Especificação e Verificação Sistemática, Formal e Modular de Sistemas Embarcados

Leandro Dias da Silva

Tese de Doutorado submetida ao Programa de Pós-Gradução em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Angelo Perkusich, D.Sc.
Orientador

Campina Grande, Paraíba, Brasil

©Leandro Dias da Silva, Abril de 2006

UFCG - BIBLIOTECA - CAMPUS I
3769 03-08-06

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586e Silva, Leandro Dias da
2006 Especificação e verificação sistemática, formal e modular de sistemas embarcados/ Leandro Dias da Silva. — Campina Grande, 2006.
89f. il.

Inclui bibliografia.
Tese (Doutorado em Engenharia Elétrica) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Dr. Ângelo Perkusich.

1— Redes de Petri - Desenvolvimento Baseado em Componentes 2— Verificação de Sistemas Embarcados 3— Desenvolvimento Baseado em Componentes I— Título

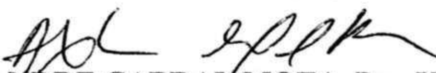
CDU 681.3.02

ESPECIFICAÇÃO E VERIFICAÇÃO SISTEMÁTICA, FORMAL E MODULAR DE SISTEMAS EMBARCADOS

LEANDRO DIAS DA SILVA

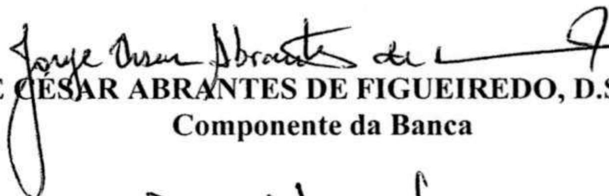
Tese Aprovada em 28.04.2006

ANGELO PERKUSICH, D.Sc., UFCG
Orientador



ALEXANDRE CABRAL MOTA, Dr., UFPE
Componente da Banca

VICENTE FERREIRA DE LUCENA JÚNIOR, Dr., UFAM
Componente da Banca



JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc., UFCG
Componente da Banca

DALTON DARIO SEREY GUERRERO, D.Sc., UFCG
Componente da Banca

CAMPINA GRANDE – PB
ABRIL - 2006

Agradecimentos

Gostaria de agradecer ao meu orientador, Angelo Perkusich, pelo apoio no desenvolvimento deste trabalho, tanto a nível técnico quanto pessoal, inclusive com uma relação de amizade que se desenvolveu ao longo destes anos que foi de fundamental importância.

Aos meus pais, Luiz e Marlene, e meus irmãos, Luiz, Lilian e Luciane, por tudo que fizeram e têm feito por mim, por acreditar no meu potencial e me apoiar.

Aos meus amigos, Elizeu e Kyller, que mesmo distantes fisicamente em parte do tempo, sempre se mostraram presentes e foram fundamentais nos momentos mais difícies.

A todos os amigos e companheiros de laboratório, prédio e outras instituições durante a maior parte desta jornada, pelos momentos de ajuda técnica, sinergia de idéias, inclusive nos momentos de descontração menos científicos.

Aos professores e funcionários do DEE, em especial Angela por todo apoio e ajuda.

À CAPES e ao CNPq pelo apoio financeiro.

Resumo

Este trabalho está inserido no contexto de engenharia de software baseada em componentes para o domínio de sistemas embarcados, no nível de modelagem. A fase de modelagem é importante pois permite um melhor entendimento de um dado problema e de suas possíveis soluções, além de facilitar a manutenção e a evolução do sistema. Redes de Petri Coloridas Hierárquicas (HCPN - *Hierarchical Coloured Petri Nets*) são utilizadas para a especificação e verificação formal de tais sistemas. A utilização de HCPN na modelagem de sistemas complexos promove a descrição de modelos de forma compacta e organizada.

A utilização de componentes promove a modularização da construção do espaço de estados, utilizando descrições de suas interfaces. Para tanto, utilizamos o conceito de grafos de ocorrência com classes de equivalência para definir a interação entre os componentes do sistema, ignorando seus comportamentos internos. Além de classes de equivalência, é utilizado o formalismo Autômato com Interface Temporizado (TIA - *Timed Interface Automata*) para expressar as interfaces dos componentes modelados com HCPN para análise de compatibilidade.

Abstract

This work is defined on the context of component based software engineering for the embedded systems domain. The focus is on the modelling level. The modelling phase is very important because it promotes a better understanding of the problem at hand and its possible solutions, and also an easier maintenance and evolving of a system. Specifically, *Hierarchical Coloured Petri Nets* (HCPN) are used for the formal specification and verification of such systems. The use of HCPN in the modelling of complex systems promotes a compact and organized model's description.

The use of components promotes a modular construction of the state space using descriptions of their interfaces. In order to achieve this, occurrence graphs with equivalence classes are used to define the interaction among the system's components, ignoring their internal behavior. Moreover, *Timed Interface Automata* (TIA) are used to express the interface of the components modelled with HCPN for compatibility analysis.

Conteúdo

1 Introdução **1**

1.1 O Problema 3

1.2 A solução 4

1.3 Organização da Proposta 6

2 Fundamentação Teórica **8**

2.1 Redes de Petri Coloridas Hierárquicas 8

2.2 Introdução à Verificação de Modelos 14

2.3 Teoria de Interfaces 17

2.4 Sistemas Embarcados 20

2.5 Explosão do Espaço de Estados 22

2.5.1 Verificação Simbólica de Modelos 23

2.5.2 Redução de Ordem Parcial 24

2.5.3 Raciocínio Modular 25

2.5.4 Abstração 26

2.6 Trabalhos Correlatos 27

3 Processo de Desenvolvimento Baseado em Componentes e Reúso de Modelos **29**

3.1 Linha de Produto, Componentes, e Arquitetura de Software 31

3.2 Ciclo de Vida do Desenvolvimento Baseado em Componentes 32

3.3 Modelagem Sistemática 35

3.3.1 Recuperação 37

3.3.2 Adaptação 38

3.4 Integração de Modelos 38

3.5 Verificação de Uso 40
3.6 Conclusões 42

4 Estudo de Caso: Rede de Transdutores 44

4.1 O Sistema de Controle de Redes de Transdutores 45
4.2 Especificação e Verificação 47
4.2.1 Especificação 47
4.2.2 Verificação 54
4.2.3 Considerações 61
4.3 Conclusões 61

5 Análise Modular de Modelos HCPN 63

5.1 Introdução 63
5.2 Grafos de Ocorrência com Classes de Equivalência Compatíveis 63
5.2.1 Prova do Fluxo de Lugar 67
5.2.2 Especificação de Equivalência Compatível 69
5.2.3 Prova da Especificação de Equivalência Compatível 71
5.3 Definição de um Autômato com Interface para um Modelo HCPN 72
5.4 Algoritmo para Extração do TIA de um HCPN 73
5.5 Estratégia Híbrida de Análise Modular 77
5.5.1 Considerações a Respeito da Abordagem Híbrida 77

6 Conclusões 79

6.1 Sugestões para Trabalhos Futuros 80

Bibliografia 82

Lista de Figuras

2.1 Exemplo de redes de Petri. 10

2.2 Rede de Petri colorida hierárquica. 15

2.3 TIA para detector de incêndio. 20

2.4 Representações: (a) explícita; (b) simbólica. 24

2.5 Raciocínio circular. 25

3.1 Ciclo de vida do desenvolvimento baseado em componentes. 34

3.2 Esquema para a solução de reúso de modelos CPN. 36

4.1 Topologia do sistema. 45

4.2 Hierarquia do modelo. 47

4.3 Página do modelo dos sensores e atuadores. 48

4.4 Página do modelo do servidor de comunicação. 49

4.5 Página do modelo do interpretador de entrada e saída. 50

4.6 Página do modelo do conversor de dados. 51

4.7 Página do modelo do controlador de dispositivos. 52

4.8 Página do modelo do sincronizador. 52

4.9 Página do modelo do servidor de tempo real. 53

4.10 Página do modelo do controlador de dados. 54

4.11 Página do modelo do módulo de interface com o usuário. 55

4.12 MSC para o fluxo do conversor de dados para controle. 56

4.13 MSC para o fluxo do conversor de dados para informação. 58

4.14 MSC para o fluxo no sincronizador. 59

5.1 Página do modelo do controlador de dados. 66

5.2 Interface para o modelo do controlador de dados. 67
5.3 Autômato com interface para um modelo simples. 72
5.4 Autômato com interface para o controlador de dados. 73

Capítulo 1

Introdução

Devido à constante busca pelo aperfeiçoamento da qualidade dos programas de computadores, de *software*, várias técnicas, métodos e ferramentas têm sido pesquisados e aplicados. Alguns exemplos são novos processos de desenvolvimento, linguagens de programação, e ferramentas CASE (*Computer-Aided Software Engineering*).

O conceito de qualidade pode englobar várias características do software, dependendo do contexto e domínio de aplicação. Exemplos de características de qualidade analisadas são consumo de energia e memória, serviço de comunicação para sistemas multimídia, controle de concorrência em bancos de dados em tempo-real, seqüência de ações ou protocolo de funcionamento. Esta última característica está relacionada com o funcionamento do sistema em si, ou seja, que ações devem preceder ou suceder certas ações sem que, por exemplo, ocorram impasses.

De forma geral, na disciplina de engenharia de software sempre há a preocupação com dois requisitos essenciais do projeto: custo e tempo de desenvolvimento. Estes geralmente vão de encontro com requisitos de qualidade, ou seja, quanto menor o orçamento e tempo disponível menor tende a ser a qualidade do software desenvolvido e vice-versa.

Sistemas críticos são sistemas em que uma falha pode ocasionar grandes perdas financeiras ou até mesmo de vidas humanas. Exemplos de sistemas críticos são controle de aviões e tráfego aéreo, controle de sistemas de manufatura, processos químicos, sistemas bancários. Neste trabalho um estudo de caso no contexto de sistemas embarcados é considerado. Geralmente sistemas embarcados são também sistemas críticos. Apesar da abordagem apresentada neste trabalho poder ser utilizada em sistemas críticos de forma geral, a definição de um domínio especifico é

importante para o sucesso do processo de reúso.

Para manter o orçamento e o tempo de desenvolvimento de sistemas críticos em níveis aceitáveis comercialmente, mantendo a qualidade, é necessário dispor de técnicas e métodos para sistematizar o processo e verificar propriedades que devem ser satisfeitas para seu correto funcionamento. A sistematização possibilita a redução de erros humanos e promove o desenvolvimento com economia de recursos de tempo e dinheiro. Além disso, facilita a manutenção e evolução do software. Métodos formais são ferramentas com base matemática, que em alguns casos possuem uma representação gráfica, utilizados para especificação e análise de sistemas. A base matemática promove algumas vantagens que incluem documentação precisa, simulação e prova de propriedades automáticas, antes que o sistema real seja desenvolvido.

O formalismo utilizado neste trabalho são as Redes de Petri Coloridas Hierárquicas (HCPN-*Hierarchical Coloured Petri Nets*) [50, 51], que são CPNs *Coloured Petri Nets*) com mecanismos de hierarquia. O conjunto de ferramentas Design/CPN [49] é utilizado para a edição e análise dos modelos HCPN. A utilização de HCPN promove a descrição de modelos mais compactos e organizados visto que estas incorporam conceitos de tipos de dados complexos e hierarquia.

Um processo de reúso de modelos formal, sistemático, organizado e controlado é utilizado neste trabalho [30, 23, 26, 58, 59, 43, 44, 18, 45]. Para aplicar o reúso, é utilizado um processo de desenvolvimento baseado em componentes [65, 17, 71]. Desta forma os modelos não precisam mais ser desenvolvidos sempre do zero, mas sim utilizando modelos de componentes já utilizados com sucesso em projetos anteriores.

A justificativa do domínio de aplicação, sistemas embarcados [55, 57], se deve ao fato de se tratar de aplicações críticas. Ao se desenvolver e usar sistemas críticos é desejável um alto grau de confiança no seu funcionamento. Desta forma, as dificuldades de desenvolvimento acrescentadas pelo uso de métodos formais são justificadas pelos benefícios providos.

Como no processo de reúso os modelos são construídos a partir de vários outros "montados" para formar um novo, é importante poder verificar a interação entre estes. Além disso, também é desejável poder verificar partes específicas do novo modelo. As técnicas para prova de propriedades aplicadas atualmente são prova automática de teoremas [5] e verificação de modelos [11]. A verificação de modelos para a "interface" dos componentes é utilizada para verificar a interação destes em um novo modelo, para analisar o comportamento emergente desta composição, e verificar se este comportamento é o esperado ou desejável para satisfazer os requisitos do projeto

atual. Além disso, é desejável dispor de técnicas de verificação modular pois, uma vez que estamos construindo um novo modelo de forma modular, também é desejável poder analisá-lo modularmente, evitando o problema de explosão do espaço de estados. Este problema pode ocorrer quando há muitos componentes concorrentes e distribuídos que se comunicam entre si, ou quando há manipulação de tipos de dados. Nesses casos, o espaço de estados do modelo pode ser muito grande.

Apesar dos mecanismos de hierarquia de HCPN, estes são puramente visuais, sem uma semântica definida que permita análise modular. Desta forma, ao modelar sistemas complexos com HCPN, a verificação de modelos sempre será realizada no espaço de estados de toda a rede. Logo, pode facilmente ocorrer o problema da explosão de espaço de estados citado anteriormente.

Nos últimos anos o uso de métodos formais no desenvolvimento de software tem crescido significativamente. Em particular, várias pesquisas têm sido desenvolvidas no contexto de teoria e aplicação de CPN [54, 53, 52, 8, 14, 73].

A modelagem pode ser considerada uma atividade chave em um processo de desenvolvimento pois permite um melhor entendimento do problema a ser solucionado bem como de possíveis soluções. Permite ainda uma melhor manutenção e evolução do sistema pois soluções e mudanças podem ser analisadas antes de serem inseridas no sistema real, reduzindo o custo e tempo de desenvolvimento. Por outro lado, a falta de modelagem, ou uma modelagem deficiente, pode comprometer o sistema. Uma abordagem sistemática promove a diminuição de erros humanos, enquanto que a modularização promove uma melhor organização, facilitando tanto a especificação quanto a análise.

1.1 O Problema

Pode-se definir o problema a ser abordado nesta tese da seguinte forma:

> "Aumentar a confiança no funcionamento de sistemas críticos, sistematizando o processo de desenvolvimento, e lidar com o problema de explosão de espaço de estados"

1.2 A solução

A solução para o problema apresentado anteriormente foi a definição de um processo de desenvolvimento baseado em componentes e reúso de modelos CPN [24, 27]. Este processo foi aplicado ao domínio de sistemas embarcados em [28].A sistematização é atingida através de um processo de reúso de modelos HCPN. O reúso de modelos de componentes promove um desenvolvimento mais rápido, além de aumentar a confiança no funcionamento do modelo pois é de se esperar que os modelos reutilizados tenham sido usados em projetos anteriores bem sucedidos, tendo sido, portanto, validados. É desejável também que a abordagem modular seja aplicada na análise dos modelos.

No processo definido, o principal problema a ser abordado foi a verificação. Além de considerar o reúso de modelos de componentes, também é desejável reduzir a possibilidade de explosão de espaço de estados. Para tratar este problema específico uma abordagem híbrida foi adotada. Uma possibilidade é a especificação das interfaces dos modelos de componentes para fazer verificação de compatibilidade. A outra possibilidade é utilizar as interfaces para verificar propriedades de vivacidade e limitação utilizando-se grafos de ocorrência com classes de equivalência. No contexto deste trabalho interfaces são o que os componentes esperam (suposições) do ambiente que o utiliza e o que ele garante (garantias) a este ambiente uma vez que as suposições foram estabelecidas. Grafo de ocorrência é a representação do espaço de estados. Esses conceitos serão detalhados quando os formalismos forem apresentados ao longo deste trabalho.

A especificação das interfaces dos modelos HCPN é feita com o uso de autômatos com interfaces temporizados (TIA - *Timed Interface Automata*) [34, 35]. Com este formalismo é possível especificar interfaces de componentes, inclusive propriedades temporais, e realizar verificação de compatibilidade. No contexto de TIA, duas interfaces são compatíveis se a composição é bem formada, isto é, se existe algum ambiente em que estas operam conjuntamente [34]. Uma ferramenta chamada TICC[1] foi desenvolvida baseada no conceito de Interfaces Sociáveis para a verificação de compatibilidade de interfaces [31].

O uso de HCPN para especificar os modelos de componentes e TIA para especificar suas interfaces se justifica pelos objetivos de aplicação de cada formalismo. Enquanto que HCPN é mais adequada para especificar o comportamento de sistemas complexos, TIA é mais adequado

[1] http://www.dvlab.net/dvlab/Ticc

para especificar interfaces.

Quanto mais abstrato o modelo, mais compacto ele é, e pode ser mais fácil analisá-lo e verificá-lo. Entretanto, deve-se observar que dependendo do nível de abstração pode-se perder informação. Tal perda pode levar à impossibilidade de provar algumas propriedades desejáveis. No desenvolvimento baseado em componentes, é importante modelar as funcionalidades dos componentes pois essa é a característica mais importante de um componente. Desta forma, um modelo abstrato pode não ser suficiente para garantir que propriedades um componente satisfaz.

Por outro lado, quanto mais detalhado o modelo mais completa é a modelagem das funcionalidades, validação do modelo e documentação. Entretanto, mais complexo é o seu entendimento e sua verificação. Por exemplo, quando se compõem vários autômatos através da composição por produto ou composição paralela pode-se ter um número exponencial de estados com relação aos estados de cada autômato. Isto leva facilmente à explosão de estados quando se modela sistemas complexos. Além disso, a manipulação de tipos complexos de dados também pode levar à explosão de estados.

Quando se utiliza conceitos de componentes, é importante ter confiança no funcionamento dos componentes, pois estes são utilizados de forma caixa-preta, ou seja, não se tem conhecimento de sua estrutura e funcionamento interno. Portanto, é de se esperar que modelos de componentes disponíveis para reúso tenham sido validados anteriormente. Uma validação baseada em modelos muito abstratos pode não garantir tal confiança. Por outro lado, quando se compõem componentes, não é importante considerar seu funcionamento interno. Ao se montar sistemas a partir de componentes, é mais importante analisar e verificar as interações, conexões, dos componentes. Além disso, como geralmente se utiliza uma arquitetura para conectar os componentes, a verificação desta arquitetura também é importante neste processo de desenvolvimento.

De acordo com o exposto anteriormente, utiliza-se HCPN para modelar os componentes e arquitetura, bem como as composições, e TIA para modelar as interfaces dos componentes. Desta forma, é realizada a modelagem e verificação de componentes com um formalismo que permite um maior nível de expressão, desejável para sistemas e funcionalidades complexas. E no caso dos novos modelos, baseados em um arcabouço HCPN e modelos de componentes reusados, se utiliza TIA para verificar as composições, interações e arquitetura.

Assim como em outros contextos da engenharia de software, não existe uma "bala de prata" para a especificação e verificação de sistemas. Então, utiliza-se dois formalismos diferentes, para

resolver problemas diferentes, dentro de um mesmo processo e estratégia de modelagem. Além disso, cada formalismo é utilizado onde melhor se adequa: HCPN para modelar e verificar entidades complexas e com manipulação de dados; e TIA para especificar e verificar as interfaces e interações de componentes.

1.3 Organização da Proposta

O restante deste trabalho está organizado da seguinte forma:

Capítulo 2: Fundamentação Teórica

Neste capítulo é apresentada a técnica de verificação de modelos e o formalismo redes de Petri coloridas hierárquicas, TIA, bem como as técnicas que têm sido empregadas para contornar o problema de explosão de espaço de estados. Além disso é apresentada uma contextualização de sistemas embarcados.

Capítulo 3: Processo de Desenvolvimento Baseado em Componentes e Reúso de Modelos

Neste capítulo são descritos o processo de reúso de modelos de redes de Petri coloridas e o processo de desenvolvimento baseado em componentes que guia o reúso.

Capítulo 4: Estudo de Caso: Rede de Transdutores

Um estudo de caso no domínio de sistemas embutidos é apresentado neste capítulo com o objetivo de ilustrar o processo de reúso e de desenvolvimento baseado em componentes apresentado no Capítulo 3.

Capítulo 5: Análise Modular de Modelos HCPN

Neste capítulo, a solução híbrida de análise modular para o processo descrito no Capítulo 3 é apresentada, assim como alguns exemplos de uso para o estudo de caso apresentado no Capítulo 4.

Capítulo 6: Conclusões

Algumas considerações finais, bem como as contribuições deste trabalho e sugestões para trabalhos futuros são apresentados e discutidos neste capítulo.

Capítulo 2

Fundamentação Teórica

Neste capítulo, conceitos sobre verificação de modelos, redes de Petri coloridas hierárquicas, autômatos com interface, além de uma contextualização sobre trabalhos relacionados são apresentados. A discussão sobre trabalhos relacionados será dividida em quatro partes. Na primeira é realizada uma introdução sobre verificação de modelos. Na segunda, sistemas embarcados e a aplicação de verificação de modelos neste domínio são discutidos. Na terceira, outras técnicas existentes para contornar o problema da explosão do espaço de estados são apresentadas. N quarta, trabalhos relacionados na área de desenvolvimento baseado em componentes são discutidos.

2.1 Redes de Petri Coloridas Hierárquicas

As redes de Petri são um formalismo matemático com uma representação gráfica. A representação gráfica é de grande auxílio para a modelagem, enquanto que a base matemática permite simulação e prova de propriedades automáticas, entre outras vantagens. As redes de Petri coloridas (CPN - *Coloured Petri Nets*) permitem que tipos complexos de dados sejam manipulados pela rede, permitindo a representação de sistemas complexos de forma mais compacta. Devido a esta capacidade as CPN estão inseridas na classificação de redes de alto nível.

A representação gráfica de uma CPN é um grafo bipartido com lugares, representados por elipses, transições, representadas por retângulos e arcos ligando esses dois elementos. As transições representam ações e as marcações dos lugares representam o estado do modelo. A marcação de um lugar é o conjunto de fichas presente no lugar em um dado instante. Fichas são as informações

que são manipuladas pela rede. A marcação de uma rede é a marcação de todos os lugares da rede em um dado instante. Uma ficha pode ser um tipo de dados complexo na linguagem CPN/ML [6].

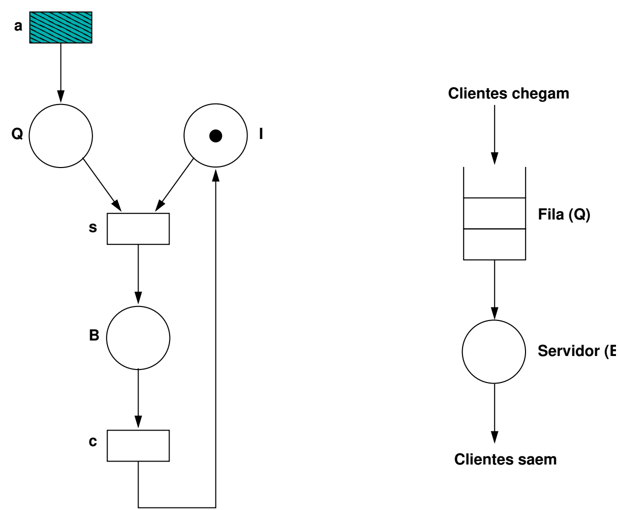
Cada lugar têm um conjunto de cores associado que representa o tipo de fichas que o lugar pode ter. As transições podem ter guardas e código associadas a elas. Guardas são expressões booleanas que devem ser satisfeitas para que a transição seja considerada habilitada a disparar. Códigos podem ser funções que são executadas toda vez que a transição dispara.

Os arcos podem ligar lugares a transições e transições a lugares, mas nunca lugar a lugar ou transição a transição. Os arcos também podem ter expressões complexas e chamadas de funções associadas a eles.

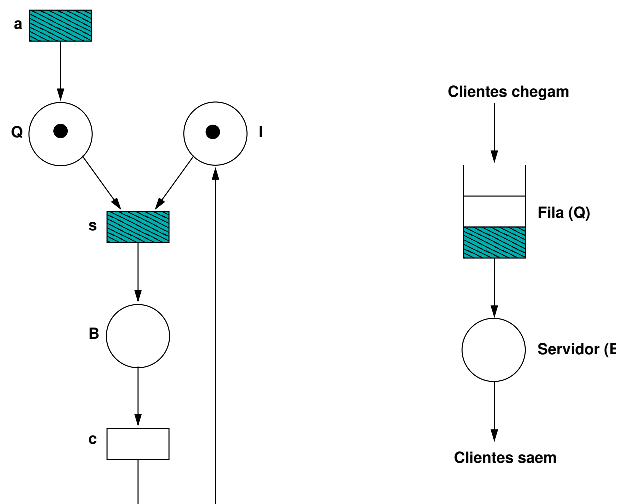
Para que uma transição dispare é necessário que todos os lugares de entrada, isto é, lugares que têm arcos indo do lugar para a transição, tenham um número de fichas maior ou igual ao peso do arco, $W(p,t)$, e que a guarda da transição seja verdadeira. Uma transição habilitada a disparar pode disparar a qualquer momento, não necessariamente imediatamente. Quando a transição dispara ela remove $W(p_i,t)$ fichas de cada lugar de entrada p_i, e cada lugar de saída p_o, isto é, lugares que têm arcos ligando a transição ao lugar, recebe fichas de acordo com a expressão do arco da transição para o lugar: $W(t,p_o)$.

