Universidade Federal da Bahia
Escola Politécnica / Instituto de Matemática
Programa de Pós-Graduação em Mecatrônica

André Luís Nunes Muniz

TANGRAM: UMA FERRAMENTA DE APOIO À VERIFICAÇÃO FORMAL DE SISTEMAS DE TEMPO REAL BASEADOS EM COMPONENTES

DISSERTAÇÃO DE MESTRADO

Salvador
2009

André Luís Nunes Muniz

TANGRAM: UMA FERRAMENTA DE APOIO À VERIFICAÇÃO FORMAL DE SISTEMAS DE TEMPO REAL BASEADOS EM COMPONENTES

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica da Escola Politécnica e do Instituto de Matemática, Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre.

Orientadora: Profa. Dra. Aline Maria Santos Andrade
Co-orientador: Prof. Dr. George Marconi de Araújo Lima

Salvador
2009

Sistema de Bibliotecas - UFBA

Muniz, André Luís Nunes.
Tangram : uma ferramenta de apoio à verificação formal de sistemas de tempo real baseados em componentes / André Luís Nunes Muniz. - 2010.
133 f. : il.

Orientadora: Profa. Dra. Aline Maria Santos Andrade.
Co-orientador: Prof. Dr. George Marconi de Araújo Lima.
Dissertação (mestrado) - Universidade Federal da Bahia, Instituto de Matemática e Escola Politécnica, Salvador, 2009
.

1. Projeto de sistemas. 2. Componente de software. 3. UML (Computação). 4. Sistemas de computação. I. Andrade, Aline Maria Santos. II. Lima, George Marconi de Araújo. III. Universidade Federal da Bahia. Instituto de Matemática. IV. Universidade Federal da Bahia. Escola Politécnica. V. Título.

CDD - 005.11
CDU - 681.3.02

TERMO DE APROVAÇÃO

ANDRÉ LUÍS NUNES MUNIZ

TANGRAM: UMA FERRAMENTA DE APOIO À VERIFICAÇÃO FORMAL DE SISTEMAS DE TEMPO REAL BASEADOS EM COMPONENTES

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia, pela seguinte banca examinadora:

Dra. Aline Maria Santos Andrade – Orientadora
Doutora em Informática pela Pontifícia Universidade Católica do Rio de Janeiro, Brasil
Universidade Federal da Bahia - UFBA

Dr. Jean-Marie Farines
Doutor em Engenharia Elétrica pelo Institut National Polytechnique de Toulouse, França
Universidade Federal de Santa Catarina - UFSC

Dr. Raimundo José de Araújo Macêdo
Doutor em Ciência da Computação pela University of Newcastle Upon Tyne, Inglaterra
Universidade Federal da Bahia - UFBA

Salvador, 30 de outubro de 2009

Dedico esta dissertação aos meus pais, Dalila e Severino, pelo carinho e pelo esforço empregado por eles na minha formação. Dedico também à minha noiva e futura esposa, Luiza, por estar ao meu lado em todos os momentos deste mestrado e por dar toda a motivação de que eu precisava para concluir esta dissertação. Sem vocês eu não teria chegado até aqui.

AGRADECIMENTOS

Gostaria de agradecer primeiramente aos meus orientadores, Aline e George, pela excelente orientação que recebi deles e pela dedicação com a qual me conduziram neste trabalho. Agradeço também aos demais professores do PPGM, pelo conhecimento e experiência transmitidos nas aulas e seminários.

Agradeço aos meus colegas de curso, pelo companherismo e pela troca de experiências durante as disciplinas iniciais do mestrado. Agradeço aos alunos de graduação Paulo e Anderson, por terem contribuído diretamente na realização deste trabalho.

Agradeço ao amigo Aryldo (Dinho), que mesmo em São Paulo, deu grandes contribuições para esta dissertação. Agradeço aos professores que fizeram parte da banca examinadora, Macêdo e Jean-Marie, pela aprovação, elogios, sugestões e críticas construtivas apresentadas em minha defesa.

Agradeço às agências de fomento CAPES e FAPESB pelo apoio financeiro dado a este projeto.

Agradeço a toda minha família pelo carinho e pelo apoio prestado para que eu pudesse atingir mais este objetivo na vida. Agradeço aos amigos, pela compreensão por eu não estar presente nos momentos em que tive que me dedicar mais ao mestrado do que a eles.

A todos vocês, que de algum modo contribuíram para a realização deste sonho, o meu sincero muito obrigado!

Nós somos aquilo que fazemos com frequência, portanto, a excelência não é um ato, porém um hábito.

—ARISTÓTELES (384/322 A.C.)

RESUMO

Sistemas computacionais são utilizados atualmente em aplicações consideradas críticas e com alto grau de complexidade, como é o exemplo da automação industrial, controle de navegação em aeronaves, equipamentos médicos, entre outros. Com o intuito de lidar com esta complexidade e, ao mesmo tempo, dar garantias de confiabilidade, novas abordagens de desenvolvimento e validação de *software* têm sido empregadas neste contexto. No que tange os chamados sistemas de tempo real, nos quais a correção do sistema depende do cumprimento de suas restrições temporais, duas abordagens vem ganhando muita atenção nos últimos anos, o desenvolvimento baseado em componentes, ou CBD (*Component-Based Development*), e os métodos formais de verificação de *software*, com atenção especial para a verificação de modelos (Model-Checking), que está entre as técnicas formais mais utilizadas na indústria e na academia. Entretanto, existem poucas abordagens propostas no sentido de aplicar a verificação de modelos a sistemas de tempo real críticos baseados em componentes. Um dos principais desafios nesta área é a falta de abordagens/ferramentas que dêem suporte a uma fácil integração dos métodos formais ao processo de desenvolvimento baseado em componentes. Isto é causado muitas vezes pela falta de compatibilidade entre as abordagens atuais de verificação formal e os diversos modelos de componentes existentes no mercado. Diante disto, este trabalho propõe uma abordagem para a integração da verificação de modelos ao processo de desenvolvimento baseado em componentes para sistemas de tempo real, através de uma tradução automática de modelos semi-formais de sistemas baseados em componentes para modelos formais passíveis de verificação. A linguagem de modelagem considerada neste trabalho é a UML (*Unified Modeling Language*), a qual já é um padrão em termos de especificação de sistemas, sendo largamente utilizada na indústria e na academia, para diversos tipos de aplicação e abordagens de desenvolvimento. Os modelos formais gerados pela tradução são autômatos temporizados do verificador de modelos UPPAAL, o qual é um verificador de modelos voltado para sistemas de tempo real. Para dar suporte à abordagem deste trabalho, foi desenvolvida uma ferramenta chamada TANGRAM (*Tool for Analysis of Diagrams*), a qual é capaz de traduzir diagramas da UML em autômatos temporizados. Um estudo de caso da utilização de TANGRAM foi realizado e é apresentado neste trabalho.

Palavras-chave: Sistemas de tempo real, componentes, UML, UPPAAL.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

A evolução de *software* e *hardware* nos últimos anos gerou uma rápida expansão dos sistemas computacionais nas mais diversas áreas da sociedade. Desde o ambiente doméstico até os setores industriais mais complexos, a maioria das funções e processos que eram executados por pessoas ou equipamentos puramente mecânicos passaram a ser controlados por computador, o que deu origem aos sistemas mecatrônicos. A mecatrônica é a área de conhecimento que busca a criação de produtos e processos capazes de desempenhar as suas atividades a partir da integração sinérgica entre a mecânica, eletrônica e a computação [4]. A propagação dos sistemas mecatrônicos deu-se significativamente através dos chamados sistemas embarcados. Um sistema embarcado (ou embutido) é um sistema computacional que desempenha uma função dentro de um sistema maior[5]. Exemplos de sistemas embarcados podem ser encontrados hoje em eletrodomésticos, celulares, automóveis, aeronaves, ferramentas industriais, robôs, e diversos outros dispositivos.

Muitos destes sistemas têm sido utilizados para a realização de atividades consideradas críticas, tais como controle de processos industriais, aeronaves e equipamentos médicos. Em geral, estas aplicações possuem um alto grau de interatividade com o ambiente, por exemplo, um sistema de controle de processo industrial deve monitorar diversas variáveis do ambiente medidas através de sensores e agir sobre este ambiente através dos atuadores. Dessa forma, este sistema deve estar sincronizado com o mundo real e respeitar as suas restrições temporais. A este tipo de sistema dá-se o nome de Sistema de Tempo Real. Um sistema de tempo real é aquele cuja correção não depende apenas da realização das suas funções, mas depende também do cumprimento de prazos (*deadlines*) associados à estas funções. Os sistemas que são obrigados a cumprir todos os *deadlines* são chamados

de sistemas de tempo real *hard* ou críticos. Geralmente, esta obrigatoriedade vem do fato de que a perda de um *deadline* pode pôr em risco vidas humanas ou o patrimônio de alguém. Por exemplo, um sistema que controla os freios de um automóvel é crítico, pois um atraso em sua execução pode causar um acidente grave. Já os sistemas em que a perda de *deadlines* não provoca prejuízos graves são chamados de sistemas *soft* ou não-críticos. Sistemas multimídia são considerados não-críticos, pois um atraso na sincronização entre áudio e vídeo, por exemplo, podem passar despercebidos.

Para lidar com a complexidade do desenvolvimento de sistemas embarcados de tempo real, uma abordagem que tem se destacado é o Desenvolvimento Baseado em Componentes ou CBD (*Component-Based Development*) [5]. Segundo esta abordagem, um sistema deve ser construído a partir da junção harmônica entre partes de *software* pré-concebidas (componentes), de modo que o resultado final atenda às necessidades especificadas. A idéia é que cada componente seja uma unidade de *software* auto-contida capaz de oferecer determinadas funcionalidades (serviços) para os demais componentes. O acesso aos serviços de um componente é feito através de interfaces bem definidas. Dessa forma, pode-se montar um sistema a partir de componentes desenvolvidos por diferentes grupos, desde que as interfaces entre os componentes sejam compatíveis.

A forma de se especificar um componente, suas interfaces e sua implementação deve ser definida por um modelo de componentes. Atualmente não existe um modelo de componentes padrão, utilizado por todos os desenvolvedores. Dentre os modelos de componentes mais difundidos comercialmente, podemos destacar o *Enterprise JavaBeans* (EJB) [6], o COM (Component Object Model) [7] e o Modelo de Componentes CORBA ou CCM (*CORBA Component Model*) [8].

Em geral, os sistemas baseados em componentes devem possuir uma infra-estrutura de suporte à criação, execução e comunicação destes componentes. Esta infra-estrutura, mais conhecida como *middleware* de componentes, implementa todo o mecanismo de gerenciamento dos mesmos e oferece ao desenvolvedor uma abstração sobre o sistema operacional e *hardware* no qual cada componente será executado.

No contexto dos sistemas de tempo real embarcados, a especificação dos componentes deve levar em consideração os requisitos temporais da aplicação, suas restrições de memória e consumo de energia, e outros aspectos relacionados à confiança no funcionamento (*dependability*) do sistema [5]. No entanto, os modelos de componentes mais utilizados atualmente não oferecem mecanismos para tratar este tipo de requisito. De modo a contornar este problema, pesquisas recentes têm caminhado nesta direção, a fim de possibilitar o desenvolvimento baseado em componentes para sistemas de tempo real com um certo grau de previsibilidade sobre a execução dos componentes. Um importante resultado nesta área é o *middleware* de componentes para aplicações de tempo real chamado CIAO (*Component-Integrated ACE ORB*) [9], implementado a partir do padrão CCM com extensões para tratar requisitos de tempo. Pesquisas recentes [3, 1] têm demonstrado a importância deste *middleware* e destacam as suas vantagens no contexto de sistemas mecatrônicos e de tempo real.

Embora os avanços na área de projeto e implementação de sistemas de tempo real baseados em componentes tenham sido significativos, pouco tem sido feito no que diz respeito à sua validação formal [2]. No que tange o domínio dos sistemas de tempo real críticos, métodos formais de verificação são largamente utilizados com o intuito de prover garantias de correção de *software*. Através dos métodos formais, é possível dirimir a presença de erros de projeto no sistema, ainda em fases iniciais de desenvolvimento. Particularmente, a utilização da técnica de verificação de modelos (*model-checking*) [10] tem sido um tema bastante pesquisado ultimamente, tal como em [11, 12] e outros. Entretanto, muitos destes esforços estão associados a modelos de componentes próprios, ou pouco difundidos, o que consequentemente levará a uma diversificação dos *middlewares* necessários à execução de cada um desses modelos. Além disso, as abordagens existentes geralmente são direcionadas a um domínio específico de aplicação como, por exemplo, sistemas embarcados em aeronaves [13]. Desta forma, faltam abordagens mais genéricas, que possam ser aplicadas em diversos domínios de sistemas de tempo real baseados em componentes.

1.1 OBJETIVOS

O presente trabalho propõe uma ferramenta chamada TANGRAM (***T**ool for **An**alysis of Dia**gram**s*) para integrar a verificação de modelos ao processo de desenvolvimento baseado em componentes para sistemas de tempo real. Esta integração será feita através da tradução de sistemas baseados em componentes modelados em UML para modelos de autômatos com tempo do verificador de modelos UPPAAL [14]. O modelo de componentes utilizado como base para a construção do sistema será o CCM e o *middleware* de componentes CIAO será considerado como plataforma de execução dos mesmos.

O uso da UML e do CCM na construção dos modelos de componentes contribui para uma generalização em termos de aplicação da ferramenta proposta. Isto se deve ao fato de que a UML é uma linguagem de modelagem flexível e independente do tipo de aplicação que está sendo desenvolvida. Contudo, embora a UML seja um padrão para projeto de sistemas [15], ela é uma linguagem semi-formal, e por isso não é passível de verificação automática. Portanto, é necessário mapear os modelos da UML em modelos formais, tais como os autômatos com tempo do UPPAAL.

O objetivo geral deste trabalho deverá ser atingido através das seguintes contribuições:

- proposição de uma abordagem para a construção de modelos de sistemas baseados em CCM/CIAO utilizando diagramas UML estendidos;
- elaboração de uma tradução entre sistemas de componentes modelados em UML para autômatos temporizados de UPPAAL, levando-se em consideração os aspectos do *middleware* que podem ser incorporados ao modelo gerado;
- implementação e validação da ferramenta proposta para a tradução dos modelos;
- realização de um estudo de caso para medir o impacto desta abordagem sobre o processo de desenvolvimento baseado em componentes para sistemas de tempo real, analisando quais foram os benefícios trazidos e os problemas encontrados durante o desenvolvimento.

Vale ressaltar que não faz parte dos objetivos deste trabalho a verificação da correção do *middleware* de componentes, o que será assumido como correto. As características do *middleware* CIAO estão sendo consideradas neste trabalho para verificar a correção da aplicação submetidas às suas políticas e para medir o impacto da inclusão destas características sobre o tamanho do espaço de estados gerado. Alguns autores [11] afirmam que a inclusão de funcionalidades do *middleware* ao modelo formal contribui para a diminuir o espaço de estados. Além disso, esta abordagem permite uma verificação com um nível maior de detalhe sobre as funções do sistema.

1.2 TRABALHOS RELACIONADOS

Em [11] é proposto um ambiente, chamado Cadena, para modelagem e construção de sistemas baseados no *framework Bold Stroke* da Boeing, que possui características semelhantes às do CCM. Porém, enquanto que no CCM é possível "plugar" e "desplugar" componentes dinamicamente, no *Bold Stroke* é adotada uma política comum em sistemas críticos, que é a de alocar e configurar os componentes estaticamente, apenas durante a inicialização do sistema. Entre outras coisas, o ambiente permite a verificação de propriedades de correção de modelos de sistemas derivados a partir da IDL (*Interface Definition Language*).

No Cadena, a aplicação da verificação de modelos está baseada na tradução em camadas a partir das linguagens de especificação derivadas do CCM para a linguagem do verificador de modelos *dSPIN* [16]. Este verificador permite suporte a recursos como objetos, funções e referências, entre outras funcionalidades, o que não se encontra comumente em outros verificadores. A tradução inclui a modelagem de: (i) comportamento e interfaces dos componentes do projeto, (ii) a semântica dos componentes do *middleware*, e (iii) elementos do ambiente que interagem com o sistema.

O *middleware* fornece as *threads* nas quais os componentes executam e define a sua política de escalonamento. Segundo os autores, quando tal política está disponível, ela pode ser explorada para reduzir o espaço de estados do modelo gerado através da elim-

inação de intercalações de execuções de *threads* que violem a política de escalonamento. Esta eliminação também possui um efeito benéfico de melhorar a precisão da análise realizada sobre o modelo.

No caso do Cadena, o *Bold Stroke* utiliza a política de escalonamento *Rate Monotonic* [17]. Tal como a maioria dos verificadores de modelo, o dSPIN não possui a definição de uma política de escalonamento específica, então o Cadena codifica esta política diretamente no modelo do dSPIN. Isto é feito através da restrição de todas as ações do componente com um teste que bloqueia a ação se a prioridade do componente for menor que a prioridade executável no momento.

Além das funcionalidades do *middleware*, o Cadena também incorpora os eventos disparados pelo ambiente aos modelos especificados no dSPIN, com o objetivo de melhorar a qualidade da verificação. Vale ressaltar que a semântica desses eventos e o padrão de chegada de eventos no sistema variam para cada aplicação.

Outra abordagem para aplicação de verificação de modelos sobre sistemas embarcados de tempo real baseados em componentes é apresentado em [18]. Neste trabalho, os autores utilizam uma álgebra de processos chamada *Finite State Processes* (FSP) para especificar formalmente aspectos dinâmicos do comportamento de sistemas utilizados em aeronaves. O verificador de modelos utilizado é o *Labeled Transition System Analyser* (LTSA). As propriedades que podem ser verificadas são ausência de *deadlock*, *reachability*, progresso e restrições de sequenciamento.

Entretanto, ao contrário do que foi proposto pelo Cadena, a abordagem em [18] não contempla a tradução automática dos modelos de componentes para os modelos formais em FSP. Outra diferença é que ela não incorpora características do *middleware*. Segundo os autores, esta abordagem demonstrou ser adequada para verificação de propriedades de concorrência no nível da aplicação, com um alto nível de abstração. Neste caso, os autores afirmam que ignorar detalhes do *middleware* subjacente, tais como políticas de escalonamento e transmissão de eventos, pode reduzir significativamente o espaço de estados do sistema.

Em [19], os autores apontam dois problemas na integração da verificação formal com a tecnologia baseada em componentes: o primeiro está na concepção de um modelo comum que seja formal o suficiente para ser passível de verificação e que ao mesmo tempo possibilite a modelagem dos componentes com todos os seus detalhes; o segundo problema está em aliar a metodologia de desenvolvimento baseada em componentes com uma plataforma de verificação formal. Para resolver o primeiro problema, os autores propõem um modelo intermediário entre um modelo de componentes e um modelo formal, chamada de *Formal Object-Oriented Model* (FOOM). Para resolver o segundo problema é proposta a utilização de uma técnica chamada *Formal Syntesis and Model Checking* (FSMC), que pressupõe a utilização de métodos formais também durante a composição do sistema. As propriedades verificadas se limitam a um conjunto de restrições temporais especificadas em *Timed Computation Tree Logic* (TCTL).

Em [20] é proposta uma plataforma chamada VERCORS para verificação semi-automática de aplicações construídas a partir de componentes hierárquicos distribuídos, que seguem o modelo de componentes Fractal
[21]. Nesta plataforma, o usuário deve entrar com a descrição arquitetural dos componentes mais a especificação comportamental dos mesmos. Em seguida, um conjunto de ferramentas que compõe a plataforma executam a tradução destes modelos para um formalismo proposto pelos próprios autores em outro trabalho, chamado pNet
[22], que consiste em um modelo parametrizado que estende uma rede de autômatos comunicantes. A partir destes modelos parametrizados, a plataforma gera uma instância dos mesmos a partir de parâmetros específicos do domínio, os quais também devem ser especificados pelo usuário. O resultado disto é uma rede hierárquica de sistemas de transições rotuladas (ou *Labeled Transition Systems*), as quais por fim são mapeadas para o formato de entrada do verificador de modelos CADP [23]. Para este verificador, as propriedades foram escritas em fórmulas do μ-*calculus* [24].

Uma diferença importante da abordagem de [20] para as demais que foram descritas anteriormente está no modelo de componentes utilizado. De acordo com seus autores, o modelo Fractal foi utilizado por ser hierárquico, isto é, um componente pode ser formado

a partir da composição de outros componentes, o que não é contemplado pelo CCM, por exemplo.

A abordagem apresentada em [25] possui um foco sobre a verificação de propriedades relacionadas à compatibilidade comportamental entre componentes. Os autores propõem um modelo de componentes chamado Kmelia, no qual as interfaces de componentes são especificadas através de serviços requeridos e fornecidos. Para cada serviço fornecido por um componente, é associado um comportamento especificado por um sistema de transição de estados rotulados estendido (*extended labelled transition system* - eLTS). Para realizar a verificação, os componentes do Kmelia são transformados em processos da linguagem LOTOS [26] e analisados pelo *kit* de ferramentas CADP.

Em [12], o verificador de modelos UPPAAL também é utilizado para verificar propriedades sobre sistemas de tempo real baseados em componentes. O modelo de componentes utilizado foi o *SaveComp Component Model*, ou SaveCCM [27]. O foco do trabalho foi definir uma semântica formal, baseada em autômatos temporizados, para os elementos do núcleo (*core*) da linguagem de modelagem do SaveCCM. Os autores em [12] mostraram que a semântica definida por eles poderia ser utilizada para verificação de propriedades não-triviais através da ferramenta UPPAAL. No entanto, não foi apresentada uma ferramenta que desse suporte à tradução dos elementos do SaveCCM para os autômatos temporizados de forma automática.

Um *framework* chamado DREAM (*Distributed Real-Time Embedded Analysis Method*) [13] foi proposto direcionado à análise de escalonamento não-preemptivo para sistemas embarcados em aeronaves. DREAM utiliza uma linguagem específica de domínio (DSML) associada ao *middleware* Bold Stroke. O *framework* traduz os modelos de componentes em autômatos temporizados, sobre os quais o verificador de modelos UPPAAL é aplicado. Os autores propõem uma abordagem que transforma a análise de escalonamento em uma verificação de propriedades de *reachability*. Isto é feito através da inclusão de uma localização que representa a perda do *deadline* em cada autômato relativo a uma tarefa do sistema. Se o verificador de modelos provar que esta localização pode ser

alcançada, significa que o *deadline* daquela tarefa será perdido durante a execução do sistema. Funcionalidades do *middleware* também são incorporadas ao modelo de autômatos pelo DREAM, como a transmissão baseada em eventos, *timeouts* periódicos e principalmente a política de escalonamento. Entretanto, os detalhes sobre a execução das tarefas são abstraídos, pois o objetivo é verificar a escalonabilidade. Desta forma, propriedades mais detalhadas sobre as funcionalidades do sistema não podem ser verificadas.

A abordagem proposta pelo presente trabalho possui diferenças em relação às demais em diversos aspectos. Primeiramente, pode-se destacar o modelo de componentes utilizado, o CCM, o qual é um modelo flexível, interoperável e independente de plataforma. Uma outra diferença a ser destacada é o uso da UML como linguagem de modelagem dos sistemas baseados em componentes, o que não foi considerado em nenhuma das abordagens pesquisadas. Outra característica deste projeto é a tradução automática dos modelos de componentes para os autômatos temporizados do UPPAAL. Este verificador também foi utilizado em [12] e [13]. No primeiro, porém, não foi apresentada uma ferramenta para gerar os autômatos de forma automática. No segundo, o foco está sobre a verificação de escalonabilidade e, por este motivo, o comportamento interno dos componentes é abstraído nos autômatos gerados em uma única localização chamada *Run*. Consequentemente, propriedades funcionais da aplicação não poderão ser verificadas, ao contrário do que propõe o presente trabalho. Por fim, nem todos os trabalhos pesquisados consideraram a integração das funcionalidades do *middleware* aos modelos gerados como forma de reduzir o espaço de estados durante a verificação.

1.3 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está estruturada da seguinte maneira: no capítulo 2 são apresentados os principais conceitos relacionados a sistemas de tempo real, Desenvolvimento Baseado em Componentes, o Modelo de Componentes Corba (CCM) e o *middleware* de componentes para tempo real CIAO. O capítulo 3 apresenta os diagramas da UML utilizados neste trabalho e propõe uma abordagem para especificação de sistemas baseados em CCM

e CIAO com estes diagramas. Os conceitos relacionados à verificação de modelos para sistemas de tempo real e a ferramenta UPPAAL são apresentados no capítulo 4. Os principais resultados deste trabalho e a proposta da ferramenta TANGRAM são apresentados no capítulo 5. No capítulo 6, é apresentado um estudo de caso da utilização do TANGRAM sobre um sistema mecatrônico real utilizado em estações de trem e metrô. O último capítulo apresenta as conclusões do trabalho e perspectivas de trabalhos futuros.

CAPÍTULO 2

DESENVOLVIMENTO BASEADO EM COMPONENTES PARA SISTEMAS EMBARCADOS DE TEMPO REAL

O principal objetivo do Desenvolvimento Baseado em Componentes (*Component-Based Development* - CBD) é proporcionar a construção de sistemas a partir da reutilização de partes de *software* já implementadas, ao invés de se desenvolver todo o código desde o início. Embora a reutilização de *software* seja um tema discutido há muito tempo, nesta abordagem ela adquire um caráter mais amplo, que independe de paradigma de programação. Para o CBD, reutilizar um componente significa poder implantá-lo num sistema de forma que não seja necessário recompilar toda a aplicação, podendo até mesmo ser feita em tempo de execução.

Existem diferentes sugestões na literatura para uma definição precisa do que seria um componente. A mais aceita atualmente é a de Szyperski [28], na qual um componente é definido como uma unidade de composição que pode ser implantada de forma independente e está sujeita à composição por terceiros, ou seja, não necessariamente um componente será implantado pela mesma pessoa que o desenvolveu.

Embora não haja uma definição padrão para componentes, todas as definições concordam que um componente seja uma unidade de composição e que deve ser especificado de forma que seja possível integrá-lo em um sistema de maneira previsível
[2]. Para que um componente seja integrado, atualizado e mantido de forma correta, o mesmo deve consistir dos seguintes elementos: (i) um conjunto de interfaces fornecidas e requeridas pelo componente; (ii) um código executável que pode ser acoplado ao de

outros componentes através das interfaces.

As vantagens trazidas pela reutilização de software são muitas e bastante conhecidas. Dentre as vantagens no reuso de componentes, podemos destacar: aumento da produtividade, melhoria de qualidade, maior flexibilidade, escalabilidade e maior interoperabilidade. Entretanto, [2] apontam algumas desvantagens deste conceito, as quais são: (i) construir unidades genéricas e reusáveis pode implicar em maior complexidade de projeto e implementação do componente em si; (ii) uma alteração no ambiente de execução, na aplicação ou nos componentes pode provocar uma falha em um componente devido a características desconhecidas pelo desenvolvedor sobre aquele componente.

Uma dificuldade encontrada no desenvolvimento baseado em componentes para sistemas de tempo-real é que ainda não há um padrão para a especificação de atributos não-funcionais nas interfaces, tais como restrições temporais e outros atributos de qualidade de serviço [5]. No entanto, este é um tópico que vem sendo bastante pesquisado na academia e algumas abordagens têm sido propostas [11, 29, 30].

2.1 SISTEMAS DE TEMPO REAL

Sistemas de tempo real (STR) são aqueles que possuem, além de requisitos tradicionais a todos os sistemas de computação, requisitos de tempo. Mais especificamente, estes requisitos impõem restrições sobre o tempo de execução das tarefas executadas pelo sistema [31]. O termo tarefas é aqui empregado pois este tipo de sistema é geralmente visto como um sistema concorrente, no qual existem várias tarefas (ou *threads*) compartilhando um ou mais processadores.

A correção de um STR não depende apenas da correção lógica das suas funções, mas depende também do cumprimento destas funções dentro dos prazos estabelecidos (*deadlines*). O termo *deadline* é utilizado para representar o prazo máximo que uma tarefa possui para executar completamente. Os STR que são obrigados a cumprir todos os *deadlines* são chamados de sistemas de tempo real *hard* ou críticos. Geralmente, esta

obrigatoriedade vem do fato de que a perda de um *deadline* pode pôr em risco vidas humanas ou o patrimônio de alguém. Por exemplo, um sistema que controla os freios de um automóvel é crítico, pois um atraso em sua execução pode causar um acidente grave. Já os sistemas em que a perda de *deadlines* não provoca prejuízos graves são chamados de sistemas *soft* ou não-críticos. Sistemas multimídia são considerados não-críticos, pois um atraso na sincronização entre áudio e vídeo, por exemplo, podem passar despercebidos.

Diante disto, é possível perceber que o projeto de um STR requer a identificação antecipada de que suas tarefas serão executadas dentro dos prazos especificados. A esta característica dá-se o nome de previsibilidade. No entanto, garantir a previsibilidade nem sempre é trivial, pois ela possui implicações em praticamente todos os níveis do STR (infraestrutra física, sistema operacional, linguagem de programação, etc).

Desta forma, é necessário que o ambiente de execução forneça um gerenciamento adequado dos recursos computacionais a fim de que as tarefas não sofram atrasos imprevistos, por exemplo, devido à indisponibilidade de memória, CPU, banda passante na rede de comunicação ou mesmo a execução simultânea de diferentes tarefas (acionadas a partir de estímulos diversos do ambiente). O trabalho de distribuir os recursos computacionais necessários para que o sistema funcione respeitando suas restrições temporais é definido como *escalonamento*. As regras que determinam como os recursos serão alocados às tarefas durante a execução do sistema são definidas por uma política de escalonamento.

Em geral, as políticas de escalonamento são construídas a partir dos atributos associados às tarefas do sistema. Por exemplo, existem políticas que consideram o período de ativação das tarefas como fator determinante para alocação de recursos. O período de uma tarefa é o intervalo entre duas execuções consecutivas da mesma. Quando uma tarefa possui um período conhecido e constante durante toda a execução do sistema, ela é chamada de periódica. Caso contrário, ela pode ser esporádica ou aperiódica. No primeiro caso, a tarefa não possui um período definido, porém sabe-se que existe um intervalo mínimo permitido entre duas ativações. No segundo, nada se sabe sobre o momento em que a tarefa será ativada.

