UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Glademir Maria Silveira Sartori

SUPORTE À GERAÇÃO SEMI-AUTOMATIZADA DE ADAPTAÇÃO PARA COMPONENTES NO AMBIENTE SEA

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. Ricardo Pereira e Silva, Dr.

Florianópolis, Fevereiro de 2005

SUPORTE À GERAÇÃO SEMI-AUTOMATIZADA DE ADAPTAÇÃO PARA COMPONENTES NO AMBIENTE SEA

Glademir Maria Silveira Sartori

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Engenharia de Software e Banco de Dados e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____
Raul Sidnei Wazlawick, Dr.

Banca Examinadora

_____
Ricardo Pereira e Silva, Dr.

_____
Marcello Thiry Comicholi da Costa, Dr.

_____
Patrícia Vilain, Dr.

_____
Frank Augusto Siqueira, Dr.

"A dificuldade traz facilidade e
a facilidade traz dificuldade."
(Tomio Kikushi)

Para André e Altiva.

Agradecimentos

Ao meu orientador, pela compreensão e pelas idéias brilhantes que permitiram este trabalho.

Ao meu amigo Roberto, que em nenhum momento, deixou o desânimo e as atribulações de nossas vidas profissionais afetarem a continuidade deste trabalho.

Aos amigos Richard e Mirela, que das maneiras mais diversas contribuíram em palavras, gestos e ações.

À Bruna, pelos finais de semana sem passeio, sem diversão e sem reclamação.

Ao André, que jamais deixou de acreditar me fazendo ir sempre em frente.

À minha mãe Altiva e ao meu pai Adelino, que com amor e trabalho permitiram que sonhos se realizassem.

E, finalmente, às minhas irmãs, que são exemplos de seres humanos e nas quais me espelho para ser quem sou.

Conteúdo

Lista de Figuras

Lista de Tabelas

Resumo

O Desenvolvimento de Sistemas Baseado em Componentes (DSBC) estabelece que uma aplicação, ou mesmo, outro componente seja construído através da integração de componentes já existentes. Esta visão quebra-cabeça do desenvolvimento de software só pode ser atingida quando tivermos bibliotecas de componentes da onde serão selecionados além de padrões para representar a interface e os mecanismos de conexão. Como usar um componente exatamente como ele é oferecido é muito difícil, o sucesso desta abordagem depende em muito da capacidade de adaptar os componentes existentes para efetivar o reuso.

Este trabalho se concentra na adaptação de componentes, apresentando uma ferramenta semi-automatizada para geração de especificação de interfaces de adaptadores dentro de um ambiente de desenvolvimento, o SEA[1] (SILVA 2000).

Esta ferramenta, a partir de uma dada incompatibilidade, seja estrutural – na assinatura dos serviços, ou comportamental – erros na ordem de chamadas dos serviços, oferece ao desenvolvedor uma lista de possíveis soluções. A partir da escolha da solução desejada a ferramenta cria automaticamente a interface do componente "cola" que resolve a incompatibilidade.Este processo é iterativo.

Após a seleção, a ferramenta atua em conjunto com o Editor Gráfico[2] e com outras unidades de desenvolvimento do ambiente SEA e cria automaticamente um componente "cola" contendo a especificação da interface que resolve a incompatibilidade. O desenvolvedor deverá acessar este componente criado e inserir na sua especificação para que o adaptador resolva a incompatibilidade

Palavras-chave: Componente, Adaptação, Ambiente de Desenvolvimento, Desenvolvimento Baseado em Componentes.

---

[1] SEA é o nome do ambiente desenvolvido por (SILVA 2000) sobre o *framework* OCEAN que suporta desenvolvimento de aplicações baseadas em componentes, dentre outras abordagens e será visto no capítulo sobre o estado da arte.

[2] Editor Gráfico desenvolvido dentro do ambiente SEA para tratar a representação de componentes e acolher as ferramentas de análise (SILVINO 2005) e adaptação desenvolvidas neste trabalho.

Abstract

Component based development establishes that an application is to be built by assemblage of multi-vendors components. This puzzle-like approach can only be achieved when patterns for interfaces and compositions mechanisms are widely used and known. Since is very rare to use a component just like it is, the success of this approach lies on the capacity of adaptation.

This work focuses adaptation, offering a tool that intends to generate interface specifications for adaptors within a specification environment called SEA3 (SILVA 2000).

The presented here, offers a list of possible solutions for a given incompatibility. Both behavior and structural incompatibilities are addressed: the first concerns the syntax of the services' signature, and the latter is about the correct order in which those services should be invoked. From the selection of the best solution of adaptation presented to the developer, the adaptation tool generates the structure and behavior specifications for the component adaptor.

This is an iterative process and this tool works along a components Editor4 and another tool offered by SEA: the tool for incompatibility identification that supplies the input to the construction of adaptors (CUNHA 2005). After the choice of the best option, the adaptation takes place and creates the whole specification for the new component ("glue"). All the results can be visualized and manipulated by the developer, offering a total control over the adaptors and allows trying on different compositions.

Key-words: Componets, Adaptation, Developting Enviroment, Component-Basead Developing Enviroment.

3 SEA is an environment developed by Silva in (SILVA 2000) using a framework, also developed by the author, which supports construction of component-oriented applications, besides others approaches, and will be detailed in the chapter of the state of the art.

4 This Graphic Editor was developed to give a unique and correct representation to components and aggregate both tools (compatibility analyses (CUNHA 2005) and adaptation).

Introdução

O desenvolvimento de software, baseado em componentes (DSBC), define as aplicações como um conjunto de unidades de software individuais e autocontidas, acessíveis apenas por suas interfaces (artefatos caixa-preta), que, a partir de sua interação, constituem uma nova aplicação ou um novo componente. (SAMETINGER, 1997). A idéia principal desta abordagem é o desenvolvimento de novas aplicações, reusando componentes já existentes, ao invés de gerar todo o código do zero. Por isso, esta abordagem pode ser um caminho para a engenharia de software, onde componentes encapsulariam complexidade, e, como peças de um quebra-cabeça, encaixariam-se para realizar novos artefatos, diminuindo o esforço no desenvolvimento. (DONG, 2000).

O caminho para se atingir este objetivo ainda é longo, já que esta abordagem traz uma série de questões em aberto, como, por exemplo, a definição correta da interface, a organização de componentes em espécies de bibliotecas para auxiliar na seleção e a explicitação de formas e mecanismos de integração com outros componentes.

Desenvolver, usando componentes, passa pela seleção correta destes. É necessário poder identificar sua funcionalidade, estrutura e comportamento, a partir da interface. Neste trabalho, funcionalidade é a tarefa que o componente executa, estrutura é a sintaxe para chamar esta funcionalidade, comportamento é a ordem no tempo em que devem ou podem ser chamados os serviços e interface é uma coleção de pontos de acesso a serviços, com cada um deles incluindo uma especificação semântica. (BOSCH, 1997b).

Como o processo de desenvolver utilizando componentes, passa, primeiro, pela oferta destes, tanto o mercado (CCM[5], DCOM™[6], e J2EE™[7]) como o meio acadêmico estão trabalhando no sentido de gerar padrões para a interface de componentes e para os mecanismos de conexão entre eles. (SAMETINGER, 1997).

---

[5] CORBA *Component Model* – modelo da CORBA para componentes.

[6] Java™ 2 Platform, Enterprise Edition (J2EE™) é um *framework* de componentes da Sun™, DCOM™ é o modelo de componentes da Microsoft™.

[7] Java™ 2 *Platform, Enterprise Edition* (J2EE™) é um *framework* de componentes da Sun™.

Mesmo considerando que existam padrões, utilizar componentes como eles foram desenvolvidos, ou seja, “como são”, utilizando a expressão de Bosch (1997a), é muito improvável, já que nem o desenvolvedor do componente pode prever todos os usos, e nem o desenvolvedor de aplicação pode testar todos os componentes, até encontrar um que seja aplicável. Em conseqüência desta afirmação, surge a necessidade da adaptação dos já existentes, primeiramente, na intenção de realizar o reuso e, em segundo, para diminuir o esforço na sua procura, exatamente como os desejados.

Possibilitar a adaptação, não apenas a configuração de componentes, deve ser uma preocupação tanto dos desenvolvedores de componentes quanto de ambientes de desenvolvimento baseados neles. (WELCH, 98).

1.1 Motivação

Segundo Alencar (1998), para atingir reuso efetivo dentro da abordagem baseada em componentes não dependemos apenas da seleção correta de um componente para ser usado, mas, também, de maneiras de combinar e adaptá-los. Assim, integrar componentes juntos numa nova aplicação é a essência desta abordagem. Como, normalmente, estes componentes não foram projetados para serem integrados com quaisquer outros, existe a necessidade de descrevê-los em sua sintática e semântica.

Mesmo após selecionar e integrar estes componentes, isto é, conectar suas interfaces, ainda falta verificar se as estruturas destas combinam, ou seja, se a sintaxe das descrições dos serviços solicitados por um componente é suprida da mesma maneira pelo outro chamado naquela interação, como, por exemplo, os tipos, a ordem e a quantidade de parâmetros (compatibilidade estrutural). Ainda é preciso verificar se o comportamento dos componentes, quando conectados juntos, é correto como no caso de, em algum momento da interação, os componentes ficarem em bloqueio e nada mais funcionar na aplicação (compatibilidade comportamental).

Em Welch (1998), Beach (1992), Cunha (2005), Sametinger (1997), Silva (2000) e em muitos outros trabalhos, verifica-se a intenção de formalizar não apenas a descrição das interfaces, mas mecanismos de verificação destas compatibilidades.

Um passo seguinte, ao encontrar uma incompatibilidade, seja estrutural, funcional ou comportamental, é a geração de adaptadores. Isto é proposto em vários

trabalhos, como alguns dos citados acima, e em Yellin (1994), Inverardi (2002) e Truyen (2000). Cada um deles apresenta uma solução parcial para os problemas da adaptação de componentes, utilizando modelos formais diversos. Algumas delas apresentam ambientes, porém, trabalham apenas com tipos específicos de componentes e com arquiteturas definidas, como será mostrado no capítulo sobre o estado da arte.

Como a adaptação é uma necessidade real e como uma ferramenta semi-automatizada, que colaborasse nesta tarefa, pode auxiliar o DSBC, considera-se o estudo de técnicas e formas de adaptação de componentes e a conseqüente implementação de uma ferramenta integrada com um ambiente de desenvolvimento, como uma contribuição relevante para a disseminação desta abordagem.

1.2 Objetivos

1.2.1 Geral

Definir uma estratégia semi-automatizável de adaptação de componentes, que trate incompatibilidades estruturais e comportamentais, e implementá-la em uma ferramenta a ser integrada ao ambiente SEA.

1.2.2 Específicos

Levantar informações sobre o DSBC e sua situação, tanto na prática do mercado atual, como nas pesquisas acadêmicas, para justificar e embasar o desenvolvimento de uma ferramenta automatizada dentro de um ambiente de desenvolvimento de sistemas, que suporte a adaptação de componentes.

Provar que a adaptação de componentes é uma etapa relevante para o efetivo reuso de componentes no DSBC, e desenvolver uma ferramenta que auxilie nesta tarefa.

Viabilizar a integração de uma ferramenta automatizada dentro de um ambiente de desenvolvimento como o SEA[8] (SILVA, 2000), em conjunto com uma ferramenta de análise de compatibilidade (CUNHA, 2005).

[8] SEA é o nome do ambiente desenvolvido por (SILVA 2000), que suporta desenvolvimento de aplicações baseadas em componentes, dentre outras abordagens, e será melhor visto no capítulo sobre o estado da arte.

Desenvolver, dentro do ambiente SEA, um Editor Gráfico, onde o usuário possa representar seus componentes e fazer a conexão entre suas portas, para que a ferramenta de adaptação possa, a partir de uma dada incompatibilidade, sugerir adaptações, que, também, seriam manipuladas visualmente.

Apresentar adaptações, como especificação de interfaces com estrutura e comportamento, que resolvam, se possível, as incompatibilidades encontradas.

1.3 Apresentação do Trabalho

No Capítulo 2, são apresentadas descrições de componentes e de DSBC, que serão usados neste trabalho. Também são vistos alguns modelos de componentes, já em uso no mercado, como CCM (CORBA *Component Model*) e J2EE™ (Java™ 2 *Platform Enterprise Edition*). Estes modelos têm o mérito de popularizar o termo componente, levando engenheiros de software e a própria indústria a aumentar a sua oferta e ambientes de suporte ao seu desenvolvimento.

No capítulo 3, vê-se que o aspecto caixa-preta dos componentes, onde nada no componente é acessado, senão pela sua interface, é a chave para a integração dos mesmos. Este aspecto é, ao mesmo tempo, uma vantagem e uma desvantagem do DSBC, já que, por um lado, a complexidade da tarefa é encapsulada dentro do componente e, por outro, esta mesma interface pode levar ao seu uso incorreto, se for mal especificada.

Além dos aspectos relacionados a uma boa documentação da interface, que levariam à escolha correta dos componentes, no capítulo 4, será visto como efetivar o reuso, sabendo que é ingênua a idéia de que componentes podem ser usados exatamente "como são" (BOSCH, 1997a). Surge, assim, a necessidade de adaptar componentes, ou seja, modificar, externamente, o que vai ser reusado, para que atenda aos requisitos do sistema projetado.

Abordagens sobre a adaptação de componentes são apresentadas no capítulo 5, O Estado da Arte, onde adaptações dinâmicas e estáticas são vistas na prática, em alguns exemplos de uso. Nesse capítulo, também é introduzido o ambiente SEA (SILVA, 2000), onde será implementada a solução proposta neste trabalho: apresentar,

de maneira automática, especificações de adaptação para componentes dados como incompatíveis.

No capítulo 6, é apresentado o resultado da pesquisa, englobando a motivação para disponibilizar esta ferramenta de suporte, onde ela se encaixa dentro do ambiente SEA e o que ela melhora em relação ao tratamento dado a componentes neste ambiente. São apresentados, também, os algoritmos que geram os adaptadores, tanto para incompatibilidades estruturais quanto comportamentais. Estes dois tipos de incompatibilidades são tratados separadamente, pois envolvem questões diferentes sobre as interfaces dos componentes: um vê a interface sob a perspectiva da assinatura dos serviços disponibilizados e o outro, sobre o comportamento do conjunto de serviços disponibilizados.

2 *Componentes de Software*

2.1 Definições

O termo componente é muito genérico, sendo utilizado para descrever diferentes conceitos, como subsistemas, DLL, ActiveX, J2EE™, DCOM™ (*Distributed Component Object Model,* da Microsoft™), CORBA (*Common Object Broker Request Architecture*), entre outros. Por isso, seguem, abaixo, as definições que serão utilizadas como base para este trabalho:

> "O que torna alguma coisa um componente não é uma aplicação específica e nem uma tecnologia de implementação específica. Assim, qualquer dispositivo de software pode ser considerado um componente, desde que possua uma interface definida. Esta interface deve ser uma coleção de pontos de acesso a serviços, cada um com uma semântica estabelecida." (BOSCH, 1997b).

Em Sametinger (1997), temos que:

> "Componentes de software reusáveis são artefatos auto-contidos, facilmente identificáveis que descrevem e/ou executam funções específicas e têm interfaces claras, documentação apropriada e uma condição de reuso definida".

Para entender este conceito, alguns esclarecimentos são necessários:

Auto-contido: Característica de o componente ser reusável sem a necessidade de incluir/depender de outros componentes. Caso exista alguma dependência, todo o conjunto deve ser visto como um componente reutilizável e estar explicitado na interface.

Identificado: Componentes devem ser facilmente identificados pelos desenvolvedores e estar contidos em um único lugar, ao invés de espalhados e misturados com outros sistemas de software e documentação, sendo tratados como artefatos de software diferenciados.

Funcionalidade: Componentes têm uma funcionalidade clara e específica.

Interface: Componentes devem ter definições de interface, que indicam como podem ser reusados e conectados a outros componentes, e devem ocultar detalhes que não são necessários para seu reuso.

Documentação: A existência de documentação é indispensável para reuso. Descrições como a interface do componente, como ele pode ser usado, como integrar o componente, o que ele faz etc., são informações essenciais.

A definição anterior é parecida com a dada em Bosch (1997b):

> "Um componente é uma unidade de composição independente que explicita dependências de contexto com interface especificada contratualmente...".

Dentro deste contexto, onde um componente é definido como uma caixa-preta e pode ser acessado apenas através de sua interface, é importante esclarecer que esta é uma coleção de pontos de acesso a serviços, com cada um deles incluindo uma especificação semântica. Isto significa que, além da descrição pura e simples da interface (as assinaturas), informações sobre a ordem de acesso, como pré-condições e contexto, devem estar disponíveis.

Esta especificação semântica precisa ser passada para os desenvolvedores de aplicações, sem trazer dúvidas e ambigüidades. Para isso, formalismos são usados para representar esta semântica. Alguns deles serão vistos neste capítulo.

2.2 Representação Gráfica do Componente

Na figura 2.1, vê-se a representação gráfica de um componente que foi adotada para este trabalho. Esta representação foi desenvolvida, pois, para o SEA, componentes eram representados como estruturas de classes num diagrama de classes. Na figura, o componente está dentro do ambiente de desenvolvimento SEA, onde um Editor[9] de Componentes foi elaborado para atender às necessidades do trabalho aqui descrito. SEA e o Editor serão vistos em detalhes no capítulo 6.

[9] Este editor de componentes foi desenvolvido para atender a necessidade de uma representação única e diferenciada de componentes no ambiente SEA e suportar a conexão entre eles, como será visto no capítulo 6.

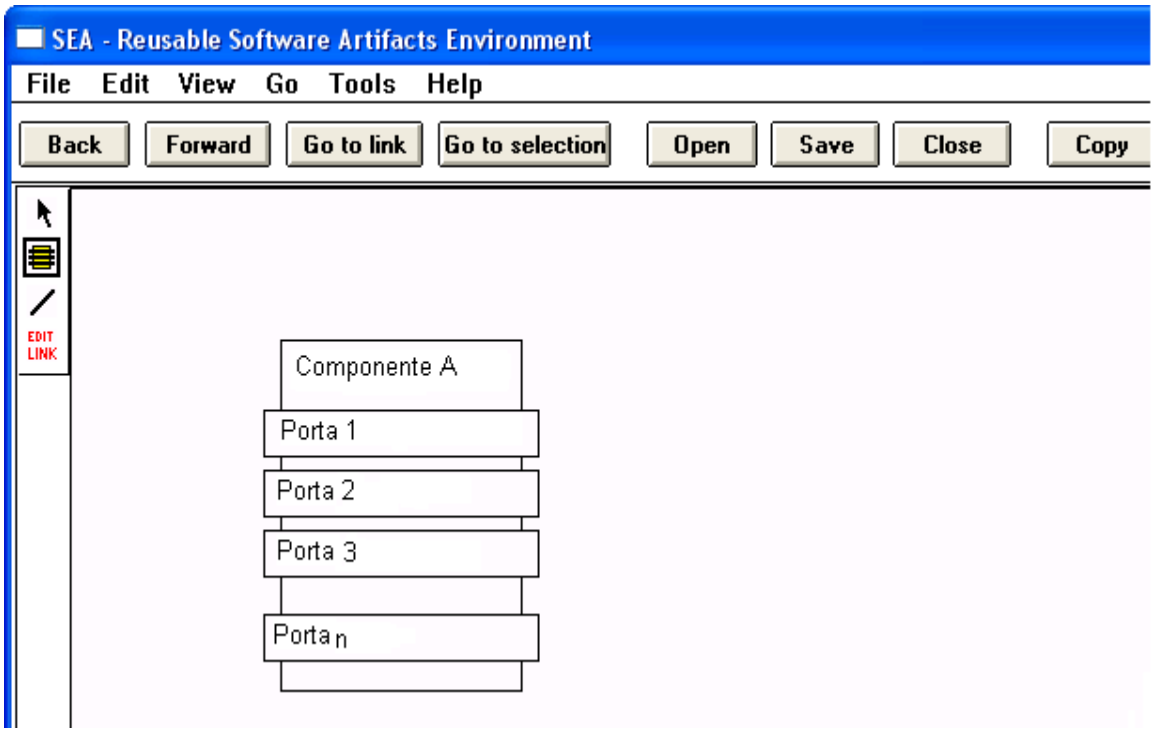

Figura 2.1 Representação de componentes dentro do Editor Gráfico desenvolvido no ambiente SEA.

O componente da figura 2.1 é representado pela sua interface, que consiste em uma agregação dos canais[10] de comunicação (SILVA, 2000), chamados portas.

A interface de um componente poderá ter "n" portas de comunicação e cada uma destas portas pode prover X serviços e necessitar de outros (Y), providos por outro componente a ela conectado. Assim, portas são mais do que entradas de requisições ou invocação de métodos, são pontos de acesso a serviços, que podem necessitar de outros para serem executados, caracterizando a bidirecionalidade de uma porta. (SILVA, 2000).

A conexão entre os componentes se dá através da ligação entre as portas de doisdeles. O tratamento semântico desta conexão será visto na proposta da ferramenta, no capítulo 6. Abaixo, um exemplo de como os componentes podem ser conectados no editor desenvolvido, formando uma nova aplicação ou outro componente.

[10] O que é chamado de canal, nesta referência bibliográfica, pode ser traduzido, aqui diretamente, por porta.

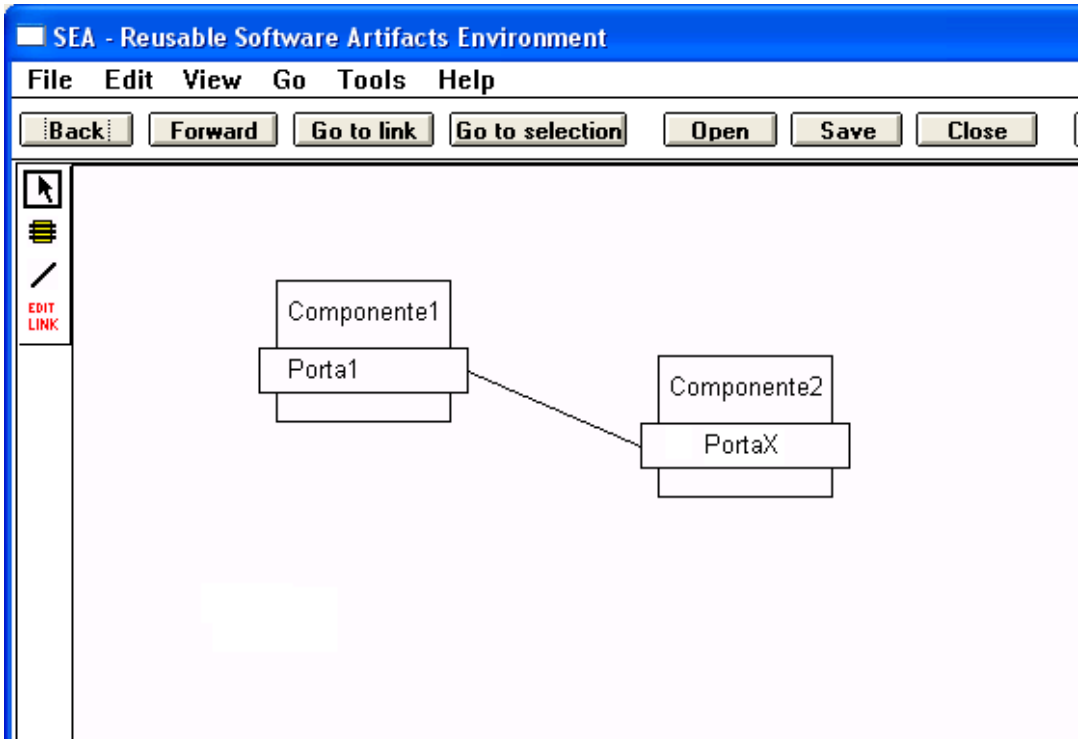

Figura 2.2 Conexão entre componentes dentro do Editor.

Na figura 2.2, os Componentes Componente1 e Componente2 interagem através das portas Componente1.Porta1/Componente2.PortaX. Na representação gráfica, não fica definida a direção nem a ordem da chamada. A conexão é bidirecional. Assim, em cada porta, podemos ter serviços requeridos e supridos. Questões semânticas são resolvidas no nível da especificação do componente (SILVA, 2000; CUNHA, 2005)[11], pois, assim, quando criamos um dentro do ambiente SEA, associamos ele a uma interface, que contém as descrições dos serviços supridos e requeridos ligados com cada uma das portas.

2.3 Interface de Componentes

Um aspecto sobre a especificação das interfaces é que ela engloba um contrato entre os desenvolvedores dos serviços providos e os clientes deste componente. (BOSCH, 1997b). Como é impossível testar um com todos os clientes e vice-versa, é

[11] O Editor foi desenvolvido dentro do ambiente SEA, em parceria com este trabalho, por ser utilizado em ambas as ferramentas, tanto de Compatibilidade (SILVINO, 2005) quando de adaptação, aqui proposta.

preciso verificar, através da interface, os serviços que são providos e solicitados, além de contexto e ordem de chamada, no que diz respeito a inicializações e dependências de sincronismo.

Bosch (1997b) define a interface como "uma coleção de pontos de acesso a serviços, cada um deles inclui uma semântica de especificação", porém, ainda fica a pergunta: -Como fornecer a especificação semântica da interface externa de um componente, de maneira que facilite e assegure seu uso correto na nova aplicação?

Assim, surge a idéia de protocolo, ou seja, as regras que envolvam a comunicação entre estes componentes. Este protocolo deve estar disponível para o usuário, para que ele consiga entender a interface, bem como as restrições de ordem e dependências. Para isso, métodos formais são geralmente utilizados. Em Schneider (1999)[12], é formulada a idéia de protocolos compatíveis, envolvendo a descrição das interfaces dos componentes como mensagens e protocolos, que definem a seqüência legal de mensagens e que podem ser trocadas entre um componente e seu cliente/usuário. Este protocolo é especificado, utilizando uma máquina de estados finitos para determinar em que estado está cada porta e quais as possíveis transições liberadas em cada um.

Formalismos algébricos têm sido usados para descrever protocolos de comunicação de sistemas distribuídos, e são adequados à abordagem de componentes, por permitirem a avaliação de propriedades, como equivalência e inexistência de bloqueio, entre outras, como, por exemplo, *Lambda Calculus*. Porém, é notória a complexidade de descrever interfaces de componentes no mercado, utilizando este modelo formal. Por isso, a pesquisa acadêmica tem direcionado esforços no sentido de aproximar esta pesquisa de algum mecanismo formal, que seja melhor aceito por desenvolvedores em geral. (SILVA, 2000).

Em Dias (2000), usa-se o termo "estado da prática" para definir como os desenvolvedores e o mercado estão trabalhando com o conceito de componentes, sugerindo que a lacuna que existe entre estes dois mundos seja resolvida, buscando maneiras de trazer o estado da arte para mais perto do "estado da prática", no sentido de simplificar os modelos formais apresentados. Assim, o modelo formal utilizado, já que

[12] Protocolo, aqui, é definido como o conjunto de regras de comunicação, envolvendo sintaxe e sincronização.

essencial, deve ser mais simples, fácil de aprender e de representar, para não adicionar ao DSBC uma dificuldade que poderia, em tese, inviabilizar seu uso.

2.4 Abordagens para Definição de Interfaces

2.4.1 Reuse Contract

Reuse Contract (LUCAS, 1997) é um mecanismo de descrição de interface e da interligação destas para um conjunto de componentes participantes de uma colaboração[13]. A notação mostra os participantes que desempenham um papel em um *Reuse Contract*, suas interfaces, suas relações de conhecimento (componentes referenciados) e a estrutura de interação entre os conhecidos. A figura 2.3 apresenta um exemplo deste mecanismo, usando notação gráfica.

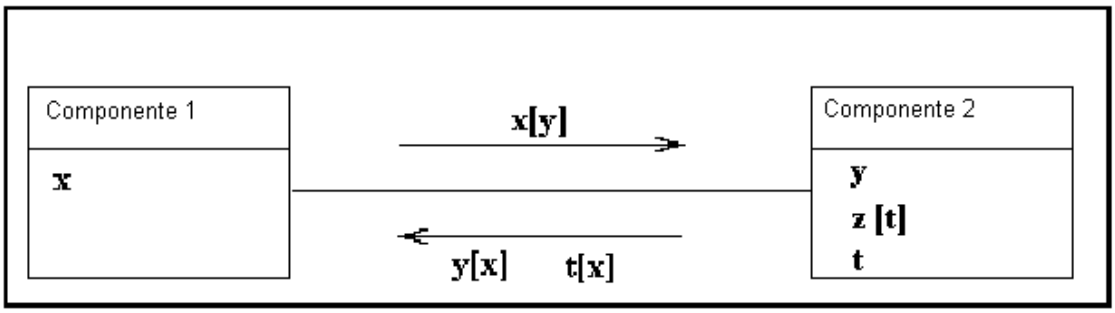

Figura 2.3 Exemplo de Reuse Contract (SILVA, 2002).

No exemplo da figura 2.3, o Componente 1 provê o serviço "x" e o Componente 2 provê os serviços "y", "z" e "t". Em "x", chamando a execução de "y", com y entre colchetes, indica a invocação de "y" por "x", representando qual serviço precisa de outro para executar, e, assim, por diante.

Silva (2002) mostra as deficiências de *Reuse Contracts* para descrever interfaces de maneira eficiente:

> "*Reuse Contracts* descrevem o conhecimento sobre o relacionamento entre componentes e serviços requeridos para rodar um dado serviço suprido. Esta abordagem não é suficiente para descrever a ordem de chamada em uma dada arquitetura."

[13] Colaboração é a interação de um componente com um outro componente.

Assim, vê-se que apenas o uso deste modelo não poderia englobar a necessidade de expressar na interface de componentes a estrutura dos serviços requeridos e supridos, conjuntamente com as restrições de ordem de chamada e comportamento.