Um exemplo de rede de Petri simples é ilustrado na Figura 2.1. Neste exemplo todos os pesos de arcos são iguais: $W = 1$. Quando não há indicação do peso no arco este é unitário. Neste exemplo também omitimos informações de tipos de dados e hierarquia. Este exemplo ilustra o funcionamento de uma fila, onde do lado esquerdo temos a rede e do lado direito a fila. Nessa figura é ilustrada a seqüencia de disparo as. No estado inicial somente a está habilitada a disparar. Quando a dispara s passa a estar também habilitada. Note que como a não tem lugares de entrada ela sempre está habilitada a disparar. Esse tipo de transição é chamada de *fonte*. O caso contrário, ou seja, uma transição que não tem lugares de saída é chamada de *pia*. Em uma CPN, cada lugar deve ter um conjunto de cor associado, as fichas não são representadas por pontos nos lugares e sim por uma string especificando as fichas presentes no lugar. Além disso, os tipos de fichas produzidos pelas expressões dos arcos de saída de uma transição devem ser compatíveis com o conjunto de cores do lugar de saída da transição.

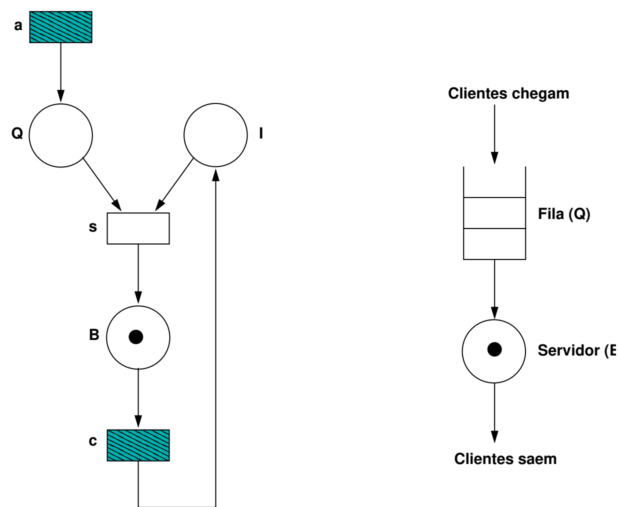
Os lugares podem ter em um dado momento um conjunto de fichas. Esse conjunto pode conter elementos repetidos, ou seja, duas ou mais fichas iguais. Para representar essa possibilidade precisamos definir o conceito de multi-conjunto. Um multi-conjunto é similar a um conjunto, com



(a) Estado inicial.



(b) Disparo de a.



(c) Disparo de s.

Figura 2.1: Exemplo de redes de Petri.

a diferença que pode conter elementos repetidos [50].

Definição 2.1 *Um multi-conjunto* m*, sobre um conjunto* S *não vazio, é uma função* $m \in [S \rightarrow \mathbb{N}]$*. O número inteiro não negativo* $m(s) \in \mathbb{N}$ *é o número de ocorrências do elemento* s *no multi-conjunto* m*. Um multi-conjunto m é representado pela soma formal:*

- $\sum_{s \in S} m(s)`s$.

S_{MS} *denota o conjunto de todos os multi-conjuntos sobre* S*. O número de inteiros não negativos* $\{m(s)|s \in S\}$ *são chamados coeficientes do multi-conjunto* m*, e* $m(s)$ *é chamado coeficiente de* s*. Um elemento* $s \in S$ *é dito pertencente ao multi-conjunto* m *se, e somente se,* $m(s) \neq 0$ *e é escrito* $s \in m$*.*

Considere que $Tipo(v)$ denota o tipo da variável v, $Tipo(exp)$ denota o tipo da expressão exp e $Var(exp)$ denota o conjunto variáveis em uma expressão. A seguir é apresentada a definição formal de CPN conforme [50].

Definição 2.2 *Uma rede de Petri colorida é uma tupla*

$CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *onde:*

i. Σ *é um conjunto finito de tipos não vazios, denominado conjuntos de cores.*

ii. P *é um conjunto finito de lugares.*

iii. T *é um conjunto finito de transições.*

iv. A *é um conjunto finito de arcos tal que:*

   - $P \cap T = P \cap A = T \cap A = \emptyset$

v. N *é uma função de nó. É definida de* A *em* $P \times T \cup T \times P$*.*

vi. C *é uma função de cor. É definida de* P *em* Σ*.*

vii. G *é uma função de guarda. É definida de* T *em expressões tal que:*

   - $\forall t \in T : [Tipo(G(t)) = \mathbb{B} \wedge Tipo(Var(G(t))) \subseteq \Sigma]$.

viii. *E é uma função de expressão de arco. É definida de A em expressões tal que:*

$\bullet \forall a \in A : [Tipo(E(a)) = C(p(a))_{MS} \wedge Tipo(Var(E(a))) \subseteq \Sigma]$.

onde $p(a)$ é um lugar de $N(a)$.

ix. *I é uma função de inicialização. É definida de P em expressões fechadas tal que:*

$\bullet \forall p \in P : [Tipo(I(p)) = C(p)_{MS}]$.

Uma extensão de CPN denominada redes de Petri coloridas hierárquicas [50, 51] (HCPN - *Hierarchical Coloured Petri Nets*) foi definida para permitir a modelagem compondo redes menores, por especialização, abstração, ou uma combinação de ambas. A hierarquia é realizada através de *páginas*. Podemos ter em uma página uma transição, chamada de *transição de substituição* que representará uma outra página que nada mais é do que outro modelo CPN. A página onde a transição de substituição se encontra é chamada de *super-página*, enquanto que a página que esta representa é chamada de *sub-página*. Na sub-página temos lugares denominados de *portas*, que podem ser de entrada, de saída, ou de entrada e saída. Na super-página temos lugares de entrada e saída da transição de substituição que são denominados de *sockets*. Os sockets são associados às portas. Desta forma um modelo complexo pode ser construído a partir de modelos CPN menores.

Existe ainda outro mecanismo de organização denominado lugar de fusão. Vários lugares podem ser reunidos em um conjunto de fusão. Todos os lugares que pertencem a um mesmo conjunto de fusão são denominados lugares de fusão e terão sempre a mesma marcação. A marcação de um lugar são as fichas presentes naquele lugar em um dado instante de tempo. A marcação da rede será a distribuição de fichas em seus lugares. A mudança na marcação de um lugar de fusão implica na mudança, de forma idêntica, na marcação de todos os lugares que pertencem ao conjunto de fusão.

Além das definições já apresentadas também é usado $X = P \cup T$ para denotar o conjunto de todos os nós, e $X \in [X \rightarrow X_S]$ mapeia cada nó x em um conjunto de nós vizinhos, isto é, nós que estão conectados a x por um arco:

$X(x) = \{x' \in X | \exists a \in A : [N(a) = (x, x') \vee N(a) = (x', x)]\}$.

Utiliza-se ainda a notação $^\bullet x$ e $x^\bullet$ para denotar o conjunto de nós de entrada e saída de x, respectivamente. Além disso é definido também a função de tipo de socket, que mapeia cada par de nó socket e transição de substituição em $\{in, out, i/o\}$.

$$ST(p,t) = \begin{cases} in & \text{if } p \in ({}^{\bullet} - t^{\bullet}) \\ out & \text{if } p \in (t^{\bullet} - {}^{\bullet}) \\ i/o & \text{if } p \in ({}^{\bullet} \cup t^{\bullet}) \end{cases}$$

A seguir é apresentada a definição formal de HCPN conforme [50].

Definição 2.3 *Uma CPN hierárquica é uma tupla*

$HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ *onde:*

i. S *é um conjunto finito de páginas tal que:*

- *Cada página* $s \in S$ *é uma CPN:*

$(\Sigma_s, P_s, T_s, A_s, N_s, C_s, G_s, E_s, I_s)$.

- *O conjunto de elementos da rede são disjuntos:*

$\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1} \cup A_{s_1}) \cap (P_{s_2} \cup T_{s_2} \cup A_{s_2}) = \emptyset]$.

ii. $SN \subseteq T$ *é um conjunto de nós de substituição.*

iii. SA *é uma função de associação de página. É definida de* SN *em* S *tal que:*

- *Uma página não é sub-página de si mesma:*

$\{s_0 s_1 ... s_n \in S^* | n \in \mathbb{N}_+ \wedge s_0 = s_n \wedge \forall k \in 1..n : s_k \in SA(SN_{s_{k-1}})\} = \emptyset$.

iv. $PN \subseteq P$ *é um conjunto de nós porta.*

v. PT *é uma função de tipo de portas. É definida de* PN *em* $\{in, out, i/o, general\}$.

vi. PA *é uma função de associação de porta. É definida de* SN *em relações binárias tal que:*

- *Nós socket são associados à nós porta:*

$\forall t \in SN : PA(t) \subseteq X(t) \times PN_{SA(t)}$.

- *Nós socket são do tipo correto:*

$\forall t \in SN \; \forall (p_1, p_2) \in PA(t) : [PT(p_2) \neq general \Rightarrow ST(p_1, t) = PT(p_2)]$.

- *Nós relacionados têm conjunto de cores idênticos e expressões de inicialização equivalentes:*

$\forall t \in SN \; \forall (p_1, p_2) \in PA(t) : [C(p_1) = C(p_2) \wedge I(p_1)\langle\rangle \; = I(p_2)\langle\rangle]$.

vii. $FS \subseteq Ps$ *é um conjunto finito de conjuntos de fusão tal que:*

- *Membros de um conjunto de fusão têm conjuntos de cores idênticas e expressões de inicialização equivalentes:*

  $\forall fs \in FS : \forall p_1, p_2 \in fs : [C(p_1) = C(p_2) \wedge I(p_1)\langle\rangle = I(p_2)\langle\rangle]$.

viii. FT *é uma função de tipos de fusão. É definida de conjuntos de fusão em* $\{global, page, instance\}$ *tal que:*

- *Conjuntos de fusão de página e instância pertencem a uma única página:*

  $\forall fs \in FS : [FT(fs) \neq global \Rightarrow \exists s \in S : fs \subseteq Ps]$.

ix. $PP \in S_{ms}$ *é um multi-conjunto de páginas principais.*

Na Figura 2.2 são ilustrados exemplos simples de lugares de fusão e transições de substituição. A transição de substituição TS representa toda a sub-página. Os lugares de fusão LF1 e LF2 pertencem ao mesmo conjunto de fusão, de forma que qualquer alteração na marcação em um provoca a alteração no outro, como se fossem um só, apesar de graficamente serem distintos. O socket de entrada da transição de substituição na super-página SE está associado à porta de entrada da sub-página PE. Da mesma forma que SS está associado à PS.

2.2 Introdução à Verificação de Modelos

A necessidade de aumentar a confiança no funcionamento de sistemas de software motivou a definição e aplicação de métodos e técnicas rigorosos de desenvolvimento. Essa necessidade se torna mais evidente quando os sistemas em questão são sistemas críticos e de tempo real. Com o aumento da complexidade dos sistemas, os métodos tradicionais baseados em testes, por exemplo, não são mais suficientes para garantir a confiança no funcionamento, nem tampouco estabelecer qualidade. No contexto deste trabalho confiança no funcionamento significa que erros que podem levar a catástrofes não ocorrem, enquanto qualidade significa que o sistema funciona como desejado, ou descrito nos requisitos.

O uso de métodos formais no desenvolvimento pode aumentar a confiança e, portanto a qualidade dos sistemas. Na especificação, o uso de métodos formais tem uma grande importância pois

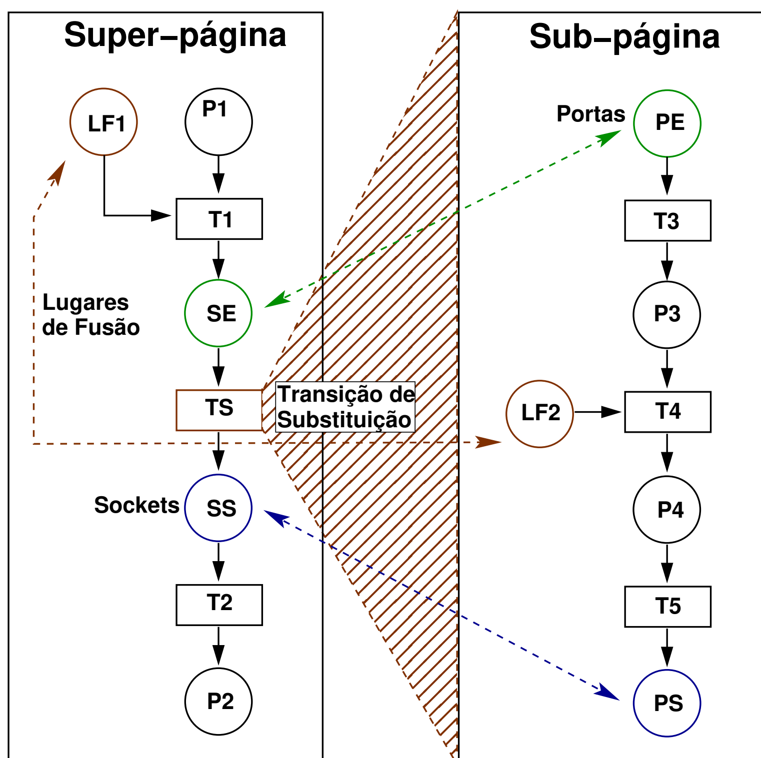


Figura 2.2: Rede de Petri colorida hierárquica.

permite descobrir erros mais difíceis como lógicos e de projeto, por exemplo, antes do desenvolvimento do sistema. Métodos tradicionais baseados em testes e simulação aplicados a sistemas complexos podem conseguir detectar erros simples, como manipulação de dados, por exemplo, em estágios iniciais. Mas dificilmente conseguiriam detectar erros quando os erros mais simples tiverem sido corrigidos pois não são exaustivos.

Técnicas como prova de teoremas e verificação de modelos [11] são aplicadas para verificar especificações formais. Apesar de não ser possível garantir completude, ou seja, que todas as propriedades necessárias para garantir a segurança do sistema foram verificadas, métodos formais são utilizados para verificar propriedades de forma exaustiva. Isto é, enquanto que utilizando testes e simulação é possível analisar apenas algumas possibilidades de comportamento, utilizando métodos formais é possível analisar todos os comportamentos possíveis do modelo.

A grande vantagem de verificação de modelos sobre outras técnicas é que esta é totalmente automática. Além disso, os algoritmos de verificação de modelos definem que no caso em que a validação de uma propriedade falha, é apresentado um contra-exemplo indicando o caminho em que a propriedade é falsa.

A grande desvantagem da verificação de modelos é a explosão do espaço de estados. Isto ocorre principalmente quando temos vários componentes em paralelo se comunicando, ou quando temos tipos complexos de dados nos modelos. Algumas técnicas que têm sido desenvolvidas para contornar este problema incluem algoritmos simbólicos [61], e redução de ordem parcial [66, 72].

Basicamente, a verificação consiste em verificar se uma propriedade é satisfeita em um modelo, isto é, verificar se o modelo é modelo da especificação. As propriedades são descritas em lógica temporal [39], enquanto que os modelos podem ser descritos em autômatos ou redes de Petri [63]. Então, seja um modelo $\mathcal{M}$ e uma fórmula f que expressa alguma propriedade desejável com respeito a $\mathcal{M}$. A verificação de modelos consiste em verificar se $\mathcal{M}$ modela f: $\mathcal{M} \models f$.

A verificação de modelos consiste em três atividades:

1. Modelagem

   A modelagem consiste em descrever o sistema em algum formalismo. O formalismo a ser utilizado depende da ferramenta que será utilizada na verificação, do conhecimento do projetista, ou cultura da instituição onde o projeto está sendo desenvolvido. É possível ainda, transformar um formalismo em outro, para a realização da verificação.

2. Especificação

   A especificação geralmente é feita em lógica temporal, que serve para especificar como o comportamento do sistema evolui ao longo do tempo. Conforme dito anteriormente não é possível garantir completude, ou seja, mesmo que uma propriedade seja provada de forma exaustiva para todo comportamento possível, não é possível garantir que todas as propriedades que o sistema deve satisfazer foram especificadas. Não é possível garantir a completude da especificação.

3. Verificação

   Dado um modelo e uma especificação, a verificação é completamente automática. Mas no caso de falsidade, o projetista deve analisar o contra-exemplo para resolver possíveis erros na modelagem, ou reformular a especificação. Além disso, técnicas de abstração ou modularização podem depender do projetista, para permitir que a verificação possa ser executada, contornando o problema da explosão do espaço de estados.

Existem ainda diferentes abordagens para verificação de modelos. Dois exemplos são os projetos *Bandera*[1] [13], e *Java Path Finder*[2] [46]. Nesses dois projetos são implementados mecanismos para extrair um modelo a partir de um código descrito em linguagem Java para realizar verificação de modelos.

2.3 Teoria de Interfaces

A teoria de Autômatos com Interface (Temporizados) (*TIA - Timed Interface Automata*), introduzida em [34, 33, 32] é utilizada para a especificação de interface de componentes. Essas especificações são utilizadas para a verificação de compatibilidade, isto é, para verificar se duas interfaces podem ser utilizadas juntas. Essa verificação é realizada utilizando teoria dos jogos. As definições formais e algoritmos podem ser encontradas nas referências citadas e serão omitidas neste trabalho. Especificamente, com a necessidade de uma maior expressividade na especificação de interfaces para sistemas complexos, uma extensão dessas teorias, chamada interfaces sociáveis [31], foi definida. Esta extensão permite a definição de ações compartilhadas e variáveis, inclusive globais.

Uma interface pode ter requisitos (suposições) de entrada, ações de entrada, e comportamento (garantias) de saída, ações de saída. Pode-se dizer que duas interfaces são compatívies se elas podem ser utilizadas (através de um ambiente) de forma que os requisitos de entrada de ambas são simultaneamente satisfeitos.

No contexto do processo de desenvolvimento baseado em componentes e reúso de modelos apresentado no Capítulo 3, TIA pode ser utilizado de duas formas. Uma possibilidade é a especificação da interface de novos componentes, para a validação deste antes da inserção no repositório. Outra possibilidade é a decomposição hierárquica quando o modelo está inserido em um projeto, garantindo que se um modelo satisfaz sua especificação, ele irá interagir de forma correta com o restante do projeto.

A seguir a definição formal de TIA é apresentada.

Definição 2.4 *Dados dois conjuntos A e B, $A \rightrightarrows B$ denota o conjunto de funções não-determinísticas de A para B, isto é, $A \rightarrow 2^B$.*

---

[1] http://bandera.projects.cis.ksu.edu/

[2] http://ase.arc.nasa.gov/visser/jpf/

Definição 2.5 *Um autômato com interface sociável é a tupla* $M = (Act, S, \tau^I, \tau^O, \varphi^I, \varphi^O)$*, onde:*

- Act *é um conjunto de ações.*
- S *é um conjunto de estados.*
- $\tau^I : Act \times S \rightarrow S$ *é a função de transição de entrada.*
- $\tau^O : Act \times S \rightarrow S$ *é a função de transição de saída.*
- $\varphi^I \subseteq S$ *é o invariante de entrada.*
- $\varphi^O \subseteq S$ *é o invariante de saída.*

É requerido que τ^I *seja determinístico, isto é: para todo* $s \in S$ *e* $a \in Act$*,* $|\tau^I(a, s)| \leq 1$.

Para diferenciar na notação gráfica de autômato as ações de entrada e saída, estas são denotadas com o sufixo ? e !, respectivamente. A sincronização ocorre nas ações de entrada e saída. A composição de duas interfaces é realizada primeiro calculando-se o produto dos dois autômatos. O resultado da sincronização de uma ação de entrada com uma de saída é a ação de saída. A sincronização também pode ser realizada nas ações de entrada. Neste caso os dois autômatos podem receber simultaneamente esta ação. No caso de ações de saída eles não sincronizam, isto é, cada um pode disparar sua ação de saída independentemente.

No produto de dois autômatos é feita a distinção entre estados *bons* e estados *ruins*. Estados bons são aqueles em que toda ação de saída emitida por um componente pode ser aceita pelo outro, quando as ações são compartilhadas por ambos, caso contrario o estado é ruim. No caso em que as ações não são compartilhadas é realizado o produto. A composição é realizada retirando-se do produto todos os estados que levam à estados ruins. Se a composição é não-vazia dizemos que as interfaces são compatíveis. A ferramenta TICC (*Timed Interfaces Compatibility Checker*)[3] [2] foi desenvolvida para especificação e verificação de compatilidade de interfaces.

Neste trabalho, abstraímos as estruturas de dados devido à limitação de TIA de possuir, na versão atual, somente os tipos faixa de inteiros, booleano, e clocks (relógios) que são, também, implementados como uma faixa de inteiros.

[3]http://www.dvlab.net/dvlab/Ticc

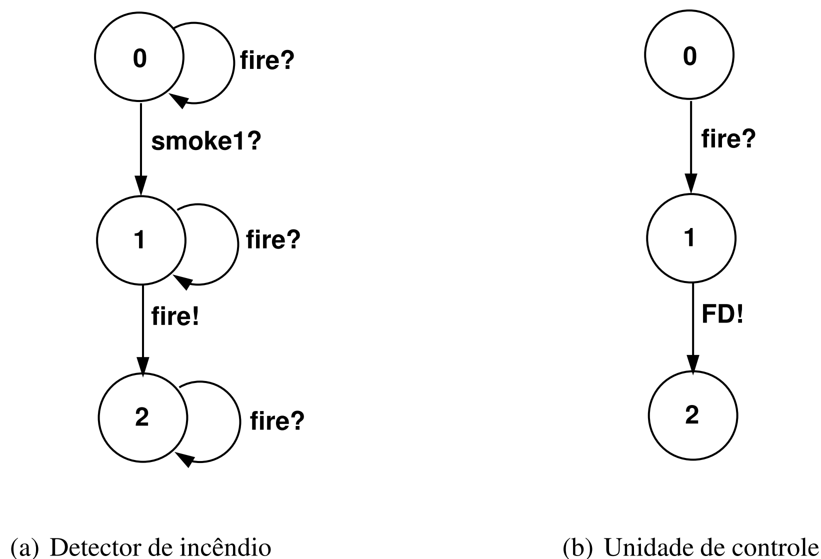
Na Figura 2.3 um exemplo de TIA para um detector de incêndio é ilustrado. Na figura 2.3(a) um sensor de fumaça pode se visto. Ele recebe sinal de fumaça, ação de entrada smoke1? e envia alarme de incêndio, ação de saída fire!. Observe que a cada estado ele também pode receber alarme de incêndio. Isto é devido ao fato de o modelo ser aberto, ou seja, poder ser composto com vários outros sensores. Na Figura 2.3(b) a unidade controladora pode ser vista. Ao receber um alerta de incêndio de um sensor, esta envia um chamado ao departamento de corpo de bombeiros, ação de saída FD!. Observando estas duas figuras pode-se observar que a única ação compartilhada por ambas é fire. Para que estas duas interfaces sejam compatíveis, toda vez que uma envia uma saída fire! a outra deve estar apta a recebe-la através da entrada fire?. Esta situação só ocorre quando o detector está no estado 2. Neste caso, a unidade de controle deve estar no estado 1. Se algum outro detector enviar o alarme, e for recebido pela unidade de controle, esta não recebe mais outros alarmes, e as combinações possíveis a partir desse estado são removidas da composição. No caso deste exemplo, é fácil perceber que as interfaces são compatíveis. Um estudo mais detalhado sobre TIA, bem como a análise completa deste exemplo pode ser encontrada em [31]. O Código na linguagem de entrada para a ferramenta TICC referente a Figura 2.3 pode ser visto a seguir.

```
module ControlUnit:
    var s: [0..2]
    input fire : { local: s = 0 ==> s′ := 1 }
    output FD  : {        s = 1 ==> s′  = 2 }
endmodule

module FireDetector1:
    var s: [0..2]
    input fire : { }
    input smoke1 : { local: s = 0 ==> s′ := 1 }
    output fire  : {        s = 1 ==> s′  = 2 }
endmodule
```

As interfaces são especificadas com a palavra-chave module. A seguir as variáveis locais são definidas. No caso das ações de entrada, podemos ter uma parte local e outra global. A local deve ser determinística.

Apesar de em HCPN ser possível modelar paralelismo e em TIA, assim como em outros formalismos baseados em autômato, não ser possível, isso não é considerado como uma deficiência pois, assim como nos grafos de ocorrência, o paralelismo pode ser descrito como uma ordenação parcial, sem perda semântica.



(a) Detector de incêndio (b) Unidade de controle

Figura 2.3: TIA para detector de incêndio.

É importante salientar que duas interfaces podem ser verificadas através da ferramenta para se analisar se podem ser utilizadas juntas, conforme apresentado em [31].

2.4 Sistemas Embarcados

Cada vez mais os sistemas embarcados estão fazendo parte da vida das pessoas nas mais diferentes aplicações. O aumento do poder de processamento dos dispositivos tem permitido o desenvolvimento de aplicações cada vez mais complexas. Além disso, geralmente os sistemas embarcados devem satisfazer restrições de tempo, e consumo de energia, por exemplo.

Várias pesquisas e aplicações têm sido desenvolvidas na área de sistemas embarcados nos últimos anos. É cada vez mais necessário dispor de técnicas, métodos, e ferramentas que possibilitem tratar os diferentes problemas neste domínio específico.

Na literatura podemos encontrar várias definições para sistemas embarcados. Apesar das muitas definições serem diferentes, pode-se observar alguns pontos em comum entre várias definições.

