ checks the previous existence of $\mathtt{q}_i$, the loop unrolling creates unique representations of each $\mathtt{q}_i$.

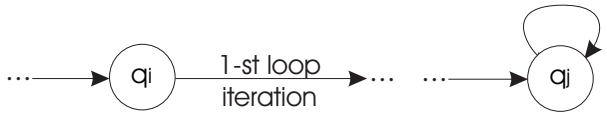


Figure 7.1: Endless `while` modeling.

7.4 Construction of the Symbolic Representation

The symbolic model deals only with Boolean variables, represented by means of BDDs. Integer variables are encoded in binary, being also represented by BDDs. There are two sets of Boolean variables, X and $X^{'}$, such that for each IF variable x there are corresponding symbolic variables $x \in X$ and $x^{'} \in X^{'}$. The symbolic variable $x \in X$ represents the value of the IF variable x in the current state and the symbolic variable $x^{'} \in X^{'}$ represents its value in the next state. So, transitions are relations between variables in X and $X^{'}$. The techniques and algorithms used by our implementation to convert the statements that compose a system description into the symbolic representation have been almost fully taken from Verus. Minor adaptations have been done to make Verus algorithms compatible with our description language and to remove quantitative timing information from models.

Let (Q_s, X_s, T_s, q_{0s}) be the description of an IF process. Given an execution sequence leading from q_{s_i} to q_{s_j} without intermediate transitions, where $q_{s_i}, q_{s_j} \in Q_s$, the relation $t_{s_{ij}} = (q_{s_i}, q_{s_j}) \in T_s$ which records the effect of moving the system from q_{s_i} to q_{s_j} must be modeled into the symbolic representation. The function R below is used to construct the relations between symbolic states:

$$R : (STMT \times Q_s \times Q_s \times T_s) \rightarrow (Q_s \times Q_s \times T_s)$$

where $STMT$ stands for the set of statements that compose the system description. Intuitively, given the relation $t_{s_{ij}}$ describing the behavior of the system until the execution of the statements within q_{s_i}, R produces the relation $t'_{s_{ij}}$ describing the system after executing those statements. Note that the function R also constructs the relation T_s by accumulating the relations constructed for all the transitions. Given a set of statements $STMT$, the corresponding state-transition graph T_s is produced according to the function R:

$$\langle t, T_s \rangle = R[STMT]\langle Q_{s0} \times Q_s, \emptyset \rangle$$

where T_s is the final symbolic transition relation of the state-transition graph for the system being modeled, and Q_{s0} is the set of symbolic initial states.

7.4.1 `state` Statements

As done for the explicit model, each transition is identified by the clock number of the current control state and the clock number of the next control state, according to `state` and `nextstate` statements. `state` statements are used to initialize the transition when a new control state is found in the description of a system. They reflect the fact that a new set of transitions will be computed by determining the state counter of the current state and by establishing that the values of variables do not change across transitions, according to the function below:

$$R[\texttt{state state}_i\ \texttt{endstate}]\langle t_s, T_s \rangle = \langle ((sc = i) \wedge \bigwedge_{x \in X} x = x'), T_s \rangle$$

where X is the set of variables in the system.

The new relation specifies that transitions start in $\texttt{state}_i$ $(sc = i)$, that is, the state counter of the current state variable in the transition will be i. The destination of the new set of transitions will be established when the `nextstate` statement is found. Together, those conditions are used to specify that each transition have state counter values for both current and next states. The expression $\bigwedge_{x \in X} x = x'$ guarantees that each variable maintains its previous value across transitions, unless assigned a new value.

7.4.2 `nextstate` Statements

Below, we show the function used to compute symbolic transitions:

$$R[\texttt{nextstate state}_i]\langle t_s, T_s \rangle = \langle ((sc = i) \vee t_s), (((sc = i) \vee t_s) \vee T_s) \rangle$$

The function above changes the relation in two ways. First, it updates the current symbolic transition to reflect the fact that it leads to $\texttt{state}_i$. This is accomplished by assigning the identifier of $\texttt{state}_i$ to sc, the state counter of the process of interest. Second, it updates the transition relation with the current symbolic transition after its updating.

7.4.3 Expressions

In the symbolic model, expressions built over symbolic variables are translated into corresponding Boolean formulas. Roughly, there are three basic kinds of values that an IF variable can assume: Boolean, integer and float values. Float values are not supported in our implementation. The translation of variables with Boolean values is straightforward, as shown by the function E below:

$$E[\texttt{true}] = True$$
$$E[\texttt{false}] = False$$

Regarding IF integer variables, their values are determined by the array of BDDs corresponding to the next state variables.

$$E[\texttt{x}] = x^{'}$$

Operations over integer values have symbolic versions that operate over the individual variables in the next state array of variables. Each symbolic variable corresponds to a bit of the value being represented. So, operators defined over symbolic variables are bitwise.

References to IF variables denote their corresponding next state variables, it does not matter if the IF variable occurs on the left-hand side or on the right-hand side of an assignment. When several values are assigned to the same variable within a control state, the last value is the one found in the next state variables. It means that an assignment overrides the previous value of the variable on the left-hand side, as done in regular programming languages. This is the expected behavior, because an assignment reads the most recent value from the variables on the right-hand side in order to determine the value that the variable on the left-hand side will assume in the next state. Before starting the scanning of a control state, the symbolic transition relation is initialized in order to enforce that the current and next state variables have the same value, by using the following expression:

$$\bigwedge_{x \in X} x = x^{'}$$

which considers both local and global variables. Consequently, the value of a variable does not change if no assignments are made, even across transitions.

7.4.4 Assignments

Given the set t_s of valid transitions in the graph since the computation of the last transition, the expression below determines the largest set of transitions that satisfies the assignment and satisfies t_s for variables other than x:

$$R[\texttt{task x:=expr}]\langle t_s, T_s\rangle = \langle(\exists y[x = Expr\,{}^{y}/_{x} \wedge t_s\,{}^{y}/_{x}]), T_s\rangle$$

where $x = E[\texttt{x}]$, $Expr = E[\texttt{expr}]$ and y is a new variable. This expression computes the strongest post-condition for the assignment `task x:=expr` given the pre-condition t_s. Intuitively, this expression substitutes the previous value of x in t_s for $Expr$, while maintaining the values of all the other variables.