A partir da avaliação dos atributos das tarefas, a política de escalonamento atribui uma prioridade a cada uma delas. Em geral, a tarefa que possuir a maior prioridade em um dado momento, terá o direito de utilizar os recursos computacionais. Neste sentido, as políticas de escalonamento podem ser divididas em dois grupos: (i) aquelas em que as prioridades podem ser definidas em tempo de projeto (prioridade fixa); ou (ii) aquelas em que as prioridades são definidas em tempo de execução (prioridade variável). Exemplos clássicos de políticas de prioridade fixa e variável são *Rate Monotonic* (RM) e *Earliest Deadline First* (EDF) [17], respectivamente. Na RM, as tarefas com menor período recebem maior prioridade. Como o período das tarefas não muda, é possível determinar a sequência de execução das tarefas em tempo de projeto. De acordo com a EDF, a tarefa mais prioritária em um dado instante de execução será aquela com *deadline* mais próximo. Desta forma, a prioridade de cada tarefa somente pode ser determinada em tempo de execução.

Outra característica importante das políticas de escalonamento é a capacidade de interromper a execução de uma tarefa menos prioritária para dar lugar a uma de maior prioridade. Esta característica é também chamada de preempção. As políticas preemptivas são geralmente mais flexíveis do que as não-preemptivas e conseguem atender a uma maior variedade de domínios de aplicação. As duas políticas citadas como exemplo no parágrafo anterior são preemptivas.

Em termos de previsibilidade, existem meios analíticos de se determinar *a priori* se um dado conjunto de tarefas é escalonável (atende aos *deadlines*) com base na política de escalonamento e nos atributos conhecidos de cada tarefa. Em geral, esta *análise de escalonamento* é feita através de fórmulas matemáticas que exprimem o comportamento temporal do sistema no pior caso. Neste contexto, duas abordagens são comumente utilizadas: (i) cálculo do limite de utilização do processador e (ii) cálculo do tempo de resposta das tarefas [32]. No primeiro caso, uma fórmula é usada para determinar o percentual máximo de utilização do processador. Caso o somatório de utilização de cada tarefa exceda este valor limite, significa que o conjunto de tarefas não é escalonável. No segundo caso, calcula-se o tempo de resposta no pior caso para cada tarefa do sistema.

Se o valor do tempo de resposta ultrapassar o valor do *deadline*, significa que a tarefa poderá não cumprir o seu prazo durante a execução. Vale ressaltar que, em alguns casos, a análise baseada na utilização do processador é apenas suficiente, ou seja, é possível que um conjunto de tarefas ultrapasse o limite indicado pela fórmula, mas seja escalonável. Isto pode ocorrer para a política RM, por exemplo.

Além dos conceitos apresentados nesta seção, a área de tempo real possui diversos outros desafios, tais como interdependência entre tarefas, tratamento de *jitter*, compartilhamento de recursos, entre outros [33]. Não faz parte do escopo deste trabalho explorar estes conceitos, portanto os mesmos não serão abordados aqui.

As seções a seguir apresentam os conceitos relacionados ao Desenvolvimento Baseado em Componentes e como esta abordagem tem sido utilizada com o objetivo de reduzir o esforço no desenvolvimento de STR, melhorar sua qualidade e fornecer os insumos necessários para dar garantias de correção temporal.

2.2 COMPONENTES

2.2.1 Interfaces

Segundo Szyperski [28], uma interface de um componente é o seu ponto de acesso, sendo que cada interface define um serviço que o componente oferece. Em geral, cada componente é responsável por oferecer mais de um serviço, portanto ele deve possuir mais de uma interface. Desta forma, as interfaces separam a implementação concreta dos serviços de um componente em relação aos seus possíveis clientes.

O uso de interfaces traz duas vantagens que devem ser destacadas. A primeira é a possibilidade de substituir a implementação de um componente sem mudar a sua interface. Com isso, pode-se, por exemplo, melhorar o desempenho de uma parte do sistema sem a necessidade de reconstruí-lo por inteiro. A segunda vantagem está em adicionar novas interfaces ao componente, sem modificar as que já existem, o que aumenta o poder de

adaptação do componente a outros sistemas (reusabilidade).

Podem-se dividir as interfaces de um componente em dois tipos: as interfaces fornecidas (ou exportadas) e as requeridas (ou importadas). As interfaces fornecidas especificam os serviços que aquele componente implementa e oferece ao ambiente. As interfaces requeridas, por outro lado, especificam os serviços dos quais o componente depende e que devem ser oferecidos por outros componentes. A composição do sistema se dá basicamente através do acoplamento entre as interfaces requeridas de um componente com as fornecidas pelos demais.

2.2.2 Modelo de Componentes e Middleware

As abordagens de desenvolvimento existentes até o surgimento do CBD ainda não supriam completamente as necessidades dos desenvolvedores que se deparavam com sistemas complexos, desenvolvidos em diversas linguagens e paradigmas de programação. Estas e outras razões motivaram a necessidade pela criação de modelos e *middlewares* baseados em componentes que preenchessem as lacunas deixadas pelas outras abordagens. Em particular, o CBD surgiu como uma melhoria em relação ao paradigma orientado a objetos [34].

De acordo com o apresentado em [3], o CBD apresenta as seguintes características:

- padronização na comunicação entre componentes. Os pontos de acesso do componente são bem defindos pelas interfaces, de modo que este não precisa "conhecer" os componentes com os quais ele está se comunicando;
- cada componente em execução é encapsulado por uma estrutura chamada *container*, responsável por intermediar qualquer interação daquele componente com os demais e gerenciar o seu ciclo de vida;
- separação dos atributos não-funcionais, os quais são gerenciados pelo *container* e podem ser configurados através de descritores externos;

- criação do sistema a partir da montagem de componentes pré-construídos. A conexão entre componentes também é definida em descritores de implantação, os quais podem ser gerados com o auxílio de ferramentas visuais, evitando assim tarefas enfadonhas e propensas a erros.

Um modelo de componentes define: (i) as propriedades dos componentes, tais como versão de cada componente, dependência entre componentes, modos de acesso, identificadores de componentes e políticas operacionais; (ii) o conjunto de interfaces que os componentes exportam; e (iii) a infra-estrutura necessária para dar suporte ao empacotamento, montagem, implantação e gerenciamento da execução das instâncias dos componentes.

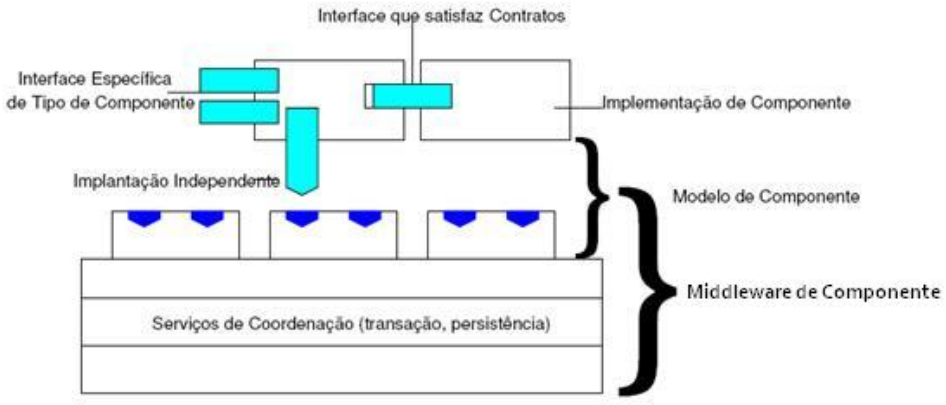

Figura 2.1. Visão geral de um sistema baseado em componentes [1].

A infra-estrutura definida pelo modelo de componentes dá origem ao chamado *middleware* de componentes. O *middleware*, através da implementação de serviços ligados às camadas mais baixas do sistema operacional e *hardware*, permite que os desenvolvedores se preocupem apenas com as funcionalidades específicas da aplicação.

Dentre as funcionalidades comumente disponibilizadas por um *middleware*, pode-se destacar a comunicação em ambiente distribuído, balanceamento de carga, segurança, comunicação baseada em eventos e persistência. Em especial, um *middleware* de componentes deve facilitar a interação entre componentes apenas através de suas interfaces, definir os mecanismos necessários à execução do componente em seu *container* e especi-

ficar a infra-estrutura de montagem, empacotamento e implantação através do ambiente distribuído [34].

Como não há uma definição padrão para um modelo de componentes, diversas empresas, organizações e centros de pesquisas propuseram o seu próprio modelo. Entre os mais difundidos atualmente estão o *Enterprise JavaBeans* (EJB) [6], o *Component Object Model* (COM) [7] e o *CORBA Component Model* (CCM) [8], sendo que este último é o mais genérico, pois não está vinculado a nenhuma plataforma ou linguagem de programação específica.

2.3 CORBA COMPONENT MODEL - CCM

O CCM é a especificação sobre componentes da OMG (*Object Management Group*), cuja primeira versão surgiu em 2002 com o lançamento do padrão CORBA 3.0. As características de um componente CORBA são compatíveis com as definidas por Szypersky [28]. De acordo com o CCM, um componente é um meta-tipo básico, que estende e especializa o meta-tipo objeto do padrão CORBA. A linguagem de especificação dos componentes é a *Interface Definition Language* (IDL).

Os componentes CCM podem interagir com seus clientes através das seguintes portas:

- Facetas (*facets*) - interfaces nomeadas que são fornecidas pelo componente para interação com os demais;
- Receptáculos (*receptacles*) - pontos de conexão identificados que descrevem a habilidade do componente em usar uma referência fornecida por um outro componente;
- Fontes de eventos (*event sources*) - pontos de conexão identificados que emitem eventos de um tipo específico para determinados consumidores de eventos ou para um canal de eventos;
- Consumidores de eventos (*event sinks*) - pontos de conexão identificados aos quais eventos de um tipo específico são enviados;

- Atributos (*attributes*) - valores identificados expostos através de operações de acesso ou alteração.

O CORBA define em seu modelo dois tipos de componentes: básico e estendido. Um componente básico é apenas um meio de transformar um objeto em um componente e não oferece nada além de seus atributos. Por outro lado, os componentes estendidos oferecem um conjunto maior de funcionalidades, podendo oferecer qualquer tipo de porta aos seus clientes. O termo componente será utilizado neste texto para se referir ao componente estendido. Em IDL, os componentes são declarados através da palavra-chave `component` e devem possuir um nome para definir o seu tipo.

Para cada componente, é definido um *component home*, responsável por gerenciar as instâncias deste componente. O *component home* também é um meta-tipo e deve ser declarado para cada declaração de componente. Pode-se declarar mais de um tipo de *home* para gerenciar o mesmo tipo de componente, porém eles não podem gerenciar as mesmas instâncias deste componente.

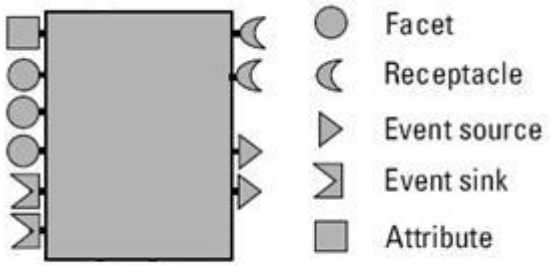

Figura 2.2. Pontos de conexão de um componente de acordo com o CCM [2].

Para o CCM, as portas de um componente possuem um nome e um tipo definido, por exemplo, um componente pode ter duas ou mais facetas de um mesmo tipo, porém com nomes (identificadores) diferentes. Além disso, os componentes por si próprios definem tipos e podem ser herdados por outros componentes, isto é, é possível um componente possuir as mesmas interfaces de outro componente através de uma relação de herança.

Um tipo de componente pode oferecer diversas interfaces aos seus clientes através das

facetas (*facets*). Este tipo de porta é o principal meio de expor as funcionalidades do componente durante a execução. Um componente por ter zero ou mais facetas, as quais são declaradas através da palavra-chave `provides`.

Os receptáculos (*receptacles*) são os pontos de conexão através dos quais um componente pode invocar operações. Um tipo de componente pode ter zero ou mais receptáculos, os quais são declarados pela palavra-chave `uses`. Um receptáculo pode ainda ser do tipo *simplex* ou *multiplex*. O tipo *simplex* permite apenas uma conexão com o componente, enquanto que o tipo *multiplex* pode permitir qualquer número de conexões ou até um certo limite, definido pela implementação do componente. Se o limite de conexões for atingido, uma exceção é levantada durante a execução.

As fontes de eventos (*event sources*) são portas utilizadas para gerar eventos de um determinado tipo e podem ser classificadas em duas categorias: *emitters* e *publishers*. As fontes do tipo *emitter* podem enviar eventos para no máximo um consumidor de eventos. Por outro lado, as fontes do tipo *publisher* podem estar conectadas, através de um canal, a várias consumidores, que se "inscrevem" (*subscribe*) para receber os eventos daquela fonte. Um componente pode possuir zero ou mais *publishers* e *emitters*, os quais são declarados pelas palavras-chave `publishes` e `emits`, respectivamente.

Os consumidores de eventos (*event sinks*) são as portas utilizadas pelos componentes para receber eventos de um determinado tipo. Ao contrário das fontes, os consumidores de eventos não são classificados em categorias e podem receber eventos de qualquer número de fontes. Um tipo de componente pode ter zero ou mais consumidores de eventos, os quais são declarados pela palavra-chave `consumes`.

O serviço de eventos é fornecido aos componentes e seus clientes através da implementação do *container* no qual o componente é executado. Dessa forma, é responsabilidade deste *container* fornecer os mecanismos necessários para o funcionamento deste serviço e determinar a qualidade de serviço e políticas de roteamento para a entrega dos eventos.

Listing 2.1. Exemplo de declaração de um componente em IDL

```
1  interface ISensor{
2    long read_value();
3  };
4  eventtype alarm{};
5  component Controller{
6    uses ISensor sensor;
7    publishes alarm danger;
8    attribute long set_point;
9  };
10 home ControllerHome manages Controller{};
```

O código 2.1 apresenta um exemplo de declaração de um componente em IDL. Nas linhas 1-3, há a declaração de um tipo de interface (`ISensor`) com a assinatura de um método (`read_value`). A declaração de um tipo de evento (`alarm`) é feita na linha 4. Nas linhas 5-9, encontra-se a declaração do componente `Controller`. Este componente possui um receptáculo do tipo `ISensor` (`sensor` - linha 6), uma fonte de eventos *multiplex* do tipo `alarm` (`danger` - linha 7) e um atributo (`set_point` - linha 8). A declaração do *home* responsável por criar as instâncias do componente `Controller` está definida na linha 10.

2.4 MIDDLEWARE DE COMPONENTES PARA TEMPO REAL - CIAO

Prover Qualidade de Serviço (QoS) em sistemas de tempo real baseados em componentes requer que o *middleware* de componentes esteja habilitado para este fim. Em [35], são apontadas diversas razões pelas quais os requisitos de QoS devem ser gerenciados pelo *middleware* ao invés de estar embutido no código de cada componente de forma isolada. Dentre estas razões, podemos destacar: (i) QoS deve ser fornecida a um conjunto de componentes ao mesmo tempo, logo, implementá-la em um único componente prejudicará a reusabilidade do mesmo; (ii) a QoS fornecida por um componente pode não satisfazer às necessidades de outro componente, pois os desenvolvedores não têm conhecimento *a priori* sobre os outros componentes que farão parte do sistema. Dessa forma, para superar as limitações dos *middlewares* de componentes no contexto dos sistemas de tempo real,

é necessário que as políticas de QoS sejam parte integrante do próprio *middleware*.

O CIAO (*Component-Integrated ACE ORB*) é um *middleware* de componentes que implementa uma versão simplificada do CCM, chamada *Lightweight* CCM, com extensões para tempo real. O *Lightweight* CCM não oferece algumas funcionalidades previstas pelo padrão completo de modo a diminuir o seu custo de execução. Desta forma, torna-se possível utilizá-lo em ambientes com recursos limitados, tais como sistemas embarcados. A implementação do CIAO está baseada sobre duas ferramentas importantes para o fornecimento de QoS. A primeira delas é um *framework* para desenvolvimento de sistemas distribuídos, chamado ACE (*ADAPTIVE Communication Environment*). A segunda é um *middleware* para objetos distribuídos, ou ORB (*Object Request Broker*), chamada TAO (*The ACE ORB*), que implementa o padrão CORBA 2.x com extensões para tempo real. ACE, CIAO e TAO foram desenvolvidos pelo DOC Group, um grupo de pesquisa multiinstitucional que reúne pesquisadores das universidades de Washington, California e Vanderbilt, nos Estados Unidos.

Dentre as políticas de QoS fornecidas pelo CIAO, pode-se destacar:

- alocação de recursos de CPU baseada em prioridades dos componentes;
- reserva e alocação de recursos de comunicação entre conexões de componentes.

O CIAO permite que os requisitos de QoS associados aos componentes sejam configurados separadamente em descritores de implantação (arquivos XML), os quais serão gerenciados pela infra-estrutura de execução dos componentes (*container*) [9].

Nas subseções a seguir, serão apresentadas algumas ferramentas relacionadas ao CIAO, as quais em conjunto compõem uma plataforma para projeto, desenvolvimento, implantação e execução de sistemas distribuídos, de tempo real e embarcados (*Distributed, Real-time and Embedded Systems* - DRE).

2.4.1 ADAPTIVE Communication Environment - ACE

O ACE é um *framework* para desenvolvimento de sistemas distribuídos, que oferece soluções para manipular sinais, iniciar serviços, estabelecer comunicação entre processos, gerenciar memória compartilhada e sincronização [3].

Dentre os benefícios obtidos pela utilização do ACE, pode-se destacar:

- portabilidade facilitada para diversos sistemas operacionais, através de mecanismos de adaptação para cada sistema;
- melhoria de qualidade através do uso de padrões de projeto que facilitam a construção de sistemas flexíveis, extensíveis e modulares;
- previsibilidade para aplicações de tempo real, através de suporte a requisitos temporais.

O ACE é dividido nas seguintes camadas:

- Camada de adaptação ao sistema operacional - esta camada está no nível mais baixo do *framework*. Ela é responsável por mapear as chamadas específicas de um determinado sistema operacional para interfaces padronizadas que serão utilizadas pelas camadas superiores. Esta camada permite uma maior portabilidade do ACE.
- *Facades* empacotadores - esta camada constitui uma biblioteca orientada a objetos que encapsula as funções oferecidas pela camada de adaptação do sistema operacional, oferecendo abstrações que facilitam o uso destes recursos.
- *Framework* - camada mais alta da arquitetura, responsável por oferecer mecanismos para o tratamento de eventos gerados por operações de entrada e saída, temporizadores, sinais ou operações de sincronização. Também permite a configuração estática ou dinâmica de aplicações e provê facilidades para a construção de aplicações baseadas em camadas, tais como pilhas de protocolos em nível de aplicação.

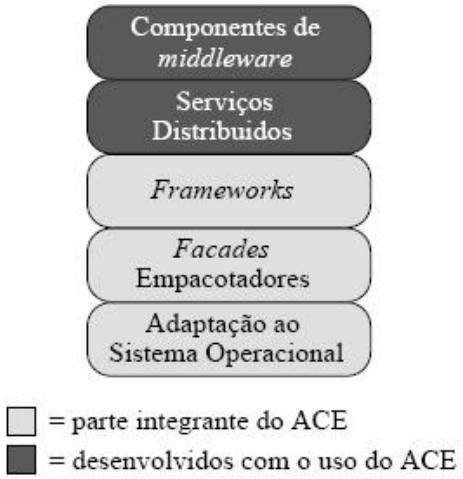

Figura 2.3. Camadas que compõem o ACE [3].

2.4.2 The ACE ORB - TAO

O TAO é um *middleware* para execução de sistemas distribuídos baseados em objetos, também chamado de *Object Request Broker* (ORB), que implementa o padrão CORBA 2.x [36] com extensões para tempo real. De acordo com
[37], o projeto do TAO visa atender aos requisitos necessários para a implementação de ORB's para sistemas de tempo real, dentre os quais, pode-se destacar:

- políticas e mecanismos para especificação de requisitos de QoS da aplicação;
- suporte a garantias de QoS por parte do sistema operacional e do sub-sistema de comunicação;
- protocolos de comunicação eficientes e previsíveis;
- serviço de entrega eficiente e previsível para as requisições dos clientes;
- transformação de dados eficiente e previsível na camada de apresentação;
- gerenciamento de memória eficiente e previsível.

De modo a garantir as restrições temporais definidas para a aplicação, o TAO disponibiliza dois serviços: Serviço de Escalonamento e Serviço de Eventos de Tempo Real.

2.4.2.1 Serviço de Escalonamento O serviço de escalonamento (*scheduling service*) do TAO oferece às aplicações CORBA a flexibilidade para especificar e utilizar políticas de escalonamento diferentes, de acordo com seus requisitos temporais e características disponibilizadas pelo sistema operacional [38].

Os objetivos de projeto do serviço de escalonamento do TAO são: (i) aumentar a utilização do processador, (ii) garantir cumprimento de *deadlines* para operações críticas e (iii) proporcionar maior flexibilidade às aplicações, que se adeque à variabilidade dos requisitos e ambientes de execução.

Para prover uma melhor flexibilidade de escalonamento, o TAO suporta políticas variadas, as quais são: *Rate Monotonic Scheduling* (RMS), *Earliest Deadline First* (EDF), *Maximum Laxity First* (MLF) e *Maximum Urgency First* (MUF) [38]. A política RMS é puramente estática, com atribuição de prioridade fixa. Já a EDF e MLF são políticas com atribuição de prioridade variável. Por fim, a MUF é uma política híbrida, onde algumas tarefas mais críticas podem receber prioridades fixas mais altas e as demais tarefas podem assumir prioridades variáveis.

A arquitetura do comportamento do serviço de escalonamento é dividida em etapas em tempo de projeto e em tempo de execução. A etapa em tempo de projeto envolve a especificação dos requisitos temporais, atribuição de prioridades e a análise do escalonamento. Na outra etapa, em tempo de execução, as informações geradas na etapa anterior são utilizadas para configuração dos módulos do ORB, os quais serão encarregados de liberar as operações e os eventos de acordo com suas prioridades. A especificação dos requisitos temporais é feita através de uma estrutura chamada `RT_Info`, associada a cada operação registrada no serviço de escalonamento.

2.4.2.2 Serviço de Eventos de Tempo Real Um modelo de comunicação baseada em eventos representa uma forma de interação entre entidades de um sistema geralmente realizada de forma assíncrona. Um evento é uma mensagem, ou sinalização, enviada por um objeto produtor a um ou mais objetos consumidores.

O padrão CORBA 2.x possui um serviço de eventos chamado *COS Event Service*, no qual a troca de eventos é mediada por um módulo chamado Canal de Eventos. Este Canal de Eventos é responsável por receber todos os eventos produzidos e redirecioná-los para seus respectivos consumidores-alvo.

Embora seja uma boa alternativa para o desenvolvimento de sistemas distribuídos, o *COS Event Service* não possui mecanismos para garantir a previsibilidade sobre o tempo de entrega dos eventos. Por este motivo, o serviço de eventos foi modificado no TAO para adequá-lo ao uso em sistemas de tempo real [39]. De acordo com o apresentado em [3], as principais modificações foram:

- os requisitos e características temporais dos objetos consumidores e produtores (período, pior caso do tempo de execução, etc) são enviados ao canal de eventos. Este parâmetros são utilizados de forma integrada com o serviço de escalonamento para se determinar as prioridades dos eventos e a ordem de liberação dos mesmos.

- as mensagens são filtradas para que os consumidores recebam apenas os eventos de seu interesse, diminuindo a sobrecarga no sistema. Além disso, os eventos podem ser agrupados e enviados em uma única mensagem para seus respectivos consumidores.

- o serviço de eventos de tempo real pode permitir que os consumidores especifiquem eventos de temporização/alarmes (*timeout*) dos quais eles necessitem. Um evento de *timeout* pode ser configurado para ser disparado a cada intervalo de tempo, ou também pode ser disparado caso uma determinada condição não seja satisfeita até um determinado prazo.

2.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os principais conceitos relacionados ao Desenvolvimento Baseado em Componentes (*Component-Based Developemnt* - CBD) e ao Modelo de Componentes Corba (CCM). Além disto, destacou as características de um *middleware* de

componentes voltado para sistemas de tempo real, CIAO. As principais funcionalidades deste *middleware*, que são utilizadas neste trabalho, são o serviço de escalonamento e o serviço de eventos de tempo real.

O próximo capítulo apresenta em linhas gerais a UML e descreve os diagramas interpretados por TANGRAM para geração de modelos formais. Além disso, o capítulo apresenta as extensões feitas sobre estes diagramas de modo a permitir uma modelagem mais detalhada de sistemas baseados em CCM e CIAO.

CAPÍTULO 3

ESPECIFICAÇÃO E PROJETO DE SISTEMAS DE TEMPO REAL COM UML

O padrão CCM utiliza a *Interface Definition Language* (IDL) para realizar a declaração de componentes. Esta linguagem contém os elementos necessários à declaração de atributos, portas, interfaces, eventos e *component homes*. Todavia, este trabalho considera a UML (*Unified Modeling Language*) [40] como linguagem de entrada na modelagem do sistema. A escolha de UML deve-se aos seguintes motivos:

- a UML é uma linguagem independente de domínio, o que permite a sua aplicação em diversos tipos de sistema, incluindo sistemas de tempo real baseados em componentes;
- facilidade de extensão da linguagem, se necessário, através da inclusão de novos elementos de modelagem, definidos a partir de estereótipos e *tagged values*;
- os diagramas da UML são geralmente utilizados desde as fases iniciais do projeto do sistema, o que contribui para a aplicação da verificação de modelos no processo de desenvolvimento, segundo a proposta deste trabalho;
- a UML é uma linguagem essencialmente gráfica, permitindo a criação de modelos mais próximos do entendimento do desenvolvedor, o que exige um menor grau de especialização na linguagem para uma boa compreensão;
- a IDL não possui mecanismos para modelagem de comportamento dos componentes, ou seja, apenas a parte estrutural é contemplada. A UML, ao contrário, possui diversos diagramas para modelagem comportamental, tais como os de sequência,

máquina de estados e comunicação. A modelagem comportamental é fundamental para a verificação de modelos, pois a partir dela é possível explorar o espaço de estados do sistema.

Este capítulo apresenta os diagramas da UML utilizados neste trabalho e as extensões propostas para contemplar elementos específicos do CCM e do CIAO. Estas extensões são essenciais para que a ferramenta de tradução faça a correta interpretação dos diagramas e, consequentemente, a correta geração dos autômatos temporizados correspondentes.

3.1 INTRODUÇÃO A UML

A UML é uma linguagem destinada a visualizar, especificar, construir e documentar os artefatos de um sistema complexo de *software* [41]. Dentre estes itens, os dois mais importantes para este trabalho são especificar e construir. Diz-se que a UML é uma linguagem de especificação pois ela possibilita a criação de modelos que representem a estrutura e/ou funcionamento de um software. Neste trabalho, a especificação em questão é de sistemas de tempo real baseados em componentes. No quesito seguinte, a UML permite que seus modelos sejam utilizados diretamente para a construção de um sistema, por exemplo, é possível mapear elementos da UML em elementos de uma linguagem de programação. Desta forma, o desenvolvedor pode gerar o código-fonte da sua aplicação (ou uma boa parte dele) diretamente de modelos UML, através de ferramentas apropriadas. No contexto deste trabalho, os modelos UML são utilizados para a construção de modelos formais passíveis de verificação automática de propriedades.

A UML oferece diagramas para se modelar um sistema em duas macro-visões diferentes: uma estrutural e outra comportamental. A modelagem estrutural representa o *software* na sua forma estática, ou seja, descreve a sua arquitetura. Por outro lado, a modelagem comportamental tem por objetivo representar uma visão do *software* em execução, isto é, uma visão dinâmica, com a troca de mensagens, eventos disparados, ações executadas, interações do usuário com o sistema, transição de estados, linha de vida dos

objetos, observação de execuções em relação ao tempo, e outros aspectos importantes de comportamento.

Alguns elementos de construção da UML não possuem restrições sintáticas, de modo que o desenvolvedor pode informar qualquer texto livre e o mesmo será aceito pela linguagem. Entretanto, para que a ferramenta desenvolvida neste trabalho possa interpretar corretamente todos os elementos que serão absorvidos pela tradução, estes devem ser especificados de acordo com algumas restrições sintáticas definidas neste trabalho. Por exemplo, a UML não define como é a sintaxe para o comando de atribuição de valor a uma variável. Neste caso, a sintaxe adotada é a mesma utilizada na linguagem C, através do operador "`=`". Esta e outras regras serão apresentadas ao longo das próximas seções, conforme os diagramas forem sendo apresentados. As restrições de sintaxe, no entanto, não reduzem a expressividade da modelagem em relação aos parâmetros que são úteis na verificação das propriedades. Desta forma, também é possível garantir que os autômatos gerados serão consistentes em relação aos diagramas UML de origem.

As seções a seguir apresentam os dois diagramas da UML utilizados neste trabalho para geração dos autômatos temporizados: o diagrama de componentes e o diagrama de máquina de estados. O primeiro diagrama fornece a estrutura do sistema, enquanto que o segundo especifica o comportamento. No contexto de sistemas baseados em componentes, os dois diagramas citados oferecem elementos suficientes para aplicação da verificação de modelos. Além disto, são apresentadas as extensões feitas a estes diagramas de modo a contemplar os elementos específicos do CCM e do CIAO.

3.2 DIAGRAMA DE COMPONENTES

O Diagrama de Componentes é um diagrama estrutural utilizado para especificar o sistema em termos de componentes, portas genéricas e interfaces. Os principais elementos deste diagrama são componentes, portas, interfaces e conectores. A Figura 3.1 apresenta um diagrama de componentes com seus principais elementos de construção definidos pela UML. Os componentes são representados por retângulos com o estereótipo `component` e

um ícone específico no canto direito superior. Estereótipos são mecanismos de extensão da UML e serão explicados mais adiante. As portas são representadas por pequenos retângulos anexados à borda dos componentes. As interfaces são geralmente representadas pelo ícone *ball-and-socket*, fazendo a ligação entre dois componentes. Quando uma porta fornece uma interface, um conector de realização (*realization*) é utilizado para ligar a porta à interface correspondente. Neste caso, o ícone formado sobre a interface será o círculo (*ball*). Por outro lado, quando a porta requer uma interface, um conector de dependência (*dependency*) é utilizado. Neste caso, o ícone formado sobre a interface será o encaixe (*socket*).