No contexto deste trabalho adotamos a seguinte definição para sistemas embarcados:

Definição 2.6 *Sistemas embarcados são aqueles que possivelmente fazem parte de um sistema maior, que interage diretamente com o mundo físico, satisfazendo restrições de tempo e de consumo de energia, para os quais nem sempre o usuário tem acesso direto ao dispositivo.*

Esta definição de sistemas embarcados é bastante genérica. Uma definição mais restrita seria de sistemas que sempre interagem com o mundo físico e o usuário não tem acesso ao dispositivo. Esta definição apesar de ser bem aplicada, excluiria os atuais dispositivos chamados de eletrônica de consumo, como telefones celulares, computadores de mão, assistentes pessoais, entre outros. Neste momento tal restrição não é necessária, portanto relaxamos um pouco a definição de sistemas embarcados adotada no contexto deste trabalho. Esta restrição pode ser aplicada, caso seja necessário restringir ainda mais o domínio de aplicação para sistemas embarcados de missão crítica. Como exemplo de tais sistemas podemos citar sistemas de controle de vôo, computadores de bordo de automóveis, redes de transdutores inteligentes para controle de processos, entre outros.

A confusão com relação à definição de sistemas embarcados e à imagem que as pessoas criaram está relacionada com alguns falsos conceitos a respeito de computadores e sistemas embarcados. Grande parte dos sistemas embarcados em eletrônica de consumo, como celulares, assistentes, computadores de mão, exploram a mobilidade como principal característica e vantagem. Mas mobilidade não é um fator determinante. O controle do laser de um tocador de discos compacto realiza, através de um sistema embarcado, cálculos complexos para manter o foco e a estabilidade do canhão laser e do disco. Além disso, um computador pessoal normal pode ser utilizado com um sistema embarcado, num sistema maior como, por exemplo, no controle de sensores e atuadores inteligentes no chão de uma fábrica. Além disso, as pessoas ainda vêem os sistemas embarcados como sistemas de poder computacional limitado, o que não é verdade.

A grande vantagem de sistemas embarcados é na utilização de seu poder computacional para servir o usuário, e não o contrário. Além disso, os sistemas embarcados sempre são destinados a uma aplicação específica diferentemente dos computadores de propósito geral, como os computadores pessoais. É desejável tirar proveito das novas possibilidades que sistemas embarcados possibilitam, explorando sua capacidade de processamento, localidade da ação, processamento

distribuído, e o constante barateamento dos dispositivos. Além disso, é importante dispor de processos de desenvolvimento que permitam atingir esse objetivo [74].

Um projeto importante na área de sistemas embarcados é o projeto *Ptolemy*[4] [56]. O Ptolemy é um editor e simulador de diagramas de blocos para tempo contínuo, sistemas híbridos, de fluxo de dados, concorrentes, e de tempo real. Uma característica importante do Ptolemy é a escolha de modelos de computação para controlar a simulação e interação entre os componentes do sistema. Os modelos de computação mais úteis para o domínio de sistemas embarcados, o principal foco do Ptolemy, são modelos que controlam tempo e concorrência. Isto é devido ao fato de que sistemas embarcados geralmente possuem componentes que executam concorrentemente e em um ambiente temporizado. O principal objetivo no contexto do projeto Ptolemy é a construção e interoperabilidade de modelos construídos através de vários modelos de computação.

Apesar do Ptolemy disponibilizar vários modelos de computação para controlar a interação dos componentes, não existe uma ferramenta de verificação formal para os modelos desenvolvidos nesta ferramenta. A análise é feita por meio de simulação. Além disso, não existe uma metodologia sistemática de desenvolvimento, considerando reúso, por exemplo. Uma outra diferença entre esta pesquisa e o projeto Ptolemy é que este último não considera a especificação, ou modelagem, da interface dos componentes. Consideramos que no desenvolvimento baseado em componentes, é importante ter a interface, e sua especificação, separada do componente, e sua especificação.

2.5 Explosão do Espaço de Estados

A especificação de sistemas concorrentes e que manipulam tipos complexos de dados pode levar a um crescimento exponencial dos estados possíveis do sistema com relação aos estados dos componentes do sistema. Este problema é denominado de explosão do espaço de estados. Em técnicas de análise baseadas na enumeração do espaço de estados, como verificação de modelos, é necessário desenvolver e aplicar técnicas para contornar este problema.

As duas principais abordagens para redução do espaço de estados são a verificação simbólica de modelos e a redução de ordem parcial. Mas para sistemas complexos, mesmo utilizando estas técnicas ainda podemos ter o problema da explosão do espaço de estados. Portanto outras técnicas como, por exemplo, raciocínio modular, e abstração, são utilizadas em conjunto com as duas

[4] http://ptolemy.eecs.berkeley.edu/

primeiras para tentar contornar o problema.

2.5.1 Verificação Simbólica de Modelos

A verificação simbólica foi proposta inicialmente por Kenneth McMillan em sua tese de doutorado de 1992 e publicada em livro em 1993 [61]. Nos primeiros algoritmos de verificação de modelos, a enumeração dos estados do modelo era explícita [11], através de uma estrutura de *Kripke*, definida a seguir:

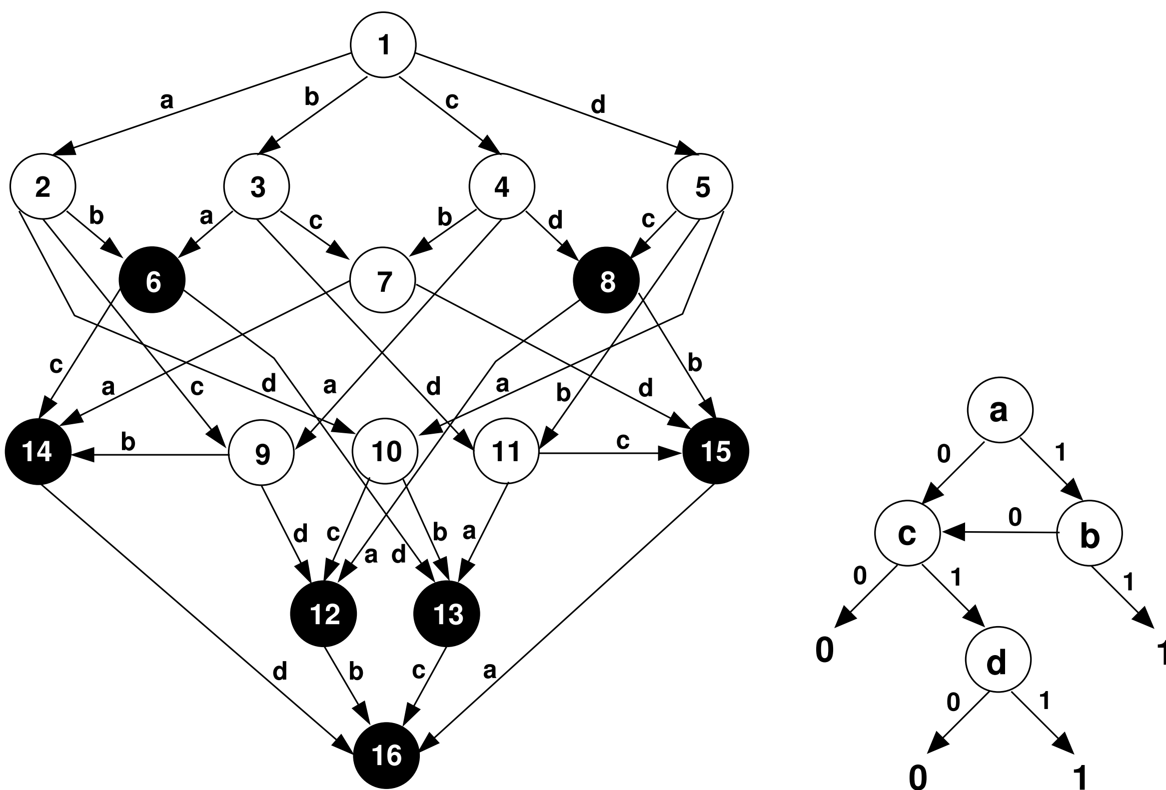
Definição 2.7 *Seja AP um conjunto de proposições atômicas. Uma estrutura de Kripke M sobre AP é uma tupla de 4 elementos $M = (S, S_0, R, L)$, onde*

1. *S é um conjunto finito de estados.*
2. *$S_0 \subseteq S$ é o conjunto de estados iniciais.*
3. *$R \subseteq S \times S$ é uma relação de transição que deve ser total, isto é, para cada estado $s \in S$ existe um estado $s\prime \in S$ tal que $R(s, s\prime)$.*
4. *$L : S \rightarrow 2^{AP}$ é uma função que rotula cada estado com um conjunto de proposições atômicas que são verdadeiras naquele estado.*

Como exemplo, considere a fórmula $a \wedge b \vee c \wedge d$. Utilizando enumeração explícita de todos os valores possíveis, o espaço de estados teria $2^4 = 16$ estados. Como pode ser visto na figura 2.4(a), onde os estados sombreados representam os estados onde a fórmula acima é válida.

No caso da verificação simbólica, a representação do espaço de estados se dá através de diagramas de decisão binários ordenados (OBDD - *Ordered Binary Decision Diagram*). Nesta representação os estados são representados por valores booleanos associados às variáveis do sistema que são verdadeiras naquele estado. A relação de transição é uma fórmula booleana em termos de dois conjuntos de variáveis, uma do estado anterior e outra do estado futuro. Esta fórmula é representada por um diagrama de decisão binário. O algoritmo de verificação de modelos é baseado na computação de pontos fixos dos transformadores de predicados que são obtidos da relação de transição [11].

Utilizando o mesmo exemplo teremos uma representação mais compacta utilizando OBDD como é mostrado na Figura 2.4(b). O símbolo 1 representa verdadeiro, enquanto que 0 representa falso.



(a) Espaço de estados com enumeração explícita

(b) Diagrama de decisão binário ordenado.

Figura 2.4: Representações: (a) explícita; (b) simbólica.

2.5.2 Redução de Ordem Parcial

Em sistemas concorrentes, quando os componentes se comunicam entre si podemos ter um número enorme de estados possíveis. Mas quando as ações dos componentes são independentes, ou seja, a ordem de execução não altera o resultado final, o que se tem é o entrelaçamento das ações dos componentes. Desta forma é possível considerar apenas algumas seqüências ao provar propriedades do modelo, descartando os outros entrelaçamentos possíveis.

A técnica de redução de ordem parcial consiste em decidir quais ordenações parciais, ou entrelaçamentos, considerar e quais descartar para verificar propriedades [66, 72]. A idéia é que

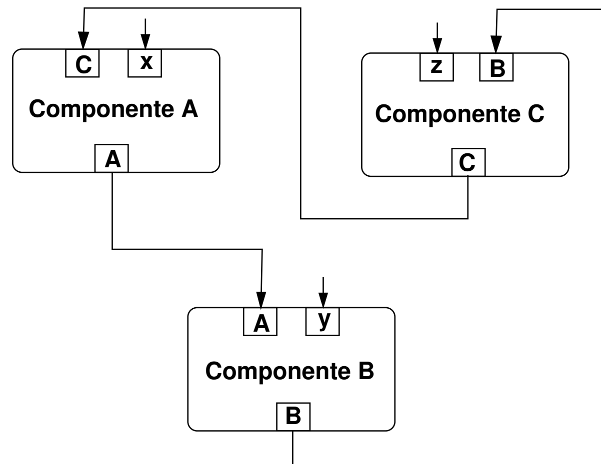


Figura 2.5: Raciocínio circular.

quando uma especificação não pode distinguir dois entrelaçamentos que diferem somente pela ordem em que eventos concorrentes ocorrem, então é possível analisar apenas um deles.

2.5.3 Raciocínio Modular

O raciocínio modular é uma técnica que se baseia na idéia de que os sistemas finitos são compostos de vários componentes. Estes por sua vez podem ser decompostos em propriedades que descrevem seu comportamento. Desta forma é possível analisar propriedades locais de cada componente, e se todas forem validadas e a conjunção destas implica na especificação total, então o sistema completo satisfaz tal especificação [11]. Apesar da simplicidade conceitual, o problema central com este tipo de raciocínio é o fato de que os componentes podem fazer suposições entre eles. A medida que os componentes são validados, as suposições sobre eles vão sendo utilizadas para verificar outros componentes. Isto é denominado de raciocínio circular.

Técnicas mais complexas devem ser utilizadas para realizar a verificação modular no problema de raciocínio circular. Um exemplo do problema de raciocínio circular pode ser visto na Figura 2.5. Neste exemplo, para verificar o Componente A é necessário verificar o Componente C pois o primeiro faz suposições a respeito deste último. Mas para verificar o Componente C é necessário verificar o Componente B. E assim sucessivamente.

A técnica de especificação das suposições e garantias dos componentes é denominada de paradigma assume-garante. Considerando que dado o que o componente assume, temos o que ele

garante, podemos formalizar o assume-garante para o exemplo ilustrado na Figura 2.5 da seguinte forma:

$$\frac{C, x}{A} \qquad \frac{A, y}{B} \qquad \frac{z, B}{C} \tag{2.1}$$

A primeira regra especifica que se C e x forem satisfeitos, então A é satisfeito. Desta formalização concluímos que A depende de C, mas C depende de B, que por sua vez depende de A. Este é o problema do raciocínio circular. É necessário definir uma técnica que permita verificar propriedades locais de cada componente tratando a possibilidade de existência de raciocínio circular, se não, anulando-o.

2.5.4 Abstração

Mesmo com o uso de técnicas como verificação simbólica de modelos e redução de ordem parcial, ainda é possível ocorrer o problema da explosão do espaço de estados. A abstração permite que a verificação simbólica seja aplicada mesmo quando existe a manipulação de tipos complexos de dados. É possível mapear tipos complexos de dados em tipos mais simples, pois nem sempre todos os elementos de um conjunto são utilizados. Esta técnica é conhecida como abstração [9, 68]. A abstração consiste em mapear valores reais em um conjunto menor de valores abstratos. Considere, por exemplo, que se deseja verificar um multiplicador de dois números inteiros. O modelo recebe duas entradas inteiras e retorna uma saída também inteira. Mas podemos ter infinitos valores de entrada e saída para este modelo, o que tornaria a verificação impraticável. Uma solução usando abstração neste problema seria com relação aos sinais. Para saber o sinal da saída é necessário saber apenas o sinal das entradas. Portanto pode-se mapear todos os inteiros positivos no conjunto abstrato dos positivos. O mesmo raciocínio serve para os inteiros negativos e o zero. Como exemplo, a formalização para o problema de verificação do sinal de saída de uma multiplicação de dois inteiros, utilizando abstração é:

$$\frac{p, n}{n} \qquad \frac{n, p}{n} \qquad \frac{n, n}{p} \qquad \frac{p, p}{p} \qquad \frac{p, z}{z} \qquad \frac{n, z}{z} \tag{2.2}$$

Onde p é um inteiro positivo, n um inteiro negativo. Então, com base nessas regras é possível decidir se o resultado da soma é positiva ou negativa somente considerando o sinal das entradas.

Note que é muito mais simples realizar a verificação neste modelo abstrato do que em um modelo possivelmente infinito. Uma técnica utilizada para realizar este tipo de mapeamento é conhecida como interpretação abstrata [15]. Na interpretação abstrata, quando se prova algo para o modelo abstrato, esta prova vale também para o modelo original.

2.6 Trabalhos Correlatos

Vários trabalhos têm sido desenvolvidos no contexto de componentes e reúso de software. Szyperski [71], em seu livro, defende componentes como unidades binárias, e discute vários aspectos desta tecnologia como mercado, padrões, e questões técnicas de implementação de componentes e software baseado em componentes.

Outros trabalhos interessantes de Ivica Crnkovic [17] tratam da especificação, implementação, e desenvolvimento de componentes. Alguns conceitos como interface, contratos, arcabouços e padrões são tratados com relação a especificação e implementação de aspectos funcionais e não funcionais. Um exemplo de especificação informal de comportamento é mostrado. Neste trabalho, alguns problemas do CBD são apontados como, por exemplo, falta de padronização, e falta de técnicas, métodos, e ferramentas para tratar aspectos não funcionais dos componentes. Em outro trabalho, Crnkovic trata dos desafios e problemas do CBD [16]. Algumas tecnologias como arquitetura de software e *Unified Modeling Language* (UML) [4] são citadas como suporte ao CBD. Além disso, algumas necessidades e desafios atuais de CBD são descritos.

Oscar Nierstrasz [62] também tem desenvolvido pesquisas interessantes no contexto de CBD. Em um de seus trabalhos ele trata de questões de reúso de software, composição de componentes, de objetos, e de classes. Questões como reúso caixa-preta, onde os detalhes interiores do que está sendo reusado não precisam ser conhecidos, são tratados também. Além dessas questões, são discutidas as vantagens de CBD e requisitos para um ambiente de composição visual. Em seu outro trabalho são tratados questões de composição [69]. Neste contexto, ele trata de linguagens de *script* para composição de componentes, e fala sobre uma linguagem experimental chamada Piccola [1].

Além desses trabalhos, os conceitos e pesquisas em arquitetura de software [70, 3] e linha de produto [12], entre outros, também estão relacionados com CBD e CBSE.

Em nosso trabalho, tratamos explicitamente da fase de modelagem de sistemas complexos.

A abordagem utilizada é o reúso de modelos de componentes. Para que o reúso seja possível, utilizamos conceitos de arquitetura de software para desenvolver um arcabouço para a composição de componentes. Os modelos são descritos e compostos gráfica e formalmente com redes de Petri. Para guiar esta estratégia, utilizamos um processo de linha de produto que engloba o conjunto de técnicas, métodos, estratégias, ferramentas, negócio, entre outros, para o desenvolvimento de família de sistemas com características e mercados semelhantes a partir de componentes comuns.

Capítulo 3

Processo de Desenvolvimento Baseado em Componentes e Reúso de Modelos

A aplicação de técnicas e métodos utilizados em engenharia no desenvolvimento de sistemas de software deu origem a disciplina de engenharia de software. Para atingir esse objetivo foi necessária uma mudança na forma como o software é desenvolvido, não apenas a nível de implementação, mas também no nível de processo. Neste capítulo, descrevemos um novo processo de desenvolvimento baseado em componentes e reúso de modelos [19, 24].

Em se tratando de sistemas complexos de software, a utilização de uma estratégia para o uso das técnicas e métodos definidos favorece não apenas o desenvolvimento mas também o gerenciamento do negócio. Ainda, é necessário um processo que guie o desenvolvimento de soluções, utilizando estratégias, métodos, técnicas, e ferramentas, em várias etapas do desenvolvimento como o gerencial, de negócios, legal, entre outros.

Linhas de produtos têm sido utilizadas no desenvolvimento em série de artefatos com características semelhantes [12]. Esse processo favorece a obtenção de produtos similares com uma diminuição do esforço, tempo e custo. Para que uma linha de produto seja possível é necessário mudar o gerenciamento e as etapas da produção para aproveitar as vantagens que esse processo proporciona.

No desenvolvimento de sistemas de software, linhas de produtos têm sido utilizadas para obter diferentes produtos baseados em partes ou blocos comuns. Para isso é necessário definir uma estratégia de negócio que guie o processo de desenvolvimento para produzir valores comuns a

serem utilizados em um domínio definido e conhecido. Os valores desenvolvidos e utilizados em uma linha de produto podem ser desde estratégias para coleta de requisitos e decisões de projeto até trechos de código ou código executável. A partir dessas partes, vários sistemas podem ser desenvolvidos com características específicas, baseados em valores comuns. O conjunto desses sistemas é denominado de família de sistemas.

Com base nesse processo podemos ter dois tipos de negócios possíveis. O primeiro é desenvolver partes de sistemas para servirem de base para uma família em um domínio comum. O segundo é o desenvolvimento de sistemas com partes existentes. É importante notar que uma mesma entidade pode realizar os dois tipos de negócios.

Mas a utilização de partes para desenvolvimento implica na definição de uma estratégia para integração. É necessário definir como e onde as partes serão integradas, como elas se relacionam entre si, e possivelmente restrições de integração e relacionamento, resultando em uma arquitetura de software [70]. Portanto, é possível definir uma arquitetura padrão para uma família de sistemas a serem desenvolvidas com partes comuns.

As partes podem ser classificadas de acordo com suas funcionalidades e chamadas de componentes [65, 62, 71, 16, 17]. Assim, o sistema passa a ser desenvolvido a partir de componentes utilizados como blocos de construção para sistemas complexos. Um componente pode ser visto como um sistema autônomo que implementa uma funcionalidade específica com uma interface bem definida.

Finalmente é necessário definir técnicas e métodos para gerenciamento e reúso dos componentes. A utilização de métodos formais na modelagem de sistemas agrega diversas vantagens como, por exemplo, simulação automática, e prova de propriedades. Como queremos além de sistematizar a modelagem, garantir uma maior confiança no funcionamento, utilizamos métodos e técnicas para classificação e recuperação, adaptação, integração e verificação de uso baseados em redes de Petri [63] e lógica temporal [39]. No contexto deste trabalho, sistemático quer dizer: pertencente a um sistema; metódico; ordenado; organizado; cuidadosamente planejado; feito com intenção determinada. Portanto, é desejável aproveitar esforços dentro de domínios específicos, sistematizando a modelagem. O conjunto de ferramentas Design/CPN [48, 49] é utilizado para a edição, e análise de modelos HCPN.

Várias fases formam o ciclo de vida de um software. Entre elas podemos citar a análise de requisitos, a modelagem, e a implementação, entre outras. Acreditamos que a fase de modelagem,

realizada gráfica, formal, e sistematicamente, é a mais importante. A modelagem permite, entre outras coisas, documentação, validação de requisitos, prova de propriedades e identificação de erros de projeto ou de especificação antes da concepção do sistema. A correção de erros na fase de modelagem consome menos recursos do que em fases como a implementação, por exemplo.

Neste trabalho estamos interessados na fase de modelagem, e em como uma modelagem sistemática e formal pode melhorar o processo de desenvolvimento de sistemas complexos. Uma modelagem sistemática visa o uso de experiências anteriores e a definição de modelos e teorias a serem utilizados. Isso significa que ao invés de solucionar um problema específico, queremos definir uma estratégia genérica.

3.1 Linha de Produto, Componentes, e Arquitetura de Software

Nesta seção apresentamos alguns conceitos e comentários sobre temas utilizados neste trabalho. O uso de componentes em uma linha de produto, com a definição de uma arquitetura, tem o objetivo de desenvolver sistematicamente sistemas complexos de software. Por sistemas complexos, entendemos como sistemas grandes.

Muitos conceitos diferentes para linha de produto, componentes, e arquitetura podem ser encontrados na literatura. Para este trabalho adotamos as seguintes definições.

Definição 3.1 *Arquitetura de Software: Descrição da estrutura de um software com relação as suas funcionalidades. Fazem parte desta descrição a especificação do relacionamento entre funcionalidades, localização dessas, e possíveis restrições de relacionamentos e participações.*

Definição 3.2 *Componentes: Unidades autônomas, com ciclos de vida independentes, que representam uma funcionalidade específica. Um componente é composto de funcionalidade, interface, e possivelmente outras características não funcionais.*

Definição 3.3 *Linha de Produto: Produtos diferentes, baseados em um conjunto de valores comuns, em um domínio definido. Esses valores são modelos de componentes, arquitetura, ferramentas de suporte, estratégias, métodos, e técnicas.*

No contexto específico de modelagem, a arquitetura é descrita por um arcabouço para a integração dos modelos de componentes. Os componentes têm suas funcionalidades modeladas por redes de Petri. Além disso, outras características não funcionais podem ser adicionadas aos modelos no repositório. O método de gerenciamento do repositório explicita, separado do modelo do componente, o seu contrato.

Definir um domínio, uma linha de produto nesse domínio, e tirar proveito de esforços em projetos no mesmo domínio, guiados por esse processo é uma forma de usar eficientemente recursos como dinheiro e tempo de desenvolvimento. Por outro lado o uso de métodos formais nos possibilita abordar o problema de confiança no funcionamento.

De acordo com alguns trabalhos, componentes são uma forma de desenvolver sistemas utilizando componentes como blocos de construção para sistemas maiores [62, 71, 16]. Isso é chamado de Desenvolvimento Baseado em Componentes (*Component Based Development - CBD*). Mas, para que isso seja possível, as técnicas e métodos até então utilizados no desenvolvimento de software devem ser modificados, ou definidos novos, para atender requisitos específicos de CBD. No contexto da Engenharia de Software Baseada em Componentes (*Component Based Software Engineering - CBSE*) procura-se definir o conjunto de disciplinas que possibilitam o CBD.

Os conceitos de linha de produto, entre outros como, por exemplo, os processos da engenharia de software tradicional e orientada a objetos, podem ser utilizados para guiar o desenvolvimento baseado em componentes. Desde a fase de análise de requisitos deve-se ter em mente que os componentes que estão sendo desenvolvidos serão utilizados em outros projetos. Além disso, o sistema a ser desenvolvido deve utilizar componentes existentes. A definição de um processo deste tipo permite viabilizar o uso de componentes pois seu sucesso depende diretamente da manutenção e evolução de uma base de componentes reusáveis e de mudar o foco de desenvolvimento para montagem de sistemas.

3.2 Ciclo de Vida do Desenvolvimento Baseado em Componentes

Vários modelos de ciclo de vida estão presentes na engenharia de software tradicional. Esses ciclos devem ser adaptados ao uso com componentes. Um exemplo no artigo de Crnkovic [16]

mostra como um ciclo tradicional é utilizado no contexto de componentes. Apesar da adaptação, não apenas no exemplo citado, mas em outros modelos, esses não enfatizam a fase de modelagem no desenvolvimento de sistemas.