7.4.5 Sequential Execution

The sequential execution updates the symbolic transition by accumulating the consecutive statements found along the scanning of transitions of a control state:

$$R[\texttt{stmt}_1; \texttt{stmt}_2]\langle t_s, T_s\rangle = R[\texttt{stmt}_2](R[\texttt{stmt}_1]\langle t_s, T_s\rangle)$$

The expression above considers statements of an individual transition statement. Two or more transitions are executed in a nondeterministic manner.

7.4.6 Nondeterministic Execution

IF transitions are executed nondeterministically and all the possibilities have to be considered. They are evaluated according to the following function:

$$\begin{aligned} R[\texttt{transition}_1; \texttt{transition}_2]\langle t_s, T_s\rangle = \langle t_s, T_s^{'} \vee T_s^{''}\rangle \ \texttt{where} \\ \langle t_s^{'}, T_s^{'}\rangle = R[\texttt{transition}_1]\langle t_s, T_s\rangle\texttt{,} \\ \langle t_s^{''}, T_s^{''}\rangle = R[\texttt{transition}_2]\langle t_s, T_s\rangle \end{aligned}$$

The relation for the nondeterministic execution of transitions is the disjunction of the transition relations obtained for each possible IF transition. The extension of R for the case in which more than two transitions exist is a simple extension of the disjunction shown.

7.4.7 Conditionals

The branches in `if` statements are executed by restricting their parameters to the set of transitions that satisfy the appropriate conditional: $\texttt{stmt}_1$ receives those transitions satisfying `expr` and $\texttt{stmt}_2$ receives those transitions not satisfying `expr`. Thus, whether control reaches the `if` statement through a state that satisfies the condition, control will proceed to $\texttt{stmt}_1$. Otherwise, control proceeds to $\texttt{stmt}_2$. The representation of a conditional is the disjunction of the representation of its branches.

$$\begin{aligned} R[\texttt{if expr then stmt}_1 \texttt{ else stmt}_2 \texttt{ endif}]\langle t_s, T_s\rangle = \langle t_s^{'} \vee t_s^{''}, T_s\rangle \ \texttt{where} \\ \langle t_s^{'}, T_s\rangle = R[\texttt{stmt}_1]\langle(t_s \wedge \texttt{expr}), T_s\rangle \\ \langle t_S^{''}, T_s\rangle = R[\texttt{stmt}_2]\langle(t_s \wedge \neg\texttt{expr}), T_s\rangle \end{aligned}$$

7.4.8 Loops

The modeling of a `while` loop can be seen as unrolling the loop into nested `if` statements, that is `while expr do stmt endwhile` $\equiv$ `if expr then {stmt; if expr then {stmt;...} endif} endif`. Consequently, `while` loops could be modeled as shown below:

$$
\begin{aligned}
R[\texttt{while expr do stmt endwhile}]\langle t_s, T_s\rangle = \langle t_s' \vee t_s'', T_s' \vee T_s''\rangle \ \texttt{where} \\
\langle t_s', T_s'\rangle = R[\texttt{while expr do stmt endwhile}](R[\texttt{stmt}]\langle (t_s \wedge expr), T_s\rangle \\
\langle t_s'', T_s''\rangle = \langle (t_s \wedge \neg expr), T_s\rangle
\end{aligned}
$$

However, the function R above cannot be computed because it is circular. On the other hand, `while` loops can be modeled by using a least fixpoint characterization:

$$
\begin{aligned}
R[\texttt{while expr do stmt endwhile}]\langle t_s, T_s\rangle = lfp(\lambda f \lambda \langle t_s, T_s\rangle . \langle t_s' \vee t_s'', T_s' \vee T_s''\rangle \\
\texttt{where } \langle t_s', T_s'\rangle = f(R[\texttt{stmt}]\langle (t_s \wedge expr), T_s\rangle) \\
\langle t_s'', T_s''\rangle = \langle (t_s \wedge \neg expr), T_s\rangle)
\end{aligned}
$$

The operations performed above are projection, from the result of the application of f into t_s' and T_s, disjunction involving t_s', t_s'' and T_s', T_s'', and pairing of the results of the disjunctions. Because these operations are continuous [Winskel, 1993], any function constructed from them is also continuous. By being continuous, the function is also monotonic and, therefore, has a fixpoint.

Nevertheless, not all `while` statements can be fully modeled only by means of the fixpoint computation above. Consider a `while` statement that models an endless loop by using a tautological expression as its condition. According to the fixpoint characterization, the relation for this endless loop is false, which corresponds to non-termination, as expected. In terms of the transition relation being constructed, it means that there will be no outgoing transitions from the current state. So, the non-terminating behavior is not observable [Campos, 1996]. In order to avoid this anomalous modeling, we artificially insert a `nextstate` statement followed by a `state` statement at the beginning of the loop body. In other words, each iteration of the loop is modeled by a transition. This ensures that even non-terminating `while` loops are always observable and that no states without outgoing transitions will be created.

7.5 Construction of the Combined Model

The construction of the combined explicit-symbolic model corresponds to the process of creating and coordinating explicit and symbolic representations derived from IF statements. At the initial stage of this process, all the symbols of the system are collected and stored into a symbol table with hash access. During this stage, the symbols used to represent basic data types and to control model-checking procedures are also inserted into the symbol table. However, the created symbols have undefined representations, neither explicit nor

symbolic. In the second stage, symbols are assigned representations according to the configuration file, supplied together with the description of the system. After those stages, we are able to measure important aspects of the global model, such as the scope of variables and the number of symbols in each underlying model. The initialization of data structures and the number of bytes required by the state vector are examples of decisions taken based on the information gathered.