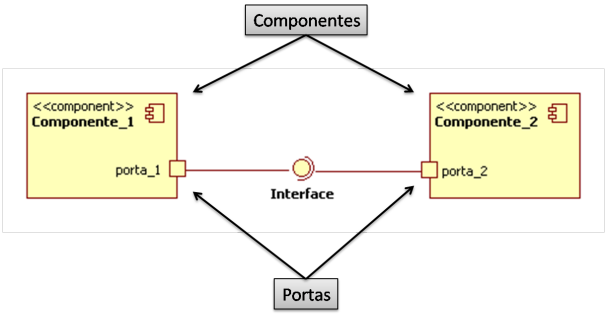

Figura 3.1. Diagrama de componentes com seus principais elementos.

Além destes elementos visíveis do diagrama, a UML também permite a definição de atributos para os componentes. No entanto, estes atributos são geralmente manipulados apenas através da ferramenta de modelagem, logo, não ficam explícitos no modelo (ver Figura 3.2).

Apesar de permitir a modelagem de um sistema baseado em componentes, este diagrama não está relacionado a nenhum modelo de componentes específico, logo, algumas características específicas do CCM não são contempladas. Uma alternativa para modelagem de componentes baseados no CCM é a utilização de uma especificação da OMG chamada *UML Profile for CCM* [42], que contém estereótipos pré-definidos para mapear a IDL em elementos da UML. Contudo, o *UML Profile for CCM* prevê apenas a modelagem de componentes isolados e não oferece recursos para elaboração da composição entre componentes. A modelagem desta composição é de fundamental importância para

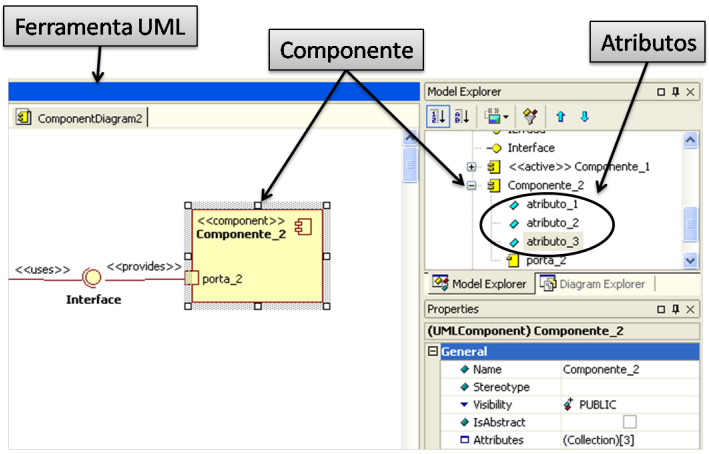

Figura 3.2. Visualização de atributos através da ferramenta de UML.

este trabalho, pois através dela é possível determinar como se dá a troca de eventos e mensagens entre os componentes. Devido a estas características decidiu-se propor extensões para o diagrama de componentes de modo a permitir a modelagem de elementos específicos do CCM e também do CIAO.

3.2.1 Mecanismos de extensão da UML

A UML oferece dois mecanismos básicos para extensão de diagramas: estereótipos e *tagged values*[1].

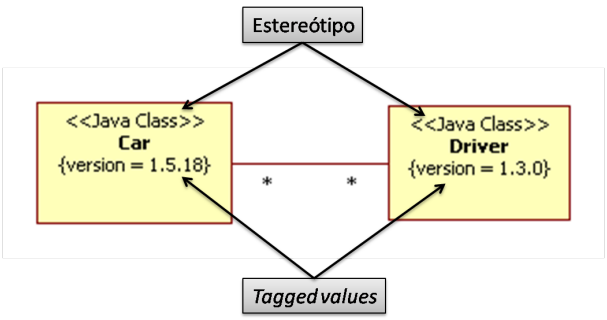

Figura 3.3. Estereótipos e *tagged values* - mecanismos de extensão da UML.

Estereótipos são rótulos geralmente utilizados para associar elementos da UML com

[1]O termo *tagged values* não possui uma tradução padronizada para o português. Em [41], foi traduzido como "valores atribuídos".

elementos de outras linguagens ou outros elementos externos. Os estereótipos sempre são apresentados entre "`<< >>`". Por exemplo, para representar classes da linguagem Java em um diagrama de classes, o desenvolvedor poderia utilizar o estereótipo "`<<Java Class>>`" nas classes deste diagrama (ver Figura 3.3). *Tagged values* são propriedades adicionais que o desenvolvedor pode associar a um determinado elemento da UML. Por exemplo, o desenvolvedor poderia adicionar um *tagged value* chamado `version` às classes de um diagrama de classes para representar a versão em que se encontra cada uma delas no repositório de arquivos do *software*(ver Figura 3.3).

3.2.2 Extensões relativas ao CCM

Ambos os mecanismos, estereótipos e *tagged values*, foram utilizados neste trabalho de modo a permitir a especificação de características encontradas no CCM e também de algumas funcionalidades oferecidas pelo *middleware* CIAO, as quais serão interpretadas pela ferramenta de tradução. Um exemplo de diagrama de componentes estendido pode ser visto na Figura 3.4.

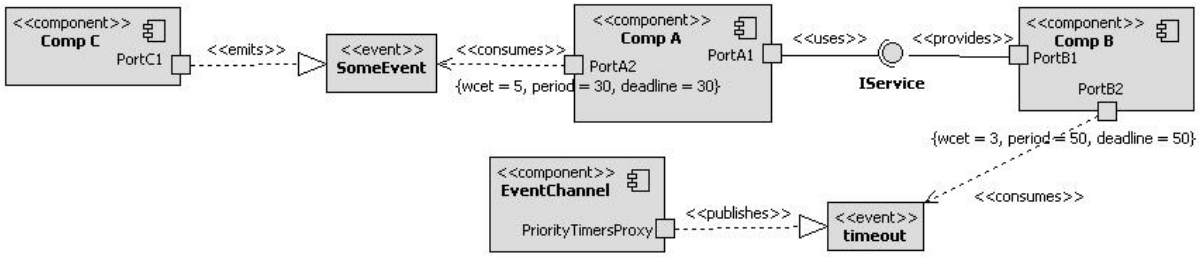

Figura 3.4. Exemplo de diagrama de componentes estendido.

Para cada tipo de porta CCM (apresentadas na Seção 2.3) foi criado um estereótipo correspondente em UML, com os mesmos nomes utilizados em IDL para declaração de portas. Desta forma, os estereótipos foram organizados da seguinte maneira:

De modo a melhorar a organização visual do diagrama, os estereótipos citados na Tabela 3.1 foram aplicados aos conectores que ligam as portas a suas interfaces correspondentes. Por exemplo, na Figura 3.4, o estereótipo `provides` foi aplicado ao conector

Estereótipo	Porta correspondente	Exemplo (Figura 3.4)
`provides`	*facet*	`PortB1`
`uses`	*receptacle*	`PortA1`
`consumes`	*event sink*	`PortA2`
`emits`	*event source (simplex)*	`PortC1`
`publishes`	*event source (multiplex)*	`PriorityTimersProxy`

Tabela 3.1. Esterótipos correspondentes às portas do CCM.

que liga a porta `PortB1` à interface `IService`, indicando que esta porta é um *facet*.

O mecanismo de comunicação através de eventos, definido pelo CCM, também não possui elementos de modelagem correspondentes no diagrama de componentes da UML. Para isto, foi feita uma outra extensão deste diagrama, na qual os eventos (*event types*) são representados por classes com o estereótipo `event` (ver `SomeEvent`, Figura 3.4). De forma análoga à interface, um conector de realização (*realization*) deve ser utilizado para ligar a porta ao seu evento fornecido. Para modelar um evento requerido, o conector de dependência (*dependency*) deve ser utilizado. Neste caso não haverá a formação de um ícone específico para representar um evento, a exemplo do que ocorre para uma interface.

3.2.3 Extensões relativas ao CIAO

De acordo com o serviço de eventos de tempo real do CIAO, *event sinks* podem possuir atributos temporais que serão tratados pelo serviço de escalonamento. Três destes atributos temporais foram incorporados ao modelo UML através dos seguintes *tagged values*: (i) `wcet`; que representa o custo de execução no pior caso; (ii) `period`, que representa o período de ativação do *event sink*; e (iii) `deadline`, que representa o prazo para conclusão da execução após a sua ativação. Um exemplo de atribuição destes *tagged values* é apresentado na Figura 3.4 (`PortA2` e `PortB2`).

A especificação destes atributos temporais é importante tanto para fins de documentação do sistema, quanto para fins de verificação. Isto decorre do fato de que a ferramenta de tradução precisa destes atributos para configurar os aspectos de escalonamento de componentes baseado em prioridades sobre os autômatos gerados.

Outra importante funcionalidade oferecida pelo serviço de eventos de tempo real do CIAO são os eventos de *timeout* periódicos. Quando um componente necessitar de um *timeout*, ele pode se registrar neste serviço fornecido pelo *middleware*. Para representar esta funcionalidade no diagrama de componentes, dois elementos foram introduzidos: (i) um componente chamado `EventChannel`, para simbolizar o canal de eventos do CIAO, e (ii) um evento específico chamado `timeout`, que é produzido pelo `EventChannel`. A Figura 3.4 apresenta um exemplo de utilização do *timeout*. A taxa com a qual os eventos serão produzidos é determinada pelo atributo `period` da porta que está recebendo o *timeout*. Por exemplo, a porta `PortB2` na Figura 3.4 possui um período de 50 unidades de tempo para o recebimento dos eventos de *timeout*.

A última funcionalidade obtida do CIAO é a definição de componentes ativos [43]. Um componente ativo possui a sua própria *thread* de execução, definida através de uma função de *callback* chamada `start`. No momento em que o sistema for inicializado, o *middleware* executará a função `start` de cada componente que for declarado como ativo. Para definir componentes ativos foi criado o estereótipo `active`, que deve ser aplicado aos componentes desse tipo (ver `Componente_1` na Figura 3.5).

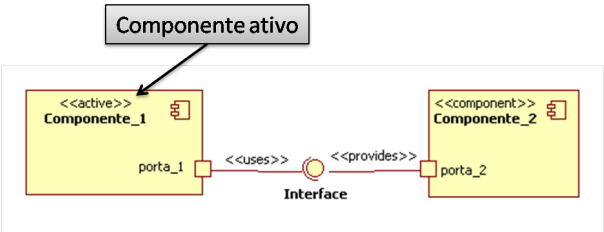

Figura 3.5. Exemplo de componente ativo.

3.2.4 Regras sintáticas

As regras sintáticas do diagrama de componentes estão principalmente relacionadas com a nomenclatura dos elementos inseridos no diagrama. O padrão utilizado neste trabalho segue a regra comumente utilizada para identificadores nas linguagens de programação, por exemplo, na linguagem C. Desta forma, os nomes utilizados em compo-

nentes, atributos, portas, eventos, interfaces e operações podem ser formados por qualquer combinação de letras, números e o caractere "_", começando por uma letra ou "_".

Além disto, o desenvolvedor deve tomar cuidado com as palavras que foram reservadas para representação dos elementos do CCM e do CIAO. Por exemplo, uma classe com estereótipo `event` será interpretada como um evento do CCM, mesmo que o usuário a tenha utilizado para outro propósito. As palavras reservadas e seus elementos vinculados estão listados na Tabela 3.2.

Palavra reservada	**Tipo**	**Elemento vinculado**
`provides`	Estereótipo	Associação do tipo realização
`uses`	Estereótipo	Associação do tipo dependência
`consumes`	Estereótipo	Associação do tipo dependência
`emits`	Estereótipo	Associação do tipo realização
`publishes`	Estereótipo	Associação do tipo realização
`event`	Estereótipo	Classe
`active`	Estereótipo	Componente
`timeout`	Nome	Classe
`EventChannel`	Nome	Componente
`PriorityTimersProxy`	Nome	Porta
`wcet`	*Tagged value*	Porta
`period`	*Tagged value*	Porta
`deadline`	*Tagged value*	Porta

Tabela 3.2. Palavras reservadas para o diagrama de componentes.

Os atributos dos componentes podem ser de dois tipos: `Integer` ou `Boolean`. O primeiro representa valores inteiros e o segundo valores booleanos. Os valores booleanos podem ser `true` ou `false`. Outros tipos de dados poderão ser definidos pelo usuário, porém não serão capturados pela versão atual da ferramenta de tradução.

Neste momento, a tradução de interfaces ainda não está preparada para capturar os parâmetros e o tipo de retorno das operações. Pretende-se incorporar esta funcionalidade em versões futuras da ferramenta de tradução.

Outra regra sintática importante é a utilização de uma porta para especificar a ligação de um componente para uma interface ou evento. Embora a UML permita fazer isto sem o uso de portas, esta regra é importante para manter a compatibilidade com o CCM

e garantir a consistência dos modelos gerados. Um exemplo comparativo entre uma especificação errada e uma correta é apresentado na Figura 3.6.

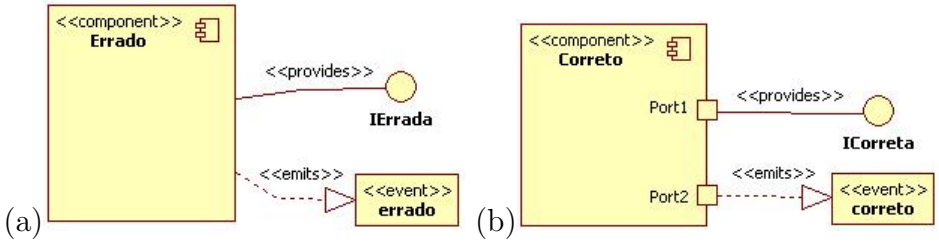

Figura 3.6. Regra de especificação de portas: (a) incorreto; (b) correto.

3.3 DIAGRAMA DE MÁQUINA DE ESTADOS

O diagrama de máquina de estados é utilizado para representar o comportamento do sistema sob a visão de estados e transições. No contexto de um sistema baseado em componentes, este diagrama pode ser usado para modelar todo o conjunto de estados pelos quais os componentes passarão durante o seu ciclo de execução. Desta forma, a tradução deste diagrama para autômatos temporizados permitirá que a ferramenta de verificação de modelos possa explorar todo o espaço de estados do sistema para validar as propriedades da aplicação.

Os principais elementos deste diagrama são estados, pseudo-estados e transições. A Figura 3.7 apresenta um diagrama de máquina de estados com seus principais elementos. Este diagrama pode também ser visto como um grafo, onde os estados são os vértices e as transições são as arestas. Os estados são utilizados para representar as etapas ou configurações estáveis dentro do contexto de execução do diagrama. Podem também ser utilizados para abstrair trechos de execução que não necessitem de um detalhamento mais aprofundado. O pseudo-estado inicial, que representa o ponto de início da execução, deve possuir apenas uma transição de saída, a qual não pode ter evento de ativação, restrições de guarda ou efeitos. O estado final, que representa o término do fluxo de execução, não pode possuir transições de saída.

Uma transição pode conter um evento de ativação (*trigger*), uma restrição de guarda

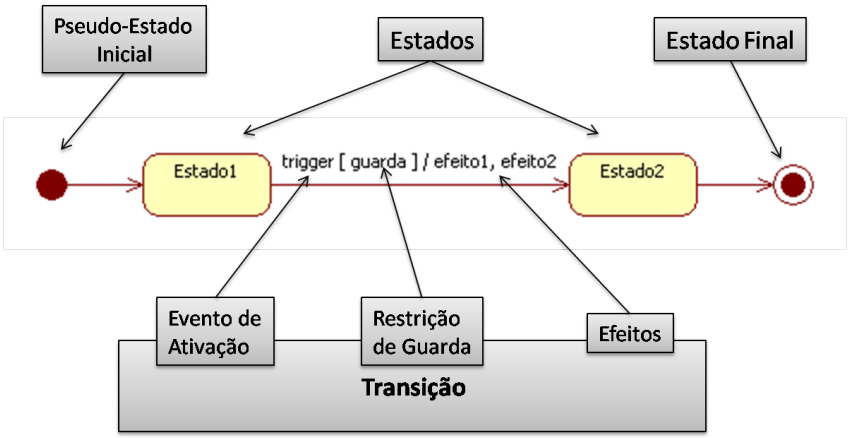

Figura 3.7. Principais elementos que compõem um diagrama de máquina de estados.

e um ou mais efeitos. A restrição de guarda é exibida entre colchetes ("[ ]") e os efeitos são precedidos por uma barra ("/"), conforme apresentado na Figura 3.7.

Um *trigger* especifica o evento que ativará a transição. Uma transição ativada somente será executada caso a sua restrição de guarda seja válida. Quando o evento de ativação não estiver definido, a transição será ativada pelo término da execução do seu estado de origem. A UML permite a especificação dos seguintes tipos de eventos de ativação:

- **sinais**, que representam o recebimento de mensagens assíncronas;
- **chamadas**, que representam solicitações de execução de operações;
- **temporais (*time triggers*)**, que representam a contagem de uma certa quantidade de tempo;
- **alteração**, que representa a mudança no valor de uma condição de falso para verdadeiro.

Este trabalho considera apenas eventos de ativação do tipo *time trigger*, conforme será explicado na seção 3.3.1.

As restrições de guarda são expressões booleanas que devem ser satisfeitas para que a transição seja executada. Os efeitos são ações executadas no momento em que a transição

ocorre. Em geral, estes efeitos são utilizados para especificar atualizações de variáveis, envio de sinais e chamada de operações. Caso uma transição possua mais de um efeito, estes serão executados sequencialmente, na ordem em que aparecem no diagrama. Uma transição somente será concluída após a execução de todos os seus efeitos.

Os elementos apresentados nesta seção representam apenas um subconjunto dos elementos aceitos por um diagrama de máquina de estados. Por exemplo, a UML permite ainda a modelagem de subestados dentro de um estado [41]. Entretanto, este subconjunto foi escolhido por ser suficientemente expressivo para a modelagem do comportamento de sistemas de tempo real baseados em componentes, que é o foco da abordagem proposta neste trabalho. Entretanto, a incorporação de outros elementos deste diagrama à ferramenta de tradução não foi descartada e deve ser objeto de trabalhos futuros.

3.3.1 Modelagem comportamental de sistemas baseados em CCM e CIAO

No contexto de sistemas baseados em componentes, construídos sobre a plataforma CCM/CIAO, os modelos comportamentais de UML devem estar associados aos elementos definidos por estas tecnologias que possuem algum fluxo de execução.

De acordo com o *CCM Implementation Framework* [8], as operações definidas pelos *facets* e os *event sinks* possuem o seu próprio fluxo de execução, ao contrário dos *receptacles* e dos *event sources*, que são apenas meios para acessar outros componentes. Similarmente, conforme mencionado na seção 3.2.3, um componente ativo também possui a sua própria *thread* de execução, que deve ser implementada pela função `start`.

Consequentemente, a abordagem proposta neste trabalho considera que diagramas de máquina de estados devem ser modelados para estes três pontos de execução citados (*facets*, *event sinks* e função `start`). Vale ressaltar que um *facet* implementa uma interface, a qual pode possuir diversas operações. Desta forma, cada operação deve ter a sua própria máquina de estados. O comportamento total do sistema será dado pela união das máquinas de estados definidas para todos os pontos de execução de cada componente.

A Figura 3.8 apresenta um exemplo de diagrama de máquina de estados e aponta o seu *event sink* correspondente. A execução desta máquina de estados será disparada quando o componente receber um evento do tipo `SomeEvent` pela porta `PortA2`, conforme destacado na figura. O diagrama contém 3 estados organizados em um único fluxo de execução, uma restrição de guarda, um *time trigger* e um efeito.

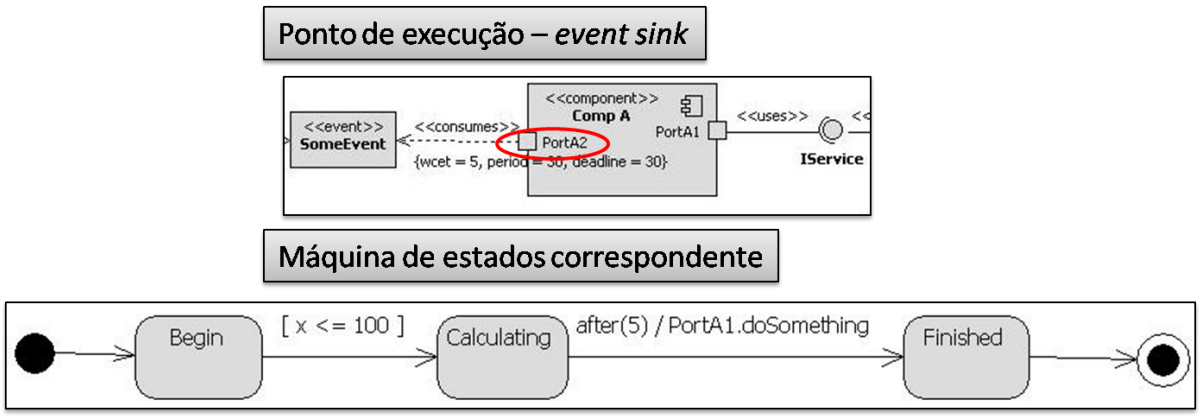

Figura 3.8. Exemplo de diagrama de máquina de estados da UML 2.0

Ao modelar o comportamento de uma operação no contexto de componentes, as restrições de guarda podem ser utilizadas para especificar condições sobre os atributos ou variáveis manipuladas pelos componentes. No exemplo da Figura 3.8, a restrição `[ x <= 100 ]` especifica uma condição sobre o atributo `x`, do tipo inteiro, que está definido no componente `CompA`. Ainda nesta figura, `after(5)` representa a duração da execução do estado `Calculating` e `PortA1.doSomething` representa uma chamada de operação. Estes elementos serão explicados mais adiante.

De forma análoga, os efeitos também podem ser utilizados para manipular atributos, através de operações de atribuição de valor. Por exemplo, se o desenvolvedor precisasse atribuir o valor 0 (zero) ao atributo `x` mencionado anteriormente, ele poderia incluir o efeito `x = 0` na transição de sua escolha. Vale lembrar que a transição que parte do pseudo-estado inicial não pode conter efeitos, restrições de guarda ou *triggers*.

Além de manipular atributos, os efeitos também podem ser usados para especificar envio de eventos ou chamada de operações através das portas dos componentes. Para es-

pecificar uma chamada de operação, é necessário informar o *receptacle* que será utilizado e qual operação será chamada. O nome da operação é definido pela interface requerida pelo *receptacle*. Na Figura 3.8, um efeito de chamada de operação está especificado na transição entre `Calculating` e `Finished`. O efeito especificado é `PortA1.doSomething`, onde `PortA1` é o *receptacle* do componente e `doSomething` é uma operação definida na interface `IService`. Ao executar este efeito, o comportamento (máquina de estados) correspondente à operação `doSomething` será disparado no componente de destino. Como uma chamada de operação é uma comunicação síncrona, a transição será concluída somente quando a operação chamada terminar a sua execução.

A especificação do envio de um evento é similar à da chamada de operação, a diferença é que basta informar o *event source* que será utilizado. Isto é suficiente pois cada *event source* envia apenas um tipo de evento. Por exemplo, se o componente da Figura 3.8 possuísse um *event source* chamado `PortA3`, bastaria incluir este nome na lista de efeitos da transição desejada para representar o envio do evento para o(s) componente(s) de destino. Considerando o *middleware* CIAO, todo evento é interceptado pelo Canal de Eventos, o qual será encarregado por despachar cada evento para seu respectivo consumidor, seguindo um escalonamento de prioridades. A comunicação através de eventos é assíncrona, permitindo que a transição seja concluída logo após o envio do evento, sem ter de aguardar pelo término da execução do(s) componente(s) de destino.

Considerando a semântica dos efeitos de envio de eventos e chamada de operações explicada nos parágrafos anteriores, é possível perceber, no contexto deste trabalho, que o conjunto de *triggers* que podem ocorrer não contempla o recebimento de eventos ou recebimento de chamada de operações. Isto decorre do fato de que o recebimento de um evento ou de uma chamada de operação desencadeará a execução de um comportamento completo, representado por uma máquina de estados, ao invés de simplesmente disparar um transição de estado. Desta maneira, conclui-se que apenas os *time triggers* e os eventos de alteração devem ser considerados.

Os eventos de alteração são representados através de uma função definida pela UML

chamada `when`. Esta função recebe como parâmetro uma condição que será avaliada continuamente. Quando esta condição assumir um valor verdadeiro, o evento será disparado e a transição ativada. Por exemplo, considere a transição apresentada na Figura 3.9. Neste caso, a transição será ativada quando o valor de `x` se tornar maior do que `10`.

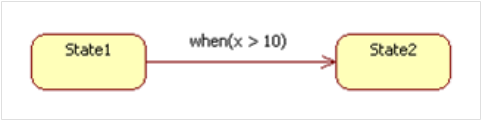

Figura 3.9. Exemplo de um evento de alteração.

Os *time triggers* podem ser representados por duas funções definidas pela UML: `at` e `after`. A função `at` recebe como parâmetro um valor de tempo relativo ao tempo global do sistema, desde o início da sua execução. Ela também pode receber um valor temporal completo, com data, hora, minutos e segundos ("18/07/2009 18:07:00"). Um exemplo de utilização desta função é apresentado na Figura 3.10, no qual a transição é disparada quando o tempo global do sistema atingir a marca de `1000` unidades de tempo.

Figura 3.10. Exemplo de um *time trigger* disparado pela função `at`.

A função `after` recebe como parâmetro um valor de tempo relativo ao início da execução do estado de origem da transição, ou seja, esta função mede o tempo relativo a um único estado. Um exemplo de aplicação desta função é apresentado na Figura 3.8, no qual a transição de saída do estado `Calculating` será ativada 5 unidades de tempo após a entrada neste estado.

No contexto de sistemas de tempo real, os *time triggers* representam peças importantes em termos de modelagem de requisitos temporais, pois eles podem ser utilizados para especificar a duração ou o custo de execução de cada estado do diagrama.

Para fins de avaliação da proposta deste trabalho, dentre os eventos de ativação apresentados, apenas o *time trigger* `after` está sendo interpretado pela ferramenta de

tradução. Pretende-se incluir as demais funções em versões futuras desta ferramenta. No caso da função `when` é possível representar o seu comportamento utilizando restrições de guarda e uma transição em *loop* com a função `after`. Uma representação do exemplo da Figura 3.9 é exibido na Figura 3.11.

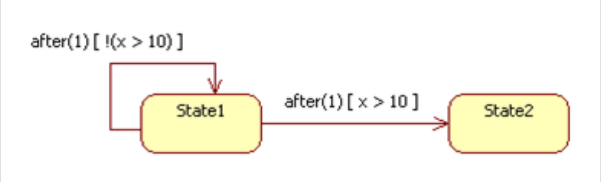

Figura 3.11. Representação da função `when` utilizando a função `after`.

3.3.2 Regras sintáticas

A primeira regra sintática que deve ser destacada é o padrão para identificação dos diagramas de máquina de estados. Devido ao fato de que a UML não prevê um vínculo formal do diagrama de máquina de estados para todos os pontos de execução definidos na seção anterior, a solução adotada foi estabelecer este vínculo através do nome do diagrama. O padrão adotado é apresentado na Tabela 3.3.

Ponto de execução	**Padrão**
Operação de um *facet*	<Componente>_<*Facet*>_<Operação>
Event sink	<Componente>_<*Event sink*>
Função `start`	<Componente>_`start`

Tabela 3.3. Padrão de identificação dos diagramas de máquina de estados.

Para exemplificar este padrão, o diagrama de máquina de estados da Figura 3.8 deve possuir o nome "CompA_PortA2", pois ele está associado a um *event sink*.

De forma análoga ao que acontece com o diagrama de componentes, os identificadores utilizados para os estados devem obedecer às regras comumente encontradas em linguagens de programação para declaração de variáveis. Os nomes podem ser formados por qualquer combinação de letras, números e o caractere "_", começando por uma letra ou por "_". Isto é feito para manter a compatibilidade com os autômatos que serão gerados

pela tradução.

A inclusão do pseudo-estado inicial é obrigatória, porém a do estado final é opcional. Não é permitido estabelecer uma transição do pseudo-estado inicial para o estado final diretamente. É necessário existir pelo menos 1 (um) estado simples. Como já foi citado anteriormente, o pseudo-estado inicial pode ter apenas 1 (uma) transição de saída e não pode ter transições de chegada. O estado final pode ter qualquer número de transições de chegada, porém não pode ter transições de saída.

Os eventos de ativação devem ser do tipo *time trigger*, utilizando a função `after`. Esta função deve obrigatoriamente receber uma constante inteira não-negativa como parâmetro, por exemplo, `after(10)`. Não é permitido a passagem de variáveis ou atributos para esta função na versão atual da ferramenta proposta por este trabalho, porém pretende-se incluir esta funcionalidade em versões futuras.

As restrições de guarda são expressões booleanas envolvendo atributos dos componentes. Expressões booleanas são geralmente na forma `variável` $*$ `valor`, onde $*$ pode ser substituído por $\{=, >, >=, <, <=, != \}$. Caso a variável seja booleana, é permitido também `!variável`, para indicar a negação da variável. De fato, toda expressão booleana que for válida em UPPAAL, também será válida aqui, pois ela será traduzida com a mesma sintaxe. Caso algum nome não-declarado como atributo do componente seja usado, um erro será reportado no momento da tradução.

Os efeitos podem ser de três tipos: atribuição de valor, envio de evento e chamada de operação. A atribuição de valor deve ser no formato `variável` = `expressão`, onde a expressão deverá retornar um valor compatível com o tipo da variável que está sendo atribuída. O envio de evento deve conter apenas um nome, que deve ser o identificador do *event source* através do qual o evento será enviado. A chamada de operação deve ser no formato `<Nome do receptacle>.<Nome da operação>`. A versão atual da ferramenta de tradução não suporta chamada de operações com parâmetros, logo, não devem ser colocados parênteses após o nome da operação.

Em geral, as ferramentas UML colocam automaticamente os "[ ]", "/" e as vírgulas

que compõem o rótulo de uma transição. Entretanto, caso a ferramenta já não faça isto, o desenvolvedor deve estar atento para inserir estes caracteres.

3.4 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a abordagem proposta neste trabalho para a especificação de sistemas de tempo real baseados em componentes através de diagramas UML. Esta proposta está baseada na construção de diagramas de componentes e de máquina de estados estendidos, de modo a contemplar as características de CCM e CIAO.