2.4.2 IDL – Interface Definition Language (Linguagem de Definição de Interface)

A Linguagem de Definição de Interface (IDL) é usada para expressar os serviços que um componente provê, que são, geralmente, métodos que podem ser chamados por outros componentes. O Object Management Group (OMG – Grupo de Gerenciamento de Objetos) define uma IDL para *Common Object Request Broker Architecture* (CORBA). *Component Object Model* (COM™) define uma IDL para a Microsoft™ e, ainda, a Sun™ define uma IDL para trabalhar com objetos em Java™. Portanto, a IDL se mostra flexível e genérica, podendo ser usada como uma maneira de descrever a interface. Entretanto, ela dá uma descrição limitada das interfaces, apresentando apenas estrutura, ou seja, as assinaturas dos métodos amarradas a cada porta. Sendo assimnão é suficiente para descrever comportamento (SILVA, 2002). Por exemplo, uma IDL não consegue descrever a semântica dos serviços que um componente precisa (serviços requeridos). Desta forma, descrições adicionais sobre os componentes são necessárias. Geralmente, a OCL (*Object Constraint Language* – Linguagem de Restrição de Objetos) é usada para suprir esta lacuna.

2.4.3 OCL – Object Constraints Language (Linguagem de Restrição de Objetos)

Linguagem de Restrição de Objetos (OCL) é uma linguagem de expressão formal, considerada mais fácil de usar do que outras linguagens formais. Geralmente, especifica condições de uma aliança (ou condições), que devem ser mantidas no sistema que está sendo modelado. É usada para especificar restrições em UML (Unified Modeling Language), e utiliza Cálculo de Predicados de Baixa Ordem (LPC – sigla em inglês para Lower Predicate Calculus), somada à teoria de conjuntos. Parte de sua sintaxe tem sido influenciada pela linguagem SMALLTALK.

A IBM propôs a OCL para UML, na versão 1.1, além de disponibilizar um parser (analisador) de OCL, escrito em Java™, gratuitamente, tentando popularizar a linguagem e encorajar o uso de OCL para os desenvolvedores que trabalham com UML no desenvolvimento de componentes.

OCL pode ser usada para vários propósitos, tais como:

- Especificar tipos;
- Especificar invariantes de classes com operações, simplesmente dizendo o que é idêntico;
- Especificar pré-condições e pós-condições sobre métodos – incluindo restrições de ordem e tempo;
- Etc.

O conjunto das duas linguagens (IDL e OCL) poderia ser usado pra especificar interfaces e restrições de interações, porém, um modelo mais simples e unificado seria mais interessante, pois o usuário não precisaria aprender duas maneiras de descrever a interface pra poder entender o componente.

2.4.4 O Modelo de Componentes CCM – CORBA *Component Model*[14]

O modelo CCM (CORBA *Component Model*), parte de CORBA 3.0, introduz uma evolução para IDL, estabelecendo uma interface mais complexa que essa visão, onde aparecem os serviços supridos e requeridos, além dos conceitos de eventos síncronos (SZYPERSKY, 2002), como vemos na figura 2.4:

[14] Uma ampliação do CORBA direcionando para tecnologia de componentes, e resolvendo algumas deficiências do modelo anterior.

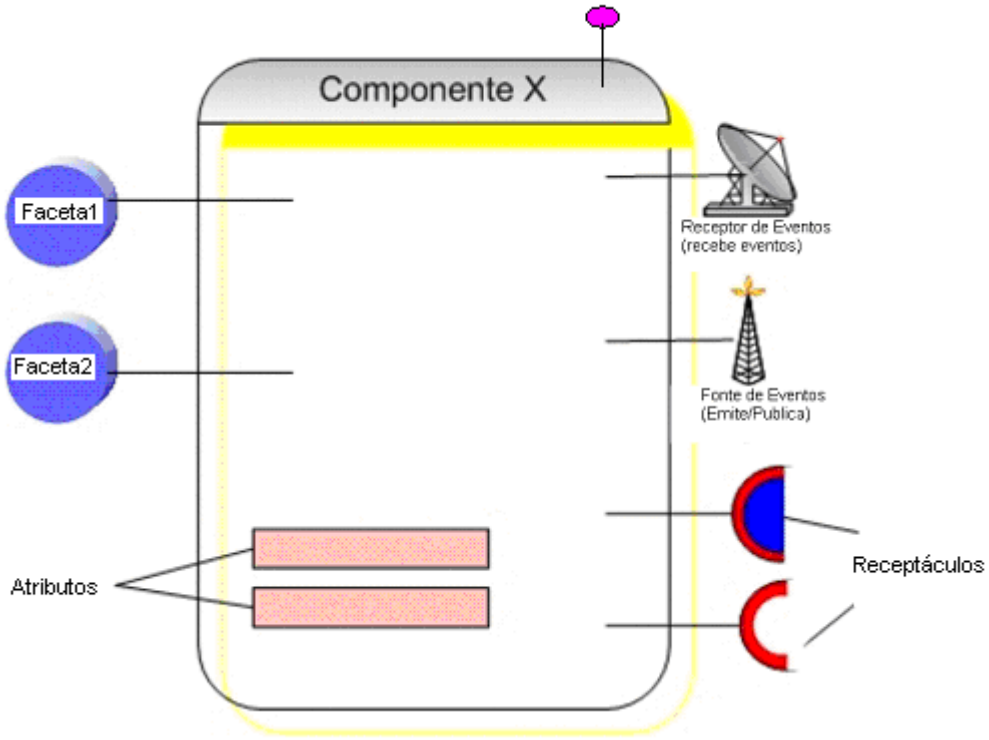

Figura 2.4 Representação de um componente CCM. Baseado em IBM (2004).

Onde:

Faceta1 e Faceta2: São os serviços providos para interação com outros componentes. O modelo não se limita a apenas dois, sendo apenas um exemplo.

Receptáculos: São os pontos de serviços requeridos, fornecidos por um componente externo.

Fonte de Eventos: Ponto de conexão que emite um evento para um ou mais componentes interessados (consumidores), ou para um canal de eventos em um padrão de projeto *Observer* (GAMMA, 1995), onde é definida uma dependência um-para-muitos entre objetos, para, quando um objeto mudar de estado, todos os seus observadores serem notificados e atualizados automaticamente. O acoplamento entre os objetos relacionados é aliviado pelo uso de interfaces abstratas e gerais. Desta maneira, a mesma mensagem é disparada, executando determinado código, dependendo do componente conectado.

Receptor de Eventos: Ponto de conexão, onde eventos gerados fora do componente são recebidos. Aqui, o componente está na outra ponta do padrão de

projeto *Observer*, sendo um observador que recebe uma notificação de outro componente.

A questão de uma definição de um ponto de conexão um-para-muitos é um avanço deste modelo, além da definição de serviços supridos e requeridos em relação à visão anterior, que era apenas de servidor. Entretanto, as questões de ordem e pré-condições ainda não são endereçadas.

2.4.5 Padrão de Projeto para Interfaces

O padrão de interface para componentes, proposto em Silva, 2002, e usado em Silva, 2000, estabelece uma estrutura para a construção destas interfaces, que permite avaliar e assegurar qual compatibilidade comportamental entre os componentes conectados, somadas com a descrição estrutural mostrada na figura 2.5, englobaria a visão estrutural e comportamental:

		serviços requeridos	serviços supridos	
		método requerido 1	**método suprido 1**	**método suprido 2**
Canais	**Canal 1**	X	X	
	Canal 2		X	X

Figura 2.5 Quadro de serviços supridos e requeridos. (SILVA, 2002).

Na figura 2.5, é representado como cada canal[15] se relaciona com os outros, além de suas respectivas assinaturas. (SILVA, 2002).

O padrão de interface de componentes, apresentado em Silva (2002), descreve a interface de um componente englobando a estrutura e o comportamento deste, que é modelado, usando redes de Petri[16], permitindo, assim, a descrição inequívoca da ordem de chamada dos métodos de uma interface, que, segundo Silva, não pode ser endereçada por CCM. Na rede de Petri, cada X (marcado) na tabela de serviços requeridos e supridos da figura 2.5 corresponde a uma ou mais transições na rede, que descreve aquela interface, apresentada, a seguir, na figura 2.6:

---

[15] Porta e canal são similares para este trabalho.

[16] O modelo de rede de Petri foi proposto por Carl Petri para modelar a comunicação entre autômatos, utilizados para representar sistemas.

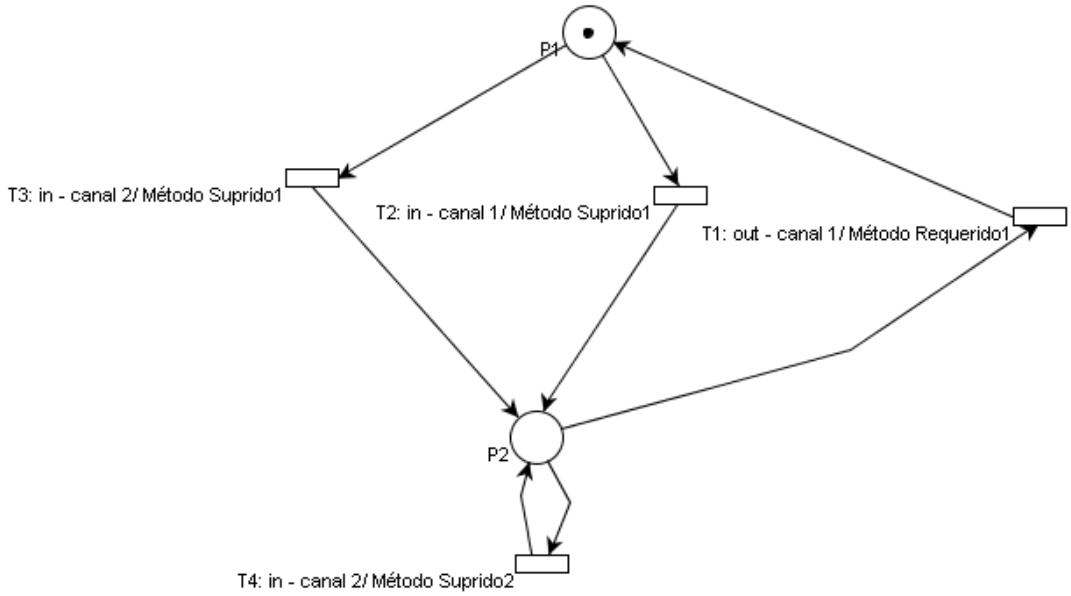

Figura 2.6 Especificação comportamental de um componente utilizando redes de Petri. (SILVA, 2002).

Na figura 2.6, tem-se as transições T1, T2, T3 e T4, que podemos associar ao conceito de chamada de serviços ou métodos. Os lugares P1 e P2 representam os possíveis estados em que o componente pode se encontrar, dado o disparo de uma transição. E, por fim, aparece uma ficha representada por um círculo preto preenchido, indicando a marcação[17] inicial da porta.

Pode-se olhar para a figura 2.6 e verificar como se comporta a interface, quais os métodos supridos e requeridos e, ainda, através da restrição das marcações, visualizar quais as transições habilitadas. A marcação inicial estabelece que apenas as transições T2 e T3 podem ser disparadas, e ambas estão relacionadas como serviços supridos, que podem, segundo esta marcação, ser disparados. Na evolução natural da rede, a ficha seria deslocada para o lugar P2, habilitando as transições T4 e T1. A rede de Petri, desta maneira, especifica restrições relacionadas à ordem de invocação e fornecimento de serviços de um componente, através da ordem estabelecida pela marcação inicial, com a presença ou não de fichas nos lugares. (SILVA, 2002).

Esta abordagem permite descrever a interação entre uma coleção de componentes, utilizando a integração das redes de Petri. A rede de Petri resultante da integração das interfaces conectadas de todos os componentes do sistema permite

[17] Marcação é a disposição das fichas dentro da rede de Petri em um dado momento.

avaliar compatibilidade comportamental (CUNHA, 2005), como, por exemplo, bloqueios, que são verificados, se nenhuma transição estiver habilitada.

Este padrão de interface foi adotado no ambiente SEA (SILVA, 2000), um ambiente que suporta o desenvolvimento e uso de componentes e de arcabouços Orientados a Objetos[18]. O ambiente SEA suporta a especificação de interfaces e a construção automática destas, através da tradução da especificação, em um arcabouço Orientado a Objetos. (SILVA, 2000). O ambiente SEA e o *framework* OCEAN, sobre o qual ele foi construído, serão detalhados no capítulo 5.

2.5 Modelos de Componentes do Mercado

2.5.1 COM™/DCOM□

Javabeans™ da Sun™ e *Component Object Model* (COM™) da Microsoft™ são as duas maiores correntes de especificação de componentes do mercado. A Sun™ e a Microsoft™ têm sua própria definição de um componente:

> "Uma unidade discreta de software que roda dentro de alguns contêineres, expõe um grupo de métodos e propriedades para o seu contêiner e pode mandar e receber eventos para esse contêiner." (SZYPERSKY, 2002).

COM™ é uma linguagem independente, mas sua implementação é, principalmente, integrada com o sistema operacional Windows™. O COM™ distribuído (DCOM™) é a sua extensão, e suporta a comunicação entre componentes distribuídos. Estes objetos podem estar distribuídos sobre diferentes computadores, através da rede, vivendo dentro de seus próprios espaços de crescimento, e, ainda, aparecerem como se estivessem na aplicação local.

O DCOM™ suporta objetos remotos rodando sobre um protocolo chamado *Object Remote Procedure Call* (ORPC). Esta camada interage com os serviços em tempo de execução, direcionando as chamadas dos serviços e entregando para o componente correto, através da rede. (SURESH, 2004).

---

[18] Orientação a Objetos é um paradigma de desenvolvimento de software, que organiza o mundo real como uma coleção de objetos, que incorporam estrutura de dados e comportamento.

Um servidor DCOM™ é o corpo do código, que é capaz de servir objetos de um determinado tipo em tempo de execução. Cada objeto servidor pode suportar múltiplas interfaces, com cada uma representando um comportamento diferente desse objeto. Um cliente usa os serviços exportados por estes servidores, através da aquisição de um ponteiro para uma das interfaces do objeto servidor. (SZYPERSKY, 2002).

Desta maneira, todos os componentes da aplicação devem ser inscritos no contêiner para que os demais componentes do sistema possam ver seus serviços e utilizá-los.

2.5.2 J2EE™

A J2EE™ (*Java Enterprise Edition*) é mais uma tecnologia que proporciona uma plataforma padrão para o desenvolvimento de componentes de servidor Java™ reutilizáveis, executados em um servidor de aplicação.

Esta tecnologia proporciona portabilidade de componentes distribuídos para aplicações empresariais, sendo portáveis e independentes de fornecedor de plataforma, de banco de dados e sistema operacional. É uma plataforma de desenvolvimento para Internet e possui as seguintes camadas, segundo Alur (2002):

Cliente: Interage com o usuário, clientes páginas html, applets java etc.

Web: Lógica de apresentação com servidores, onde roda o contêiner e as aplicações *server-side*.

Negócios: Nesta camada, rodam os componentes de negócio.

EIS (*Enterprise Infomation System*): Responsável pelo sistema de informação da empresa, incluindo banco de dados. É o ponto, no qual as aplicações J2EE™ se integram com os sistemas não J2EE™ e os sistemas legados. (SURESH, 2004).

Os componentes J2EE™ são componentes de granularidade grossa executados em um contêiner, mas, normalmente, em um servidor de aplicação. São de dois tipos: *beans*[19] de sessão e *beans* de entidade, e são específicos de componentes, que executam determinados serviços dentro do *framework*.

Esta plataforma fornece um padrão para construção de componentes de negócio e sessão para aplicações Internet, fornecendo comunicabilidade e persistência, e, nela, o

[19] Bean é nome dado aos componentes deste *framework*.

fornecedor de componentes de aplicação é preenchido pelos desenvolvedores das empresas, utilizando as API's[20] do ambiente.

Em J2EE™, temos um *framework*, que, à exceção de alguns padrões de projetos que estão sendo veiculados pela Internet, não explica nem demonstra como desenvolver ou integrar estes componentes. Fica a cargo dos analistas e especialistas em desenvolvimento montar as aplicações. Recentemente, foi lançado o *framework* BC4J (*Bussiness Components for* Java™), que utiliza padrões de projeto para facilitar e melhor explicar a montagem de tais componentes.

Nas duas abordagens de componentes utilizados no mercado, percebemos duas preocupações: o desenvolvimento para Internet e a necessidade do contêiner para gerenciar a comunicação via rede.

Aspectos direcionados para a tecnologia, ou um tipo específico de componente, não são o foco deste trabalho, mas, sim, aspectos de sua especificação num nível de abstração, onde apenas a interface e os modos de interação são visualizados, independentemente de tecnologia de implementação.

2.6 Etapas do Desenvolvimento de Software Baseado em Componentes

O processo de desenvolvimento de software baseado em componentes pode ser dividido em duas fases: o desenvolvimento do componente e o desenvolvimento da aplicação de software que vai usá-lo. A primeira não está no escopo deste trabalho, mas, sim, as características de sua integração com outros.

O desenvolvimento de aplicações utilizando componentes é realizado através da seleção, ajustamento e integração de componentes, que se adequam aos requisitos da aplicação projetada.

Novos componentes podem também ser necessários, caso nenhum dos existentes atenda às necessidades, ou, ainda, quando o ajustamento de um componente for mais oneroso que desenvolver um novo. Como já foi visto, o sucesso deste processo depende da descrição da interface do componente, pois é através desta descrição que os clientes

[20] API (*Application Program Interface*) é um conjunto normalizado de rotinas e chamadas de software, que podem ser referenciadas por um programa aplicativo para acessar serviços essenciais de uma rede.

dos componentes poderão fazer as escolhas corretamente. Após selecioná-los, o passo seguinte é integrar todos os componentes juntos.

A figura 2.7 mostra o desenvolvimento de uma aplicação baseada em componentes, cuja utilização para a criação de uma nova aplicação passa pelo reuso, adaptação[21] e criação de novos componentes, num processo interativo.

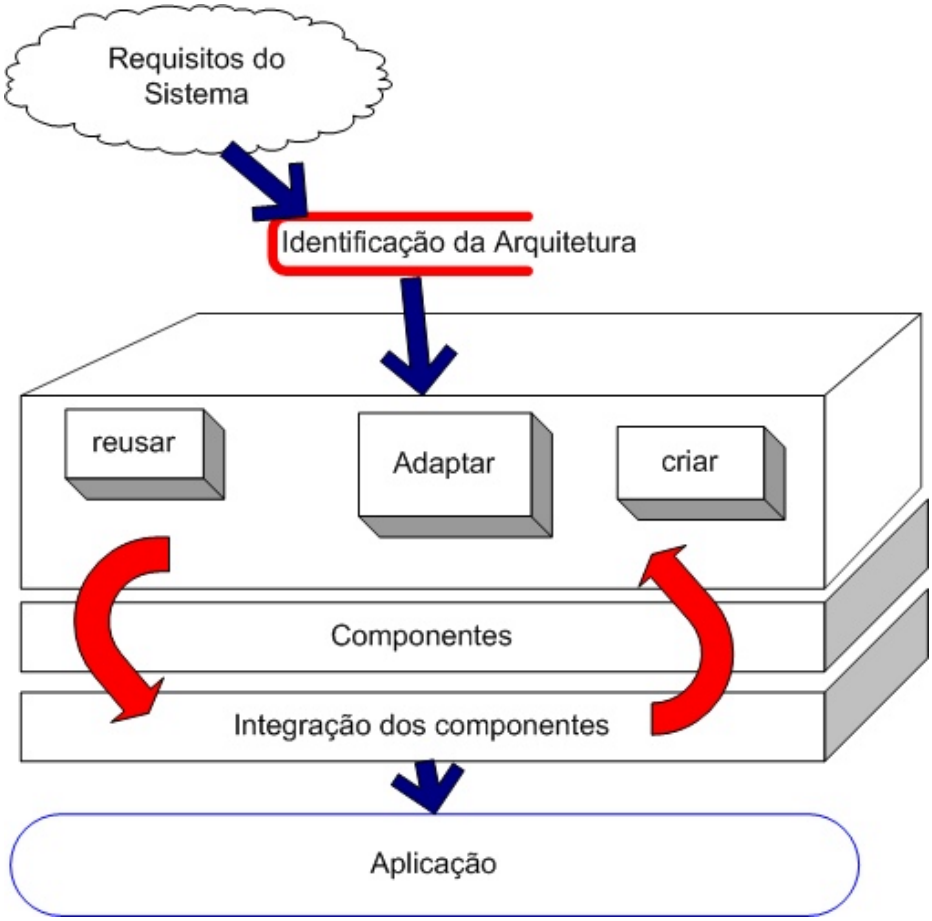

Figura 2.7 Processo do DSBC.

A figura 2.7 ilustra a diferença entre o desenvolvimento baseado em componentes e as abordagens tradicionais, que desenvolvem todo o código do zero. O reuso, utilizando componentes já disponíveis, é o ponto central desta diferença. Após escolhê-los e tentar conectar suas interfaces, o desenvolvedor poderá encontrar obstáculos, como, por exemplo, definições de interfaces não compatíveis, tornando impossível fazer um trabalhar com o outro sem algum tipo de adaptação.

[21] Este termo aparece aqui no lugar de ajustamento e será detalhado no capítulo 4.

Quando o componente não puder ser usado imediatamente como foi fornecido, aparece a etapa de adaptação. O desenvolvedor deveria ser capaz de resolver esta incompatibilidade para poder usá-lo. Este aspecto será visto no capítulo 4.

A dificuldade da seleção, bem como da adaptação, está associada às deficiências apresentadas pelos mecanismos de descrição de componentes em especificar o que fazem e como interagem. Existe a necessidade de ambientes de desenvolvimento baseados em componentes, que facilitem esta seleção e apresentem condições de verificação de incompatibilidade em tempo de projeto, para que o desenvolvedor possa resolver estes problemas. (BOSCH, 1997b ; WELCH, 1999).

A seguir, veremos como integrar os componentes selecionados e questões que envolvem esta etapa do desenvolvimento.

3 *Integração de Componentes*

Os termos integração e composição são usados como sinônimos dentro de DSBC, para definir como os sistemas são montados a partir de componentes. (FELIX, 2002).

A integração para formar uma nova aplicação é a etapa seguinte da seleção dos componentes que são apropriados para o desenvolvimento. A etapa de seleção e os mecanismos de auxílio na sua identificação, para se adequarem às funcionalidades desejadas, não são foco deste trabalho.

As características das tentativas de padronizar métodos de integração serão vistas neste capítulo.

3.1 Linguagens e Modelos de Integração

3.1.1 XML – Extensible Markup Language (Linguagem Estendível de Marcação)

A XML foi introduzida em 1998, (*Extensible Markup Language* – Linguagem Estendível de Marcação). Sua utilidade está diretamente relacionada com a necessidade de padronização relacionada com *e-bussiness* (negócios da internet). Desde que ela surgiu, muitas aplicações estão utilizando XML para representar dados estruturados e realizar a troca de dados entre serviços. Neste sentido, a XML toma a forma de um padrão de conexão num nível de protocolo, a fim de representar dados duráveis que podem ser intercambiáveis entre aplicações, porém, a maioria dos modelos não define formas genéricas para a integração com componentes de outras tecnologias. (FELIX, 2002).

Um exemplo de modelo de integração é a BML (*Bean Markup Language* – Linguagem de Marcação para *Bean*) (IBM, 1998), que é um exemplo de linguagem para integrar componentes *JavaBeans*™, feita pela IBM em 1998. BML é uma linguagem

baseada em XML, própria para sistema *JavaBeans*™, que suporta criação, acesso e configuração de instâncias de *beans*. Apesar do uso de XML, a BML utiliza definições ligadas à linguagem de programação Java™ em que são programados os *beans*.

3.1.2 Definições de Arquitetura para Integração de Componentes

Segundo Shaw (1996), a arquitetura de software surgiu como uma evolução natural das abstrações de projeto, na busca de novas formas de construir sistemas de software maiores e mais complexos e de descrever organizações dos já existentes para promover o reuso de experiência de projeto, possibilitando a escolha da forma de organização adequada ou inerente de um sistema .

A definição feita em Bosch (1997b), de arquitetura para a visão de componentes, é a seguinte:

> "A arquitetura define a composição dos componentes, e os relacionamentos entre eles. Isto requer consideração sobre as interfaces dos componentes. Arquitetura é definida pelas interfaces ao invés de pela implementação dos componentes. Em contraste, como a arquitetura pode ser vista como um padrão de projeto para a composição[22], uma arquitetura concreta pode não ser realizável porque os componentes disponíveis não combinam (erro arquitetural). Por outro lado, em uma dada arquitetura, componentes são substituídos por outros se implementam a mesma interface. Assim, arquitetura representa um projeto durável e de modificação lenta em oposição à implementação do componente. Precisamente a arquitetura consiste numa coleção de interfaces que representam *slots* (encaixes) a serem preenchidos, papéis a serem executados pelos componentes."

Como visto na definição acima, uma abordagem de integração baseada na arquitetura do software impõe sobre ela uma série de regras permitindo a detecção e recuperação de anomalias de integração. A verificação da compatibilidade arquitetural sendo realizada antecipa alguns problemas relacionados com restrição estrutural e/ou comportamental das interfaces dos componentes envolvidos na interação, já que, antes de serem conectados diretamente a semântica da arquitetura, podem antecipar uma conexão impossível ou errada. (INVERARDI, 2002).

---

[22] Composição e integração podem ser vistas como sinônimos neste trabalho.

A noção de arquitetura de software assume um papel chave, uma vez que ela representa o esqueleto sobre o qual são integrados os componentes. No domínio da arquitetura de software, a interação entre os componentes é representada pela noção de conectores de software, canais ou portas. (INVERARDI, 2002). Nesta linha, muitos trabalhos estão sendo feitos, no sentido de utilizar ADL (*Architecture Definition Language)* para capturar a visão arquitetural e integrar suas restrições na integração de componentes.

A utilização de linguagem de descrição de arquiteturas ADL (*Architecture Definition Language*) segue a idéia de programação orientada à conexão para explicitar a composição. A ADL possui componentes e conectores como base para a composição. A composição é expressa como a seleção de componentes e ligação entre eles, através de conectores pré-estabelecidos, descrevendo como as instâncias componentes devem ser "ligadas" umas às outras. (SZYPERSKY, 2002).

A ADL pode configurar toda a conexão dentro de um sistema com componentes, que descrevam sua interface, usando esta linguagem e suas estruturas, conectores, papéis, portas e protocolos.

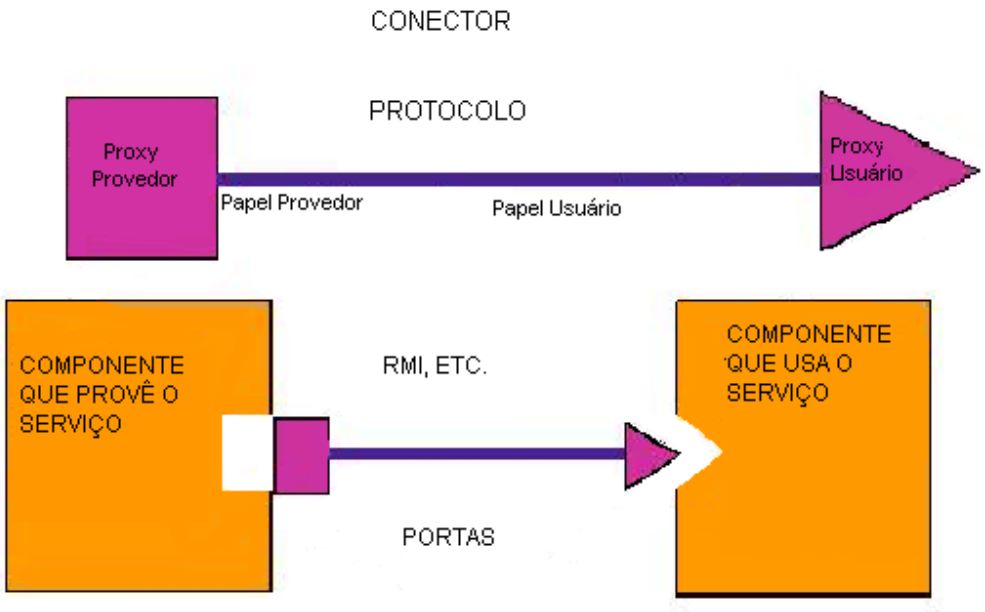

Figura 3.1 Notação da ADL.

Na figura 3.1, vemos as estruturas da ADL para definir conexão entre componentes. Duas estruturas são bem visíveis: os componentes e os conectores. Cada componente contém conectores que podem ser amarrados com outros componentes,

caso esta conexão esteja previamente definida como seu conector possível. Assim, cada componente possui um padrão de conexão que deve ser atendido para poder ser usado. Utilizando estas estruturas, o desenvolvedor pode modelar e explicitar conexões dos componentes no seu ambiente.

Em Dias (2000), temos um exemplo do uso de ADL para criar um ambiente de desenvolvimento e conexão de componentes. O autor acredita que a utilização de ADL pode simplificar a tarefa de criar modelos de conexão (arquitetura de sistemas), para dar suporte a adaptações e prover documentação, porém, os desenvolvedores de componentes e os utilizadores devem conhecer ao modelo para poder executar a integração.

A inclusão de semântica na arquitetura dos sistemas para verificação de características e possíveis adaptações é levada em conta na ferramenta desenvolvida. Entretanto, não há dúvida quanto à importância e a possibilidade de agregação desta característica em trabalhos futuros.

3.1.3 Integração Usando Proxy

Neste modelo de integração, os componentes não ficam sabendo quem está fazendo os serviços e onde o componente servidor está; as requisições de serviço são delegadas a um *proxy* (procurador), que intermedia a comunicação entre eles, funcionando como a cola entre todos os componentes do sistema.