The next stage is concerned with the creation of the combined explicit-symbolic model itself. According to the modeling, the behavior of the combined model is obtained by associating symbolic transitions with explicit transitions, in order to synchronize the behavior of the underlying models. For each IF control state, several explicit transitions are generated, each one with a corresponding symbolic transition. The scanning of control states begins by creating the set of origin explicit states and the set of origin symbolic states, which defines the initial configuration prior the execution of the statements within the control state. The nature of symbolic representations allows us to represent several states by means of just one Boolean formula, but explicit states must be enumerated. The most important information within an explicit state is its state vector, which records the values assigned to the set of explicit variables. Each explicit state demands the creation of a particular explicit transition defined by an origin state vector and a destination state vector. Initially, the destination state vector is the same as the origin state vector. During the scanning of the system, origin state vectors remain the same but destination state vectors are updated to consider the effect of the statements that compose the system design. In order to determine the whole set of explicit transitions that corresponds to a control state, an explicit transition is created per each origin explicit state. Each transition demands one scanning of the statements within the control state. At the end of each scanning, explicit states must be created for the origin and the destination state vectors and they must be linked together with the corresponding symbolic transition. The explicit state-graph must be updated with those transitions. Below, we show the general procedure used for scanning control states.

```
01 for each IF control state do
02   begin
03     Q_e ← origin explicit states
04     t_s ← initial symbolic transition
05     for each q_e ∈ Q_e do
06       for each transition of the control state do
07         begin
08           compute t_e = (q_e, q_e') ∈ T_e
09           update t_s
10           link t_e and t_s together
11         end
```

```
12   end
```

Note that the algorithm requires several computations of the symbolic transition, one computation per each explicit transition. Because the execution of IF transitions must be considered to occur non-deterministically and all computations have to be considered, each IF transition demands the creation of individual explicit and symbolic transitions. Before processing IF transitions, the algorithms update both explicit and symbolic transitions with the state counter information, that determines the control state being modeled. For each IF transition, we link the final symbolic transition together with the explicit transition.

Regarding the several computations of the same symbolic transition, there is an overhead inherent to the creation of the combined model. Fortunately, the requisition for the computation of the same symbolic transition does not increase the memory costs, because symbolic algorithms automatically remove redundant subgraphs of the resulting BDD. There is just one representation of subgraphs produced several times. Regarding the executing time, the overhead is reduced due to the cache of already computed BDDs, supplied by the BDD library. It keeps a cache of outcome nodes associated with recent computations. Besides, we implement a cache of recently used BDDs, useful to delay the release of BDDs and to reduce the time involved in computing BDDs.

7.5.1 Initial States

The set of initial states defines the initial configuration of a system. Rigorously, all the variables have random values in the initial configuration of any system and all states are reachable. However, when modeling an IF system the set of assignments found within the initial control states are intended to restrict the set of practical initial states to those we reach after those assignments. The set of practical initial states may be artificially produced by considering the set of initial states as the set of states reached by executing exactly one transition from the formal set of initial control states. Thus, the behaviors corresponding to configurations prior to the assignments found in the formal set of initial states are discarded. Consider the fragment below, where `start` is an initial control state:

```
01 process transmitter(1);
02 var t clock;
03 var b boolean;
03 var c boolean;
04 var m data;
05
06 state start #start ;
07     task b := false;
08       nextstate idle;
09 endstate;
```

The practical initial states only include configurations where `b` is false, due to the assignment at line 07. Otherwise, the modeled system would include configurations where `b` is true, which is not intended by the system description. However, notice that variables `t`, `c` and `m` still can assume random values, because they were not assigned any value at the initial control state. Summarizing, our checking algorithms consider the initial states as those we reach from the set of initial control states through just one transition.

The semantics of IF assumes that the overall behavior of a system is given by interleaving the execution of its processes asynchronously. Nevertheless, during the generation of the set of practical inital states we consider that the transitions starting from the set of initial control states are executed synchronously. It means that the execution considers that the model checker starts only after all the processes have been set up. The asynchronous execution takes place for the transitions starting from the set of practical initial states on.

7.5.2 State Counters

In our implementation, the state counter is represented by an integer variable which assumes values from zero to $n-1$, where n stands for the number of states in the process. During the construction of the explicit and symbolic models, the state counters are used to determine the linkage of transitions. In a general way, given transitions $(q_1, q_1^{'})$ and $(q_2, q_2^{'})$ they must be linked if and only if the state counter of $q_1^{'}$ corresponds to the state counter of q_2, as depicted in figure 7.2.

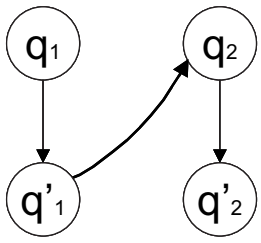


Figure 7.2: Linkage of states.

The construction of models depends on the state counter information. Without the state counter information the model created would potentially present erroneous transitions and erroneous reachable states. The absence of criterion for the linkage of states would produce fake execution paths and, consequently, it would be possible to reach states that would not be reached by using only valid transitions. Because symbolic transitions are associated with explicit transitions, we represent the state counters on the explicit model and the symbolic control flow is given as a consequence. Also, the control flow on the explicit-symbolic model is determined by the control flow of the explicit model.

7.5.3 Activation of Processes

According to the IF semantics, processes of a system are asynchronously executed. It means that there exists one and only one process active at each moment. Because all processes have the same chance of being executed, at the end of each transition all combinations for activation of processes are considered. For each process, the combined explicit-symbolic checker employs a Boolean variable to represent its activation information, its enabling variable x_e. The overall set of enabling variables determines which process is currently active on the system. We compute process activations according to the following expression:

$$\bigvee_{i=0}^{n-1}(x'_{e_i} \wedge \bigwedge_{j=0,j\neq i}^{n-1} \neg x'_{e_j})$$

where we assume that there exist n processes and consequently n enabling variables on the system. It does not matter how the enabling variables have been partitioned into the explicit and symbolic sets of representations. Explicit enabling variables trigger updates on the state vector. Symbolic enabling variables have their corresponding Boolean formulas conjuncted with the symbolic transition relation.