A modelagem de sistemas segundo a proposta deste capítulo é a principal forma de entrada para TANGRAM. A partir destes modelos, o desenvolvedor poderá utilizar TANGRAM para obter os autômatos temporizados passíveis de verificação automática pela ferramenta UPPAAL.

O capítulo a seguir trata da verificação de modelos para sistemas de tempo real, abordando os principais conceitos relacionados ao verificador de modelos UPPAAL e seus autômatos temporizados.

CAPÍTULO 4

VERIFICAÇÃO DE MODELOS PARA SISTEMAS DE TEMPO REAL

A Verificação de Modelos (*Model-Checking*) é uma técnica de validação de sistemas concorrentes de estados finitos, na qual se verifica se um determinado modelo de sistema, representado como um sistema de transição de estados finitos, satisfaz a uma propriedade especificada por uma fórmula da lógica temporal.

Diversas abordagens existem para aplicar a verificação de modelos em sistemas de tempo real. Uma das principais abordagens é apresentada em [44], na qual os autores estendem tanto o sistema de transições de estados quanto as fórmulas da lógica temporal com informações quantitativas de tempo. Nesta abordagem, o tempo é tratado como uma variável contínua e para cada transição do grafo pode-se associar um valor pertencente ao conjunto dos números reais não-negativos. A lógica proposta é uma extensão da CTL [10], chamada *timed* CTL (ou TCTL). As fórmulas da TCTL são derivadas da CTL, porém, adicionando-se um quantificador de tempo sobre seus operadores.

Em geral, a técnica de verificação de modelos é aplicada de forma automática por ferramentas chamadas de verificadores de modelos (*model-checkers*), ou seja, a partir do modelo do sistema e da propriedade especificada pelo desenvolvedor, o verificador de modelos é capaz de identificar automaticamente se aquela propriedade é satisfeita pelo modelo.

Um dos verificadores de modelos mais difundidos para sistemas de tempo real é o UPPAAL [14], o qual é utilizado neste trabalho para verificação dos sistemas de tempo real baseados em componentes. UPPAAL toma como referência os modelos de transição

de estados finitos proposto por [44], também chamados de autômatos temporizados (*timed automata*).

As seções a seguir apresentam o verificador de modelos UPPAAL, seus autômatos temporizados e como as propriedades dos sistemas são verificadas por esta ferramenta.

4.1 UPPAAL

O UPPAAL [14] é um verificador de modelos voltado para sistemas de tempo real. Ele foi desenvolvido para verificar sistemas modelados como uma rede de autômatos finitos com tempo (*timed automata*) [45], na qual cada autômato é executado em paralelo em relação aos outros.

O UPPAAL possui um ambiente para construção dos autômatos, outro para simular a execução do sistema e um ambiente para verificação de propriedades. No ambiente de construção, é possível modelar os autômatos de forma gráfica, editar todas as suas características, fazer a declaração de variáveis globais e locais e instanciar os autômatos que farão parte do sistema. No ambiente de simulação, o usuário pode realizar uma observação passo-a-passo do comportamento do sistema, visualizando a interação entre os autômatos, além de poder escolher as transições que serão executadas, caso mais de uma esteja habilitada em um determinado momento. O ambiente de verificação permite que o usuário entre com as propriedades que deseja verificar e, após realizar a verificação, caso a sua fórmula não seja satisfeita, ele também poderá visualizar a sequência de transições do sistema que configuram um contra-exemplo para a sua fórmula.

4.1.1 Autômatos Temporizados

Um autômato temporizado é um autômato finito adicionado de variáveis de relógio (Figura 4.1). O modelo de tempo é contínuo, isto é, os relógios podem ser avaliados em números reais não-negativos. Apesar de cada autômato possuir seus próprios relógios, to-

dos os relógios do sistema avançam de forma sincronizada. A seguir será dada a definição formal básica de um autômato temporizado. Para isto, deve-se considerar as seguintes notações: (i) C, um conjunto de relógios; e (ii) $B(C)$, um conjunto de conjunções sobre condições simples da forma $x * c$ ou $x - y * c$, onde $x, y \in C$, $c \in \mathbb{N}$ e $* \in \{<, \leq, =, >, \geq\}$.

Definição: autômato temporizado. *Um autômato temporizado é uma tupla (L, l_0,C,A,E,I), onde L é o conjunto de localizações, $l_0 \in L$ é a localização inicial, C é o conjunto de relógios, A é o conjunto de ações, $E \subseteq L \times A \times B(C) \times 2^C \times L$ é um conjunto de arestas entre localizações com uma ação, um guarda e um conjunto de atualizações de relógio, e $I : L \rightarrow B(C)$ atribui invariantes às localizações.*

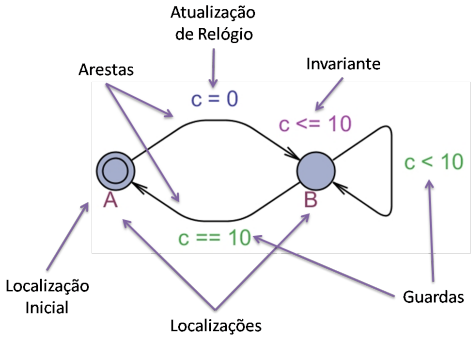

Figura 4.1. Exemplo de um autômato temporizado.

Ao construir um autômato em UPPAAL, o usuário define um modelo de autômato (*template*) que deverá ser instanciado posteriormente na definição do sistema a ser verificado/simulado. Modelos de autômatos podem ser definidos com parâmetros de qualquer tipo de forma que o desenvolvedor possa instanciar mais de um autômato do mesmo modelo, substituindo os parâmetros pelos argumentos desejados para cada um destes autômatos.

Um estado em UPPAAL é definido como os valores da variáveis, dos relógios e as localizações nas quais os autômatos se encontram num dado instante. A aresta representa uma relação entre duas localizações. Graficamente, uma aresta é uma seta que parte de uma localização para a outra. O termo transição também é usado no escopo do sistema como um todo e representa a passagem de um estado para outro.

As localizações de um autômato podem ser associadas a invariantes. Um invariante é uma expressão de condição que deve ser válida enquanto o autômato estiver naquela localização. Esta expressão pode envolver relógios ou variáveis inteiras. Além disso, uma localização pode ser especificada como `urgent` ou `commited`. A primeira não permite que o tempo avance enquanto o autômato estiver na referida localização, enquanto que o segundo é ainda mais restritivo: além do tempo não avançar, ele tem prioridade de execução sobre as outras localizações que não são desse tipo.

As variáveis podem ser dos seguintes tipos: inteiro, booleano, canal e relógio. Constantes inteiras também podem ser definidas. Além disso, é possível construir *arrays* para relógios, canais e variáveis ou constantes inteiras. Um canal é uma variável que representa uma comunicação síncrona entre dois autômatos, isto é, no momento em que dois autômatos se comunicam através de um canal, suas transições são executadas simultaneamente. Os relógios são variáveis que medem a passagem do tempo de forma contínua, podendo ser reiniciados pelo autômato em qualquer transição. Como já foi mencionado anteriormente, todos os relógios do sistema avançam igualmente no tempo, ou seja, a frequência de atualização é a mesma para todos os relógios.

As arestas, por sua vez, podem conter restrições de guarda, sincronização de canais, inicialização de relógios e atualização de variáveis. Restrições de guarda são expressões de condição que devem ser válidas para que a aresta seja ativada. A sincronização pode ser do tipo `s!` ou `s?`, o primeiro uma sincronização de envio e o segundo de recebimento sobre o canal `s`. A atribuição de valor inicial aos relógios deve ser feita apenas com valores inteiros. Cada autômato pode disparar uma aresta separadamente ou em sincronia, através de um canal, com outro autômato para mudar de localização. Um canal de sincronização pode ser especificado como `broadcast`, o que significa que quando um autômato disparar uma sincronização sobre este canal, um ou mais outros autômatos poderão sincronizar com ele sobre este canal, caso estejam aptos a receber esta sincronização no momento em que ela foi disparada.

Um recurso disponibilizado recentemente pelas últimas versões do UPPAAL foi a

declaração de funções, as quais podem ser chamadas nas arestas dos autômatos, juntamente com as atualizações de variáveis. A sintaxe das funções é bastante semelhante à da linguagem de programação C, porém mais restrita. As funções podem possuir parâmetros ou retornar valores de qualquer tipo.

4.1.2 Verificação de Propriedades

Para verificarmos se uma determinada propriedade é satisfeita pelo modelo, é necessário escrevê-la utilizando uma linguagem formal definida pelo verificador de modelos. Em UPPAAL, a linguagem usada para especificar propriedades é um subconjunto da lógica temporal TCTL (*Timed Computation Tree Logic*). As fórmulas das propriedades podem ser escritas como: (i) `A[] expressão`, significando que invariavelmente a expressão será válida; (ii) `A<> expressão`, para verificar se no futuro a expressão será válida; (iii) `E[] expressão`, significando que em pelo menos um caminho computacional a expressão é sempre verdade; (iv) `E<> expressão`, para investigar se em pelo menos um estado no futuro a expressão será válida; (v) `p-->q` para denotar que sempre que p for verdade, q também o será. Esta última propriedade também pode ser lida como "p leva a q" e pode ser escrita da seguinte forma `A[]p->A<>q`.

Para exemplificar, será apresentado um modelo de uma lâmpada simples com dois autômatos, um representando a lâmpada e o outro um usuário (baseado no exemplo contido em [14]). O autômato da lâmpada tem três estados: desligada, baixa e alta. O autômato do usuário tem apenas um estado: inativo. O funcionamento é dado por um botão que ao ser pressionado uma vez acende a lâmpada em baixa luminosidade e a desliga se for pressionado novamente. Porém, caso o intervalo entre o primeiro acionamento do botão e o segundo for menor do que 5 unidades de tempo, a lâmpada é acesa em alta luminosidade. O modelo é mostrado na Figura 4.2.

Para demonstrar o uso das fórmulas temporais, neste exemplo foram feitas três verificações simples. A primeira foi `E<> Lampada.Alta` para determinar se em algum estado no futuro, a lâmpada poderia ser acesa em alta luminosidade. A segunda foi verifi-

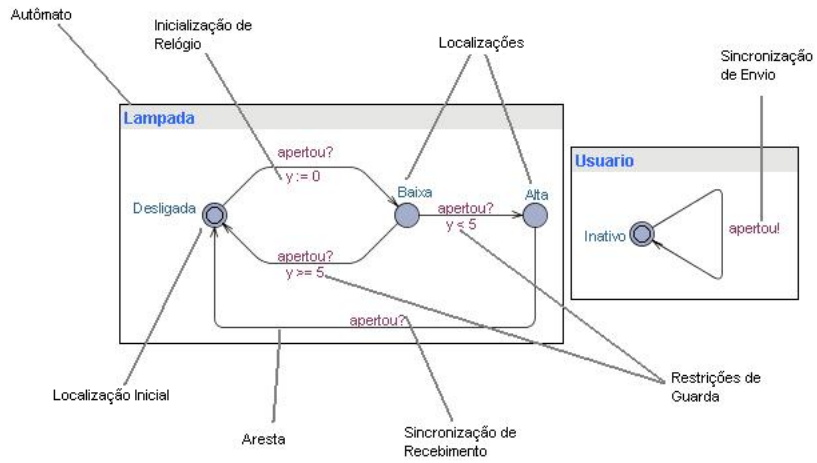

Figura 4.2. *Exemplo da lâmpada em UPPAAL*

cada usando a especificação `E[] Lampada.Desligada`, para determinar se existia algum caminho computacional no qual sempre a lâmpada estaria desligada. Por fim, `A[] not deadlock` foi a especificação usada para verificar se o sistema invariavelmente não entraria em *deadlock*. As respostas para as verificações estão apresentadas na Figura 4.3, as quais demonstram que todas estas propriedades foram satisfeitas.

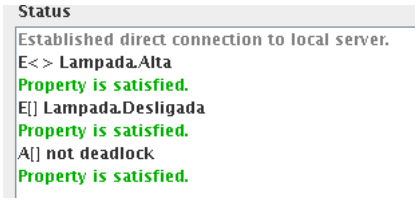

Figura 4.3. *Verificações de três propriedades do sistema da lâmpada.*

Em relação à segunda propriedade, `E[] Lampada.Desligada`, a razão para a mesma ser verdadeira está na existência da transição de atraso (*Delay Transition*) do UPPAAL. Este tipo de transição modela a passagem do tempo sem que haja uma mudança de localização dos autômatos. No entanto, esta permanência nas localizações só é permitida nas seguintes condições: (i) as transições de atraso não podem violar as invariantes; (ii) as localizações envolvidas não podem ser do tipo `urgent` ou `commited`; (iii) não pode haver um canal de sincronização urgente sobre uma aresta de saída em alguma das localizações envolvidas. Dessa maneira, dado que a localização Desligada não possui nenhuma destas restrições, o autômato pode permanecer nesta localização por tempo indeterminado, o

que valida a propriedade.

4.2 CONSIDERAÇÕES FINAIS

O objetivo deste capítulo foi apresentar o verificador de modelos UPPAAL, seus autômatos temporizados e sua linguagem de especificação de propriedades, de modo a ter um melhor entendimento destes conceitos para os próximos capítulos.

Embora a verificação de modelos seja uma técnica já bastante utilizada no contexto de sistemas críticos, poucas abordagens existem no sentido de compatibilizar esta técnica aos sistemas baseados em componentes. Um dos principais desafios é encontrar ferramentas que dêem suporte à integração entre projeto e verificação de sistemas baseados em componentes.

CAPÍTULO 5

TANGRAM - TOOL FOR ANALYSIS OF DIAGRAMS

Este capítulo apresenta o projeto e a implementação de TANGRAM (***T**ool for **An**alysis of Dia**gram**s*), uma ferramenta capaz de traduzir modelos de sistemas de tempo real baseados em componentes, construídos a partir de diagramas UML, em autômatos temporizados. O principal objetivo desta tradução é possibilitar uma fácil inserção da técnica de verificação de modelos ao contexto do desenvolvimento de sistemas de tempo real baseados em componentes, conforme esquema apresentado na Figura 5.1.

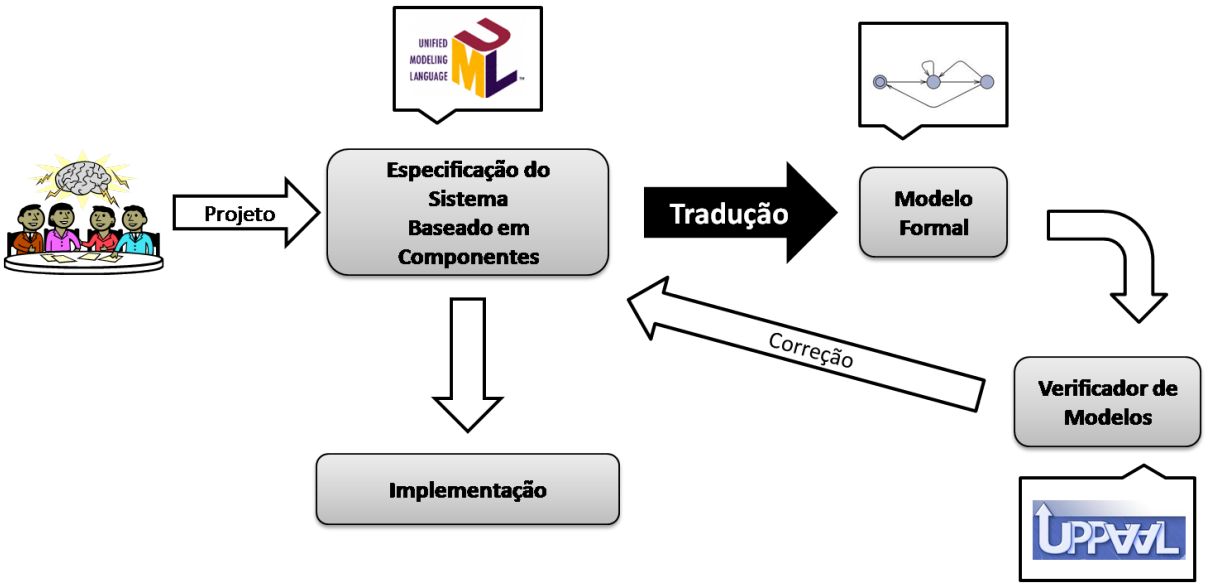

Figura 5.1. Inserção da verificação de modelos no processo de desenvolvimento através da tradução de diagramas da UML.

Em particular, sistemas construídos sobre o Modelo de Componentes Corba (CCM) e voltados para o *middleware* de componentes CIAO são foco desta ferramenta. Os autômatos obtidos através da tradução são direcionados para verificação através do *model checker* UPPAAL.

5.1 VISÃO GERAL

TANGRAM possui três módulos básicos, sendo que cada um desempenha um papel no processo de tradução, conforme apresentado na Figura 5.2. Primeiro, o arquivo UML é fornecido como entrada e uma representação interna dos diagramas de componentes e máquina de estados é feita a partir do conteúdo deste arquivo. Neste passo, as inconsistências dos diagramas já podem ser detectadas, com base nas regras sintáticas definidas no capítulo 3. Em seguida, a ferramenta aplica todas as regras de tradução sobre esta representação dos diagramas para gerar o conjunto de objetos que correspondem aos autômatos temporizados do UPPAAL. Durante este passo, a política de escalonamento e o mecanismo de comunicação através de eventos do CIAO são introduzidos no modelo gerado. A versão atual do TANGRAM permite a utilização de duas políticas de escalonamento, ambas de prioridade fixa, porém sendo uma não-preemptiva e a outra preemptiva, baseada na política *Rate Monotonic* (RM). Todavia, políticas baseadas em prioridade dinâmica são também suportadas pelo CIAO, logo, pretende-se implementar tais políticas em versões futuras da ferramenta. Por fim, os autômatos são exportados para um arquivo XML, de acordo com o formato de entrada definido pelo UPPAAL.

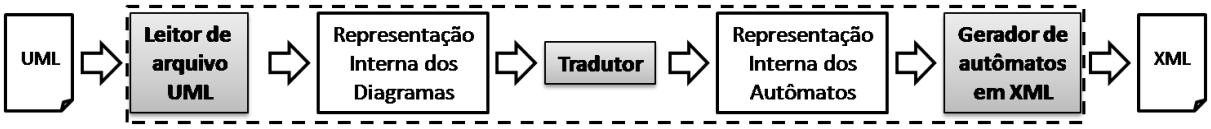

Figura 5.2. Papel de cada módulo do TANGRAM no processo de tradução.

Cada um dos passos citados é implementado por um módulo dedicado dentro da ferramenta. Consequentemente, qualquer módulo pode ser modificado sem afetar os demais. Por exemplo, caso o formato do arquivo de entrada UML sofra alterações, é possível modificar apenas o módulo que lê o arquivo, mantendo os mesmos objetos de saída. Desta forma, o módulo de tradução continuaria funcionando da mesma forma.

5.2 MODELO DE TRADUÇÃO

O procedimento de tradução pode ser dividido em três etapas principais. A primeira etapa consiste basicamente em mapear as declarações de atributos, *facets* e *event sinks* dos componentes em variáveis e canais globais que serão utilizados posteriormente pelos autômatos. Em seguida, são extraídas as informações necessárias à criação e configuração dos autômatos associados às funcionalidades do *middleware* de componentes CIAO. Por fim, a terceira etapa tem a finalidade de traduzir o comportamento dos componentes, especificado através das máquinas de estados.

5.2.1 Primeira etapa: declaração de variáveis globais

Todos os atributos definidos pelo desenvolvedor são mapeados em variáveis globais em UPPAAL. Isto decorre do fato de que cada componente pode possuir diversas máquinas de estado associadas, logo, possuirá diversos autômatos associados. Desta forma, os atributos de um componente devem estar compartilhados entre os seus autômatos.

Conforme apresentado na seção 3.2, os atributos podem ser booleanos ou inteiros. Um atributo booleano será declarado em UPPAAL como uma variável do tipo `bool`. Um atributo inteiro será declarado como `int`.

O identificador das variáveis globais é formado pela concatenação entre o nome do componente e o nome do seu atributo, separados por um caractere "_". Por exemplo, suponha que um componente chamado `Comp1` possua um atributo inteiro `x`. A sua declaração em UPPAAL será dada por "`int Comp1_x;`".

Os atributos dos componentes podem possuir valores iniciais associados. Desta forma, a declaração destes atributos em UPPAAL também será configurada com este valor. Por exemplo, suponha que o atributo `x` do parágrafo anterior possua valor inicial `10`. Neste caso, sua declaração em UPPAAL será dada por "`int Comp1_x = 10;`". Nesta tradução, as variáveis booleanas sempre recebem um valor inicial, mesmo que o seu atributo de

origem não possua este valor de forma explícita. O valor inicial padrão definido para a tradução é `false`. Isto é feito apenas para explicitar o valor inicial desta variável, pois o UPPAAL implicitamente já se encarrega de configurar as variáveis booleanas com este valor.

De forma semelhante ao que acontece para os atributos, as portas do tipo *facet* ou *event sink* também são mapeadas em variáveis globais. Todo *facet* representa uma interface, a qual possui um conjunto de operações. Para cada uma dessas operações é criado um *array* de canais de tamanho 2 em UPPAAL, o qual será utilizado para executar a chamada da operação. O nome deste *array* é dado pela concatenação entre o nome do componente, o nome do *facet* e o nome da operação, separados pelo caractere "_". Por exemplo, suponha que um componente `Comp1` possua um *facet* chamado `port1`, o qual implementa a operação `getValue`. Neste caso, a declaração do *array* de canais em UPPAAL seria dada por "`chan Comp1_port1_getValue[2];`". A primeira posição do *array* é utilizada para chamar a operação, enquanto que a segunda serve para finalizá-la.

Cada *event sink* representa uma porta de entrada para eventos do sistema, os quais são manipulados pelo *middleware* de componentes. Desta forma, o *middleware* deve ser capaz de identificar cada *event sink* e qual tipo de evento ele consome, de modo a despachar os eventos corretamente. Por este motivo, cada *event sink* é mapeado em uma constante inteira em UPPAAL, utilizada para identificar unicamente o *event sink* durante toda a execução do sistema. O nome desta constante é dado pela concatenação entre o nome do componente, o nome do *event sink* e o sufixo `id`, separados pelo caractere "_". O valor da constante é dado por um índice entre zero e a quantidade de *event sinks* no sistema, menos um. Vale ressaltar que nenhum *event sink* possui identificador igual ao outro. Por exemplo, suponha que um componente `Comp1` possua um *event sink* chamado `port2`. Neste caso, a constante será declarada como "`const int Comp1_port2_id = 3`", sendo `3` um índice menor do que a quantidade de *event sinks*.

5.2.2 Segunda etapa: autômatos do middleware

A segunda etapa consiste em configurar e criar instâncias dos autômatos relacionados ao *middleware* de componentes CIAO. A configuração destes autômatos também envolve a declaração de variáveis e canais globais exclusivos à interação com o *middleware*, bem como a declaração de funções que auxiliarão na execução destes autômatos.

5.2.2.1 Variáveis e canais O primeiro passo para a configuração dos autômatos é a construção de uma lista de tarefas com requisitos temporais a partir dos *event sinks*. Segundo esta abordagem, cada *event sink* determina uma tarefa do sistema e os atributos temporais desta tarefa são os atributos definidos pelo desenvolvedor para o *event sink*, conforme extensão proposta na seção 3.2.3.

Com o auxílio desta lista de tarefas, as seguintes variáveis e canais são criadas para atender às funcionalidades do CIAO:

- `NUM_EVENTS` - constante que contém o número de tarefas (ou *event sinks*) do sistema.
- `CAPACITY` - constante que determina a capacidade máxima da fila de eventos do sistema. O valor desta constante é derivado de `NUM_EVENTS`, sendo considerado atualmente com o mesmo valor. Isto decorre do fato de que o modelo de tarefas adotado no momento assume que as tarefas sejam periódicas e que o período de uma tarefa seja igual ao seu *deadline*. Desta forma, sempre haverá no máximo uma instância de cada tarefa em execução, caso contrário, alguma tarefa violou o seu *deadline*.
- `ready_queue` - *array* que representa a fila de eventos do sistema. Esta fila armazena apenas os identificadores dos eventos.
- `queue_size` - variável inteira que representa o tamanho ocupado da fila.
- `queue_overflow` - variável booleana que indica quando um evento é recebido em um momento em que `queue_size` é igual à `CAPACITY`.

- `is_ev_running` - variável booleana que indica se existe alguma tarefa em execução no sistema.
- `in_events` - *array* de canais utilizado pelo *middleware* para receber os eventos do sistema.
- `out_events` - *array* de canais utilizado pelo *middleware* para despachar eventos para o sistema.
- `preemp` - *array* de canais utilizado pelo *middleware* para preemptar uma tarefa em execução, caso alguma política de escalonamento preemptiva esteja sendo utilizada.
- `dispatch` - canal utilizado internamente pelo *middleware* para sinalizar que a próxima tarefa da fila pode ser liberada.
- `finish` - canal utilizado para sinalizar o término de execução de uma tarefa para o *middleware*.

5.2.2.2 Funções Quatro funções foram criadas em UPPAAL para dar suporte à execução dos autômatos do CIAO: (i) `get_priority`; (ii) `push_queue`; (iii) `pop_queue`; e (iv) `next_event`.

A função `get_priority` retorna o valor de prioridade de uma tarefa em um dado momento. A sua estrutura deve variar de acordo com a política de escalonamento adotada. Para uma política baseada em prioridade fixa, o conjunto de prioridades das tarefas do sistema pode ser calculado durante o processo de tradução, a partir da análise das propriedades temporais da lista de tarefas. Por outro lado, para uma política baseada em prioridade dinâmica, a estrutura da função deve permitir o cálculo da prioridade em tempo de execução, a partir das propriedades temporais da tarefa e da configuração do sistema naquele instante.

Até o presente momento, as duas políticas de escalonamento implementadas para o TANGRAM são baseadas em prioridade fixa. Para ambas as políticas, a atribuição

da prioridade é feita de acordo com a abordagem utilizada na política *Rate Monotonic*. Nesta política, o valor da prioridade é proporcional à frequência de ativação das tarefas, ou seja, quanto maior a frequência, mais prioritária será a tarefa. Em outras palavras, quanto menor for o valor do atributo `period` do *event sink*, maior será a sua prioridade. Por questões de implementação, adotou-se que a atribuição do valor da prioridade será crescente em relação à ordem de prioridade, isto é, a tarefa que receber prioridade 1 será a mais prioritária, a que receber prioridade 2 será a segunda mais prioritária e assim por diante.

Um exemplo da função `get_priority` é apresentado no Código 5.1. A estrutura apresentada neste código é de prioridade fixa, conforme mencionado no parágrafo anterior. Na linha 3 está a declaração do *array* de prioridades, que é calculado a partir do período das tarefas. O índice deste *array* corresponde ao ID do *event sink*, por exemplo, a prioridade do *event sink* cujo ID é 4 é dado por `priorities[4]`. Este ID deve ser passado como parâmetro para esta função (linha 1) e é utilizado para obter o valor de retorno (linha 5). O tamanho do *array* deve ser, obviamente, igual à quantidade de tarefas do sistema, o que é indicado pela constante `NUM_EVENTS` (linha 3).

Listing 5.1. Exemplo da função `get_priority`.

```
1 int get_priority(int ev_id){
2
3   const int priorities[NUM_EVENTS] = {5,2,3,4,1};
4
5   return priorities[ev_id];
6 }
```

A segunda função utilizada pelo *middleware* é a `push_queue`, cujo papel é inserir os eventos na fila e ordená-los de acordo com a prioridade. O corpo desta função é apresentado no Código 5.2. Ao contrário da função `get_priority`, esta função pode ser utilizada tanto para políticas de escalonamento baseadas em prioridade fixa, quanto para políticas baseadas em prioridade dinâmica. Para tanto, ela utiliza a função `get_priority` para obter a prioridade da tarefa no momento de inserí-la na fila (linha 11). Entretanto,

embora seja possível construir uma função mais otimizada para atender a políticas de prioridade fixa, com complexidade da ordem de *O(1)*, decidiu-se manter apenas uma função genérica, pois a complexidade das funções em UPPAAL não impacta no tamanho do espaço de estados resultante [14].

O comportamento desta função é dado da seguinte forma: (i) quando a fila está vazia, o evento é inserido na primeira posição (linhas 2-5); (ii) quando a fila possuir algum evento, é feita uma comparação da prioridade do evento a ser inserido com os demais da fila, de modo a inserir o novo evento na posição correta da fila. Os demais eventos menos prioritários serão realocados após o novo evento (linhas 6-19); (iii) quando a fila estiver cheia, a *flag* `queue_overflow` recebe o valor `true` (linha 20), indicando a ocorrência de um estouro da capacidade da fila.

Listing 5.2. Função `push_queue`.

```
1  void push_queue(int ev_id){
2    if (queue_size == 0){
3       ready_queue[0] = ev_id;
4       queue_size++;
5    }
6    else if (queue_size < CAPACITY){
7       int loose_ev = ev_id;
8       int swap;
9       int j=0;
10      for (j=0; j<queue_size; j++){
11         if (get_priority(loose_ev) < get_priority(ready_queue[j])){
12             swap = ready_queue[j];
13             ready_queue[j] = loose_ev;
14             loose_ev = swap;
15         }
16      }
17      ready_queue[queue_size] = loose_ev;
18      queue_size++;
19   }
20   else queue_overflow = true;
```

```
21 }
```

A função `pop_queue` tem o papel de retirar o próximo evento da fila para que o mesmo possa ser utilizado pelo *middleware*. O seu comportamento é bem simples e está apresentado no Código 5.3. Basicamente, esta função retira o evento da primeira posição (linha 2), realoca os eventos na fila (linhas 3-6), atualiza o seu tamanho (linhas 7-8) e retorna o evento obtido no início (linha 9).

Listing 5.3. Função `pop_queue`.

```
1  int pop_queue(){
2    int result = ready_queue[0];
3    int j=0;
4    for (j=0; j<queue_size-1; j++){
5        ready_queue[j] = ready_queue[j+1];
6    }
7    queue_size--;
8    ready_queue[queue_size] = 0; //limpando o fim da fila
9    return result;
10 }
```

Por fim, a função `next_event` é utilizada apenas para identificar qual o próximo evento da fila, sem removê-lo. Ela é apresentada no Código 5.4.