Beach (1992) apresenta uma solução independente de plataforma e de linguagem para realizar a integração de componentes, usando o modelo BART. BART é uma metáfora do *Bus*[23] de hardware, pois serve de canal comum de comunicação entre os componentes do software, assim como o *Bus* de hardware possibilita a comunicação entre seus componentes. As conexões entre estes são feitas através de uma linguagem específica, chamada Linguagem Declarativa de Integração, que define os relacionamentos entre os modelos de dados dos diferentes componentes. Desta maneira, todas as trocas de mensagens são desviadas para o BART, que é um software que gerencia o transporte de mensagens, o compartilhamento de dados e a tradução de dados, operando em um ambiente distribuído.

---

[23] Tradução literal: ônibus, porém, aqui, quer dizer o meio físico de transporte de informação.

Quando um componente precisa de um serviço, ele realiza uma operação equivalente a uma chamada remota, passando argumentos e esperando um valor de retorno. O BART aparece, então, para os componentes, como sendo um banco de dados ativo. Depois que um componente manifesta interesse em um serviço, o BART intercede e vai buscar este serviço em outro conectado a ele, gerenciando seus três níveis: transporte de mensagens, compartilhamento de dados e transformação de dados:

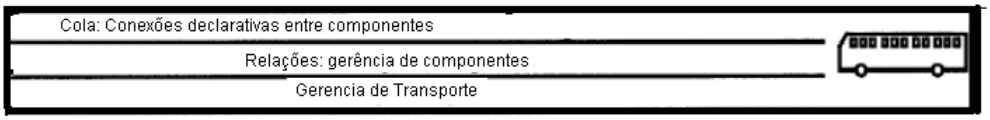

Figura 3.2 Visão da arquitetura BART. (BEACH, 1992).

Na figura 3.2, vemos as camadas tratadas pela arquitetura: onde aparece a declaração explícita da conexão entre os componentes, a relação de gerência e o transporte de mensagens, todos estes aspectos são tratados pelo *Bus* que entrega e gerencia toda a comunicação. Alguns problemas desta abordagem podem ser relacionados à performance. Outro problema é como os componentes devem ser feitos para encaixarem nesta arquitetura específica, pois todos têm que entender e compilar a linguagem de definição pelos autores.

3.2 Considerações

A maioria das abordagens vistas até agora aponta para a integração de componentes, considerando interfaces específicas, isto é, os componentes devem ser expressos de acordo com um tipo de interface que leve em conta certo formalismo, usando ou não conceitos relacionados à arquitetura de sistemas.

A necessidade de padrões para expressar a interface fica evidente, porém, o que pode ainda ser questionado é a exigência de um padrão de arquitetura que deve ser imposto sobre os componentes. Claro que a existência deste padrão tornaria mais simples a substituição de um componente por uma nova versão deste, bastando que a nova versão implementasse a mesma interface, mas nada impede que se desenvolvam

mecanismos de descrição de interfaces e integração independentes do modelo arquitetural.

Quando esta integração se torna impossível, devido a erros não previstos na definição, tanto da arquitetura quanto da própria interface dos componentes, surge a necessidade de adaptar, caso realmente se deseje usar aquele componente, sempre pensando no seu reuso.

A seguir, são apresentadas características e possibilidades desta adaptação.

4 *Adaptação de Componentes*

Componentes encontrados em prateleiras, dificilmente podem ser usados e combinados uns com os outros, dentro de uma aplicação, exatamente "como são". (BOSCH, 1997a). Bosch justifica esta afirmação no fato de que o desenvolvedor de um componente não consegue prever todas as aplicações possíveis e nem todos os ambientes e circunstâncias em que ele pode ser usado. Por isso, considera difícil usar um exatamente como foi projetado, diante das possibilidades de combinações nas aplicações atuais. Além disso, a inexistência de um padrão para modelos de componentes e de técnicas para realizar a integração dentro de sistemas colabora com a afirmação acima.

Dentro desta perspectiva, a adaptação se torna uma etapa fundamental para o desenvolvimento baseado em componentes.

Os termos adaptação, conformação e evolução são, muitas vezes, usados como sinônimos. A primeira vista, pode parecer que querem dizer a mesma coisa, porém, do ponto de vista de componentes, a conformação é a seleção dentre alternativas de uso de uma lista de opções, já previstas no desenvolvimento do componente, caracterizando uma adaptação caixa-preta. A evolução é a melhoria do componente em vários aspectos (estrutural, comportamental e/ou funcional), geralmente, feita por quem o projetou (numa visão caixa-branca do componente). Finalmente, a adaptação é uma extensão, não prevista no desenvolvimento, e é realizada por quem está utilizando o componente para fazer outro aplicativo.

Assim, evolução e adaptação são maneiras de estender os componentes, porém, realizados por atores diferentes dentro do processo de DSBC. A primeira é feita por quem desenvolve o componente e a segunda por quem o usa, a fim de desenvolver outro aplicativo. O foco deste trabalho é o uso do componente para fazer novos artefatos de software, por isso, sua adaptação será abordada do ponto de vista de quem o usa, isto é, a partir de uma abordagem caixa-preta.

O trabalho da adaptação está relacionado com o encontro de incompatibilidade estrutural, comportamental e funcional. A existência de padrões e formalismos na definição da interface pode auxiliar na construção de ferramentas automatizadas, que propiciam a verificação destas incompatibilidades, e de gerações de adaptadores, que as resolvam. (BOSCH, 1997b; WELCH, 1999).

A incompatibilidade estrutural ocorre, quando a sintaxe da chamada aos serviços do componente, como, por exemplo, o nome do serviço, a ordem dos parâmetros, o tipo de cada parâmetro (inteiro, caractere, decimal com duas casas etc.), o tamanho e tipo do retorno etc., não é idêntica à assinatura do serviço suprido. Estrutura está relacionada, neste trabalho, a toda e qualquer incompatibilidade relacionada apenas à assinatura.

A adaptação comportamental está relacionada com a incompatibilidade na ordem de chamada dos serviços e possíveis anomalias durante a execução do sistema, como, por exemplo, bloqueios, quando determinado serviço de um componente aguarda a execução de um outro que ainda não foi e pode ser executado.

A ocorrência de bloqueios pode estar associada à ordem incorreta com que o desenvolvedor conectou as portas dos componentes. Por exemplo, o desenvolvedor solicitou que um determinado componente executasse um serviço, porém, jamais solicitou que o mesmo fosse inicializado, sendo, deste modo, incapaz de executar qualquer coisa.

Adaptação funcional é a substituição de funcionalidade provida pelo componente ou inclusão de funcionalidade não provida pelo componente (nos dois casos, realizados “fora” do componente em questão). A adaptação funcional não será alvo deste trabalho.

Aparentemente, a incompatibilidade comportamental parece mais difícil de ser detectada e resolvida. Entretanto, ferramentas de análise comportamental como a proposta em Cunha (2005) e Welch (1998), podem identificar estas anomalias analisando a especificação formal das interfaces.

Incompatibilidade funcional está relacionada com a tarefa executada pelo componente, ou seja, a sua essência, e deve ser detectada pelo desenvolvedor. Dificilmente, uma ferramenta automatizada pode assegurar e testar a funcionalidade de um componente, pois necessitaria de uma maneira formal para expressá-la, como detalhes da sua computação interna, que podem explodir em complexidade. Cabe ao

desenvolvedor verificar se a sugestão da adaptação realmente é válida para a tarefa a ser executada. Por exemplo, caso um serviço de um componente A seja chamado de dispara_foguete (int, quando; char*, pra_onde) e for conectado a um componente B pela chamada dispara_nave_espacial (int, quando; char*, pra_onde), o analisador de compatibilidade pode dizer que a interface tem um erro estrutural, e a ferramenta de adaptação substituirá o nome dispara_foguete por dispara_nave_espacial, na assinatura da chamada, resolvendo a incompatibilidade estrutural. Porém, a funcionalidade esperada pode apresentar anomalias, pois podem ser bem diferentes (por exemplo, um componente lança fogos de artifício e outro dispara naves espaciais).

No próximo capítulo, serão vistas características das adaptações, que servirão de base para o desenvolvimento da ferramenta de suporte à geração automatizada de adaptadores.

4.1 Características Desejáveis na Adaptação de Componentes

Segundo Bosch (1997a), temos os seguintes requisitos para técnicas de adaptação de componentes:

Transparência: A adaptação de um componente deveria ser tão transparente quanto possível. Transparência, neste contexto, indica que ambos, o usuário do componente adaptado e o próprio componente não ficariam sabendo da adaptação. Além disto, aspectos do componente que não precisam ser adaptados poderiam ser acessados sem esforço explícito da adaptação.

A caixa-preta: O engenheiro de software deverá desenvolver um modelo mental da funcionalidade do componente, antes que ele possa ser reusado. Esta funcionalidade deveria ser capturada de maneira simples e exata, apenas através da descrição da interface, para que a adaptação não precise ver como o componente é por dentro.

Composição: A técnica da adaptação deveria ser facilmente composta ao componente no qual ela se aplica. Este, adaptado, deve poder ser composto com outros

componentes, tanto antes quanto depois de adaptado. A adaptação deve poder ser integrada com outras adaptações.

Reusabilidade: Um problema da adaptação é que, dificilmente, ela é genérica o suficiente para ser reusada. A criação de padrões de interface, como em Silva (2002) e Dong (2000), sugere o uso de um conjunto de formalismos para descrever as interfaces dos componentes, através de padrões de projeto para resolver este problema.

***Eficiência*:** Para adaptar um componente, geralmente, um modelo maior e mais complexo tem que ser construído ao redor do antigo componente, gerando um aumento no tempo de resposta do serviço, já que ele será desviado. A adaptação deve tentar minimizar este atraso e otimizar o uso de recursos.

Os critérios acima são os fundamentos para o desenvolvimento de adaptadores neste trabalho.

4.2 Técnicas de Adaptação

4.2.1 Técnicas de Adaptação Caixa-Branca

Copiar e Colar: Quando um componente existente provê alguma funcionalidade útil para o desenvolvimento de uma aplicação ou para outro componente, o desenvolvedor poderia, simplesmente, copiar o código da parte do componente que pode ser reusada para o outro componente/sistema, que está sendo desenvolvido. Depois de copiar o código, o engenheiro de software vai fazer as modificações.

Embora a técnica de copiar e colar consista em algum reuso, ela tem muitas desvantagens, como o fato de que múltiplas cópias do código reusado podem existir e que o engenheiro de software precise entender intimamente o código reusado. (BOSCH, 1997a).

Herança: Dependendo do modelo da linguagem de programação, todos os aspectos internos, não apenas parte deles, são disponibilizados para reusar. A herança

tem uma importante vantagem: o código permanece em um único lugar. Entretanto, uma desvantagem é que o engenheiro de software, geralmente, deve ter conhecimento detalhado da funcionalidade interna da superclasse.

Ambas as técnicas estão relacionadas com o conhecimento interno dos componentes e não se encaixam na visão caixa-preta do componente. Por isso, não serão consideradas como opções para o desenvolvimento de adaptação para componentes neste trabalho.

4.2.2 Técnicas de Adaptação Caixa-Preta

Diferentemente da adaptação de caixa-branca, a adaptação de caixa-preta não requer que o engenheiro de software entenda detalhes internos do componente.

Esta técnica se preocupa apenas em adaptar a interface do componente, ao invés de sua estrutura interna. Esta será a técnica utilizada para a geração de adaptação neste trabalho.

A seguir, as técnicas de adaptação que podem ser feitas:

Agregação: Quando um componente tem muitos serviços associados a muitas portas, os desenvolvedores podem considerar a opção de repartir este componente em vários e dividir os serviços entre eles, ou usar componentes existentes que provêm parte destes serviços, criando um arcabouço deles, onde um seria o agrupador de outros. Neste arcabouço, os componentes poderiam ser encaixados e trocados de acordo com a afinidade entre eles. Embora a agregação seja mais orientada ao reuso do que à adaptação, o desenvolvimento de código no componente agregado possibilita que versões novas de componentes possam ser encaixadas no arcabouço, caso respeitem as regras expressas nele, gerando, desta maneira, um padrão para a adaptação. A solução para a adaptação é dada a partir da definição de uma semântica para a arquitetura, que definirá as restrições de conexão de componentes.

Utilizando esta técnica, o gerador de adaptação deveria considerar os aspectos arquiteturais definidos no arcabouço, quando realiza a adaptação. Como esta estrutura semântica não foi definida no Editor, nem na ferramenta de análise de compatibilidade (CUNHA, 2005), a arquitetura não foi considerada elemento semântico na adaptação,

pois esta técnica parece mais endereçar extensão de funcionalidade e não diretamente adaptação de componentes.

Empacotamento (*Wrapping*): O empacotamento gera código ao redor do componente adaptado, desviando as requisições que chegam ao original. Empacotamento é usado para adaptar o comportamento e estrutura de componentes, enquanto agregação é usada para compor novas funcionalidades aos já existentes.

Uma desvantagem importante do empacotamento é que ele pode resultar numa implementação consideravelmente pesada, pois todas as requisições para os componentes empacotados são gerenciadas pelo empacotador.

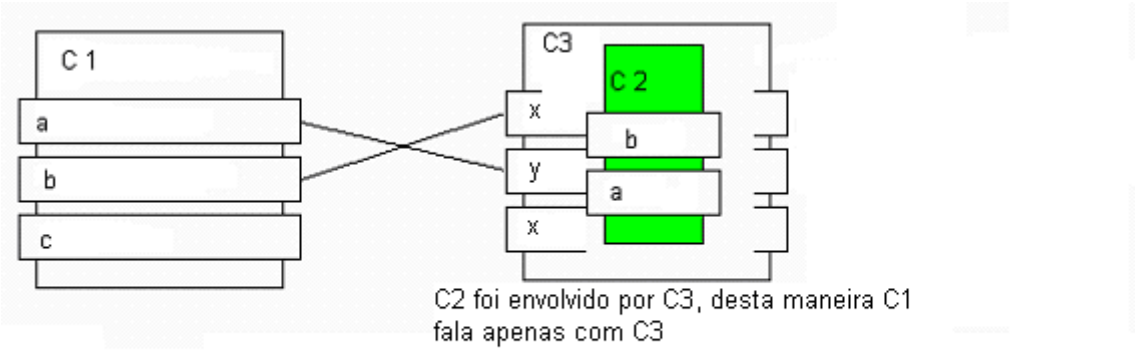

Figura 4.1 O componente C3 funcionando como um empacotador de C2.

A figura 4.1 ilustra a construção de um empacotador, onde o componente C2 foi totalmente envolvido pelo componente C3. De agora em diante, C1 fala apenas com C3 e não mais vê C2. Um problema desta abordagem é que, agora, toda requisição que chegaria em C2 passa antes pelo empacotador C3, mesmo aquelas que não precisem de adaptação, ferindo os critérios de transparência e eficiência vistos anteriormente.

Colagem (*Glue*): Em Bosch (1997b), foi definido:

> "Por colagem (*Glue*) os participantes entendem o pedaço de código que fica entre componentes usados para conectá-los (middleware). Podendo ser desde código Tcl/Tk[24], mecanismos de scripts, até mesmo arquivos. Algum suporte para tipagem seria interessante devido ao grande número de tipos de componentes que podem ser

[24] Tcl/ Tk é uma linguagem de programação de script gratuita e muita usada ao redor do mundo. Foi criada por John Ousterhout.

> usados para isso. Em geral a cola é mais flexível que os componentes colados...”

Pela definição acima, percebe-se que a colagem é o processo de adaptação, que inclui certa programação entre dois componentes.

Esta Cola pode ser usada para apenas conectar componentes, como, também, resolver questões de compatibilização, caso seja desenvolvida para realizar a integração de componentes incompatíveis. Nada impede, também, que seja vista como um outro componente dentro da aplicação, estruturado e respondendo às mesmas restrições que os componentes “de prateleiras”. Esta visão uniforme será utilizada neste trabalho.

Figura 4.2 Uma Cola ligando C1 e C2.

A figura 4.2 ilustra a utilização de um *Glue* (Cola) ligando C1 e C2, para resolver uma possível incompatibilidade entre as portas conectadas de C1 e C2. Da mesma maneira que o empacotador, ela intercepta e resolve incompatibilidade, porém, intercepta apenas as interações que estão envolvidas nesta última.

Embora empacotamento (*Wrapping*) e colagem (*Glue*) sejam semanticamente parecidos, a implementação de cada uma destas soluções é diferente. Enquanto na Cola, criamos um código/componente como ponte entre duas portas de componentes incompatíveis, o empacotamento prevê a criação de código que encapsularia o componente original dentro de si, numa visão de incorporação. Neste trabalho, apenas a colagem é utilizada, por questões relacionadas à simplificação da implementação que serão vistos no capítulo 7.

4.3 Tipos de Adaptação de Componente

4.3.1 Mudanças na Interface do Componente

Mudanças nos nomes das operações: Talvez seja o problema mais típico de reuso de componentes: a assinatura dos serviços providos por um componente não combina com a interface esperada. Este problema tem sido identificado por muitos engenheiros de software e o padrão de projeto *Adapter* (GAMMA, 1995) usado para resolver esse problema.

Padrões de projeto são soluções genéricas para problemas específicos, ou seja, quando um problema de implementação é encontrado muitas vezes durante o projeto de software em vários domínios, especialistas identificam soluções genéricas, que podem ser usadas sempre que aquele problema for encontrado. São problemas de projeto ligados, geralmente, a aspectos não-funcionais. (GAMMA, 1995).

Quando a definição de uma interface prevê o uso de padrões de projeto deste tipo, ao desenvolvedor, cabe entender o projeto apresentado e criar a implementação da interface solicitada para realizar a adaptação.

Porém, em termos de componentes, a utilização de padrões de projetos, como o citado acima, só seria possível se toda a estrutura da aplicação (arquitetura) já fosse definida usando estes padrões e os componentes apenas se encaixariam nestes. Uma situação incomum e que não será foco deste trabalho.

Restringir parte da interface: É a exclusão de parte da interface. No contexto de reuso, uma parte da interface pode não ser relevante ou mesmo contraprodutiva. A adaptação do componente deveria, então, restringir o acesso para operações excluídas. Desta maneira, o adaptador seria uma espécie de repassador (*proxy).* (GAMMA, 1995), por onde passariam todas as chamadas para o componente, e ele decide se lhe entrega ou não esta solicitação. Pode ser visto, também, como a utilização prática de um empacotador.

Restrições de clientes baseadas em estado: Um componente usado por vários clientes de vários tipos pode necessitar agir de várias maneiras. Este comportamento exige que o componente tenha interface diferenciada para cada tipo de cliente e cada um

pode somente acessar parte da interface suprida. Em suma, partes da interface do componente podem ser acessíveis ou não baseadas no estado desse. Esta restrição depende de uma verificação comportamental do componente, e só poderia ser implementada, se a interface definisse o comportamento e permitisse intervenções.

Delegação de requisições: Uma maneira para um componente prover um serviço não disponível dentro do próprio é delegar a requisição para outro, que seja capaz de fornecer o serviço. Para isso, o componente precisa ser estendido com o comportamento que delega certos serviços para outros componentes. Um empacotador, ou gerente de solicitação, verificaria se o serviço pode ser executado pelo componente, e, se não puder, desviaria a solicitação para outro, realizando numa adaptação funcional do componente. Neste caso, o empacotador deveria ter conhecimento total de todos os participantes da colaboração e repassar as solicitações para outros componentes, quase como um *middleware* na visão CORBA.

A adaptação funcional não é escopo deste trabalho. No nível de especificação de componentes, que é o nível de abstração tratada neste trabalho, seria difícil verificar este tipo de adaptação, já que, se um componente não provê um serviço, não haveria especificação da porta para fazer a conexão, tornando a solicitação deste serviço para o componente impossível. Esta adaptação está ligada à adaptação dinâmica de componentes, que será vista no Estado da Arte, no capítulo 5, porém, um empacotador, durante a fase de especificação, poderia fazer isso: repassar o que está implementado e redirecionar o que falta, caso esta adaptação esteja prevista na fase de desenvolvimento do componente/aplicação.

4.3.2 Adaptação Dinâmica

A adaptação dinâmica se refere à substituição de um componente por outro, ou de parte de seu comportamento, em uma aplicação em funcionamento, sem que o sistema tenha que ser abortado. Para sistemas que devem ficar em funcionamento ininterruptamente, esta adaptação é necessária, porém, difícil de ser implementada.

Em Mcgurren (2002), é apresentada uma taxonomia para sistemas dinâmicos, introduzindo uma proposta para suportar adaptação dinâmica, utilizando um tipo de adaptação chamada substituição de instância, usando terminologia de Orientação a

Objetos. Algumas linguagens de programação provêem mecanismos que podem ser usados para suportar esta substituição de instância em tempo de execução. Por exemplo, Java™ suporta a interceptação e o despacho de invocação de métodos em tempo de execução,porém, é necessário criar um mecanismo de gerência desta intervenção em tempo de execução, para assegurar a integridade do sistema. Em Mcgurren (2002), temos uma infra-estrutura que combina a adaptação com gerenciamento de integridade, que prevê o uso de um protocolo para esta interferência, ou seja, o momento correto de inserção da adaptação dentro do sistema em funcionamento. Geralmente, este mecanismo utiliza reflexão computacional[25].

4.3.2.1 Tipos de Adaptação Dinâmica de Componentes

Substituição de instância: É a substituição completa de um componente em tempo de execução. Uma atualização de versão usando o mesmo tipo[26] de componente, porém, com nova implementação.

A adaptação em nível de serviço: Quando um componente tem um grande número de clientes, outros componentes que o utilizam, ao invés de suprir um conjunto geral de serviços para seus clientes, o componente pode prover um conjunto mais especializado de serviços, através da reconfiguração das chamadas ele redireciona a chamada para o componente sob seu controle que executa esta funcionalidade.

Adaptação de interface: Refere-se à adaptação que ocorre, quando os componentes têm interfaces incompatíveis estruturalmente. Ela trata, especificamente, de renomear, reordenar e adaptar tipos primitivos dos parâmetros da interface. Pode-se verificar, aí, os mesmos problemas de adaptação em tempo de especificação, pois a verificação da compatibilidade deve ser feita antes da tentativa de conectar o novo componente na aplicação.

O mecanismo utilizado para inserir a adaptação no sistema em funcionamento não faz parte do escopo deste trabalho.

---

[25] Capacidade de um sistema refletir sobre sua estrutura, podendo, então, desviar a execução.

[26] Tipo, aqui, é entendido como a definição estrutural da interface dos componentes, ou seja, os serviços que estão amarrados às portas.

Ao se trabalhar no nível de abstração da especificação da aplicação, o tempo em que esta adaptação será efetivada não afeta o desenvolvimento da ferramenta proposta, pois o trabalho está localizado num nível semântico anterior.

No capítulo seguinte, tratamos o estado da arte da adaptação de componentes.

5 Estado da Arte

Neste, são vistas algumas propostas relacionadas com a adaptação de componentes, como por exemplo, adaptação dinâmica, usando reflexão computacional, soluções envolvendo linguagem de descrição de arquitetura, ambientes integrados de desenvolvimento baseado em componentes com verificação de compatibilidade etc. Este levantamento serviu como base para a implementação da ferramenta de adaptação.

5.1 Adaptação Dinâmica

5.1.1 Arquitetura X-Adapt

A arquitetura X-Adapt (MCGURREN, 2002) combina a adaptação com gerenciamento de integridade de sistemas. Isto é alcançado, usando uma arquitetura baseada em reflexão computacional, que divide o sistema em duas partes: o meta nível e o nível base.

O meta nível mantém informação sobre as propriedades do sistema, que fica no nível base, e implementa meta classes e meta objetos, que contém informações sobre os objetos do nível base. A comunicação entre os dois níveis é feita através de um Protocolo de Meta Objetos (MOP[27] – sigla do inglês para *Meta Object Protocol*).

A reflexão computacional não está relacionada a nenhuma característica específica de linguagem de programação, pois várias linguagens, como Java™, C++ e SMALLTALK, fornecem mecanismos para reflexão computacional.

Elementos arquiteturais da X-Adapt:

[27] É a interface entre o meta nível e o nível base para obter informação e alterar o comportamento do nível base.

Contêiner: Análogo ao contêiner *Enterprise JavaBeans*™ (EJB)[28], onde todos os objetos do sistema são implantados. Ele age como uma fábrica que pode ser usada para criar instâncias dos objetos e manter informações sobre eles.

***Proxy* (procurador):** Age como intermediário entre os níveis: meta nível e o nível base. O seu propósito principal é controlar o acesso aos componentes do segundo.

Gerente de configuração: Quando um *proxy* é criado, ele se registra dentro do gerenciador de configuração, assim, o gerenciador sabe qual componente, no nível base, é referenciado por qual componente no meta-nível. Dentro do gerenciador, estão as definições das conexões entre os componentes, e, com este conhecimento, ele pode intervir na execução. Existe apenas um gerente de configuração para cada aplicação e ele é criado junto com o contêiner. É ele que realiza as intervenções no nível base para modificar a execução.

A adaptação realizada é feita em tempo de execução da seguinte maneira: quando o gerente precisa interceptar chamadas, ou seja, caso o gerente queira desviar uma chamada para outra instância de um determinado componente, ele intervém, utilizando reflexão computacional, e captura a chamada do nível base, obtendo, através do *proxy,* o componente que responderá a chamada.

Este desvio já deve ser programado ou previsto dentro do gerente em tempo de desenvolvimento, não exatamente para qual instância, mas para qual componente, numa visão estática.

Nesta abordagem, o autor apresenta uma modificação na arquitetura, semelhante à EJB, para facilitar a adaptação em tempo de execução. Não apresenta ferramentas para fazê-lo, nem como testar este trabalho antes de executar, por isso, não se trata de uma proposta de desenvolvimento apoiado por ambiente e, sim, uma arquitetura genérica para os programadores usarem ou não.

A implementação da adaptação utilizando reflexão computacional é uma estratégia que vai ao encontro da necessidade e tecnologias atuais, pois pode ser executada, enquanto o sistema está funcionando. Porém, todo o projeto do sistema deve antecipar esta necessidade. Por isso, a fase de especificação dos componentes é a chave para a adaptação dinâmica acontecer, validando a proposta a ser vista no capítulo 6.

---

[28] EJB - *Enterprise JavaBean* é uma arquitetura de componentes da Sun™ para desenvolvimento de aplicações distribuídas, utilizando linguagem Java.

Em Jarir; David e Ledoux (2002), temos outra proposta de adaptação dinâmica dentro do EJB, para fazer a conexão entre os *beans* do projeto original e a adaptação dos componentes de middleware, sempre utilizando reflexão computacional, no sentido de adaptar as interações entre os componentes dos dois tipos.

O objetivo é melhorar o *Enterprise JavaBeans*™, permitindo que as aplicações fiquem cientes e se adaptem a variações do contexto da execução. Os autores propõem adaptar dinamicamente a associação entre componentes EJB e os serviços de middleware necessários. Assim, aplicações em *Enterprise JavaBeans*™ têm a vantagem de adaptar-se dinamicamente de acordo com mudanças relacionadas ao contexto de execução. Esta abordagem se relaciona intimamente com a arquitetura EJB, e parte dela para a realização de uma proposta de adaptação com os componentes de middleware. Porém, mais uma vez, surge a necessidade de uma especificação destes componentes em alguma fase anterior à sua aplicação do sistema em execução. Nesta fase, concentra-se este trabalho.

5.2 Adaptação em Tempo de Execução, Utilizando uma Linguagem Específica

A comunicação que ocorre entre os níveis base e meta nível, numa visão reflexiva de um sistema, é feita, usando protocolos de meta objetos, que, de uma maneira geral, implementam alteração de comportamento no nível base, especificando protocolos particulares de interação entre componentes. Esta comunicação pode ser feita de várias maneiras, não havendo um padrão conhecido.Por isso, muitos trabalhos apresentam linguagens padrão para realizar esta comunicação. (WELCH, 1998).

Segundo Welche Strout (1998), não existe um modelo formal geral para desenvolvimento de protocolos de meta objetos, o que dificulta a justificativa do porquê usá-lo. Entretanto, os autores acreditam que o modelo **WRIGHT**, uma linguagem de definição arquitetural (ADL[29]), permite modelar protocolos de meta objetos, de forma reutilizável e aplicável a componentes.

[29] ADL – É uma notação que permite uma descrição precisa e uma análise de propriedades externamente visíveis da arquitetura de software, suportando diferentes estilos arquiteturais e níveis de abstração.

Nesta abordagem, toda a comunicação entre os componentes é feita através de uma ADL específica, a **WRIGHT,** que interconecta todos os componentes do sistema, no nível base ou meta nível. Esta linguagem permite uma série de análises, como todos os modelos formais, e, por isso, os autores consideram esta abordagem mais segura e reutilizável do que outras maneiras de representar a conexão entre componentes.

Como ADL é uma linguagem que especifica a conexão entre os componentes, numa visão arquitetural, e vê as conexões como estruturas semânticas ativas, a **WRIGHT** cria conectores parametrizáveis para dar certa flexibilidade a estas conexões, e é através deles, que é implementado o meta protocolo.

Segundo os autores, modelar meta protocolos como conectores pode aumentar a compreensibilidade e explicitar como ele se relaciona com a arquitetura do sistema. Além de que o uso de linguagem formal permite uma série de análises como consistência, laços infinitos e bloqueios mortais.

Os autores consideram este um caminho para a criação de um ambiente, que junte a flexibilidade de trabalhar com componentes, a reflexão computacional e o formalismo de ADL, objetivando, assim, trabalhar com a visão arquitetural de composição de sistemas baseados em componentes.

A sugestão de usar os conectores para implementar o meta protocolo é apenas uma opção. O meta protocolo também pode ser implementado como componentes iguais aos outros, sem ser de um tipo específico (conector), e previsto em tempo de desenvolvimento.