7.6 Signal Queues

In this section, we present the implementation of the signal queues of IF processes. Each process has an input queue which is shared by all the signals sent to that process. Only the signal at the head of the queue can be read. In order to keep track of the number of signals stored in the queue and for distinguishing the signal stored in each position of the queue, we have the following queue control variables:

1. Each process P has a queue of input signals $P.Q$, where n is the length of $P.Q$. Every queue position $P.Q(i)$, where $0 \leq i < n$, must be able to store different signals at different times. Because the signals sent to the process P can have different numbers of parameters, each one with a specific type, we must associate several different storages with each position $P.Q(i)$. For each position $P.Q(i)$, there exists a storage $P.Q(i).SignalX$ for each signal $SignalX$ received by process P. So, each position $P.Q(i)$ must have an integer field $P.Q(i).ID$ used to identify the signal stored at the current time. Every signal $SignalX$ has a unique integer identifier $SignalX.ID$.

2. There are integer variables $P.WP$ and $P.RP$ for controlling the writing and reading positions of the queue $P.Q$, respectively. The writing of $SignalX$ into $P.Q(i)$, indicated by $P.WP = i$, must assign the signal identifier $SignalX.ID$ to $P.Q(i).ID$. The reading of $SignalX$ from $P.Q(i)$, indicated by $P.RP = i$, requires the signal identifier $P.Q(i).ID$ to be equal to $SignalX.ID$. $P.RP$ indicates the queue head. A Boolean variable $P.EP$ is used to keep track of the emptiness of the queue of P.

According to the IF semantics, the reading of $SignalX$ from $P.Q$ is successful only if $P.RP = i$ and $P.Q(i).ID = SignalX.ID$. In figure 7.3, each queue variable is represented by a rectangle. Within each rectangle, we show the type of the variable. Empty rectangles represent the cases where the type depends on the stored signal.

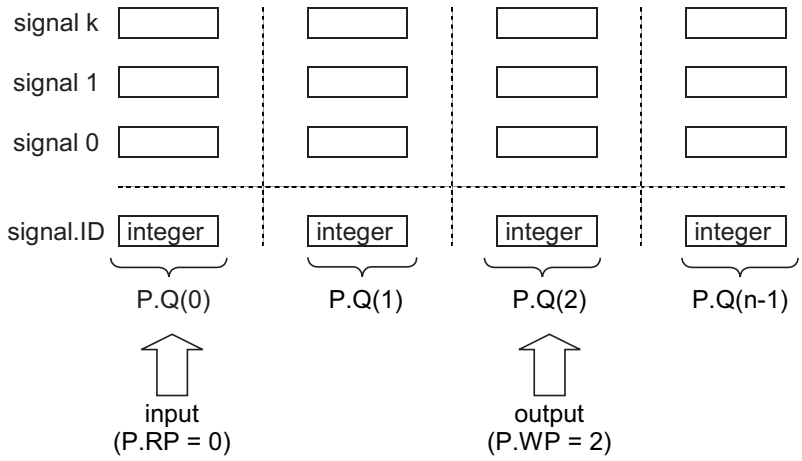


Figure 7.3: Signal queue modeling.

Remember that write operations into the queues correspond to signal outputs in the IF descriptions, while readings correspond to signal inputs. Note that the user processes can interact with the environment, sending and receiving signals to and from the environment. Because we cannot control the environment, in our implementation the queue for the environment process has just one position and we do not need to store writing, reading and emptiness queue variables. All the other processes are controled by the next algorithms. Initially the queues are empty and the writing and reading positions are zero.

```
01 input (signal X)
02 begin
03   if (not(P.EP))
04     begin
05       if(P.Q[P.RP].ID == X.ID)
06         begin
07           read(P.Q[P.RP].X)
08           P.RP ← (P.RP + 1) mod n
09           P.EP ← (P.RP == P.WP)
10         end
11       else
12         P gets blocked
```

```
13     end
14   else
15     P gets blocked
16 end
```

The algorithm above controls queue inputs. First, it checks if the queue of interest $P.Q$ is empty, at line 03. If $P.Q$ is empty, P gets blocked. Otherwise, the algorithm compares the identifier of the signal stored at the head of the queue and the identifier of the signal to be read, at line 05. In the case where the signal to be read is found at the head of the queue, the reading proceeds and the variables that control the reading position and queue emptiness are updated, by using lines from 06 to 10. Otherwise, the process gets blocked.

```
01 output (signal X)
02 begin
03   if (P.EP || (P.RP ≠ P.WP))
04     begin
05       P.Q[P.WP].ID ← X.ID
06       write(P.Q[P.WP].X)
07       P.WP ← (P.WP + 1) mod n
08       P.EP ← false
09     end
10   else
11     overflow
12 end
```

The previous algorithm controls the output of signals, that is, the writing of signals into queues. Line 03 makes sure that there are free positions in the queue, by checking that the queue is either empty or that there are free positions to be written yet, which means that the queue is neither empty nor full. When there are available positions in the queue, the writing proceeds and the writing and emptiness variables are updated, by using lines from 04 to 09. Note that when the queue is full the emptiness variable is false and both writing and reading positions are zero, due to lines 08 and 07 in the `input` and `output` algorithms, respectively. In this case, the queue suffers an overflow.

Blocking and overflow conditions are modeled at the same manner in our implementation. Their occurrence forces the algorithm to freeze the values of all variables in the model at that point of execution, except for `clock` variables. `clock` variables are incremented by one time unit, in order to model that time elapses while the process is waiting for some condition to be true. Such conditions may be the presence of the signal required by an `input` statement or the availability of free positions for the writing of a signal by an `output` statement. Dummy transitions $(\overline{X}, \overline{X'})$ are used for that purpose, such that:

$$\forall i, 0 \leq i < |\overline{X}|, \texttt{ if type}(x_i)\texttt{ is clock then } x'_i = x_i + 1$$
$$\texttt{otherwise } x'_i = x_i$$

Writing, reading and emptiness variables are automatically included for IF descriptions with communication through signals. The length of the queue of each process is heuristically set as the number of outputs found to that process. Notice that the queue modeling requires a large number of variables, what increases the complexity of the model. So, we also offer the option of employing user-defined queue lengths by means of the configuration file. Our algorithms use the minimal number of bits required to represent a particular set of signal identifiers. The number of bits used for representing the integer identifiers of each queue is computed as the nearest integer greater than or equal to $\log_2^m$, where m stands for the number of distinct signals sent to the queue of interest.