Listing 5.4. Função `next_event`.

```
1  int next_event(){
2    return ready_queue[0];
3  }
```

5.2.2.3 Autômatos As funcionalidades do CIAO são traduzidas em três autômatos: *DispatchingModule*, *EventChannel* e *Timer*. O primeiro representa o módulo de entrega de eventos do Serviço de Eventos de Tempo Real, que atua em conjunto com o Serviço de Escalonamento de Tempo Real. Este módulo tem a responsabilidade de despachar

os eventos do sistema para seus respectivos clientes, de acordo com uma política de escalonamento baseada em prioridades. O segundo autômato representa o módulo de recepção de eventos, cujo papel é interceptar os eventos produzidos pelo sistema e utilizar o serviço de escalonamento para construir a fila de eventos baseada em prioridades. O terceiro e último autômato representa a funcionalidade de eventos de *timeout* oferecida pelo CIAO. Este autômato é instanciado para cada *event sink* que consome um evento de *timeout*.

As variações entre políticas de escalonamento são refletidas diretamente no autômato *DispatchingModule*, pois ele tem o papel de escolher o próximo evento da fila e interromper a execução do evento corrente, se for o caso. Por este motivo, duas variações deste autômato foram elaboradas, uma para contemplar o escalonamento preemptivo e outra para o não-preemptivo.

O autômato mostrado na Figura 5.3 representa a variação preemptiva do *DispatchingModule*. Este autômato contém cinco localizalções: `Idle`, `Start_1`, `Dispatching_1`, `Start_2` e `Dispatching_2`. Com exceção de `Idle`, todas as localizações são do tipo *Committed*, que é utilizado para dar uma prioridade maior ao escalonador e evitar que o mesmo interfira no tempo da aplicação. A partir da localização `Idle`, o autômato aguarda por uma sincronização sobre o canal `dispatch` (ver `dispatch?`). Quando esta sincronização ocorre, o autômato pode tomar três arestas diferentes. A primeira leva à localização `Start_1` e deve ser executada se não existir nenhum evento em execução no sistema no momento (`!is_ev_running`) e existir pelo menos um evento pendente na fila ($queue\_size > 0$). Por outro lado, a segunda aresta leva à localização `Start_2`, a qual deve ser executada caso exista algum evento em execução no sistema (`is_ev_running`). Neste caso, é necessário verificar se a execução deste evento deve ser preemptada, de acordo com sua prioridade. Desta forma, a variável `next_event_id` é atualizada pela função `next_event()`, que retorna a identificação do próximo evento da fila. A terceira aresta mantém o autômato em `Idle` e deve ser tomada caso não exista nenhum evento pendente na fila.

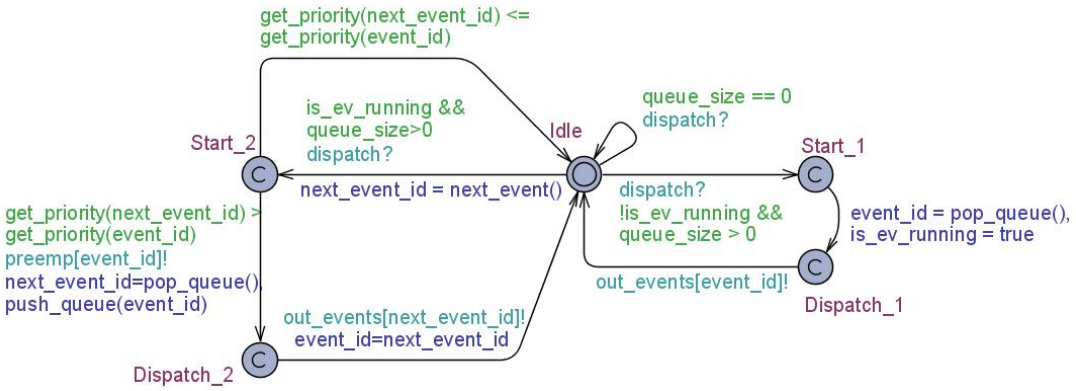

Figura 5.3. Autômato *DispatchingModule*, baseado no Serviço de Escalonamento de Tempo Real do CIAO.

Como pode ser visto na Figura 5.3, há apenas um caminho possível a partir da localização `Start_1`. Este caminho consiste em retirar o próximo evento da fila, utilizando a função `pop_queue()`, e em seguida enviá-lo através do canal `out_events`. O identificador do evento retirado da fila é armazenado na variável local `event_id`. Esta variável sempre contém o identificador do evento cuja tarefa está em execução no sistema.

A partir da localização `Start_2` existem dois caminhos possíveis. O primeiro retorna diretamente para a localização `Idle`, pelo fato da prioridade do evento em execução no sistema ser maior ou igual à prioridade do próximo evento da fila. O segundo caminho, por outro lado, deve ser tomado caso a prioridade do próximo evento da fila seja maior que a do evento corrente. Neste caso, o evento em execução deve ser interrompido através da sincronização sobre o canal `preemp`. Assim, o evento corrente é inserido de volta na fila através da função `push_queue(event_id)` e o próximo evento é enviado através da sincronização `out_events[next_event_id]!`.

A variação não-preemptiva do *DispatchingModule* (Figura 5.4) difere da sua versão preemptiva por não possuir as localizações `Start_2` e `Dispatching_2`, as quais configuram os caminhos que tornam possível a existência de preempção sobre os autômatos.

O autômato *EventChannel* (Figura 5.5) possui duas localizações: `Idle` e `Receive`. A localização `Receive` é do tipo *Urgent*, para evitar que a execução do canal de eventos também interfira no tempo do sistema, e para manter a sua prioridade inferior à do

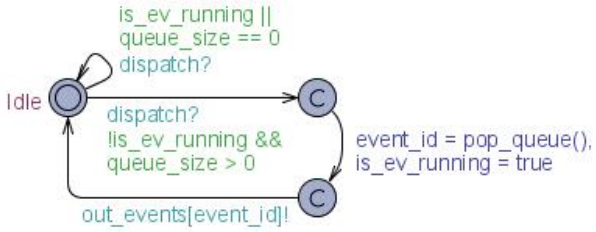

Figura 5.4. Variação não-preemptiva do autômato *DispatchingModule*

DispatchingModule.

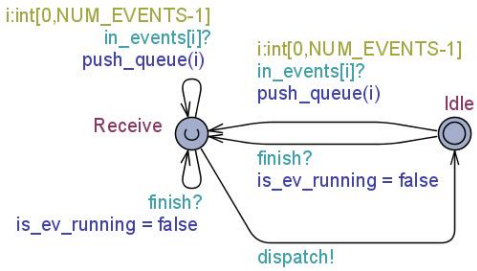

Figura 5.5. Autômato *EventChannel*, baseado no Serviço de Eventos de Tempo Real do CIAO.

A localização `Receive` pode ser alcançada a partir da localização `Idle` através de duas arestas. A primeira é disparada quando um evento é lançado no sistema, através da sincronizaçao `in_events[i]?`. O evento recebido é inserido na fila pela função `push_queue(i)`. A segunda aresta é disparada quando algum evento conclui a sua execução, sincronizando através de `finish?`. O autômato permanece em `Receive` até que todos os eventos produzidos no sistema naquele momento sejam interceptados. Por fim, o autômato *EventChannel* retorna para a localização `Idle`, sincronizando com o autômato *DispatchingModule* através de `dispatch!`. Esta sincronização é utilizada para indicar que a fila de eventos foi atualizada, logo, o escalonador deve decidir qual será o próximo evento a ser executado.

O autômato *Timer* (Figura 5.6) conta um dado período de tempo e lança um evento de *timeout* após a passagem deste período através da sincronização `in_events[ev_id]!`. O tempo gasto na localização `Counting` é incrementado pelo relógio x, o qual é restringido pela invariante $x <= period$. Toda vez que um *timeout* ocorre, o relógio x é zerado para que o evento requerido seja disparado a cada *period* unidades de tempo. Para cada *event*

sink que consome um evento de *timeout*, cria-se uma instância do autômato *Timer*. No momento de criação da instância, o identificador do *event sink* e o seu período de ativação são passados como parâmetro para as variáveis `ev_id` e `period`, respectivamente.

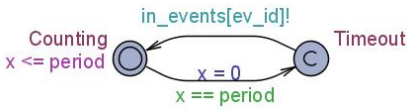

Figura 5.6. Autômato *Timer*, baseado no serviço de eventos de *timeout* do CIAO.

5.2.3 Terceira etapa: tradução das máquinas de estados

Conforme apresentado na seção 3.3.1, devem ser construídas máquinas de estados para as operações dos *facets*, para os *event sinks* e para a funções `start` dos componentes ativos. Para cada máquina de estados modelada em UML será criado um autômato temporizado em UPPAAL. Em geral, a tradução do diagrama de máquina de estados para cada um destes elementos é conduzida da mesma forma, porém existem algumas diferenças, as quais serão apresentadas ao longo desta seção.

5.2.3.1 Declaração de variáveis Para cada autômato são declarados dois relógios básicos: `x` e `timeTrigger`. O primeiro é o relógio utilizado para contabilizar o tempo total de execução do autômato. O segundo, conforme será apresentado mais adiante, será utilizado para a tradução dos *time triggers*.

5.2.3.2 Estados Em geral, cada estado do diagrama UML será mapeado em uma localização no autômato, com o mesmo nome. O pseudo-estado inicial será mapeado em uma localização inicial no autômato chamada `Start`. Para os autômatos correspondentes aos componentes ativos, esta localização inicial deve ser do tipo `Committed`. Desta forma, pode-se garantir que UPPAAL executará estes autômatos assim que o sistema for iniciado.

O estado final, se existir, será mapeado em uma localização do tipo `Committed`

chamada `Final`. Conforme será visto no mapeamento das transições, isto é feito para que o autômato possa retornar imediatamente para sua localização inicial, para que ele esteja pronto para ser re-executado, se for requisitado. Para os autômatos correspondentes aos componentes ativos, esta localização não será `Committed`, pois estes autômatos são executados apenas uma vez, ou seja, não devem retornar para sua localização inicial. Desta forma, se um componente ativo entrar no estado final, ele permanecerá nesse estado até a finalização do sistema.

A Figura 5.7 apresenta a tradução dos estados do diagrama (a), sendo que em (b) estão as localizações que seriam geradas para um autômato relacionado a uma operação de um *facet* ou a um *event sink*, e em (c) estão as localizações que seriam geradas para um autômato relacionado a um componente ativo. Basicamente, a diferença entre as localizações de (b) e (c) são os tipos de `Start` e `Final`.

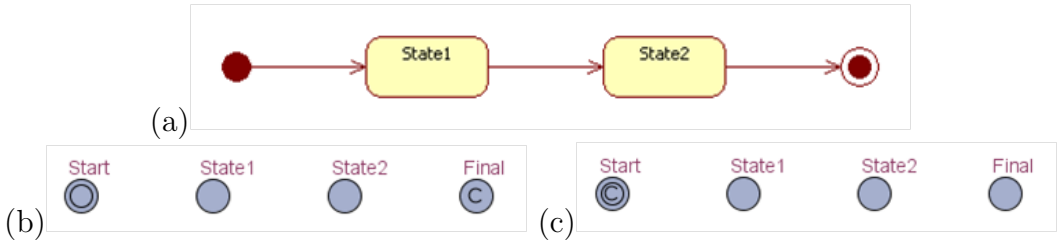

Figura 5.7. Tradução dos estados: (a) diagrama de origem; (b) localizações para *facets* e *event sinks*; (c) localizações para componentes ativos.

5.2.3.3 Transições A princípio, cada transição do diagrama será mapeada em uma aresta no autômato. Entretanto, novas arestas poderão surgir a depender dos efeitos, tipo de elemento ao qual o diagrama está relacionado e política de escalonamento.

A aresta que parte da localização inicial possui uma característica bem particular nos autômatos relacionados às operações dos *facets* ou aos *event sinks*, que é a de viabilizar a invocação destes autômatos, quando os mesmos forem requisitados. Por exemplo, toda vez que uma operação for requisitada para um *facet*, o autômato referente àquela operação deve ser executado. De forma análoga, toda vez que um evento for enviado pelo *middleware* para um *event sink*, o autômato referente àquele *event sink* deve ser

executado.

Esta característica é contemplada através da adição de uma sincronização específica à aresta que parte da localização inicial. Para os autômatos correspondentes às operações dos *facets*, é adicionada uma sincronização de recebimento através da posição zero do *array* de canais relativo à operação, obtido na primeira etapa de tradução, conforme apresentado na Seção 5.2.1. Para os autômatos correspondentes aos *event sinks*, esta sincronização deve ser feita sobre o *array* de canais `out_events`, criado para transmitir os eventos despachados pelo *middleware*, conforme apresentado na Seção 5.2.2.1. A posição a ser utilizada deste *array* é dada pelo identificador do *event sink*, obtido na primeira etapa da tradução (ver Seção 5.2.1).

A Figura 5.8 mostra um exemplo da sincronização de ativação de autômatos referentes a uma operação de um *facet* (a) e a um *event sink* (b), ambos para um componente chamado *Comp*. Em (a), o *facet* seria chamado `port1` e a operação chamada de `operation`. Em (b), o nome do *event sink* correspondente seria `port2`.

(a) 

(b) 

Figura 5.8. Sincronização de ativação: (a) autômato de uma operação de um *facet*; (b) autômato de um *event sink*.

A aresta que parte da localização final também é tratada de forma diferenciada para os autômatos dos *event sinks* e das operações dos *facets*. No primeiro caso, uma sincronização de envio sobre o canal `finish` é adicionada para comunicar ao *middleware* a finalização de uma tarefa do sistema (ver Figura 5.9b). No segundo, uma sincronização de envio sobre a posição 1 do *array* de canais relativo à operação é adicionada para devolver o controle da execução ao componente que requisitou a operação em questão (ver Figura 5.9a).

A tradução das transições envolve também a tradução dos eventos de ativação temporal (*time triggers*), das restrições de guarda e dos efeitos.

Cada *time trigger* é traduzido em um conjunto de inicializações de relógio, um invari-

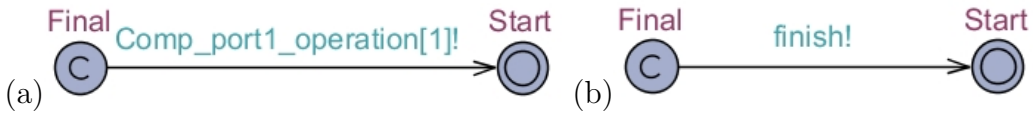

Figura 5.9. Sincronização de finalização: (a) autômato de uma operação de um *facet*; (b) autômato de um *event sink*.

ante e um guarda. Conforme mencionado na Seção 5.2.3.1, o relógio utilizado neste caso é o `timeTrigger`. Seja `E` a aresta correspondente à transição que contém o *time trigger*, `L` a localização de origem de `E` e `A` o conjunto de arestas de entrada de `L`. Para cada aresta em `A`, o valor de `timeTrigger` é zerado. Em `L`, é adicionada a invariante `timeTrigger <= valor`, onde `valor` deve ser substituído pelo valor especificado pela função `after` do *time trigger*. Por fim, em `E`, acrescenta-se o guarda `timeTrigger >= valor`. Desta forma, UPPAAL ativará a aresta `E` assim que o tempo especificado por `valor` foi atingido. Um exemplo do resultado da tradução de um *time trigger* é apresentado na Figura 5.10.

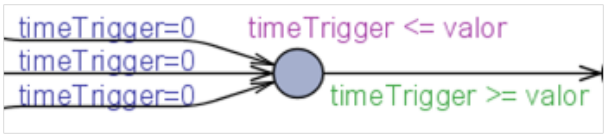

Figura 5.10. Resultado da tradução de um *time trigger*.

Vale ressaltar que caso haja mais de um *time trigger* envolvendo o mesmo estado, o valor da invariante deve ser definido pelo maior valor entre os *time triggers*. Desta forma, caso este valor seja atingido, todas as arestas do autômato estarão ativadas.

As restrições de guarda são traduzidas diretamente para guardas em UPPAAL, mantendo-se a mesma sintaxe. Somente são substituídos os nomes dos atributos dos componentes pelos seus devidos identificadores declarados em UPPAAL, conforme apresentado na seção 5.2.1. Caso já exista algum guarda na aresta de destino, uma conjunção é feita entre o novo guarda e o guarda existente.

Os efeitos podem ser de três tipos: envio de sinal, chamada de operação e atribuição de valor a um atributo. A tradução mais simples é da atribuição de valor, pois o seu mapeamento é quase direto, dada a similaridade da sua sintaxe com a do UPPAAL. Entretanto, tal como ocorre com a restrição de guarda, os nomes dos atributos dos componentes

devem ser substituídos pelos seus identificadores declarados em UPPAAL.

O envio de evento é traduzido em uma sincronização de envio sobre o *array* de canais `in_events`, na posição definida pelo identificador do *event sink* associado, conforme declaração apresentada na Seção 5.2.1. Por exemplo, suponha que `Comp_port2_id` seja o identificador do *event sink* que recebe o evento enviado na transição em questão. Neste caso, este envio será traduzido na sincronização `in_events[Comp_port2_id]!` (ver Figura 5.11a). Por outro lado, caso o *event source* seja do tipo *Multiplex*, isto é, com múltiplos *event sinks* associados, deve-se adicionar uma localização urgente intermediária e uma aresta para cada *event sink* além do primeiro. Em cada localização intermediária, uma sincronização é adicionada para o seu respectivo *event sink*. Por exemplo, suponha que `evId1`, `evId2` e `evId3` sejam os identificadores dos *event sinks* que recebem o evento enviado na transição em questão. Neste caso, duas localizações urgentes e arestas intermediárias serão adicionadas para contemplar as sincronizações sobre `in_events[evId2]!` e `in_events[evId3]!`, tal como apresentado na Figura 5.11b.

Figura 5.11. Tradução de efeito de envio de evento: (a) *event source Simplex*; (b) *event source Multiplex*.

Por fim, um efeito de chamada de operação é traduzido em duas sincronizações, uma aresta e uma localização intermediária. Considere `E` a aresta correspondente à transição em questão, `L` a localização intermediária criada e `A` a aresta intermediária criada. Define-se `L` como a nova localização de destino de `E` e como localização de origem de `A`. A antiga localização de destino de `E` passa a ser a localização de destino de `A`. Uma sincronização de envio sobre a posição 0 (zero) do *array* de canais declarado para a operação em questão é adicionada à `E`. Em seguida, uma sincronização de recebimento sobre a posição 1 (um) deste mesmo *array* de canais é adicionada sobre `A`. A primeira sincronização representa a chamada da operação, enquanto que a segunda representa o seu retorno. A Figura 5.12 apresenta o resultado da tradução de um efeito de chamada de operação. Nesta figura, a localização `wait_Comp_port_func` representa a localização `L` da explicação anterior.

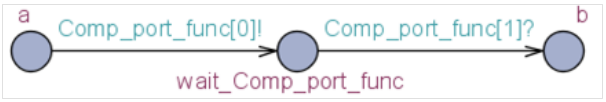

Figura 5.12. Resultado da tradução de um efeito de chamada de operação.

Estas duas sincronizações são complementares às sincronizações de inicialização e finalização de autômatos referentes às operações dos *facets*, conforme apresentado no início desta seção. Desta forma, elas têm a capacidade de iniciar e aguardar o término de uma operação, respectivamente.

5.2.3.4 Suporte à preempção Para dar suporte às políticas de escalonamento preemptivas, adotou-se uma solução de criar uma localização especial chamada `suspend`, na qual o autômato deve permanecer quando a sua execução for suspensa pelo *Dispatching-Module* (ver Seção 5.2.2.3). Este tipo de localização deve ser adicionado aos autômatos correspondentes aos *event sinks*, pois estes são gerenciados pelos autômatos do *middleware*.

A Figura 5.13 apresenta a localização `suspend` e a forma como ela é ligada a uma localização convencional do autômato. A grosso modo, ao receber um comando de preempção, o autômato vai para a localização `suspend` e inicia a contagem do tempo de suspensão. Quando um comando de ativação for enviado novamente ao autômato, ele deve retornar para a localização em que parou e continuar sua execução. O tempo contabilizado de suspensão deve ser utilizado para ajustar a avaliação de guardas e invariantes associadas à localização que sofreu preempção.

De modo a viabilizar esta solução, quatro novas variáveis devem ser declaradas no autômato. A primeira delas é o relógio `z`, utilizado para cronometrar o tempo de cada preempção. Duas outras variáveis inteiras são utilizadas para armazenar a quantidade de tempo em que o autômato esteve suspenso, `timeSuspLoc` e `timeSuspGlob`. A variável `timeSuspLoc` guarda o tempo de suspensão de uma localização específica, enquanto que `timeSuspGlob` grava a duração de todas as preempções sofridas pelo autômato. Por fim,

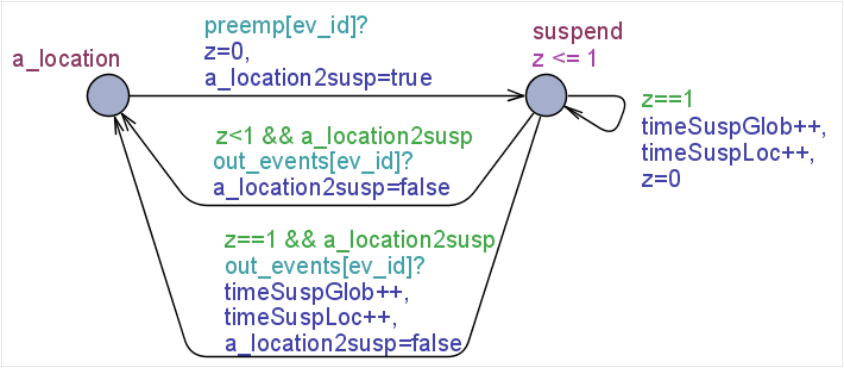

Figura 5.13. Localização especial `suspend` utilizada para dar suporte à preempção.

uma variável booleana é utilizada para identificar em que localização o autômato sofreu preempção, de modo que seja possível retornar para esta mesma localização quando este autômato for reativado. O nome desta variável é dado por `<nome da localização>2susp`. Desta forma, haverá uma variável booleana para cada localização passível de preempção no autômato.

Na aresta que leva à localização `suspend` (ver Figura 5.13), é adicionada uma sincronização de recebimento sobre o *array* de canais `preemp`, na posição correspondente ao identificador do evento relativo ao autômato em questão (ver `preemp[ev_id]?` na figura). Nesta mesma aresta, o relógio `z` é zerado e a variável booleana que indica a localização de origem recebe o valor `true`.

Na localização `suspend`, uma aresta circular incrementa o valor de `timeSuspLoc` e `timeSuspGlob` para cada unidade de tempo contabilizada pelo relógio `z`. Quando um comando de reativação do autômato for recebido (indicado na figura por `out_events[ev_id]?`), deve-se verificar o valor de `z` para saber se as variáveis `timeSuspLoc` e `timeSuspGlob` devem ser incrementadas antes de retornar à localização de origem. Desta forma, deve existir uma aresta de retorno que incrementa estas variáveis e outra aresta que não incrementa. Ambas as arestas utilizam a variável booleana como guarda para retornar à localização correta (ver `&& a_location2susp` na figura). Além disso, as arestas também atualizam o valor da variável booleana para `false`.

A princípio, todas as localizações do autômato são passíveis de preempção, exceto a

localização inicial e a final, se existirem. Em todas as arestas que levam a uma localização passível de preempção, deve-se inicializar o valor da variável `timeSuspLoc`. Além disso, todos os guardas e invariantes sobre relógios devem ser ajustados para verificar o valor do relógio, decrementando o valor de `timeSuspLoc`. Por exemplo, seja o invariante sobre o relógio `x` dado por `x <= 10`. Neste caso, o invariante deve passar para a forma `(x - timeSuspLoc) <= 10`. Isto deve ser feito para garantir que o tempo de preempção seja transparente para o autômato. A variável `timeSuspGlob`, por sua vez, deve ser inicializada apenas na aresta que parte da localização inicial.

5.3 IMPLEMENTAÇÃO

TANGRAM foi desenvolvido na linguagem Java, o que permite uma fácil portabilidade da ferramenta para diversos sistemas operacionais e plataformas de *hardware*. Além disso, Java oferece bibliotecas-padrão para leitura e interpretação de arquivos XML, o que proporcionou um ganho de produtividade no desenvolvimento do módulo de leitura dos diagramas da UML.

A ferramenta foi projetada de forma modular, de modo a permitir um melhor desacoplamento e facilitar possíveis correções e melhorias posteriores.

5.3.1 Estrutura de pacotes

A estrutura de pacotes Java que compõem TANGRAM está apresentada na Figura 5.14. No topo desta estrutura está o pacote *Tangram*, que representa o *front-end* da aplicação. Este pacote contém as classes de interface com o usuário e a classe principal que controla as etapas do processo de tradução. No nível intermediário existem três pacotes, os quais estão relacionados aos três módulos do processo de tradução citados na Seção 5.1. O pacote *UMLInterpreter* implementa o módulo que lê o arquivo de entrada UML. O pacote *UML2UPPAAL* implementa a tradução propriamente dita entre UML e UPPAAL, e é o principal pacote da ferramenta. O terceiro pacote deste nível, *Xml-*

Producer, implementa o módulo que exporta os arquivos XML de acordo com o formato de entrada do UPPAAL. Os pacotes do último nível contém as classes que representam os elementos dos diagramas UML, os elementos de configuração do *middleware* CIAO, as políticas de escalonamento (*RT*) e a representação dos autômatos temporizados do UPPAAL.

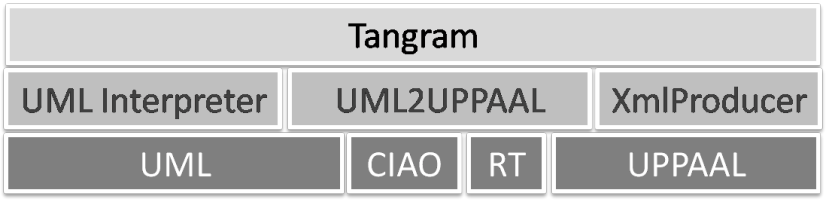

Figura 5.14. Estrutura de pacotes que compõem TANGRAM.

5.3.1.1 UMLInterpreter O pacote *UMLInterpreter* é responsável por gerar uma representação interna dos diagramas UML a partir do arquivo de entrada fornecido pelo usuário. Para cada um dos elementos dos diagramas definidos no Capítulo 3, foi criada uma classe Java correspondente. Todas essas classes estão definidas no pacote *UML*. A execução deste módulo consiste em percorrer o arquivo XML de entrada em busca dos elementos que constituem os diagramas envolvidos neste trabalho e instanciar as classes correspondentes.

Embora o *Object Management Group* (OMG) tenha definido um formato XML padronizado para representação dos diagramas UML, a implementação do *UMLInterpreter* foi feita direcionada para o formato de saída específico da ferramenta StarUML [46]. Isto foi feito para contornar o problema de que, durante a elaboração deste projeto, não foi encontrada uma outra ferramenta livre (*open source*) ou gratuita que atendesse aos requisitos de modelagem para TANGRAM e aos padrões de formatação XML definidos pela OMG, ao mesmo tempo. Entretanto, dada a característica modular de TANGRAM, é possível alterar o código do *UMLInterpreter* para ler um arquivo no formato padrão da OMG, mantendo a mesma representação dos diagramas em Java.

5.3.1.2 UML2UPPAAL Este é o principal módulo da ferramenta, pois nele está a implementação da tradução entre diagramas e autômatos. O projeto deste módulo permite que qualquer mudança nas regras de mapeamento não causem impacto sobre os demais módulos.

UML2UPPAAL depende de dois pacotes essenciais no processo de tradução: CIAO e RT. O pacote CIAO contém as regras de tradução associadas ao *middleware* de componentes, como a criação dos autômatos relacionados aos Serviços de Eventos e de Escalonamento. Esta separação de papéis entre os pacotes permite que a ferramenta esteja apta a realizar traduções para outros *middlewares* de componentes futuramente.

O pacote RT contém as classes e as regras de negócio relacionadas a tempo real, como a classe que representa uma "tarefa" e as políticas de escalonamento que podem ser adotadas. Neste momento, o pacote RT possui apenas uma política de prioridade fixa não-preemptiva e a política *Rate Monotonic*. Este pacote poderá ser utilizado também para executar uma análise de escalonamento das tarefas do sistema, numa próxima versão desta ferramenta.

A Figura 5.15 apresenta uma abstração da sequência de execução do módulo de tradução. Primeiro, é feita a declaração de todas as variáveis globais que puderem ser extraídas do diagrama de componentes enviado ao tradutor (*Parse Global Declarations*). Em seguida, o módulo CIAO é utilizado para realizar a criação de autômatos e variáveis relacionadas ao *middleware* de componentes (*Set Middleware Configuration*). Por fim, os diagramas de máquina de estados são traduzidos em autômatos temporizados (*Translate Statecharts*) e retornados à camada superior da ferramenta. Esta sequência de execução está diretamente relacionada às três etapas do modelo de tradução descritas na Seção 5.2.

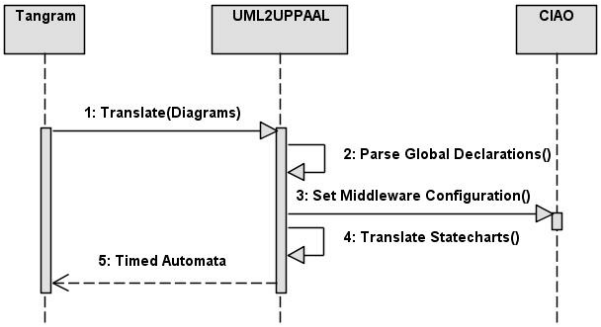

Figura 5.15. Esquema de execução do módulo UML2UPPAAL.

5.4 EXEMPLO

Nesta seção, uma aplicação real, porém simples, com requisitos temporais, que faz parte de um sistema de controle de trens[1], é utilizada para exemplificar a aplicação da abordagem proposta neste trabalho. O principal objetivo desta aplicação é detectar a atividade do operador no controle da velocidade e do freio de um trem. Este sistema é também conhecido como "Sistema Homem-Morto", ou em inglês "*Dead-man's Vigilance Device*".

O sistema é estruturado fisicamente conforme apresentado na Figura 5.16. Ele contém uma chave principal associada com as entradas A e B. De acordo com a figura, apenas uma destas entradas pode estar ativa a cada vez. A entrada C é utilizada para desativar ambas as saídas D e E e deve permanecer ligada enquanto o sistema estiver operacional. A saída E dispara um alarme sonoro (*buzzer*) enquanto que a saída D aciona o freio de emergência do trem.