Esta abordagem tem a grande vantagem de ser uma visão sobre a arquitetura da aplicação, o que pode ser visto como uma possibilidade para futuras implementações para o trabalho aqui apresentado.

5.3 Adaptação Utilizando Código Binário do Componente

Keller, em sua obra de 1998, propõe um sistema de adaptação de componentes em código binário como um mecanismo para modificar componentes já existentes, tais

como arquivos ***.class*** [30] da linguagem Java™, permitindo, desta maneira, que se adaptem e evoluam, quando não se tem acesso ao código fonte, para garantir compatibilidade entre versões.

A reflexão computacional, disponibilizada em Java, permite a reificação[31] de arquivos ***.class*** para se obter a estrutura e compreender o seu comportamento.

A adaptação de componentes usando seu código binário permite mais flexibilidade na composição de objetos, segundo o autor, movendo algumas decisões importantes, como o nome do meta objeto, o relacionamento e como eles serão chamados, do tempo de produção de um componente para o tempo de integração, possibilitando aos programadores adaptar os componentes às suas necessidades.

Para Keller (1998), produtores de componentes e consumidores gastam considerável esforço para integrar e evoluir os componentes. Adaptação de componentes binários pode reduzir o esforço e permitir seu reuso efetivo para as necessidades particulares de uma determinada aplicação, através do suporte de evoluções previstas e não previstas para o componente.

Esta abordagem difere das outras, porque ela reescreve os componentes binários antes ou durante a carga. Essa adaptação toma lugar, depois que o componente foi entregue para o programador. Esta solução é totalmente dependente de linguagem e fere a noção de caixa-preta de componentes usada neste trabalho, por isso, está aqui apenas para ilustrar as possibilidades de adaptação, que estão disponíveis para os desenvolvedores, mas não se encaixa na visão aqui apresentada, ou seja, onde componentes são caixas-pretas e não se pode ver dentro deles; o que se pode inferir deve derivar apenas de suas interfaces.

5.4 Adaptação Usando Componentes de Middleware

Componentes de *middleware*[32] estão relacionados com a interoperabilidade de sistemas. Arquiteturas como DCOM™ e CORBA apresentam soluções para

---

[30] *.class* é a extensão dos arquivos compilados na linguagem Java™.

[31] Reificação é o processo pelo qual o sistema obtém informação de estrutura de si mesmo em tempo de execução.

[32] Componentes geralmente relacionados com interoperabilidade, como CORBA e DCOM™.

middleware, que implementam código não-funcional, ou seja, não ligado às funcionalidades do sistema. (TRUYEN. 2000).

Normalmente, sistemas distribuídos utilizam um destes dois tipos de arquitetura de componentes para operacionalizar a comunicação entre os componentes funcionais do sistema. Seria interessante adaptar as associações entre esses dois tipos de código, funcional e não-funcional, quando tratados e endereçados por diferentes componentes. (TRUYEN, 2000).

Em Truyen (2000), são criticadas as estruturas monolíticas e inflexíveis dos modelos CORBA e DCOM™. Para flexibilizar estes modelos e dar ao *middleware* mais adaptabilidade, é proposta uma ferramenta de reconfiguração para a atualização de sistemas em execução, garantindo a disponibilidade de aplicações, tais como banco de dados e telecomunicações. Esta reconfiguração precisa ser capaz de refletir sobre as funcionalidades já existentes do sistema e reconfigurar sua funcionalidade num ponto apropriado de intersecção.

Para refletir sobre o aplicativo e obter informações estruturais dos componentes envolvidos na interação, foi desenvolvido um protótipo em Java, que reifica todas as trocas de mensagens para descobrir se deve ou não intervir.

Esta abordagem pretende reunir a visão de componentes de *middleware*, principalmente CORBA, com desenvolvimento de componentes funcionais em Java, utilizando o *BeanBox*[33], e estendendo suas funcionalidade para prever reflexão computacional e integrar estes dois tipos de componentes em tempo de execução.

A principal vantagem é a utilização de um ambiente gráfico, onde os componentes podem ser visualizados e as conexões, feitas e testadas, porém, uma questão importante é a utilização de tecnologias específicas para realizar esta tarefa, como Java. Por isso, esta abordagem parece voltada mais para um mercado específico, enquanto este trabalho é mais genérico que isto.

---

[33] Ambiente de desenvolvimento de componentes Java.

5.5 Adaptação Facilitada por Ambiente em Tempo de Desenvolvimento

Em Lee, Know, Kim e Shun (2003), os autores tentaram resolver os três problemas, segundo eles, da tecnologia de desenvolvimento baseada em componentes: a identificação do componente, a sua adaptação e a utilização de componentes visuais, utilizando *plug-and-play*[34].

A primeira tarefa foi desenvolver uma metodologia, a chamada MARMI II, que especifica todo o processo de desenvolvimento e procedimentos para suportar o desenvolvimento dentro da ferramenta. A segunda tarefa do projeto foi desenvolver uma ferramenta que suporta todo o processo de desenvolvimento baseado em componente, como sua identificação modelagem, implementação, extração, desdobramento, teste e geração de relatórios. Este ambiente, chamado COBALT, suporta a modelagem baseada em arquitetura. Segundo os autores, a força desta proposta está em integrar uma metodologia a um ambiente para desenvolvimento de componentes EJB/J2EE™.

A adaptação ocorre, quando se quer usar um componente que não tem exatamente a mesma interface que é chamada. Desta maneira, o ambiente adapta esta chamada.

Para atender a demanda originária das mudanças constantes de tecnologia, a ferramenta encara os problemas, tais como integração e interoperabilidade entre diferentes tecnologias, usando MDA (*Model Driven Architecture* – Modelo Orientado pela Arquitetura), que é um método de desenvolvimento de sistemas baseados em padrões OMG.

Portanto, esta abordagem permitiria aos engenheiros de software integrar, mudar e manter um software, independentemente do modelo e tecnologia, mas a ferramenta hoje só atende EJB e J2EE™. Um ponto forte deste ambiente é que ele prevê a geração de código e o teste, além de ser um ambiente para desenvolvimento, suportando vários modelos de componentes no mercado.

---

[34] Ligar e Executar, no sentido de apenas se conectar os componentes, sem ter que realizar nenhum trabalho extra.

Este trabalho não está relacionado a nenhuma tecnologia específica, já que se trabalha em nível de especificação de interface, então, a preocupação com o modelo de mercado não é determinante.

Outra diferença entre as abordagens é o uso de um Editor Gráfico desenvolvido para o SEA, que usa representação específica para componentes, o que não ocorre no ambiente COBALT, que utiliza o diagrama de classes da UML.

O ambiente COBALT trata os componentes visualmente com notação UML (Diagrama de Classes), o que pode gerar confusão de conceitos, já que podem ser desenvolvidos em qualquer paradigma de desenvolvimento, importando apenas a interface.

Outro ponto importante a destacar sobre o ambiente COBALT é a maneira como tratam a adaptação: apenas num nível estrutural, ou seja, verificam a incompatibilidade de interfaces e a ajustam, enquanto o trabalho apresentado aqui realiza a adaptação no nível comportamental, também, e, ainda, representa visualmente todos os elementos da adaptação igualmente: componentes e adaptadores.

5.5.1 Ferramenta Geradora de Adaptadores Utilizando Grafos

Temos em Inverardi (2002), uma proposta de arquitetura, na qual é possível verificar quando e por que o sistema apresenta anomalias, através da análise do comportamento de um grafo, que é gerado a partir das interações dos componentes do sistema.

É assumido que a especificação formal dos comportamentos é fornecida, ou através de diagramas de seqüência de mensagens ou pela especificação de alto nível dos componentes, ou, ainda, por algum tipo de linguagem de descrição formal, além de uma definição precisa das propriedades do sistema, sua arquitetura.

Com essas duas premissas, segundo Inverardi e Tívoli (2002), seria possível desenvolver uma ferramenta automática, a qual derivaria o código conector para um conjunto de componentes de um sistema, que satisfizesse as propriedades previamente estabelecidas. Os autores assumiram que especificações formais dos comportamentos dos componentes deveriam estar disponíveis.

Um grafo individual para cada componente é criado a partir das definições da interface (definição formal) e juntado com os grafos dos outros componentes, na medida

em que estes são conectados, gerando, assim, um grafo resultante, sobre o qual são aplicadas análises, disponíveis sobre grafos.

Alguns comportamentos podem ser levantados e resolvidos através desta ferramenta, como bloqueio e laços infinitos, porém, a geração do código que resolve estes problemas ainda não é automática. Outro problema é que a geração dos grafos e sua análise podem causar uma explosão de espaço/tempo, prejudicando o uso desta abordagem.

5.6 Ambiente de Especificação de Software SEA/OCEAN

OCEAN é o nome do *framework* que possibilita a criação de ambientes de desenvolvimento de software, ou seja, a partir dele pode-se criar ambientes que manipulam diferentes estruturas voltadas para a especificação e diferentes funcionalidades. (SILVA, 2000).

Este *framework* deu origem ao ambiente SEA, que provê suporte para o desenvolvimento de software, possibilitando a utilização integrada das abordagens de desenvolvimento orientado a objetos, baseado em componentes e *frameworks.* Neste ambiente, o desenvolvimento de *frameworks*, componentes e aplicações consiste em construir a especificação de projeto destes artefatos, utilizando UML (com modificações), e convertendo esta especificação em código, de forma automatizada. (SILVA, 2000).

O *framework* OCEAN e, conseqüentemente, o ambiente SEA, foram implementados em SMALLTALK, sob o ambiente *VisualWorks*, reutilizando classes da biblioteca de classes deste ambiente.

5.6.1 Suporte ao Desenvolvimento e Uso de Componentes no Ambiente SEA

O ambiente SEA suporta especificação de interface de componentes, usando notação UML (*Unified Modeling Language* – Linguagem Unificada de Modelagem), para a representação visual das portas e seus métodos associados, e uma outra para definição de comportamento, utilizando a rede de Petri, formando o primeiro

documento de especificação de componentes, que é o documento de especificação de interface (*Component Interface*).

Depois desta especificação de interface, o desenvolvedor acessa a ferramenta que gera a estrutura, que permite avaliar e assegurar a compatibilidade estrutural e comportamental na conexão entre as interfaces dos componentes: o padrão de projeto para interface de componentes apresentado em Silva (2002), inicialmente concebido em Silva, 2000, que gera um outro documento de especificação dentro do SEA (*framework* específico para o padrão de projeto. (SILVA, 2002)).

Em seguida, é especificada uma classe herdada da classe *SEAComponent*, a fim de criar um conceito agrupador das especificações anteriormente criadas: o Componente. Finalmente, tendo-se o Componente, pode-se utilizá-lo, agora, para especificar aplicações.

O ambiente SEA vê e trabalha com componentes, de forma semelhante a classes para UML: classes que implementam as interfaces dos componentes são importadas para especificações orientadas a objetos e, nelas, interligadas, através da especificação de algoritmos que implementam as conexões. Esta forma de tratamento de componentes pode levar a uma confusão de conceitos, pois estamos falando de componentes caixas-pretas, com portas de conexão e comportamento, não simplesmente uma classe com métodos e atributos.

Esta deficiência na manipulação de componentes e representação visual dos mesmos foi um motivador para o desenvolvimento de um Editor Gráfico, dentro do SEA, para manipular suas arquiteturas de forma visual. Outra motivação para desenvolver a ferramenta dentro do SEA foi a existência do OCEAN, sobre o qual ele foi implementado, e que pôde ser usado para agilizar o desenvolvimento, utilizando classes e objetos já disponibilizados no *framework* para estender o SEA, além da existência de uma ferramenta de análise comportamental (CUNHA, 2005), que gera a entrada de dados para a realização da adaptação.

5.6.2 Especificação de Componentes no SEA

A figura 5.1 mostra a tela inicial do Ambiente SEA:

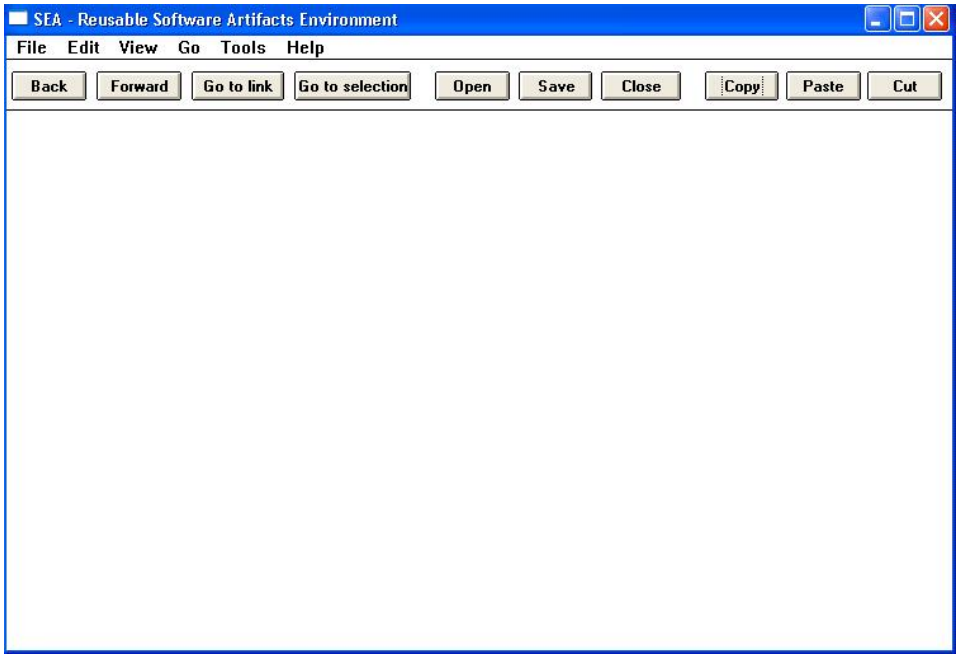

Figura 5.1 Ambiente SEA.

5.6.2.1 Criando a Interface de Componentes no SEA

Neste ambiente é possível editar especificações de interfaces de componentes utilizando a opção de menu: *File* → *New Especification* que abre a janela mostrada na figura 5.2, que é o primeiro passo para a criação da interface de componentes.

Figure 5.2 Seleção de especificação de interface no SEA.

Na tela, apresentada pela figura 5.2, o desenvolvedor pode selecionar entre criar as especificações de interface de componentes, arquitetura de componentes, que é a especificação de uma nova aplicação, usando especificações de componentes existentes, ou outras especificações desenvolvidas por outros trabalhos junto ao ambiente SEA, como WorkFlow e estrutura de documentos textuais.

Na tela da figura 5.2, pode ser criada uma especificação de interface de componente. Todo o processo de criação de interface de componente está disponibilizado no ambiente SEA (SILVA, 2000). O objetivo de mostrar como criar estas especificações se justifica por serem estas definições que determinam como será disponibilizada a especificação das interfaces dos adaptadores. Criá-la dentro do SEA é criar a sua especificação estrutural: portas e serviços e a rede de Petri, que expressa o seu comportamento.

5.6.2.2 Criando a Estrutura da Interface no Ambiente SEA

Para criar a especificação da estrutura da interface do componente IMestrado, seleciona-se a opção mostrada na figura 5.3.

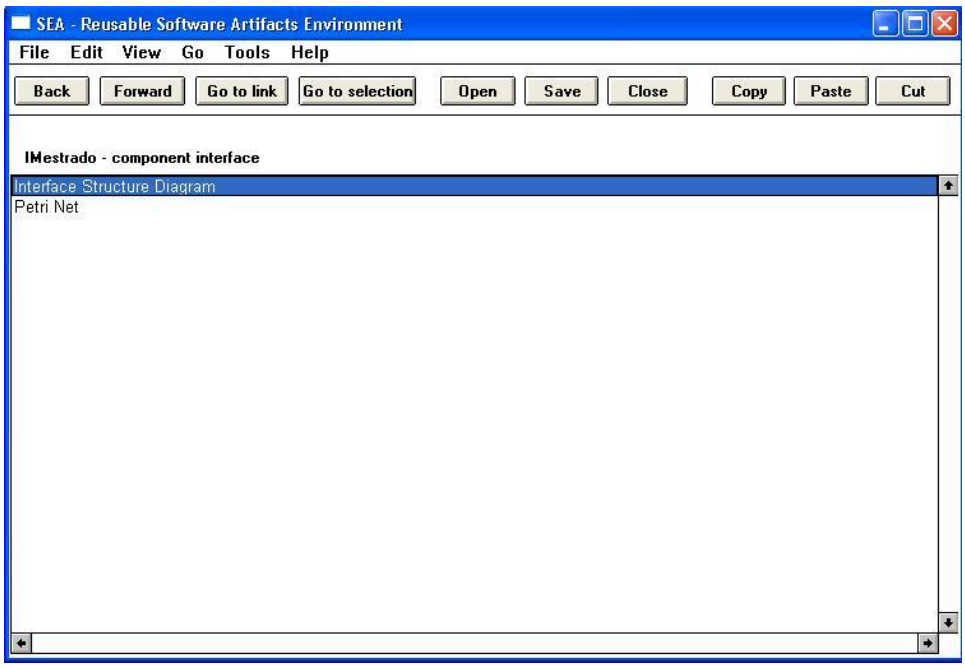

Figura 5.3 Criação de diagrama de estrutura.

Na figura 5.3 o desenvolvedor seleciona a opção "*Interface Structure Diagram*" para poder especificar a estrutura: as Portas do componente, ou canais, e os serviços associados às portas. Para isso aperta-se o botão "*Go to selection*".

A figura 5.4 mostra como editar um novo diagrama de estrutura. Para isso basta selecionar do menu a Opção "*Edit*" seguida da opção "*newModel*".

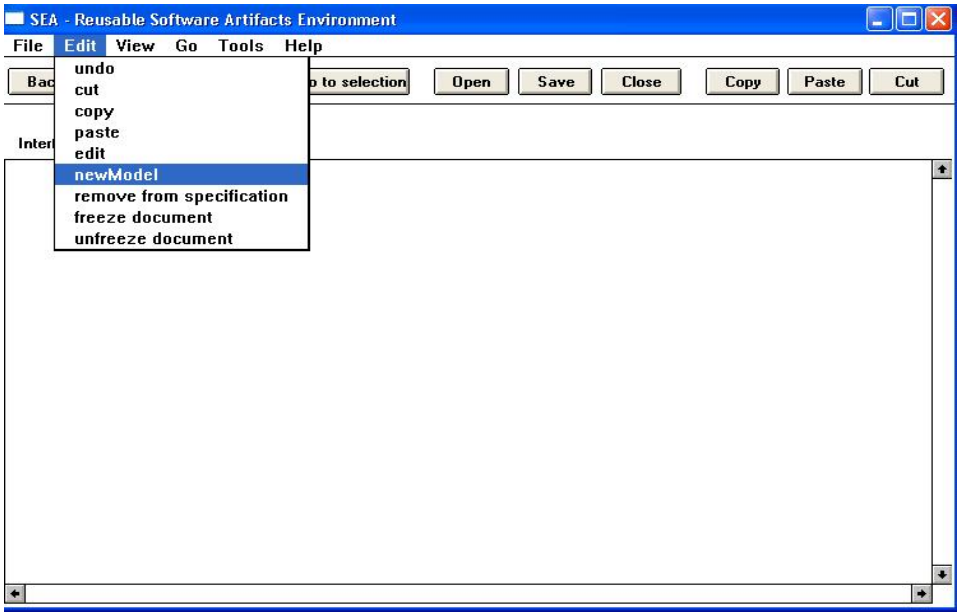

Figura 5.4 Criando um novo diagrama de estrutura.

A figura 5.5 mostra o diagrama de edição da estrutura dos componentes:

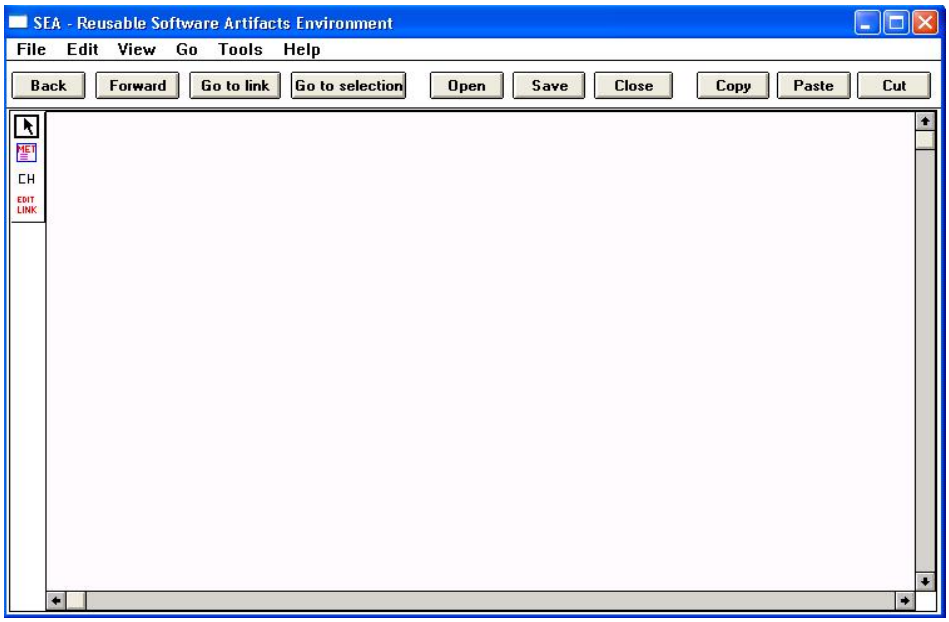

Figura 5.5 Criação da estrutura da interface.

Na figura 5.5 o desenvolvedor seleciona as opções de criar serviços supridos ou requeridos e portas onde estarão estes serviços.

Para criar métodos supridos, o desenvolvedor deve selecionar a opção "*MET*" da barra de ferramentas, à esquerda que está grifada na figura 5.6 e clicar duas vezes sobre o ambiente de edição para criar os desenhos que representam as estruturas de métodos supridos e requeridos como mostrado na figura 5.6.

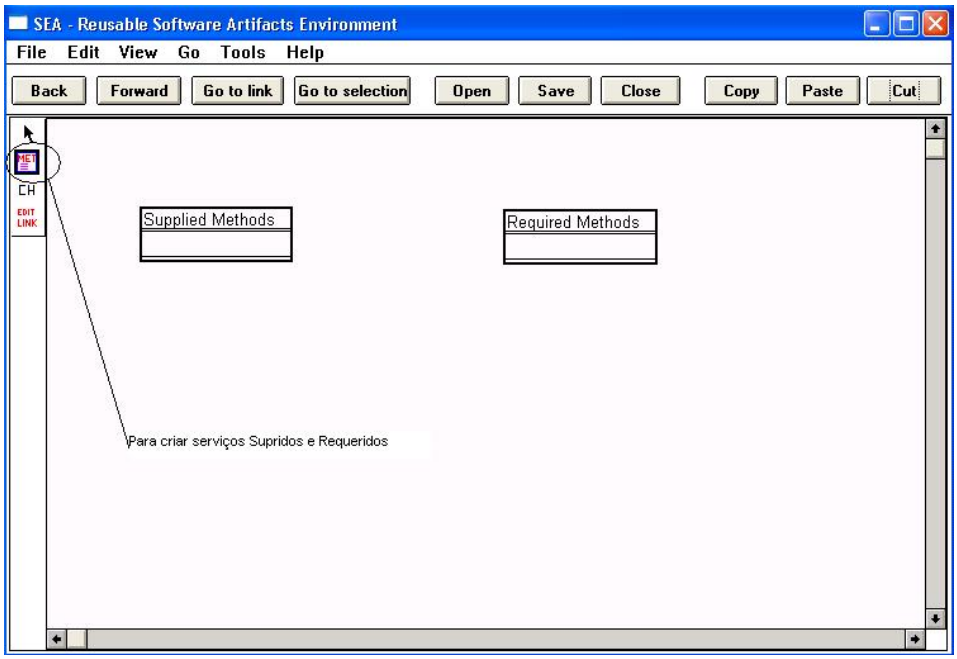

Figura 5.6 Edição de serviços requeridos e supridos.

Para editar os serviços dentro da caixa "Supplied Methods" basta selecionar o desenho e apertar o botão de menu "*Go to selection*", a seguir a opção de edição de serviços é mostrada da figura 5.7:

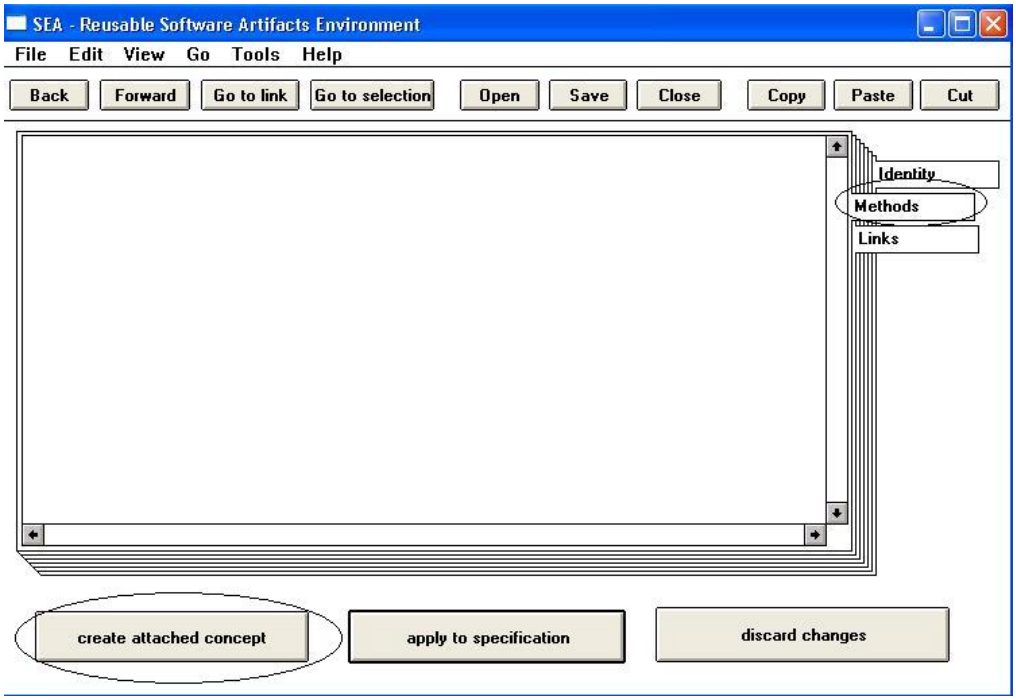

Figura 5.7 Adicionando serviços supridos e requeridos.

Para criar serviços selecione a lingueta "*Methods*" grifada na figura 5.7 e aperte o botão "*create attached concept*".

Para criar portas ou canais utilize a opção "*CH*" da barra de ferramentas, como mostra a figura 5.8.

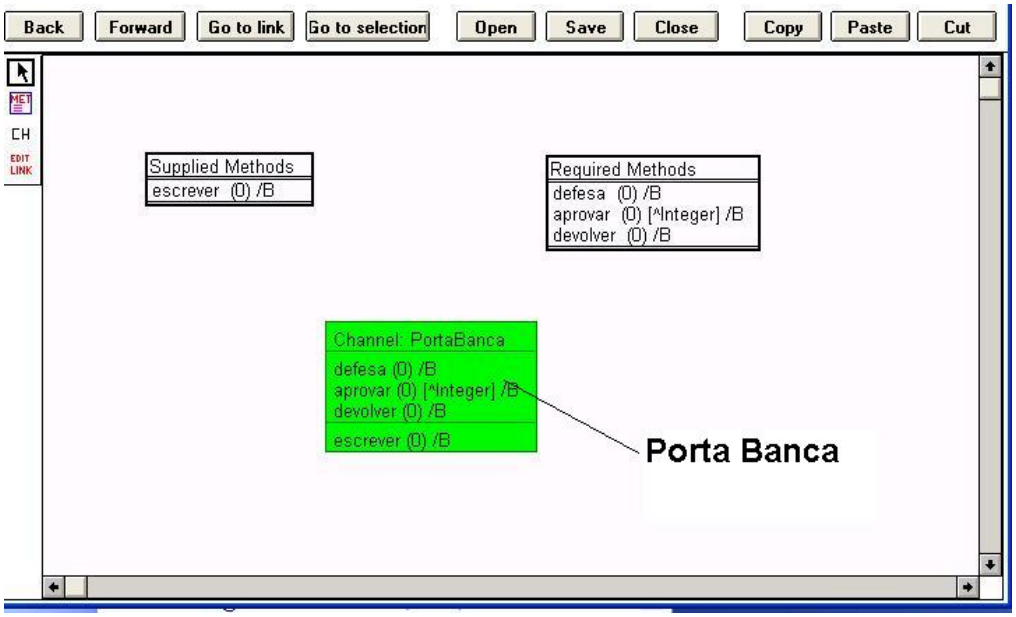

Figura 5.8 Criação das portas.

Na figura 5.8, o desenvolvedor deve, após selecionar a opção de criação de portas "*CH*" clicar sobre o ambiente e nomear a porta, no exemplo acima o nome escolhido foi "PortaBanca". O desenvolvedor poderá criar quantas portas quiser, porém, deverá associar, pelo menos, um serviço a cada uma delas.

Para associar serviços à porta deve-se clicar sobre a porta criada e apertar o botão "*Go to selection*".

Após selecionar todos os serviços da porta o desenvolvedor tem a visão da estrutura de seu componente como mostra a figura 5.8.

Na tela da figura 5.8, o desenvolvedor pode editar a estrutura com todos os serviços do componente, divididos entre supridos (*Suplied Methods*), aqueles que são fornecidos pelo componente para serem usados, e os requeridos (*Required Methods*), aqueles que outros componentes oferecem.

Também são definidas as portas (chamadas no ambiente por *Channel* – consideradas sinônimos nesse trabalho), apresentadas em cor verde no ambiente SEA.