7.7 Cache of Explicit States

The main problem with explicit model checking algorithms is the storage space required to record already visited states during the generation of states. This decision determines the tractability of the problem to be checked when verifying large designs. Traditionally the set of states is represented as a large hash table of states. When a new state is generated it is hashed to obtain the index into the table. Since the states must be stored in its entirety, in order to allow for comparisons during the resolving of possible hash conflicts, this method is not very efficient when a large number of states must be stored.

In our project, the set of explicit states is represented by means of a tree structure, called cache of explicit states from now on, due to its behavior and benefits. The implementation of the cache system is intended to reduce the access time. Before the creation of an explicit state, its corresponding state vector is used to trigger a cache query in order to determine the previous existence of such an explicit state. If the state has been previously created, the cache returns a pointer to it. Otherwise, the creation of the state proceeds and the cache must be updated. Consequently, the cache system avoids the occurence of several representations of the same state vector, which reduces time and memory costs. Unique representations of state vectors also allow us to use simpler model checking algorithms, because each state vector correspond exactly to one explicit state.

Cache queries are based on raw state vectors, that is, non-compressed state vectors where each variable is represented as a decimal value. Concerning the cache system, non-compressed state vectors are much more advantageous when compared to compressed state vector, where variables have binary encoding, due to following reasons:

- Reduced number of state vector compressions.
- Short retrieval time.
- Memory saving.

Our approach reduces the number of compressions because only the state vectors corresponding to explicit states not found within the cache are converted to binary versions, during the state creation. Queries on the cache take a constant time, given by the number of explicit variables represented in the state vectors. Given n explicit variables, each query requires at most n comparisons before returning an explicit state or signalizing a cache fault. A cache fault occurs when the require state is not found in the cache.

Each level in the cache corresponds to an explicit variable. The $n-th$ level corresponds to the $n-th$ variable in the state vector. A level may have several nodes, where each cache node corresponds to the array of values that can be assigned to the variable represented on that level. Starting from the root node, queries looks for the next pointer leading to the required explicit state at the $i-th$ position of the relevant node, where i is the value of the variable represented on that level and the relevant node is the one belonging to the chaining of pointers. During the search, if a null pointer is found the cache system indicates a fault and demands the creation of the required explicit state. After that, the cache is updated with the additional explicit state. See the figure 7.4 below, where we show a generic cache system with n levels. The arrows shown correspond to the chaining of pointers leading to the explicit state given by the state vector 100...1. In the figure, m_n stands for the number of possible values for the variable represented on level n.

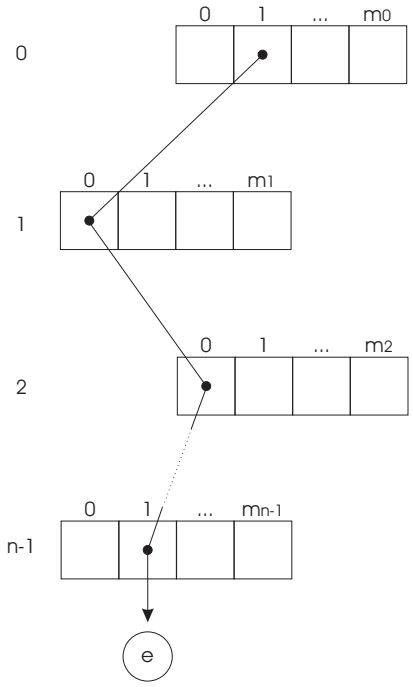


Figure 7.4: Cache of explicit states.

Below, we show that caches where queries are based on non-compressed state vectors are also more efficient in terms of memory. In the worst case, caches based on non-compressed

state vectors require

$$\sum_{i=0}^{n-1}(\prod_{j=0}^{i} 2^{w_j}) \tag{7.1}$$

cache pointers, where w_j stands for the number of bits required by the representation of the $j-th$ variable in the state vector. On the other hand, caches where queries are triggered by compressed state vectors would require

$$\sum_{i=1}^{\sum_{j=0}^{n-1} w_j} 2^i \tag{7.2}$$

cache pointers. Note that in both approaches each cache pointer requires only the storage relative to the pointer, since the variable value is given by the node index. From the expansion of equation 7.1 we have that:

$$\sum_{i=0}^{n-1}(\prod_{j=0}^{i} 2^{w_j}) = \prod_{j=0}^{0} 2^{w_j} + \prod_{j=0}^{1} 2^{w_j} + \cdots + \prod_{j=0}^{n-1} 2^{w_j}$$

$$\sum_{i=0}^{n-1}(\prod_{j=0}^{i} 2^{w_j}) = 2^{w_0} + 2^{w_0+w_1} + \cdots + 2^{w_0+w_1+\cdots+w_{n-1}} \tag{7.3}$$

Regarding caches based on compressed state vectors, equation 7.2, we have that:

$$\sum_{i=1}^{\sum_{j=0}^{n-1} w_j} 2^i = 2^1 + 2^2 + \cdots + 2^{w_0} + \cdots + 2^{w_0+w_1} + \cdots + 2^{\sum_{j=0}^{n-1} w_j}$$

$$\sum_{i=1}^{\sum_{j=0}^{n-1} w_j} 2^i = 2^1 + 2^2 + \cdots + 2^{w_0} + \cdots + 2^{w_0+w_1} + \cdots + 2^{w_0+w_1+\cdots+w_{n-1}} \tag{7.4}$$

By comparing equations 7.3 and 7.4, we conclude that each term of equation 7.3 is also a term of equation 7.4, and for each pair of terms 2^{w_j} and $2^{w_{j+1}}$ found within the equation 7.3 $\sum_{i=w_j+1}^{w_j+w_{j+1}-1} 2^i$ additional cache pointers are required by equation 7.4. So, we have that:

$$\sum_{i=0}^{n-1}(\prod_{j=0}^{i} 2^{w_j}) \ll \sum_{i=1}^{\sum_{j=0}^{n-1} w_j} 2^i$$

The cache of explicit states closely relates to the trie data structure. The trie is a tree that can be used to efficiently retrieve strings of symbols in a large text, being mainly concerned with information retrieval systems. In a trie, strings are represented along the paths of the tree. Each edge is labeled with a string and each node corresponds to the concatenation of the strings found along the path from the root to the current node being visited. For more information on the trie data structure, please refer to [Larsson, 1999].