O sistema deve operar da seguinte maneira: a cada T unidades de tempo, o operador deve pressionar/soltar a chave principal. Caso isto não seja feito, o sistema deve ativar o *buzzer* durante 4 segundos ou até que o operador pressione/solte a chave novamente. Se o operador não responder ao sinal do *buzzer*, o que significa que ele não está mais controlando a velocidade e o freio do trem como deveria, o sistema deve acionar o freio

[1]A especificação do sistema foi gentilmente oferecida pelo grupo **AeS**, uma companhia especializada em sistemas embarcados e de controle.

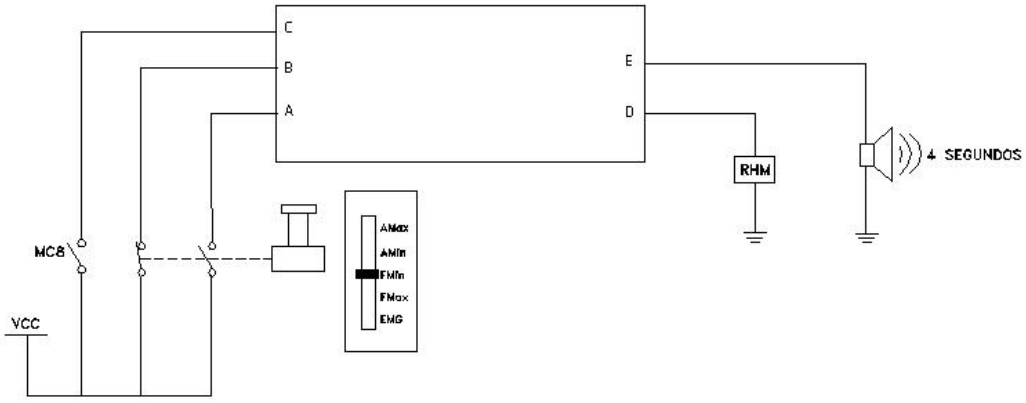

Figura 5.16. Estrutura física do Sistema Homem-Morto.

de emergência do trem imediatamente.

5.4.1 Modelagem estrutural

O diagrama de componentes construído para este exemplo está apresentado na Figura 5.17. Dois componentes ativos foram criados, um para representar a interação entre o operador e o sistema (ver componente `Input` na Figura 5.17), e outro para representar o controle do sistema (ver componente `Control` na figura).

O componente `Input` possui dois *event sources* (`esource_C` e `esource_Switch`), os quais produzem os eventos `C` e `Switch`, respectivamente. O componente `Control` consome estes dois eventos através das portas `esink_C` e `esink_Switch`. Conforme apresentado no modelo, restrições temporais foram atribuídas a estas portas. Estas restrições foram definidas de acordo com os requisitos da aplicação.

É importante levar em consideração algumas características da aplicação durante o processo de modelagem. Primeiro, está claro que ambas as portas `esource_C` e `esource_Switch` não são periódicas, já que as mesmas são ativadas pelo operador. Entretanto, supondo que as restrições do sistema sejam do tipo *hard* (nenhum *deadline* pode ser perdido), deve-se considerar o tempo mínimo entre chegadas como sendo seus

períodos, tal como é recomendado pela comunidade de tempo real para modelar eventos esporádicos [33]. Além disto, as portas `esink_C` e `esink_Switch` apenas atualizam os valores dos atributos no componente `Control` e esta operação possui um custo de execução muito baixo. Portanto, por questões de simplificação, seus WCETs (*worst-case execution time*) foram considerados desprezíveis.

O componente `Control` possui dois atributos booleanos: `activated_C` e `turned_Switch`. Em geral, os atributos não são explicitados no diagrama, mas podem ser manipulados através da ferramenta de UML.

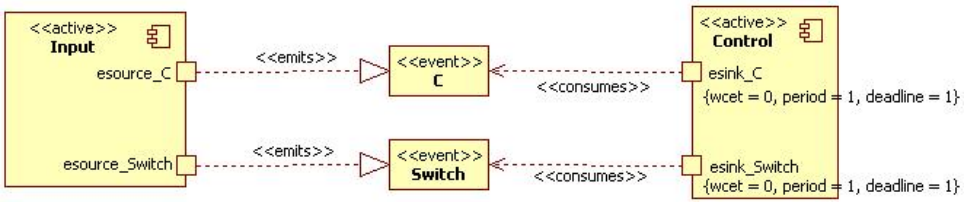

Figura 5.17. Diagrama de componentes para o Sistema Homem-Morto.

5.4.2 Modelagem comportamental

A estrutura do sistema dada na Figura 5.17 contém dois componentes ativos e dois *event sinks*. Considerando a abordagem para modelagem comportamental apresentada na Seção 3.3.1, quatro diagramas de máquina de estados foram construídos para especificar o comportamento do sistema.

O diagrama de máquina de estados relacionado à porta `esink_Switch` é mostrado na Figura 5.18(a). O papel deste *event sink* é atualizar o valor do atributo `turned_Switch`. Neste caso, apenas um estado (`Update`) é necessário para modelar o seu comportamento. A ação de atualizar o atributo é modelada como um efeito na transição que parte do estado `Update`. Esta transição possui uma restrição temporal representada pelo *time trigger* `after(0)`, o que significa que ela deve ser executada imediatamente após a entrada no seu estado de origem. Consequentemente, o tempo total gasto por este *event sink* é insignificante quando comparado aos requisitos temporais do sistema, os quais

estão definidos em termos de segundos. Uma idéia similar foi utilizada para modelar o comportamento da porta `esink_C`, cujo diagrama de máquina de estados é apresentado na Figura 5.18(b).

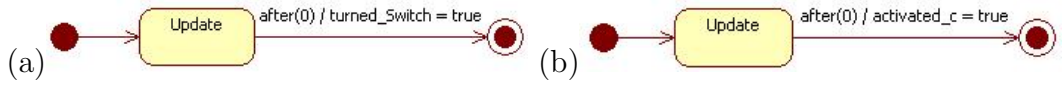

Figura 5.18. Modelagem comportamental dos *event sinks* do componente `Control`: `esink_Switch` (a) e `esink_C` (b).

O comportamento relacionado à função *start* do componente ativo `Input` está especificado no diagrama da Figura 5.19. A idéia por trás desta máquina de estados é representar as interações do operador com o sistema através da chave principal. O operador pode estar no estado `Normal` ou no estado `Dead`. No primeiro estado, o operador pode pressionar a chave, desativar o sistema (evento `C`) ou ir para o estado `Dead`. No segundo caso, nenhuma interação posterior com o sistema pode ser realizada. Conforme pode ser notado na figura, o comportamento do operador foi restringido para que o mesmo pudesse realizar apenas uma operação por unidade de tempo. Isto é feito através da adição do *time trigger* `after(1)` em todas as transições da máquina de estados.

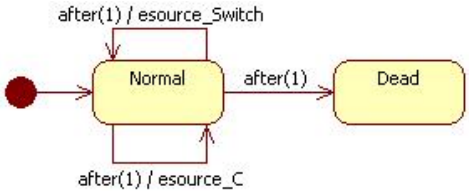

Figura 5.19. Modelagem comportamental da função *start* do componente ativo `Input`.

Por fim, a máquina de estados associada com a função *start* do componente ativo `Control` é apresentada na Figura 5.20. Inicialmente, o controle está no estado `Operating`. Enquanto o operador pressionar regularmente a chave principal, o sistema permanecerá neste estado. Caso o operador ative a entrada `C`, o sistema irá para o estado `Off`, até que ele seja ativado novamente. Após 10 unidades de tempo sem nenhuma ação do operador, o sistema dispara o alarme sonoro (estado `Buzz`) e então aguarda 4 unidades de tempo por uma resposta. Se nada ocorrer dentro deste intervalo, o freio de emergência do trem é acionado e o sistema vai para o estado `Emergency`.

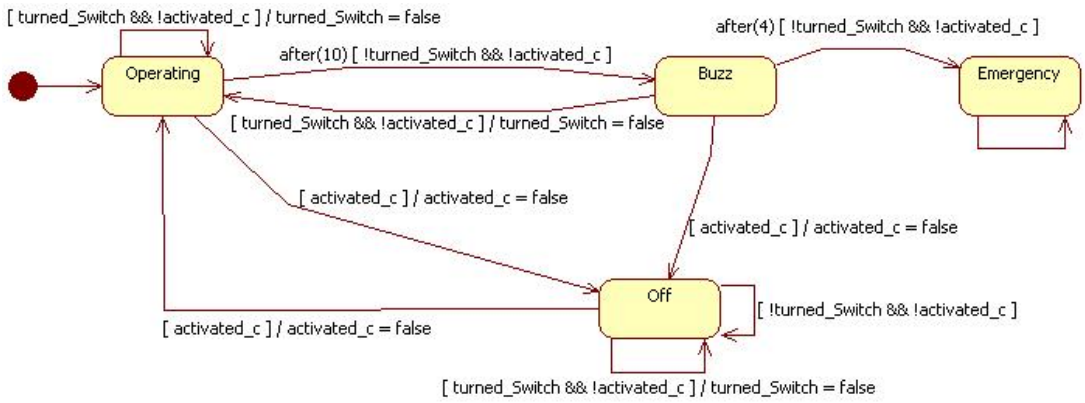

Figura 5.20. Modelagem comportamental da função *start* do componente ativo `Control`.

5.4.3 Tradução

O primeiro passo da tradução é a declaração de variáveis globais a partir da informações contidas no diagrama de componentes. O Código 5.5 mostra a declaração de variáveis globais em UPPAAL para este exemplo.

Listing 5.5. Declaração de variáveis globais em UPPAAL para o Sistema Homem-Morto.

```
1  bool Control_activated_c = false;
2  bool Control_turned_Switch = false;
3  const int Control_esink_C_wcet = 0;
4  const int Control_esink_C_period = 1;
5  const int Control_esink_C_deadline = 1;
6  const int Control_esink_C_id = 0;
7  const int Control_esink_Switch_wcet = 0;
8  const int Control_esink_Switch_period = 1;
9  const int Control_esink_Switch_deadline = 1;
10 const int Control_esink_Switch_id = 1;
11 const int NUM_EVENTS = 2;
12 clock global_time;
13 chan in_events[2];
14 chan out_events[2];
15 chan preemp[2];
16 chan dispatch;
17 chan finish;
```

```
18 bool is_ev_running = false;
19 const int CAPACITY = 2;
20 int ready_queue[2];
21 int queue_size = 0;
22 bool queue_overflow = false;
```

As linhas 1 e 2 apresentam a tradução dos atributos do componente `Control`. As linhas 3 a 10 contêm a declaração das variáveis relacionadas aos *event sinks* `esink_C` e `esink_Switch`, pertencentes ao componente `Control`. Por fim, as linhas 11 a 22 apresentam as variáveis destinadas à configuração e execução dos autômatos do *middleware*. O valor 2 utilizado para inicializar algumas destas variáveis é obtido do valor de `NUM_EVENTS`, que representa a quantidade de *event sinks* no sistema.

Em seguida, os atributos temporais dos *event sinks* são avaliados para a construção da função `get_priority`, conforme abordagem apresentada na Seção 5.2.2.2. A declaração da função `get_priority` para este exemplo é mostrada pelo Código 5.6. Por questões de implementação, adotou-se que as tarefas sempre terão valores de prioridade diferentes, mesmo que seus períodos sejam iguais.

Listing 5.6. Função `get_priority`.

```
1 int get_priority(int ev_id){
2   const int priorities[NUM_EVENTS] = {1,2};
3   return priorities[ev_id];
4 }
```

Os autômatos relativos às funcionalidades do *middleware* foram construídos segundo a abordagem da Seção 5.2.2.3. Dado que a política de escalonamento escolhida para este exemplo foi a *Rate Monotonic*, foi utilizado o autômato `DispatchingModule` em sua versão preemptiva (ver Figura 5.3). Os demais autômatos do *middleware* não variam, portanto são sempre aplicados da mesma forma. Vale ressaltar que neste exemplo não foi necessário utilizar o recurso dos eventos de *timeout* periódicos, logo, nenhuma instância do autômato `Timer` foi criada. Isto se deve ao fato de que os atributos temporais foram modelados através *time triggers*.

Por fim, as máquinas de estados são traduzidas em autômatos temporizados. A Figura 5.21 mostra o autômato obtido a partir da tradução da máquina de estados do *event sink* `esink_C` (Figura 5.18b). Em geral, cada estado do diagrama possui uma localização correspondente no autômato e cada transição possui uma aresta. Neste caso, foi adicionada a localização `suspend` para dar suporte à preempção, conforme abordado na Seção 5.2.3.4. Vale lembrar que tanto a seleção do autômato a ser executado, quanto a própria preempção são realizados pelo autômato `DispatchingModule`.

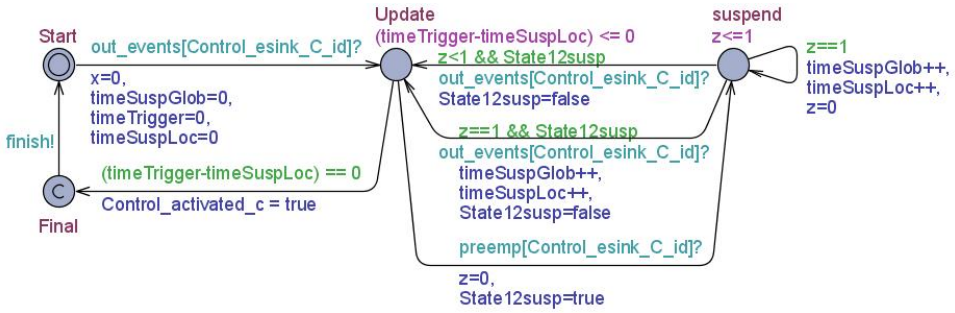

Figura 5.21. Autômato obtido a partir da tradução do *event sink esink_C* do componente `Control`.

O *time trigger* `after(0)` foi traduzido em um guarda e um invariante sobre o relógio `timeTrigger`. Como pode ser visto na figura, o tempo gasto na localização `Update`, menos a duração da preempção, não pode ser maior do que zero `(timeTrigger-timeSuspLoc <= 0)`. Isto significa que o autômato não poderá permanecer nesta localização, a menos que ele sofra preempção.

O efeito de atribuição *activated_c = true* do diagrama é traduzido em uma atribuição equivalente em UPPAAL (`Control_activated_c = true`). A principal diferença é que todos os atributos traduzidos devem receber o nome do seu componente associado como prefixo, para evitar identificadores duplicados.

O autômato relacionado ao *event sink* `esink_Switch` é muito similar ao anterior, logo, ele não será apresentado aqui. Os autômatos correspondentes à função *start* dos componentes `Control` e `Input` são mostrados nas Figuras 5.22 e 5.23, respectivamente.

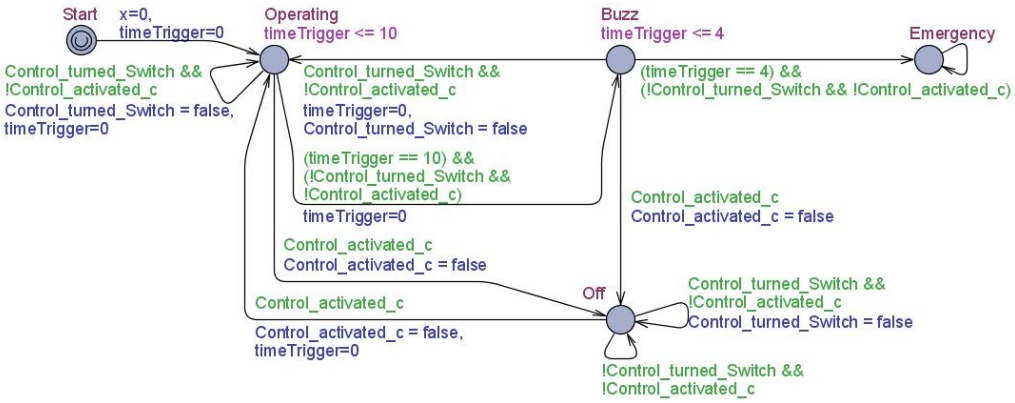

Figura 5.22. Autômato correspondente à função *start* do componente ativo `Control`.

Os autômatos de ambos os componentes `Control` e *Input* preservam grande similaridade com os seus diagramas de origem. Em outras palavras, existe um mapeamento um-para-um entre as localizações do autômato e os estados do diagrama. Isto decorre do fato de que a abordagem proposta neste trabalho foi projetada de modo a preservar a modelagem original do desenvolvedor. Sob uma perspectiva de engenharia de *software*, esta é uma característica importante, uma vez que ela permite que o desenvolvedor mantenha uma rastreabilidade sobre as funcionalidades originais do componente, tanto sobre os diagramas de máquinas de estados, quanto sobre seus autômatos correspondentes.

O comportamento da função *start* do componente ativo `Control` é principalmente baseado no monitoramento dos valores dos atributos `Control_turned_Switch` e `Control_activated_c`, além de controlar o tempo gasto nas localizações `Operating` e `Buzz`. Este autômato não possui interação com outros autômatos. Os *time triggers* *after(10)* e *after(4)* foram tratados de maneira similar à do autômato do *event sink* `esink_C`, com exceção apenas do ajuste feito neste último para suporte à preempção.

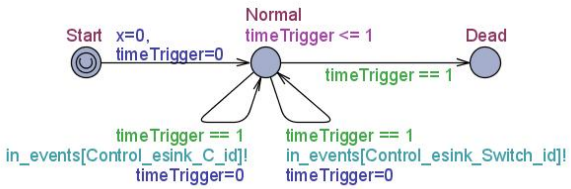

Figura 5.23. Autômato correspondente à função *start* do componente ativo `Input`.

O autômato correspondente à função *start* do componente ativo `Input` (Figura 5.23) usa o canal `in_events` para sincronizar com o autômato `EventChannel` (*middleware*), o que representa o envio de um novo evento que deve ser inserido na fila de eventos baseada em prioridades.

5.4.4 Verificação de propriedades

A seguir são apresentadas algumas propriedades especificadas em TCTL, as quais foram verificadas para os autômatos do sistema gerados por TANGRAM.

- `A[] not deadlock`: ausência de *deadlock*. Esta propriedade é satisfeita pelo modelo.
- `E<> Control_start_proc.Emergency`: esta fórmula de *reachability* verifica se existe um caminho computacional onde, no futuro, a função *start* do componente `Control` se encontre na localização `Emergency`. Ela é satisfeita pelo modelo.
- `A[] Control_start_proc.Emergency imply Input_start_proc.Dead`: esta propriedade está relacionada à anterior, pois ela verifica se para todos os casos onde a função *start* do componente `Control` esteja na localização `Emergency`, então a função *start* do componente `Input` estará na localização `Dead`. Isto exclui a possibilidade de ter o freio de emergência do trem ativado enquanto o operador esteja em sua condição normal. Esta propriedade também é satisfeita pela modelo.
- `Input_start_proc.Dead --> Control_start_proc.Emergency`: esta propriedade verifica se a localização `Emergency` sempre seria alcançada quando o autômato da função *start* do componente `Input` estivesse na localização `Dead`. O objetivo era verificar se o trem não continuaria a andar mesmo após a "morte" do operador, o que é obviamente um cenário indesejado. De fato, o verificador de modelos apontou que este cenário era possível. O contra-exemplo apresentado pelo verificador de modelos mostrou que esta situação pode acontecer quando o operador ativa a entrada `C` no instante anterior à sua "morte", desativando todo o sistema.

Outras propriedades, não mostradas aqui explicitamente, foram verificadas de modo a identificar possíveis erros no modelo de tradução e outras propriedades da aplicação. Algumas delas estão relacionadas à análise de escalonamento. Embora existam derivações analíticas que podem ser utilizadas [33], a verificação de modelos pode apontar cenários não-escalonáveis. Esta informação pode ser muito útil para tomada de decisão por parte dos desenvolvedores sobre os seus sistemas.

5.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a ferramenta TANGRAM, desenvolvida para dar suporte à abordagem proposta por este trabalho de verificação de sistemas baseados em componentes através da tradução de diagramas da UML em autômatos temporizados da ferramenta UPPAAL.

A implementação de TANGRAM é dividida em três módulos básicos, onde cada um possui um papel específico no processo de tradução. O módulo *UMLInterpreter* é responsável por ler o arquivo dos diagramas UML e gerar uma representação interna dos mesmos. Embora este módulo esteja tratando um arquivo de entrada que não segue o padrão definido pela OMG, pois não foi encontrada uma ferramenta livre (*open source*) ou gratuita que fizesse isto e que atendesse aos requisitos de TANGRAM, é possível (e desejável) alterar este módulo futuramente para ler um arquivo padronizado. Esta modificação poderá ser feita sem impactar nos demais módulos já implementados.

O módulo *UML2UPPAAL* contempla a tradução entre diagramas e autômatos. Embora esta tradução não tenha sido validada formalmente, os resultados obtidos até o momento apontam que o comportamento dos modelos gerados é consistente com o comportamento dos seus diagramas originais. Este tipo de constatação pode ser feita através da própria verificação dos modelos gerados em UPPAAL. Caso as respostas das verificações e os contra-exemplos gerados por UPPAAL estejam de acordo com o que era esperado para os modelos especificados em UML, então o comportamento dos autômatos corresponde ao comportamento dos diagramas. Todavia, uma validação formal desta

consistência é muito importante, porém muito trabalhosa também, portanto será tratada em trabalhos futuros.

Um aspecto interessante sobre a verificação dos modelos gerados por TANGRAM é como o desenvolvedor interpretará as respostas obtidas pelo verificador e como ele utilizará isto para melhorar os seus diagramas originais. Isto pode ser feito através de uma análise manual por parte do desenvolvedor, onde ele irá comparar a resposta da verificação com o seus diagrama em UML e fará as correções necessárias. No caso de TANGRAM, este tipo de análise não será algo tão complexo, pois os modelos gerados em UPPAAL mantêm grande semelhança com seus diagramas originais, diferindo muitas vezes apenas na nomenclatura. Por outro lado, uma abordagem ideal seria uma tradução automática reversa, da resposta gerada pelo verificador para o diagrama em UML. Por exemplo, uma idéia interessante seria gerar um Diagrama de Sequência [41] para cada contra-exemplo gerado pelo verificador de modelos. Desta forma, a análise do desenvolvedor seria feita de diagrama UML para diagrama UML, sem dependência direta sobre os autômatos gerados.

Este capítulo apresentou ainda um pequeno exemplo de modelagem, tradução e verificação para um sistema simples, porém real, utilizado em trens. Embora algumas das observações feitas neste exemplo sejam simples, este tipo de verificação ilustra os benefícios da integração de métodos formais no desenvolvimento de sistemas, auxiliando o desenvolvedor nas etapas iniciais do projeto do sistema. Um aspecto interessante da abordagem oferecida por TANGRAM é a possibilidade de verificar detalhes internos do comportamento dos componentes, o que nem sempre é oferecido por abordagens semelhantes. Algumas destas abordagens se abstraem do comportamento dos componentes, considerando-o como uma coisa única, sem particularidades internas. Verificar os detalhes dos componentes pode contribuir muito para a prevenção de erros nas fases seguintes do desenvolvimento do sistema.

O capítulo a seguir apresenta um estudo de caso mais completo sobre a utilização de TANGRAM para a tradução automática de um modelo de sistema mecatrônico, de modo que este modelo pudesse ser verificado em UPPAAL.

CAPÍTULO 6

ESTUDO DE CASO

Este capítulo apresenta um estudo de caso da utilização de TANGRAM para verificar um sistema mecatrônico baseado em componentes segundo a proposta deste trabalho. Os principais objetivos deste estudo foram: (i) avaliar de forma concreta os benefícios da ferramenta TANGRAM sobre um projeto real de sistema mecatrônico; (ii) observar o impacto das funcionalidades do *middleware* sobre o espaço de estados do modelo gerado; e (iii) identificar possíveis correções e melhorias para as versões futuras da ferramenta.

Para atingir estes objetivos, o estudo foi realizado sobre um sistema automático de portas deslizantes utilizado em estações de metrô. Este sistema serve para aumentar a segurança dos passageiros presentes na plataforma e para restringir o acesso aos túneis e vias somente para pessoas autorizadas.

A seção a seguir apresenta o problema de forma mais detalhada, com seus requisitos funcionais associados. Em seguida, são apresentados os modelos construídos em UML para este sistema, a verificação feita após a tradução deste modelos e, por fim, os resultados obtidos.

6.1 DESCRIÇÃO DO PROBLEMA

O sistema considerado neste estudo de caso é o Sistema de Portas de Plataforma ou *Platform Screen Doors* (PSD). Este sistema consiste basicamente em um conjunto de portas deslizantes automáticas posicionadas entre o trem e a plataforma de embarque. No momento em que o trem chega na estação, as portas do sistema PSD devem ser abertas em sincronia com as portas do trem, permitindo o embarque e desembarque dos

passageiros. O conjunto de portas deste sistema forma uma barreira entre a plataforma e a região das vias e túneis do metrô, o que oferece um maior nível de segurança aos passageiros e também restringe o acesso a estas áreas somente para pessoas autorizadas.

6.1.1 Estrutura do sistema PSD

Esta seção descreve os principais elementos que compõem o sistema PSD e que fazem parte do escopo deste trabalho, o qual se refere ao controle da abertura e fechamento das portas. A Figura 6.1 apresenta a organização destes elementos e a comunicação entre eles. Vale ressaltar que um sistema PSD completo possui diversos outros elementos, os quais, porém, não exercem influência direta sobre o comportamento da abertura e fechamento das portas, logo, não foram considerados neste estudo de caso.

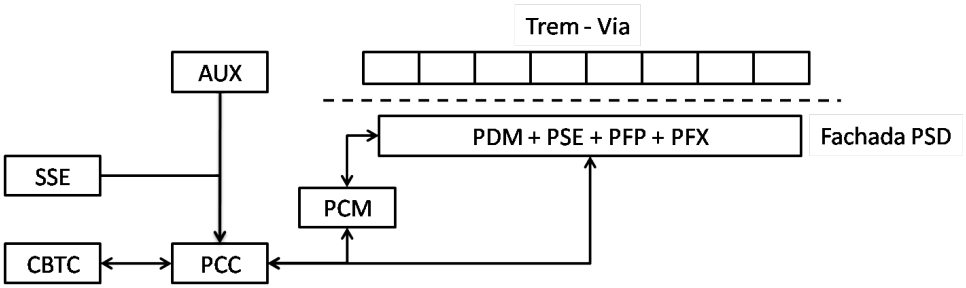

Figura 6.1. Principais elementos que compõem o sistema PSD.

Os itens apresentados na Figura 6.1 estão descritos abaixo:

- PDM - Porta Deslizante Motorizada: cada porta possui duas folhas deslizantes, cujo eixo central deve estar alinhado com o eixo central de cada porta do trem, quando o mesmo estiver posicionado corretamente na plataforma. Além disso, as portas devem possuir um dispositivo de destravamento, acionado por uma chave especial, para uso em caso de emergência. Devem possuir também uma chave comutadora, capaz de isolar a porta em relação ao sistema PSD, em caso de falha da porta;
- PSE - Porta de Saída de Emergência: possui uma única folha deslizante e deve ser instalada ao lado de cada PDM. Deve possuir um dispositivo de destravamento,

acionado por uma chave especial, bem como um dispositivo mecânico de retorno, acionado por uma mola que possibilite o fechamento automático da porta tão logo ela deixe de ser segurada em posição aberta;

- PFP - Porta de Final de Plataforma: esta porta é igual a uma PSE, porém deve ser instalada apenas nas extremidades da fachada do sistema PSD;

- PFX - Painel Fixo: painel fixo que compõe a fachada do sistema PSD e deve ser instalado entre uma porta e outra;

- PCM - Painel de Controle Manual: painel de controle que deve ser instalado nas extremidades da plataforma para permitir que os operadores do metrô possam comandar a abertura e fechamento manual das portas PDM, em caso de emergência ou degradação do sistema. Um PCM deve possuir uma chave "Habilitar PCM", uma indicação "Todas as portas fechadas e travadas", um botão "Abrir todas as portas PDM", um botão "Fechar todas as portas PDM", um botão "Inibir movimentação do trem" e uma chave "Sistema de portas isolado";

- PCC - Painel de Controle Central: este painel controla as funcionalidades do sistema PSD e seus componentes de forma automática. O PCC possui uma unidade de controle e uma interface homem-máquina (IHM) para indicação de estado, alarmes e diagnósticos do sistema PSD;

- SSE - Sistema de Suprimento de Energia: compreende todos os componentes necessários ao suprimento de energia para os sistema PSD. Deve enviar informações ao PCC indicando se o suprimento de energia está funcionando normalmente ou se o sistema de alimentação de emergência (gerador) está em uso;

- CBTC - *Communication Based Train Control*: sistema de controle geral do metrô, o qual envia comandos para abertura e fechamento das portas PDM, inibição de abertura de portas e monitora todo o sistema PSD. Se o PCM estiver ativado, os comandos do CBTC são ignorados;

- AUX - Módulo Auxiliar: módulo que detecta o correto posicionamento do trem através de sensores de posição. Deve ser utilizado em caso de falha de comunicação com o CBTC para permitir a abertura e fechamento das portas PDM de forma automática. Os comandos oriundos do CBTC têm maior prioridade do que os do módulo AUX.