No exemplo da figura 5.8, tem-se o *Channel* PortaBanca indicando que a interface de componente IMestrando terá uma interface com Banca.

Dentro da porta PortaBanca, estão os serviços: defesa, aprovar e devolver. Estas estruturas são serializadas, podendo ser alteradas pelo desenvolvedor, e acessadas por qualquer outra definição de componentes, pois, assim, a interface é completamente separada do componente e é a ele agregada depois. Para salvar a estrutura aperte o botão "*Save*".

5.6.2.3 Criando o Comportamento no Ambiente SEA

Comportamento, dentro do ambiente SEA é expresso como uma rede de Petri simples com lugares e transições, a única alteração é que a rede de Petri tem serviços associados às suas transições, dando a orientação da ordem de chamada destes serviços. A presença da ficha orienta o início do processamento, ou seja, por onde o componente inicia sua operação, ou qual serviço é habilitado primeiro, depois disso o disparo natural da rede mostra o andamento correto desta operação.

Para criar a rede de Petri do componente o desenvolvedor deve selecionar a opção "*Petri Net*" mostrada na figura 5.3, a mesma onde se seleciona o diagrama de estrutura do componente. Ao selecionar esta opção deve-se seguir os mesmos passos

para criar o diagrama de estrutura, ou seja, apertar o botão "*Go to selection*" e selecionar do menu a opção "*Edit*" → "*Newmodel*". Ao selecionar esta opção o seguinte editor de rede de Petri do ambiente SEA aparecerá, como mostra a figura 5.9.

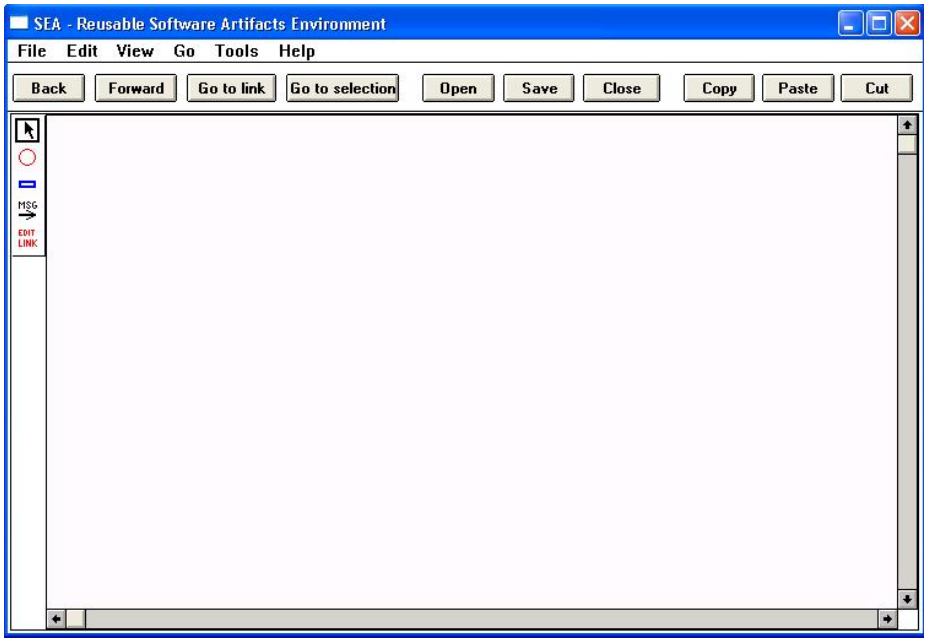

Figura 5.9 Editor de rede de Petri.

Na tela representada pela figura 5.9 o desenvolvedor poderá selecionar da barra de ferramentas, à esquerda, as opções de lugares e transições representados por um círculo e um retângulo, respectivamente.

Ao editar os lugares e as transições o desenvolvedor seleciona da mesma barra de ferramentas, a opção "*MSG*" que é o arco que liga as transições aos lugares gerando uma rede de Petri como mostrada da Figura 5.10.

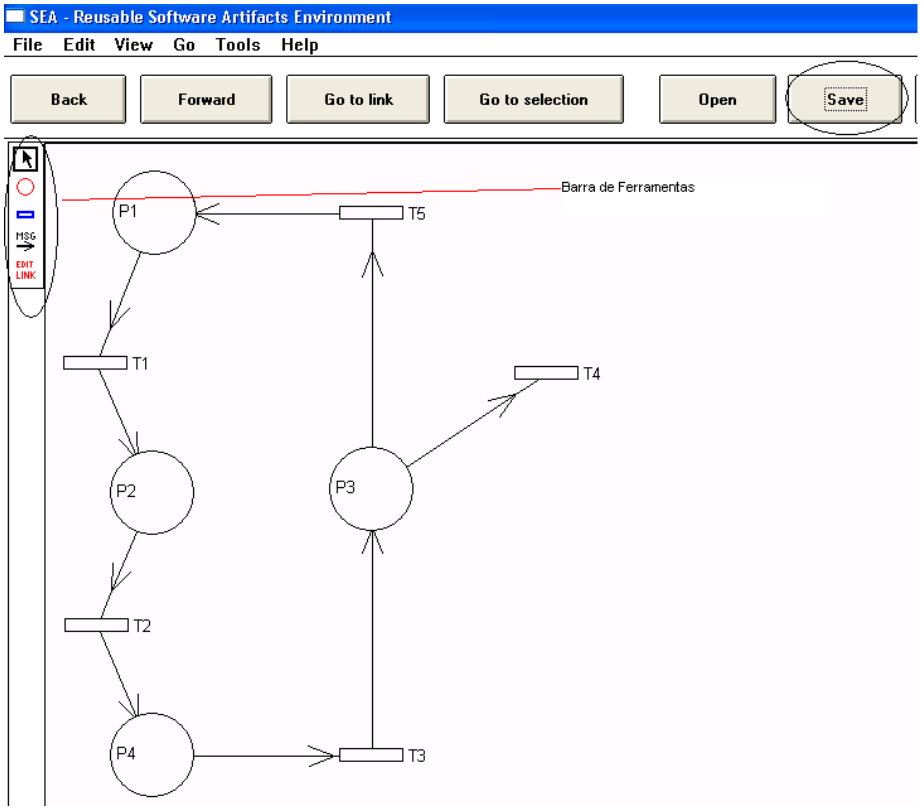

Figura 5.10 Rede de Petri.

Após criar os lugares, transições e ligar através dos arcos, o desenvolvedor deve associar serviços às transições criadas, para isso deverá clicar sobre a transição e apertar no botão "*Go to selection*". A figura 5.11 mostra como o desenvolvedor deverá associar os serviços às transições.

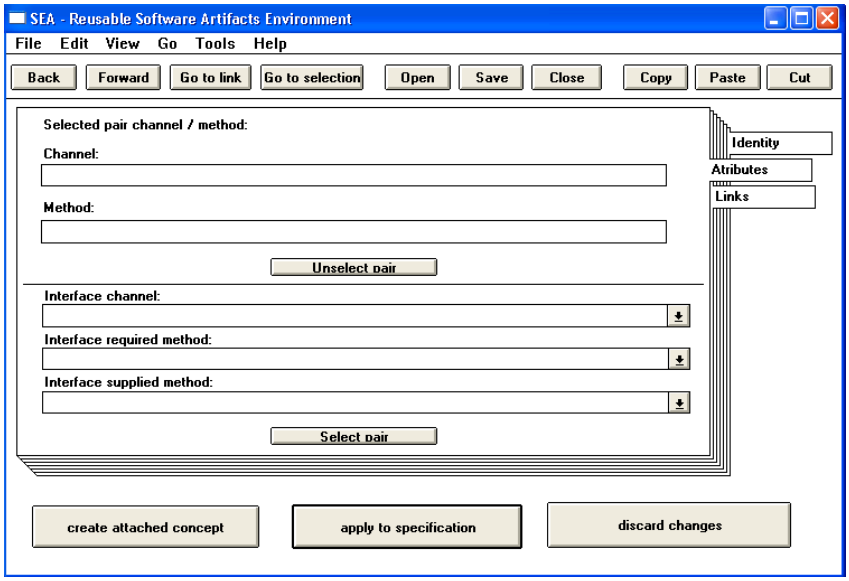

Figura 5.11 Associar serviços às transições da rede de Petri.

A figura 5.11 mostra como o desenvolvedor poderá associar o par Porta/Serviço a uma transição, para isso deve selecionar na combo "*Interface Channel*" a porta do serviço a ser associada, em seguida, se o serviço for requerido, selecionar o serviços na combo "*Interface required methods*", ou então, se ele for suprido por aquela porta, selecionar o serviço na combo "*Interface supplied method*". Após selecionar o par pertinente, deve apertar o botão "*Select pair*" e "*Apply to specification*", desta maneira a seleção dos pares fica ativa para a transição na rede de Petri, e assim sucessivamente para todas as transições da rede de Petri editada.

Para inserir uma ficha dentro de um lugar o desenvolvedor deverá selecionar o lugar, clicando sobre ele, apertar o botão "*Go to selection*" e clicar na lingüeta "*Attributes*", como ilustrado na figura 5.12.

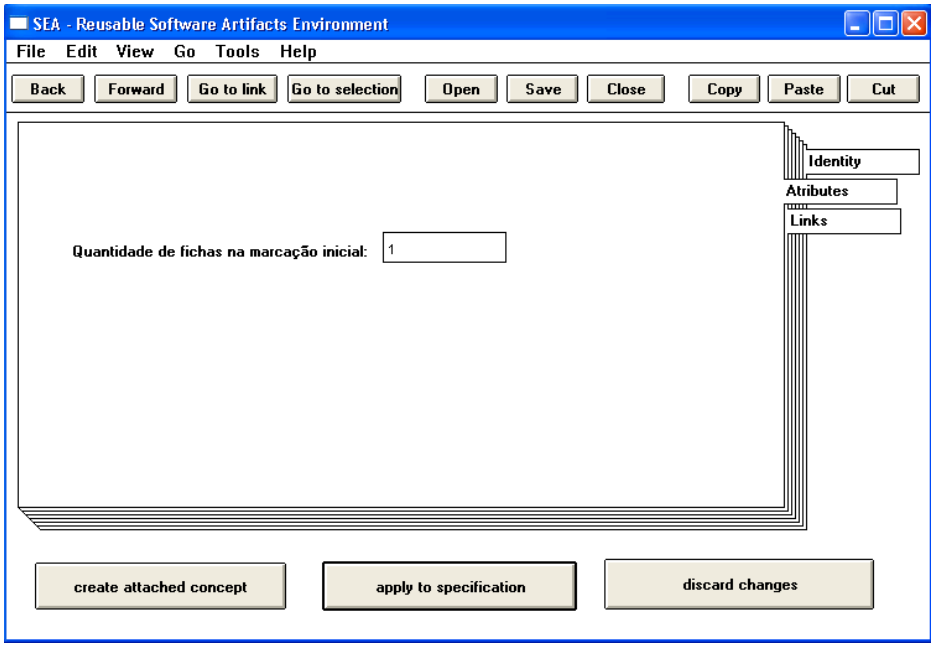

Figura 5.12 Inserindo fichas em um lugar.

Para finalizar, deve-se apertar o botão da "*apply to specification*" para voltar à edição da rede de Petri.

Após a criação da interface do componente deve ser executada uma análise para verificar se o componente foi criado com correção. Para isso selecione no menu principal a opção "*Tool*" → "*Load Tool*", como mostra a figura 5.13.

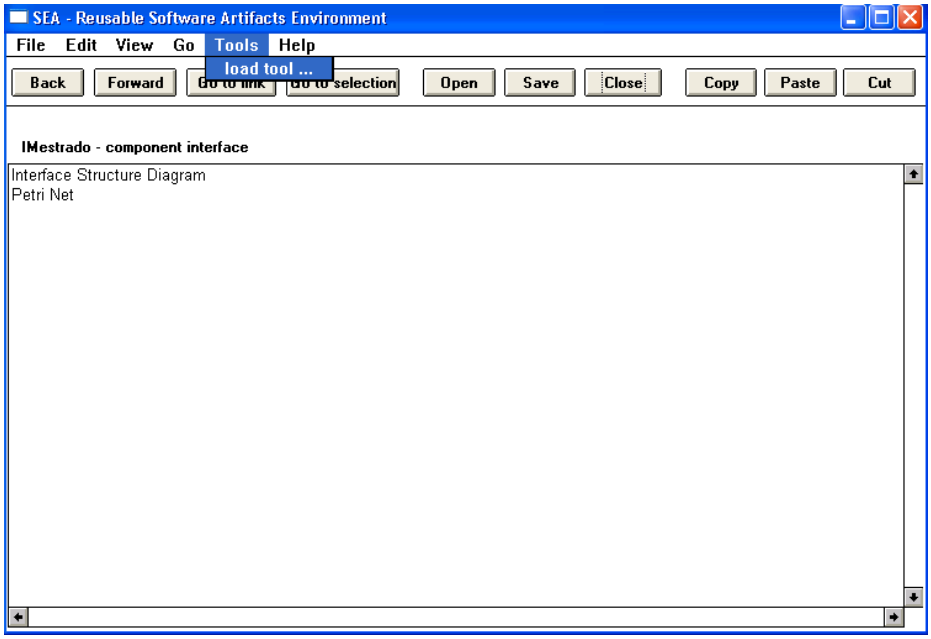

Figura 5.13 Ferramenta de análise da estrutura.

Esta ferramenta irá realizar uma análise inicial sobre a especificação da interface criada.

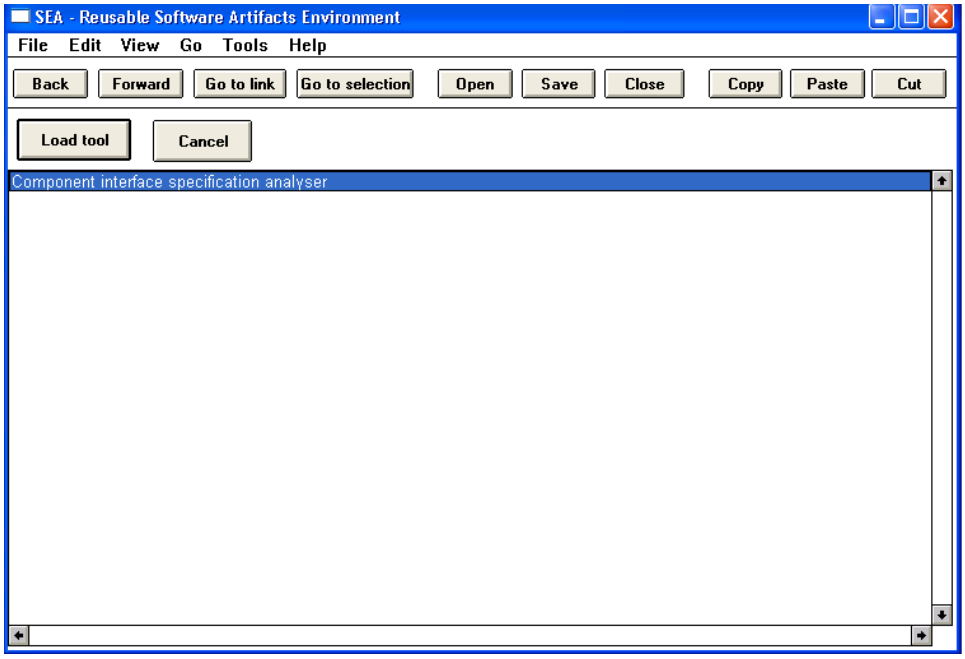

Figura 5.14 Carregando a ferramenta de análise da estrutura.

Após selecionar a ferramenta "*Component interface specification analyser*", deve-se clicar no botão "*Load tool*" para carregar a ferramenta e caso tudo esteja certo

aparecerá a mensagem: “Fim do procedimento de análise. Especificação validada”, senão, deve-se abrir o arquivo checking.chk que fica no diretório C:\VISUAL\SEA\CHECKING para verificar o erro encontrado.

5.6.3 Especificando Aplicações no Ambiente SEA

Após criar as interfaces, elas poderão ser usadas no desenvolvimento de especificações de aplicações. Para criar uma aplicação baseada em componentes, o desenvolvedor deve escolher “*Arquitetura de componentes*”, na janela apresentada na figura 5.15.

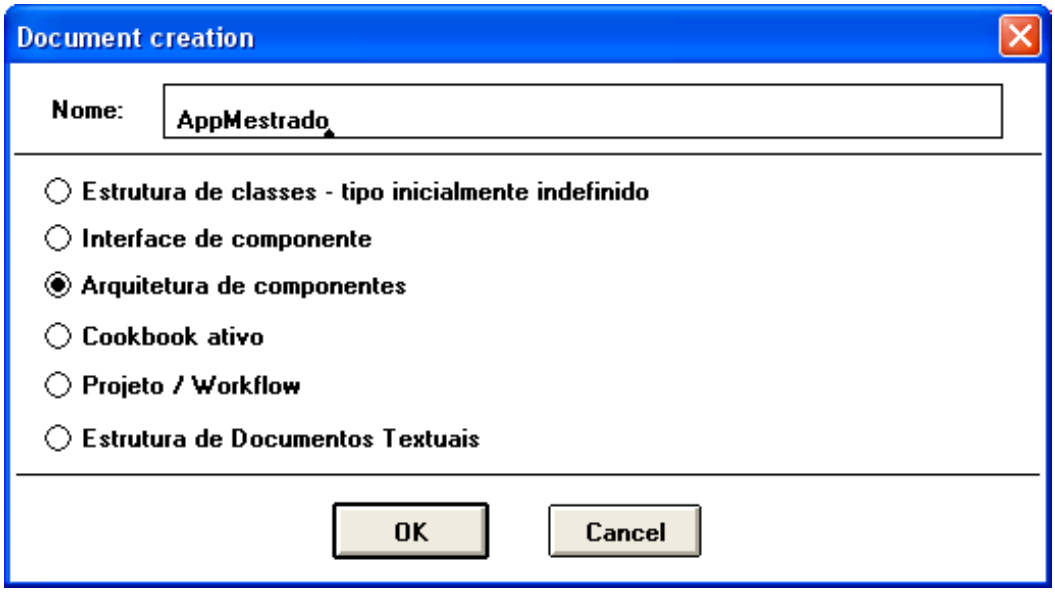

Figure 5.15 Escolha do tipo de especificação do SEA.

Após selecionar o nome da Especificação de Arquitetura de componentes, o desenvolvedor pode selecionar qualquer interface pronta na biblioteca de componentes do SEA.

A figura 5.16 mostra a seleção do diagrama de Arquitetura de Componentes.

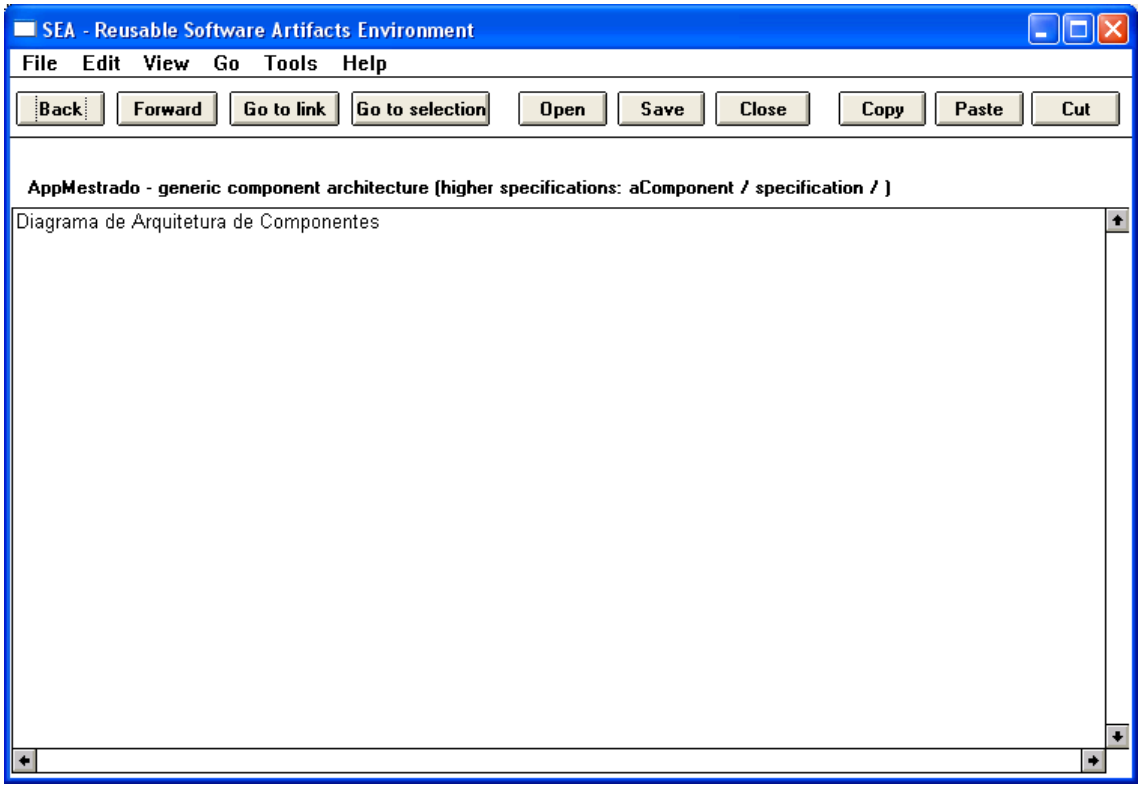

Figure 5.16 Janela de seleção do modelo de componentes do SEA.

Na figura 5.16, o desenvolvedor pode escolher se quer criar uma arquitetura de componentes, que é uma aplicação para usá-los. Ao selecionar esta opção, o Editor é aberto, disponibilizando o ambiente gráfico para realizar a aplicação, conforme a figura 5.17.

Editor Gráfico de Componentes:

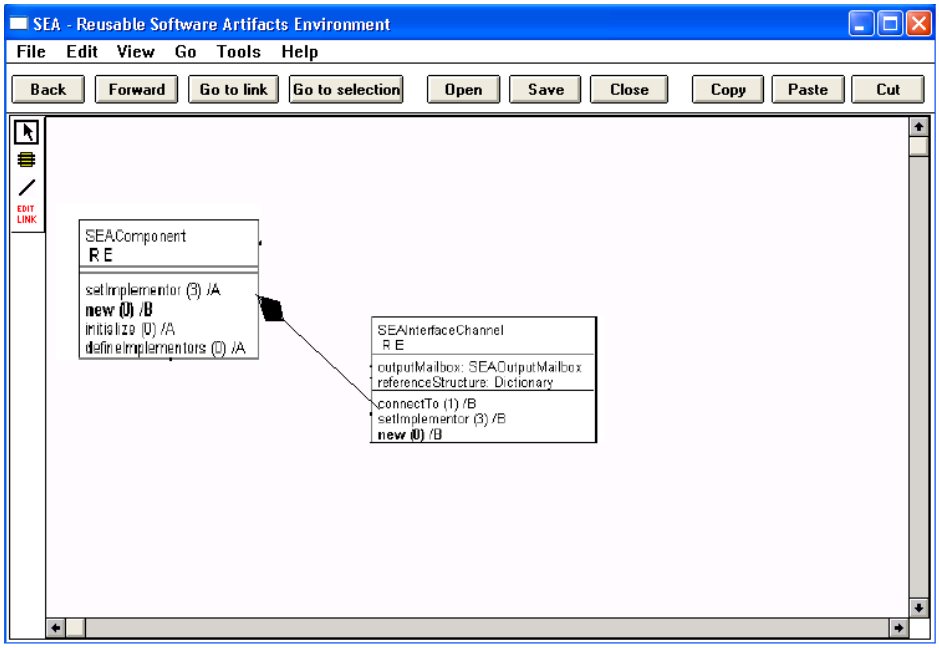

Figure 5.17 Editor de componentes.

A Figura 5.17 ilustra o tratamento de componentes dado pelo ambiente SEA. Antes da implementação deste trabalho, o SEA tratava componentes com representação de classes, não tinha semântica para as ligações e não executava análise de compatibilidade (CUNHA, 2005), nem tão pouco realizava adaptação como proposto neste trabalho. A implementação de um editor de componentes com representação e semânticas próprias era um trabalho futuro da tese de Silva, 2000 e sua realização será mostrada no capítulo 6, que trata da pesquisa realizada.

5.6.4 Considerações Finais

O ambiente SEA vê e trabalha com componentes, de forma semelhante a classes para UML: classes que implementam as interfaces dos componentes são importadas para especificações orientadas a objetos e, nelas, interligadas, através da especificação de algoritmos que implementam as conexões. Esta forma de tratamento de componentes pode levar a uma confusão de conceitos, pois estamos falando de componentes caixas-pretas, com portas de conexão e comportamento, não simplesmente uma classe com métodos e atributos.

Esta deficiência na manipulação de componentes e na representação visual dos mesmos foi um motivador para o desenvolvimento de um Editor Gráfico, dentro do

SEA, para manipular suas arquiteturas, que faz parte do visual da ferramenta desenvolvida, ou seja, onde o usuário pode acessar a adaptação e visualizar a sua realização.

Outro motivador para desenvolver a ferramenta dentro do SEA foi a existência do OCEAN, sobre o qual ele foi implementado, e que pôde ser usado para agilizar o desenvolvimento, utilizando classes e objetos já disponibilizados no *framework* para estender o SEA, além da existência de uma ferramenta de análise comportamental (CUNHA, 2005), que gera a entrada de dados para a realização da adaptação.

A utilização do ambiente SEA, ao invés de um outro apresentado neste trabalho, como, por exemplo, o Wrigth, para a implementação desta pesquisa foi motivada pela necessidade de envolver estrutura e comportamento e a existência de uma ferramenta de análise já pronta (CUNHA, 2005) o que adiantou muito o trabalho da adaptação.

Com base nos conceitos levantados nos capítulos iniciais deste trabalho e a situação atual de desenvolvimento do ambiente SEA, além da motivação exposta acima, a seguir, serão vistos os resultados da pesquisa e como a ferramenta foi desenvolvida.

6 A Abordagem de Adaptação de Componentes Desenvolvida

6.1 Introdução

Neste capítulo, é apresentado o desenvolvimento da solução proposta para a geração semi-automatizada de adaptadores de componentes e a ferramenta de suporte à adaptação inserida no Ambiente SEA.

O objetivo deste esforço foi encontrar maneiras de resolver algumas pendências da abordagem baseada em componentes relacionadas com o reuso, além de demonstrar que a existência de padrões de interface pode ajudar e, inclusive, viabilizar a construção de ferramentas para suportar a geração automática de adaptadores, a fim de efetivar o reuso, mesmo em condições improváveis.

O reuso, referido acima, seria efetivado, pois, quando dois componentes numa interação apresentam incompatibilidades, estrutural e/ou comportamental, o adaptador desenvolvido nesta pesquisa apresenta a definição da interface do componente que resolve esta incompatibilidade. A implementação da estrutura interna dos adaptadores não está contemplada nesta pesquisa, mas sim a geração da interface destes adaptadores. Este trabalho foi possível devido à disponibilização do framework OCEAN, que possibilitou o reuso de uma série de classes já prontas, e também a existência da ferramenta de análise (CUNHA, 2005), que gera a entrada de dados para a escolha e apresentação dos adaptadores.

Será visto, também, como são criados e exibidos os componentes dentro do ambiente SEA. O processo de criação das especificações dos componentes já fazia parte do ambiente, apenas a visualização foi alterada, como será mostrado, pois foi criada uma representação única para componentes e um ambiente gráfico onde ela pode ser manipulada.

Esta representação foi criada e implementada em conjunto com a construção da ferramenta de análise (CUNHA, 2005), pois ambas utilizam este ambiente gráfico para disponibilizar as ferramentas e as telas relacionadas com a verificação da incompatibilidade e com a geração de adaptadores.

Após a etapa de criação das interfaces dos componentes e da análise final da arquitetura dos componentes feitos em Cunha, 2005, são disponibilizadas ferramentas e opções para realizar a adaptação.

As adaptações tratadas são a estrutural e a comportamental, já que a adaptação funcional não está no escopo deste trabalho. A seguir, será mostrado como criar as especificações dos componentes, que serão usados como exemplos para apresentar os resultados desta pesquisa.

6.2 O Editor de Componentes desenvolvido

Este editor foi desenvolvido em parceria com Cunha, 2005, pois ambas as pesquisas tratavam de componentes e necessitavam de um ambiente diferenciado para visualizar os componentes. O desenvolvimento deste Editor é um acréscimo ao ambiente SEA, porém, não é o foco final desta pesquisa, sendo, por isso, tratado aqui apenas como um facilitador do tratamento visual, apesar da dificuldade e complexidade em se desenvolver ambientes gráficos em SMALLTALK.

A figura 6.1 mostra o Editor Gráfico desenvolvido:

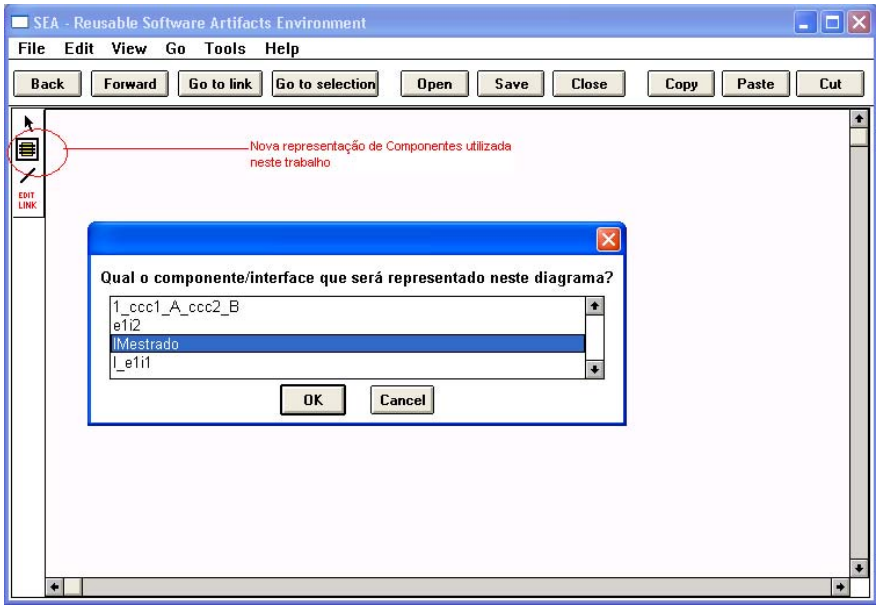

Figura 6.1 Versão nova, com editor de diagrama de componentes.