Chapter 8

Experiments and Final Remarks

In this chapter we present some experimental results of the combined explicit-symbolic model. Based on those results, we discuss the strengths and weaknesses of our work. After that, we suggest future improvements to the current work and present some final remarks.

8.1 Experiments

Our experiments have been made on an AMD Athlon XP 2000+, 1,67GHz, with 256MB of RAM, running Mandrake Linux 9.1. The system has been generated by using GCC 3.2.2, flex version 2.5.4 and yacc 4.3 Berkeley distribution. Three designs have been used for experimenting with different partitioning of variables into the explicit and symbolic models. The first design is that of the microwave oven used for illustrating our discussion along this document. The second design is a converter from binary to decimal numbers. The third design represents a calculator which receives some variables and one constant and updates such variables accordingly to a computation mainly based on multipliers. Those designs have been verified by encoding integer numbers into eight-bits binary versions. We have measured the overall time involved in the verification, since the compilation of symbols until the output of results, the time spent during the checking of properties itself, and time involved with compressions and decompressions between state vectors and their compressed versions. Table 8.1 below summarizes the results. Time is given in seconds and considers the processor time. Each design has been verified first representing all variables in the explicit model, next in the symbolic model and, finally by combining variables in both models.

For the microwave system, the CTL specification checked has been $\texttt{EX}(\neg close \vee error)$. During the verification of the combined model we have partitioned the whole set of variables into two balanced subsets. The results have shown to be the same in spite of changing the representations of the variables of the microwave description between the underlying models. Note that the non-combined explicit and symbolic models present the same verification time. Although our construction algorithms for explicit models do not present static

analysis in order to avoid or reduce the generation of unreachable states, the construction of the model has not been affected due to the small size of the design. On the other hand, the combination of explicit and symbolic variables has imposed the overhead of scanning the design several times, one scanning per explicit transition. Two costs must be taken into account with regard to this overhead, the time for successive scannings of the design and the time for computing the symbolic transition over and over again. Also due to the size of the design, the checking time and the time used for compressing state vectors into the cache of explicit states and decompressing them during cache queries has not been meaningful.

Microwave Oven	***Overall Time***	***Checking Time***	***Compression Time***
Explicit Model	0.01	0.00	0.00
Symbolic Model	0.01	0.00	-
Combined Model	0.03	0.00	0.00
Binary/Decimal	***Overall Time***	***Checking Time***	***Compression Time***
Explicit Model	33.24	0.00	0.63
Symbolic Model	0.05	0.00	-
Combined Model	65.85	0.00	0.05
Calculator	***Overall Time***	***Checking Time***	***Compression Time***
Explicit Model	>2,100.00	-	-
Symbolic Model	611.44	0.00	-
Combined Model	50.26	0.00	0.00

Table 8.1: Experimental Results.

Regarding the binary to decimal converter, there is a much higher number of states when compared to the microwave design. The model has around 500 reachable states, from the total of 655,000 states. That is the reason why the non-combined explicit model takes about 33 seconds of overall time. The time spent with state vector conversions is around 1.9 percent of the global time. On the other hand, the symbolic verification takes only 0.05 seconds to perform all the work involved with the verification. Symbolic representations naturally take advantage of regularities in the search space to achieve compact representations. Also, symbolic algorithms can handle set of states at a time. However, note that efficient techniques to reduce the set of explicit states generated would greatly improve our explicit results. Only 0.08 percent of the total explicit states handled are reachable. So, the overall time proportional to the set of reachable states would be only 0.0266, less than the time required by the non-combined symbolic model. Now, let us consider the overall time required by the combined model. It takes almost 66 seconds, being bigger that the non-combined models. When we compare the overall time with the checking and state vector compression times, it is easy to note that the overhead is due to the construction of the model. Each explicit state, including the unreachable ones, requires a scanning of the system and the computation of the symbolic transition. Consequently,

the construction of the combined model requires the time for computing its subset of the explicit states and the time for computing a symbolic transition for each explicit transition being handled. Compared to the non-combined explicit model, the time for compressing and decompressing the state vector is reduced in the combined model because there is a smaller number of explicit symbols to represent in each state vector.

The Calculator Example

The calculator receives four input variables a, b, c, d and one constant e, and updates such variables accordingly to a computation mainly based on multipliers. In our experiments, the specification of interest is $EX(a = 0 \wedge b = 0)$. We consider that the system is initialized so that a, b and e may assume any value represented by eight bits, but the values of c and d do not change along computations. For the first experiment, all the variables in the system were assigned symbolic representations. About ten minutes were spent for constructing the state diagram. Usually, symbolic representations are not well suited for dealing with data transformations and, specially, multipliers. Due to the nature of symbolic representations, all the variables involved with the multiplication are represented by the same Boolean function and the ordering of the arguments of this function affects the size of the state graph. Unfortunately, the size of the graph representing multiplication grows exponentially with the number of variables being operated, it does not matter the order of variables. The bigger the number of such variables, the bigger the size of the corresponding graph. Traditionally, the verification of systems focused on data transformations tends to be more efficient when those systems are given by means of explicit representations.

In the second experiment, all the variables have been assigned explicit representations. Because a, b and e can assume any value from 0 to 2^8-1, there are at least 2^{24} explicit states to be checked. Because every state needs to be checked individually, the construction of the explicit state graph becomes very expensive. After thirty-five minutes, the explicit model was not completely constructed yet. Generally, explicit techniques reduce the number of visited states by using partial order reduction and static analysis, as discussed in section 3.4. However, those techniques would not be useful for reducing the number of visited explicit states when considering variables a, b and e, because all their possible values must be taken into account.