6.1.2 Requisitos funcionais do sistema PSD

Os seguintes requisitos fazem parte do sistema PSD:

- operação das portas PDM em três modos distintos: automático (CBTC/AUX), remoto-PCM e local;
- abertura e fechamento automático das portas PDM, quando operando através do sistema CBTC ou módulo AUX;
- abertura e fechamento manuais das portas PDM, quando operando através do painel PCM;
- abertura e fechamento manuais das portas PDM, PSE e PFP, quando operadas através de chaves especiais instaladas na plataforma;
- seleção de modo de operação Normal/Isolado, para cada porta PDM, acionado por uma chave comutadora especial, para manobras de manutenção ou operação degradada;
- controle e sinalização individualizados por porta PDM;
- receber e processar as informações de inibição de abertura individual de porta PDM, provenientes do sistema CBTC;
- quando o sistema estiver operando no modo automático, a abertura das portas PDM somente deve ocorrer quando um comando de abertura proveniente do sistema CBTC ou módulo AUX for recebido;

- quando o sistema estiver operando no modo automático, o fechameneto das portas PDM somente deve ocorrer quando um comando de fechamento proveniente do sistema CBTC ou módulo AUX for recebido;
- uma porta PDM deverá iniciar automaticamente o processo de fechamento após transcorrido um determinado tempo a partir do momento em que a mesma tenha sido aberta localmente e o sistema PSD esteja operando em modo automático;
- envio da indicação "Portas fechadas e travadas" para o sistema PSD e para o módulo AUX, quando todas as portas estiverem realmente fechadas e travadas;
- envio da indicação "Inibe movimento de Trem" para o sistema CBTC, quando o respectivo botão no painel PCM estiver ativado;
- envio da indicação "Sistema de portas isolado" para o CBTC, quando esta respectiva função estiver ativada pela chave comutadora;
- detecção automática do uso do sistema de alimentação de emergência (gerador), e alteração do processo de acionamento dos motores das portas PDM para que ocorram de forma escalonada, eliminando os picos de corrente causados pelo acionamento simultâneo dos motores;
- na ausência de alimentação elétrica normal e de emergência, a abertura ou fechamento das portas PDM somente poderá ser efetuado manualmente através das chaves especiais na plataforma;
- um painel PCM somente poderá ser habilitado quando a sua respectiva chave de ativação estiver ativada;
- desconsiderar os comandos de abertura e fechamento recebidos do sistema CBTC quando o painel PCM estive habilitado para operação;
- deve existir um botão "Sistema de Portas Isolado", do tipo contato momentâneo, que possibilite ao operador permitir a partida de um trem parado na plataforma

quando ocorrer algum tipo de falha que acarrete na impossibilidade da ativação do sinal "Todas as Portas Fechadas e Travadas", independente do estados das portas PDM, PSE e PFP;

- a função "Sistema de Portas Isolado" somente deverá ser ativada quando o painel PCM estiver ativado e o trem estiver parado na plataforma;
- a função "Sistema de Portas Isolado" deverá ser automaticamente desativada quando o trem deixar a plataforma, mesmo que o operador mantenha o seu respectivo botão pressionado;
- o módulo eletrônico de cada porta PDM deverá desconsiderar os comandos recebidos de abertura e fechamento quando o mesmo estiver no modo isolado;
- caso ocorra alguma abertura indevida de qualquer porta da plataforma, uma indicação deve ser enviada ao sistema CBTC para impedir a entrada do trem na região da plataforma.

6.1.3 Modos operacionais

O sistema PSD deve possibilitar a operação das portas PDM nos modos automático (CBTC/AUX), remoto-PCM e local. O modo normal de operação é o automático, no qual os comandos de abertura e fechamento são enviados pelo sistema CBTC ou, em caso de falha ou ausência deste, enviados pelo módulo auxiliar (AUX) de detecção da posição do trem na plataforma.

Em caso de falha ou degradação do modo de operação automático, as portas PDM também poderão ser comandadas remotamente através dos painéis PCM instalados nas extremidades da plataforma. Se por acaso o modo remoto-PCM também falhar, ou se houver falha no sistema de alimentação elétrica, as portas PDM poderão ser abertas ou fechadas localmente, através de chaves especiais.

No caso das portas PSE e PFP, estas somente poderão ser abertas ou fechadas local-

mente, através de chaves especiais.

A seguir serão detalhados os procedimentos de abertura e fechamento das portas PDM no modo automático com o sistema CBTC e com o módulo AUX.

Abertura de portas - CBTC:

- o trem se aproxima do ponto de parada correto, com velocidade abaixo de 3km/h;
- o sistema CBTC envia para o sistema PSD a relação das portas que devem ter sua abertura inibida (opcional);
- o sistema CBTC envia um comando de abertura das portas, tanto para o trem quanto para o sistema PSD;
- as portas do sistema PSD abrem-se de forma sincronizada com as portas do trem, exceto aquelas que estiverem com sua abertura inibida;
- o sistema PSD envia ao sistema CBTC a informação de portas abertas, que bloqueará o movimento do trem.

Fechamento de portas - CBTC:

- expirado o tempo de parada na estação, o sistema CBTC envia um comando de fechamento de portas ao sistema PSD e ao trem;
- o sistema PSD e as portas do trem iniciam sinalização audio visual de fechamento iminente;
- após o tempo de aviso, as portas do trem e as portas PDM iniciam o fechamento de forma sincronizada;
- assim que as portas estiverem fechadas e travadas, o sistema PSD envia a informação de portas fechadas para o sistema CBTC, o que liberará o movimento do trem.

Caso haja uma obstrução na porta PDM, será realizado um reciclo automático de fechamento. Se após três tentativas de fechamento, o obstáculo não for removido, as portas obstruídas deixam de se movimentar, permanecendo entreabertas e aguardando por um novo comando de fechamento, ou por uma intervenção do operador.

Abertura de portas - AUX:

- os sensores de posicionamento detectam que o trem está parado e alinhado dentro da tolerância admitida;
- o operador do trem envia um comando de abertura para o sistema de portas do trem e para o módulo AUX;
- o módulo AUX transmite o comando de abertura para o sistema PSD, que inicia a abertura das portas PDM de forma sincronizada com as portas do trem;
- quando todas as portas PDM estiverem abertas, o módulo AUX envia uma indicação de portas abertas para o sistema de controle do trem, que bloqueará o seu movimento.

Fechamento de portas - AUX:

- expirado o tempo de parada, o operador do trem envia um comando de fechamento para o sistema de portas do trem e para o módulo AUX;
- o módulo AUX transmite o comando para o sistema PSD, que inicia o aviso de fechamento iminente;
- após o tempo de aviso, as portas PDM iniciam o seu fechamento de forma sincronizada com as do trem;
- assim que as portas estiverem fechadas e travadas, o sistema PSD envia a indicação de portas fechadas para o módulo AUX, que libera o movimento do trem;
- o operador do trem inicia a partida da estação.

De forma semelhante ao que ocorre para o fechamento comandado pelo CBTC, caso haja uma obstrução na porta PDM, será realizado um reciclo automático de fechamento. Se após três tentativas de fechamento, o obstáculo não for removido, as portas obstruídas deixam de se movimentar, permanecendo entreabertas e aguardando por um novo comando de fechamento, ou por uma intervenção do operador.

6.2 MODELAGEM

A partir dos requisitos apresentados na seção anterior, foi elaborada uma modelagem em UML para representar o sistema PSD como um sistema baseado em componentes, de acordo com a abordagem proposta por este trabalho. A estrutura e o comportamento do sistema foram especificados através de diagramas de componentes e de máquinas de estados, os quais serão apresentados nesta seção.

Considerando que o contexto deste estudo de caso está relacionado com o controle de abertura e fechamento de portas, o foco da modelagem gira em torno do sistema em execução no Painel de Controle Central (PCC). Entende-se que o sistema de controle do PCC será executado em apenas um *host*, logo, existirá apenas uma instância do *middleware* de componentes, o que configura um cenário ideal para aplicação da proposta deste trabalho. Isto decorre do fato de que TANGRAM ainda não está preparada para suportar cenários de execução com múltiplos *hosts*. A Figura 6.2 apresenta uma visão mais abstrata dos principais componentes relacionados ao PCC e que estão diretamente ligados ao controle de abertura e fechamentos de portas.

Os componentes CBTC, AUX, PCM e Controle de Portas funcionam como meio de acesso para os dispositivos que se comunicam com estes equipamentos. No caso específico do Controle de Portas, foi feita uma abstração sobre a quantidade e tipo de portas existente no sistema. A idéia é simular os estados possíveis que as portas podem assumir e verificar o impacto desta variação de estados sobre a interação com o PCC.

A seguir serão apresentadas as modelagens estrutural e comportamental construídas

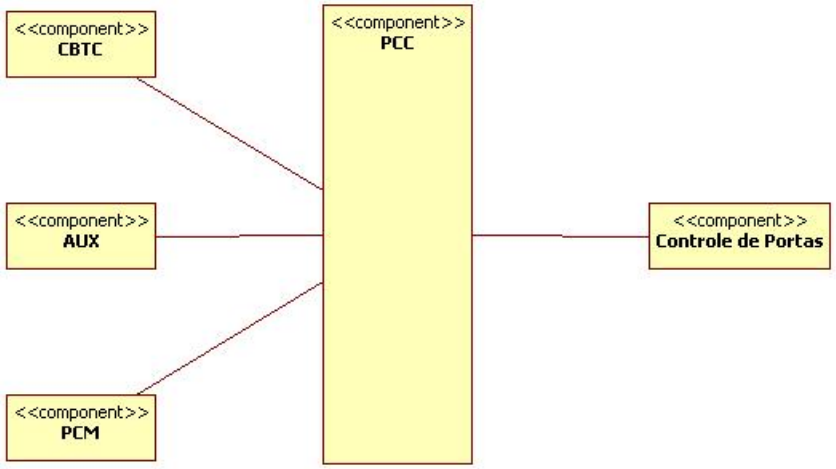

Figura 6.2. Componentes diretamente ligados ao PCC e ao controle de abertura e fechamento de portas.

para o sistema PSD.

6.2.1 Modelagem estrutural

O diagrama de componentes construído para este estudo de caso (Figura 6.3) possui seis componentes, os quais representam os principais elementos que compõem o sistema PSD. O componente `CBTC` tem o papel de realizar a comunicação entre este sistema e o sistema PSD, recebendo os comandos de abertura e fechamento e enviando informações de volta ao CBTC. Este componente foi modelado como um componente ativo, logo, possui um comportamento definido. Este comportamento foi utilizado para simular a chegada e a partida de um trem da estação e disparar as solicitações de abertura e fechamento de portas.

O componente `AUX` é utilizado para realizar a comunicação entre o PCC e o módulo AUX, de forma que seja possível receber os comandos de abertura e fechamento de portas no modo automático, mesmo que o sistema CBTC esteja inativo. Este componente também transmite de volta ao módulo AUX as indicações de "todas as portas abertas" e "todas as portas fechadas". De forma análoga ao componente `CBTC`, o componente `AUX` também é um componente ativo e seu comportamento é utilizado para simular a chegada e

partida de um trem da estação, porém com comandos de abertura e fechamento enviados pelo módulo AUX.

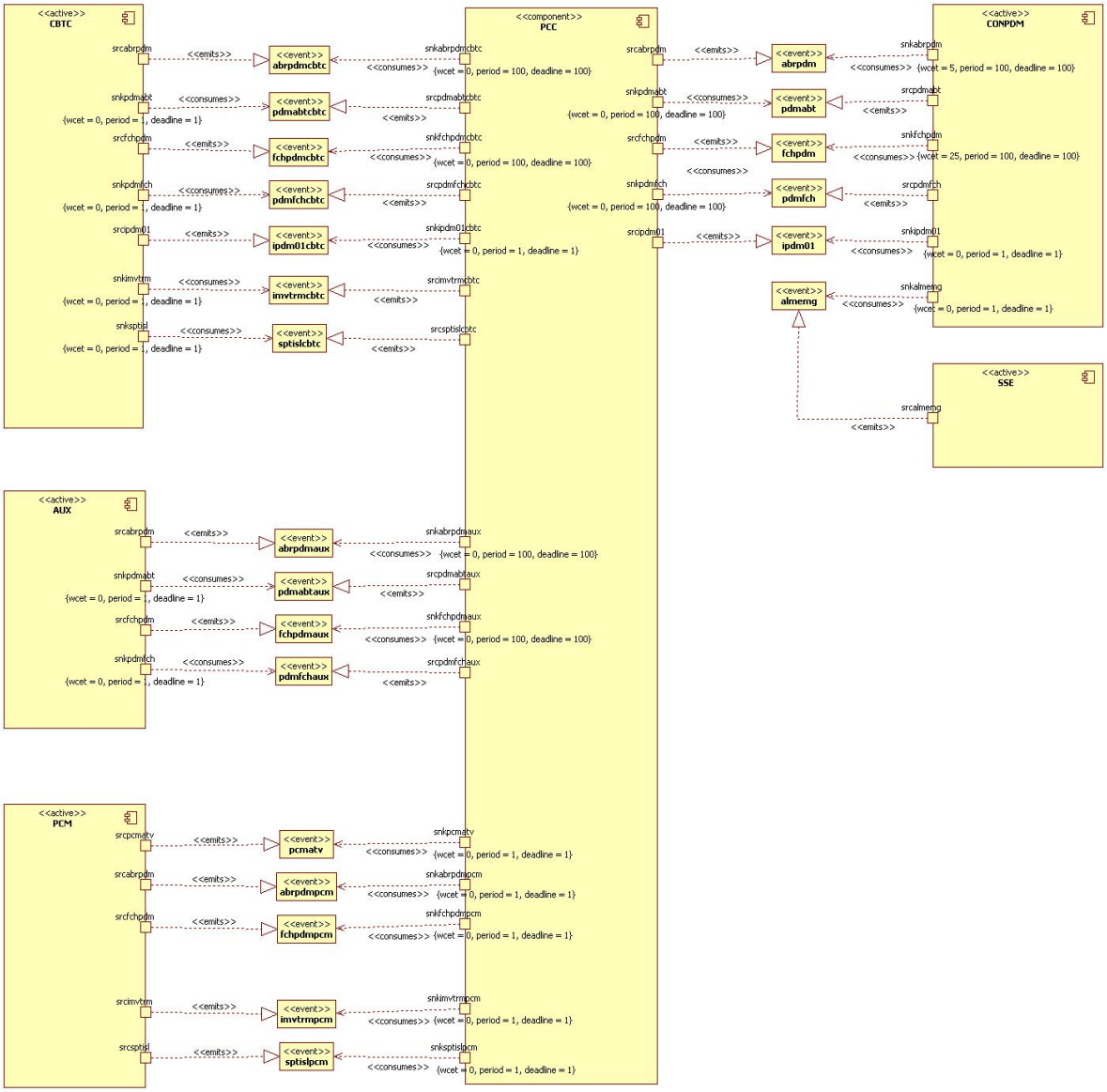

Figura 6.3. Diagrama de componentes para o sistema PSD.

O componente `PCM` representa as funcionalidades do painel PCM, as quais são acionadas por um operador da estação quando há uma falha ou degradação no modo automático. Este componente foi definido como um componente ativo para simular a atuação do operador sobre o painel PCM.

O componente `PCC` funciona como um mediador capaz de receber os comandos externos (CBTC, AUX e PCM) e transmití-los às portas PDM do sistema PSD. Além disso, ele também tem o papel de repassar as informações sobre o estado das portas (abertas ou fechadas) para o meio externo. Este componente também é responsável por gerenciar o modo de operação do sistema PSD (automático-CBTC, automático-AUX ou remoto-PCM).

O componente `CONPDM` desempenha o papel de se comunicar diretamente com os módulos de controle das portas PDM, para os quais são enviados os comandos de abertura, fechamento e inibição de portas, ou são solicitadas as informações sobre o estado das portas. Além disto, este também é um componente ativo e o seu comportamento é utilizado para representar a abertura de portas localmente, através das chaves especiais.

Um dos objetivos em se projetar um componente único para realizar uma abstração sobre o comportamento de todo o conjunto de portas, ao invés de projetar cada porta individualmente, foi evitar um aumento de complexidade desnecessário sobre o modelo, o que poderia levar a uma explosão do espaço de estados nas fases seguintes do estudo de caso, durante a verificação formal. Por exemplo, no contexto deste estudo, é suficiente saber que "*todas as portas estão abertas*", ao invés de "*porta 1 está aberta, porta 2 está aberta, ..., porta n está aberta*". Esse tipo de abstração pode ser feita utilizando um único componente para tratar o controle de portas.

Por fim, o componente `SSE` representa o Sistema de Suprimento de Energia. A sua única função é indicar ao módulo de controle de portas PDM que a alimentação de emergência está sendo utilizada. Desta forma, o controle de portas poderá executar a abertura de portas de forma escalonada, para evitar os picos de consumo de energia.

A partir da construção destes componentes e das características do sistema PSD, o qual se baseia principalmente no envio de comandos e de informações de estado das portas, toda a comunicação do sistema foi feita baseada em eventos. Em outras palavras, isto significa que nenhum dos componentes oferece um *facet* ou possui um *receptacle*. Os eventos oferecidos por cada componente estão descritos nos itens a seguir.

Eventos oferecidos pelo componente `CBTC`:

- `abrpdmcbtc` - comando para abertura das portas PDM;
- `fchpdmcbtc` - comando para fechamento das portas PDM;
- `ipdm01cbtc` - comando para inibir a abertura de uma das portas PDM.

Eventos oferecidos pelo componente `AUX`:

- `abrpdmaux` - comando para abertura das portas PDM;
- `fchpdmaux` - comando para fechamento das portas PDM.

Eventos oferecidos pelo componente `PCM`:

- `pcmatv` - indicação de que o painel PCM foi ativado;
- `abrpdmpcm` - comando para abertura das portas PDM;
- `fchpdmpcm` - comando para fechamento das portas PDM;
- `imvtrmpcm` - comando para inibir a movimentação do trem;
- `sptislpcm` - indicação "Sistema de Portas Isolado".

Eventos oferecidos pelo componente `PCC`:

- `abrpdm` - comando para abertura das portas PDM;
- `fchpdm` - comando para fechamento das portas PDM;
- `ipdm01` - comando para inibir a abertura de uma das portas PDM;
- `pdmabtcbtc` - indicação "Portas Abertas" para o CBTC;
- `pdmfchcbtc` - indicação "Portas Fechadas e Travadas" para o CBTC;

- `imvtrmcbtc` - comando para inibir a movimentação do trem para o CBTC;
- `sptislcbtc` - indicação "Sistema de Portas Isolado" para o CBTC;
- `pdmabtaux` - indicação "Portas Abertas" para o módulo AUX;
- `pdmfchaux` - indicação "Portas Fechadas e Travadas" para o módulo AUX.

Eventos oferecidos pelo componente `CONPDM`:

- `pdmabt` - indicação "Portas Abertas";
- `pdmfch` - indicação "Portas Fechadas e Travadas".

Eventos oferecidos pelo componente `SSE`:

- `almemg` - indicação "alimentação de emergência em uso".

De acordo com a proposta deste trabalho, para cada *event sink* que consome algum dos eventos listados anteriormente, devem ser definidos os seus respectivos atributos temporais associados. Entretanto, os requisitos funcionais apresentados para o sistema PSD não especificam valores de tempo para as funções do sistema, eles apenas indicam a existência de tais valores, por exemplo, tempo de parada na estação, de abertura de porta e de fechamento de porta. Para efeitos de modelagem, os seguintes valores de tempo foram considerados (sem especificar a unidade temporal):

- abertura de portas: 5;
- fechamento de portas: 5;
- parada na estação: 40;
- entre trens: 100.

Vale lembrar que, no pior caso, o tempo de abertura e fechamento das portas deve levar em consideração a possibilidade de execução alternada, devido ao uso de alimentação de emergência. Além disso, o fechamento deve considerar a possibilidade de ocorrerem as três tentativas consecutivas para conclusão da operação.

Neste estudo de caso, não serão explorados mais detalhes sobre os aspectos temporais do sistema PSD, pois o foco aqui não é realizar análise de escalonamento ou dar garantias de correção temporal. Além disto, este sistema pode ser considerado como um sistema de tempo real *soft*, pois os atrasos de suas operações não causarão prejuízos muito graves.

A seguir serão apresentados os aspectos relacionados à modelagem comportamental do sistema PSD.

6.2.2 Modelagem comportamental

Devido à grande quantidade de máquinas de estados construídas para este sistema (28 no total), apenas algumas destas máquinas serão descritas aqui. Mais especificamente, serão apresentadas as máquinas de estados envolvidas no fluxo de execução em modo automático, com CBTC ativo, desde a chegada do trem até a conclusão do procedimento de abertura das portas PDM.

O fluxo de execução tem início na máquina de estados que representa o comportamento ativo do componente `CBTC` (Figura 6.4). Ao ser executada, esta máquina assume o estado "Trem_Chegou". Em seguida, há uma decisão não-determinística entre inibir uma porta ou simplesmente enviar o comando de abertura de portas. Considerando o caso ideal, um evento é enviado através do *event source* `srcabrpdm`, o qual será consumido pelo componente `PCC`. Após enviar o evento, o componente `CBTC` passa para o estado `Envia_Comando_Abertura`, onde ele ficará aguardando por uma confirmação de que as portas foram abertas.

No componente `PCC`, o comando de abertura é consumido pelo *event sink* `snkabrpdmcbtc`, cuja máquina de estados é apresentada na Figura 6.5. Neste comportamento, o PCC veri-

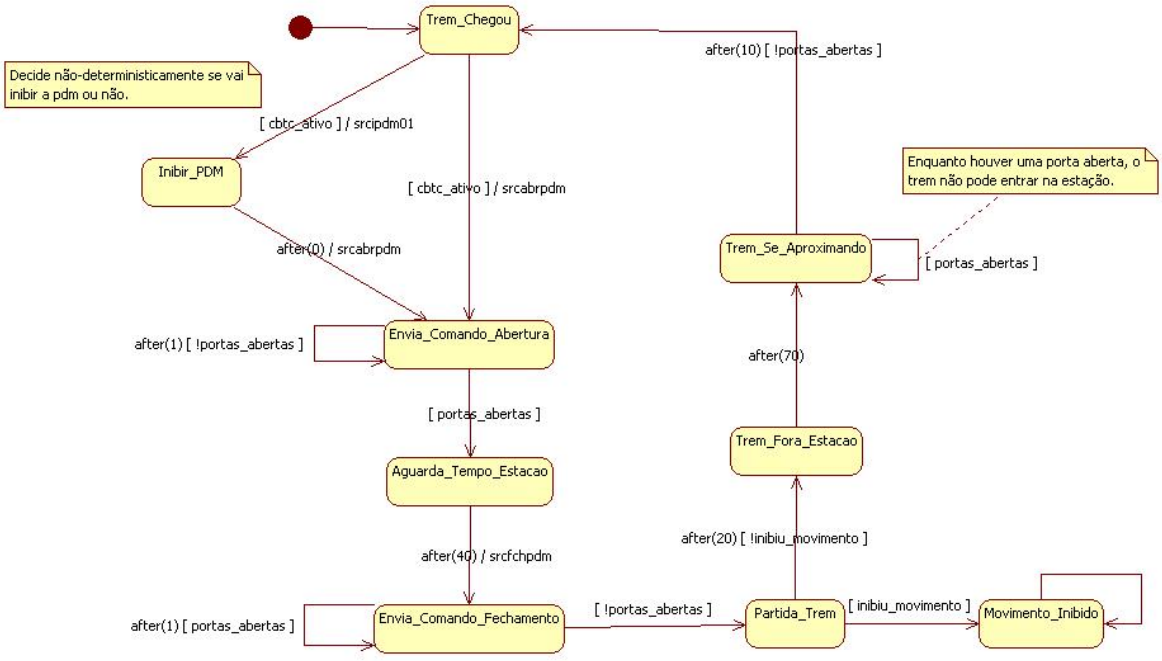

Figura 6.4. Máquina de estados para o comportamento ativo do componente CBTC.

fica se o modo de operação se refere ao CBTC e se o painel PCM não está ativado. Neste caso, um evento é enviado pelo *event source* `srcabrpdm`, o qual será consumido pelo componente `CONPDM`. Após enviar o evento, a máquina de estados encerra a sua execução.

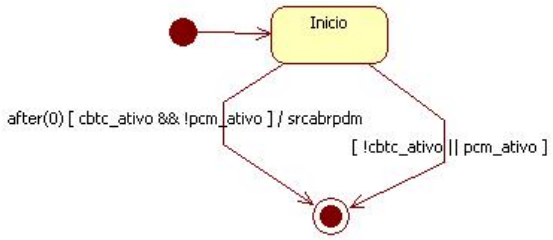

Figura 6.5. Máquina de estados para o *event sink* `snkabrpdmcbtc` do componente PCC.

O componente `CONPDM` recebe o comando de abertura através do *event sink* `snkabrpdm`, cuja máquina de estados é apresentada na Figura 6.6. Ao receber o comando de abertura, o controle de portas deve verificar se o sistema de alimentação de emergência está em uso. Em caso positivo, uma abertura escalonada deverá ser realizada. Tanto na abertura normal, quanto na abertura escalonada, é possível que as portas estejam isoladas ou

em falha. Desta forma, nenhuma confirmação de abertura será devolvida ao sistema e o comportamento será encerrado. Considerando a situação ideal, após o tempo de abertura das portas, um evento de indicação "Portas Abertas" é enviado pelo *event source* `srcpdmabt`, que será consumido pelo componente `PCC`.

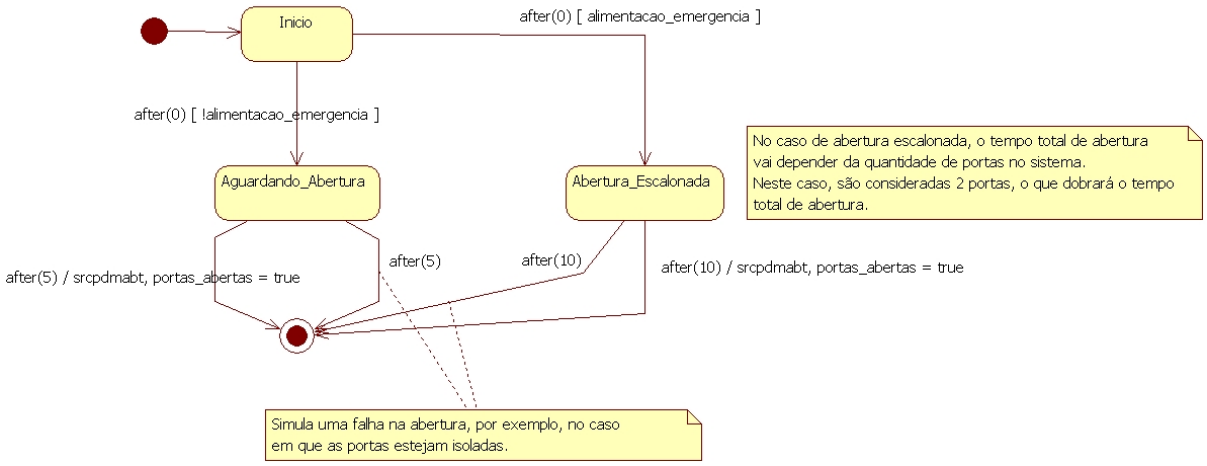

Figura 6.6. Máquina de estados para o *event sink* `snkabrpdm` do componente CONPDM.

Ao receber uma confirmação de abertura pelo *event sink* `snkpdmabt` (Figura 6.7), o componente `PCC` verifica qual deve ser o destinatário desta confirmação, `CBTC` ou `AUX`. Na situação em questão, a confirmação será retransmitida para o componente `CBTC`.

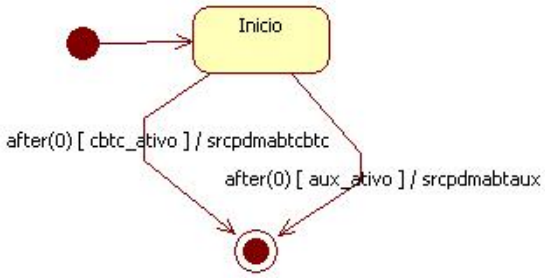

Figura 6.7. Máquina de estados para o *event sink* `snkpdmabt` do componente PCC.

No momento em que o componente `CBTC` receber a confirmação de leitura pelo *event sink* `snkpdmabt` (Figura 6.8), a variável `portas_abertas` deste componente é atualizada para `true`. A partir desta atualização, aquela primeira máquina de estados (Figura 6.4), que estava aguardando uma confirmação de abertura, passa para o estado

`Aguarda_Tempo_Estacao`, que se refere ao estado em que o sistema PSD aguarda o embarque de passageiros, antes de enviar o comando para fechamento das portas.

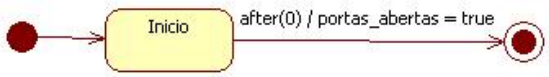

Figura 6.8. Máquina de estados para o *event sink* `snkpdmabt` do componente CBTC.

Após feita a modelagem em UML, os modelos foram traduzidos para autômatos temporizados através da ferramenta desenvolvida neste trabalho, TANGRAM. O papel de TANGRAM neste processo foi gerar modelos formais de autômatos, que representassem o mesmo comportamento especificado pelos diagramas UML originais, de modo que fosse possível verificar formalmente as propriedades do sistema utilizando o verificador de modelos UPPAAL.

Devido ao fato de que a proposta deste trabalho não requer que o desenvolvedor tenha contato direto com os autômatos gerados pela tradução desnecessariamente, a formatação destes autômatos não é gerada de um modo amigável à leitura para o desenvolvedor. Por este motivo, aliado à considerável quantidade de autômatos gerados, os autômatos deste estudo de caso não serão apresentados aqui. Vale ressaltar que nenhuma intervenção manual posterior sobre os autômatos se fez necessária.

Feitas as devidas considerações, a próxima seção apresenta os aspectos relacionados à etapa de verificação de propriedades para o sistema PSD.

6.3 VERIFICAÇÃO DE PROPRIEDADES

A verificação de propriedades para o sistema PSD foi feita considerando diversas configurações possíveis, as quais se diferenciavam pelas seguintes características: (i) ativação do sistema CBTC ou do módulo AUX; (ii) habilitação ou não do painel PCM; (iii) habilitação ou não do sistema de alimentação de emergência; e (iv) possibilidade de ter as portas abertas localmente a qualquer momento, através das chaves especiais, representada

pelo comportamento ativo do componente CONPDM. A partir da combinação entre estas características, a verificação foi dividida em 12 casos diferentes, conforme apresentado na Tabela 6.1.

	CBTC/AUX	**PCM**	**SSE**	**CONPDM**
Caso 1	CBTC	Não	Não	Não
Caso 2	CBTC	Sim	Não	Não
Caso 3	CBTC	Não	Sim	Não
Caso 4	CBTC	Sim	Sim	Não
Caso 5	AUX	Não	Não	Não
Caso 6	AUX	Sim	Não	Não
Caso 7	AUX	Não	Sim	Não
Caso 8	AUX	Sim	Sim	Não
Caso 9	CBTC	Não	Não	Sim
Caso 10	CBTC	Sim	Não	Sim
Caso 11	CBTC	Não	Sim	Sim
Caso 12	CBTC	Sim	Sim	Sim

Tabela 6.1. Variação dos casos de verificação.

Pode-se perceber que não existem configurações que contemplem ao mesmo tempo o módulo AUX e a abertura de portas localmente. Isto decorre do fato de que os requisitos funcionais do Sistema PSD não especificam qualquer interferência da abertura ou fechamento de portas localmente sobre a execução do módulo AUX.

O conjunto de propriedades utilizado para verificar os casos mencionados na Tabela 6.1 é descrito a seguir:

i) `A[] not deadlock` - ausência de *deadlock*;

ii) `E<> CONPDM_portas_abertas == true` - verifica se as portas serão abertas em algum momento;

iii) `E[] CONPDM_portas_abertas == false` - verifica se existe um caminho no qual as portas nunca são abertas;

iv) `E<> CONPDM_pdm01_inibida == true` - verifica se em algum momento a porta poderá ser inibida pelo CBTC;

v) `E[] CONPDM_pdm01_inibida == false` - verifica se é possível a porta nunca ser inibida pelo CBTC;

vi) `E<> CONPDM_alimentacao_emergencia` - verifica se em algum momento o sistema de alimentação de emergência será acionado;

vii) `CBTC_start_proc.Trem_Chegou --> CBTC_start_proc.Partida_Trem` - verifica se o fato do trem ter chegado implica que ele partirá da estação em seguida;

viii) `E<> CBTC_start_proc.Trem_Se_Aproximando && CBTC_portas_abertas` - verifica se em algum momento as portas podem se abrir enquanto o trem está se aproximando da estação;

ix) `E<> CBTC_start_proc.Envia_Comando_Fechamento && !CBTC_portas_abertas` - verifica se em algum momento o sistema fechará as portas após a solicitação pelo CBTC;

x) `E<> CBTC_start_proc.Aguarda_Tempo_Estacao && !CONPDM_portas_abertas` - verifica se durante o embarque de passageiros as portas podem ser fechadas;

xi) `E<> CBTC_start_proc.Partida_Trem && CONPDM_portas_abertas` - verifica se durante a partida do trem as portas podem ser abertas;

xii) `E<> CBTC_start_proc.Movimento_Inibido` - verifica se em algum momento o trem pode ter o seu movimento inibido através do painel PCM;

xiii) `AUX_start_proc.Inicio --> AUX_start_proc.Partida_Trem` - verifica se o fato do trem ter chegado implica que ele partirá da estação em seguida, utilizando o módulo AUX;

xiv) `E<> AUX_start_proc.Envia_Comando_Fechamento && !AUX_portas_abertas` - verifica que em algum momento o sistema fechará as portas após a solicitação pelo módulo AUX;

xv) `E<> AUX_start_proc.Aguarda_Tempo_Estacao && !CONPDM_portas_abertas` - verifica se durante o embarque de passageiros as portas podem ser fechadas, utilizando o módulo AUX;

xvi) `E<> AUX_start_proc.Partida_Trem && CONPDM_portas_abertas` - verifica se durante a partida do trem as portas podem ser abertas, utilizando o módulo AUX.

A partir da interpretação das propriedades e as configurações estabelecidas nos casos de verificação, é possível perceber que algumas propriedades se aplicam mais em alguns casos do que em outros. Por exemplo, a última propriedade da lista anterior terá efeito apenas nos casos em que o módulo AUX estiver ativado, pois envolve o comportamento deste componente. Desta forma, para cada caso de verificação, foram submetidas apenas as propriedades que possuíam uma aplicação significativa sobre aquela configuração do sistema.

Os casos de verificação foram executados em ordem de complexidade final esperada, do mais simples ao mais complexo. Esta complexidade final esperada pode ser observada através da quantidade de componentes envolvidos na configuração do caso de verificação e pelo grau de não-determinismo do comportamento destes componentes. Em geral, os componentes que representam a interação do operador com o sistema, no qual diversos tipos de comandos podem ser submetidos, possuem um maior grau de não-determinismo associado. No contexto deste estudo, as configurações que apresentam esta característica são aquelas que envolvem a execução do painel PCM e as que permitem a abertura ou fechamento de portas localmente. Desta forma, a sequência de verificação foi dada na seguinte ordem dos casos da Tabela 6.1: 1, 5, 3, 7, 2, 6, 4, 8, 9, 11, 10 e 12. Algumas propriedades não foram submetidas à verificação em alguns casos, pois não se adequavam à configuração de componentes daquele caso em questão. A seguir, serão apresentados os resultados obtidos para os casos de verificação, utilizando a seguinte legenda: S - propriedade satisfeita; NS - propriedade não-satisfeita; E - explosão do espaço de estados.

Os detalhes referentes aos resultados da verificação de cada propriedade não serão discutidos aqui, devido à quantidade de itens que foram verificados. Além disto, o objetivo

Propriedades	i	ii	iii	iv	v	vii	viii	ix	x	xi	Verificado	Explodiu
Resultados - Caso 1	S	S	S	S	S	NS	NS	S	NS	NS	100,00%	0,00%

Figura 6.9. Resultado da verificação do caso 1.

Propriedades	i	ii	iii	xiii	xiv	xv	xvi	Verificado	Explodiu
Resultados - Caso 5	S	S	S	NS	S	NS	NS	100,00%	0,00%

Figura 6.10. Resultado da verificação do caso 5.

Propriedades	i	ii	iii	iv	v	vi	vii	viii	ix	x	xi	Verificado	Explodiu
Resultados - Caso 3	S	S	S	S	S	S	NS	NS	S	NS	NS	100,00%	0,00%

Figura 6.11. Resultado da verificação do caso 3.

Propriedades	i	ii	iii	vi	xiii	xiv	xv	xvi	Verificado	Explodiu
Resultados - Caso 7	S	S	S	S	NS	S	NS	NS	100,00%	0,00%

Figura 6.12. Resultado da verificação do caso 7.

Propriedades	i	ii	iii	iv	v	vii	viii	ix	x	xi	xii	Verificado	Explodiu
Resultados - Caso 2	E	S	S	S	S	NS	E	E	E	E	E	45,45%	54,55%

Figura 6.13. Resultado da verificação do caso 2.

Propriedades	i	ii	iii	xiii	xiv	xv	xvi	Verificado	Explodiu
Resultados - Caso 6	E	S	S	NS	S	S	S	85,71%	14,29%

Figura 6.14. Resultado da verificação do caso 6.

Propriedades	i	ii	iii	iv	v	vi	vii	viii	ix	x	xi	xii	Verificado	Explodiu
Resultados - Caso 4	E	S	S	S	S	S	NS	E	E	E	E	E	50,00%	50,00%

Figura 6.15. Resultado da verificação do caso 4.

Propriedades	i	ii	iii	vi	xiii	xiv	xv	xvi	Verificado	Explodiu
Resultados - Caso 8	E	S	S	S	NS	S	S	S	87,50%	12,50%

Figura 6.16. Resultado da verificação do caso 8.

Propriedades	i	ii	iii	iv	v	vii	viii	ix	x	xi	Verificado	Explodiu
Resultados - Caso 9	E	S	S	S	S	NS	E	E	S	E	60,00%	40,00%

Figura 6.17. Resultado da verificação do caso 9.

deste estudo de caso não é provar a correção do modelo elaborado para o sistema PSD, mas avaliar os benefícios da utilização de TANGRAM em um projeto de sistema real, além de observar o impacto das funcionalidade do *middleware* sobre o espaço de estados

Propriedades	i	ii	iii	iv	v	vi	vii	viii	ix	x	xi	Verificado	Explodiu
Resultados - Caso 11	E	S	S	S	S	S	NS	E	E	S	E	63,64%	36,36%

Figura 6.18. Resultado da verificação do caso 11.

Propriedades	i	ii	iii	iv	v	vii	viii	ix	x	xi	xii	Verificado	Explodiu
Resultados - Caso 10	E	S	S	S	S	NS	E	E	S	E	E	54,55%	45,45%

Figura 6.19. Resultado da verificação do caso 10.

Propriedades	i	ii	iii	iv	v	vi	vii	viii	ix	x	xi	xii	Verificado	Explodiu
Resultados - Caso 12	E	S	S	S	S	S	NS	E	E	S	E	E	58,33%	41,67%

Figura 6.20. Resultado da verificação do caso 12.

e também identificar possíveis correções e melhorias para versões futuras da ferramenta.

De um modo geral, os resultados demonstraram que a abordagem proposta é viável, ou seja, foi possível aplicar a verificação de modelos sobre um sistema baseado em componentes, através da tradução realizada pela ferramenta TANGRAM. Pode-se observar que uma parcela das propriedades submetidas à verificação incorreram em explosão do espaço de estados. Nota-se que esta parcela está concentrada sobre as propriedades mais complexas submetidas aos casos de verificação que possuem um maior grau de não-determinismo, conforme mencionado anteriormente. Para se ter uma noção deste grau, a cada transição de estado, o comportamento do componente PCM possui sempre quatro outras possibilidades diferentes de estados a serem tomados. Aliado a isto, o comportamento do modo local de operação pode decidir a qualquer momento por abrir ou fechar uma de suas portas, o que pode causar um sério impacto na verificação do sistema. Desta forma, a quantidade de possibilidades de execuções tende a crescer vertiginosamente quando estes dois componentes são verificados. Diante disto, pode-se concluir que o fator determinante para a explosão do espaço de estados neste estudo foi o alto grau de não-determinismo dos componentes envolvidos na verificação.

Por outro lado, os resultados também demonstram que o verificador de modelos conseguiu encontrar a resposta para propriedades mais simples, mesmo nos piores cenários. Vide, por exemplo, as propriedades ii a vii verificadas para o caso 12. A única propriedade que realmente não pôde ser verificada foi a xii, pois ela depende significativamente do

comportamento do componente PCM. Todas as outras puderam ser verificadas em pelo menos dois casos diferentes. Dadas estas características, pode-se concluir que as funcionalidades do *middleware* não configuram um fator determinante para a explosão do espaço de estados, pois na maioria dos casos a verificação foi bem-sucedida.

6.4 CONSIDERAÇÕES FINAIS

Este capítulo apresentou um estudo de caso realizado sobre a utilização da ferramenta TANGRAM sobre a verificação formal de um sistema mecatrônico baseado em componentes. O sistema utilizado neste estudo foi o *Platform Screen Doors* ou PSD, o qual é um sistema de portas automáticas utilizado em estações de metrô. Foram especificadas e modeladas apenas as funcionalidades relativas ao controle da abertura e fechamento das Portas Deslizantes Motorizadas (PDM), o qual é executado pelo Painel de Controle Central (PCC).

Diversas propriedades relacionadas às funcionalidades do sistema PSD foram verificadas sobre os autômatos gerados pela ferramenta TANGRAM. A verificação foi organizada em 12 configurações diferentes, cada uma delas envolvendo um conjunto distinto de componentes.

A partir dos resultados obtidos neste estudo, foi possível concluir que a abordagem proposta por este trabalho contribui significativamente para a aplicação da verificação de modelos ao contexto dos sistemas baseados em componentes. Um dos pontos positivos desta abordagem é a possibilidade de verificar detalhes sobre o comportamento interno dos componentes, o que muitas vezes é abstraído por outras abordagens semelhantes, tal como em [13].

Além disto, os resultados demonstraram que o fator determinante para a explosão do espaço de estados foi o grau de não-determinismo dos componentes envolvidos na verificação, logo, as funcionalidades do *middleware* não podem ser consideradas como fontes causadoras deste problema. Entretanto, também não foi possível afirmar se estas

funcionalidades têm o poder de reduzir o espaço de estados gerado. Para isto, seria necessário realizar um estudo de caso comparativo minucioso entre a inclusão e a não-inclusão de características do *middleware* no modelo formal.

De modo a contornar o problema da explosão de estados causado pelo não-determinismo, uma solução seria elaborar os modelos visando previamente a verificação de um determinado conjunto de propriedades do sistema. Desta forma, o desenvolvedor pode construir um modelo mais determinístico, considerando aquele conjunto de propriedades selecionada. Isto pode ser feito principalmente quando se está modelando o comportamento da interação entre um operador humano e o sistema, no qual este operador pode submeter diversos comandos de forma aleatória e irrestrita. Neste caso, é recomendado que o desenvolvedor elabore o seu modelo considerando aquele conjunto de comandos submetidos pelo operador que mais se aplique às propriedades que ele deseja verificar. Além disto, quando for possível, é interessante que o desenvolvedor construa uma ordem específica de execução destes comandos, e também restrinja o tempo da execução dos mesmos. Por exemplo, se um dado sistema permitir que o operador acione duas entradas A e B sem restrição de ordem ou tempo, mas para o conjunto de propriedades que se deseja verificar é necessário que a entrada B seja acionada antes de A, com um intervalo de 30 milisegundos, então é recomendado que o desenvolvedor faça a modelagem considerando estas restrições. Certamente, medidas como estas contribuirão para a redução do não-determinismo presente nos modelos gerados, o que resultará em um melhor desempenho na verificação formal.

Durante este estudo de caso, algumas melhorias foram identificadas para a ferramenta TANGRAM. Uma delas é criar uma extensão para o diagrama de máquina de estados, no qual o desenvolvedor possa marcar um estado para que seja verificada a alcançabilidade (*reachability*) do mesmo na etapa de verificação. Para isto, a tradução de TANGRAM poderia identificar os estados marcados com este atributo e gerar automaticamente uma propriedade em TCTL que será verificada em UPPAAL. Este tipo de propriedade é muito comum e gerá-la automaticamente através da ferramenta pode contribuir muito para reduzir a dependência do desenvolvedor sobre a especificação de propriedades em

lógica temporal.

Outra melhoria identificada seria criar um tipo de máquina de estados especial utilizada para representar as interações do ambiente com o sistema, desvinculada de qualquer elemento específico dos componentes (*facets*, *event sinks* ou componentes ativos). Neste estudo de caso, a modelagem do comportamento do ambiente foi feita através das máquinas de estados referentes à função `start` dos componentes ativos, ou seja, o próprio componente estava "simulando" a sua interação com o ambiente. Porém, esta solução não é muito adequada, pois obriga que o desenvolvedor modele comportamentos que não são dos componentes dentro de uma área reservada para o comportamento dos mesmos. Desta forma, ter uma ou mais máquinas de estados destinadas a este fim aumentaria a flexibilidade da ferramenta e deixaria os modelos mais elegantes e concisos.

O capítulo a seguir apresenta as conclusões gerais deste trabalho e aponta uma perspectiva para trabalhos futuros envolvendo a ferramenta TANGRAM.

CAPÍTULO 7

CONCLUSÕES

Este trabalho tratou da aplicação da verificação de modelos ao processo de desenvolvimento de sistemas de tempo real baseados em componentes. A abordagem apresentada aqui propõe uma tradução automática de sistemas baseados em CCM e modelados em UML para autômatos temporizados passíveis de verificação pela ferramenta UPPAAL. Esta tradução levou em consideração as características do *middleware* de componentes nos modelos formais gerados.

As principais contribuições deste trabalho envolvem: (i) a proposta de uma abordagem para especificação de sistemas baseados em CCM/CIAO através de UML; (ii) a elaboração de uma tradução entre diagramas UML e autômatos temporizados do verificador de modelos UPPAAL; (iii) o desenvolvimento da ferramenta TANGRAM, que dá suporte à abordagem proposta; e (iv) a realização de um estudo de caso sobre a aplicação desta abordagem, utilizando TANGRAM, para verificar um sistema mecatrônico real.

Um dos diferenciais deste trabalho em relação a trabalhos relacionados encontrados na literatura está associado à utilização de tecnologias abertas, interoperáveis e independentes de plataforma de execução, como é o caso do CCM e da UML. Outro ponto de destaque é a proposta de uma ferramenta de tradução automática implementada para dar suporte à esta abordagem. Além disto, a ferramenta proposta possibilita a verificação de propriedades relacionadas a detalhes internos do comportamento dos componentes, o que não é comumente encontrado em trabalhos semelhantes. Por fim, pode-se destacar o verificador de modelos utilizado, UPPAAL, o qual já é uma ferramenta bastante utilizada no contexto de sistemas de tempo real.

Os resultados obtidos através das realização do estudo de caso demonstraram a via-

bilidade da aplicação de TANGRAM no suporte à integração entre projeto e validação de sistemas baseados em componentes com restrições temporais. Além disto, os resultados indicaram que as funcionalidades do *middleware* incorporadas aos modelos formais gerados não configuram um fator determinante para a explosão do espaço de estados do sistema. Mas ao invés disto, o grau de não-determinismo dos modelos elaborados pelo desenvolvedor pode sim contribuir para a ocorrência deste problema. No entanto, ainda não foi possível determinar se a inclusão das funcionalidades do *middleware* contribuem para a redução do espaço de estados, o que requer uma análise mais extensa e detalhada.

De um modo geral, embora os resultados apontem para a viabilidade da utilização de TANGRAM em projetos reais de sistemas de tempo real baseados em componentes, entende-se que a sua contribuição é ainda limitada nesta versão, levando em consideração a quantidade de melhorias identificadas durante a realização desta pesquisa. Para que esta ferramenta possa abranger uma maior variedade de aplicações e se torne mais robusta, as seguintes melhorias devem ser realizadas em versões futuras:

- implementar políticas de escalonamento de prioridade variável, tal como a EDF [17];
- permitir que o desenvolvedor configure a granularidade temporal utilizada nos modelos;
- tratar questões como compartilhamento de recursos e dependência cíclica entre componentes;
- dar suporte à passagem de parâmetros e retorno de funções no modelo de tradução;
- estender o diagrama de máquina de estados para permitir a marcação dos estados para os quais deseja-se verificar alcançabilidade;
- permitir a criação de máquinas de estados especiais para modelagem da interação do ambiente com o sistema;

- implementar a leitura dos arquivos UML de acordo com os padrões definidos pela OMG;
- implementar uma interface gráfica para a ferramenta.

Por fim, além das melhorias identificadas acima, uma especificação formal das regras de tradução de forma a se verificar formalmente a consistência do mapeamento seria importante no contexto deste trabalho, mas esta tarefa exigiria um esforço que está além do escopo desta dissertação. Neste trabalho, a validação do modelo de tradução em si pôde ser analisada através dos próprios testes e verificações realizadas sobre os modelos formais gerados por TANGRAM. Além desta análise, também foram feitas simulações em UPPAAL para acompanhar passo-a-passo o comportamento dos autômatos gerados. Desta forma, todos os erros detectados nas análises realizadas foram corrigidos e a ferramenta se encontra atualmente em uma versão estável de utilização.

REFERÊNCIAS

[1] TATIBANA, C. Y. *Garantia Dinâmica de Tempo Real em Sistemas Baseado em Componentes*. Tese (Doutorado) — CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA, UNIVERSIDADE FEDERAL DE SANTA CATARINA, 2007.

[2] CRNKOVIC, I.; LARSSON, M. *Building Reliable Component-Based Software Systems*. [S.l.]: Artech House, 2002.

[3] ANDRADE, S. S. *Sistemas Distribuídos de Supervisão e Controle Baseados em Componentes de Tempo Real*. Dissertação (Mestrado) — Programa de Pós-Graduação em Mecatrônica, Universidade Federal da Bahia, 2006.

[4] AUSLANDER, D. What is mechatronics? *Mechatronics, IEEE/ASME Transactions on*, v. 1, n. 1, p. 5–9, 1996. ISSN 1083-4435.

[5] CRNKOVIC, I. Component-based approach for embedded systems. In: *Proceedings of the 9th Workshop on Component-Oriented Programming*. [S.l.: s.n.], 2004.

[6] SUN MICROSYSTEMS. *Enterprise JavaBeans Specification*. [S.l.]. Disponível em: <http://java.sun.com/products/ejb/docs.html>.

[7] MICROSOFT. *Microsoft COM: Component Object Model Technologies*. [S.l.]. Disponível em: <http://www.microsoft.com/com/default.mspx>.

[8] OMG. *CORBA Component Model*. [S.l.], 2007. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/06-04-01>.

[9] WANG, N. et al. Total quality of service provisioning in middleware and applications. *The Journal of Microprocessors and Microsystems*, v. 2, n. 27, p. 45–54, 2003.

[10] CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 8, n. 2, p. 244–263, 1986. ISSN 0164-0925.

[11] HATCLIFF, J. et al. Cadena: An integrated development, analysis, and verification environment for component-based systems. In: *Proceedings of the 25th International Conference on Software Engineering*. [s.n.], 2003. Disponível em: <citeseer.ist.psu.edu/hatcliff01cadena.html>.

[12] CARLSON, J.; HAKANSSON, J.; PETTERSON, P. Saveccm: An analysable component model for real-time systems. In: *Proceedings of FACS 2005*. [S.l.: s.n.], 2005.

[13] MADL, G.; ABDELWAHED, S.; SCHMIDT, D. C. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Systems*, v. 33, n. 1-3, p. 77–100, 2006.

[14] BEHRMANN, G.; DAVID, A.; LARSEN, K. G. A tutorial on uppaal. In: *Formal Methods for the Design of Real-Time Systems*. [S.l.]: Springer Berlin / Heidelberg, 2004. v. 3185/2004, p. 200–236.

[15] ISO. *ISO/IEC 19501:2005 information technology - open distributed programming - unified modelling language (UML) version 1.4.2*. 2005. Disponível em: <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=32620>.

[16] DEMARTINI, C.; IOSIF, R.; SISTO, R. dspin: A dynamic extension of spin. In: *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. London, UK: Springer-Verlag, 1999. p. 261–276. ISBN 3-540-66499-8.

[17] LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, ACM Press, New York, NY, USA, v. 20, n. 1, p. 46–61, 1973. ISSN 0004-5411.

[18] GU, Z.; SHIN, K. G. Model-checking of component-based event-driven real-time embedded software. In: *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*. [S.l.: s.n.], 2005. p. 410–417.

[19] HSIUNG, P. A. et al. Formal verification of embedded real-time software in component-based application frameworks. In: *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*. [S.l.: s.n.], 2001.

[20] BARROS, T.; CANSADO, A.; MADELAINE, E. Model-checking distributed components: The vercors platform. In: *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software*. [S.l.: s.n.], 2006.

[21] BRUNETON, E. et al. An open component model and its support in java. In: *CBSE*. [S.l.: s.n.], 2004. p. 7–22.

[22] BARROS, T.; BOULIFA, R.; MADELAINE, E. Parameterized models for distributed java objects. In: _____. *FORTE*. [S.l.: s.n.], 2004. p. 43–60.

[23] FERNANDEZ, J.-C. et al. Cadp - a protocol validation and verification toolbox. In: *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1996. p. 437–440. ISBN 3-540-61474-5.

[24] EMERSON, E. A.; LEI, C. L. Efficient model checking in fragments of the propositional mu-calculus. In: *Proceedings of the 1st Symp. on Logic in Computer Science*. [S.l.: s.n.], 1986.

[25] ATTIOGBé, C.; ANDRé, P.; ARDOUREL, G. Checking component composability. *SC 2006*, p. 18–33, 2006.

[26] BOLOGNESI, T.; BRINKSMA, E. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 14, n. 1, p. 25–59, 1987. ISSN 0169-7552.

[27] HANSSON, H. et al. Saveccm - a component model for safety-critical real-time systems. *EUROMICRO Conference*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 627–635, 2004. ISSN 1089-6503.

[28] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. [S.l.]: Addison-Wesley, 1998.

[29] STANKOVIC, J. A. et al. Vest: An aspect-based composition tool for real-time systems. In: *Proceedings of the 9th IEEE Real Time and Embedded Technology and Applications Symposium*. [S.l.: s.n.], 2003.

[30] WANG, S. et al. Real-time component-based systems. In: *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*. [S.l.: s.n.], 2005.

[31] FARINES, J.; FRAGA, J.; OLIVEIRA, R. *Sistemas de Tempo Real*. IME-USP, São Paulo-SP: 12ª Escola de Computação, 2000.

[32] MACEDO, R. et al. Tratando a previsibilidade em sistemas de tempo-real distribuídos: Especificação. linguagens, middleware e mecanismos básicos. In: *Capítulo 3 do Livro texto para o mini-curso apresentado no do 22o. Simpósio Brasileiro de Redes de Computadores*. Gramado, RS: [s.n.], 2004. p. 105–163. ISBN 85-88442-82-5.

[33] LIU, J. W. S. *Real-Time Systems*. [S.l.]: Prentice-Hall, 2000.

[34] SCHMIDT, D. C.; VINOSKI, S. The corba component model: Part 1, evolving towards component middleware. *Dr. Dobb's Portal*, 2004. Disponível em: <http://www.ddj.com/cpp/184403884>.

[35] WANG, N. et al. Configuring real-time aspects in component middleware. In: *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*. [S.l.: s.n.], 2004. p. 1520–1537.

[36] OMG. *CORBA 2.6 Specification*. [S.l.], 2001. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/01-12-35>.

[37] SCHMIDT, D. C.; LEVINE, D. L.; MUNGEE, S. The design of the tao real-time object request broker. *Computer Communications*, Elsevier Science, v. 21, n. 4, 1998.

[38] GILL, C. D.; LEVINE, D. L.; SCHMIDT, D. C. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems*, v. 20, n. 2, p. 117–154, 2001.

[39] HARRISON, T. H.; LEVINE, D. L.; SCHMIDT, D. C. The design and performance of a real-time corba event service. In: *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1997. p. 184–200. ISBN 0-89791-908-4.

[40] OMG. *UML 2.0 Superstructure Specification*. [S.l.], 2005. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/05-07-04>.

[41] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML Guia do Usuário*. [S.l.]: Elsevier Editora LTDA., 2006. ISBN 8535217843.

[42] OMG. *UML Profile for CCM, v 1.0*. [S.l.], 2005. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/05-07-06>.

[43] NATARAJAN, B.; SCHMIDT, D. C.; VINOSKI, S. The corba component model part 4: Implementing components with ccm. *Dr. Dobb's Portal*, 2004. Disponível em: <http://www.ddj.com/cpp/184403884>.

[44] ALUR, R.; COURCOUBETIS, C.; DILL, D. Model-checking for real-time systems. *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, p. 414–425, 1990.

[45] ALUR, R.; DILL, D. L. A theory of timed automata. *Theoretical Computer Science*, v. 126, n. 2, p. 183–235, 1994. Disponível em: <citeseer.ist.psu.edu/alur94theory.html>.

[46] STARUML. *StarUML - The Open Source UML/MDA Platform*. 2005. Disponível em: <http://staruml.sourceforge.net/en/>.

[47] ANDRADE, S. S.; MACEDO, R. J. de A. Engineering components for flexible and interoperable real-time distributed supervision and control systems. *Emerging Technologies & Factory Automation, 2007. ETFA. IEEE Conference on*, p. 376–383, 2007.

[48] ANDRADE, S. S.; MACEDO, R. J. de A. A component-based real-time architecture for distributed supervision and control applications. *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, v. 1, p. 8 pp.–, 2005.

[49] BALASUBRAMANIAN, K. et al. A platform-independent component modeling language for distributed real-time and embedded systems. *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, p. 190–199, 2005. ISSN 1080-1812.

[50] BéRARD, B. et al. *Systems and software verification: model-checking techniques and tools*. [S.l.]: Springer-Verlag New York, Inc., 1999.

[51] CLARKE, E. M. Foundations of software technology and theoretical computer science. In: _____. [S.l.]: Springer Berlin / Heidelberg, 1999. (Lecture Notes in Computer Science, v. 1346/1997), cap. Model checking, p. 54–56.

[52] CRNKOVIC, I. et al. Specification, implementation, and deployment of components. *Comunications of the ACM*, v. 45, n. 10, p. 35–40, October 2002.

[53] DENG, G. et al. Dance: A qos-enabled component deployment and configuration engine. In: *Proceedings of the 3rd Working Conference on Component Deployment*. [S.l.: s.n.], 2005.

[54] DIETHERS, K.; GOLTZ, U.; HUHN, M. Model checking uml statecharts with time. In: *UML'02 Workshop on Critical Systems Development with UML*. [S.l.: s.n.], 2002.

[55] DOUGLASS, B. P. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-49837-5.

[56] GOKHALE, A. et al. Cosmic: An mda generative tool for distributed real-time and embdedded component middleware and applications. In: *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*. [S.l.: s.n.], 2002.

[57] HE, W.; GODDARD, S. Capturing an application's temporal properties with uml for real-time. *hase*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 65, 2000. ISSN 1530-2059.

[58] KANDL, S.; KIRNER, R.; FRASER, G. Verification of platform-independent and platform-specific semantics of dependable embedded systems. In: *Proc. 3rd International Workshop on Dependable Embedded Systems*. [S.l.: s.n.], 2006. p. 21–25.

[59] KICZALES, G. et al. Aspect-oriented programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP '97*. [S.l.: s.n.], 1997.

[60] KNAPP, A.; MERZ, S.; RAUH, C. *Model checking timed UML state machines and collaborations*. 2002. Disponível em: <citeseer.ist.psu.edu/knapp02model.html>.

[61] LATELLA, D.; MAJZIK, I.; MASSINK, M. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal Aspects of Computing*, V11, n. 6, p. 637–664, December 1999. Disponível em: <http://dx.doi.org/10.1007/s001659970003>.

[62] MADL, G.; DUTT, N.; ABDELWAHED, S. A conservative approximation method for the verification of preemptive scheduling using timed automata. In: *15th IEEE Real-Time and Embedded Technology and Applications Symposium*. [S.l.: s.n.], 2009.

[63] MUNIZ, A.; ANDRADE, A. M. S. Mapeamento de diagramas da uml 2.0 em uppaal. In: *Anais da VI Escola Regional de Computação Bahia-Sergipe*. [S.l.: s.n.], 2006.

[64] MUNIZ, A.; ANDRADE, A. M. S. *Uma abordagem para especificação de sistemas de tempo real baseada em métodos formais a partir de diagramas UML*. [S.l.], 2006.

[65] MUNIZ, A. L. N.; ANDRADE, A. M. S.; LIMA, G. An automatic translation approach for component-based real-time systems verification. In: *Proceedings of the Brazilian Symposium on Formal Methods (SBMF) 2008, Student Papers Track*. Salvador/BA, Brasil: [s.n.], 2008.

[66] OMG. *Deployment and Configuration of Component-based Distributed Applications, v4.0*. [S.l.], 2006. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/06-04-02>.

[67] PARIZEK, P.; PLASIL, F.; KOFRON, J. Model checking of software components: Combining java pathfinder and behavior protocol model checker. *sew*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 133–141, 2006. ISSN 1550-6215.

[68] SMITH, M. H.; HOLZMANN, G. J.; ETESSAMI, K. Events and constraints: A graphical editor for capturing logic requirements of programs. *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, IEEE Computer Society, Los Alamitos, CA, USA, 2001.