Na figura 6.1 vemos, grifado, à esquerda, a nova representação de componentes que ao ser clicada apresenta a janela mostrada na figura, onde aparecem a lista de interfaces de componentes que podem ser inseridos na especificação da nova aplicação baseada em componentes. Após selecionar o componente o Editor fica como mostrado na figura 6.2, abaixo:

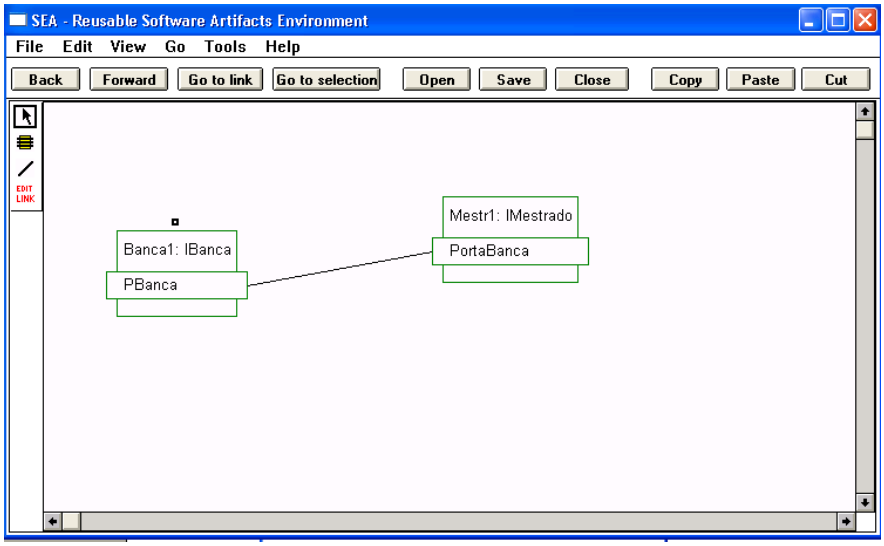

Figura 6.2 Nova versão do editor gráfico com componentes.

Na figura 6.2 o componente aparece em verde porque é ainda apenas uma interface de componente, se fosse um componente pronto apareceria em preto.

A diferença entre este modelo e o uso do modelo de classes da versão anterior é que o componente encapsula os dados de seus serviços supridos e requeridos nas portas, numa representação única.

Dentro do Editor foram adicionadas as ferramentas que fazem a análise de compatibilidade (CUNHA, 2005) e a ferramenta de geração dos adaptadores, que é apresentada neste trabalho.

6.3 Análise da Compatibilidade dos Componentes

Para realizar a análise Cunha, 2005, inseriu uma série de ferramentas diretamente do Editor, como mostra a figura 6.3.

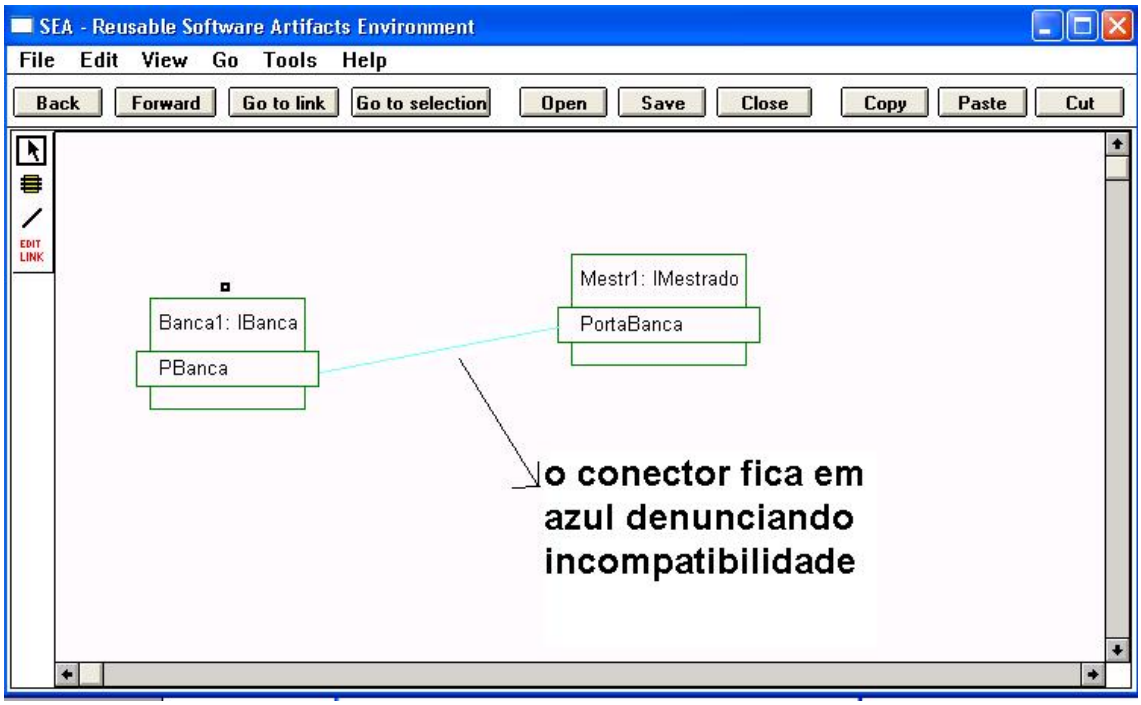

Figura 6.3 Editor de componentes - incompatibilidade.

Na figura 6.3, a ligação entre os componentes Banca1 e o componente Mestr1 apresenta-se em azul, pois houve incompatibilidade. O conector foi implementado para possuir uma semântica visual que endereça o estado atual da compatibilização dos componentes (CUNHA, 2005). Para resolver a incompatibilidade deve-se clicar sobre o conector e apertar o botão "*Go to selection*".

A janela representada pela figura 6.4 é a janela onde a compatibilização estrutural acontece.

No quadro ao pé da janela tem-se a incompatibilidade encontrada, ou seja, o serviço defender requerido por Mestr1 a Banca1 não tem correspondência em Banca1 assim ele é forçadamente compatibilizado.

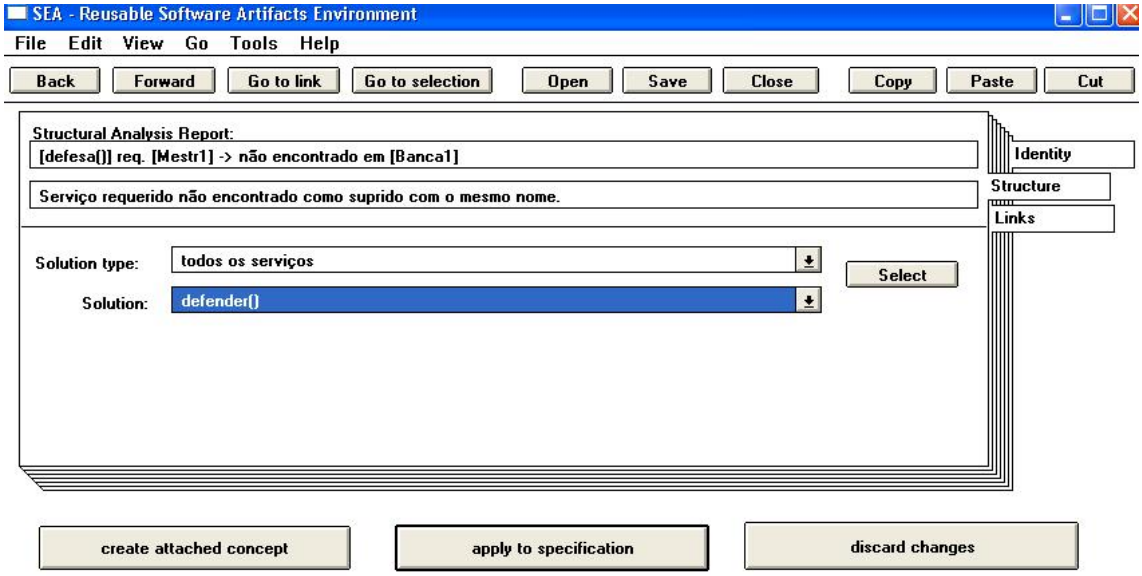

Figura 6.4 Descrição da incompatibilidade estrutural.

O desenvolvedor deve, como mostrado na figura 6.4 selecionar uma opção da combo "*Solution Type*" e um método da combo "*Solution*" a apertar o botão "*Select*". Então a ferramenta de compatibilização estrutural realiza a compatibilização (CUNHA, 2005).

A partir daí o conector se torna preto para identificar a compatibilização entre os componentes.

A ferramenta de análise de compatibilidade de Cunha, 2005. realiza a compatibilização da rede de Petri resultante da junção das redes de Petri dos componentes Mestr1 e Banca1. A ferramenta de análise de compatibilidade varre esta rede de Petri resultante e verifica a existência de anomalias, como bloqueios, e relata estes erros para serem usadas pela ferramenta de geração de adaptadores.

O resultado da análise de compatibilidade feita pela ferramenta de análise estrutural e comportamental, (CUNHA, 2005) pode ser acessado no arquivo que fica no diretório C:\SEA\CHECKING\. No arquivo ficam descritos os erros encontrados, no

caso de exemplo, temos que o serviço chamado ler, que é requerido de pesquisa não foi encontrado.

Quando esta análise termina, e encontra erros de compatibilização estrutural e/ou comportamental pode ser disparada a ferramenta de geração de adaptadores.

6.4 Geração de Especificação de Adaptadores Estruturais no Ambiente SEA

Os adaptadores de estrutura devem interceptar a chamada do serviço, e desviá-la, dentro do adaptador, para o componente originalmente chamado. Neste desvio, o adaptador não deve interferir no comportamento do sistema como um todo, apenas naquele desvio, e não interferindo no comportamento interno do componente (caixa-preta). Para isso, o algoritmo de geração de adaptador estrutural deve seguir os passos demonstrados a seguir.

Algoritmo de construção do adaptador estrutural

Etapa 1: Criar a estrutura da Cola[35]:

A Cola deve ter uma porta para receber a solicitação e outra para fazer a chamada ao serviço adaptado, e ter os mesmos serviços envolvidos nas portas que estão sendo adaptadas.

1 Criar a estrutura da Cola;

1.1 Criar a especificação do componente Cola com duas portas P1 e P2;

1.2 Ler os serviços das portas de origem e destino envolvidas;

1.2.1 Para origem e destino, faça: inserir na porta P1/origem e P2/destino, os serviços com direção contrária (requerido=suprido e suprido=requerido.

Etapa 2: Criar o comportamento da Cola:

[35] Cola será usado aqui como sinônimo de adaptador.

O comportamento da Cola deve manter o comportamento original da interação entre os componentes adaptados, isto é, a ordem original de chamada deve ser respeitada pela Cola, salvo o desvio para adaptação a ser feita.

1 Criar o comportamento da Cola

1.1 Criar as transições de acordo com a definição estrutural;

1.2 Associar as transições com as Portas/Serviços criados;

2 Verificar, nas redes de Petri dos componentes adaptados, as restrições de ordem dos serviços[36];

3 Verificar a restrição de ordem associada às ligações da chamada original do componente requisitante, com a chamada a ser feita dentro da Cola[37];

4 Aplicar a mesma lógica de restrição para todas as transições das redes de Petri pertencentes à porta;

5 Verificar quais das transições podem ser "quebradas", para acomodar outra(s) transição (ões) no meio, e quais não podem[38];

6 Construir uma única seqüência de transições[39], a partir das considerações do passo anterior. Para isso, devemos:

6.1 Tomar uma das duplas que não podem ser quebradas (qualquer uma das restrições encontradas acima);

6.2 Buscar, nas duplas restantes, uma que comece com o final da atual restrição. Caso não encontre, vá para o passo 7;

6.3 Gerar uma nova restrição, agregando o serviço da restrição encontrada;

6.4 Quando a cadeia encontrada, puder ser quebrada, considerar como últimos elementos o final da cadeia encontrada e buscar uma dupla que comece com essas transições;

6.5 Aplicar os passos anteriores (6.2, 6.3 e 6.4).;

7 Criar a rede de Petri do adaptador, inserindo lugares entre as transições na ordem definida em 7;

8 Colocar as fichas nos mesmos lugares que haviam nas redes dos componentes adaptados.

---

[36] Restrição de ordem significa quais as transições que são sucessoras das transições. Esta varredura na rede de Petri dos componentes é feita para cada transição criada na Cola para saber qual será colocada antes e depois, a fim de manter o comportamento de antes da adaptação.

[37] Isto implica que através da Cola, a chamada do serviço adaptado do componente 1 deve invocar imediatamente o serviço suprido pelo componente 2, surgindo, desta maneira, a restrição de invocação entre estes serviços.

[38] Para isso, é preciso ver as restrições de ordem levantadas anteriormente, verificar quais devem ser executadas imediatamente após a outra e, quais são independentes, ou seja, que podem ser quebradas para acomodar a adaptação.

[39] A seqüência dá a orientação dos arcos e lugares dentro da rede de Petri da Cola.

6.5 Geração do Adaptador Comportamental

Na adaptação comportamental, diferentemente da adaptação estrutural, é o comportamento que deve mudar, e nenhuma transição é criada.

Nesta pesquisa, o único comportamento a ser adaptado é o bloqueio. Por bloqueio, entende-se que é uma rede de Petri, onde uma transição aguarda outra para executar e esta aguarda a primeira, portanto, nenhuma das duas é executada, até que fichas sejam colocadas nos lugares que as antecedem, liberando a transição.

Exemplo de rede de Petri em bloqueio:

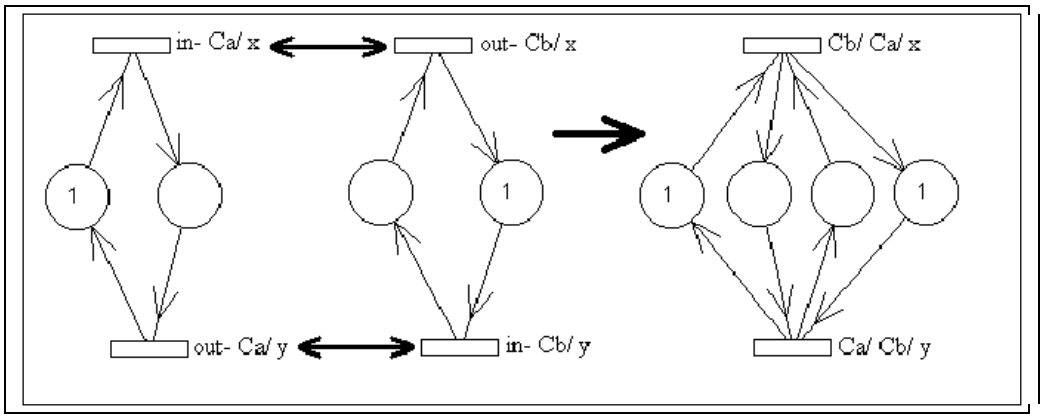

Figure 6.5 Exemplo de rede resultante em bloqueio. (SILVA, 2002)

Na figura 6.5, vê-se como Silva (2002) previa a realização da rede resultante da união das redes de Petri de dois componentes, com o objetivo de verificar se houve ou haverá um bloqueio na execução. Neste exemplo, percebemos que a união de Ca/x com Cb/x e Ca/y com Cb/y, numa única transição, implica em manter os lugares de entrada e saída e as fichas na posição original. Ao verificar a resultante, pode-se constatar que nenhuma transição está habilitada, pois, nos lugares de entrada de Ca/Cb/y e Cb/Ca/x, falta uma ficha, assim, entende-se que há um bloqueio na rede de Petri resultante. Para entender melhor a verificação deste bloqueio deve-se ler Cunha (2005). A ferramenta desenvolvida naquele trabalho permite que se atinja a rede resultante de uma interação. Deste modo, pode-se avaliar se existe ou não bloqueio na evolução da rede.

A seguir, será visto como o bloqueio será tratado do ponto de vista da geração de adaptadores.

6.6 Algoritmo para Criar o Adaptador Comportamental

1 Criar a estrutura da Cola;

1.1 Criar a especificação do componente Cola com duas portas P1 e P2;

1.2 Ler os serviços das portas de origem e destino envolvidas;

1.2.1 Para origem e destino faça: inserir na porta P1/origem e P2/destino os serviços com direção contrária (requerido=suprido e suprido=requerido).

A Cola deve ter uma porta para receber a solicitação e outra para fazer a chamada ao serviço adaptado, devendo ter os mesmos serviços envolvidos nas portas que estão sendo adaptadas.

Deve-se então:

Repetir o procedimento do Algoritmo para a adaptação estrutural, apenas diferenciando o seguinte ponto:

Alterar o Passo 8 (Colocar as fichas nos mesmos lugares, que haviam nas redes dos componentes adaptados) para:

8 Colocar as fichas nos lugares que antecedem a transição que se deseja liberar.

A definição de bloqueios e os bloqueios encontrados podem ser melhor vistos na ferramenta de Cunha, 2005.

6.7 Exemplos

Exemplo de adaptação estrutural

Dados os componentes:

Mestrando com as seguintes definições:

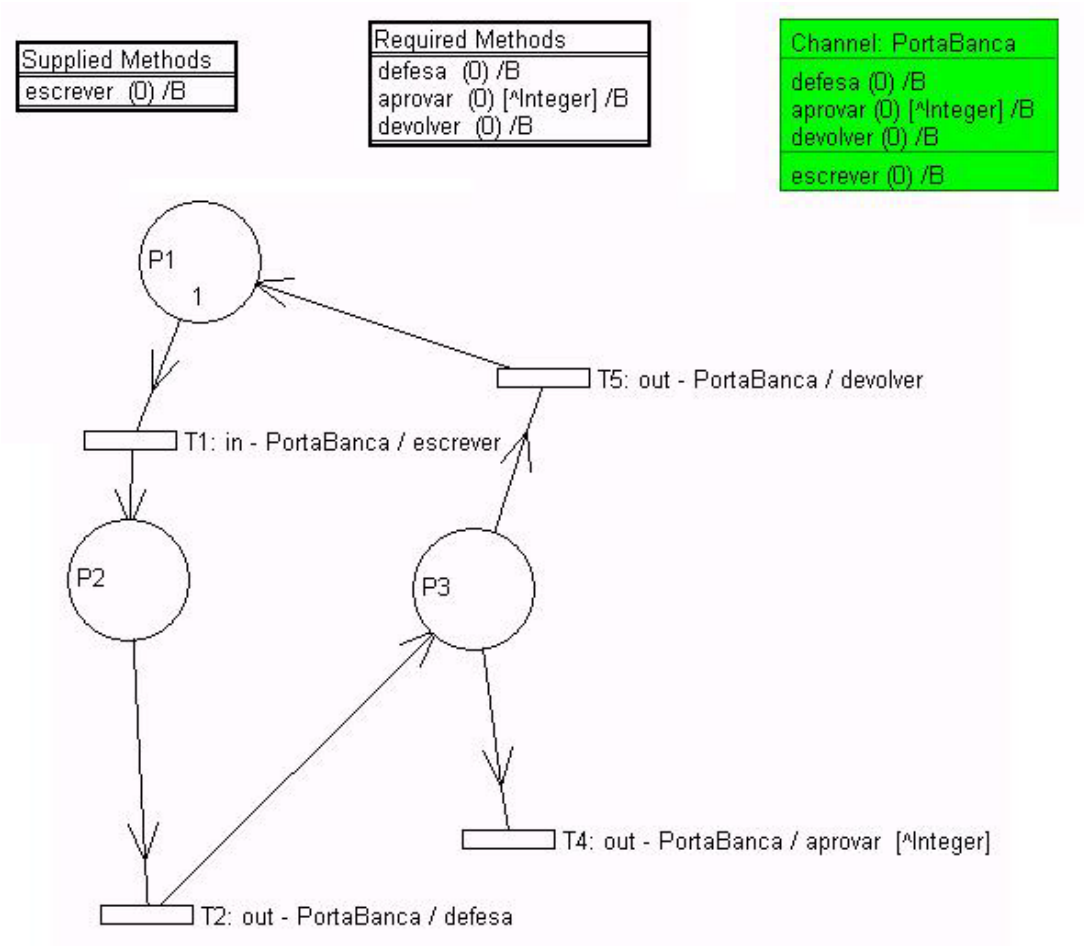

Figura 6.6 Estrutura e comportamento do componente Mestrando.

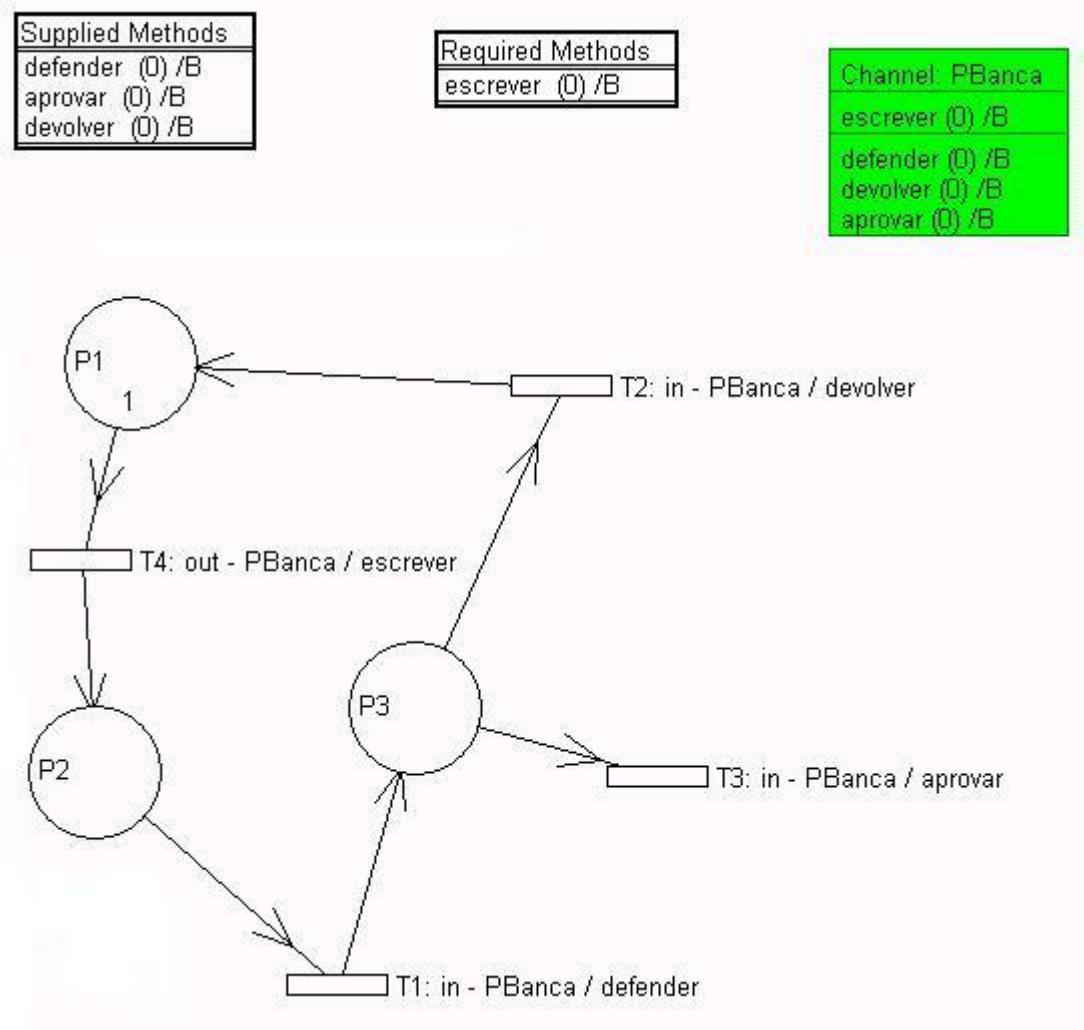

Figura 6.7 Estrutura e comportamento do componente Banca.

Como visto nas figuras 6.6 e 6.7, o componente Banca possui os serviços defender, devolver e aprovar supridos e o serviço escrever requerido.

Enquanto que o componente Mestrando possui o serviço escrever suprido e os serviços defesa, devolver e aprovar requeridos.

À primeira vista percebemos que o serviço requerido defender do componente Mestrando não é encontrado no componente Banca

Para executar a adaptação necessária para que estes dois componentes trabalhem juntos deve-se:

Chamar a ferramenta de adaptação dentro do ambiente SEA:

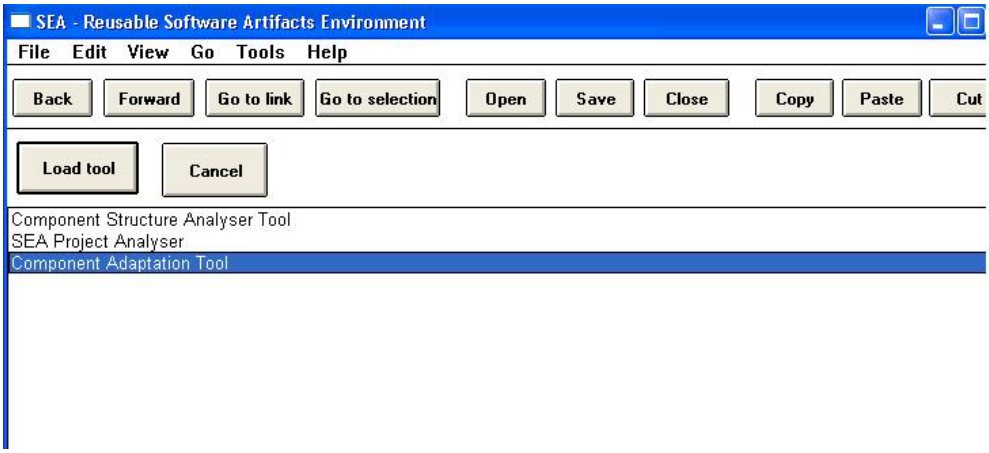

Figura 6.8 Ferramenta de adaptação.

Para se chegar à opção mostrada na figura 6.8 tem-se que acessar o modelo da arquitetura de componentes criada e selecionar o menu "*Tools*" → "*load tool*" então a lista de ferramentas mostrada na figura 6.8 aparecerá na tela. Ao selecionar a opção "*Component Adaptation Tool*" a tela da Figura 6.9 aparecerá para o desenvolvedor.

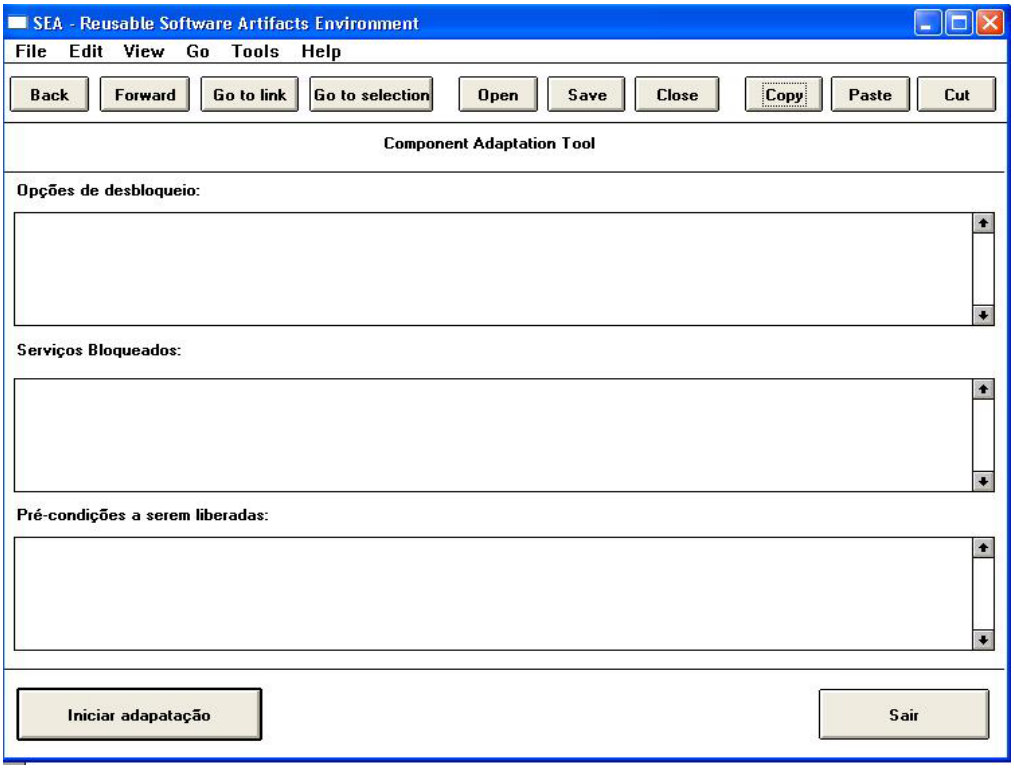

Figura 6. 9 Ferramenta de adaptação - Tela de inicialização.

Ao clicar no botão Iniciar Adaptação, mostrado na figura 6.9, a ferramenta inicializa o processo de geração do adaptador, tanto na parte estrutural como na parte comportamental, ou seja, teremos a interface do componente, bem como a rede de Petri do mesmo.

A Estrutura do adaptador gerado é:

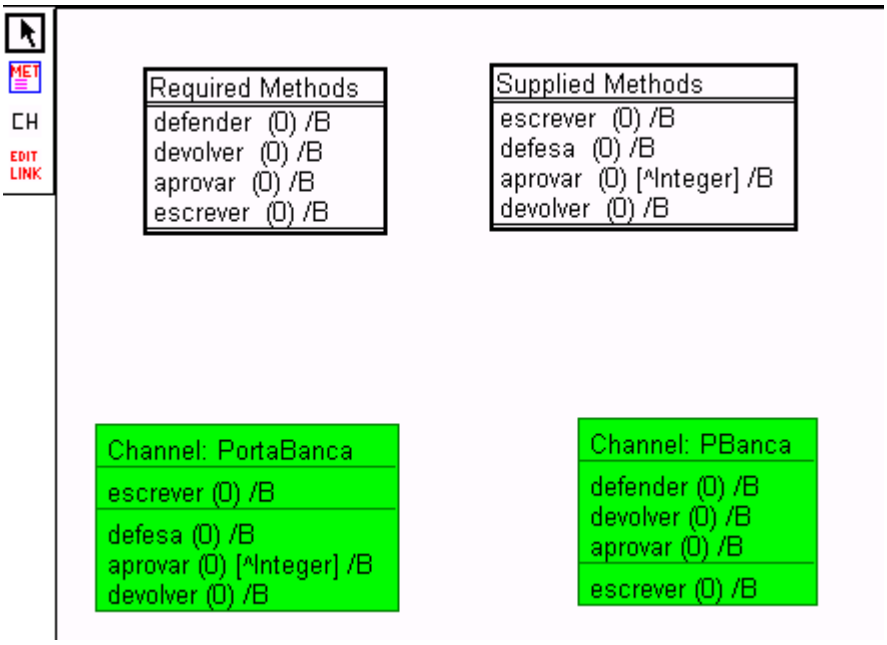

Figura 6.10 Estrutura do adaptador gerado.

O Adaptador tem agora duas portas, uma para interagir com o componente Banca e outra para interagir com o componente Mestrando, conforme passo 1.1 do algoritmo de criação da estrutura da cola:

1.1 Criar a especificação do componente Cola com duas portas P1 e P2;

O Adaptador deve possuir todos os serviços requeridos e supridos nos dois componentes adaptados para poder desviar as chamadas entre os dois, seguindo-se os passos 1.1 e 1.2 do algoritmo apresentado anteriormente:

1.2 Ler os serviços das portas de origem e destino envolvidas;

1.2.1 Para origem e destino faça: inserir na porta P1/origem e P2/destino os serviços com direção contrária (requerido=suprido e suprido=requerido).

O comportamento gerado é o seguinte:

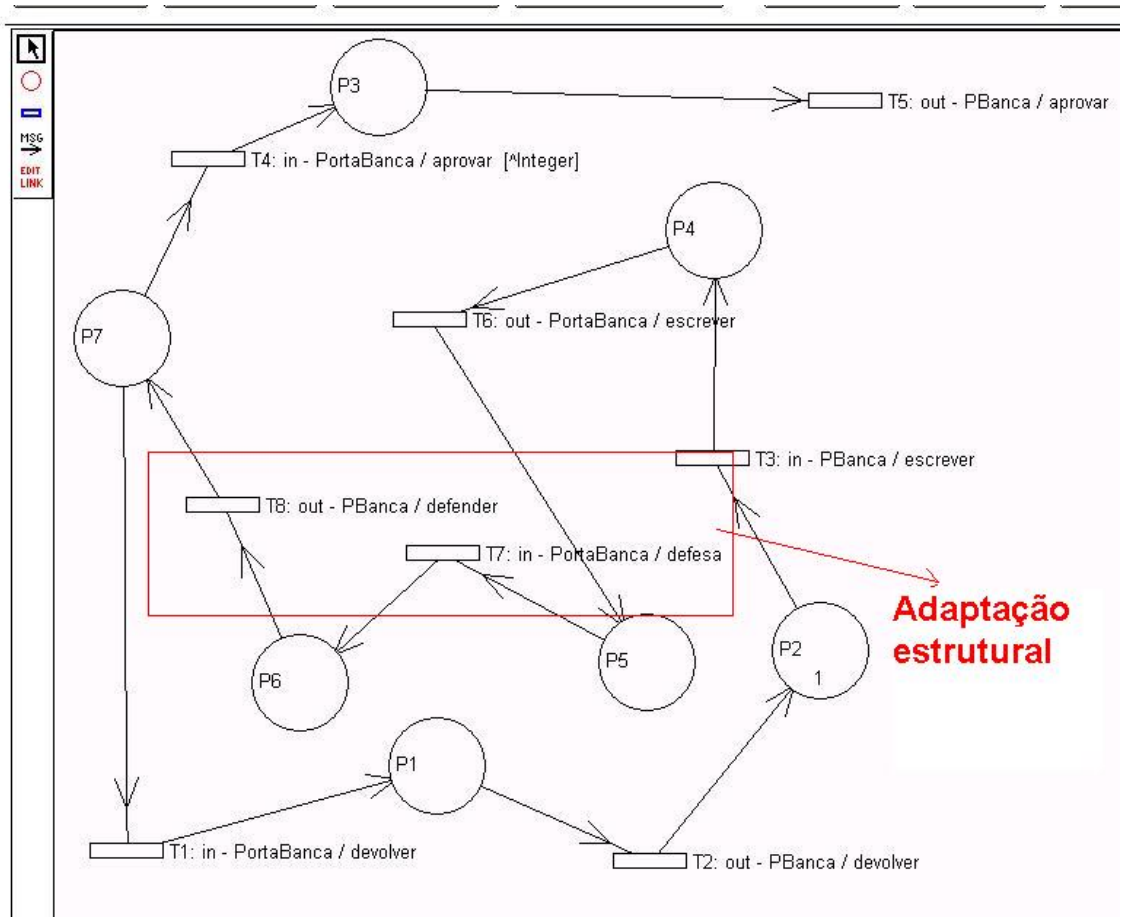

Figura 6.11 Rede de Petri do adaptador.

A rede de Petri mostrada na figura 6.11 mostra que a ordem das chamadas não foi alterada, apenas a adaptação do desvio da chamada do defesa/defender dentro do adaptador é feita, nas outras chamadas o adaptador apenas reencaminha para o mesmo serviço originalmente chamado, não alterando o comportamento da interação.

Seguindo o algoritmo:

2. Verificar, nas redes de Petri dos componentes adaptados, as restrições de ordem dos serviços;

Restrições encontradas no componente Mestrando: escreverM- defesaM, onde o "M" ao final do nome do serviço identifica que o serviço é do componente Mestrando.

Restrição encontrada no componente Mestrando: defesaM-devolverM, ou defesaM-aprovarM (^integer).

Restrição encontrada no componente Banca: escreverB-defenderB, onde “B” identifica o componente Banca.

Restrição encontrada no componente Banca: defenderB-devolverB, ou defenderB-aprovarB.

Verificar restrição de ordem associada às ligações da chamada original do componente requisitante Mestrando/Banca, pois ambos requerem serviços um do outro, com a chamada a ser feita dentro da Cola[40]:

Tabela 6.2 Tabela de restrições da Cola.

Índice	Restrição1	Restrição2	Restrição3	Restrição4
a	Mestrando	Cola	Cola	Banca
b	“defesa”	“defesa”	“defender”	“defender”
c	“aprovar(integer)”	“aprovar(integer)”	“aprovar”	“aprovar”
d	“devolver”	“devolver”	“devolver”	“devolver”
e	“escrever”	“escrever”	“escrever”	“escrever”

Assim fica definido que a ‘Restrição1 a’ antecede a ‘Restrição2 a’ e assim sucessivamente.

Arquitetura final com a presença do Adaptador:

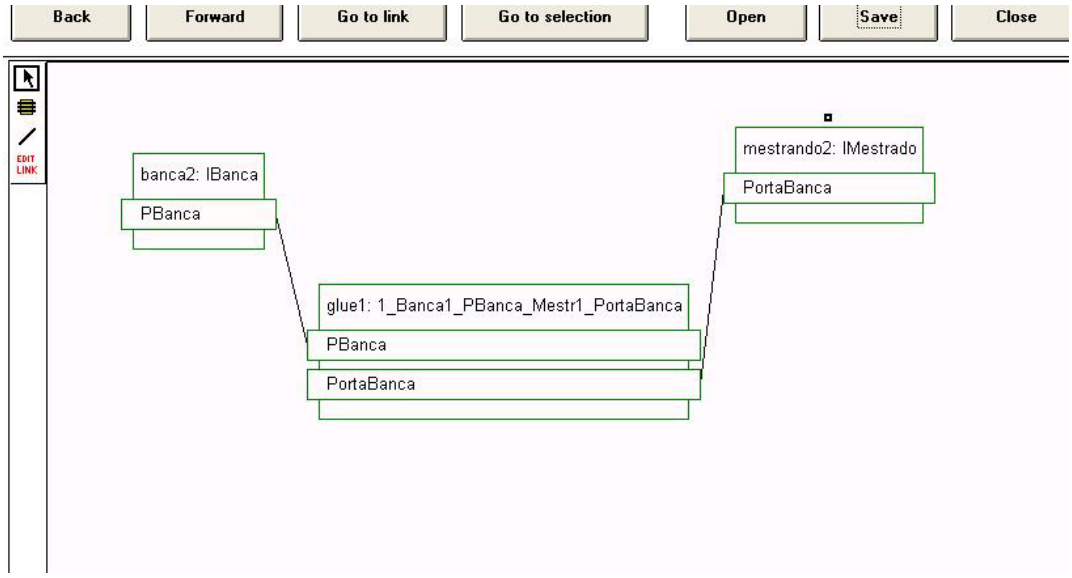

Figura 6.12 Arquitetura de componentes com a presença do adaptador.

[40] Isto implica que através da Cola, a chamada do serviço adaptado do componente Mestrando deve invocar imediatamente o serviço suprido pelo componente Banca e vice-versa, surgindo, desta maneira, a restrição de invocação entre estes serviços.

Neste passo o desenvolvedor deve retornar a arquitetura original, apagar o conector original e inserir o componente Adaptador criando os novos conectores. Os conectores estão em cor preta significando que não há incompatibilidade a ser resolvida.

Resultante final

Para obter a rede de Petri resultante final do sistema com a inclusão do Adaptador, o desenvolvedor deve selecionar a opção mostrada na figura 6.13, selecionando a opção de menu "*Tools*" → "*Load Tool*"

Figura 6.13 Gerando a rede de Petri resultante.

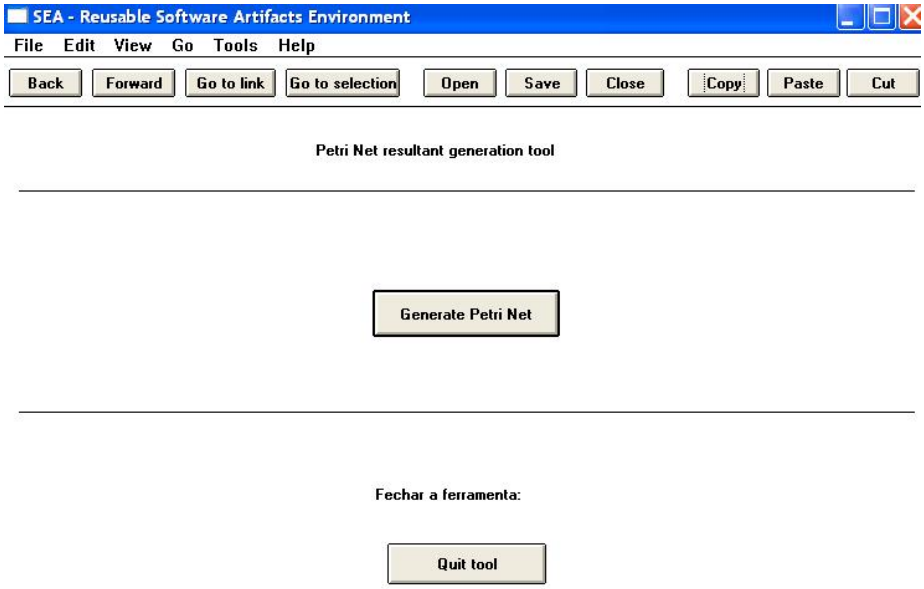

Figura 6.14 Tela de chamada das ferramentas dentro do SEA – ferramenta de geração da rede de Petri resultante.

Na figura 6.14 o desenvolvedor deverá selecionar "*Generate Peri Net*" para gerar a rede de Petri resultante, figura 6.15.

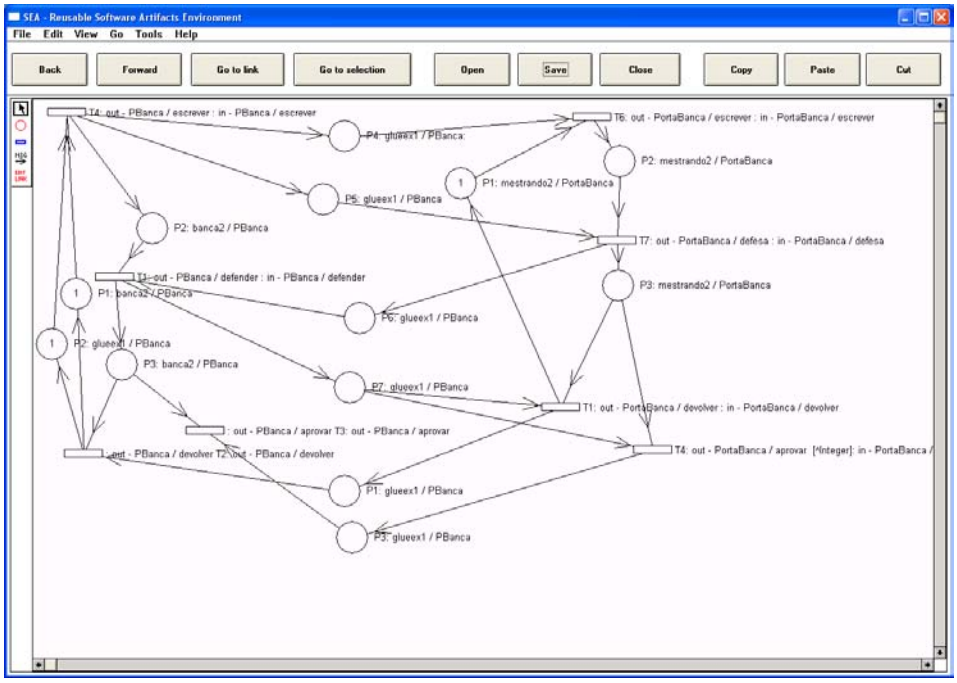

Figura 6.15 Rede de Petri resultante do exemplo.

Exemplo 2

Geração de adaptação comportamental para bloqueio de rede de Petri.

Dados os componentes:

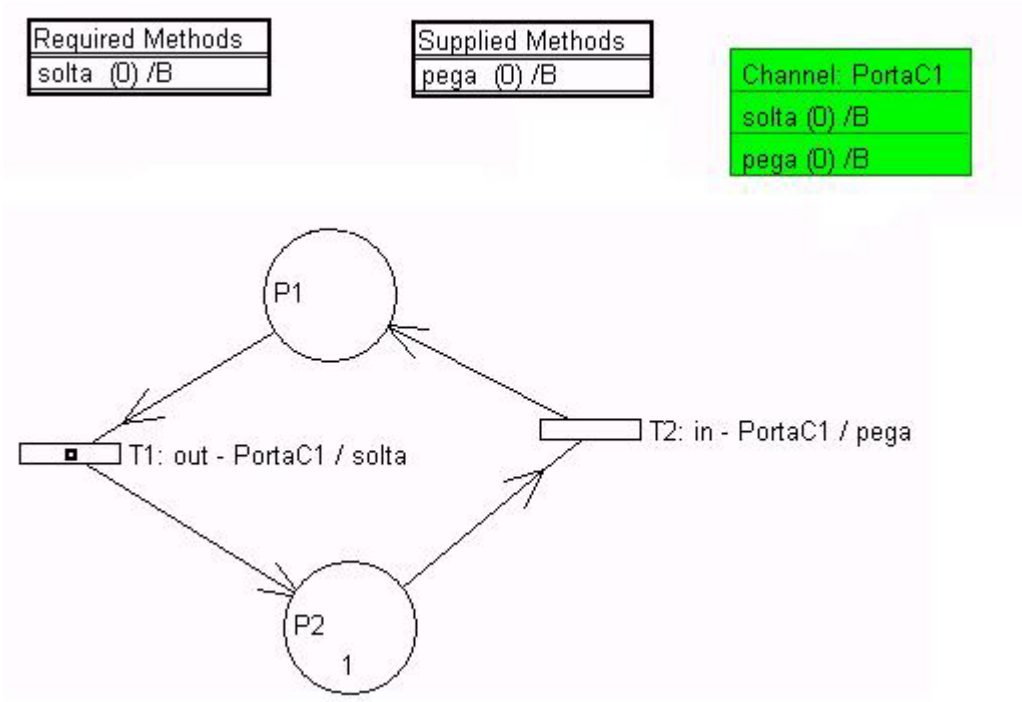

Figura 6.16 Estrutura e comportamento do componente C1

Na figura 6.16 tem-se a estrutura do componente C1 que possui os serviços solta, que é requerido e o serviço pega, que é suprido.

No comportamento mostrado pela rede de Petri da figura 6.16 o componente C1 espera a chamada de seu serviço suprido pega.

O componente C2, mostrado na figura 6.17, possui o serviço pegar, que é requerido e o serviço, que é suprido, e na representação de seu comportamento, visto na rede de Petri, ele também aguarda a chamada de seu método suprido pegar.

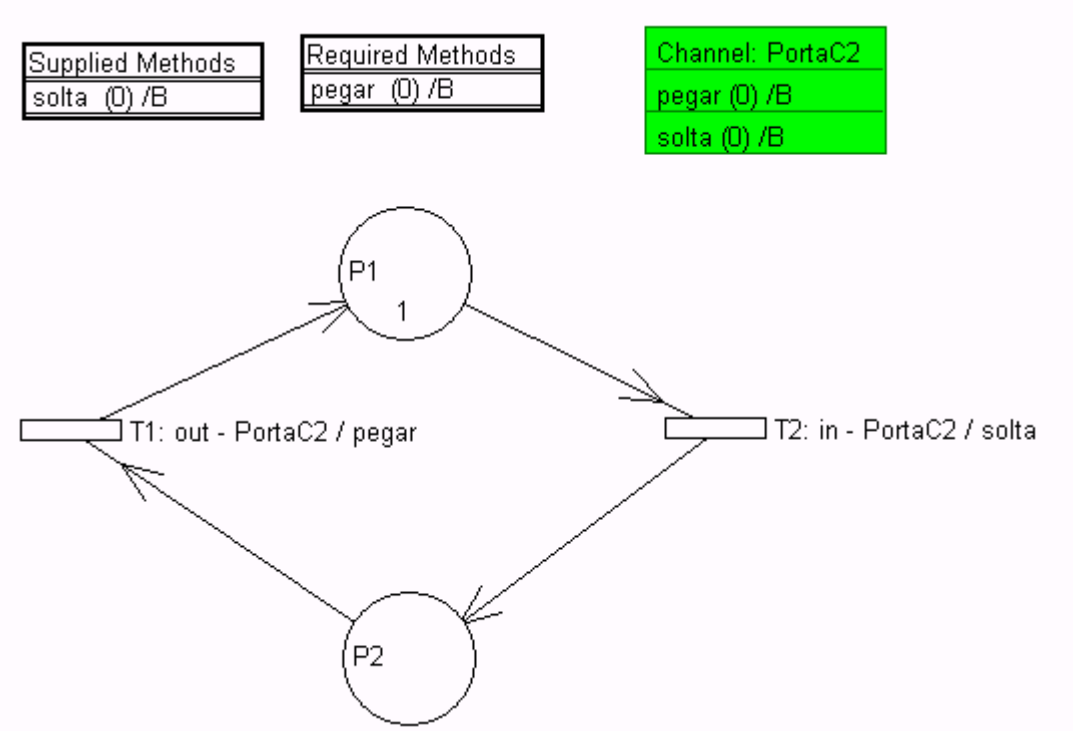

Figura 6.17 Estrutura e comportamento do componente C2.

Para realizar uma arquitetura onde estes dois componentes interajam devemos construir o diagrama da arquitetura no Editor de Componentes:

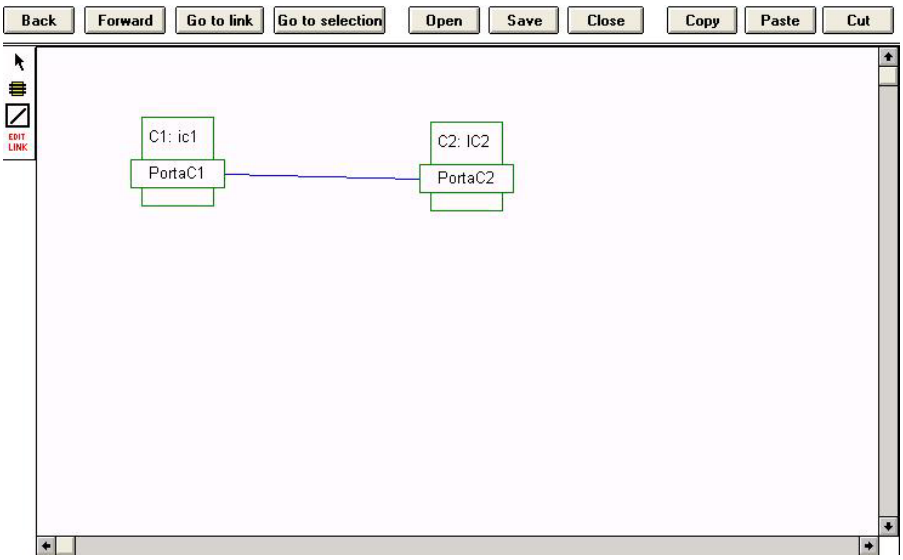

Figura 6.18 Arquitetura de componentes contendo C1 e C2.

Na figura 6.18 tem-se o conector em azul indicando que existe incompatibilidade.

Para resolver esta incompatibilidade a ferramenta criada por Cunha, 2005 sugere que selecione-se o conector clique sobre o botão "*Go to selection*" para se obter a descrição das incompatibilidades encontradas como mostra a figura 6.19.

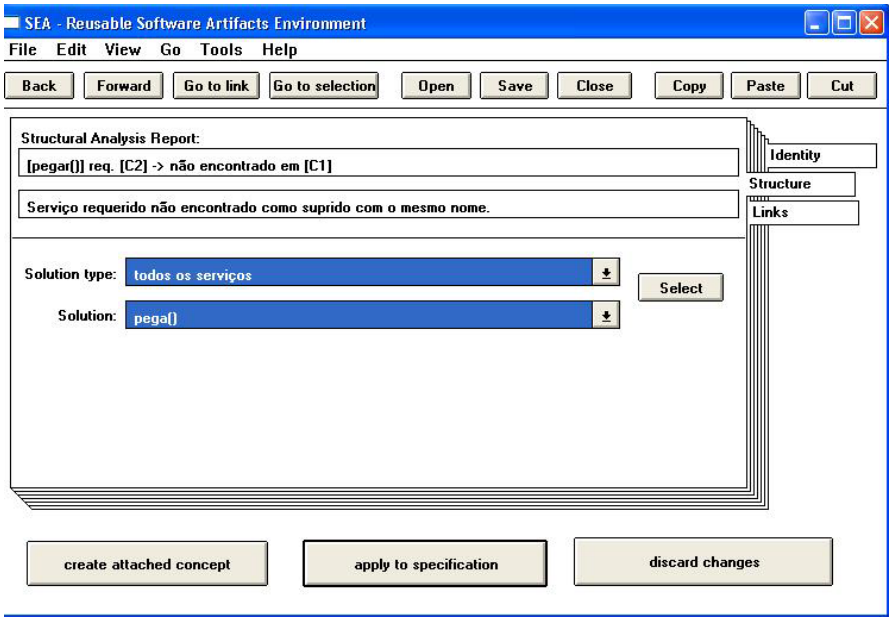

Figura 6.19 Incompatibilidade encontrada pela ferramenta desenvolvida por Cunha , 2005.

Na figura 6.18 tem-se a lista das incompatibilidades encontradas, o serviço requerido **pegar** não é encontrado no componente C1, a sugestão dada pela ferramenta é compatibilizar **pega** com **pegar,** clicando-se no botão "*Select"* a compatibilização é feita e o conector volta a ficar da cor preta, se todas as incompatibilidades estiverem resolvidas.

Depois deste passo, para encontrar as incompatibilidades comportamentais, deve-se realizar a rede de Petri resultante do modelo, para isso, seleciona-se a ferramenta de geração da rede de petri resultante desenvolvida em Cunha, 2005 que gera a rede mostrada na figura 6.20.

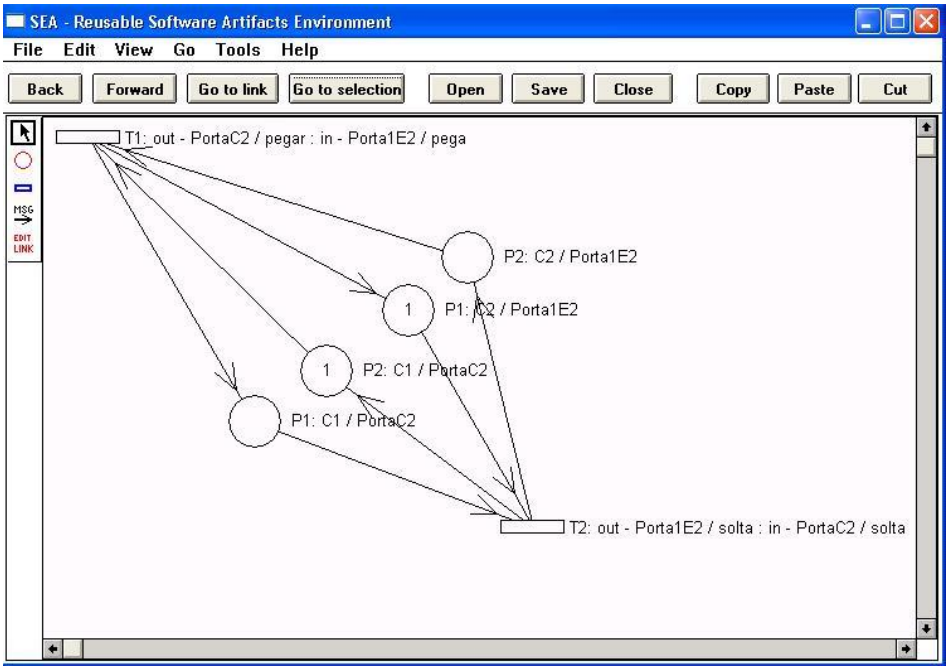

Figura 6.20 Rede de Petri resultante do Exemplo 2.

Na figura 6.20 tem-se a rede de Petri resultante do digrama de componentes do exemplo 2, nesta rede de Petri é fácil perceber que nenhuma transição está habilitada pois os lugares que levam a cada uma delas estão, um com ficha e outro sem, desta maneira nenhuma transição poderá ser disparada, um bloqueio acontecerá. Para eliminar o bloqueio a ferramenta de compatibilização desenvolvida por Cunha, 2005 gera um relatório que será usado pela ferramenta de adaptação desenvolvida neste trabalho, como mostra a figura 6.21.

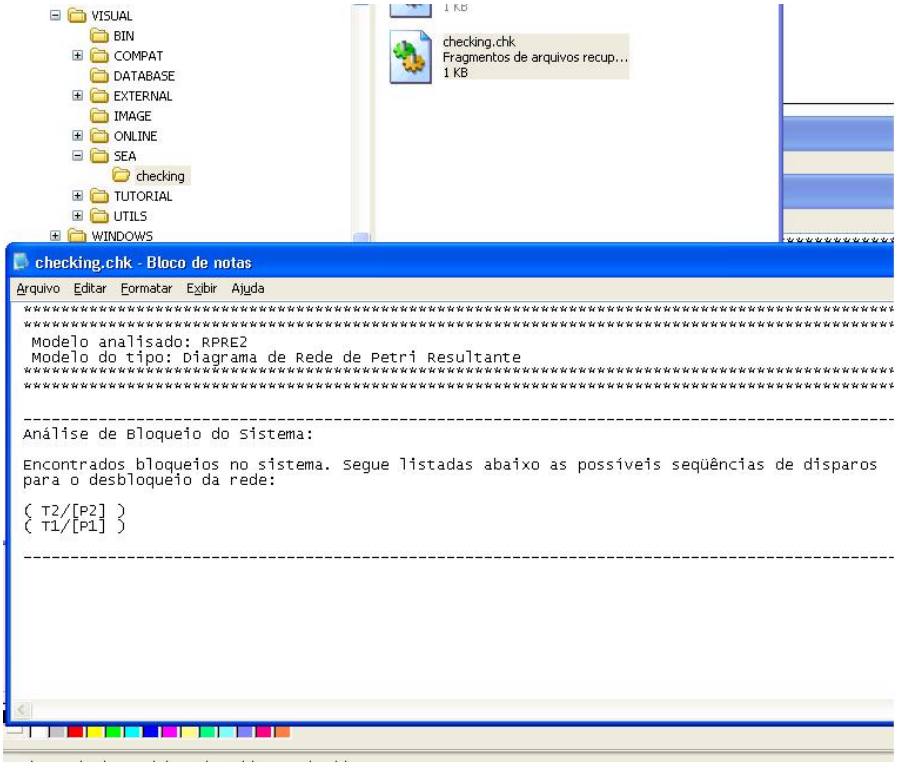

Figura 6.21 Relatório de compatibilização comportamental.

Na figura 6.21 tem-se o relatório que é usado pela ferramenta de adaptação para criar o adaptador comportamental. Neste relatório é mostrado todo o bloqueio encontrado no sistema. Além deste relatório a ferramenta de análise de incompatibilidade cria uma série de objetos que são usados para verificar a incompatibilidade encontrada.

Para gerar o adaptador deve-se selecionar a ferramenta de geração "*Component Adaptation Tool*", como mostra a figura 6.22.

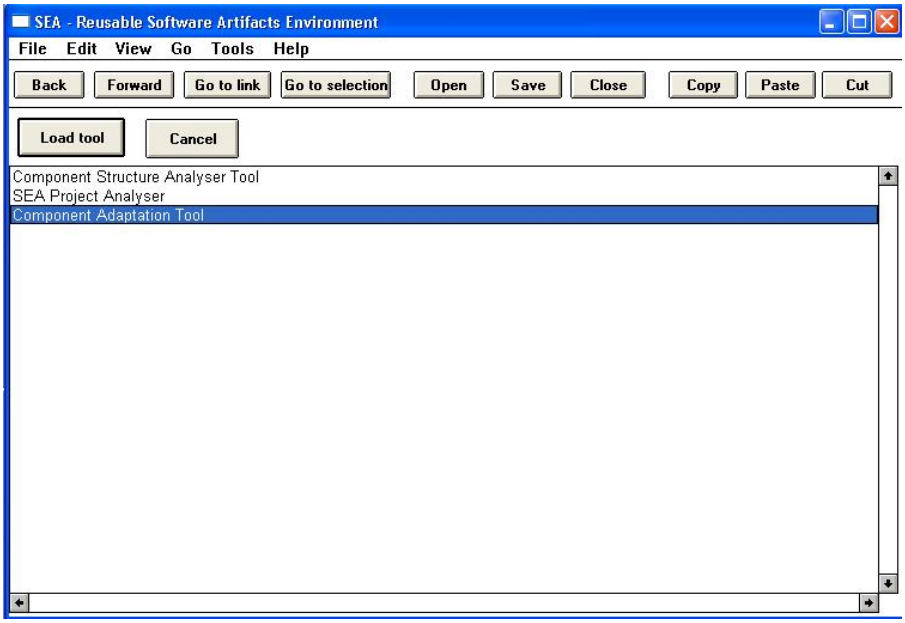

Figura 6.22 Acesso à ferramenta de geração de Adaptação.

Na figura 6.22 tem-se a seleção da ferramenta de adaptação, esta ferramenta gera uma interface de componente que resolve as incompatibilidades encontradas pela ferramenta de compatibilização, isto é, este adaptador permitirá que os componentes antes incompatíveis agora possam interagir através da adaptação.

Ao selecionar esta opção a janela 6.23 aparecerá na tela.

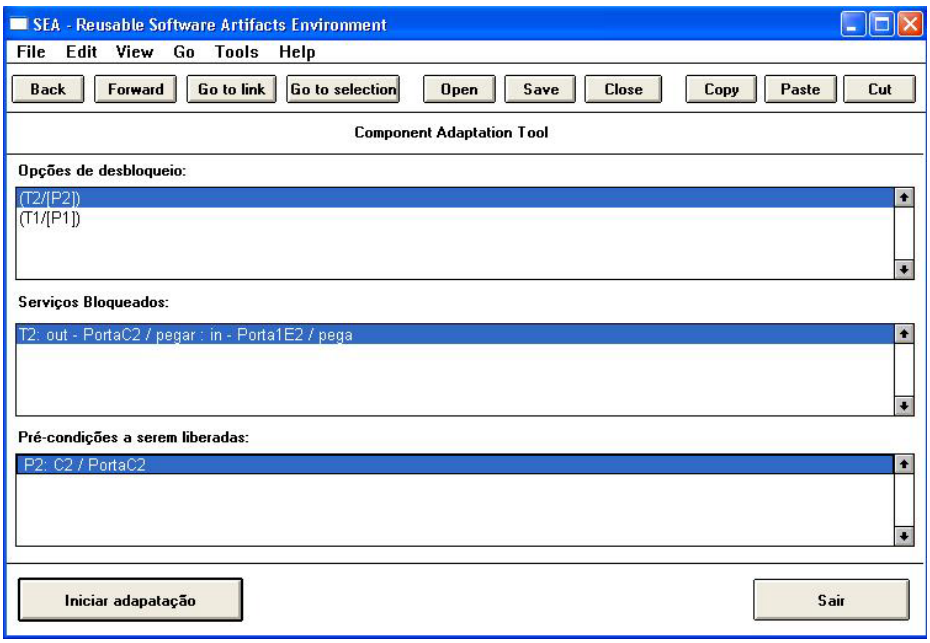

Figura 6.23 Ferramenta de adaptação.

Na figura 6.23 tem-se acesso à ferramenta de geração da adaptação. A partir desta seleção a ferramenta gera, segundo os passos dos algoritmos mostrados, a estrutura e o comportamento do componente adaptador.

A estrutura do componente adaptador gerado é pode ser vista na figura 6.24.

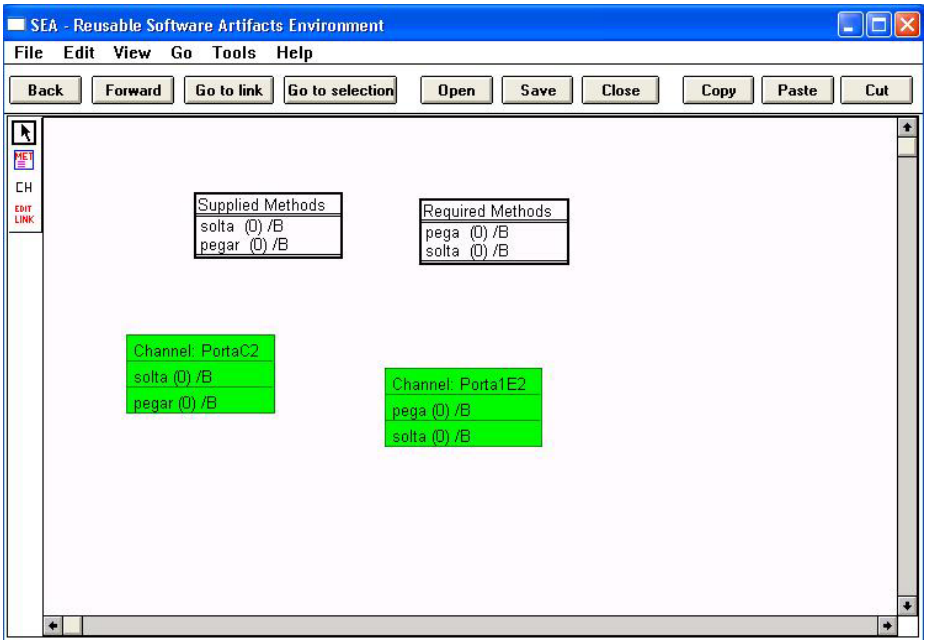

Figura 6.24 Estrutura do adaptador gerado pela ferramenta de adaptação.

Na figura 6.24 tem-se a estrutura do componente adaptador. Ele possui todos os serviços requeridos e supridos pelos componentes a serem adaptados e duas portas para servir de ponte entre os dois componentes, numa porta o adaptador se conecta ao componente C1 e pela outra porta o componente se comunica com C2.

Desta maneira os componentes não se comunicarão mais entre si e sim através do adaptador que executará os desvios necessários para a adaptação. Por exemplo: ao ser requerido o serviço **pegar** pelo componente C1 ele irá encontrar o respectivo serviço suprido no adaptador que desvia a chamada para o serviço **pega** de C2, realizando a adaptação estrutural. Para resolver o bloqueio existente a ferramenta seguiu os passos do algoritmo e inseriu uma ficha no lugar que antecede uma das transições bloqueadas habilitando seu disparo.

Na figura 6.25 vemos a rede de Petri do componente adaptador gerado.

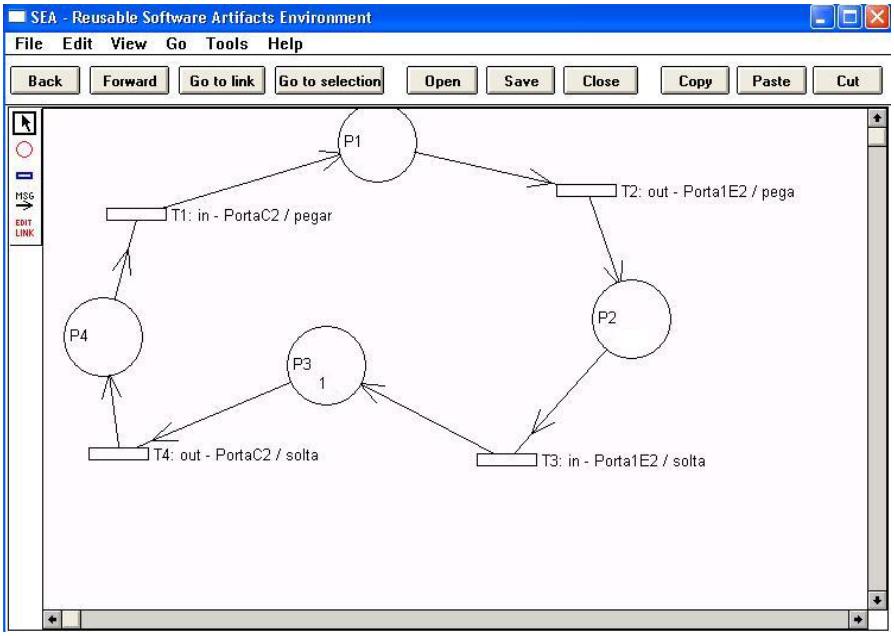

Figura 6.25 Rede de Petri do adaptador gerado pela Ferramenta de Adaptação.

A rede de Petri do adaptador possui todos os serviços requeridos e supridos pelos componentes a serem adaptados e que respeita a ordem de chamada **requerido-suprido** que é uma regra para manter o comportamento anterior da interação dos dois componentes adaptados. Além de ter trocado dentro do adaptador a ordem de chamada, ou seja, tudo que era requerido pelo componente C1 ficou suprido pelo adaptador e tudo que era suprido em C1 se torna requerido no adaptador, valendo esta regra para o componente C2, desta maneira os serviços de um lado da interação C1 com o adaptador atende todos os serviços requeridos e supridos do componente C1 e por outro lado na interação com C2 o adaptador realiza a mesma tarefa. Dentro do adaptador os desvios para chamar o componente que realimente realiza o serviço é transparente para o componente C1 ou C2 envolvido na comunicação.

A posição da ficha na figura 6.25 mostra que o serviço **solta** suprido pelo componente C2 é requerido pelo componente adaptador; como em ambas as redes de Petri este serviço esta habilitado o bloqueio termina. Desta maneira o sistema volta a funcionar

Para construir o novo modelo de componentes, agora com o adaptador, o desenvolvedor deverá inserir a nova interface criada dentro do modelo anterior e alterar as conexões para utilizarem o adaptador como mostra a figura 6.26.

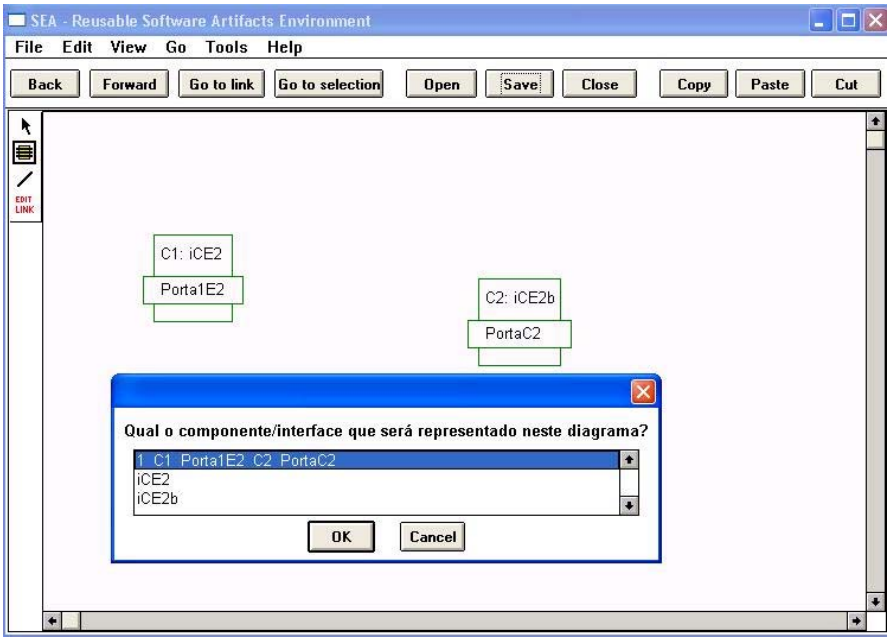

Figura 6.26 Seleção do componente adaptador para o modelo.

Na figura 6.26 o desenvolvedor faz a seleção do componente adaptador criado pela ferramenta de adaptação, a interface do componente fica disponível como qualquer outro componente já desenvolvido e poderá ser usado na especificação de novas aplicações.

A nova especificação fica como mostra a figura 6.27.

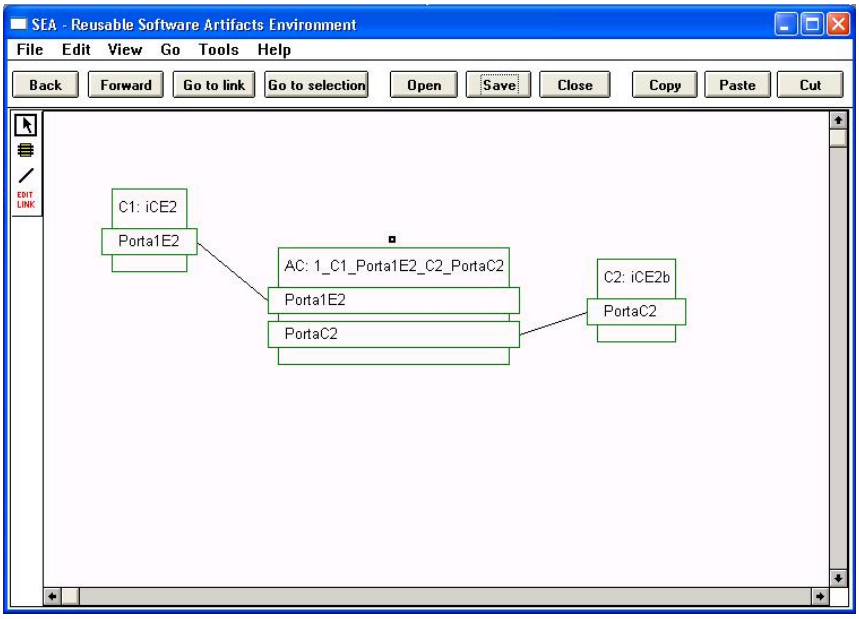

Figura 6.27 Novo diagrama de componentes agora com o Adaptador.

Na figura 6.27 tem-se a inclusão do adaptador na especificação anterior, note que os conectores estão marcados em preto demonstrando que não há incompatibilidades entre eles.

O último passo para ter certeza do que foi feito deve-se acessar novamente a ferramenta de geração de rede de Petri resultante para certificar que não haverá bloqueios na rede com a inclusão do adaptador.

A rede de Petri resultante é mostrada na figura 6.28.

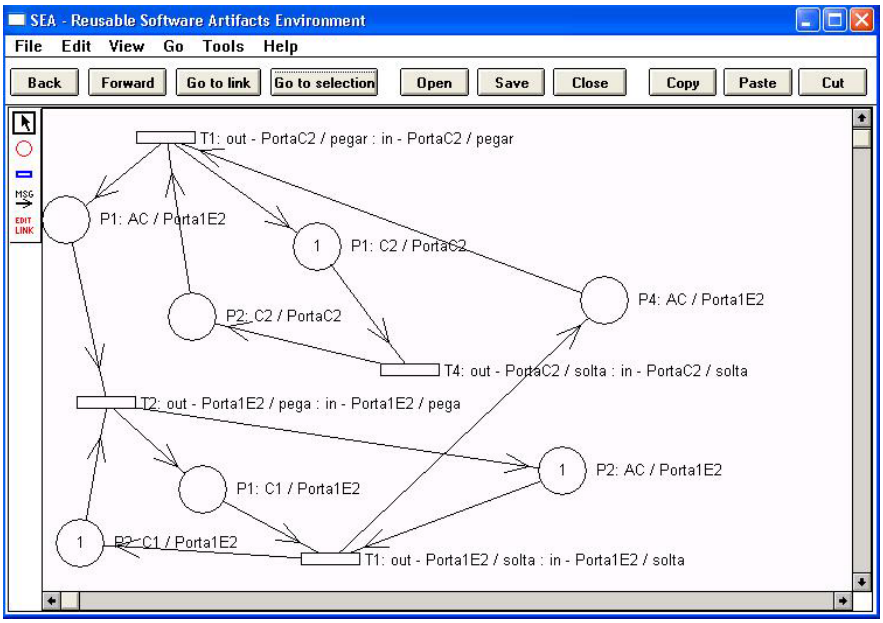

Figura 6.28 Rede de Petri resultante do exemplo 2.

A rede resultante vista na figura 6.28 deve ser analisada para que a ferramenta avise se encontrou ou não bloqueios. Pode ser fácil perceber apenas olhando que o serviço **solta** está habilitado, porém apenas o algoritmo poderá dizer se na evolução da rede encontrará algum bloqueio futuro.

O procedimento para criar o adaptador é sempre este dentro do ambiente SEA, a ferramenta de análise de incompatibilidade criada por Cunha, 2005 foi essencial para que a ferramenta de geração de adaptadores pudesse ser desenvolvida neste trabalho, pois a partir das análises feita por ela a ferramenta de geração semi-automatizada de adaptadores pode gerar a interface do componente adaptador que resolve a incompatibilidades encontradas.

7 Conclusões

O principal objetivo deste trabalho foi se concentrar sobre um aspecto do desenvolvimento de sistemas baseados em componentes (DSBC) que envolve adaptação de componentes, para isso foram levantadas informações sobre esta etapa no DSBC, e ainda os tipos de adaptações possíveis.

Como a abordagem de DSBC prevê a existência de componentes, previamente desenvolvidos, este adaptador gerado deveria ficar disponível também para ser utilizado posteriormente, surgiu a necessidade de criar uma ferramenta que gerasse uma adaptação reusável, ou seja, que ficasse a disposição do desenvolvedor.

Assim, utilizando as informações disponibilizadas na interface e apenas nela, o usuário do componente deve ser capaz de integrar/adaptar este componente na sua aplicação, sem muitas dificuldades, pois disso depende o sucesso do DSBC (TRUYEN 2000).

Atingir este objetivo depende de vários aspectos, como alguns dos levantados neste trabalho: uso de descrições precisas na descrição das interfaces, utilização de uma visão arquitetural para a integração, existência de suporte automatizado por um ambiente de DSBC, entre outros.

A seguir são apresentados os resultados obtidos nesta pesquisa, as limitações da solução encontrada e os trabalhos futuros que podem ser realizados.

Principais resultados

A realização desta pesquisa demonstrou, na prática, que interfaces definidas estruturalmente e comportamentalmente, com regras precisas, são fundamentais para permitir a geração de ferramentas para disponibilizar adaptadores que resolvam um conjunto limitado de incompatibilidades.

A pesquisa resultou no desenvolvimento de um ambiente gráfico, usado e implementado em parceria com o trabalho de (CUNHA 2005), para conter e permitir a visualização e manipulação de componentes de maneira diferenciada, sem usar notação UML, já que componentes podem ser desenvolvidos sob qualquer paradigma. Neste sentido, o Editor ampliou a visão de componentes utilizada na versão anterior do ambiente SEA.

A partir de interfaces definidas com certo rigor, como é o caso do padrão de projeto implementado no SEA (SILVA 2000), foi possível implementar soluções para as incompatibilidades encontradas por ferramentas de análise (CUNHA 2005) durante a integração de componentes. Com base nos dados obtidos nestas análises foi possível gerar dentro do ambiente SEA, especificações de interface para componentes adaptadores (cola).

A pesquisa conseguiu demonstrar que é possível automatizar parcialmente a construção de colas, através do desenvolvimento e teste de algoritmos, e isto é um passo importante na diminuição do esforço feito na adaptação de componentes contribuindo para a abordagem DSBC.

Limitações

A tentativa de automatizar parte da especificação de sistemas baseados em componentes ficará restrita a um nível de especificação de interface, ou seja, os componentes adaptadores gerados contêm as descrições de sua interface estrutural e comportamental. Nestas interfaces ficam sugeridas as funcionalidades necessárias, porém a implementação fica a cargo do desenvolvedor.

A adaptação estrutural é levada a cabo pela construção de um adaptador que faz a ponte entre dois componentes. A criação da especificação das interfaces estrutural e comportamental é automática, porém o desenvolvedor terá que criar o *framework* e o componente dentro do SEA, para então inseri-lo na arquitetura da aplicação que está desenvolvendo.

A adaptação comportamental resolve bloqueios encontrados através da colocação de fichas em lugares que habilitam as transições desejadas, porém não avalia se esta inserção afeta outras qualidades da rede resultante como, por exemplo, se a

limitação da rede foi alterada, se antes da adaptação a rede era salva e depois perdeu esta propriedade etc.

No capítulo do estado da arte, foi visto que existe muita preocupação sobre adaptar componentes durante a execução da aplicação, porém esta visão não levada em conta na implementação da ferramenta.

O Adaptador não infere nenhuma lógica sobre funcionalidade e nem sobre a possibilidade de utilizar esta composição para outros casos, a solução, apesar de usar um algoritmo genérico, é específica.

Trabalhos futuros

Restrições baseadas em arquiteturas de aplicações não são implementadas na ferramenta, porém a preocupação com arquiteturas pré-definidas, tanto para conectar componentes quanto para gerar os adaptadores, poderia ser uma característica futura desta ferramenta.

Verificar se algumas das análises feitas pela ferramenta de verificação de incompatibilidade comportamental (CUNHA 2005), além do bloqueio, poderiam ser resolvidas com adaptadores, como por exemplo, limitabilidade de fichas na evolução da rede de Petri resultante.

Gerar aromaticamente o componente e não apenas a especificação da interface

A geração de código dentro dos adaptadores.

Inserir os adaptadores automaticamente na arquitetura sem a intervenção do desenvolvedor.

Considerações finais

A principal questão levantada nesta pesquisa diz respeito à possibilidade de automatizar o esforço de adaptação de componentes, levando à necessidade de buscar algoritmos capazes de fazê-lo.

A resposta não é simples e veio junto com uma série de fatores que envolvem o DSBC, como por exemplo, o que são componentes, e, além disso, como usar os componentes para criar um novo artefato de software.

Como a interface é o único meio de comunicação para conhecer e usar o componente, a sua definição e manipulação se tornam fundamentais para esta abordagem.

Segundo (BOSCH 1997), é muito difícil usar componentes assim como eles foram disponibilizados. Considerando esta afirmação, a adaptação surge como uma facilitadora da abordagem DSBC, viabilizando o reuso, ao permitir a utilização de componentes mesmo em condições em que, a priori, isto seria impossível.

Finalmente, a disponibilização de uma ferramenta para adaptar componentes dentro de um ambiente de desenvolvimento colabora com a abordagem DSBC quando ela minimiza o esforço da adaptação, possibilita o reuso real incentivando o avanço na utilização desta abordagem, tanto para o desenvolvimento de novos componentes para serem utilizados quanto para desenvolvedores de novas aplicações usando componentes.

Referências bibliográficas

ALENCAR, P.S.C.; COWAN, D.D.; LUCENA, C.J.P. **A Model for Gluing Components**. Terceiro Workshop Internacional sobre Programação Orientada a Componentes (WCOP'98). Bruxelas, Bélgica, 1998.

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core J2EE Patterns - As Melhores Estratégias e Práticas de Design**. Rio de Janeiro: Editora Campus, 2002.

BEACH, Brian W. **Connecting Software Components with Declarative Glue**. Proceedings of the 14th international Conference on Software Engineering. Melbourne, Australia, Pgs: 120 – 137, Maio, 1992.

IBM Bean Markup Language, 2004. Em: http://www.alphaworks.ibm.com/formula/bml Acesso em: 09 fev. 2005.

BACHMANN, Felix; BASS, Len; BUHMAN, Charles; et al. **Technical Concepts of Componet-Based Software Engineering**. 2nd Edition, Pittsburgh, PA, May, 2000.

BOSCH, Jan. **Adapting Object-Oriented Components**. University of Karlskrona/Ronneby. Proceedings of the 2nd Workshop on Component-Oriented Programming (ECOOP'97). Jyväskylä, Finland, June, 1997.

BOSCH, Jan. **Summary of the Second International Workshop on Component-Oriented Programming**. International Workshop on Component-Oriented Programming – WCOP, 1997. Jyväskylä, Finland, June, 1997.

CUNHA, Roberto da Cunha. **Suporte à Análise de Compatibilidade Comportamental e Estrutural entre Componentes no Ambiente SEA**. Dissertação de Mestrado, Florianópolis, UFSC 2005.

DIAS, Marcio; VIEIRA, Marlon E. R. **Software Architecture Analyses Based on State Chart Semantics**. Proceedings of the 10th International Workshop on Software and Design, IEEE, San Diego, CA, 2000.

DONG, Jing; ALENCAR, Paulo S.C.; COWAN, Donald D. **Correct Composition of Design Components**. Quarto Workshop Internacional sobre Programação Orientada a Componentes (WCOP'99). Lisboa, Portugal, 1999.

DONG, Jing. **Integration in Component-Based Software Development Using Design Patterns**. Arizona University. Phd Thesis, Dec 2000.

GAMMA, Erich; HELM, Richard; JONHSON, Ralph; et. al. **Design Patterns – Elements of Reusable Object-Oriented Software**. Massachusetts: Addison-Wesley, 1995.

HONDT, Koen De; LUCAS, Carine; STEYAERT, Patrick. **Reuse Contracts as Component Interface Descriptions**. Proceedings of the 2nd Workshop on Component-Oriented Programming (ECOOP'97). Jyväskylä, Finland, June, 1997.

(IBM 2004) CORBA Component Model (CCM), Introducing next-generation CORBA. em: http://www-128.ibm.com/developerworks/webservices/library/co-cjct6/
Acesso em: 09 fev. 2005.

INVERARDI, Paola; TÍVOLI, Massimo. **The Role of Architecture in Component Assembly**. Sétimo Workshop Internacional sobre Programação Orientada a Componentes (WCOP'02). Malaga, Spain, 2002.

JARIR, Zahi; DAVID, Pierre-Charles; LEDOUX, Thomas. **Dynamic Adaptability of Services**. Em: Enterprise JavaBeans. Sétimo Workshop Internacional sobre Programação Orientada a Componentes (WCOP'02). Malaga, Spain, 2002.

KELLER, Ralph; HÖLZLE, Urs. **Late Component Adaptation**. Terceiro Workshop Internacional sobre Programação Orientada a Componentes (WCOP'98). Bruxelas, Bélgica, 1998.

KNIESEL, Gunter. **Type-Safe Delegation for Run-Time Component Adaptation**. Workshop on Component-Oriented Programming (ECOOP '99). 13th European Conference, Lisbon, Portugal, 1999.

LEE, Seung-Yun; KNOW, OhCheon; KIM, Min-Jung; et. al. **Research on MDA Based COP Approach**. Oitavo Workshop Internacional sobre Programação Orientada a Componentes (WCOP'03). Darmstadt, Germany, 2003.

LUCAS, Carine. **Documenting Reuse an Evolution with Reuse Contracts**. PhD Thesis. Department of Computer Science, Vrije Universiteit, Brussel, 1997.

MCGURREN, Finbar; CONROY, Damien. **X-Adapt: An Architecture for Dynamic Systems**. Sétimo Workshop Internacional sobre Programação Orientada a Componentes (WCOP'02). Malaga, Spain, 2002.

RAJ, Gopalan SureSh. **A Detailed Comparison of CORBA, DCOM an Java/RMI**. Em: http://my.execpc.com/~gopalan/misc/compare.html Acesso em: 09 fev. 2005.

SAMETINGER, Johannes. **Software Engineering with Reusable Components**. Springer-Verlag. 1997. Pág. 271.

SCHNEIDER, Jean-Guy. **Components, Scripts, and Glue:** A Conceptual Framework for Software Composition. Phd Thesis. Universidade de Bern, Outubro de 1999.

SHAW, Mary; GARLAN, David. **Software Architecture:** Perspectives on a Emerging Discipline. Prentice Hall, Abril, 1996.

SILVA, Ricardo Pereira. **Suporte ao Desenvolvimento e Uso de Frameworks e Componentes**. PhD Thesis. Porto Alegre, UFRGS/II/PPGC, março 2000.

SILVA, Ricardo Pereira; PRICE, R. T. **Component Interface Pattern**. Em: Proceedings of 9th Conference on Pattern Language of Programs 2002 (PLOP 2002), Monticello, Setembro, 2002.

SZYPERSKY, Clemens; GRUNTZ, Dominik; MURER, Stepahn. **Component Software Beyond Object-Oriented Programming**. New York: Addison–Wesley, 1997. Segunda Edição. Pág: 231 a 340.

TRUYEN, Eddy; JOERGENSEN, Bo Noerregaard; JOOSEN, Wouter; et. al. **On Interaction Refinement in Middleware**. Quinto Workshop Internacional sobre Programação Orientada a Componentes (WCOP'00). Cannes, France, 2000.

WELCH, Ian; STROUT, Robert. **Adaptation of Connectors in Software Architectures**. Encontrado em: http://www.cs.ncl.ac.uk, WCOP 98.
Acesso em: 09 fev. 2005.

YELLIN, Daniel M.; STROM, Robert E. **Interfaces, Protocols, and the Semi-Automatic Construction of Adaptors**. ACM SIGPLAN Notices 1994. Pág. 176 a 190.