Finally, we have experimented with the combined explicit-symbolic model. During the partitioning of the set of variables, we were concerned with balancing the variables into explicit and symbolic subsets in order to reduce the inherent cost of the symbolic multiplication, by reducing the number of variables involved, without having the overhead of dealing with at least 2^{24} states on the explicit model. Removing variables from the symbolic model tends to minimize the graph size because we reduce the number of restrictions to the variable ordering and consequently smaller BDDs can be generated. On the other hand, variables with wide ranges increase the number of explicit states. So, the partitioning criterion used has been the range of values assumed by variables. We have moved variables c and d from the symbolic model to the explicit one. Such a partitioning minimizes the

complexity of the symbolic model without the need of handling a huge number of explicit states. That is possible because we assume that variables c and d have just one value to be considered along the execution time. By using such a partitioning, the combined model takes only 8 percent of the time required to check the same specification on the symbolic model. Note that the combined model is even more efficient when compared to the explicit model. Such an improvement is obtained because the explicit-symbolic model can combine techniques of different representations to exploit each modeled component according to the nature of the involved variables.

8.2 Analysis of Results

The experiments show that the choice of representations has significant impact on the efficiency of model checking techniques and they reinforce the need of optimizing techniques in order to make explicit model checking competitive. On the other hand, when considering our implementation as a prototype devoted to prove the viability of combining explicit and symbolic representations together, our work fulfills its goals. It is now clear that explicit and symbolic representations can be integrated in a general model-checking framework to face the problem of verifying designs. Even more, the experiments show that the combined explicit-symbolic model can yield better results than explicit and symbolic techniques for some problems. Another important benefit of this work is the fact that our modeling is general and can support optimization techniques for both underlying models. Our implementation has been conceived having such a generality as a basis. Consequently, several optimization techniques can be used to make the combined model checker much more competitive.

The process of constructing the combined model offers many opportunities with regard to efficiency improvements. Particularly, instead of generating the same symbolic transition several times, the construction process could be accomplished in two different steps. During the first step, all the explicit transitions would be generated, but without associations with symbolic transitions. After the construction of the explicit model, just another scanning of the design would be necessary to associate symbolic transitions with explicit transitions. In order to guarantee the synchronization of the models, this last scanning would generate appropriate descriptors for accessing the previously created explicit transitions before associating symbolic transitions with them. Obviously, the usage of aggressive static analysis techniques is still prominent.

8.3 Future Works

Other techniques can also be incorporated in the combined explicit-symbolic framework to confer more efficiency and functionality. Here, we cite some suggestions:

- Rewriting of specifications - As many subformulas as possible should be checked in each underlying model before requiring a synchronization step with the counterpart model. So, automated techniques for generating equivalent specification where explicit and symbolic propositions are grouped in subformulas can benefit the performance of the model.

- Dynamic change of representations - Currently, the representations of variables are chosen before the construction of models. It would be more flexible to be able to change the partitioning of variables without having to construct the models again. Such flexibility could be used to fine tune the combined model and improve its performance.

- Dynamic conversion of values between explicit and symbolic variables - This facility would make possible to choose different representations to variables that interact with each other via signal sending, expressions or assignments. It means more flexibility to choose variable representations.

- Application of techniques to increase the efficiency of the explicit model, such as partial order reduction and static analysis.

8.4 Final Remarks

In this work, we proposed a model that combines explicit and symbolic representations. The conceived explicit-symbolic model considers that explicit and symbolic techniques should be used in an integrated and synchronized fashion, allowing us to have a better exploration of the search space of the modeled system. Thus, our main contributions are the proposal of a flexible environment for the formal verification of systems and the computational implementation of the explicit-symbolic model and its algorithms, considering systems specified in the Interchange Format. The implementation has shown that explicit and symbolic representations can be integrated in a general model-checking framework to face the problem of verifying designs. In addition, the adoption of the Interchange Format offers another description language for explicit, symbolic and explicit-symbolic model checking. New explicit techniques have been implemented, one for compressing the state vector and another for storing the set of visited states.

Also, this work discusses important aspects concerned with the partitioning of a system and the composition and synchronization of its underlying models. So, it can help investigating the integration of other formal verification techniques and representations.

Finally, the combined explicit-symbolic model promotes the comparison between the solutions and techniques offered by the explicit and symbolic approaches during the different stages of model checking a system. It is important to measure the impact that different representations have over the verification of different systems and choose the approach more competitive and efficient to each problem being checked.

Bibliography

[Andrews, 1986] Andrews, P. B. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press Professional, Inc, San Diego, CA, USA.

[Ball et al., 2001] Ball, T., Majumdar, R., Millstein, T. D., and Rajamani, S. K. (2001). Automatic Predicate Abstraction of C Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, volume 36, pages 203–213.

[Ball and Rajamani, 2000] Ball, T. and Rajamani, S. K. (2000). Bebop: A Symbolic Model Checker for Boolean Programs. In *7th International SPIN Workshop on Model Checking of Software*, pages 113–130, Stanford University, California, USA.

[Ball and Rajamani, 2002] Ball, T. and Rajamani, S. K. (2002). The SLAM Project: Debugging System Software via Static Analysis. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, USA.

[Bause et al., 1998] Bause, F., Buchholz, P., and Kemper, P. (1998). A Toolbox for Functional and Quantitative Analysis of DEDS. 1469:356–359.

[Berezin, 2002] Berezin, S. (2002). *Model Checking and Theorem Proving: a Unified Framework*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

[Bollig et al., 1996] Bollig, B. et al. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002.

[Bozga et al., 1999] Bozga, M., Fernandez, J.-C., Ghirvu, L., Graf, S., Krimm, J.-P., and Mounier, L. (1999). IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In Wing, J. M., Woodcock, J., and Davies, J., editors, *FM'99—Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 307–327, Toulouse, France. Springer.

[Brace et al., 1990] Brace, K., Rudell, R., and Bryant, R. (1990). Efficient implementation of a BDD package. In *27th ACM/IEEEE Design Automation Conference*, pages 40–45.

[Bryant, 1986] Bryant, R. E. (1986). Graph-Based Algoritms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

[Büchi, 1960] Büchi, J. R. (1960). On a Decision Method in Restricted Second Order Arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, Stanford, CA, USA. Stanford University Press.