O ciclo de vida do CBD é dividido em duas partes distintas. Primeiro temos o desenvolvimento de componentes genéricos para uso em outras aplicações. Um grande problema neste caso é a definição dos requisitos pois o componente deve ser utilizado em várias aplicações, inclusive em algumas não pensadas no momento de seu desenvolvimento. A segunda parte é o uso de componentes para montar novos sistemas. Neste caso um dos maiores problemas é definir como os componentes serão conectados. Uma das soluções utilizadas é a explicitação de uma arquitetura de software.

Na Figura 3.1 o ciclo para o desenvolvimento sistemático de sistemas baseados em componentes definido para este trabalho é ilustrado. A fase de modelagem está presente em quase todos os passos. Como queremos uma maior confiança e flexibilidade explicitamos, propositalmente, esta fase.

De posse dos requisitos, uma arquitetura para o sistema é definida. Esta arquitetura pode ser desenvolvida especificamente para este sistema ou pode ser reusada de projetos anteriores. A especificação da arquitetura é em termos das funcionalidades apontadas na análise de requisitos. Neste ponto, é possível identificar os possíveis componentes que fornecem as funcionalidades. A definição da arquitetura e identificação das funcionalidades fazem parte da modelagem.

Uma busca é realizada para identificar possíveis candidatos. Se mais de um candidato para uma funcionalidade for encontrado, é necessário identificar o que mais se adequa aos outros requisitos não funcionais do sistema. Dificilmente será encontrado um componente que se adeque perfeitamente ao contexto atual. Pode ser necessário alguma adaptação no componente selecionado antes de reusá-lo em um novo projeto. Pode ser possível também que a arquitetura tenha que ser adaptada em função dos componentes selecionados. Existem algumas funcionalidades que podem não apresentar componentes correspondentes na base pesquisada ou, até mesmo, essas funcionalidades serem as que agregam valor ao sistema. Nesse último caso as funcionalidades podem ser o diferencial ou, até mesmo, o segredo comercial do sistema, e devem ser desenvolvidas localmente ao projeto.

O passo de integração consiste em utilizar um arcabouço para a composição do sistema a partir dos componentes. O arcabouço define como e onde os componentes serão inseridos e como eles

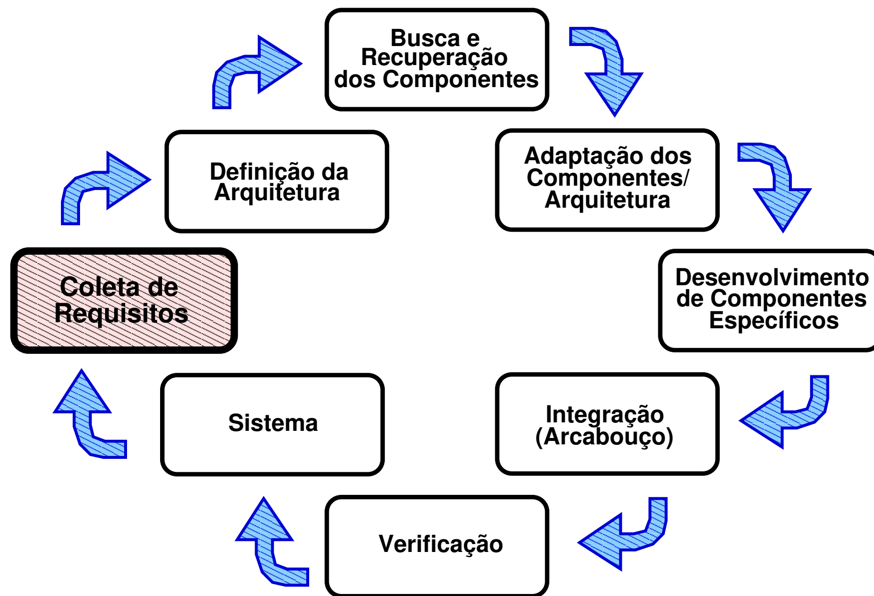


Figura 3.1: Ciclo de vida do desenvolvimento baseado em componentes.

se relacionam entre si. No arcabouço temos a parte que não muda em uma linha de produto como, por exemplo, funcionalidades básicas, mecanismos de comunicação e sincronização. Os passos de recuperação, adaptação, desenvolvimento, e integração são passos explicitamente de modelagem. No contexto deste artigo tais passos serão explicitamente tratados na Seção 3.3. Artefatos como, por exemplo, o código, podem estar associados aos modelos e ao arcabouço. Desta forma, a atividade de modelagem torna-se a mais importante no desenvolvimento dos sistemas, mudando o foco da programação para a modelagem por composição visual de software.

O último passo é o de verificação e pode consistir de vários métodos como, por exemplo, simulação, prova de propriedades e testes. Uma vez verificado o projeto, temos um novo sistema. No caso da não validação pode ser necessária uma mudança na arquitetura ou nos componentes, percorrendo novamente o ciclo ou parte dele. Uma vez que o sistema seja validado, este pode evoluir com a substituição dos componentes por novas versões. O passo de validação está presente na modelagem com a simulação automática do modelo e a prova de propriedades. Testes tradicionais também podem, e devem, ser aplicados ao código.

Mesmo que o sistema seja validado, possivelmente ocorrerão mudanças nos requisitos ao longo do tempo. Por isso a Figura 3.1 representa um ciclo fechado. Ou seja, estamos consi-

derando que um sistema sempre evolui, e esta evolução sempre é tratada pela modelagem do sistema.

O ciclo para o desenvolvimento dos componentes pode ser o mesmo encontrado na engenharia de software tradicional, entretanto uma atenção maior deve ser colocada na fase de requisitos pois estes podem não ser totalmente definidos no momento do desenvolvimento. Um componente deve ser genérico, isso dificulta a definição dos requisitos não apenas funcionais. Além disso, o componente pode ser usado em uma aplicação ainda não pensada no momento de seu desenvolvimento, e deve, portanto, possuir mecanismos de adaptação como, por exemplo, parametrização.

3.3 Modelagem Sistemática

Nesta seção descrevemos conceitualmente uma solução de modelagem baseada em reúso de modelos. Na Figura 3.2 podemos ver um esquema que ilustra a solução introduzida. Além das atividades de reúso também consideramos a verificação de uso. Essa etapa consiste em realizar a verificação de modelos com os modelos integrados para verificar se o caso de uso específico está correto. A etapa de manutenção do repositório, ou seja, recuperação e inserção de modelos neste, são tratadas em [60]. Enquanto que a adaptação é definida em [42].

A técnica de modelagem sistemática, bem como a verificação formal, foram aplicadas anteriormente aos domínios de Sistemas Flexíveis de Manufatura (SFM) [30, 18, 23, 25, 26, 21], e Sistemas Multi-Agentes (SMA) [29, 22, 37].

Em vários ramos da engenharia o processo de reúso de blocos de construção é utilizado para o desenvolvimento mais eficiente de projetos. Com o reúso pode-se conseguir uma economia de recursos como dinheiro e tempo, além de diminuir a possibilidade de erros humanos. Nos últimos anos, estudos em engenharia de software têm sido desenvolvidos na tentativa de definir técnicas e métodos para tornar o processo de desenvolvimento de software similar aos processos de engenharia. Uma das soluções adotadas é conhecida como componentes, que vem sendo discutido neste capítulo. A idéia é que o sistema de software passa a ser desenvolvido a partir de componentes utilizados como blocos de construção. A importância do reúso no desenvolvimento de sistemas complexos é incontestável.

Durante o desenvolvimento de vários projetos, verifica-se que dentro de um contexto específico de aplicação sempre existem características comuns entre esses. Essas características

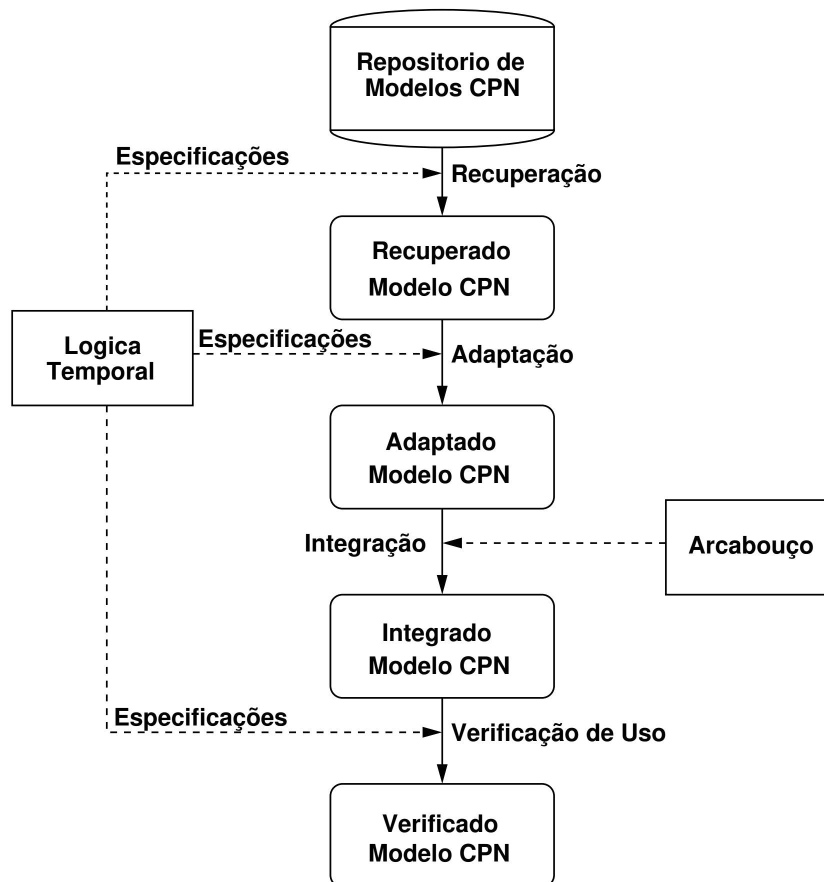


Figura 3.2: Esquema para a solução de reúso de modelos CPN.

podem estar representadas por qualquer artefato tipo trecho de código, ou modelagem em algum formalismo. Especificamente neste trabalho estamos tratando de modelos de redes de Petri coloridas, então essas redes serão os artefatos reusados. É na identificação dessas características dentro de um contexto que o reúso assume um papel importante. Primeiro porque uma vez identificado algo que provavelmente já foi feito pode-se procurar e reutilizar o artefato com algumas adaptações. Segundo porque uma vez desenvolvido algum artefato que pode ser utilizado em outros projetos este pode ser disponibilizado para possíveis reúsos. Além disso, a institucionalização do processo de reúso aumenta a confiança dos artefatos pois a tendência é reusar artefatos provenientes de esforços bem sucedidos de modelagens anteriores.

Uma vez identificada uma parte de um modelo que pode já ter sido desenvolvido, procede-se com a busca desse modelo, que é tratado pela etapa de recuperação ilustrada na Figura 3.2. Uma vez localizado o modelo, é possível que este necessite de adaptações para possibilitar o reúso, o

que é tratado pela segunda etapa do processo de reúso mostrado na Figura 3.2. Uma vez que os modelos estejam prontos para o reúso, o terceiro passo é integrar esses modelos para desenvolver o modelo desejado. Finalmente é necessário fazer a verificação de uso dos modelos integrados, como uma última etapa no processo de reúso para verificar se o reúso não alterou a semântica dos modelos reusados.

Em se tratando de métodos formais, é possível desenvolver métodos automáticos de manipulação. De fato, todos os passos mostrados na Figura 3.2 foram implementados como procedimentos totalmente automáticos.

Para o desenvolvimento deste trabalho foi utilizado o Design/CPN no desenvolvimento e análise dos modelos. Foi utilizada a biblioteca ASK/CTL [7] para a verificação de modelos [47, 9, 10, 11]. As especificações de propriedades para a recuperação, adaptação, e verificação de uso são realizadas em lógica temporal [39]. Enquanto que a integração de modelos é realizada através de funções implementadas para o Design/CPN.

3.3.1 Recuperação

Tendo identificado os elementos do sistema podemos recorrer ao repositório objetivando recuperar candidatos a reúso. Essa identificação é um resultado obtido das fases de requisitos e arquitetura do sistema como mostrado na Figura 3.2. Conforme foi definido em [58, 60], a recuperação consiste em: Escrever propriedades desejadas em lógica temporal; verificar as propriedades descritas contra um meta-modelo (repositório); se mais de um candidato for encontrado avaliar através de simulação e descrições o seu comportamento e características não funcionais para decidir o melhor candidato ao contexto atual. Caso nenhum candidato seja encontrado o modelo deve ser desenvolvido.

Após a recuperação dos modelos das entidades o próximo passo poderá ser a integração desses modelos ao arcabouço, ou a adaptação de um ou mais desses modelos. Note que podemos fazer a recuperação de todos os modelos antes de prosseguir com a integração ou adaptação, recuperar um modelo e integrá-lo em seguida antes de recuperar o próximo. Esta decisão fica a cargo do projetista não fazendo diferença para a solução apresentada. Opcionalmente pode ser realizada a adaptação dos modelos recuperados.

3.3.2 Adaptação

Quando um modelo é recuperado do repositório ele pode não satisfazer totalmente as necessidades específicas de um domínio. Por exemplo, um modelo recuperado pode ter o comportamento de um pilha, mas de uma pilha com capacidade infinita. Em um projeto pode ser necessária uma pilha com capacidade de dois elementos. O modelo foi recuperado do repositório com o uso de especificações formais em lógica temporal mas ainda assim pode necessitar de adaptações para satisfazer um caso de uso específico.

A técnica de adaptação introduzida, definida, e desenvolvida em [43, 44, 42] é utilizada com o objetivo de realizar adaptações automáticas em modelos de redes de Petri coloridas. Essa técnica consiste em: escrever propriedades em lógica temporal, que são restrições de comportamento desejadas no modelo; verificar se o modelo pode ser adaptado de maneira a satisfazer tais propriedades; em caso positivo adaptar o modelo em questão.

O primeiro passo está relacionado com a necessidade do projetista especificar, utilizando a linguagem ASK-CTL, quais são as restrições que devem ser impostas ao modelo a ser utilizado. A verificação é realizada com verificação de modelos, como na recuperação. No caso da possibilidade de adaptação, esta é realizada através de funções implementadas no Design/CPN em CPN/ML [6].

3.4 Integração de Modelos

Após um modelo ser recuperado do repositório, e possivelmente adaptado, este precisa ser integrado ao arcabouço. A integração, bem como as outras atividades, foi completamente implementada utilizando a linguagem [6] do Design/CPN.

Inicialmente, o projetista precisa indicar o nome do arquivo com o modelo CPN a ser integrado. Então, algumas funções são automaticamente executadas para criar o ambiente de integração, isto é, os lugares, transições, arcos, e seus respectivos nomes, conjunto de cores, e inscrições. O próximo passo executado pelo algoritmo é a definição das portas de entrada e saída no diagrama a ser integrado. Após este passo, a transição de substituição será definida, e os sockets na super-página são associados às respectivas portas, anteriormente definidas na sub-página. O último passo é selecionar a caixa com as declarações do modelo, na página do modelo, para

definir as cores das portas baseado nas cores dos sockets, e adicionar esta informação no nó de declarações globais.

A seleção do arquivo precisa da interação com o usuário, enquanto todos os outros passos são executados automaticamente. Para definir o algoritmo de integração, algumas restrições têm que ser consideradas, a saber:

- Nome de página único;
- Prefixo no nome do conjunto de cor indica o nome da página;
- Sufixo no nome dos lugares indica se este é porta ou não;
- Padrão de linha *Dot-dashed* deve ser aplicado à caixa auxiliar com as declarações do modelo;
- Caixa de declarações do modelo deve ser única;
- Declarações dos lugares que são porta devem estar nas primeiras linhas da caixa de declarações do modelo.

A primeira restrição a ser considerada é para garantir que o nome da página do modelo a ser integrado é único, e é de responsabilidade do projetista garantir isto. Os nomes dos conjuntos de cores devem ter um prefixo com o nome da página. Outra restrição é a respeito dos lugares porta. Os lugares que são portas devem ter um sufixo IN ou OUT no nome para portas de entrada e saída, respectivamente. Esta restrição é para permitir que o algoritmo possa reconhecer que lugares são portas e que tipo de porta eles são.

A última restrição de integração é que no modelo a ser integrado deve existir uma caixa auxiliar com as declarações da página e esta caixa deve ter o padrão de linha *dot-dashed*. As declarações do conjunto de cores dos lugares porta devem ser as primeiras nesta caixa. Isto é necessário para que o algoritmo possa adaptar corretamente os conjuntos de cores para integrar o modelo com sucesso

É importante notar que é responsabilidade do projetista garantir que as restrições são respeitadas para que o algoritmo possa funcionar corretamente.

Implementação

A integração é implementada como definida no Algoritmo 1. Os passos de 2 até 7 são totalmente automáticos. O passo 1 necessita da interação com o usuário para selecionar o nome do arquivo como o modelo CPN a ser integrado.

Algoritmo 1 Integração de Modelos

1: Seleciona o arquivo do modelo;
2: Cria lugares, transições e arcos;
3: Define as portas;
4: Define as transições de substituição;
5: Associa os sockets às portas;
6: Adapta as cores dos lugares às cores dos sockets;
7: Insere as declarações do modelo no nó de declarações global.

O passo de integração depende do arcabouço. Portanto, para cada domínio de aplicação é necessário definir a arquitetura do sistema e modelá-la, o que chamamos de arcabouço CPN, ou seja, um modelo CPN onde integramos os modelos dos componentes. Além disso, para cada arcabouço, é necessário implementar funções específicas para o passo de integração do processo de reúso. Mas esta implementação precisa ser feita apenas uma vez, e é utilizada através da evolução da linha de produto em um domínio específico de aplicação.

3.5 Verificação de Uso

Além dos passos de recuperação, adaptação, e integração de modelos, também é considerado neste trabalho um passo de verificação de uso. Esta atividade é considerada no contexto deste trabalho porque quando se modela baseado em reúso é necessário garantir que a semântica do modelo resultante não viola a semântica dos modelos reusados. Algumas partes do modelo resultante pode levar o modelo reusado a se comportar de forma diferente da esperada. Este problema pode comprometer a atividade de modelagem, e a facilidade, e flexibilidade que o processo de reúso promove.

A atividade de verificação de uso consiste em realizar verificação de modelos no arcabouço com os modelos de componentes individuais a serem verificados já integrados neste. Para isto, é

especificado em um arquivo, as fórmulas em lógica temporal para as propriedades de um modelo. A verificação de modelos é realizada no modelo resultante completo para garantir que os modelos de componentes foram utilizados corretamente.

Observe novamente que é necessário definir um arcabouço para cada domínio de aplicação. O arcabouço pode ser desenvolvido sem nenhuma consideração a respeito das funcionalidades dos componentes. Além disso, os modelos podem ser reusados em vários domínios diferentes. A interface entre o arcabouço e os modelos pode mudar para refletir as necessidades de cada domínio e para satisfazer casos de uso alternativos de cada modelo. Portanto, a medida que o arcabouço é desenvolvido as mudanças na interface podem resultar em um reúso errado para o modelo integrado. Desta forma, é necessário definir uma atividade de verificação de uso para o processo de reúso. A atividade de verificação de uso é essencial quando se desenvolvem modelos a partir de um novo arcabouço.

Outra justificativa para a definição da atividade de verificação de uso é que no caso em que nenhum candidato a reúso é encontrado no repositório, um novo modelo deve ser desenvolvido. A verificação de uso também pode ser aplicada para validar um novo modelo para ser inserido no repositório para futuros reúsos.

Implementação

No Algoritmo 2 a verificação de uso é definida. Inicialmente, o projetista deve executar a ferramenta de grafo de ocorrência no Design/CPN. O próximo passo é selecionar o nome do arquivo que contém as especificações de propriedades para o modelo a ser verificado. O algoritmo executa neste ponto a verificação de modelos no modelo do arcabouço com os modelos já integrados neste. Se as propriedades são satisfeitas no modelo resultante a verificação de uso é dita com sucesso. No caso de erro, este não é apresentado, nem corrigido, pelo procedimento de verificação de uso. Portanto, é responsabilidade do projetista corrigir os erros.

Os passos de 1 a 3 devem ser realizados pelo projetista. Todos os outros são executados automaticamente baseados nas especificações contidas no arquivo indicado pelo projetista no passo 3. Após a execução, uma mensagem é mostrada dizendo se as propriedades foram satisfeitas ou não.

Algoritmo 2 Verificação de Uso

1: Especificar propriedades a serem verificadas
2: Executar a ferramenta de grafo de ocorrência
3: Selecionar o arquivo com as especificações
4: Apagar algum grafo de ocorrência existente
5: Calcular grafo de ocorrência
6: Calcular grafo de componentes fortemente conectados
7: Ler biblioteca ASK-CTL
8: Executar verificação de modelos
9: **se** Propriedades foram satisfeitas **então**
10: Mostra mensagem SUCESSO
11: **senão**
12: Mostra mensagem FALHA
13: **fim se**

3.6 Conclusões

O desenvolvimento baseado em componentes necessita de suporte para o desenvolvimento e gerenciamento de uma base de componentes reutilizáveis. Além disso, uma forma para integração de componentes deve ser definida. Algumas questões como ferramenta de composição visual, certificação e classificação de componentes, entre outras também devem ser tratadas. Neste trabalho, abordamos essas questões com uma solução de modelagem sistemática e formal de sistemas baseados em componentes.

Para a modelagem sistemática utilizamos um processo de reúso de modelos de redes de Petri. Então a modelagem não precisa ser desenvolvida do zero, mas utilizando modelos de componentes armazenados em um meta-modelo e um arcabouço para a integração destes. A verificação de propriedades, com simulação ou verificação de modelos, permite uma maior confiança no funcionamento do modelo. Além disso, no meta-modelo temos a formalização dos contratos dos componentes, separados dos modelos. Com isso podemos realizar verificação dos contratos, além de prever o comportamento do sistema.

Suportar o desenvolvimento de sistemas baseados em componentes com uma solução sistemática e métodos formais é uma contribuição para a consolidação da disciplina de engenharia

de software baseada em componentes.

É necessário, ainda, o desenvolvimento de técnicas, métodos e ferramentas para a verificação modular de propriedades. Com raciocínio modular não precisamos raciocinar sobre todo o modelo, mas apenas nos modelos dos componentes individuais. O uso de técnicas de raciocínio modular permite tratar o problema da explosão de espaço de estados.

Finalmente um domínio de aplicação deve ser escolhido. A escolha de um domínio é importante para a manutenção da base de modelos de componentes reutilizáveis, e para o desenvolvimento de um arcabouço para o desenvolvimento de uma família de sistemas. No contexto deste trabalho o domínio escolhido foi sistemas embarcados [57]. No Capítulo 4 um estudo de caso para uma rede de transdutores é apresentado.

Capítulo 4

Estudo de Caso: Rede de Transdutores

Nos últimos anos, os sistemas embarcados [55, 57] têm sido aplicados nos mais diversos tipos de dispositivos [64]. Com a evolução da tecnologia os dispositivos têm uma capacidade cada vez maior de realização de tarefas e conseqüentemente os sistemas embarcados têm se tornado cada vez mais complexos [55]. Neste Capítulo descrevemos uma aplicação do processo apresentado no Capítulo 3 ao domínio de sistemas embarcados. Mais especificamente apresentamos a especificação e verificação de uma rede de transdutores inteligentes [20, 27, 28].

O uso de componentes para desenvolvimento de sistemas embarcados [41] permite tratar a crescente complexidade como tem sido feito em outros domínios. No caso deste estudo de caso estamos interessados especificamente na especificação e verificação formal de sistemas embarcados baseados em componentes para redes de transdutores.

O processo de desenvolvimento empregado em sistemas embarcados é, geralmente, escrever código específico para cada aplicação. Com a aplicação de conceitos como linha de produtos, arquitetura de software, e componentes, conforme descrito no Capítulo 3, é possível concretizar um processo de desenvolvimento onde os sistemas são *montados* ao redor de um arcabouço. Essa montagem é feita reutilizando componentes para entidades ou funcionalidades comuns a vários projetos, barateando e diminuindo o tempo do processo de desenvolvimento, além de reduzir a possibilidade da ocorrência de erros humanos. De fato um processo de desenvolvimento centrado na especificação e verificação formal de sistemas baseados em componentes é descrito em um trabalho anterior [19, 24].

O uso de modelos formais agrega algumas vantagens ao processo de desenvolvimento como

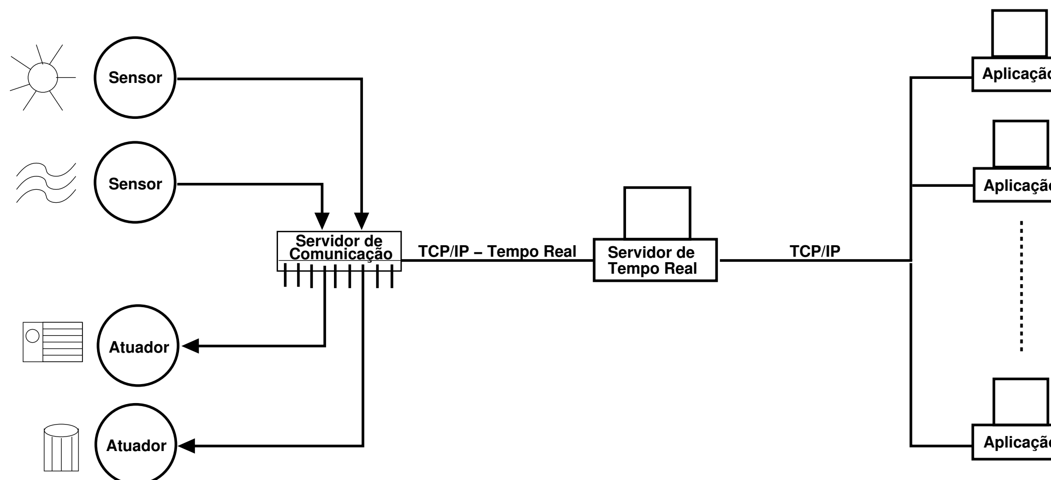


Figura 4.1: Topologia do sistema.

tornar possível verificar propriedades nos modelos antes da implementação do sistema, detecção de erros lógicos de projetos, e simulação dos modelos. Neste projeto utilizamos Redes de Petri Coloridas Hierárquicas (*HCPN - Hierarchical Coloured Petri Nets*) [50, 51], e o conjunto de ferramentas para edição e análise dos modelos chamado Design/CPN [49]. No processo de verificação utilizamos ainda a biblioteca ASK/CTL [7] para realizar a verificação de modelos [11].

Os objetivos neste capítulo são aplicar métodos e técnicas para especificação e verificação formal de sistemas embarcados utilizando conceitos de componentes e raciocínio modular. Uma aplicação no domínio de sistemas embarcados para controle de redes de transdutores é apresentada para ilustrar a técnica desenvolvida neste trabalho.

4.1 O Sistema de Controle de Redes de Transdutores

O domínio de aplicação considerado neste capítulo são as redes de transdutores. Este sistema é composto de transdutores inteligentes, um controlador, e um servidor de tempo real. Além disso há aplicações acessando o servidor de tempo real. Na Figura 4.1 podemos ver um exemplo de topologia do sistema em questão [67].

Os transdutores inteligentes são sensores e atuadores com um microcontrolador acoplado para realização de tarefas locais ao dispositivo como, por exemplo, acesso a um canal de comunicação utilizando um protocolo específico. Estes estão conectados a um servidor de comunicação (SC). Os sinais dos sensores são transformados e controlados pelo SC de forma que o servidor de

tempo real possa acessar e modificar a informação para controlar os atuadores de acordo com as aplicações. A forma como isto é feito foi descrita em [67] e permite a definição de uma arquitetura para este tipo de aplicação de forma que diferentes aplicações sejam especificadas e verificadas apenas trocando os componentes dependentes de aplicação.

Conforme citado na introdução, conceitos como linha de produtos, arquitetura de software, e componentes são utilizados para estabelecer um processo de desenvolvimento de aplicações em um domínio bem definido. Neste processo é utilizado como base um arcabouço, que é a realização de uma arquitetura para uma linha de produtos levando em conta padrões de projeto [40]. Os componentes são então reusados em projetos diferentes dentro do domínio de aplicação.

No contexto específico deste trabalho estamos abordando o domínio de aplicação dos sistemas de controle para redes de sensores e atuadores. Podemos ter várias aplicações diferentes para o controle de uma rede de sensores e atuadores com configurações diferentes dos dispositivos. Por exemplo podemos ter um sensor de temperatura e um de umidade. Os sinais desses sensores podem ser utilizados para controlar um condicionador e um desumidificador de ar.

Dependendo da aplicação esses atuadores podem ter intervalos diferentes de atuação. Esse tipo de controle é realizado pelas aplicações configurando o valor do ponto de chaveamento do atuador dependendo do valor emitido pelo sensor. Esse valor será modificado no servidor de tempo real, e se refletirá no controle efetuado pelo SC nos atuadores.

De acordo com as aplicações, os sensores, e atuadores utilizados, sistemas diferentes precisam ser especificados e verificados, mas apenas parte do sistema como um todo necessita de mudanças. Com isso é possível definir uma linha de produto para este domínio específico, utilizando uma arquitetura para reusar componentes dentro deste domínio. Desenvolver sistemas seguindo este tipo de estratégia traz vantagens como diminuição de recursos como tempo e dinheiro, e redução da possibilidade de erros humanos. Além disso é possível manter e evoluir um repositório de componentes reusáveis dentro de um domínio e até mesmo em outros domínios, aumentando a confiança no funcionamento dos modelos que são utilizados com sucesso em vários projetos subseqüentes.

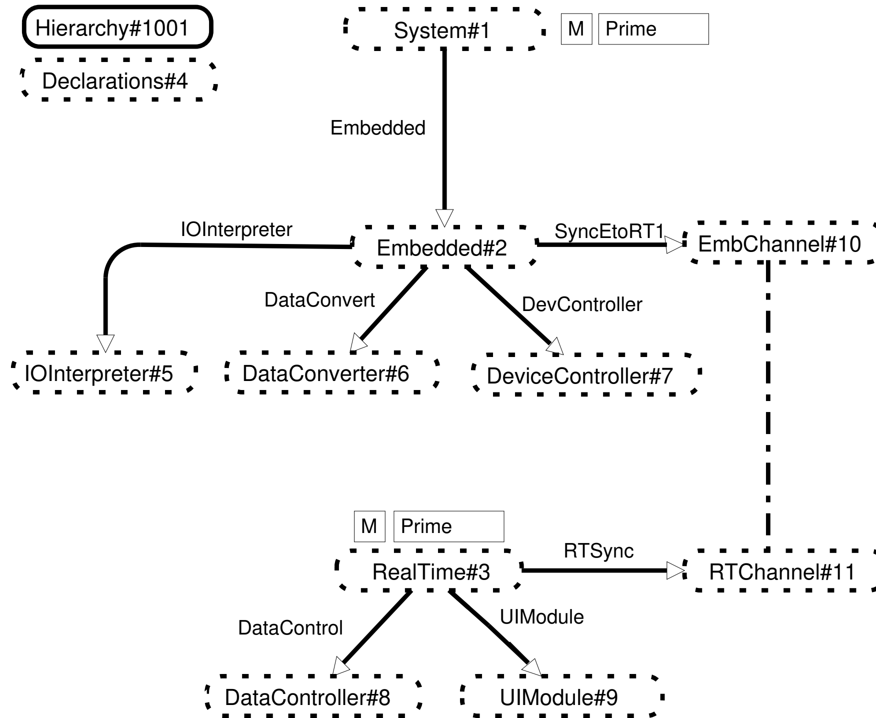


Figura 4.2: Hierarquia do modelo.

4.2 Especificação e Verificação

Nesta Seção apresentamos a especificação e verificação formal do sistema embarcado descrito na Seção 4.1 utilizando HCPN e o processo descrito no Capítulo 3.

Uma observação a respeito da especificação e deste trabalho é que abstraímos detalhes de tecnologias específicas para cada componente. Desta forma estamos mais interessados na especificação e verificação da arquitetura do sistema. Isto é devido aos objetivos deste trabalho no que diz respeito a técnicas de raciocínio modular. Portanto estaremos verificando propriedades a nível de interface de componentes e arquitetura, sem nos preocuparmos com detalhes internos dos componentes como, por exemplo, o protocolo utilizado para que os sensores e atuadores se comuniquem com o sistema embarcado.

4.2.1 Especificação

Na Figura 4.2 podemos ver a hierarquia HCPN que especifica a arquitetura para o sistema em questão. Nesta figura podemos ver como as entidades se relacionam entre si de acordo com o que foi apresentado na Figura 4.1. Na página System temos a representação dos sensores e atuadores.

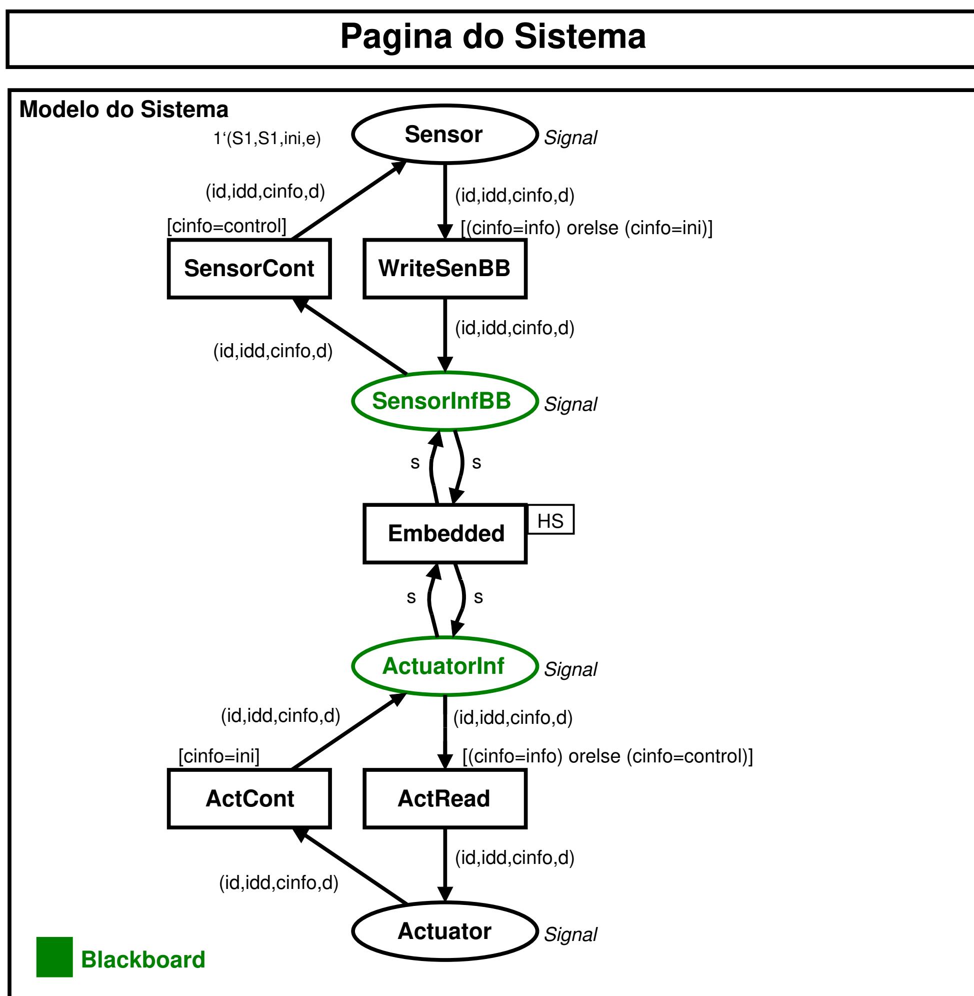


Figura 4.3: Página do modelo dos sensores e atuadores.

Estes se relacionam apenas com o sistema embarcado representado pela página Embedded. Na Figura 4.3 podemos ver a página do sistema de sensores e atuadores.

O sistema embarcado, ilustrado na Figura 4.4, é composto de vários componentes. A página IOInterpreter representa o interpretador de entrada e saída do sistema embarcado. A entrada de dados dos sensores no sistema embarcado e a saída para os atuadores é realizada através do que chamamos de Blackboard. O interpretador de entrada e saída serve para instanciar como objetos os dados escritos no Blackboard pelos sensores para serem utilizados pelo resto do sistema. Além disso, é este componente que recebe os dados em forma de objetos do sistema e os transforma para dados na forma a ser escrita no Blackboard para serem compreendidos pelos atuadores. Na Figura 4.5 podemos ver a página que representa o interpretador de entrada e saída.

O interpretador de entrada e saída é fixo na arquitetura. Isto significa que não é necessário

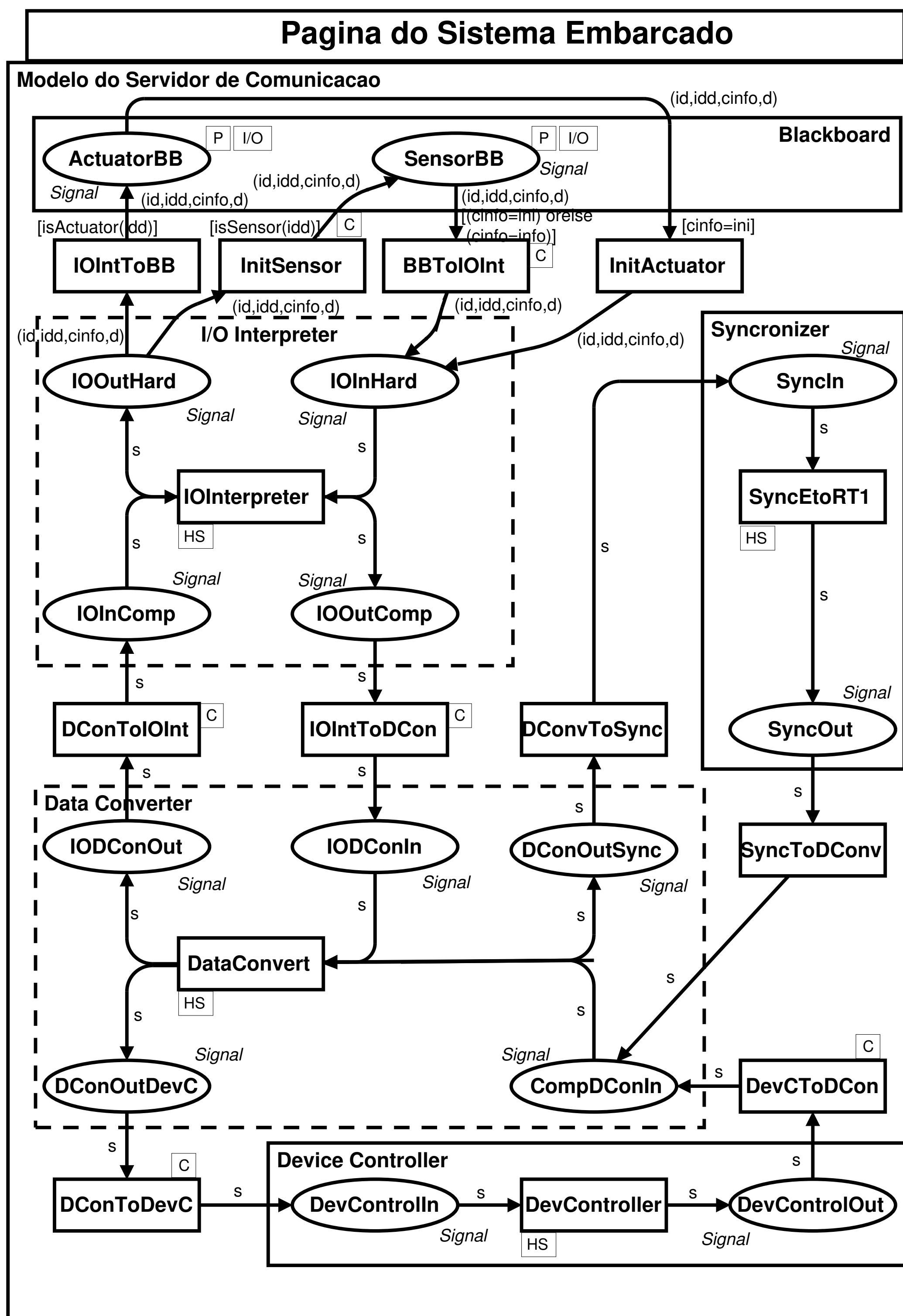


Figura 4.4: Página do modelo do servidor de comunicação.

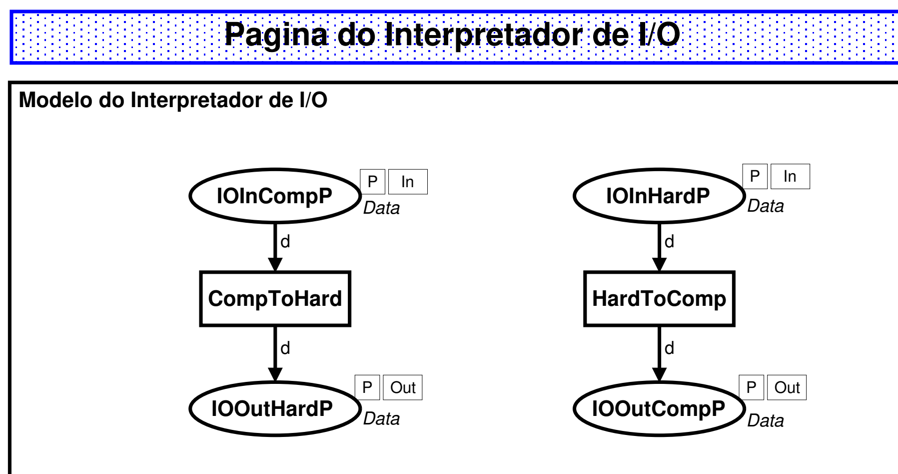


Figura 4.5: Página do modelo do interpretador de entrada e saída.

modificá-lo de uma aplicação para outra. Chamamos os componentes fixos da arquitetura de pontos frios, enquanto os que requerem modificações dependentes da aplicação são chamados de pontos quentes.

Depois do interpretador de entrada e saída temos o conversor de dados, que na hierarquia é representado pela página DataConverter. Este componente transforma os dados do interpretador para o formato esperado pelo servidor de tempo real, de acordo com as aplicações que o acessam. Como o formato dos dados são dependentes das aplicações que acessam o servidor, este componente é um ponto quente da arquitetura e requer modificações específicas de acordo com a aplicação. Na Figura 4.6 podemos ver a página do conversor de dados.

No conversor de dados também é realizada um decisão com relação ao fluxo de dados no sistema. Se o dado presente no conversor em um dado momento é uma requisição de controle como, por exemplo, inicialização ou calibração dos dispositivos, esse dado é enviado para o controlador de dispositivos. Caso o dado seja informação, então este é enviado para o sincronizador para ser transmitido para o servidor de tempo real.

O controlador de dispositivos, representado pela página DeviceController na hierarquia pode ser visto na Figura 4.7. Este componente realiza o controle dos dispositivos, tanto dos sensores como dos atuadores. Quando um sensor ou atuador entra na rede, ou quando esta é inicializada, os dispositivos enviam um sinal de controle requisitando sua inicialização. No controlador de dispositivos é especificado o procedimento de controle para cada dispositivo na rede. Desta forma ele deve ser modificado para cada aplicação, dependente dos dispositivos a serem utilizados na

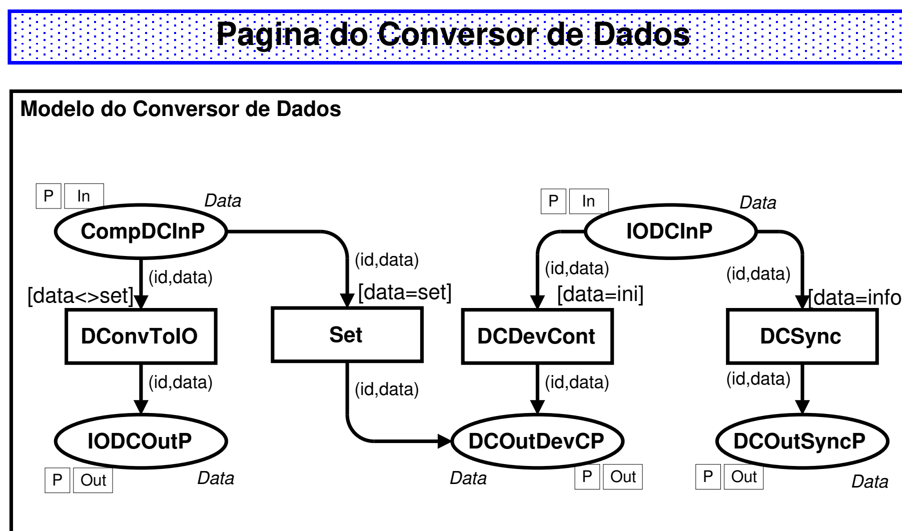


Figura 4.6: Página do modelo do conversor de dados.

rede. Por isso podemos considerá-lo um ponto quente da arquitetura. Por outro lado, este componente pode ser considerado um ponto frio, se considerarmos que o procedimento de inicialização dos sensores a atuadores são os mesmo para todos os dispositivos, sendo diferente apenas os dados necessários para realizar o procedimento. Dessa forma apenas uma base de dados, ou tabela deve ser atualizada. Este componente pode ser responsável também por outros procedimentos de controle como, por exemplo, calibração, ajuste de ponto de atuação e chaveamento, entre outros.

O sincronizador é a realização da comunicação entre o sistema embarcado e o servidor de tempo real através de um protocolo TCP/IP de tempo real. Quando o sensor envia um dado que é uma informação e não um dado de controle, essa deve ser transmitida para o servidor de tempo real, e o caminho para essa comunicação é o sincronizador. Então existe um sincronizador no sistema embarcado e outro idêntico no servidor. Como essa comunicação não muda, este componente é um ponto frio da arquitetura. A única possibilidade de mudança seria a do protocolo de comunicação e como isso não é uma mudança freqüente não consideramos esse componente um ponto quente da arquitetura. Os dois componentes sincronizadores, um no sistema embarcado e outro no servidor, formam um canal *full-duplex* de comunicação. Na Figura 4.8 podemos ver o sincronizador do sistema embarcado, que é representado na hierarquia pela página EmbChannel. Como o sincronizador do servidor é idêntico não será mostrado.

O servidor de tempo real realiza a intermediação entre o sistema embarcado e as aplicações. Para isso deve manter uma base de dados com informações sobre a rede e sobre as aplicações. As

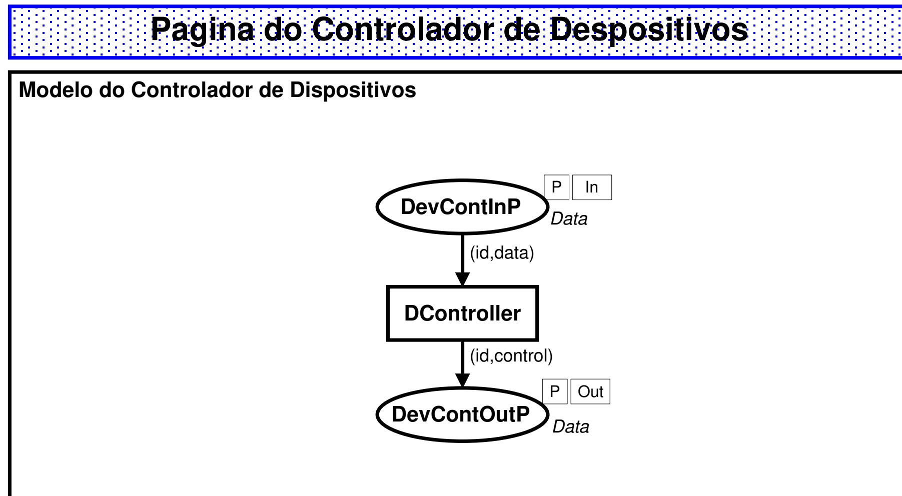


Figura 4.7: Página do modelo do controlador de dispositivos.

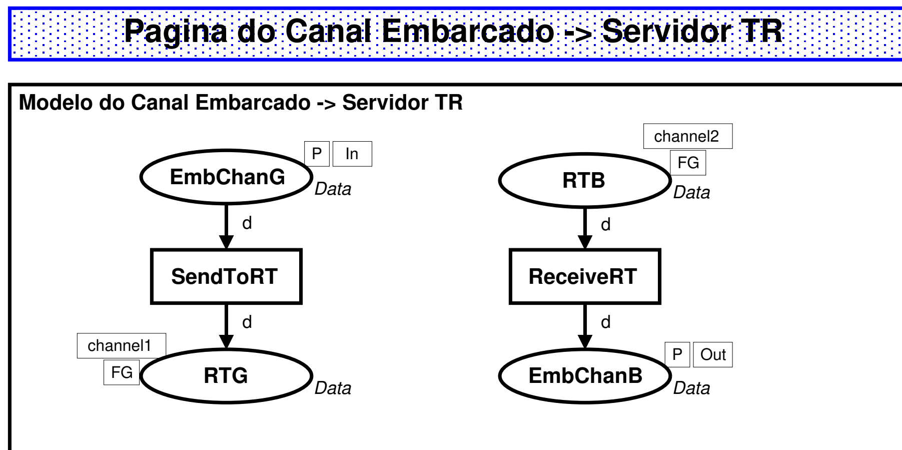


Figura 4.8: Página do modelo do sincronizador.

aplicações podem ler ou modificar essas informações para monitorar e/ou controlar o sistema. No servidor temos vários componentes, como pode ser visto na Figura 4.9.

Como já explicamos anteriormente o componente sincronizador é idêntico ao do sistema embarcado e já foi comentado e mostrado na Figura 4.8. O controlador de dados, representado na hierarquia pela página DataController pode ser visto na Figura 4.10. Este componente controla o fluxo dos dados de e para as aplicações, e decide que dados pertencem a que aplicações. Além disso, recebe os dados das aplicações como, por exemplo, modificações para o controle dos dispositivos, e os envia para o servidor de tempo real.

Finalmente, o componente de interface com usuário disponibiliza os serviços para que as aplicações possam acessar o sistema. Na hierarquia este componente é representado pela página

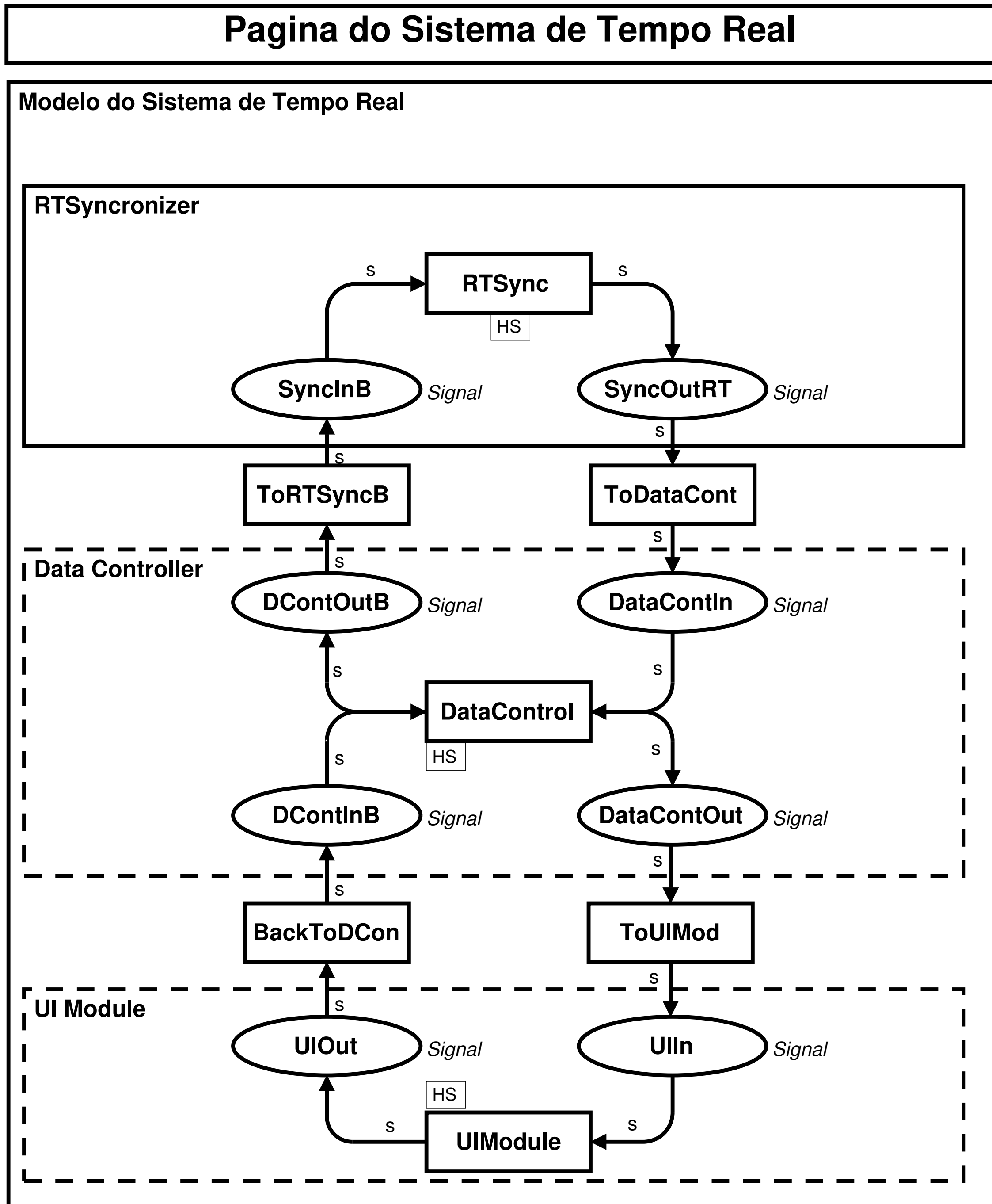


Figura 4.9: Página do modelo do servidor de tempo real.

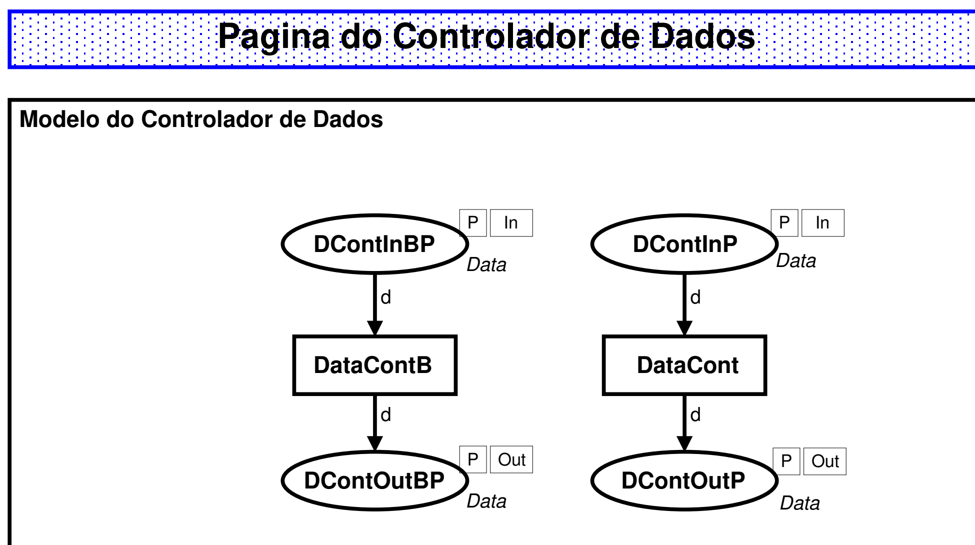


Figura 4.10: Página do modelo do controlador de dados.

UIModule e pode ser visto na Figura 4.11.

Alguns modelos de componentes mostrados na figura são simples devido ao fato de que queremos apresentar a arquitetura em si, e não o funcionamento detalhado dos componentes específicos. Modelos mais realisticos de componentes podem ser inseridos seguindo o processo descrito no Capítulo 3 de acordo com cada projeto específico. Com esta arquitetura podemos especificar qualquer tipo de sistema de controle de redes de transdutores, permitindo a evolução de uma linha de produto baseada no reúso de modelos de componentes através de uma arquitetura e um processo bem definidos. Os transdutores são sensores e atuadores inteligentes. Esses dispositivos são chamados de inteligentes pois são acoplados a um microcontrolador que permite um pré-processamento do sinal local ao dispositivo permitindo a esses, entre outras funcionalidades, a conexão em barramentos através de algum protocolo específico de comunicação.

4.2.2 Verificação

Nesta seção utilizamos os modelos apresentados neste capítulo com o objetivo de provar propriedades utilizando verificação de modelos. Uma tarefa importante nesta atividade é a identificação e definição das propriedades a serem provadas. Como nossa preocupação é em provar propriedades a nível de arquitetura e interface dos componentes podemos utilizar o modelo do arcabouço, sem detalhes internos dos componentes para provar propriedades do sistema como um todo. Para aju-

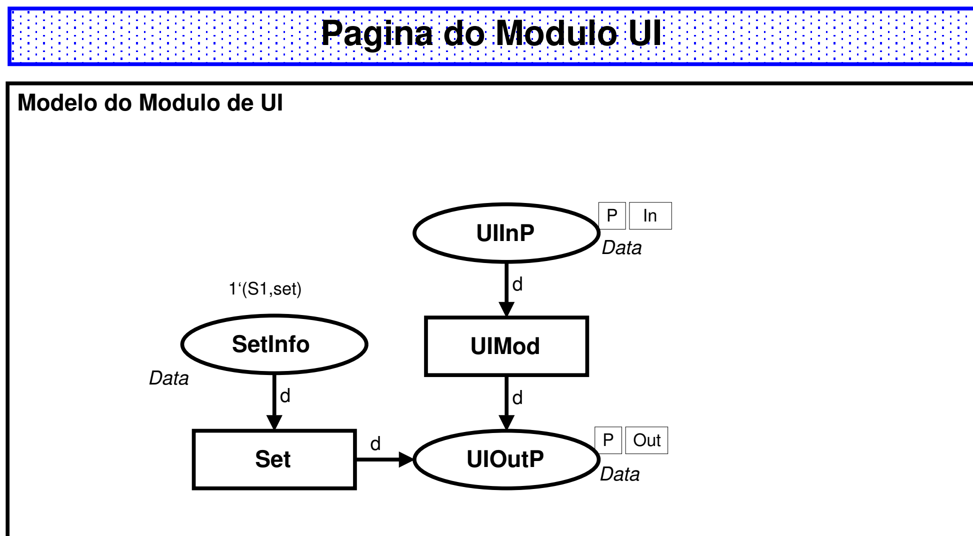


Figura 4.11: Página do modelo do módulo de interface com o usuário.

dar nesta tarefa seguimos uma técnica de identificar cenários de funcionamento do sistema para identificar propriedades interessantes a serem provadas. Para a ilustração dos cenários utilizamos os diagramas de seqüência de mensagens chamados MSC (*Message Sequence Charts*). Esses diagramas são gerados automaticamente com a ferramenta Design/CPN através de simulações.

A estratégia de verificação desenvolvida e utilizada neste trabalho é realizada seguindo a seqüência de passos a seguir:

1. Definição dos cenários;
2. Geração de MSC para os cenários;
3. Identificação das propriedades a serem provadas;
4. Especificação das propriedades e fórmulas em lógica temporal;
5. Verificação de modelos.

A etapa de definição dos cenários é uma etapa conceitual e depende do conhecimento do projetista sobre o projeto. Nesta etapa o projetista deve definir os comportamentos esperados para o modelo para as várias formas de funcionamento. Após esta fase inicial o projetista pode prosseguir com a geração dos MSCs para cada cenário, que é realizada automaticamente pela ferramenta através de simulações do modelo. Utilizando os MSCs é possível identificar que propriedades específicas necessitam ser verificadas para provar o funcionamento de cada cenário. A próxima

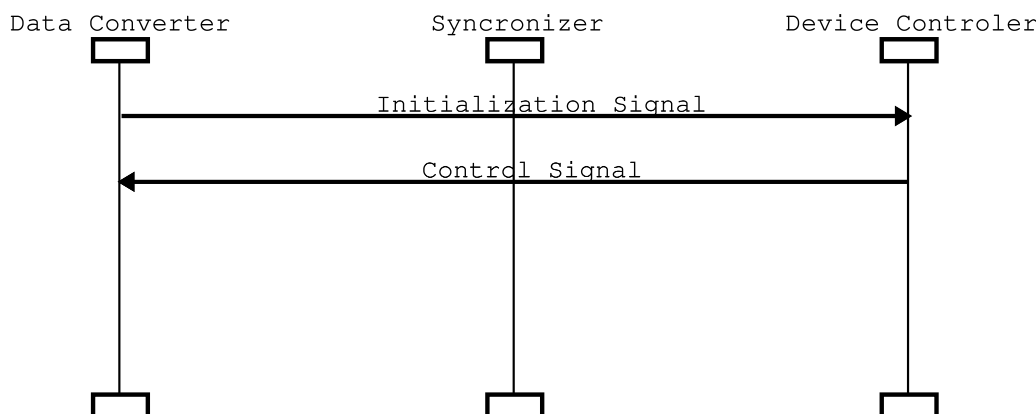


Figura 4.12: MSC para o fluxo do conversor de dados para controle.

etapa é a especificação das proposições atômicas e fórmulas em lógica temporal para a realização da verificação de modelos, que é a próxima e última etapa. Existe a possibilidade de geração automática de cenários onde a primeira etapa não é realizada pelo projetista e sim por simulação do modelo. Esta possibilidade consiste em gerar MSCs aleatoriamente permitindo ao projetista ter uma representação dos cenários proveniente do próprio funcionamento do modelo, sem que este tenha que definí-los pessoalmente. Existe sempre a possibilidade de nem todos os cenários de funcionamento serem definidos pelo projetista ou gerados automaticamente pela ferramenta baseada no modelo. Esta é uma linha de pesquisa na área de testes de software e está fora do escopo deste trabalho.

Suponha que os dispositivos enviem uma mensagem inicial quando o sistema é ligado, ou um novo dispositivo é adicionado. Suponha ainda que o sistema embarcado realize alguma tarefa quando recebe este tipo de mensagem e envia alguma outra mensagem de reconhecimento ou calibração, ou inicialização para o dispositivo. Este tipo de cenário pode ser visto na Figura 4.12.

Quando um dispositivo envia um informação de inicialização, o conversor de dados envia esta informação para o controlador de dispositivos para que este realize as tarefas de controle e calibração e o que mais seja necessário a nível de controle dos dispositivos. Mas este diagrama é baseado em simulação, o que significa que ele é somente uma possibilidade de execução do modelo. Podem existir situações em que este fluxo, que é o esperado, seja violado. Devemos então realizar a verificação de modelos para garantir que este será, sempre, o fluxo realizado

quando o sinal de inicialização for enviado por um dispositivo. No trecho de código a seguir podemos ver as proposições e a fórmula, em lógica temporal, para realizar a verificação desta propriedade.

```
fun PA n = (Mark.Embedded'IODConIn 1 n =
           ((1,(S1,S1,ini,e))!!empty));
fun PB n = (Mark.Embedded'DConOutDevC 1 n =
           ((1,(S1,S1,ini,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

A proposição PA será avaliada como verdadeira se existir uma ficha no lugar IODConIn da página Embedded na hierarquia. A proposição PB será avaliada como verdadeira se existir uma ficha no lugar DConOutDevC da página Embedded na hierarquia. A formula será, então, avaliada como verdadeira se tivermos PA avaliada como verdadeira e com certeza PB será avaliada como verdadeira. Desta forma se tivermos uma ficha na entrada do conversor de dados, esta ficha será encaminhada para a entrada do controlador de dispositivos, que é o componente que realiza as tarefas de controle como, por exemplo, inicialização, calibração, e alteração de parâmetros de funcionamento dos dispositivos. A avaliação desta fórmula como verdadeira significa que esta parte do modelo se comporta corretamente para todas as possibilidades, ou seja, sempre que uma ficha representando um sinal de inicialização aparecer na entrada do conversor de dados, esta será encaminhada para o controlador de dispositivos.

Mas falta ainda provar a segunda parte do fluxo, a volta do sinal de resposta do controlador de dispositivos para o dispositivo. O mesmo raciocínio descrito para a primeira parte do fluxo se aplica aqui, e a as proposições e a fórmula para esta verificação pode ser vista a seguir.

```
fun PA n = (Mark.Embedded'CompDConIn 1 n =
           ((1,(S1,S1,control,e))!!empty));
fun PB n = (Mark.Embedded'IODConOut 1 n =
           ((1,(S1,S1,control,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

Suponhamos agora que o sinal enviado seja um sinal de informação como uma medição de um sensor. Esta informação precisa ser encaminhada para o sincronizador para ser enviada ao

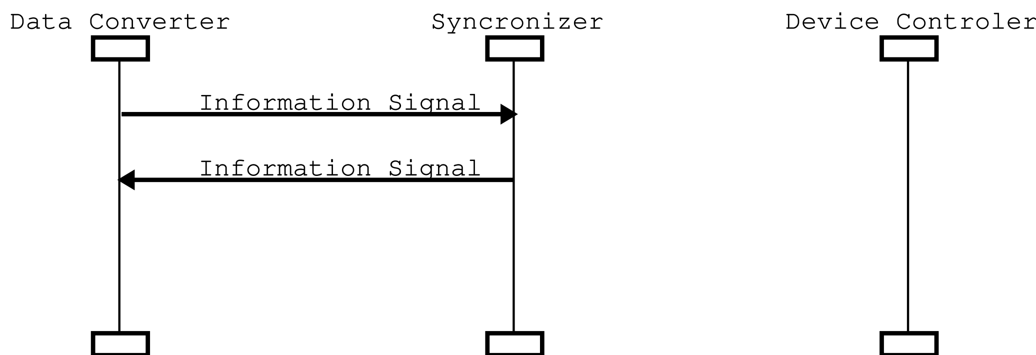


Figura 4.13: MSC para o fluxo do conversor de dados para informação.

servidor de tempo real. Na Figura 4.13 podemos ver o MSC para este cenário. Neste cenário a informação segue pelo sincronizador até o servidor de tempo real para ser tratada pela aplicação de controle do sinal em questão, dependendo do sensor. Em seguida a informação retorna ao sistema embarcado, também pelo sincronizador, para ser disponibilizada para algum atuador, se necessário, ainda dependendo da aplicação em questão.

As proposições e a fórmula para verificar este cenário podem ser vistas a seguir. A proposição PA será avaliada como verdadeira se existir uma ficha no lugar IODConIn da página Embedded na hierarquia, que é o lugar de entrada do conversor de dados. Enquanto que PA será avaliada como verdadeira se existir uma ficha no lugar DConOutSync da página Embedded na hierarquia. Portanto, a fórmula será avaliada como verdadeira se PA for avaliada como verdadeira e PB for verdadeira com certeza em algum momento no futuro. Desta forma estamos provando que em qualquer situação de funcionamento do modelo, sempre que tivermos um ficha com informação esta será encaminhada para o sincronizador para ser transmitida ao servidor de tempo real.

```
fun PA n = (Mark.Embedded'IODConIn 1 n =
           ((1,(S1,A1,info,e))!!empty));
fun PB n = (Mark.Embedded'DConOutSync 1 n =
           ((1,(S1,A1,info,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

De forma complementar devemos provar que toda informação que chegar na entrada do conversor de dados vinda do controlador de dispositivos ou do sincronizador será encaminhada para

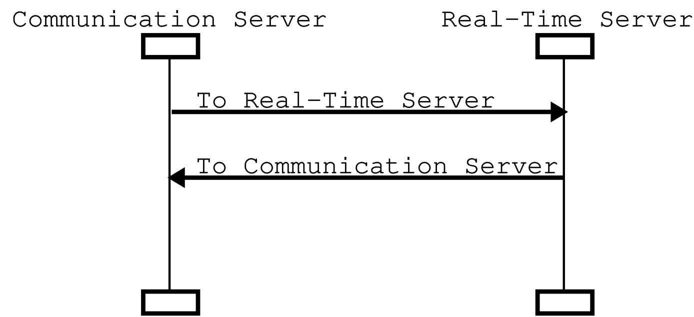


Figura 4.14: MSC para o fluxo no sincronizador.

a região de memória do microcontrolador que é acessada pelos dispositivos. As proposições e fórmulas para a realização desta prova são mostradas a seguir.

```
fun PA n = (Mark.Embedded'CompDConIn 1 n =
           ((1,(S1,A1,info,e))!!empty));
fun PB n = (Mark.Embedded'IODConOut 1 n =
           ((1,(S1,A1,info,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

A fórmula será avaliada como verdadeira se existir uma ficha no lugar de entrada do conversor de dados CompDConIn e, com certeza, em algum momento no futuro alguma ficha no lugar IODConOut.

Suponhamos agora, como último exemplo o caso do sincronizador. Existem dois modelos separados, um que transmite o sinal do sistema embarcado para o servidor de tempo real, e outro que transmite o sinal do servidor de tempo real para o sistema embarcado. Temos então um canal *full-duplex* de comunicação. A entrada de um canal leva o sinal à saída do sincronizador oposto. Por exemplo, um sinal presente na entrada do sincronizador do sistema embarcado no sentido de ida para o servidor de tempo real, será transmitido pelo canal até chegar na saída do sincronizador de chegada presente no servidor de tempo real. O mesmo raciocínio pode ser aplicado para o caminho contrário, ou seja, do servidor de tempo real para o sistema embarcado. Na Figura 4.14 podemos ver o MSC para este cenário.

A seguir podemos ver as proposições e a fórmula utilizadas para provar a primeira parte do

fluxo. A fórmula será avaliada como verdadeira se existir uma ficha no lugar SyncIn de entrada do sincronizador do sistema embarcado e, com certeza no futuro, existir uma ficha idêntica no lugar SyncOutRT de saída do sincronizador no servidor de tempo real.

```
fun PA n = (Mark.Embedded'SyncIn 1 n =
           ((1,(S1,A1,info,e))!!empty));
fun PB n = (Mark.RealTime'SyncOutRT 1 n =
           ((1,(S1,A1,info,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

A segunda parte do fluxo é provada com as proposições e fórmula a seguir. Agora, se a fórmula for verdadeira significa que existe uma ficha no lugar SyncInB de entrada do sincronizador no servidor de tempo real e, eventualmente no futuro existir uma ficha idêntica no lugar SyncOut de saída do sistema embarcado.

```
fun PA n = (Mark.RealTime'SyncInB 1 n =
           ((1,(S1,A1,info,e))!!empty));
fun PB n = (Mark.Embedded'SyncOut 1 n =
           ((1,(S1,A1,info,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

Analogamente podemos provar que uma informação transmitida para o servidor de tempo real será retornada para o sistema embarcado para ser acessada por um atuador correspondente. Seria como provar o cenário anterior com um único passo. A seguir temos as proposições e fórmula utilizadas para esta prova. Isso pode ser necessário se estivermos interessados em provar somente a interface do sistema embarcado.

```
fun PA n = (Mark.Embedded'SyncIn 1 n =
           ((1,(S1,A1,info,e))!!empty));
fun PB n = (Mark.Embedded'SyncOut 1 n =
           ((1,(S1,A1,info,e))!!empty));
val formula = AND((NF("info_before",PA)),EV(NF("info_after",PB)));
```

Neste caso a prova desta propriedade significa que se existir uma ficha no lugar SyncIn de entrada do sincronizador no sistema embarcado esta ficha existirá, eventualmente, no lugar SyncOut de saída do sincronizador do sistema embarcado.

4.2.3 Considerações

Neste capítulo estamos interessados na interação dos componentes da arquitetura e não nos detalhes específicos dos componentes. É importante perceber que as provas realizadas aqui são a nível de interface de componentes, o que pode ser visto como um mecanismo de raciocínio modular. Seguindo os conceitos de componentes, reúso de modelos, arquitetura de software, linha de produto, definimos uma estratégia de modelagem de software. Neste capítulo aplicamos esta estratégia a sistemas embarcados utilizando como exemplo uma aplicação de controle de rede de sensores. No caso de uma aplicação específica como, por exemplo, uma aplicação de monitoramento de temperatura, algum formato de dado específico pode ser esperado na saída do conversor de dados como, por exemplo, um número real. Neste caso além do fluxo de controle e informações devemos provar a corretude do modelo no que diz respeito a atividade de conversão propriamente dita. Ou seja, sempre que existir uma informação binária na entrada do conversor de dados, instanciada como objeto no interpretador de entrada e saída, com certeza no futuro uma informação no formato real existirá na saída que leva ao sincronizador.

As propriedades a serem provadas dependem de que tipo de componentes que estão sendo utilizados e estes, por sua vez, dependem do sistema a ser desenvolvido. A estratégia de modelagem e verificação formal, apresentada em trabalhos anteriores e sua aplicação ao domínio de sistemas embarcados de tempo real apresentada neste capítulo é uma contribuição ao desenvolvimento de software, especificamente sistemas embarcados, no que diz respeito a raciocínio modular e refatoramento a nível de modelos.

4.3 Conclusões

Neste capítulo apresentamos a aplicação de um processo de desenvolvimento baseado em componentes ao domínio de sistemas embarcados. Especificamos e verificamos um sistema de controle para redes de sensores e atuadores.

O uso de conceitos como linha de produto, arquitetura, e componentes, permite o desenvolvimento sistemático e formal de sistemas, em um domínio bem definido. A idéia principal é que a fase de modelagem é a mais importante e, com as estratégias, técnicas e métodos de especificação e verificação formal apresentados neste capítulo realizamos refatoramento a nível de modelos, que

é mais fácil do que a nível de código.

Além disso, definimos uma estratégia para verificação modular que permite a verificação de propriedades do sistema a nível de interface dos componentes do modelo, sem a necessidade de verificações do comportamento interno dos componentes. A idéia é que os componentes sejam reusados de projetos bem sucedidos e ao longo do tempo não é mais necessário provar propriedades especificas dos componentes, mas do novo sistema montado, ou integrado, a partir desses componentes.

Um contribuição importante deste trabalho, que será apresentada no Capítulo 5, foi o desenvolvimento de técnicas para geração de espaço de estados abstratos para modelos de sistemas seguindo o processo de desenvolvimento e as estratégias, técnicas e métodos apresentados neste capítulo e no Capítulo 3. Isto é importante pois não é necessário gerar o espaço de estados interno de cada componente.

Capítulo 5

Análise Modular de Modelos HCPN

5.1 Introdução

Neste capítulo apresentamos a definição formal para a especificação de interfaces de modelos HCPN utilizando grafo de ocorrência com classes de equivalência compatíveis e autômatos com interfaces. Também apresentamos a prova de fluxo e invariante lugar para um modelo apresentado neste trabalho para o domínio de sistemas embarcados. O uso de invariantes e fluxo de lugar permite a verificação de propriedades como bloqueio, por exemplo, no espaço de estados abstraído através de classes de equivalência. Os autômatos com interface, por sua vez, permitem a verificação de compatibilidade entre duas interfaces.

Este capítulo está organizado da seguinte forma. Na Seção 5.2 as definições e exemplos de uso de grafos de ocorrência com classes de equivalência compatíveis são apresentados. Na Seção 2.3 autômatos com interface são introduzidos e alguns exemplos de seu uso são discutidos.

5.2 Grafos de Ocorrência com Classes de Equivalência Compatíveis

Há dois tipos de classes de equivalência definidas para grafos de ocorrência para CPN. A primeira, denominada classe de equivalência *consistente* é definida considerando o sucessor direto das marcações, enquanto que a segunda, classe de equivalência *compatível* é definida considerando uma seqüencia, onde os integrantes da seqüencia pertencem a mesma classe de equivalência.

Para a especificação de interface de modelos utilizamos classes de equivalência compatíveis. Tal escolha é justificada pelo fato de que objetivamos abstrair todo o comportamento interno do modelo. É importante ressaltar que parte dos resultados apresentados neste capítulo são baseados nas definições apresentadas em [51] e na aplicação destas definições descrita em [38].

Definições

Inicialmente faremos algumas considerações antes de apresentar a formalização de interfaces, bem como sua análise. Ao longo deste capítulo consideramos que:

- $\mathbb{M}$ é o conjunto de todas as marcações de um modelo;
- BE é o conjunto de todos os elementos de ligação;
- Se x e y são equivalentes, escrevemos $x \approx y$, $[x]$ denota a classe de equivalência, onde estes elementos podem pertencer a $\mathbb{M}$ ou a BE;
- $[X]$ denota o conjunto de todos os elementos pertencentes a classe de equivalência de algum elemento de X;
- $[[M\rangle]$ denota o conjunto de marcações alcançáveis equivalentes a partir de M;
- $Next_\tau(M_1) = \{(be, M) \in BE \times \mathbb{M} : M_1[\tau be\rangle M\}$,
  onde $M_1[\tau be\rangle M$ denota que existe uma seqüencia de ocorrência finita
  $M_1[be_1\rangle M_2[be_2\rangle M_3 ... M_n[be_n\rangle M_{n+1}[be\rangle M$, tal que $n \in \mathbb{N}$, $be_i \in BE$ para todo $i \in 1..n$ e
  $M_i \approx_M M_1$ para todo $i \in 1..n+1$, enquanto que $M \not\approx_M M_1$.

A seguir apresentamos as definições referentes ao conceito de equivalência compatível. Primeiro apresentamos o conceito geral de equivalência.

Definição 5.1 *Uma* **especificação de equivalência** *para uma CPN é um par* $(\approx_M, \approx_{BE})$ *onde* $\approx_M$ *é uma relação de equivalência em* $\mathbb{M}$ *enquanto* $\approx_{BE}$ *é uma relação de equivalência em* BE*.*

Na Definição 5.2, o conceito de equivalência compatível é apresentado, visto que, conforme explicado no início deste capítulo é a classe considerada para a solução apresentada neste trabalho.

Definição 5.2 *Uma especificação de equivalência é* **compatível** *se, e somente se, a seguinte propriedade é satisfeita, para todo* $M_1, M_2 \in [[M_0\rangle]$*:*
$M_1 \approx_M M_2 \Rightarrow [Next_\tau(M_1)] = [Next_\tau(M_2)]$.

A seguir, apresentamos a definição de Grafo de Ocorrência com classes de Equivalência (GOE), considerando equivalência compatível conforme Definição 5.2.

Definição 5.3 *Dada uma rede de Petri colorida e uma especificação de equivalência compatível* ($\approx_M$,$\approx_{BE}$)*, o* **grafo de ocorrência com classes de equivalência (GOE)** *é o grafo direcionado* $GOE = (V, A, N)$*, onde:*

- $V = \{C \in M_\approx \mid C \cap [M_0\rangle \neq \varphi\}$.
- $A = \{(C_1, B, C_2) \in V \times BE_\approx \times V \mid \exists (M_1, be, M_2) \in C_1 \times B \times C_2 : M_1[be\rangle M_2\}$.
- $\forall a = (C_1, B, C_2) \in A : N(a) = (C_1, C_2)$.

Para os objetivos deste trabalho estamos interessados em analisar propriedades de vivacidade e limitação. Para atingir estes objetivos, usando classes de equivalência compatíveis, é preciso definir também fluxo e invariante de lugar.

Invariante de lugar é um conjunto de equações, relacionadas à marcação dos lugares, que são satisfeitas para toda marcação alcançável do modelo. Fluxo de lugar é uma propriedade que garante que todas as fichas que chegam aos lugares de uma rede também saem deles, e que outras fichas não são criadas.

Definição 5.4 *Para um rede de Petri não-hierárquica, um* **conjunto ponderado de lugares** *com imagem* $A \in \Sigma$ *é o conjunto de funções* $W = \{W_p\}_{p \in P}$ *tal que* $W_p \in [C(p)_{cp} \rightarrow A_{cp}]_L$ *para todo* $p \in P$*.*

1. W *é um fluxo de lugar se, e somente se:*
   $\forall (t, b) \in BE : \sum_{p \in P} W_p(E(p, t)\langle b \rangle) = \sum_{p \in P} W_p(E(t, p)\langle b \rangle)$.
2. W *determina um invariante de lugar se, e somente se:*
   $\forall M \in [M_0\rangle : \sum_{p \in P} W_p(M(p)) = \sum_{p \in P} W_p(M_0(p))$.

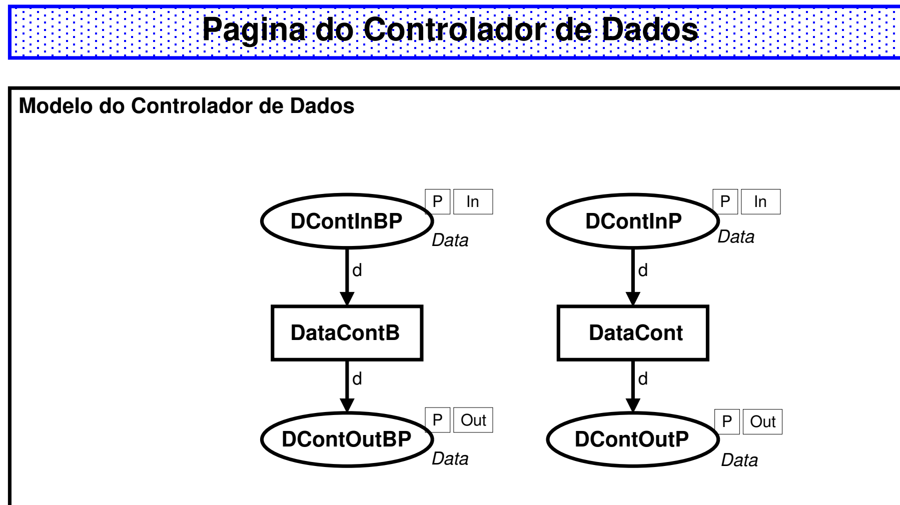


Figura 5.1: Página do modelo do controlador de dados.

Onde:

- *Cada peso* W_p *é uma função que mapeia* $C(p)$ *em algum conjunto de cor* $A \in \Sigma$ *compartilhados por todos os pesos;*
- X_{cp} *denota o conjunto de todos os conjuntos ponderados de* X*. Um conjunto ponderado é um multiconjunto em que os coeficientes dos elementos podem ser negativos;*
- $[X \rightarrow Y]_L$ *denota o conjunto de todas as funções lineares de* X *em* Y*.*

Teorema 5.1 W *é um fluxo de lugar se, e somente se,* W *determina um invariante de lugar.*

Para exemplificar o uso de classes de equivalência e invariante e fluxo de lugar suponha o modelo do controlador de dados apresentado no Capítulo 4, apresentado novamente na Figura 5.1.

Na Figura 5.2 o componente e sua interface podem ser vistos. É para esta interface, por exemplo, que desejamos provar o fluxo de lugar para poder fazer a verificação a nível de interface utilizando grafos de ocorrência com classes de equivalência compatível. É importante lembrar que o modelo ilustrado na Figura 5.1 será *colado* no modelo ilustrado na Figura 5.2, no lugar da transição DataControl.

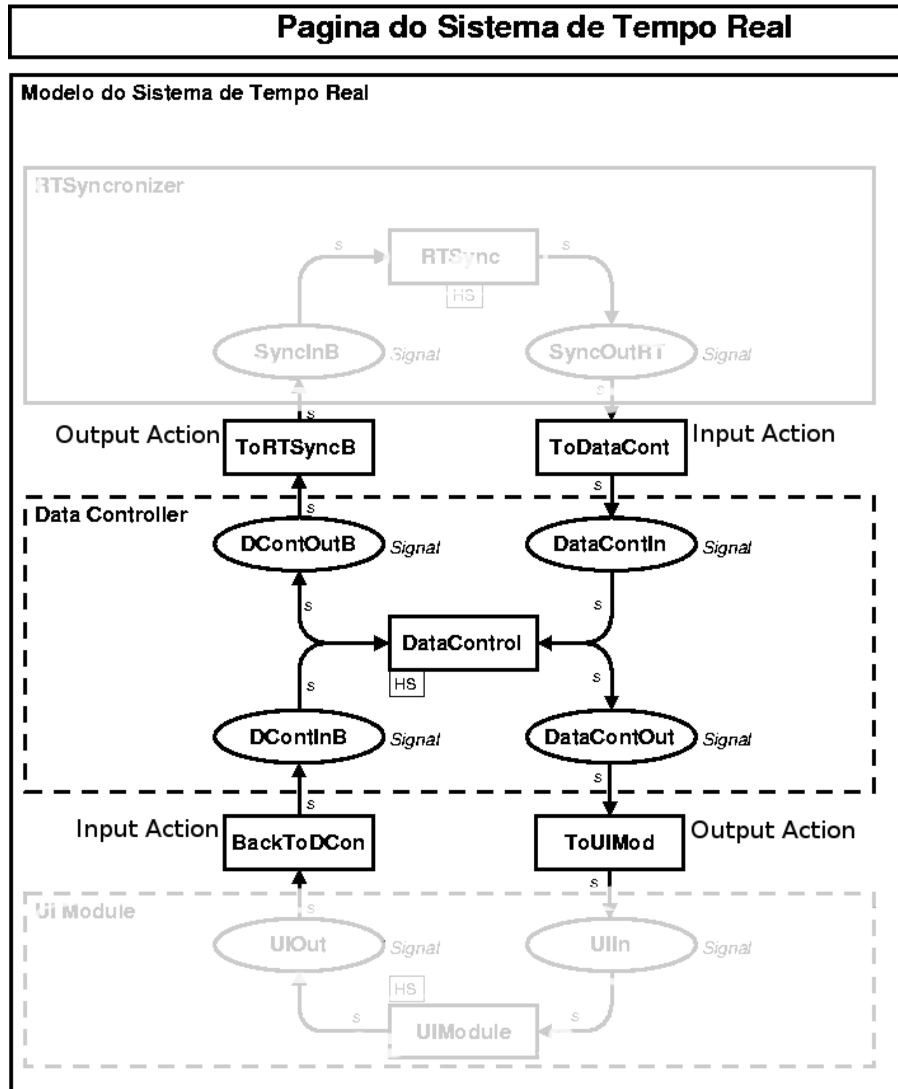


Figura 5.2: Interface para o modelo do controlador de dados.

5.2.1 Prova do Fluxo de Lugar

A seguir faremos a prova do fluxo de lugar para o modelo ilustrado na Figura 5.1, considerando sua interface ilustrada na Figura 5.2. Seguindo a Definição 5.4, temos que:

- $\forall p \in P_{BE}$ e $\forall t \in T_{BE}$:

$$W_p(e(ToDataCont, DataContIn)\langle b\rangle) = W_p(e(DataContOut, ToUIMod)\langle b\rangle) \quad (5.1)$$

$$W_p(e(DContInP, DataCont)\langle b\rangle) = W_p(e(DataCont, DContOutP)\langle b\rangle) \quad (5.2)$$

$$W_p(e(DContInBP, DataContB)\langle b\rangle) = W_p(e(DataContB, DContOutBP)\langle b\rangle) \quad (5.3)$$

$$W_p(e(BackToDCon, DataContInB)\langle b\rangle) = W_p(e(DataContOutB, ToRTSyncB)\langle b\rangle) \quad (5.4)$$

Adicionando $W_p(e(BackToDCon, DataContInB)\langle b\rangle)$ a ambos os lados da Equação 5.1, tem-se:

$$W_p(e(ToDataCont, DataContIn)\langle b\rangle) + W_p(e(BackToDCon, DataContInB)\langle b\rangle) = W_p(e(DataContOut, ToUIMod)\langle b\rangle) + W_p(e(BackToDCon, DataContInB)\langle b\rangle) \quad (5.5)$$

Mas, de acordo com a Equação 5.4, $W_p(e(BackToDCon, DataContInB)\langle b\rangle) = W_p(e(DataContOutB, ToRTSyncB)\langle b\rangle)$. Então, aplicando essa igualdade ao lado direito da Equação 5.5, tem-se:

$$W_p(e(ToDataCont, DataContIn)\langle b\rangle) + W_p(e(BackToDCon, DataContInB)\langle b\rangle) = W_p(e(DataContOut, ToUIMod)\langle b\rangle) + W_p(e(DataContOutB, ToRTSyncB)\langle b\rangle) \quad (5.6)$$

De forma semelhante, pode-se adicionar $W_p(e(DContInP, DataCont)\langle b\rangle)$ a ambos os lados da Equação 5.6 e depois substituí-lo, no lado esquerdo da nova equação, pela igualdade da Equação 5.2, resultando na seguinte equação:

$$W_p(e(ToDataCont, DataContIn)\langle b\rangle) + W_p(e(BackToDCon, DataContInB)\langle b\rangle) + W_p(e(DataCont, DContOutP)\langle b\rangle) = W_p(e(DataContOut, ToUIMod)\langle b\rangle) + W_p(e(DataContOutB, ToRTSyncB)\langle b\rangle) + W_p(e(DContInP, DataCont)\langle b\rangle) \quad (5.7)$$

De forma semelhante, pode-se adicionar $W_p(e(DContInBP, DataContB)\langle b\rangle)$ a ambos os lados da Equação 5.7 e depois substituí-lo no lado esquerdo da nova equação, pela igualdade da Equação 5.3, resultando na seguinte equação:

$$\begin{aligned} W_p(e(ToDataCont, DataContIn)\langle b\rangle) + W_p(e(BackToDCon, DataContInB)\langle b\rangle) + \\ W_p(e(DataCont, DContOutP)\langle b\rangle) + W_p(e(DataContB, DContOutBP)\langle b\rangle) = \\ W_p(e(DataContOut, ToUIMod)\langle b\rangle) + W_p(e(DataContOutB, ToRTSyncB)\langle b\rangle) + \\ W_p(e(DContInP, DataCont)\langle b\rangle) + W_p(e(DContInBP, DataContB)\langle b\rangle) \\ \Rightarrow \forall(t,b) \in BE : \sum_{p \in P_{BE}} W_p(e(t,p)\langle b\rangle) = \sum_{p \in P_{BE}} W_p(e(p,t)\langle b\rangle) \end{aligned} \tag{5.8}$$

Portanto, conclui-se que esse modelo de componente de controlador de dados apresenta um fluxo de lugar e, segundo o Teorema 5.1, um invariante de lugar. Pelo fluxo de lugar, podemos concluir que o modelo desse componente é vivo. □

Também pelo fluxo de lugar, e pelo fatos dos lugares do modelo do componente não sofrerem influência externa, dizemos que esse modelo é limitado.

5.2.2 Especificação de Equivalência Compatível

Agora que já se provou que o modelo do componente não altera as propriedades de vivacidade e limitação do modelo onde está inserido, pode-se definir uma especificação de equivalência compatível em que:

1. A relação $\approx_M$ define como equivalentes, marcações resultantes de alterações apenas nas marcações dos lugares de entrada das transições de entrada do modelo do componente e nos seus próprios lugares.

2. A relação $\approx_{BE}$ estabelece como equivalentes, elementos de ligação contendo transições pertencentes ao mesmo modelo de componente.

Para as definições de relação de equivalência compatível $\approx_M$ e $\approx_{BE}$ a seguir considere que:

- t_C é um modelo de componente visto da sua interface de entrada e saída, ou seja, é a transição que representa (abstrai) o componente;

- T_C é o conjunto de t_C;
- t_{ent} são as transições de entrada de um t_C;
- P_C é o conjunto de lugares de um t_C.

Relação de Equivalência no Conjunto de Marcações - $\approx_M$

Para duas marcações M_1, $M_2 \in [[M_0\rangle]$, tem-se que:

$$M_1 \approx_M M_2 \iff \bigvee_{\forall t_C \in T_C} \Big(\forall p \in (P - {}^\bullet t_C) : M_1(p) = M_2(p)) \\ \wedge \Big(\sum_{\forall p \in {}^\bullet t_C} |M_1(p)| + \frac{\sum_{\forall p \in {}^\bullet t_C} |e(p, t_C)|}{\sum_{\forall p' \in t_{ent}^\bullet, t_{ent} \in t_C} |e(t_{ent}, p')|} \cdot \sum_{\forall p'' \in P_C} |M_1(p'')| = \sum_{\forall p \in {}^\bullet t_C} |M_2(p)| + \\ \frac{\sum_{\forall p \in {}^\bullet t_C} |e(p, t_C)|}{\sum_{\forall p' \in t_{ent}^\bullet, t_{ent} \in t_C} |e(t_{ent}, p')|} \cdot \sum_{\forall p'' \in P_C} |M_2(p'')| \Big) \Big) \tag{5.9}$$

Duas marcações M_1 e M_2 serão equivalentes se, e somente se, o somatório do número de fichas em todo lugar p que é lugar de entrada do modelo do componente C (t_C), em M_1, mais a razão entre o somatório do número de fichas retiradas dos lugares de entrada das transições de entrada de C (t_{ent}), quando elas ocorrem, e o somatório do número de fichas depositadas nos lugares de saída dessas transições, multiplicado pelo somatório do número de fichas presentes em todos os lugares de C (P_C), é igual ao somatório do número de fichas em todos os lugares p que são lugares de entrada de C, para M_2, mais a razão entre o somatório do número de fichas retiradas dos lugares de entrada das transições de entrada de C, quando elas ocorrem, e o somatório do número de fichas depositadas nos lugares de saída dessas transições, multiplicado pelo somatório do número de fichas presentes em todos os lugares de C.

Relação de Equivalência no Conjunto de Ligações - $\approx_{BE}$

Considerando que se denota por $t(be)$ a transição de um elemento de ligação $be \in BE$, tem-se que:

$$be_1 \approx_{BE} be_2 \Longleftrightarrow \bigvee_{\forall t \in\ t_C} (t(be_1) \in t_C \wedge t(be_2) \in t_C) \tag{5.10}$$

Dois elementos de ligação são equivalentes quando a transição presente no elemento de ligação be_1 pertence ao modelo do componente C e a transição presente no elemento de ligação be_2 também pertence a esse mesmo modelo.

5.2.3 Prova da Especificação de Equivalência Compatível

Nesta seção prova-se que as especificações de equivalência definidas na seção 5.2.2 são compatíveis. A prova consiste em provar que as Equações 5.10 e 5.9 atendem à Definição 5.2, ou seja:

$$M_1 \approx_M M_2 \Rightarrow [Next_\tau(M_1)] = [Next_\tau(M_2)] \tag{5.11}$$

Prova:

Seja M_0 uma marcação inicial e $M_1,\ M_2\ \in\ [[M_0\rangle]$. Segundo as relações definidas nas Equações 5.9 e 5.10, tem-se que:

$$\begin{aligned} M_1 \approx_M M_2 \Rightarrow M_1 = M_2 \vee M_1[\tau_1\rangle M_2[\tau_2 be\rangle M \vee M_2[\tau_1\rangle M_1[\tau_2 be\rangle M : \\ \tau_1\tau_2 = \tau \wedge M_1 \not\approx_M M \not\approx_M M_2 \end{aligned} \tag{5.12}$$

Se $M_1 = M_2$ então tem-se que $[Next_\tau(M_1] = [Next_\tau(M_2]$. De forma semelhante, se $M_1[\tau_1\rangle M_2[\tau_2 be\rangle M \vee M_2[\tau_1\rangle M_1[\tau_2 be\rangle M$, ambas as marcações levarão a M e, portanto, $[Next_\tau(M_1] = [Next_\tau(M_2]$. Além disso os elementos de ligação em τ entre a marcação M_1, ou M_2, e M só podem envolver as transições ToDataCont, BackToDCon, DataContB e DataCont. O elemento de ligação be envolve as transições ToRTSyncB e ToUIMod.

Então, é preciso provar que não existe um M_2, tal que $M_1 \approx_M M_2$ e M_2 não seja nenhuma das marcações obtidas pela ocorrência dos elementos de ligação em τ. A prova será por absurdo.

Suponha que essa marcação M_2 exista. Se ela não corresponde a uma das marcações alcançáveis pela ocorrência de elemento(s) de ligação na seqüencia τ, então ela é alcançável pela ocorrência de algum elemento de ligação que contém uma transição que não pertence ao

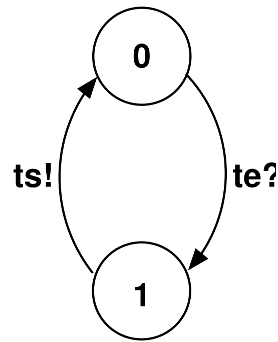


Figura 5.3: Autômato com interface para um modelo simples.

modelo do componente. Portanto, a ocorrência de uma transição externa ao modelo do componente remove e/ou deposita fichas nos lugares da super-página e, portanto, não satisfaz a relação de equivalência para as marcações, pois (i) os lugares da super-página não terão mais os mesmos multiconjuntos de fichas e (ii) o número de fichas nos lugares de entrada do modelo do componente, mais a razão entre os somatórios, multiplicado pelo número de fichas nos lugares do modelo do componente, não será mais idêntico. Portanto, por absurdo, para que duas marcações sejam equivalentes é preciso que sejam idênticas, ou que uma delas seja alcançável pela ocorrência de um dos elementos de ligação presentes em τ da outra. □

5.3 Definição de um Autômato com Interface para um Modelo HCPN

Seja M um modelo CPN presente em um modelo HCPN com transições de entrada e saída. Suponha inicialmente um modelo simples com apenas uma transição de entrada e uma de saída. Seja $(t_e, p_i) \ldots (p_j, t_s)$ um caminho que leva uma transição de entrada a uma de saída. O autômato com interface para esse modelo é mostrado na Figura 5.3. Portanto, para várias transições de entrada e saída, o autômato com interface para este modelo será a ordenação parcial de todos os caminhos possíveis entre cada transição de entrada e outra de saída. Para o exemplo do controlador de dados o autômato com interface é mostrado na Figura 5.4.

A seguir, o código na linguagem de entrada para a ferramenta TICC para o modelo ilustrado na Figura 5.3 é mostrado.

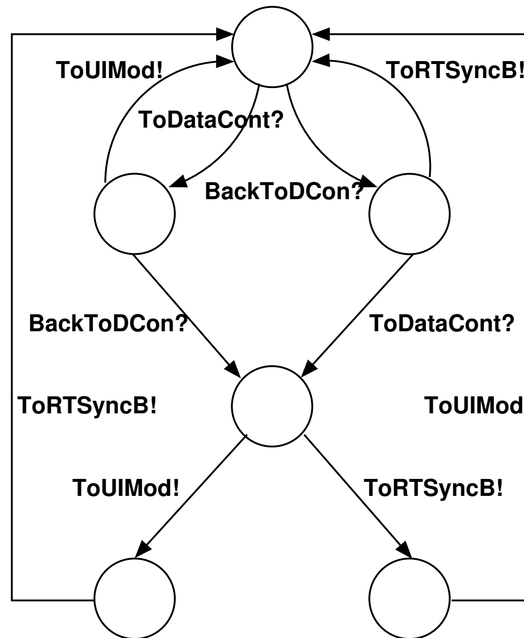


Figura 5.4: Autômato com interface para o controlador de dados.

```
module Interface:
  var s: [0..1]
  input te  : { local: s = 0 ==> s' := 1 }
  output ts : {        s = 1 ==> s'  = 0 }
endmodule
```

5.4 Algoritmo para Extração do TIA de um HCPN

Nesta seção o algoritmo para a extração dos modelos TIA a partir de um dado modelo HCPN de um componente é apresentado. É importante notar que não se pode ter lugares de fusão nos modelos dos componentes pois eles inviabilizam a prova de fluxo e invariante de lugar. Além disso, somente pode haver uma transição de substituição no modelo de um componente se os fluxos de lugares para o modelo que esta representa também tiverem sido provados. É fácil perceber que não faz sentido haver partes de um modelo de componente interferindo, ou sofrendo interferência, de modelos de outros componentes.

O Algoritmo 3, de extração, gera um TIA levando em conta todos os caminhos entre uma transição de entrada e uma transição de saída. Obviamente uma transição de entrada pode estar

no caminho de várias de saída e vice-versa. Mas no caso em que os fluxos forem independentes o algoritmo constrói um caminho onde uma certa transição de entrada só pode levar a uma transição de saída especifica. O resto do TIA nada mais é do que a ordenação parcial dos possíveis fluxos.

Antes de apresentar o Algoritmo 3 alguns esclarecimentos são importantes. Considere que $(T_{IN_i}, T_{OUT_i}) = P_i$ é um par que representa um caminho entre uma transição de entrada e uma de saída. Neste caso, AP é o conjunto de todos os caminhos P_i. O TIA para o modelo HCPN será a ordenação parcial de todas as combinações possíveis de cada T_{IN_i} e T_{OUT_i} em AP, exceto os que não se entrelaçam no modelo HCPN.

O início do algoritmo, linhas 1 e 2, é para inicializar as estruturas, e o estado inicial é iniciado com o padrão INIT_STATE, somente para diferenciá-lo dos demais. Note que isso não afeta o algoritmo ou o modelo gerado, é utilizado apenas para fins didáticos. Este poderia ser iniciado com $\emptyset$.

O ciclo iniciado na linha 3 serve para criar o início do TIA. Isto é necessário pois se algum caminho P_i não se entrelaça com outro P_j então este deve ser gerado de imediato, ou seja, sua transição de entrada T_{IN_i} deve ser imediatamente seguida pela sua transição de saída T_{OUT_i} e somente por ela. Obviamente outras transições de entrada podem ocorrer antes que T_{OUT_i} ocorra.

Se T_{IN_i} não foi criada, cria-se um novo estado e uma transição do estado atual para este novo com rótulo T_{IN_i} (linhas 4, 5 e 6), caso contrário cria-se uma transição do estado atual para o estado inicial com o rótulo T_{OUT_i} (linhas 10 e 11). Note que o estado é criado com um rótulo que representa todas as ações que levam a ele. Além disso, a função create_State() também realiza outras funções como adicionar o estado em S e atribuir um identificador à este. A função get_State() retorna o estado para um determinado rótulo.

Para cada novo estado criado, exceto o inicial, deve-se percorrer AP para criar as ordenações parciais das ações. A função state() serve para retornar o rótulo do estado. Então, se T_{IN_i} não pertence ao rótulo do estado ela deve ser criada para este estado (linha 17). Para criar esta ação é necessário verificar se um estado com o rótulo do estado atual mais T_{IN_i} já existe. Se não existir um novo deve ser criado, caso contrario o existente é utilizado (linhas 18-23). A função exists() é utilizada para verificar se existe em S um estado com um determinado rótulo. Note que o uso do sinal $+$ é genérico aqui e sua semântica dependerá da estrutura de dados especifica a ser utilizada. Portanto, este sinal pode significar concatenação de cadeias de caracteres, inserção em vetores, entre outros. Se T_{IN_i} já existe no rótulo de S_i então deve-se verificar T_{OUT_i} do par P_i atual. Se

Algoritmo 3 Algoritimo para extração de TIA

1: $S := \emptyset;\ Act := \emptyset;\ T_{IN_{DONE}} := \emptyset$
2: $S_{Ini} := INIT_STATE$
3: **para cada** $P_i \in AP$ **faça**
4:     **se** $T_{IN_i} \notin T_{IN_{DONE}}$ **então**
5:         $S_N := create_State(T_{IN_i})$
6:         $create_\tau_I(S_{Ini}, T_{IN_i}?, S_N)$
7:         $T_{IN_{DONE}} := T_{IN_{DONE}} + T_{IN_i}$
8:         $create_\tau_O(S_N, T_{OUT_i}!, S_{Ini})$
9:     **senão**
10:         $S_N := get_State(T_{IN_i})$
11:         $create_\tau_O(S_N, T_{OUT_i}!, S_{Ini})$
12:     **fim se**
13: **fim para**
14: **para cada** $S_i \in S \setminus S_{Ini}$ **faça**
15:     $S_T := S_i$
16:     **para cada** $P_i \in AP$ **faça**
17:         **se** $T_{IN_i} \notin state(S_T)$ **então**
18:             **se** $exists(S_T + T_{IN_i}) = FALSE$ **então**
19:                 **se** $state.length(S_T + T_{IN_i}) = Act.length()$ **então**
20:                     $S_N := get_State(T_{IN_i})$
21:                 **senão**
22:                     $S_N := create_State(S_T + T_{IN_i})$
23:                 **fim se**
24:             **senão**
25:                 $S_N := get_State(S_T + T_{IN_i})$
26:             **fim se**
27:             **se** $\nexists\ \tau_I = (S_T, T_{IN_i}, S_N)$ **então**
28:                 $create_\tau_I(S_T, T_{IN_i}?, S_N)$
29:             **fim se**
30:         **fim se**
31:         **se** $(\nexists\ \tau_O = (S_T, T_{OUT_i}, _)) \wedge (T_{OUT_i} \notin state(S_T)) \wedge (T_{IN_i} \in state(S_T))$ **então**
32:             **se** $state.length(S_T + T_{OUT_i}) = Act.length()$ **então**
33:                 $S_N := S_{Ini}$
34:             **senão se** $exists(S_T + T_{OUT_i}) = FALSE$ **então**
35:                 $S_N := create_State(S_T + T_{OUT_i})$
36:             **senão**
37:                 $S_N := get_State(S_T + T_{OUT_i})$
38:             **fim se**
39:             $create_\tau_O(S_T, T_{OUT_i}!, S_N)$
40:         **fim se**
41:     **fim para**
42: **fim para**

este T_{OUT_i} não pertence às transições de saída de S_T, então esta transição deve ser criada. Mas para decidir se um novo estado deve ser criado, basta verificar se o tamanho do rótulo do estado atual é igual ao número de ações. Neste caso o próximo estado deve ser o inicial, pois todas as ordenações parciais já foram geradas.

Como o primeiro ciclo (linhas 3-13) depende da quantidade de caminhos possíveis e, visto que o número de transições de entrada e saída de um modelo são finitas, este ciclo para depois de $|AP|$ iterações. O número de estados criados neste primeiro ciclo é linear com o número de transições de entrada, ou seja, um para cada. O segundo ciclo (linhas 14-42) depende dos estados criados no primeiro ciclo e dos novos que serão criados no terceiro ciclo. O terceiro ciclo (linhas 16-41) depende também de AP, que é finito. Portanto, este ciclo para após $|AP|$ iterações. O número de estados criados neste ciclo é exponencial, no pior caso, com relação as transições de entrada e saída. Entretanto, é possível analisar o Algoritmo 3 de outra forma. O grafo gerado terá profundidade $|E|+|S|$ e os estados criados em cada nível será, no pior caso, $C(|E|+|S|)$, ou seja, a combinação sem repetição das ações de entrada e saída de acordo com a profundidade do grafo. Portanto, o segundo ciclo também é finito. Finalmente, as funções create_State(), get_State(), state(), exists(), $create_\tau_O$ e $create_\tau_I$ são funções de busca e inserção em estruturas de dados simples.

É importante notar que apesar do pior caso ser exponencial, na grande maioria dos casos será muito menor. Um primeiro motivo para isso é que não podemos ter uma ação de saída antes que a entrada tenha ocorrido. Além disso, não se pode ter também uma transição de saída no mesmo estado em que sua respectiva entrada ocorre. Uma outra forma de observar essas restrições é considerarmos um identificador de estados como uma cadeia de caracteres com as ações que levam a esse estado. Neste caso, não se pode ter estados cujos rótulos comecem com qualquer combinação de ações de saída. Não se pode ter, também, estados com rótulos que contenham mais transições de saída do que de entrada. Além disso, com base na teoria de desenvolvimento baseado em componentes é de se esperar que as ações de entrada dos componentes não se relacionem. Ou seja, normalmente uma entrada gera uma saída. Em alguns casos entradas se combinam para gerar uma saída, ou uma entrada gera várias saídas. Mas é improvável que muitas entradas e saídas estejam combinadas formando vários fluxos. Portanto, para a grande maioria dos casos, a complexidade relacionada ao número de estados gerados para o TIA será bem melhor que a exponencial .

5.5 Estratégia Híbrida de Análise Modular

Conforme dito na introdução deste capítulo, GOE são utilizados para análise do modelo HCPN, sem a necessidade de geração de todo o espaço de estados para todos os componentes, para propriedades como vivência e limitação. De outro lado, TIA é utilizado para a verificação de compatibilidade entre componentes. Nesta seção discutimos como esta solução híbrida é utilizada.

Uma observação importante com relação a TIA é que se os lugares dos componentes sofrerem influência externa, fica difícil, se não impossível definir a interface do componente. Isto é devido ao fato de que teríamos que prever todas as possibilidades de comportamento para especificar o que o componente espera, e garante, para todo comportamento possível. Além disso, o fato da limitação da rede permite garantir a unicidade da interface. Ou seja, mesmo que se tenha várias instâncias de páginas, a interface é a mesma. Portanto, também para a análise de compatibilidade utilizando TIA é necessário a prova dos fluxos de lugares para os componentes.

Devido a natureza da análise usando TIA, é desejável que esta seja realizada primeiro que a análise com GOE. Desta forma, inicialmente é analisado se os componentes podem ser utilizados juntos, se sim, segue a análise com GOE para propriedades de HCPN.

A abordagem de analise modular híbrida pode ser resumida nos seguintes passos:

1. *Prova de fluxos de lugares:* Nesta etapa os fluxos de lugares para os componentes a serem inseridos no projeto são provados;

2. *Análise de interface:* Nesta etapa as interfaces são definidas e analisadas;

3. *Análise com GOE:* Se os componentes forem compatíveis, isto é, podem ser utilizados juntos, segue a análise de propriedades utilizando HCPN com GOE.

5.5.1 Considerações a Respeito da Abordagem Híbrida

A adoção de uma abordagem híbrida conforme descrita neste capítulo se deu devido a uma série de considerações envolvendo outras possibilidades. A seguir essas outras possibilidades são discutidas.

- **GOE** A abordagem utilizando somente GOE tem algumas vantagens e desvantagens com relação à híbrida. A vantagem é uma técnica formalizada, apesar de não haver ainda fer-

ramentas para manipulação automática. A desvantagem é justamente limitar a análise às propriedades que podem ser verificadas com GOE.

- **TIA** A abordagem utilizando somente TIA tem a vantagem de ser um trabalho também formalizado e possuir uma ferramenta para analise de compatibilidade. Esse tipo de verificação é muito útil quando se considera mudanças no projeto. A desvantagem é que é desejável também poder provar propriedades como limitação e vivacidade, que dependem do comportamento interno do componente. Além disso, é necessário provar que a interface é unicamente definida, o que é conseguido com a prova do fluxo de lugares para o modelo do componente.

Devido as considerações acima, decidiu-se por uma abordagem híbrida onde a prova do fluxo de lugares permite definir unicamente as interfaces dos modelos de componentes. Essas interfaces são analisadas para compatibilidade e, se forem compatíveis é realizada a analise com GOE.

Com relação ao exemplo apresentado no Capítulo 4 e a estratégia híbrida de verificação definida neste capítulo é importante esclarecer a intersecção entre eles. A estratégia de verificação utilizada no exemplo é realizada utilizando-se o grafo de ocorrência com classes de equivalência para a verificação das propriedades. Além disso, após a prova dos fluxos de lugares para os modelos dos componentes é possível substituí-los pelas suas interfaces e gerar os MSCs.

Capítulo 6

Conclusões

A contribuição deste trabalho está definida no contexto de métodos e técnicas para a especificaçãao e verificação de sistemas embarcados de forma sistemática utilizando um processo de reúso de modelos de componentes para o citado domínio. No caso específico de verificação, a contribuição é disponibilizar ferramentas para análise de sistemas embarcados baseada em uma modelagem formal e verificação de modelos. Deste modo, provendo facilidades para reduzir a possibilidade de explosão do espaço de estados.

Um processo de reúso bem definido promove a diminuição do tempo de desenvolvimento aumentando a qualidade do artefato desenvolvido. Isto é explicado pelo fato de que partes comuns definidas em um dado domínio especifico podem ser aproveitadas em outros projetos. Com relação a qualidade é de se esperar que essas partes reutilizadas em vários projetos bem sucedidos estejam validadas, aumentando portanto a confiança nos artefatos que são recuperados do repositório.

Um processo de desenvolvimento baseado em componentes é adotado como guia para o processo de reúso. Isso se justifica pelo fato de que componentes são uma boa abordagem para decomposição de sistemas complexos. Componentes têm sido utilizados em vários domínios de aplicação, com várias linguagens de programação e modelagem.

A abordagem definida neste trabalho para a verificação é híbrida. São utilizados grafos de ocorrência com classes de equivalência para evitar a geração do espaço de estados completo. Desta forma, o comportamento interno dos componentes é ignorado. Através de provas de fluxo de lugar e invariante é possível garantir propriedades de limitação e vivacidade mesmo para este novo

espaço de estados reduzido. Autômatos com interface temporizados são utilizados para descrever a interface dos componentes e realizar a verificação de compatibilidade. Isto é importante para analisar se dois componentes podem ser utilizados juntos, e se justifica devido ao surgimento de mudanças ao longo da evolução do sistema.

Os dois formalismos são utilizados juntos para resolver problemas diferentes de análise. Enquanto que HCPN é utilizada para uma validação inicial, antes que o componente seja inserido no repositório para o manter o processo de reúso, TIA (*timed interface automata* é utilizado ao longo da evolução do sistema. Além disso, as classes de equivalência são utilizadas para garantir as que as propriedades dos novos componentes possam ser analisadas sem a necessidade de geração de um grafo completo envolvendo outros componentes não modificados. A arquitetura do sistema como um todo também é analisada utilizando HCPN e classes de equivalência.

6.1 Sugestões para Trabalhos Futuros

Uma sugestão para trabalhos futuros é implementar o algoritmo para a extração do modelo TIA a partir do modelo HCPN. Isto é importante para automatizar a estratégia híbrida de verificação definida neste trabalho.

Uma outra extensão deste trabalho é a realização de estudos de caso para outras aplicações. Algumas possibilidades são controle de processos industriais e disponibilização de serviços para dispositivos móveis. No primeiro caso, assim como no estudo de caso deste trabalho, é importante analisar a corretude do funcionamento do sistema como um todo e dos componentes individuais. No caso do segundo exemplo, algumas características interessantes a serem analizadas são consumo de bateria e memória, tempo de comunicação entre os dispositivos ou entre um dispositivo e uma central de serviços.

No domínio de sistemas críticos e de tempo real características como tempo e consumo de energia e memória estão entre as mais importantes a serem verificadas. Foi desenvolvido um trabalho para uso de números nebulosos, chamado de *Fuzzy Time Coloured Petri Nets* (FTCPN) para especificar e analisar o comportamento de sistemas com relação a estas características, inclusive com verificação de modelos para propriedades nebulosas [20]. Um trabalho futuro interessante é o uso de FTCPN para analisar as citadas propriedades seguindo os processos e estratégias de modelagem e verificação apresentados neste trabalho.

Um ambiente de desenvolvimento que integre as várias partes deste trabalho, tanto a nível de modelagem quanto a nível de verificação seria extremamente útil. Neste caso, a primeira necessidade seria a construção automática de grafos de ocorrência com classes de equivalência compatíveis, que não são suportadas pelas ferramentas de HCPN disponíveis.

Ainda com relação ao ambiente de desenvolvimento, um requisito interessante seria a integração das fases de desenvolvimento. Assim, por exemplo, sempre que o modelo fosse alterado, o desenvolvedor poderia alterar o código no mesmo ambiente, e vice-versa. Além disso, o próprio ambiente poderia "sugerir" que propriedades deveriam ser verificadas novamente mediante tal mudança.

Finalmente, ainda com relação ao ambiente de desenvolvimento, um projeto que está sendo iniciado é a integração de um editor, em primeiro momento, e ferramentas de análise, no segundo momento, para HCPN ao ambiente de desenvolvimento Eclipse[1]. Esta integração deverá englobar, inclusive, o projeto Compor [36]. Esta etapa da integração é para integrar, em um mesmo ambiente, as fases de modelagem/análise e codificação de sistemas. Esta integração de fases visa também a geração automática de código.

[1] http://www.eclipse.org/

Bibliografia

[1] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[2] B. Thomas Adler, L. de Alfaro, L. Dias Da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc: A tool for interface compatibility and composition. Technical Report ucsc-crl-06-01, School of Engineering, University of California, Santa Cruz, 2006.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[5] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification – Model-Checking Techniques and Tools*. Springer, Berlin/Paris/London, 2001.

[6] Soren Christensen and Torben Bisgaard Haagh. *Design/CPN Overview of CPN ML Syntax*. University of Aarhus, 3.0 edition, 1996.

[7] Soren Christensen and Kjeld H. Mortensen. *Design/CPN ASK-CTL Manual*. University of Aarhus, 0.9 edition, 1996.

[8] Gianfranco Ciardo and Philippe Darondeau, editors. *Applications and Theory of Petri Nets 2005, 26rd International Conference, ICATPN 2005, Miami, Florida, USA, June 20-25,*

2005, Proceedings, volume 3536 of *Lecture Notes in Computer Science*. Springer, June 2005.

[9] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM - TOPLAS*, 16(5), September 1994.

[10] E. Clarke, O. Grumberg, and D. Long. Model checking. *Springer-Verlag Nato ASI Series F*, 152, 1996. a survey on model checking, abstraction and composition.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[12] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[13] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[14] Jordi Cortadella and Wolfgang Reisig, editors. *Applications and Theory of Petri Nets 2004, 25rd International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*, volume 3099 of *Lecture Notes in Computer Science*. Springer, June 2004.

[15] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engg.*, 6(1):69–95, 1999.

[16] Ivica Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[17] Ivica Crnkovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan. Specification, implementation, and deployment of components. *Communications of the ACM*, 45(10):35–40, 2002.

[18] Leandro Dias da Silva. Modelagem sistemática de sistemas flexíveis de manufatura baseada em reúso de modelos de redes de petri coloridas. Dissertação de mestrado, Universidade Federal da Paraíba, March 2002.

[19] Leandro Dias da Silva. Modelagem sistemática e formal de sistemas de software baseados em componentes. Relatório técnico, Universidade Federal de Campina Grande, March 2003.

[20] Leandro Dias da Silva. Verificação de propriedades temporais para sistemas embarcados de tempo real usando redes de petri com temporização nebulosa. Relatório técnico, Universidade Federal de Campina Grande, November 2003.

[21] Leandro Dias da Silva, Hyggo Oliveira de Almeida, Angelo Perkusich, and Péricles Rezende de Barros. Model chacking plans for flexible manufacturing systems. In *IFAC World Congress*, Praga, Tchec Republic, 2005.

[22] Leandro Dias da Silva, Hyggo Oliveira de Almeida, Angelo Perkusich, and Evandro de Barros Costa. A coloured petri net model to analyze the design of a multi-agent system. In *Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, Edinburgh, Scotland, UK, May 2004.

[23] Leandro Dias da Silva and Angelo Perkusich. Modelagem sistemática de sistemas flexíveis de manufatura. In *Anais do XIV Congresso Brasileiro de Automática*, Natal, RN - Brasil, September 2002.

[24] Leandro Dias da Silva and Angelo Perkusich. Formal verification of component-based software systems. In *Proceedings of The First International Workshop on Verification and Validation of Enterprise Information Systems VVEIS-2003*, Angers, France, April 2003.

[25] Leandro Dias da Silva and Angelo Perkusich. Uso de realidade virtual para validação de sistema flexíveis de manufatura. In *Anais do VI Simpósio Brasileiro de Automação Inteligente*, Bauru, São Paulo, Brasil, September 2003.

[26] Leandro Dias da Silva and Angelo Perkusich. A systematic and formal approach to the specification of flexible manufacturing systems reusing coloured petri nets models. In *Proceedings of The 11th IFAC Symposium on Information Control Problems in Manufacturing - INCOM'2004*, Salvador, Bahia, Brazil, April 2004.

[27] Leandro Dias da Silva and Angelo Perkusich. Composition of software artifacts modelled using coloured petri nets. *Journal of Science of Computer Programming (Elsevier)*, 56(04/2005):171–189, 2005.

[28] Leandro Dias da Silva and Angelo Perkusich. A model-based approach to formal specification and verification of embedded systems using coloured petri nets. In Colin Atkinson, Christian Bunse, Hans-Gerhard Gross, and Christian Peper, editors, *Component-Based Software Development for Embedded Systems: An Overview on Current Research Trends*, volume 3778 of *LNCS*, pages 35–58. Springer-Verlag, 2005.

[29] Leandro Dias da Silva, Angelo Perkusich, Hyggo Oliveira de Almeida, and Evandro de Barros Costa. Modelling and analysis of a multi-agent intelligent tutoring system based on coloured petri nets. In *Proceedings of The First ACIS International Conference on Software Engineering Research and Applications (SERA'03)*, San Francisco, CA, USA, June 2003.

[30] Leandro Dias da Silva, Angelo Perkusich, and Ana Luíza N. Distéfano. Um arcabouço para modelagem de sistemas de manufatura utilizando redes de petri coloridas e reúso de modelos. In *Anais do V Simpósio Brasileiro de Automação Inteligente*, Canela, RS - Brasil, November 2001.

[31] Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable interfaces. In Bernhard Gramlich, editor, *5th International Workshop on Frontiers of Combining Systems - FROCOS 2005*, volume 3717 of *LNAI*, pages 81–105, Viena, Austria, 2005. Springer-Verlag.

[32] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[33] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, pages pp. 148–165. Lecture Notes in Computer Science 2211, Springer-Verlag, 2001.

[34] Luca de Alfaro, Thomas A. Henzinger, and Marielle I. A. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software*. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[35] Luca de Alfaro and Marielle I. A. Stoelinga. Interfaces: a game-theoretic framework to reason about component-based systems. In *EMSOFT 03: 2nd Intl Workshop on Foundations of Coordination Languages and Software Architectures*, entcs. Elsevier, 2003.

[36] Hyggo Oliveira de Almeida, Leandro Dias da Silva, Elthon Allex da Silva Oliveira, and Angelo Perkusich. A formal approach for component based embedded software modelling and analysis. In *Proceedings of IEEE International Symposium on Industrial Electronics*, volume 4, page 1337, Dubrovinik, Croácia, 2005.

[37] Hyggo Oliveira de Almeida, Leandro Dias da Silva, Angelo Perkusich, and Evandro de Barros Costa. A formal approach for the modelling and verification of multiagent plans based on model checking and petri nets. In Ricardo Choren, Alessandro Garcia, Carlos Lucena, and Alexander Romanovsky, editors, *Software Engineering for Multianget Systems III: Reasearch Issues and Practical Applications*, volume 3390 of *LNCS*, pages 162–179. Springer-Verlag, 2005.

[38] Ana Karla Alves de Medeiros. Mecanismo de interação para um modelo de redes de petri orientado a objetos. Dissertação de mestrado, Universidade Federal da Paraíba, August 2000.

[39] E. Allen Emerson. Temporal and modal logic. In jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models And Semantics, chapter 16, pages 995–1072. Elsevier Science, 1990.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[41] Thomas Genßler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software: the pecos approach. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26. ACM Press, 2002.

[42] K. C. Gorgônio and A. Perkusich. Adaptation of coloured petri nets models of software artifacts for reuse. In *7th International Conference on Software Reuse, Lecture Notes in Computer Science*, Austin, EUA, April 2002.

[43] Kyller Costa Gorgônio. Adaptação de modelos em redes de petri coloridas. Dissertação de mestrado, Universidade Federal da Paraíba, Março 2001.

[44] Kyller Costa Gorgônio and Angelo Perkusich. Síntese de especificações em redes de petri para componentes de software. In *III Workshop de Métodos Formais*, pages 139 – 144, João Pessoa, PB - Brasil, October 2000.

[45] Kyller Costa Gorgônio and Angelo Perkusich. Adaptation of coloured petri nets models of software artifacts for reuse. In Cristina Gacek, editor, *Software Reuse: Methods, Techniques and Tools. VII International Conference on Software Reuse*, number 2319 in Lecture Notes in Computer Science, pages 240–254, Austin, Texas (USA), April 2002. Springer-Verlag.

[46] Klaus Havelund and Tom Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[47] E. Clarke J. R. Burch and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98, June 1992.

[48] K. Jensen and et. al. *"Design/CPN"Manuals*. Meta Software Corporation and Department of Computer Science, University of Aarhus, Denmark, 1996. On-line version:http://www.daimi.aau.dk/designCPN/.

[49] K. Jensen and et. al. *"Design/CPN"4.0*. Meta Software Corporation and Department of Computer Science, University of Aarhus, Denmark, 1999. On-line version:http://www.daimi.aau.dk/designCPN/.

[50] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use*. EACTS – Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[51] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer-Verlag, 1997.

[52] Kurt Jensen, editor. *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, August 28-30, 2002*, volume PB-560. DAIMI, 2002.

[53] Kurt Jensen, editor. *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, October 8-11, 2004*, volume PB-560. DAIMI, 2004.

[54] Kurt Jensen, editor. *Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, October 24-26, 2005*, volume PB-560. DAIMI, 2005.

[55] Edward A. Lee. Embedded software - an agenda for research. Technical Report UCB/ERL No. M99/63, University of California, Berkeley, December 1999.

[56] Edward A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M01/11, University of California, Berkeley, 2001.

[57] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.

[58] Adriano José Pinheiro Lemos. Reúso de modelos em redes de petri coloridas. Dissertação de mestrado, Universidade Federal da Paraíba, 2001.

[59] Adriano José Pinheiro Lemos and Angelo Perkusich. Reuse of coloured petri nets software models. In *Proc. of The Eighth International Conference on Software Engineering and Knowledge Engineering, SEKE'01*, pages 145–152, Buenos Aires (Argentina), June 2001.

[60] A.J.P. Lemos and A. Perkusich. Reuse of coloured petri nets software models. In *Proc. of The Eighth International Conference on Software Engineering and Knowledge Engineering, SEKE'01*, pages 145–152, Buenos Aires, Argentina, June 2001.

[61] Kenneth L. McMillan. *Symbolic Model Checking*. The Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.

[62] Theo Dirk Meijler and Oscar Nierstrasz. Beyond objects: Components. In M. P. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems: Current Trends and Directions*, pages 49–78. Academic Press, November 1997.

[63] Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[64] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genßler, and Reinier van den Born. A component model for field devices. *Lecture Notes in Computer Science*, 2370:200–216, 2002.

[65] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[66] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390. Springer-Verlag, 1994.

[67] Angelo Perkusich, Hyggo O. Almeida, and Denis H. de Araujo. A software framework for real-time embedded automation and control systems. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, volume 2, Lisbon, Portugal, 2003.

[68] Amir Pnueli and Yonit Kesten. Modularization and abstraction: The keys to practical formal verification. In *23rd Int. Symp. Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 54–71. Springer-Verlag, 1998.

[69] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[70] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[71] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1999.

[72] Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165. Springer-Verlag, 1991.

[73] Wil M. P. van der Aalst and Eike Best, editors. *Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings*, volume 2679 of *Lecture Notes in Computer Science*. Springer, June 2003.

[74] Wayne Wolf. Embedded is the new paradigm(s). *IEEE Computer*, 37(3), March 2004.