[Campos, 1996] Campos, S. V. A. (1996). *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

[Chan, 1999] Chan, W. (1999). *Symbolic Model Checking for Large Software Specifications*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Washington, DC, EUA.

[Chan et al., 1997] Chan, W., Anderson, R., Beame, P., and Notkin, D. (1997). Combining Constraint Solving and Symbolic Model Checking for a Class of a Systems with Nonlinear Constraints. In *Computer Aided Verification*, pages 316–327.

[Cheung and Kramer, 1999] Cheung, S. C. and Kramer, J. (1999). Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(1):49–78.

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68.

[Cimatti et al., 2000] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (2000). NUSMV: A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425.

[Clarke et al.,] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[Clarke et al., 1996] Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J., and Zave, P. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643.

[Corbett et al., 2000] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. (2000). Bandera: extracting finite-state models from Java

source code. In *22nd International Conference on Software Engineering, ICSE*, pages 439–448.

[Cormen et al., 1990] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts.

[Dillinger and Manolios, 2004] Dillinger, P. C. and Manolios, P. (2004). Fast and Accurate Bitstate Verification for SPIN. In *11th International SPIN Workshop on Model Checking of Software*, pages 57–75, Barcelona, Spain.

[E. Allen Emerson, 1990] E. Allen Emerson (1990). *Temporal and Modal Logic*, volume B: Formal Models and Semantics of *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, Amsterdam.

[E.M. Clarke and E.A. Emerson, 1981] E.M. Clarke and E.A. Emerson (1981). Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York. Springer-Verlag.

[Emerson and Clarke, 1980] Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In de Bakker, J. W. and van Leeuwen, J., editors, *7th International Colloquium on Automata, Languages and Programming, ICALP'80 (Noordwijkerhout, NL, July 14-18, 1980)*, volume 85 of *Lecture Notes Computer Science*, pages 169–181. Springer-Verlag, Berlin.

[Gordon and Melham, 1993] Gordon, M. J. C. and Melham, T. F., editors (1993). *Introduction to HOL: A Theorem Proving Environment*. Cambridge University Press.

[Holzmann and Peled, 1994] Holzmann, G. and Peled, D. (1994). An Improvement in Formal Verification. In *Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland. Chapman & Hall.

[Holzmann, 1995] Holzmann, G. J. (1995). An Analysis of Bitstate Hashing. In *15th International Conference on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland. Chapman & Hall.

[Holzmann, 1997a] Holzmann, G. J. (1997a). State Compression in SPIN: Recursive Indexing and Compression Training Runs. In Langerak, R., editor, *3rd International SPIN Workshop*, Twente University, Enschede, The Netherlands.

[Holzmann, 1997b] Holzmann, G. J. (1997b). The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[Holzmann et al., 1996] Holzmann, G. J., Peled, D., and Yannakakis, M. (1996). On Nested Depth First Search. In *2nd SPIN Workshop*, pages 23–32, Rutgers, Piscataway, NJ, USA. American Mathematical Society.

[Holzmann and Puri, 1999] Holzmann, G. J. and Puri, A. (1999). A Minimized Automaton Representation of Reachable States. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):270–278.

[Huth and Ryan, 2000] Huth, M. R. A. and Ryan, M. D. (2000). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK.

[K.L. McMillan, 1992] K.L. McMillan (1992). The SMV system. Technical Report CMU-CS-92-131, Pittsburgh, EUA.

[Larsson, 1999] Larsson, N. J. (1999). *Structures of String Matching and Data Compression*. PhD thesis, Department of Computer Science, Lund University, Sweden.

[McMillan, 1992] McMillan, K. L. (1992). *Symbolic Model Checking: An Approach to the State Explosion Problem*. Phd thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[McMillan, 1993] McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts.

[Milner et al., 1997] Milner, R., Tofte, M., and MacQueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[Musuvathi, 2004] Musuvathi, M. (2004). *CMC: a model checker for network protocol implementations*. Phd thesis, Department of Computer Science, Stanford University, Stanford, EUA.

[Owre et al., 1998] Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. (1998). PVS: An Experience Report. In Hutter, D., Stephan, W., Traverso, P., and Ullman, M., editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany. Springer-Verlag.

[Owre et al., 1992] Owre, S., Rushby, J. M., , and Shankar, N. (1992). PVS: A Prototype Verification System. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY. Springer-Verlag.

[Partridge, 1996] Partridge, K. E. (1996). BDDTCL: An Environment for Visualizing and Manipulating Binary Decisions Diagrams. In *ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 2, pages 111–112.

[Prior, 1957] Prior, A. N. (1957). *Time and Modality*. Oxford University Press, Oxford.

[Prior, 1967] Prior, A. N. (1967). *Past, Present and Future*. Oxford University Press, Oxford.

[Santiago, 2003] Santiago, J. S. (2003). Particionamento Simbólico de Programas. Master's thesis, Departamento de Ciência da Computação - DCC, Instituto de Ciências Exatas - ICEx, Belo Horizonte, Brazil.

[Sebastiani, 2001] Sebastiani, R. (2001). Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST.

[Shannon, 1938] Shannon, C. E. (1938). A Symbolic Analysis of Relay and Switching Circuits. *Transactions of the AIEE*, 57:713–723.

[Spivey, 1992] Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition.

[Tarjan, 1972] Tarjan, R. E. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160.

[Tarski, 1955] Tarski, A. (1955). A Lattice-Theoretic Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*, 5(2):285–309.

[Varpaaniemi et al., 1997] Varpaaniemi, K., Heljanko, K., and Lilius, J. (1997). PROD 3.2: An Advanced Tool for Efficient Reachability Analysis. In Grumberg, O., editor, *Computer Aided Verification: 9th International Conference, CAV'97, Haifa, Israel, June 22–25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 472–475. Springer-Verlag, Berlin, Germany.

[Visser et al., 2000] Visser, W., Havelund, K., Brat, G., and Park, S. (2000). Java PathFinder - Second Generation of a Java Model Checker. In *Workshop on Advances in Verification - WAVE*, Chicago, USA.

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts.