144 p

Um Banco de Dados Espaço-Temporal para Desenvolvimento de Aplicações em Sistemas de Informação Geográfica

Glaucia Faria

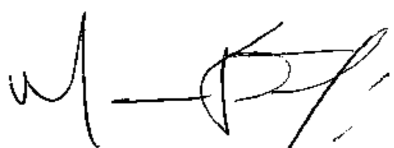
Dissertação de Mestrado

Um Banco de Dados Espaço-Temporal para Desenvolvimento de Aplicações em Sistemas de Informação Geográfica

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Glaucia Faria e aprovada pela Banca Examinadora.

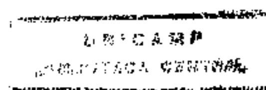
Campinas, 19 de março de 1998.

Claudia M. Bauzer Medeiros (Orientadora)



Mario A. Nascimento (Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.


UNICAMP
BIBLIOTECA CENTRAL

Instituto de Computação
Universidade Estadual de Campinas

Um Banco de Dados Espaço-Temporal para Desenvolvimento de Aplicações em Sistemas de Informação Geográfica

Glaucia Faria[1]

Fevereiro de 1998

Banca Examinadora:

- Claudia M. Bauzer Medeiros (Orientadora)
- Guido C. S. Araujo
  Instituto de Computação - Universidade Estadual de Campinas
- Nina Edelweiss
  Instituto de Informática - Universidade Federal do Rio Grande do Sul
- João C. Setubal (Suplente)
  Instituto de Computação - Universidade Estadual de Campinas

[1] A autora é bacharel em Ciências da Computação pela Universidade Federal de Goiás

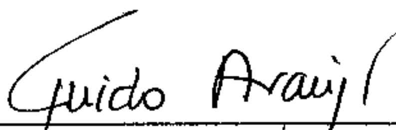
FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

	Faria, Glaucia
F225b	Um banco de dados espaço-temporal para desenvolvimento de aplicações em sistemas de informação geográfica / Glaucia Faria -- Campinas, [S.P. :s.n.], 1998.
	Orientadores:Claudia M.Bauzer Medeiros, Mario A.Nascimento
	Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.
	1. Banco de dados temporais. 2. Banco de dados orientados a objetos. 3. Sistemas de informação geográfica. I. Medeiros, Claudia Maria Bauzer. II. Nascimento, Mario Antonio do. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

© Glaucia Faria, 1998.
Todos os direitos reservados.

Tese de Mestrado defendida e aprovada em 19 de março de 1998 pela Banca Examinadora composta pelos Professores Doutores

Profª. Drª. Nina K. Edelweiss



Prof. Dr. Guido Costa Souza de Araújo

Profª. Drª. Claudia Maria Bauzer Medeiros

Aos meus pais, Manoel e Justina.

Agradecimentos

A Deus, por estar ao meu lado em todos os momentos.

À minha orientadora, Claudia Bauzer Medeiros, e ao meu co-orientador, Mario Nascimento, pela confiança, incentivo, amizade, dedicação, preciosos ensinamentos e discussões, que além de contribuírem para meu amadurecimento no trabalho de pesquisa, foram fundamentais para a elaboração e conclusão deste trabalho.

Aos meus familiares, pelo apoio constante e fé inabalável em minha capacidade.

Ao Jovair, pela paciência e compreensão.

Aos companheiros do grupo de banco de dados, Marcos André, Luis Mariano, Fátima, Marcos Antônio, Juliano, Alexandre, Cereja, Hélio Rubens, Laura, Walter, Bei Yi, Jefferson e Ricardo, pelas sugestões e valiosa cooperação. Em especial, ao Marcos André, pelas inúmeras colaborações.

Às amigas e colegas de república, Janne, Alessandra, Marília e Anne, pelo espírito de companheirismo e agradáveis momentos compartilhados.

Aos amigos e colegas de mestrado, Gutemberg, Guilherme Albuquerque, Claudio, Cristina Celia, Edicezar, Cleidson, Luciano, Rômulo, e demais, não mencionados por falta de espaço, que de alguma forma contribuíram para que este trabalho fosse realizado.

Aos meus professores, que proporcionaram o conhecimento e os ensinamentos necessários à minha formação e ao próprio desenvolvimento desta dissertação.

Aos funcionários do IC, pela atenção e pronto atendimento.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo suporte financeiro durante parte do trabalho de Mestrado.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), pelo suporte financeiro parcial a esta pesquisa, consolidado em forma de bolsa de mestrado e de recursos para o projeto.

O meu mais sincero obrigado !!!!

O trabalho desenvolvido nesta dissertação foi realizado dentro dos projetos PROTEM-GEOTEC do CNPq, com financiamento parcial do projeto ICDT 116 - GeoTOOls da Comunidade Européia.

Resumo

Esta dissertação discute a implementação de uma infra-estrutura extensível, baseada em um sistema gerenciador de banco de dados orientado a objetos, que provê suporte ao desenvolvimento de aplicações espaço-temporais. A infra-estrutura está baseada na definição e implementação de um conjunto básico de operadores e classes orientadas a objetos, que cobrem os requisitos mínimos de processamento de consultas espaciais, temporais e espaço-temporais.

As principais contribuições deste trabalho são a definição dos operadores e das classes e sua implementação efetiva, validada através de uma aplicação piloto. Uma contribuição adicional é a discussão da própria implementação, que permitiu corrigir falhas do modelo teórico adotado.

Abstract

This dissertation discusses the implementation of an extensible framework, which provides support for the development of spatio-temporal database applications. The infrastructure, developed on the O2 object-oriented database system, consists of a kernel set of operators and database classes, which meet the minimum requirements for the processing of spatial, temporal and spatio-temporal queries.

The main contributions of this work are the specification of the kernel operators and classes and their implementation, validated through a pilot geographic application. Another contribution is the analysis of this implementation, which discusses problems and shortcomings of some models proposed in the literature.

Conteúdo

Agradecimentos vi

Resumo viii

Abstract ix

1 Introdução 1

1.1 Motivação e Objetivos da Dissertação 1

1.2 Organização da Dissertação 3

1.3 Convenções 3

2 Conceitos e Revisão Bibliográfica 4

2.1 Bancos de Dados Temporais 4

2.1.1 Terminologia 5

2.1.2 Modelos de Dados Temporais 10

2.1.3 Linguagens de Consulta Temporais 13

2.1.4 Projeto de Bancos de Dados Relacionais Temporais 14

2.1.5 Implementação de Bancos de Dados Temporais 15

2.1.6 O Modelo TOODM (Temporal Object Oriented Data Model) 17

2.1.7 CGT - Implementação do Modelo TOODM 19

2.2 SIG e Tempo 20

2.2.1 Conceitos Básicos 21

2.2.2 Problemas Existentes em SIG 22

2.2.3 Modelagem de Dados Geográficos 23

2.2.4 Um Modelo de Dados Geográfico 25

2.3 Um Modelo de Dados Geográfico Temporal 27

2.3.1 Estrutura 28

2.4 Resumo 29

3 **Operadores Primitivos Espaço-temporais** 30
3.1 Introdução 30
3.2 Operadores 32
3.2.1 Operadores Espaciais 32
3.2.2 Operadores Temporais 40
3.2.3 Operadores Espaço-temporais 45
3.2.4 Outros Operadores 46
3.3 Consultas 47
3.3.1 Notação 47
3.3.2 Consultas Espaciais 50
3.3.3 Consultas Temporais 51
3.3.4 Consultas Espaço-Temporais 52
3.4 Outras Consultas 54
3.5 Resumo 56

4 **Implementação do Sistema – Estruturas e Algoritmos** 57
4.1 Classes Implementadas - Visão Geral 57
4.2 Classes de Tempo 60
4.2.1 Representação do Tempo 60
4.2.2 Estrutura Interna das Classes *Time* 62
4.3 Classes de Localização e Geometria 65
4.4 Classes Geométricas 67
4.4.1 Exemplo de Implementação de Operador Espacial 68
4.4.2 Classes Geométricas Temporalizadas 70
4.5 Classes Geográficas e Temporais 72
4.6 Funções 78
4.7 Análise das Classes e sua Implementação 80
4.8 Outras Alternativas de Modelagem 82
4.9 Resumo 83

5 **Consultas Espaço-temporais** 84
5.1 Visão Geral do Problema 84
5.2 Definições de Classes 86
5.3 Exemplo de Modelagem de Geo-objetos 86
5.4 Exemplos de Consultas 90
5.4.1 Consultas Espaciais 90
5.4.2 Consultas Temporais 91
5.4.3 Consultas Espaço-Temporais 93
5.5 Resumo 97

6 Conclusões e Extensões 98

Bibliografia 100

A Código das Classes e Operadores 106

Lista de Tabelas

5.1 Fazenda 88
5.2 Divisao_Agricola 88
5.3 SpT 89
5.4 GeomT 89
5.5 PolygonT 89
5.6 Interval 89
5.7 Event 90

Lista de Figuras

3.1 Situações ilustrando o relacionamento *disjoint* entre duas regiões (a); uma linha e uma regiões (b); uma região e um ponto (c); duas linhas (d); uma linha e um ponto (e); e dois pontos (f) 36
3.2 Situações ilustrando o relacionamento *touch* entre duas regiões (a) e (b); duas linhas (c) e (d); uma linha e uma região (e)-(h); um ponto e uma linha (i); e um ponto e uma região (j) 37
3.3 Situações ilustrando o relacionamento *overlap* entre duas regiões (a); e duas linhas (b) e (c) 38
3.4 Situações ilustrando o relacionamento *inside* entre duas regiões (a) - (c); duas linhas (d) e (e); uma linha e uma região (f) - (h); um ponto e uma linha (i); um ponto e uma região (j); e dois pontos (k) 38
3.5 Situações ilustrando o relacionamento *cross* entre duas linhas (a); e uma linha e uma região (b) - (e) 39
3.6 Situações ilustrando o relacionamento temporal *t_before* 41
3.7 Situações ilustrando o relacionamento temporal *t_overlaps* 42
3.8 Situações ilustrando o relacionamento temporal *t_equal* 43
3.9 Situações ilustrando o relacionamento temporal *t_contains* 43
3.10 Situações ilustrando o relacionamento temporal *t_meets* 44
3.11 União de mapas 55

4.1 Modelo básico de dados 58
4.2 Representação de dois geo-objetos do tipo Fazenda 68
4.3 Estados da geometria das fazendas A e B entre t1 e t4 76

5.1 Esquema da Aplicação 87
5.2 Histórico da localização da fazenda A 90

Capítulo 1

Introdução

1.1 Motivação e Objetivos da Dissertação

Esta dissertação tem por objetivo implementar a base para um banco de dados espaço-temporal para aplicações de Sistemas de Informação Geográfica (SIGs), usando um Sistema Gerenciador de Bancos de Dados (SGBD) Orientado a Objetos (OO). A implementação é baseada no SGBD O_2 [BDK92, O2T92].

Os bancos de dados convencionais foram projetados para armazenar e processar dados sobre o presente, ou seja, dados atuais. À medida que novos valores se tornam disponíveis através de atualizações, os valores que existiam antes são removidos do banco de dados. Assim, esses bancos de dados capturam apenas uma imagem instantânea da realidade. Bancos de dados convencionais são insuficientes para aquelas aplicações que necessitam dos dados passados e/ou futuros. A inexistência de um sistema gerenciador de banco de dados temporal força essas aplicações a criarem seus próprios métodos para gerenciar suas informações temporais. Num sentido mais amplo, um banco de dados que mantém dados passados, presentes e futuros é chamado de *banco de dados temporal* [SA86].

Apesar dos avanços recentes em gerenciamento do tempo, há ainda vários problemas em aberto. Um *workshop* recente na área de SGBDs temporais [SJS95] aponta tópicos tais como: utilização da tecnologia de bancos de dados temporais para a resolução de problemas práticos e do mundo real, modelagem de dados e linguagens de consulta. Alguns aspectos de linguagens de consultas devem ainda ser documentados e resolvidos convenientemente. Aplicações que vão além de gerenciamento de dados administrativos ainda desafiam a tecnologia de bancos de dados temporais. Os requisitos de aplicações de gerenciamento de séries temporais, por exemplo, são muito diferentes dos tradicionais e não estão sendo atendidos [SJS95].

A maioria das implementações atuais de bancos de dados temporais não tem sido avaliada com grandes bancos de dados, e características tradicionais de bancos de dados

como persistência, transações e concorrência nem sempre são providas.

Esta dissertação tem por objetivo tratar de uma das questões levantadas, a saber, gerenciamento temporal de dados para um grande conjunto de aplicações não administrativas: aquelas que manuseiam dados espaciais. Mais particularmente, serão abordadas aplicações que lidam com dados *geo-referenciados* – termo usado para denominar dados que são relacionados a algum local na superfície da Terra. Estas aplicações são em geral implementadas a partir de algum Sistema de Informação Geográfica (SIG). SIGs são sistemas computacionais que armazenam e manipulam informações geográficas. Atualmente, esses sistemas não provêem nenhum mecanismo para manipulação de versões antigas dos dados geo-referenciados. No entanto, os SIGs temporais podem liberar o usuário da difícil tarefa de gerenciar manualmente a crescente quantidade de dados históricos desses sistemas.

SIGs são freqüentemente apoiados por algum SGBD, que provê as funções básicas de gerenciamento de dados. O trabalho da dissertação está concentrado em expandir este SGBD, de forma a prover mecanismos básicos de gerenciamento temporal para dados geo-referenciados. Esta dissertação realizou a implementação de estruturas para suporte à evolução temporal de dados geográficos; e operações que podem ser utilizadas na elaboração de consultas sobre estes dados.

Esta implementação caracteriza-se por:

1. fundamentar-se no modelo espaço-temporal proposto por Botelho [Bot95];
2. utilizar o SGBDOO O_2 como suporte;
3. funcionar como uma infra-estrutura sobre a qual são acrescentadas as classes de uma aplicação de SIG.

O sistema O_2 não oferece suporte espaço-temporal. Por este motivo foi necessário especificar e implementar:

- estruturas que viabilizam a representação da evolução temporal de dados espaciais;
- operações básicas utilizadas para a elaboração de consultas espaço-temporais típicas.

As principais contribuições são, desta forma:

- especificação de um conjunto de operadores básicos espaciais e temporais que servem para a formulação de consultas espaço-temporais (Capítulo 3). A literatura correlata trata de operadores espaciais ou operadores temporais, mas não de sua combinação.

- especificação de um conjunto básico de classes do banco de dados para dar apoio ao gerenciamento espaço-temporal, e algoritmos para implementar os operadores nestas classes (Capítulo 4).
- implementação de classes e algoritmos, validando a proposta com implementação de consultas típicas de aplicações geográficas (Capítulo 5).

1.2 Organização da Dissertação

A dissertação está estruturada da seguinte forma:

- o capítulo 2 dá uma visão geral de modelos de dados temporais e das questões relacionadas à incorporação do conceito de tempo em bancos de dados e SIGs;
- o capítulo 3 analisa as operações básicas que podem ser utilizadas para a formulação de consultas típicas em bancos de dados geográficos, propõe como combinar as operações, e sugere uma classificação para estas consultas;
- o capítulo 4 apresenta as principais estruturas e operadores definidos para implementação de navegação espaço-temporal;
- o capítulo 5 descreve um exemplo da utilização do banco de dados implementado para uma aplicação geográfica;
- finalmente, o capítulo 6 apresenta as conclusões e possíveis extensões a este trabalho.

1.3 Convenções

São usadas as seguintes convenções neste texto:

- Os operadores primitivos são escritos usando letras maiúsculas. Por exemplo: TOUCH.
- Os métodos e funções são escritos em itálico, usando letras minúsculas. Por exemplo: *t_overlaps*.
- As classes têm letra inicial maiúscula. Exemplo: Time.
- Qualquer qualificador opcional está incluso em colchetes.

Capítulo 2

Conceitos e Revisão Bibliográfica

Este capítulo fornece uma visão geral de modelos de dados temporais e das questões relacionadas à incorporação do conceito de tempo em bancos de dados e SIGs, destacando os problemas em aberto. A seção 2.1 aborda os bancos de dados temporais; a seção 2.2, a incorporação da semântica de tempo em SIGs; e a seção 2.3 descreve o modelo de Botelho, usado como base na dissertação.

2.1 Bancos de Dados Temporais

O tempo é um aspecto importante de todos os fenômenos do mundo real. Eventos ocorrem em pontos específicos de tempo; objetos e seus relacionamentos têm existência dependente de dimensões temporais; porém bancos de dados convencionais (atemporais) representam o estado do mundo em um único momento de tempo. Embora o conteúdo do banco de dados continue a mudar porque novas informações são atualizadas, estas mudanças são modificações de estado: dados velhos, desatualizados, são apagados do banco de dados. Este conteúdo, corrente, pode ser visto como uma imagem instantânea (*snapshot*). Em tais sistemas, eventuais atributos envolvendo tempo são manipulados unicamente por programas de aplicação e o sistema de gerenciamento de banco de dados interpreta datas como valores de tipos de dados básicos [Sno92].

Dados em um banco de dados atemporal são temporalmente inconsistentes porque se tornam correntes em pontos de tempo diferentes e desconhecidos. Em contraste, um banco de dados temporal modela o mundo na sua dinâmica, rastreando pontos de mudança e retendo todos os dados.

Várias aplicações podem se beneficiar do suporte a tempo em SGBDs: sistemas financeiros, planos de tratamento médico, monitoração ecológica, gerenciamento de dados de vídeo, etc. Estas aplicações requerem suporte a dados incompletos e consultas temporais muito complexas [SJS95].

O suporte em SGBDs para informações que variam no tempo, independente de aplicação, tem sido alvo de muitas pesquisas recentes [McK86, Soo91, Kli93, ATSS93, TK96, CS94]. Este campo de pesquisa já produziu, inclusive, uma revisão bibliográfica tendo a extensão de um livro [TCG+94], e dois *proceedings* de *workshop* [Sno93, CT95b].

Durante os últimos anos, foram propostos vários modelos de bancos de dados que incorporam o conceito de tempo. A maioria destas propostas limitou-se a estender o modelo relacional. Segundo [RY94], isto se deve ao fato deste modelo ter uma teoria sistemática completa, um fundamento matemático firme e uma estrutura muito simples. Todavia, grande parte das aplicações que requerem um gerenciamento de dados dentro do contexto temporal caracteriza-se por apresentar uma natureza intrinsicamente orientada a objetos. Logo, torna-se necessário a incorporação do conceito de tempo em bancos de dados orientados a objetos.

Dados temporais também podem ser usados em bancos de dados ativos cujas transações têm restrições de tempo (bancos de dados de tempo-real). Ramamritham et al. [RSS+96] discutem políticas para a localização de dados, registro de operações e recuperação nesses sistemas de bancos de dados, de modo a alcançar um processamento de transações eficiente.

2.1.1 Terminologia

Apesar de muitos conceitos definidos na literatura sobre bancos de dados temporais serem novos, já existe um glossário para eles [JCE+94]. A seguir serão apresentados os termos principais a serem usados na dissertação. Esses termos serão definidos de acordo com [JCE+94, Sno92].

Variabilidade de Atributos (ou Entidades)

Um atributo invariante no tempo é um atributo cujo valor não muda ao longo do tempo. Também é chamado de *dado estático*. Um atributo variante no tempo é um atributo cujo valor não está restrito a ser constante ao longo do tempo, ou seja, pode mudar ou não com o tempo. Também é chamado de *dado temporal*.

Um problema temporal fundamental é como reconhecer versões diferentes de uma entidade como correspondendo à mesma entidade. Versões de entidades são facilmente relacionadas em um banco de dados usando identificadores *surrogate* [1] que servem como chaves temporais. Os problemas semânticos permanecem: qual mudança faz com que uma entidade se torne uma nova entidade em vez de uma nova versão da antiga? Claramente,

[1] Um identificador único de um item, gerado pelo sistema, que pode ser referenciado e comparado por igualdade, mas não mostrado para o usuário.

cada aplicação deve definir os elementos essenciais que identificam cada uma de suas entidades [Lan89].

Modelos de Tempo

Existem três modelos estruturais de tempo:

- Linear – o tempo avança do passado para o futuro de uma maneira totalmente ordenada.
- Ramificado (ou modelo dos futuros possíveis) – o tempo é linear do passado até *now* (constante especial que representa o tempo atual), onde então se divide em várias linhas de tempo, cada uma representando uma sequência potencial de eventos. Ao longo de qualquer caminho futuro, ramificações adicionais podem existir. Há também possibilidade de ramificação no passado, por exemplo no caso de diferentes versões históricas.
- Cíclico – pode ser usado para representar eventos recorrentes.

O modelo de tempo mais geral em uma lógica temporal representa o tempo como um conjunto arbitrário, com uma ordem parcial imposta sobre este [Sno92, Sno95a]. Axiomas adicionais podem introduzir outros modelos de tempo mais refinados. Por exemplo, o tempo linear pode ser especificado adicionando um axioma que imponha uma ordem total neste conjunto.

Granularidade Temporal

A granularidade de um valor de tempo é a precisão com a qual esse valor pode ser representado. Se uma representação tiver mais que um componente, ela é de uma granularidade composta (por exemplo, "dia/mês/ano"); senão, é de uma granularidade simples (por exemplo, "segundo").

Por exemplo, se for considerada a granularidade de um segundo relativa a 0:00h de 01/01/1980, então, em uma granularidade simples, o inteiro 164.281.022 denota 9:37:02h de 15/03/1985. Se for considerada uma granularidade composta de (ano, mês, dia, hora, minuto, segundo), então a sequência (6,3,15,9,37,2) denota aquele mesmo tempo.

O tempo em bancos de dados costuma ser definido em *chronons*. Um *chronon* é uma unidade indivisível de tempo com uma duração arbitrária, sendo fixa para uma aplicação. Um *chronon* é a menor duração de tempo que pode ser representada no modelo discreto, definido a seguir.

Densidade do Tempo

Embora o tempo seja contínuo por natureza, existem três modelos:

- Discreto – isomorfo aos números naturais, implicando que cada ponto de tempo tem um único sucessor.
- Denso – isomorfo ou aos racionais ou aos reais: entre quaisquer dois momentos de tempo existe outro momento.
- Contínuo – isomorfo aos reais, i.e., é denso e diferente dos racionais, não contém "buracos".

A maioria das propostas para a adição da dimensão temporal ao modelo de dados relacional são baseadas no modelo discreto de tempo. Isto se deve a quatro fatores. Primeiro, medidas de tempo são inerentemente imprecisas. Segundo, a maioria das referências a tempo da linguagem natural são compatíveis com o modelo discreto de tempo. Por exemplo, quando dizemos que um evento ocorreu às 8:30h, geralmente não queremos dizer que o evento ocorreu no "ponto de tempo" associado a 8:30h, mas em algum tempo no *chronon* (talvez minuto) associado a 8:30h. Terceiro, os conceitos de *chronon* e intervalos permitem modelar naturalmente eventos que não são instantâneos, mas têm duração. Finalmente, qualquer implementação de um modelo de dados com uma dimensão temporal necessariamente precisará ter uma codificação discreta para tempo.

Dimensões Temporais

São em geral utilizadas duas dimensões temporais em um SGBD OO: *tempo de validade* e *tempo de transação*. Estes tempos são ortogonais, embora haja geralmente algumas correlações entre eles, dependentes de aplicações. Podem ser suportados separadamente ou conjuntamente em um banco de dados, e não são homogêneos, já que o tempo de transação tem uma semântica diferente do tempo de validade.

- Tempo de Transação (*transaction time*)

  O tempo de transação de um valor no banco de dados é o (intervalo de) tempo em que aquele valor está/esteve armazenado. O tempo de transação associado a um conjunto de dados identifica o tempo das transações que inseriram/removeram o conjunto no banco de dados. Os valores de tempo de transação não podem ser maiores que o tempo atual. Além disso, os tempos de transação não podem ser mudados. Os valores relativos ao tempo de transação costumam ser supridos automaticamente pelo próprio SGBD.

- Tempo de Validade (*valid time*)

  O tempo de validade de um fato é o tempo em que aquele fato é verdade na realidade modelada. O tempo de validade de um evento é o tempo do mundo real quando o evento ocorreu, independente do armazenamento daquele evento em algum banco de dados. Tempos de validade podem também estar no futuro quando se sabe que algum fato ocorrerá em um tempo específico no futuro. O tempo de validade é fornecido pelo usuário na maioria das vezes.

- Tempo Definido pelo Usuário (*user-defined time*)

  O tempo definido pelo usuário é um atributo de domínio não interpretado pelo banco de dados, usado para armazenar informação temporal sobre alguns fatos. Este domínio tem mais semelhança com domínios de atributos convencionais do que com os domínios temporais, pois não existe um suporte na linguagem de consulta para tratá-los.

Estes tipos de tempo induzem tipos diferentes de bancos de dados:

- Banco de Dados Instantâneo (*snapshot database*) – suporta somente o tempo definido pelo usuário. Contém apenas uma imagem instantânea da realidade.

- Banco de Dados Tempo de Validade (*historic database*) – suporta somente o tempo de validade e contém todo o histórico do mundo real.

- Banco de Dados Tempo de Transação (*rollback database*) – suporta somente o tempo de transação e armazena todas as alterações realizadas sobre os dados. Este banco de dados permite voltar para qualquer estado passado no banco de dados e fazer consultas nesse estado, pois não há exclusão física dos dados. Ficam, portanto, registrados os erros, provendo todas as informações necessárias para uma auditoria, simplificando os mecanismos de correção de erros, recuperação de falhas e contabilidade.

- Banco de Dados Bitemporal – registra tanto o tempo de transação quanto o tempo de validade e combina as características dos anteriores. A sequência completa de atualizações e valores atualizados é disponibilizada para consulta. Um banco de dados bitemporal *degenerado* é aquele em que um fato é armazenado tão logo se torne válido na realidade, sendo o tempo de transação idêntico ao tempo de validade.

Os bancos de dados de tempo de transação armazenam o histórico de atividades do banco de dados e os bancos de dados de tempo de validade armazenam o histórico de atividades do mundo real. Além disso, a alteração do valor de um dado em um banco de

dados de tempo de validade implica na sua substituição pelo valor novo, enquanto que nos bancos de dados de tempo de transação o valor novo é adicionado ao valor antigo como um novo estado [Bot95].

Múltiplos tempos de transação podem também ser armazenados sobre um mesmo registro, o que é denominado *generalização temporal.* Estes tempos podem ser relacionados um ao outro, ou ao tempo de validade, de várias formas especializadas.

Indeterminância Temporal

Uma informação é historicamente indeterminada quando não possui um valor de tempo de validade preciso. Este tipo de informação aparece em várias situações, incluindo: técnicas de marcação de tempo imperfeitas, incerteza em planejamentos, tempos de eventos imprecisos ou desconhecidos.

Há várias propostas para a adição de indeterminância a um modelo temporal. O *modelo de chronons possíveis* [Sno92] trata desse aspecto. Neste modelo, um evento é *determinado* quando é conhecido quando ele ocorreu, i.e., durante qual *chronon*. Se não se sabe quando um evento aconteceu, mas se sabe que ele realmente ocorreu, então o evento é historicamente indeterminado.

Um evento indeterminado é completamente descrito por um conjunto de *chronons* possíveis e por uma distribuição de probabilidades de eventos. Um único *chronon* deste conjunto denota quando o evento indeterminado realmente ocorreu. Todavia, não se sabe qual dos *chronons* possíveis é o real. A distribuição de probabilidades dá a probabilidade de ocorrência do evento durante cada *chronon* do conjunto de *chronons* possíveis.

A implementação do modelo de *chronons* possíveis suporta um *chronon* de tamanho fixo, minimal. Múltiplas granularidades são manipuladas representando-se a indeterminância explicitamente. Por exemplo, se o *chronon* subjacente é um microsegundo e um evento é conhecido em um dia, então este evento indeterminado seria associado a um conjunto de 86.400.000 *chronons* possíveis, e talvez a uma distribuição de probabilidade de eventos uniforme.

A indeterminância histórica não ocorre com tempos de transação, pois o *chronon* durante o qual a transação acontece é sempre conhecido.

Marca de Tempo (*Timestamp*) e Tempo de Vida (*Lifespan*)

Uma marca de tempo é um valor de tempo associado a um objeto, por exemplo, atributo, tupla. Há três tipos básicos de marcas de tempo: instantes (eventos ou pontos), intervalos, ou elementos temporais. Snodgrass [Sno92] também utiliza como marca de tempo *durações temporais* (ou *spans*).

Um *instante* representa um ponto de tempo em um eixo temporal. Um *intervalo*

temporal é caracterizado pelo tempo decorrido entre dois instantes. Em um sistema que suporta uma linha de tempo composta por *chronons*, um intervalo pode ser representado por um conjunto de *chronons* contíguos. Um intervalo é representado pelos dois instantes que o delimitam. Dependendo da pertinência ou não dos instantes limites ao intervalo, este pode ser aberto (os limites não pertencem ao intervalo), semiaberto (um dos limites pertence ao intervalo) ou fechado (os limites pertencem ao intervalo). Quando um dos limites é representado pelo instante atual (*now*) tem-se a representação de um intervalo particular cujo tamanho varia com a passagem do tempo.

Um *elemento temporal* é um conjunto finito de intervalos de tempo n-dimensionais.

Durações são tempos relativos, podendo ser de dois tipos: fixas e variáveis. O *tempo relativo* tem sua posição relativa a alguma entidade, ou a variáveis de tempo como a variável especial *now*. O *tempo absoluto* define a posição de uma entidade no eixo temporal. Esta posição não depende nem do tempo de outra entidade, nem de variáveis de tempo, como a variável *now*.

Uma *duração fixa* é fornecido em termos de um número de *chronons*. Uma *duração variável* é dependente de um evento associado. Um exemplo de duração variável comum é o "mês". A duração representada por um mês depende se aquele mês é associado a um evento em junho (30 dias) ou julho (31 dias), ou em fevereiro (28 ou 29 dias). Uma duração pode ser positiva, denotando passagem de tempo para o futuro, ou negativa, denotando volta no tempo para o passado.

O *tempo de vida (lifespan)* de uma entidade é o intervalo de tempo sobre o qual ela é definida.

2.1.2 Modelos de Dados Temporais

Um modelo de dados consiste em um conjunto de objetos com alguma estrutura, um conjunto de restrições e operações sobre esses objetos [Sno92]. Um modelo de dados deve ainda fornecer ferramentas para descrever a organização lógica de bancos de dados, bem como definir as operações de manipulação de dados permitidas [CCH+96].

Muitas extensões ao modelo relacional para incorporar tempo foram propostas nos últimos 15 anos [Sno95a]. A maioria suporta somente o tempo de validade, uns poucos suportam somente o tempo de transação ou ambos os tipos de tempo. Maiores detalhes sobre extensões ao modelo relacional podem ser encontrados em Snodgrass [Sno92] e em Skjellaug [Skj97b].

Segev et al. [SJS95] fazem referência a alguns dos modelos de dados temporais existentes e a modelos que estão sendo definidos, inclusive modelos orientados a objetos. Estes têm demonstrado serem mais aptos para armazenar dados temporais, pois facilitam o aninhamento temporal dos dados. Edelweiss e Oliveira [EO94] também apresentam alguns dos modelos de dados temporais propostos, enfocando principalmente a forma de

representação dos aspectos temporais, e a existência de uma linguagem de consulta associada. Skjellaug [Skj97a] revisa modelos e linguagens temporais orientados a objetos, apresentando uma comparação compreensível. Outras tabelas comparativas se encontram em Oliveira [Oli93], em Brayner [Bra94], e em Jensen [Jen97].

Os modelos temporais podem ser comparados pela elaboração das seguintes questões básicas [Sno92]: "como é representado o tempo de validade?", "como é representado o tempo de transação?", "como são representados os valores de atributos?".

Tempo de Validade. Dois aspectos completamente ortogonais estão envolvidos na representação de tempo de validade. Primeiro, como representá-lo: com identificadores de chronons simples (marcas de tempo do tipo evento), com intervalos (marcas de tempo do tipo intervalo), ou como elementos históricos (i.e., como um conjunto de identificadores de chronons, ou equivalentemente como um conjunto finito de intervalos). Segundo, o tempo de validade pode ser associado a valores de atributos individuais, grupos de atributos, tuplas ou objetos. Uma terceira alternativa, associar tempo de validade a conjuntos de tuplas, não foi incorporada em nenhum dos modelos propostos, devido principalmente à alta redundância de dados.

Tempo de Transação. As mesmas questões gerais surgem quanto ao tempo de transação, mas há quase o dobro de alternativas. O tempo de transação pode ser associado a um chronon simples, um intervalo, três chronons[2], um elemento de tempo de transação (um conjunto de chronons não necessariamente contíguos). Outra questão se refere à associação do tempo de transação a valores de atributos individuais, grupos de atributos, tuplas, objetos, conjuntos de tuplas, grafo de objetos, esquema.

Estrutura de Valores de Atributos. A decisão final principal a ser feita no projeto de um modelo de dados é como representar valores de atributos. Há seis alternativas básicas, sem considerar o tempo que aparece como atributo explícito, nos modelos relacionais [Sno92]:

- Atômico Valorado – os valores não possuem qualquer estrutura interna. Exemplo: 'Maria'
- Conjunto Valorado – os valores são conjuntos de valores atômicos. Exemplo: {'Maria', 'Eli'}

[2] Utilizados, por exemplo, para armazenar (1) o tempo de transação quando o tempo de validade de início foi armazenado, (2) o tempo de transação quando o tempo de validade de finalização foi armazenado, e (3) o tempo de transação quando a tupla foi logicamente excluída.

- Funcional, Atômico Valorado – os valores são funções do domínio de tempo (geralmente o de validade) para o domínio do atributo. Exemplo: 1993 → 'Maria', 1994 → 'Eli'

- Pares Ordenados – os valores são pares ordenados, os quais são compostos por um valor do atributo e uma marca de tempo (elemento histórico). Exemplo: ('Maria', 1993)

- Terno Valorado – os valores são triplas, as quais são formadas pelo valor do atributo, tempo inicial e tempo final. É similar a pares ordenados, exceto que somente um intervalo pode ser representado. Exemplo: ('Maria', Janeiro de 1993, Março de 1994)

- Conjuntos de Ternos Valorados – os valores são um conjunto de ternos. Esta abordagem é mais geral que a de pares ordenados, visto que mais de um valor pode ser representado. E também é mais geral que a funcional valorada, desde que mais de um valor de atributo pode existir para um único tempo de validade. Exemplo: {('Maria', Janeiro de 1993, Março de 1994), ('Eli', Fevereiro de 1992, Dezembro de 1993)}

Estas alternativas são mais limitadas nos modelos orientados a objetos [Sno95a]:

- Atômico Valorado – com tempo associado a objetos inteiros ou conjunto de atributos, em vez de atributos individuais.

- Conjuntos de pares ordenados – de valores e marcas de tempo.

- Funções de um domínio de marcas de tempo.

- Funções representadas por instâncias de classes.

Separando a Semântica da Representação

Segundo [Sno92, JS96], o enfoque na *apresentação* de dados temporais, no *armazenamento* de dados (com seus requisitos de estrutura regular), e na *avaliação* eficiente *de consultas* complicou a tarefa de capturar a semântica temporal de dados. O resultado foi uma abundância de modelos de dados incompatíveis, com muitas linguagens de consulta, e um correspondente excesso de estratégias de projeto e implementação que podem ser empregadas sobre estes modelos.

Snodgrass et al. defendem, em vez disso, um modelo de dados muito simples, o *Modelo de Dados Conceitual Bitemporal* ou BCDM [JSS94], que captura a semântica essencial de conjuntos de dados variantes no tempo, mas que não tem ilusões de ser adequado

para apresentação, armazenamento, ou avaliação de consultas. Aqueles autores sugerem a utilização dos modelos de dados existentes para estas últimas tarefas. Este modelo conceitual marca o tempo de tuplas com *elementos bitemporais*, conjuntos de chronons bitemporais que são retângulos no espaço bidimensional gerado pelo tempo de validade e tempo de transação. Como não são permitidas duas tuplas que tenham valores de atributos explícitos (atributos atemporais) idênticos em uma instância de relação bitemporal, toda a história de um fato é contida em uma única tupla.

Segundo [Sno92, JS96], é possível demonstrar mapeamentos de equivalência entre o modelo conceitual e vários modelos de representação. Esta equivalência é baseada em equivalência de *snapshots*, que diz que instâncias de dois conjuntos de dados são equivalentes se todos os seus *snapshots*, obtidos em todos os tempos (de transação, de validade) forem idênticos. Uma extensão aos operadores algébricos relacionais convencionais pode ser definida no modelo de dados conceitual, e pode ser mapeada em operadores análogos nos modelos de representação.

Na essência, Snodgrass et al. defendem mover a distinção entre os vários modelos de dados temporais existentes de uma base semântica para uma base física, utilizando o modelo de dados conceitual proposto para capturar a semântica variante no tempo. Múltiplos formatos de apresentação devem estar disponíveis, porque aplicações diferentes requerem modos diferentes de visualizar dados. O armazenamento e processamento de conjuntos de dados bitemporais deve ser feito em modelos de dados que enfatizam eficiência.

2.1.3 Linguagens de Consulta Temporais

Muitas linguagens de consulta temporais têm sido propostas. Geralmente cada pesquisador define seu próprio modelo de dados e linguagem.

[Sno92] resume algumas das linguagens de consulta propostas, e discute as atividades que devem ser suportadas por uma linguagem de consulta temporal. Essas atividades são: definição de esquema, *rollback*, seleção de tempo de validade, projeção de tempo de validade, agregações, indeterminância histórica, evolução de esquema. Jensen e Snodgrass [JS97] citam, além destas, outras atividades: predicados sobre valores temporais, construtores temporais, suporte a múltiplos calendários, modificação de relações temporais, visões, restrições de integridade, manipulação de *now*, dados periódicos, *vacuuming*[3].

Brayner [Bra94] descreve as principais propriedades de algumas linguagens de consultas e álgebras propostas para operarem sobre Bancos de Dados Temporais. Snodgrass [Sno95a] resume algumas das linguagens de consulta orientadas a objeto propostas.

Segev et al. [SJS95] descrevem algumas linguagens existentes e linguagens que estão

[3]Remoção de dados antigos, possivelmente obsoletos e incorretos [Jen97].

sendo definidas. Relatam também o trabalho que está sendo feito no estabelecimento de padrões internacionais, e a discussão em torno das características desejadas para uma linguagem de consulta temporal.

A linguagem TSQL2 [Sno95c, Jen97, SAA+94b] é no momento um consenso, mas já existem propostas para extensões e esclarecimentos [SJS95, BJS95].

2.1.4 Projeto de Bancos de Dados Relacionais Temporais

O projeto de um banco de dados é tipicamente considerado em dois contextos [JS97]. No projeto conceitual, o banco de dados é modelado usando um modelo de alto-nível, que é independente do modelo particular (implementação) do SGBD que será eventualmente usado para gerenciá-lo. O segundo contexto do projeto é o modelo de dados de implementação. Neste contexto, o projeto deve ser então considerado no nível de visão, o nível lógico (originalmente denominado "conceitual"), e no nível físico (ou "interno").

Projeto Conceitual

A maioria das pesquisas relacionadas ao projeto conceitual de bancos de dados temporais tem sido no contexto do modelo Entidade-Relacionamento (ER). Este modelo, em suas formas variantes, está desfrutando uma notável e crescente popularidade na indústria.

Pesquisas em modelos ER temporais são bem motivadas. É bem conhecido que os aspectos temporais do mini-mundo[4] são muito importantes para muitas aplicações, mas também são difíceis de capturar usando o modelo ER. Dito de outra forma, quando se tenta capturar os aspectos temporais, estes tendem a obscurecer e causar desordem a diagramas de outra maneira intuitivos e fáceis de entender.

Como um resultado, alguns usuários simplesmente escolhem ignorar todos aspectos temporais em seus diagramas ER, suplementando estes com frases textuais para indicar que existe uma dimensão temporal para os dados. O resultado é que o mapeamento de diagramas ER para relações deve ser realizado a mão; e os diagramas não documentam bem os esquemas de bancos de dados relacionais estendidos usados pelos programadores de aplicação.

A resposta da comunidade de pesquisa tem sido o desenvolvimento de modelos ER temporais. Segundo [JS97], quase doze modelos já foram publicados na literatura. Estes modelos representam tentativas de modelar aspectos temporais de informações mais naturalmente e elegantemente. Eles assumem tipicamente que seus esquemas serão mapeados para esquemas no modelo relacional que serve como modelo de implementação. Os algoritmos de mapeamento são construídos para adicionar os atributos temporais aos esquemas

[4]Seguindo [JS97], mini-mundo é a parte da realidade sobre a qual o banco de dados armazena informações.

relacionais. Nenhum dos modelos tem um dos muitos modelos relacionais temporais propostos [OS95] como seu modelo de implementação. Estes modelos têm capacidades de definição de dados e linguagens de consulta que melhor suportam o gerenciamento de dados temporais e seriam, então, canditatos naturais a plataformas de implementação.

Projeto Físico

Uma das principais metas do projeto de bancos de dados relacionais convencionais é produzir um bom esquema de banco de dados, consistindo de um conjunto de esquemas de relações. Formas normais são uma tentativa de caracterizar bons esquemas. Uma grande variedade de formas normais têm sido propostas, a mais proeminente sendo a terceira forma normal e a forma normal de Boyce-Codd.

Os conceitos existentes de normalização não são aplicáveis a modelos relacionais temporais, porque estes modelos empregam estruturas relacionais que são diferentes das relações convencionais. Há, então, uma necessidade de novas formas normais e conceitos subjacentes que possam servir como diretrizes durante o projeto de bancos de dados temporais. Em resposta a esta necessidade, têm sido propostos conceitos de normalização temporal, incluindo dependências temporais, chaves, e formas normais [Jen97, JS97].

2.1.5 Implementação de Bancos de Dados Temporais

A adição de suporte temporal a um SGBD causa impacto virtualmente em todos seus componentes. Na DDL, as mudanças para o suporte a tempo envolvem acrescentar domínios adicionais, como evento, intervalo, e duração, e construtores para suportar tempo de validade e tempo de transação. A grande mudança feita no catálogo do sistema é que ele deve consistir em conjuntos de dados de tempo de transação.

Quanto ao processamento de consultas, a análise sintática e léxica da linguagem é facilmente estendida enquanto que a adição de suporte temporal complica a otimização, por várias razões [Sno92, JS97]. Antes de mais nada, ela é mais crítica pois os conjuntos de dados sobre as quais as consultas são definidas são maiores, de modo que consultas não otimizadas levam ainda mais tempo para serem executadas. Segundo, os predicados usados em consultas temporais são mais difíceis de otimizar porque utilizam operadores de comparação (menor que, maior que) mais frequentemente.

Há algumas propostas neste sentido [Sno92, JS97]: separação de dados históricos e dados correntes; novas estratégias de otimização de consultas; novos algoritmos de junção; e novos índices temporais.

O controle de concorrência e as técnicas de gerenciamento de transação devem ser adaptadas para suportar tempo de transação. As questões sutis envolvidas em escolher

se marcar o tempo no início da transação ou no fim dela têm sido resolvidas em favor da última [Sno92].

Finalmente, como um banco de dados de tempo de transação contém todas as versões passadas do banco de dados, ele pode ser usado para a recuperação de falhas que causam a perda parcial ou total da versão corrente.

Böhlen [Boh95] resume o estado da arte das implementações dos bancos de dados temporais existentes. Ele sintetiza não só sistemas de bancos de dados temporais como sistemas relacionados, como por exemplo, um gerador de bancos de dados temporais. [Boh95] classifica cada sistema de acordo com alguns critérios de seleção, e os descreve de uma forma geral. Ele cita algumas propriedades interessantes de implementações. Por exemplo, a maioria dos sistemas não foi avaliada com grandes bancos de dados. Somente dois dos treze analisados trabalharam com bancos de dados com mais de 50 MBytes. Nem todos os sistemas provêem as características dos bancos de dados tradicionais como persistência, transações e concorrência. O conjunto de operações de consulta investigado é altamente desbalanceado.

Pode-se chegar a algumas conclusões pela análise do resumo das implementações feito em [Boh95]:

- O modelo relacional com marcas-de-tempo do tipo período, em tuplas, é predominante.
- Há pouco suporte ao tempo de transação.
- Há muito trabalho em junções e seleções temporais, e quase nada em negação temporal.
- As consultas são focalizadas, enquanto que atualizações, regras, e restrições de integridade são negligenciadas.
- As aplicações existentes parecem indicar que, em geral, é mais importante ter ou suportar granularidades diferentes do que múltiplos calendários, indeterminância, e interpolação[5].
- Sistemas de informações geográficas não foram aplicações para nenhuma das implementações.

Mecanismos de Controle Temporal de Acesso

Em geral, os mecanismos de controle de acesso que fazem parte de sistemas de gerenciamento de dados comerciais não têm capacidades temporais. Em um SGBD relacional

[5] A derivação do valor de um atributo do banco de dados em um *chronon* para o qual este valor não foi explicitamente armazenado.

típico, por exemplo, não é possível especificar, usando a linguagem de comandos de autorização, que um usuário pode acessar uma relação somente por um dia ou uma semana.

Bertino et al. [BBFS96] apresentam um modelo de controle de acesso em que autorizações contêm intervalos de validade. Uma autorização é automaticamente revogada quando o intervalo temporal associado expira. O modelo de [BBFS96] provê regras para a derivação automática de novas autorizações a partir da presença ou ausência de outras autorizações. Ele também suporta autorizações positivas e negativas.

2.1.6 O Modelo TOODM (Temporal Object Oriented Data Model)

Esta seção descreve o modelo TOODM, base de extensão temporal para SIG de [Bot95], implementado na dissertação.

O modelo TOODM de Oliveira [Oli93] incorpora as dimensões tempo de validade e tempo de transação ao modelo orientado a objetos. Além disso, este modelo estende os modelos temporais anteriores para suportar a evolução temporal de todas as propriedades dos objetos e do esquema.

O TOODM possibilita a existência de três tipos de objetos: variantes no tempo, invariantes no tempo e objetos do tipo TIME, usados para representar valores de tempo. O eixo de tempo é linear e discreto. Uma classe pode incorporar até duas dimensões temporais e deve ter pelo menos as mesmas dimensões temporais que as suas superclasses. [Oli93] propôs a linguagem de consulta TOOL (Temporal Object Oriented Language) para consultar o TOODM.

O resultado de uma consulta temporal pode ser: valores de atributos atemporais (por exemplo, números reais), objetos atemporais (ou invariantes no tempo), valores de tempo (por exemplo, [1994,1996]), e objetos variantes no tempo.

Consultas que Retornam Valores de Tempo

Foram criadas duas cláusulas para a realização de consultas que retornam valores de tempo: **TWHEN** e **VWHEN**.

A cláusula **TWHEN** retorna o conjunto de tempos de transação em que o dado selecionado pela consulta foi armazenado ou atualizado. Se a cláusula **VALID** for especificada, a consulta primeiro restringe os objetos selecionados pelo predicado ao intervalo de tempo de validade dado. A sintaxe da consulta é a seguinte:

TWHEN (predicado atemporal)
FROM (repositório de classe)
VALID (expressão temporal)

O resultado desta consulta pode ser obtido através da execução dos seguintes passos [Bot95]:

1. Obter os intervalos de tempo de validade que satisfazem a cláusula VALID.
2. Encontrar os objetos que satisfazem ao predicado atemporal.
3. Retornar os tempos de transação nos quais os objetos encontrados em (2) eram válidos no tempo determinado em (1).

A cláusula **VWHEN** retorna os intervalos de tempo de validade do dado selecionado pela consulta. Se a cláusula **INDB** for especificada, a consulta primeiro restringe os objetos selecionados pelo predicado ao intervalo de tempo de transação dado. A sintaxe da consulta é a seguinte:

VWHEN (predicado atemporal)
FROM (repositório de classe)
INDB (expressão temporal)

O resultado dessa consulta pode ser obtido através da execução dos seguintes passos [Bot95]:

1. Obter os estados com os tempos de transação especificados na cláusula INDB.
2. Encontrar os objetos que satisfazem ao predicado atemporal.
3. Retornar os tempos de validade dos objetos selecionados em (2) e armazenados nos estados determinados em (1).

Consultas que Retornam Objetos Temporais

As consultas que retornam objetos temporais são chamadas de consultas de fatiamento temporal (*timeslice*). Estas consultas retornam os estados (ou fatias) de um banco de dados que satisfazem certas condições durante um período de tempo de validade ou tempo de transação. Para este tipo de consulta foram criadas as cláusulas temporais **TSLICE** e **VSLICE**, que processam operações de seleção temporal sobre os eixos *Tempo de Transação* e *Tempo de Validade*, respectivamente. A sintaxe deste tipo de consulta é a seguinte:

SELECT (predicado atemporal)
FROM (repositório de classe)
TSLICE (expressão temporal)
VSLICE (expressão temporal)

O resultado dessa consulta pode ser obtido através da execução dos seguintes passos [Bot95]:

1. Restringir o domínio da consulta para os estados do banco de dados correspondentes aos intervalos de tempo especificados em TSLICE e/ou VSLICE e seleciona os objetos ativos nesse domínio.

2. Executar a parte atemporal da consulta nos objetos determinados em (1).

2.1.7 CGT - Implementação do Modelo TOODM

Brayner [Bra94, BM94] implementou o TOODM em uma **Camada de Gerenciamento Temporal** (CGT), incorporando o conceito de tempo no banco de dados orientado a objetos O_2. Através de uma interface de menus, a camada permite adicionar ao esquema de uma classe as estruturas necessárias para armazenar as marcas-de-tempo dos seus atributos. A CGT garante a consistência temporal entre as classes do banco de dados através de restrições de integridade temporal, que impedem que uma classe temporal tenha menos dimensões temporais que sua superclasse.

O módulo de consultas da CGT faz um mapeamento das operações temporais definidas no modelo temporal em comandos pertencentes à linguagem de consulta nativa do O_2. Os comandos mapeados interagem com dois componentes da arquitetura do O_2 - o gerenciador de esquema e o gerenciador de objetos - e podem ser classificados em três conjuntos distintos:

- Comandos de manipulação de esquema - têm a função de definir esquemas, bases, classes, aplicações, programas, funções e métodos.

- Instruções imperativas - são utilizadas dentro de um corpo de programa de O_2C (linguagem de programação nativa do banco de dados O_2) para alterar objetos e valores.

- Comandos de consulta - têm a função de recuperar objetos e valores do BD.

Entre as características temporais adotadas na implementação do modelo, pode-se destacar:

- O tempo é *linear* e *discreto* nos dois eixos temporais.

- O eixo tempo de validade foi definido no formato de *intervalos*.

- O eixo tempo de transação foi definido no formato *eventos* e tem como origem o instante da criação da base de dados.

- A granularidade é fixa em *segundos* no eixo de tempo de transação.

- A granularidade é composta e variável no eixo de tempo de validade.

Para o nível mais externo de um objeto complexo é incorporada apenas uma dimensão temporal, conforme o construtor utilizado. Se o objeto tiver o construtor *tuple* no seu nível mais externo, então a dimensão tempo de validade é incorporada ao objeto com a adição do atributo <histórico-objeto>. Se o objeto tiver o construtor *set* ou *list* no seu nível mais externo, então a dimensão tempo de transação é incorporada ao objeto com a adição do atributo <tt_objeto>.

Os objetos podem ter componentes complexos. Para componentes definidos pelo construtor *set* ou *list* somente a dimensão tempo de transação é incorporada através da adição do atributo <tt_componente>. Para componentes definidos pelo construtor *tuple* somente a dimensão tempo de validade é incorporada através da adição do atributo <histórico_componente>. A regra prossegue recursivamente nos componentes do objeto até ser encontrado um componente não composto, ou seja, um tipo primitivo do O_2 (integer, real, char, string ou boolean) ou uma referência a um objeto de uma classe (OID – Identificador do objeto).

Neste caso, as duas dimensões temporais são incorporadas ao componente pela adição dos atributos <tv_início_componente>, <tv_fim_componente> e <tt_componente>.

A marcação de tempo no atributo ao invés da marcação de tempo no objeto (ou marcação de tempo na tupla do modelo relacional) diminui a redundância de dados, evitando a replicação de todos os atributos de um objeto a cada alteração de um de seus atributos. A replicação de objetos traz o problema de múltiplos OID para um mesmo objeto do mundo real.

O módulo de consultas da CGT implementou algumas das cláusulas temporais definidas pela linguagem de consulta TOOL de [Oli93]: VSlice, TSlice, VWhen e TWhen. Os comparadores implementados foram: Before, After, Before_Overlap e After_Overlap. Os operadores temporais implementados que retornam valores de tempo foram: First_Instant, Last_instant, Max_Interval, Min_Interval e All_Interval.

A implementação realizada nesta dissertação usou algumas das idéias de CGT, como se verá no capítulo 4.

2.2 SIG e Tempo

Sistemas de Informação Geográfica (SIGs) são sistemas automatizados usados para armazenar, analisar, manipular e visualizar dados geográficos, ou seja, dados que representam objetos e fenômenos em que a localização geográfica é uma característica inerente à informação e indispensável para analisá-la [CCH+96].

Em SIGs, o tempo é um conceito essencial para a compreensão e a modelagem de

fenômenos espaciais em diversas aplicações, como: ciências biofísicas, pesquisa epidemiológica, ciências políticas, econômicas e sociais e várias aplicações de tempo-real para gerenciamento e planejamento. Os SIGs atuais não provêem mecanismos para a manipulação de versões antigas dos dados geográficos. No entanto, pesquisas recentes têm dado mais atenção ao projeto de SIGs temporais e à incorporação do domínio do tempo em bancos de dados espaciais [ATSS93, Mon95, BVH96, Lan93b, NTE92, Skj96, CT95a, PW94, SB97, BJS97].

Nas seções seguintes serão apresentados conceitos básicos relacionados a SIGs, além de um modelo geográfico e um modelo geográfico espaço-temporal, que permite representar a evolução de dados espaço-temporais, comuns em sistemas de informação geográfica.

2.2.1 Conceitos Básicos

Alguns termos usados na literatura têm significados distintos para diferentes autores. A seguir serão apresentados os conceitos que serão adotados neste texto, baseados em [CCH+96].

Um *banco de dados geográfico* é um repositório de informação coletada empiricamente sobre fenômenos do mundo real. As *entidades geográficas* são os fenômenos geográficos do mundo real (por exemplo: florestas, rios, cidades).

Um *processo geográfico* é definido como uma ação exercida ou um efeito sofrido por uma entidade geográfica, modificando-a. O processo começa quando a atividade ou o efeito começa a mudar o estado da entidade e termina quando a atividade ou o efeito cessa e a entidade tem o seu estado completamente alterado.

A *modelagem de processos* refere-se a uma modelagem matemática que descreve operações envolvendo a representação e manipulação de dados, incluindo a simulação de fenômenos naturais.

O *espaço geográfico* é o meio onde as entidades geográficas coexistem. Ele é definido através dos *relacionamentos espaciais* entre as entidades.

Um *relacionamento espacial* é uma informação derivada a partir do posicionamento de uma entidade em relação a outra. Por exemplo, a distância entre duas cidades é um tipo de relacionamento entre duas entidades geográficas. Segundo [Bot95], apesar de terem grande importância numa análise geográfica, os relacionamentos espaciais não têm sido explorados satisfatoriamente nos sistemas geográficos atuais.

O termo *dado espacial* denota qualquer tipo de dado que descreve fenômenos aos quais esteja associada alguma dimensão espacial. Nesta dissertação são tratados dados espaciais geográficos ou geo-referenciados. Os dados geográficos são comumente caracterizados a partir de três componentes fundamentais: *atributo*, *localização* e *tempo*. O componente *atributo* (também chamado atributo convencional ou atributo não espacial) descreve as propriedades temáticas de uma entidade geográfica, tais como o nome. O componente

localização informa a localização espacial do fenômeno, ou seja, seu geo-referenciamento, associada a propriedades geométricas e topológicas. O componente *tempo* descreve os períodos em que os valores daqueles dados geográficos são válidos.

No contexto de aplicações de SIG, o mundo real é frequentemente modelado segundo duas visões complementares: o modelo de *campos* e o modelo de *objetos*. O modelo de campos enxerga o mundo como uma superfície contínua, sobre a qual os fenômenos geográficos a serem observados variam segundo diferentes distribuições. Um campo é formalizado como uma função matemática cujo domínio é uma (abstração da) região geográfica e cujo contradomínio é o conjunto de valores que o campo pode tomar. Caso se deseje incluir a variação ao longo do tempo, considera-se o domínio da função como um conjunto de pares (p,t) onde p representa um ponto da região geográfica e t um instante de tempo.

O *modelo de objetos* representa o mundo como uma superfície ocupada por objetos identificáveis, com geometria e características próprias. Estes objetos não são necessariamente associados a qualquer fenômeno geográfico específico e podem inclusive ocupar a mesma localização geográfica.

Segundo [CCH+96], campos são frequentemente representados no formato *matricial* e objetos geográficos, no formato *vetorial*. O formato *matricial* é caracterizado por uma matriz de células de tamanhos regulares, onde a cada célula é associado um conjunto de valores representando as características geográficas da região correspondente. O formato *vetorial* descreve dados espaciais com uma combinação de formas geométricas (pontos, linhas e polígonos). O termo *raster* designa células retangulares, mas na maioria das vezes é usado como termo genérico para a representação matricial.

Representações *topológicas* permitem armazenar, associada à localização, informações sobre relacionamentos de contiguidade e vizinhança dos elementos armazenados.

2.2.2 Problemas Existentes em SIG

Existem vários problemas em aberto em SIG, além de problemas que resultam da incorporação do tempo nestes sistemas. Dentre estes problemas se encontram:

1. Muitos dos problemas em SIG são devidos à natureza dos dados geo-referenciados [MP94]. Estes dados são complexos, ocupam muito espaço e variam com o tempo. Além disso são geralmente provenientes de fontes diferentes de aquisição de dados, com níveis distintos de generalização e escalas incompatíveis.

2. Outros problemas originam da dimensão espacial dos dados geográficos, que introduz questões de restrições de integridade espacial e processamento de consulta espacial. As estratégias de acesso a dados para predicados espaciais podem não ser as mesmas que para outros tipos de predicados.

3. As estratégias padrão de otimização de consultas nem sempre são adequadas para dados geográficos [MP94].

4. No caso de aplicações de Geoprocessamento, a interação com o banco de dados é comumente mais longa, exigindo uma revisão dos mecanismos de controle de transações, principalmente a criação de mecanismos para versionamento, atualização cooperativa e bloqueio parcial de objetos [CCH+96, NTE92].

5. Falta um modelo espaço-temporal padrão para SIG. Para cada classe específica de aplicações, existe um ou mais modelos que melhor se adaptam. [Bot95] propõe um modelo espaço-temporal para SIG e uma taxonomia para consultas, mas não apresenta estratégias de processamento dessas consultas.

6. A implementação de estruturas de dados que representem a variação no tempo de dados espaciais, e a implementação de operadores e relacionamentos espaciais é uma questão problemática [Lan89, MP94], e é rara na literatura.

Esta dissertação propõe uma solução para o último problema. Alguns dos problemas citados são discutidos a seguir.

2.2.3 Modelagem de Dados Geográficos

A modelagem de dados geográficos difere da tradicional não apenas devido às características espaciais, mas também por envolver a questão da representação, que varia conforme a perspectiva do usuário ou aplicação, ou segundo fatores técnicos.

Segundo [CCH+96], os primeiros trabalhos sobre modelos de dados geográficos se ocupavam principalmente com estruturas geométricas e espaciais. Os modelos propostos correspondiam a estruturas de dados sofisticadas. Ainda segundo aqueles autores, esta concepção de modelo foi incorporada pela maioria dos sistemas comerciais atuais, onde o usuário realiza a "modelagem" dos dados definindo diretamente estruturas de baixo nível. No entanto, esta não é uma abordagem apropriada, pois os usuários raramente são especialistas em computação, mas sim nos diferentes domínios da aplicação.

As propostas mais antigas de modelos de dados geográficos baseiam-se no modelo de dados relacional, sendo que estudos mais recentes recomendam o uso de modelos orientados a objetos.

Armazenamento/Estruturas de Dados

Os SIGs tradicionamente armazenavam os dados geográficos em arquivos internos, mas este tipo de solução vem sendo substituído cada vez mais pelo uso de SGBDs.

Há várias pesquisas na área de armazenamento da evolução temporal de dados, mas são poucas as publicações que tratam do armazenamento da evolução temporal de dados espaciais em sistemas geográficos. As aplicações que manipulam dados espaço-temporais requerem sistemas computacionais com grande capacidade de armazenamento e alta velocidade de processamento.

Ainda não há um consenso sobre como saber quais dados geográficos devem ser temporais e que tipo de informação temporal deve ser armazenada. Langran [Lan93a] cita várias questões que devem ser tratadas antes da implementação de um SIG temporal:

- É preciso decidir quais dados devem ser temporalizados. Para isso, deve-se analisar fatores como a volatilidade do dado, a importância do dado, e as responsabilidades da organização que o utiliza.
- A atualização de dados espaciais deve ser feita ou incrementalmente ou totalmente.
- O SIG deve prover suporte para operações de generalizações temporais. Pode ser mais importante saber o que foi mais comum durante um certo intervalo de tempo do que saber precisamente a realidade em um instante específico de tempo.

Consultas Espaço-Temporais em SIG

As consultas a dados em SIG podem envolver tanto o estado de um fenômeno quanto a sua distribuição espacial e temporal [CCH+96]. Elas podem se limitar a um fenômeno específico ou a relacionamentos espaço-temporais entre fenômenos geográficos distintos.

Alguns trabalhos feitos na área de consultas geográficas têm procurado classificar um conjunto básico de consultas. [CCH+96, Bot95] referenciam uma caracterização de consultas típicas de aplicações SIG. Essa abordagem permite que o usuário faça três tipos básicos de perguntas:

- quando/onde/o que: descreve o conjunto de fenômenos geográficos (o que) presentes em uma localização ou em um conjunto de localizações (onde), dada uma referência temporal (quando). Por exemplo, "Quais os tipos de uso de solo encontrados na bacia do Rio Piracicaba no período 1980-1995?"
- quando/o que/onde: descreve uma localização ou seu conjunto (onde) ocupada por um ou vários fenômenos geográficos (o que) em um dado conjunto de intervalos de tempo (quando). Por exemplo, "Quais as áreas no estado de São Paulo ocupadas por plantações de cana no período 1950-1980?"
- o que/onde/quando: descreve o conjunto de períodos (quando) em que um determinado conjunto de fenômenos geográficos (o que) ocupou um conjunto de localizações.

Por exemplo, "Qual o período em que a região onde hoje se encontra a UNICAMP foi ocupada por uma plantação de café?"

Botelho [Bot95] cita ainda outras classificações de tipos de consultas de um SIG temporal. Uma delas é a seguinte:

- Apresentação dos dados armazenados.
- Determinação dos relacionamentos espaciais entre entidades diferentes.
- Simulação e comparação de cenários alternativos baseados na combinação de camadas de dados.
- Previsão do futuro através da análise de tendências.

Outra classificação citada por [Bot95] classifica as consultas em SIG em descritivas: "Localize e mostre todas as cidades com população maior que 10.000 habitantes", e analíticas: "Por que...?" ou "E se...?". Ainda outra classificação: consultas topológicas (adjacência, borda e co-borda), de conjuntos (coincidente, elemento-de-continência) e métricas (mínimos/máximos e raios de abrangência).

2.2.4 Um Modelo de Dados Geográfico

Câmara et al. [CCH+96] descrevem um modelo de dados geográfico que apresenta uma abordagem unificada das visões de campos e objetos e permite a existência de múltiplas representações para um mesmo fenômeno geográfico. O modelo é usado como base nesta dissertação para a modelagem de dados geográficos. Ele separa a especificação em diferentes níveis de abstração, liberando assim o usuário da necessidade de se envolver com detalhes de implementação física.

Os níveis de abstração identificados são:

- nível do mundo real: contém os elementos da realidade geográfica a serem modelados como, por exemplo, rios e redes telefônicas.
- nível conceitual: comporta as ferramentas para modelar formalmente campos e objetos geográficos em um nível alto de abstração. Determina as classes básicas orientadas a objetos que deverão ser criadas no banco de dados. Deve conter definições das operações e da linguagem de manipulação de dados disponíveis para o usuário.
- nível de representação: associa as classes de campos e objetos geográficos identificadas no nível conceitual a classes de representações, que podem variar conforme a escala, a projeção cartográfica escolhida, a época de aquisição do dado, ou mesmo conforme a visão do usuário ou aplicação.

- nível de implementação (físico ou interno): define padrões, formas de armazenamento e estruturas de dados para implementar as diferentes representações. Segundo os autores, as decisões de implementação admitem um número muito grande de variações, em função das aplicações às quais o sistema é voltado, da disponibilidade de algoritmos e do desempenho do hardware.

Nível Conceitual

Seguindo [CCH+96], existem no banco de dados três classes básicas no nível conceitual: as classes convencionais, que correspondem ao conceito padrão de classes em SGBDs orientados a objetos, e as classes que refletem as visões de campos e objetos: GeoCampo e GeoObjeto.

As instâncias de GeoCampo, chamadas de *geo-campos*, possuem atributos espaciais (Localização, Contradomínio e Mapeamento) e atributos convencionais. Os atributos espaciais são obrigatórios e formam o *componente espacial* do geo-campo. Como esta dissertação não aborda geo-campos, eles não serão detalhados.

As instâncias de GeoObjeto, chamadas de geo-objetos, possuem um atributo espacial denominado Localização. A localização de um geo-objeto pode ser explicitamente armazenada ou pode ser computada, e forma o *componente espacial* do geo-objeto. Os atributos convencionais de um geo-objeto formam o seu *componente convencional*.

Ainda um geo-objeto pode ser elementar, composto ou fraco. Um *geo-objeto elementar* é um geo-objeto que não é composto por outros geo-objetos e que sempre tem a sua localização explicitamente armazenada. Um *geo-objeto composto* é um geo-objeto que contém outros geo-objetos como componentes. Ele pode ter uma localização explicitamente armazenada, ou tê-la calculada a partir das localizações dos geo-objetos componentes. A localização do geo-objeto composto pode também delimitar uma área maior que a união das áreas das localizações dos seus componentes. Um *geo-objeto fraco* é um geo-objeto que existe somente enquanto fizer parte de um geo-objeto composto.

Nível de Representação

Uma *representação* descreve o componente espacial de um geo-objeto ou de um geo-campo. [CCH+96] propõe duas hierarquias básicas de classes de representações, com as raízes RepGeoCampo e RepGeoObjeto.

Para a classe RepGeoObjeto, o modelo oferece subclasses cujas instâncias possuem atributos cujos valores são elementos geométricos simples, como pontos, linhas e regiões sem buracos, ou elementos complexos construídos a partir destes.

As instâncias de RepGeoCampo possuem atributos muito semelhantes aos dos geo-campos: Localização, Domínio, Contradomínio e Mapeamento. O modelo oferece sub-

classes de RepGeoCampo, que variam conforme a definição do Domínio.

O modelo também introduz uma classe, REPRESENTA, para capturar a associação entre geo-campos ou geo-objetos e suas representações.

Operações e Linguagem de Consulta

Em vez de apresentar uma lista exaustiva de operações espaciais, Câmara et al. [CCH+96] discutem como definir algumas categorias de operações, enfatizando os aspectos peculiares ao modelo apresentado.

Os autores analisam o efeito que a introdução de um nível de representação, separado, provoca na definição de operações sobre geo-objetos e geo-campos, e discutem o problema de especificar relacionamentos topológicos. Eles também discutem como operações sobre geo-campos são definidas, explorando o fato de geo-campos serem funções; e mostram exemplos de operações mistas, envolvendo ao mesmo tempo geo-campos e geo-objetos.

Câmara et al. [CCH+96] apresentam uma linguagem de consulta e manipulação espacial denominada LEGAL - Linguagem Espacial para Geoprocessamento Algébrico, que se baseia no modelo de dados e nas operações definidas.

A LEGAL fornece comandos para especificar o esquema conceitual de um banco de dados geográfico e comandos para criar novos objetos no banco de dados, sejam eles geo-campos, geo-objetos ou representação destes objetos.

Uma consulta em LEGAL possui dois componentes [CCH+96]: uma expressão de busca expressa em SQL estendida e uma resposta à consulta, que pode ser objeto de manipulação posterior. Segundo [CCH+96], o usuário pode formular consultas envolvendo classes tanto de geo-objetos quanto de geo-campos. No entanto, no caso de geo-campos, os operadores disponíveis em LEGAL permitem apenas recuperação baseada em atributos convencionais. Para geo-objetos, LEGAL oferece funções e relacionamentos espaciais, computados sobre suas localizações.

2.3 Um Modelo de Dados Geográfico Temporal

A seção 2.1 mostrou um modelo de dados proposto na UNICAMP por Oliveira [Oli93] para BD temporais orientados a objetos. O trabalho Brayner [Bra94] implementou este modelo, fazendo uma série de aperfeiçoamentos.

A dissertação continua esta linha de pesquisa, implementando (e aperfeiçoando) o modelo de dados geográfico temporal de Botelho [Bot95], que por sua vez estende o modelo geográfico de Câmara et al. [CCH+96] com o tempo. Esta seção dá uma visão geral deste modelo.

2.3.1 Estrutura

O modelo proposto por Botelho [Bot95, MB96] foi baseado no modelo multi-nível citado anteriormente e no modelo TOODM [Oli93], estendidos para suportar dados espaço-temporais na dimensão de tempo de validade. [Bot95] adotou a abordagem do modelo orientado a objetos baseada em classes do sistema O_2 e se restringe a geo-objetos.

O modelo de Botelho considera não só a visão histórica do mundo, como também a visão de processos. Em outras palavras, não somente os fatos geográficos são modelados, mas também os processos de mudança geográfica.

A transição de estado de uma entidade geográfica é representada como um intervalo temporal, ao invés de um instante temporal pois, segundo ele, a maioria das entidades geográficas é formada por componentes que mudam de estado em momentos distintos.

O espaço geográfico é modelado como um conjunto de geo-objetos que se relacionam, cada um com seu estado (atributos) e comportamento (métodos). As restrições de integridade (ou regras) dos objetos geográficos são formalizadas como métodos ou como outros objetos.

Botelho [Bot95] representa um processo geográfico por um geo-objeto composto, cuja lista de geo-objetos componentes contém as entidades geográficas participantes deste processo, e cujos atributos contêm os valores (ou intervalos de valores) que caracterizam a transição de estado dessas entidades.

Para o nível de representação, [Bot95] propõe um conjunto de classes que descrevem a geometria (vetorial) de um geo-objeto, cuja classe principal é denominada Geometria. Ele propõe que as classes geométricas sejam providas pelo SGBD como classes primitivas.

As definições de classes no modelo, usando construtores de tipo do O_2, são:

```
Geo-Objeto: tuple ([atributos não-espaciais],[list (Geo-Objeto)], Localização)
Localização: list (Geometria)
Geometria: tuple ([atributos não-espaciais], list (Obj_Geom))
```

Cada instância da classe Geometria corresponde a uma representação espacial do geo-objeto e é composta por atributos não-espaciais para guardar informações específicas da representação (como tipo de projeção, escala) e um atributo espacial para guardar objetos geométricos que definem a geometria da representação. A classe Geometria é usada para encapsular as representações vetoriais do componente espacial dos geo-objetos e serve como elo de comunicação com o nível conceitual.

O componente espacial da classe Geometria é uma lista de objetos da classe abstrata Obj_Geom, que é uma generalização das classes geométricas básicas: Ponto, Linha, Polígono, Multi-Polígono. A classe Multi-Polígono é composta por uma lista de polígonos para representar regiões desconexas e uma lista de polígonos para representar buracos

desconexos. A classe Polígono é composta por uma lista de linhas. A classe Linha é composta por uma lista de pontos.

Segundo [Bot95], as operações espaciais podem ser definidas como métodos da classe Obj_Geom e redefinidas nas suas subclasses. O mecanismo de ligação tardia garante que a versão correta da implementação dessas funções seja escolhida de acordo com a classe do objeto passado como parâmetro na consulta espacial.

Botelho não considerou o tempo de transação. A evolução temporal de geo-objetos foi considerada apenas segundo o eixo de tempo de validade, e se baseou no modelo de Oliveira [Oli93]. A evolução da estrutura e comportamento de um geo-objeto, que se refere ao problema de evolução de esquemas, não é abordado.

Botelho [Bot95] descreve um geo-objeto temporal da seguinte forma:

```
geo-objeto: tuple (tuple ([atributos não-espaciais temporalizados],
                          [lista de geo-objetos componentes temporalizada],
                          Localização temporalizada),
                   atributo temporal)
```

Botelho não propõe linguagem de consulta. O processamento de consultas e suporte a operações espaço-temporais é um dos tópicos abordados na dissertação.

2.4 Resumo

Este capítulo apresentou uma breve revisão de alguns conceitos relevantes ao entendimento da dissertação, além de resumir questões relacionadas à incorporação do conceito de tempo em bancos de dados e SIG.

Foi também apresentado o modelo de Botelho, usado como base na dissertação, assim como os modelos de Oliveira e Câmara et al., estendidos por Botelho para suportar dados espaço-temporais na dimensão de tempo de validade.

Capítulo 3

Operadores Primitivos Espaço-temporais

3.1 Introdução

O principal objetivo deste capítulo é definir as operações básicas que podem ser utilizadas para a formulação de consultas em bancos de dados geográficos e que envolvem espaço e tempo. Além disso, é sugerida uma classificação das consultas visando estruturar o estudo do problema.

Dado que os componentes fundamentais de informações geográficas são atributos temáticos (ou convencionais), componente espacial e possivelmente tempo, as consultas em SIGs podem, de uma forma geral, ser divididas em: consultas *temáticas*, consultas *espaciais*, consultas *temporais*, e consultas *espaço-temporais*. As consultas temáticas se referem ao conteúdo de atributos não espaciais; as consultas espaciais podem ser de localização, métricas, de orientação, ou topológicas; as consultas temporais são aquelas onde o tempo é um parâmetro, e que consideram relações temporais entre ocorrência de fenômenos; finalmente, as consultas espaço-temporais combinam características das consultas espaciais e temporais.

Nos exemplos dados no decorrer do capítulo serão consideradas as seguintes entidades: fazendas, municípios, zonas urbanas (ou cidades), rios, postes, estradas (ou rodovias) e linhas telefônicas. Supõe-se que fazendas estão situadas em municípios (compostos por zona urbana e zona rural); podem conter rios, postes de eletricidade e telefone; podem ser intersectadas por estradas (ou rodovias) e rios; e podem ter acesso a linhas telefônicas.

Consultas temáticas (estritamente convencionais, envolvendo tempo ou não) não serão abordadas aqui, já que são objeto de vários estudos e implementações [Bra94, Oli93, Jen93, EO94].

O enfoque adotado neste trabalho é que consultas espaço-temporais envolvem predi-

cados espaciais e temporais. Predicados espaciais se referem ao relacionamento espacial entre objetos, ou atuam sobre um atributo espacial ou sobre um atributo calculado a partir de um atributo espacial. Predicados temporais se referem ao relacionamento temporal entre objetos, ou atuam sobre um atributo temporal ou sobre um atributo calculado a partir de um atributo temporal.

Exemplos de consultas com predicados espaciais são:

a) Quais as fazendas com área maior que 3000 ha?

b) Quais as fazendas atravessadas por um rio?

No primeiro exemplo, o predicado espacial se refere à área de objetos do tipo fazenda; no segundo, a um relacionamento espacial (atravessado) entre objetos fazenda e rio.

Exemplos de consultas com predicados temporais são:

a) Quais as fazendas foram registradas após o ano de 1964?

b) Quais as fazendas foram registradas antes do município de Campinas?

No primeiro caso, o predicado temporal se refere ao valor calculado (ínicio) obtido do componente temporal dos objetos do tipo fazenda; no segundo, a um relacionamento temporal (antes) entre os início dos objetos fazenda e município.

Finalmente, consultas espaço-temporais envolvem combinação de ambos predicados: espaciais e temporais. Exemplos:

a) Quais as fazendas que tinham área maior que 3000 ha antes de 1950?

b) Quais as fazendas atravessadas por um rio entre 1970 e 1980?

O predicado espacial se refere à *área* de objetos fazenda no exemplo *(a)*, e ao relacionamento espacial (atravessado) entre objetos fazenda e rio no exemplo *(b)*. Ambos os predicados temporais se referem aos atributos temporais dos objetos considerados.

Este capítulo está organizado da seguinte forma: a seção 3.2 define os operadores espaciais, temporais e espaço-temporais que serão utilizados como base do trabalho; a seção 3.3 apresenta a notação adotada para a representação de consultas usando estes operadores, além de classificar algumas consultas espaciais, temporais e espaço-temporais; e a seção 3.4 apresenta exemplos de como consultas mais complexas podem ser escritas através da combinação dos operadores apresentados.

3.2 Operadores

A formulação de consultas em bancos de dados geográficos pode utilizar operadores espaciais, temporais e espaço-temporais. Os seus parâmetros (operandos) são valores, objetos (temporais, espaciais, ou espaço-temporais) ou conjuntos destes.

Para entendimento deste capítulo, deve-se lembrar que um objeto espaço-temporal é composto de três componentes básicos (vide capítulo 2): componente convencional, componente espacial e componente temporal. Objetos espaço-temporais são criados a partir de agregação do tempo a objetos espaciais e seus componentes. Assim a dimensão temporal pode se aplicar tanto ao componente convencional quanto ao componente espacial. Um objeto espaço-temporal pode ser composto por vários objetos espaço-temporais a partir da aplicação de construtores de tipo (tuple, list, set).

Seja A um objeto espaço-temporal. Então, A = ⟨[CT], SpT, T⟩, onde CT é o componente convencional de A (possivelmente associado a marcas de tempo); SpT é o componente espacial de A (associado a marcas de tempo); e T é o componente temporal de A, que representa o tempo de vida do objeto. Um componente (espacial ou convencional) pode assumir diversos valores ao longo do tempo. Assim, SpT é um conjunto de tuplas da forma $\langle Sp_i, T_i \rangle$, onde Sp_i é o valor do componente espacial em T_i. Analogamente, CT (se associado a marcas de tempo) é um conjunto de tuplas da forma $\langle C_i, T_i \rangle$, sendo C_i o valor do componente convencional em T_i. Se CT não estiver associado a marcas de tempo, CT=C.

Considera-se que objetos espaciais e temporais são casos especiais sem componentes temporais e espaciais, respectivamente. Assim, se A é um objeto espacial, A = ⟨[C], Sp⟩, onde C é o componente convencional de A e Sp, o seu componente espacial. Se A é um objeto temporal, então A = ⟨[CT], T⟩, onde CT é o componente convencional de A (possivelmente associado a marcas de tempo) e T é o tempo de validade de A.

Representamos um operador por OPER (A,B,C,...), sendo OPER o nome do operador e A, B, C, etc., os seus operandos.

3.2.1 Operadores Espaciais

Os operadores espaciais podem ser de localização, de orientação, métricos, e topológicos. Os operadores espaciais têm como operandos objetos geográficos (operador localização) ou suas localizações (operadores de orientação, métricos e topológicos). A sua aplicação depende da representação dos objetos. Nos exemplos dados neste texto considera-se que um rio, uma estrada e uma rodovia são representados por linhas, uma cidade e uma fazenda são representadas por polígonos, e um poste é representado por um ponto.

Operador Localização

O operador localização, denotado SP, retorna a localização do objeto passado como parâmetro.

A localização de um objeto A corresponde a uma representação espacial do objeto, sendo composta por objetos geométricos (delimitados por um conjunto de coordenadas geográficas) que descrevem a geometria de A.

Operadores de Orientação

As operadores de orientação verificam se existe um determinado relacionamento de orientação entre dois conjuntos de objetos geométricos. Os relacionamentos de orientação (ou relacionamentos direcionais) descrevem como os objetos estão posicionados entre si [CCH+96, Cil96]. Ou lidam com ordem no espaço [PTS94]. Exemplos de relacionamentos de orientação encontrados na literatura são: *above, below, north, south, east, west, left, right, same_position, northeast.*

A definição de relacionamentos de orientação em geral envolve um marco de referência, um objeto de referência e o objeto em questão. O marco de referência determina a direção na qual o objeto em questão está localizado em relação ao objeto de referência. Estudos sobre sentenças envolvendo relações espaciais em linguagem natural revelam que os relacionamentos direcionais dependem de aspectos cognitivos, que variam culturalmente [FM91].

Existem várias alternativas para a definição de relacionamentos de orientação entre dois objetos do tipo linha ou polígono, dependendo da quantidade de pontos que representam cada objeto [PS93, PTS94]. Por exemplo, quando o objeto é representado por um único ponto, pode ser utilizado o centróide. Por outro lado, quando um objeto é representado por dois pontos, pode ser usado o limite inferior esquerdo e o limite superior direito do r.e.m. (retângulo envolvente mínimo) do objeto em questão. O r.e.m. de um conjunto de objetos no $\mathbb{R}^2$ é o menor retângulo com lados paralelos aos eixos X e Y que contém todos os objetos do conjunto [CCH+96].

Para os propósitos deste trabalho são considerados dois relacionamentos de orientação: *north* e *east*. Outros relacionamentos direcionais podem ser expressos a partir destes. Por exemplo, o relacionamento *northeast* entre x_1 e x_2 pode ser expresso como "x_1 *north* x_2 e x_1 *east* x_2", e o relacionamento *south*, como "x_2 *north* x_1".

Nas definições de relacionamentos de orientação abaixo, baseadas em [TP95], considera-se que p_i é um ponto do objeto p, que q_i é um ponto do objeto de referência q, e que X e Y são funções que retornam, respectivamente, a coordenada x e a coordenada y de um ponto.

- O relacionamento *east* é tal que

p *east* q $\rightarrow \forall p_i\ \forall q_i$, X $(p_i) \geq$ X (q_i)

- O relacionamento *north* é tal que

  p *north* q $\rightarrow \forall p_i\ \forall q_i$, Y $(p_i) \geq$ Y (q_i)

Tendo como base estes relacionamentos direcionais, são definidos os seguintes operadores de orientação:

- NORTH (A,B): retorna o valor lógico verdadeiro se todos os elementos de A têm o relacionamento *north* com todos os elementos de B, e falso caso contrário. Ou seja,

  NORTH (A,B) $\rightarrow \forall$a $\in$ A, $\forall$b $\in$ B (a *north* b)

- EAST (A,B): retorna o valor lógico verdadeiro se todos os elementos de A têm o relacionamento *east* com todos os elementos de B, e falso caso contrário.

  EAST (A,B) $\rightarrow \forall$a $\in$ A, $\forall$b $\in$ B (a *east* b)

Operadores Métricos

Os operadores métricos geram, a partir de um ou mais objetos, um escalar que representa uma propriedade intrínseca aos objetos analisados [Cil96].

Os operadores métricos podem ser classificados em:

- unários: calculam um valor escalar usando um único conjunto de objetos;
- binários: calculam um valor escalar usando dois conjuntos de objetos.

Considera-se aqui os seguintes operadores métricos:

- AREA (A): calcula a área ocupada pelo conjunto de polígonos A.
- LENGTH (A): retorna o comprimento de A (conjunto de linhas).
- PERIMETER (A): retorna o perímetro de A (conjunto de polígonos).
- DISTANCE (A,B): calcula a distância entre dois conjuntos (A e B) de objetos geométricos (pontos, linhas, ou polígonos).

  Tendo como base [TP95], define-se aqui a distância entre dois objetos geométricos p e q (oo_dist) como a distância euclideana entre os centróides dos dois objetos. Usando a distância entre dois objetos, define-se a distância entre um objeto p e um conjunto de objetos Q (oc_dist) como a distância mínima de p a qualquer dos elementos de Q:

oc_dist (p,Q) = min (oo_dist(p,q), ∀ q ∈ Q).

Finalmente, a distância entre dois conjuntos de objetos P e Q é definida como a distância máxima de todas as distâncias oc entre os objetos de P e Q:

DISTANCE (P,Q) = max (oc_dist (p,Q), ∀ p ∈ P).

Operadores Topológicos

Os operadores topológicos retornam um valor lógico verdadeiro se há um determinado relacionamento topológico entre dois conjuntos de objetos geométricos.

Os relacionamentos topológicos são invariantes face a transformações de escala, translação e rotação [CCH+96, CdFvO93]. Os relacionamentos usados aqui foram originalmente introduzidos em Clementini et al. [CdFvO93]. São eles: *disjoint, inside, touch, cross* e *overlap*. Segundo aqueles autores, estes relacionamentos são mutuamente exclusivos e são suficientes para representar todas as situações topológicas possíveis entre dois objetos bidimensionais.

No que se segue, como em [CCH+96, CdFvO93], considera-se o espaço topológico $\mathbb{R}^2$. Os *elementos topológicos simples* são de três tipos: um *ponto*; uma *linha simples*, que não se intercepta a si mesma e que é ou *circular* ou possui apenas dois *pontos terminais*; e uma *região simples*, que é conectada, ou seja, que não é a união de conjuntos disjuntos de pontos, e que não contém buracos.

A *dimensão* de um conjunto de elementos topológicos simples Ω é dada por:

$dim(\Omega) = - \leftrightarrow \Omega = \emptyset$

$dim(\Omega) = 0 \leftrightarrow \Omega$ contém pelo menos um ponto e nenhuma linha ou região simples

$dim(\Omega) = 1 \leftrightarrow \Omega$ contém pelo menos uma linha e nenhuma região simples

$dim(\Omega) = 2 \leftrightarrow \Omega$ contém pelo menos uma região simples

A *fronteira* de um elemento topológico simples ω, denotada por $\delta\omega$, é definida da seguinte forma:

$\delta\omega = \emptyset \leftrightarrow \omega$ é um ponto ou ω é uma linha circular

$\delta\omega = \{P,Q\} \leftrightarrow \omega$ é uma linha não circular e P e Q são seus pontos terminais

$\delta\omega = L \leftrightarrow \omega$ é uma região simples e L é a linha circular formada por todos os pontos de acumulação [RS94, Lim76] de ω.

O *interior* de um elemento topológico ω, denotado por ω^0, é definido como

$\omega^0 = \omega - \delta\omega$.

Note que o interior de um ponto ou linha circular é igual ao próprio elemento.

Nas definições de relacionamentos topológicos apresentadas a seguir, ω_1 e ω_2 denotam dois elementos topológicos simples dos tipos indicados em cada caso. As figuras 3.1, 3.2,

3.3, 3.4 e 3.5 ilustram estes relacionamentos.

- O relacionamento *disjoint*, aplicável a todas as situações é tal que

  $$\omega_1 \; disjoint \; \omega_2 \leftrightarrow (\omega_1 \cap \omega_2 = \emptyset)$$

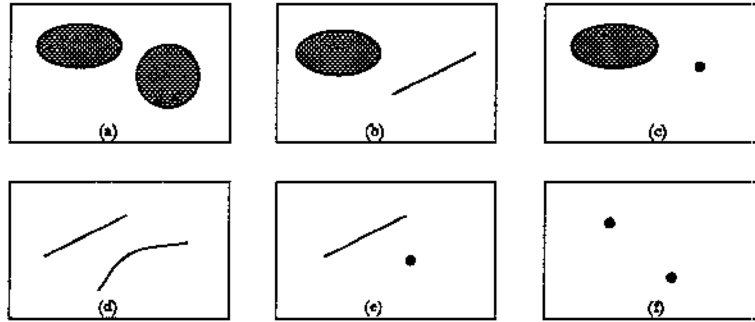


Figura 3.1: Situações ilustrando o relacionamento *disjoint* entre duas regiões (a); uma linha e uma regiões (b); uma região e um ponto (c); duas linhas (d); uma linha e um ponto (e); e dois pontos (f)

- O relacionamento *touch*, aplicável a dois elementos dos tipos região e região, linha e região, linha e linha, ponto e região, ponto e linha, é tal que

  $$\omega_1 \; touch \; \omega_2 \leftrightarrow (\omega_1 \cap \omega_2 \neq \emptyset) \wedge (\omega_1^0 \cap \omega_2^0 = \emptyset)$$

- O relacionamento *overlap*, aplicável a região e região, linha e linha, é tal que

  $$\omega_1 \; overlap \; \omega_2 \leftrightarrow (\omega_1 \cap \omega_2 \neq \omega_1) \wedge (\omega_1 \cap \omega_2 \neq \omega_2) \wedge \dim(\omega_1^0 \cap \omega_2^0) = \dim(\omega_1^0) = \dim(\omega_2^0)$$

- O relacionamento *inside*, aplicável a todas as situações, é tal que

  $$\omega_1 \; inside \; \omega_2 \leftrightarrow (\omega_1^0 \cap \omega_2^0 \neq \emptyset) \wedge (\omega_1 \cap \omega_2 = \omega_1)$$

- O relacionamento *cross*, aplicável a dois elementos dos tipos linha e região, linha e linha, é tal que

  $$\omega_1 \; cross \; \omega_2 \leftrightarrow (\omega_1 \cap \omega_2 \neq \omega_1) \wedge (\omega_1 \cap \omega_2 \neq \omega_2) \wedge \dim(\omega_1^0 \cap \omega_2^0) = (\max(\dim(\omega_1^0), \dim(\omega_2^0)) - 1)$$

Um relacionamento ou operador r é simétrico se e somente se $(\omega_1 \; r \; \omega_2) \leftrightarrow (\omega_2 \; r \; \omega_1)$. Com a exceção de *inside*, todos os relacionamentos definidos anteriormente são simétricos.

Os operadores sobre relacionamentos topológicos considerados aqui são os seguintes:

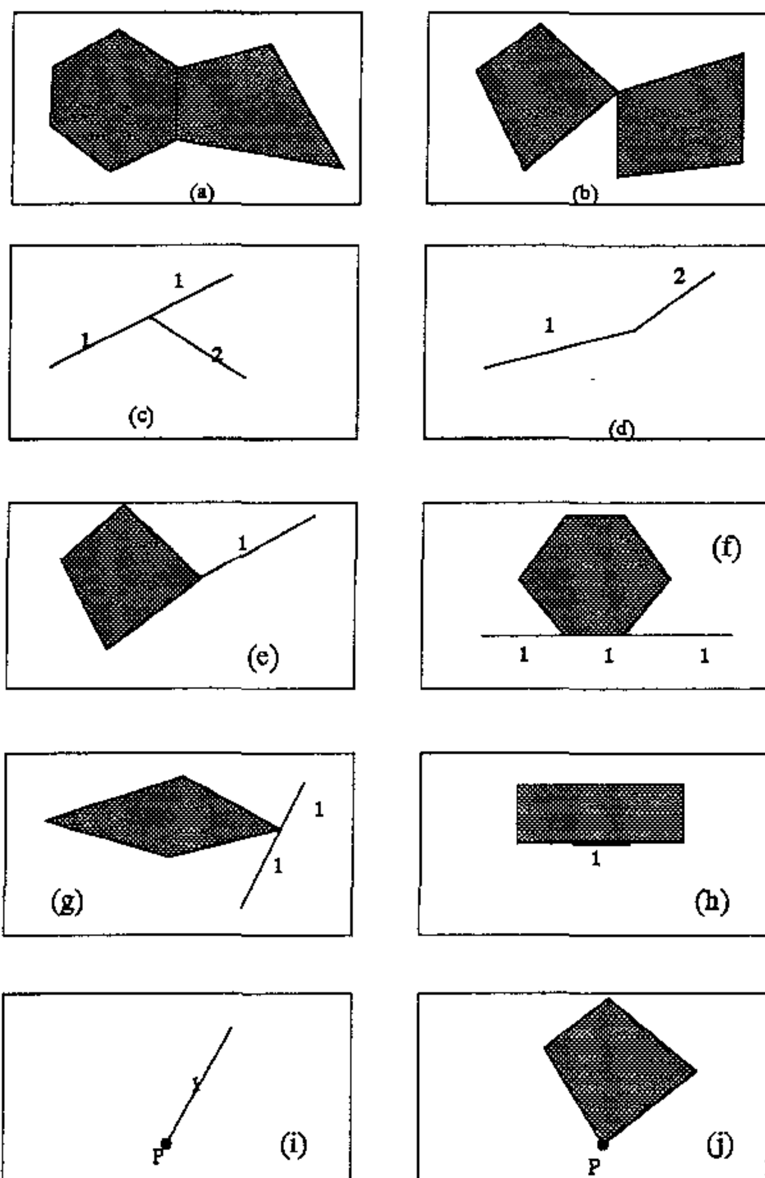


Figura 3.2: Situações ilustrando o relacionamento *touch* entre duas regiões (a) e (b); duas linhas (c) e (d); uma linha e uma região (e)-(h); um ponto e uma linha (i); e um ponto e uma região (j)

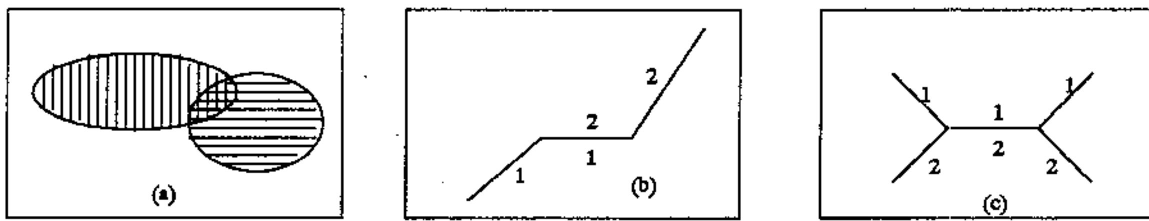


Figura 3.3: Situações ilustrando o relacionamento *overlap* entre duas regiões (a); e duas linhas (b) e (c)

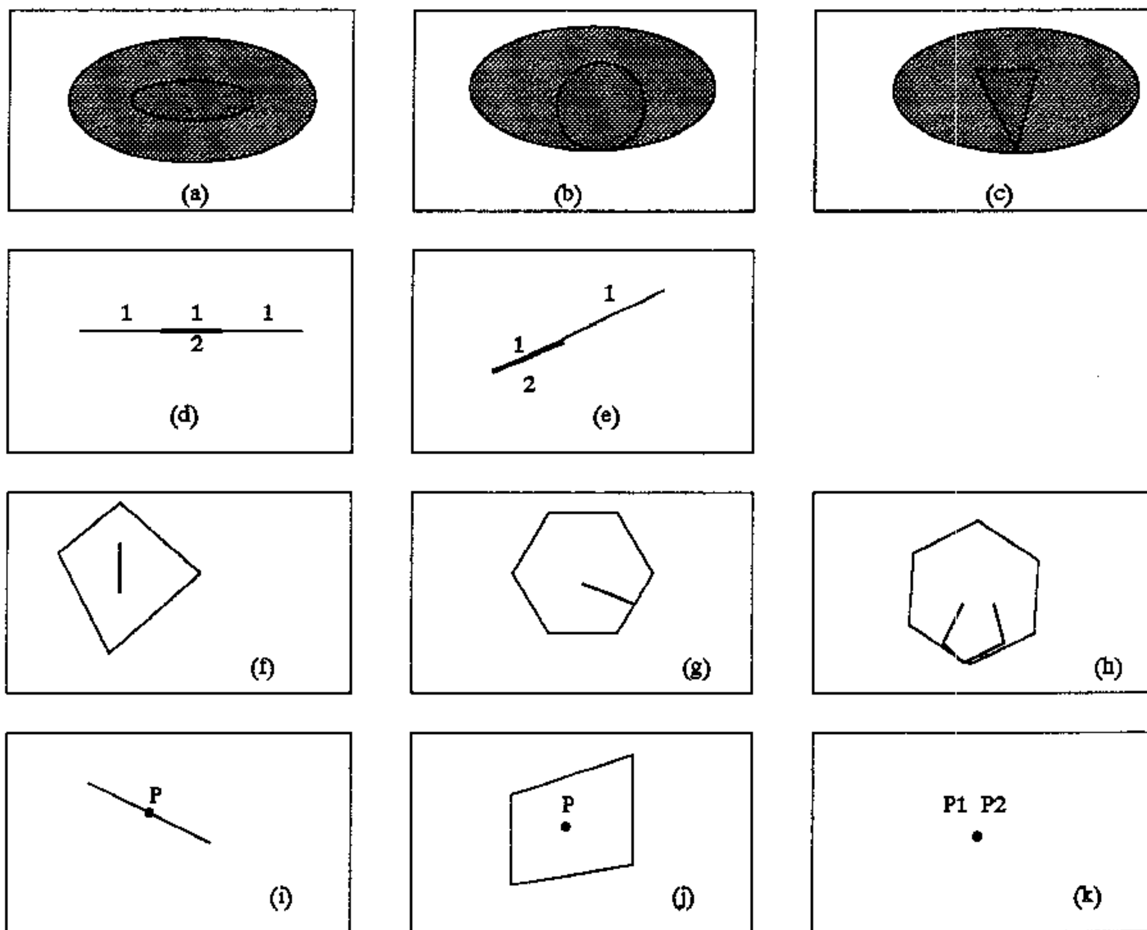


Figura 3.4: Situações ilustrando o relacionamento *inside* entre duas regiões (a) - (c); duas linhas (d) e (e); uma linha e uma região (f) - (h); um ponto e uma linha (i); um ponto e uma região (j); e dois pontos (k)

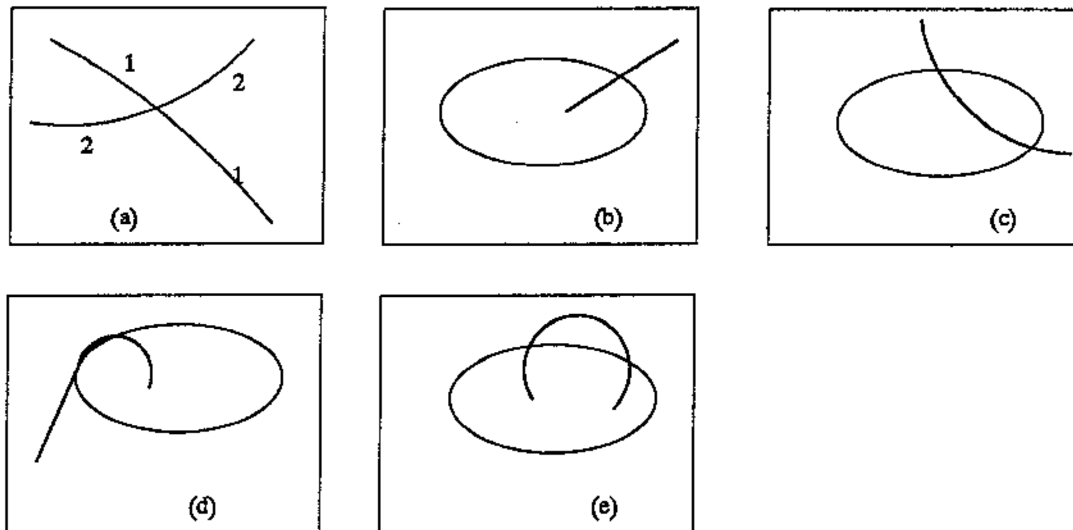


Figura 3.5: Situações ilustrando o relacionamento *cross* entre duas linhas (a); e uma linha e uma região (b) - (e)

- DISJOINT (A,B): retorna o valor lógico verdadeiro se cada elemento de A tem o relacionamento *disjoint* com cada elemento de B, e falso caso contrário. Ou seja,

  DISJOINT (A,B) $\rightarrow \forall a \in A, \forall b \in B$ (a *disjoint* b)

- TOUCH (A,B): retorna o valor lógico verdadeiro se existe algum par a, b de elementos de A e B que têm o relacionamento *touch*, e falso caso contrário. Ou seja,

  TOUCH (A,B) $\rightarrow \exists a \in A, \exists b \in B$ (a *touch* b)

- OVERLAP (A,B): retorna o valor lógico verdadeiro se existe algum par a, b de elementos de A e B que têm o relacionamento *overlap*, e falso caso contrário.

  OVERLAP (A,B) $\rightarrow \exists a \in A, \exists b \in B$ (a *overlap* b)

- INSIDE (A,B): retorna o valor lógico verdadeiro se para todo $a \in A$ existe algum b $\in$ B tal que a *in* b, e falso caso contrário.

  INSIDE (A,B) $\rightarrow \forall a \in A, \exists b \in B$ (a *in* b)

- CROSS (A,B): retorna o valor lógico verdadeiro se existe algum par a, b de elementos de A e B que têm o relacionamento *cross*, e falso caso contrário.

  CROSS (A,B) $\rightarrow \exists a \in A, \exists b \in B$ (a *cross* b)

3.2.2 Operadores Temporais

Os operadores temporais podem ser unários ou binários; retornando valores de tempo (ou marcas de tempo), valores booleanos, ou objetos (temporais ou espaço-temporais). Eles têm como operandos valores de tempo ou objetos (temporais ou espaço-temporais).

No que se segue, considera-se que os valores de tempo podem ser de três tipos: um chronon (também chamado ponto ou instante de tempo), um intervalo de tempo ou um elemento temporal. Segundo Jensen et al. [JCE+94], um intervalo de tempo é o tempo entre dois instantes, e um elemento temporal é um conjunto finito de intervalos de tempo n-dimensionais. Um ponto de tempo pode ser representado por um intervalo degenerado, onde o tempo inicial (T_S) é igual ao tempo final (T_E).

Operadores Unários

Os operadores unários retornam valores de tempo. São definidos aqui os seguintes:

- TV (A): retorna o componente temporal de A (objeto). Considera-se aqui que o componente temporal é um valor de tempo.
- BEGIN (A): retorna o valor de tempo que representa o início de A (valor de tempo). Ou seja:

  BEGIN (A) = $\{t \mid t \in A \wedge \neg\exists t_1\ (t_1 \in A \wedge t_1 < t)\}$, onde t é um instante de tempo.

- END (A): retorna o instante final de A (valor de tempo). Ou seja:

  END (A) = $\{t \mid t \in A \wedge \neg\exists t_1\ (t_1 \in A \wedge t_1 > t)\}$

- DAY (A), MONTH (A) e YEAR (A): retornam, respectivamente, *day, month, year*, considerando que A (chronon) tenha uma granularidade[1] composta formada pelos elementos (*year, month, day*).
- TWHEN (A): retorna o tempo de validade de um relacionamento espaço-temporal, ou seja, o tempo em que este relacionamento é verdadeiro. Assim, A é um conjunto de valores lógicos e tempo, sendo que TWHEN (A) retorna os valores de tempo cujo valor lógico associado é verdadeiro.

Operadores Binários

São definidos aqui três tipos de operadores binários: os operadores booleanos, que verificam um determinado relacionamento temporal entre dois valores de tempo; o operador

[1] A granularidade de um valor de tempo é a precisão com a qual esse valor pode ser representado.

VSLICE, que seleciona alguns dos estados temporais de um objeto; o operador T_INTER, que retorna a interseção temporal entre dois valores de tempo; e o operador INTERVAL, que retorna um intervalo de tempo.

Tendo [Sno95b, SAA+94a, VBH96, BVH96] como base, considera-se os seguintes operadores temporais booleanos: T_BEFORE, T_OVERLAPS, T_EQUAL, T_CONTAINS e T_MEETS. Eles buscam relacionamentos temporais entre dois valores de tempo. Outros relacionamentos temporais, como *after, during, starts, finishes, disjoint*, definidos em [All83, RP92, GS89] podem ser expressos com a utilização dos operadores acima. Por exemplo, o relacionamento *disjoint* entre x_1 e x_2 pode ser expresso como "T_BEFORE (x_1, x_2) OR T_BEFORE (x_2, x_1)".

Nas definições de operadores temporais apresentadas a seguir, A e B denotam dois valores de tempo; T_S e T_E denotam o primeiro e o último instante de seu operando (que é um valor de tempo), respectivamente; e t denota um instante de tempo qualquer. As figuras 3.6, 3.7, 3.8, 3.9 e 3.10 ilustram os relacionamentos temporais buscados por estes operadores.

- O operador T_BEFORE (A,B) retorna o valor lógico verdadeiro quando o último instante de A é anterior ao primeiro instante de B, e falso caso contrário. Ou seja,

  T_BEFORE (A,B) $\rightarrow T_E$ (A) $< T_S$ (B)

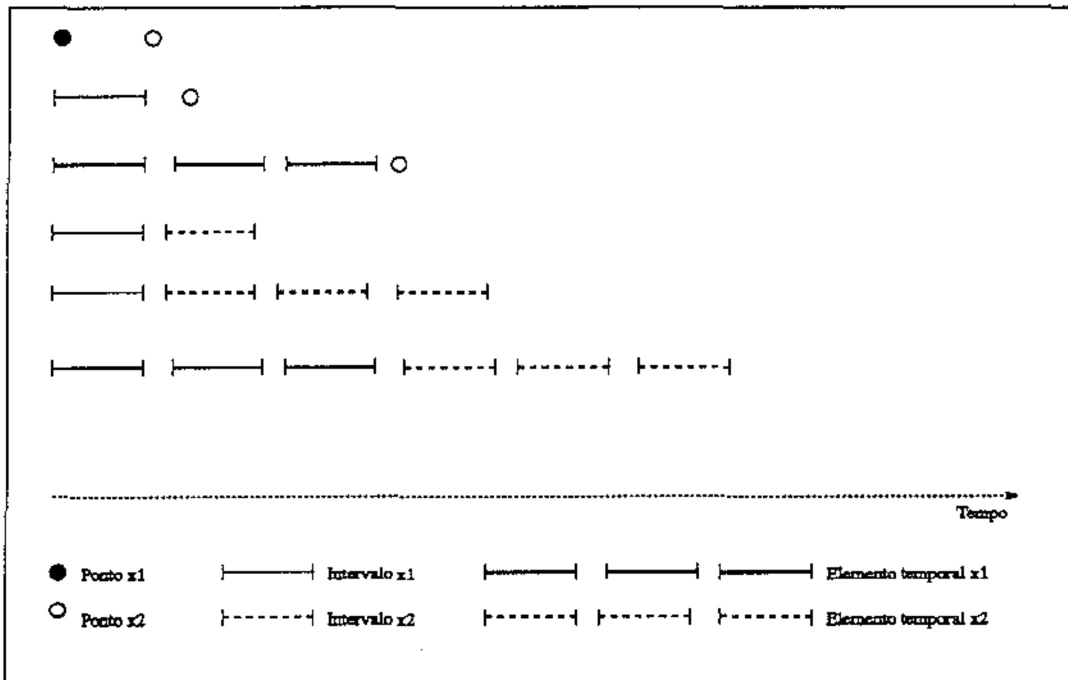


Figura 3.6: Situações ilustrando o relacionamento temporal *t_before*

- O operador T_OVERLAPS (A,B) retorna o valor lógico verdadeiro quando os dois operandos se sobrepõem em algum instante de tempo, e falso caso contrário. Ou seja,

  T_OVERLAPS (A,B) $\rightarrow \exists$t (t $\in A \wedge t \in B$)

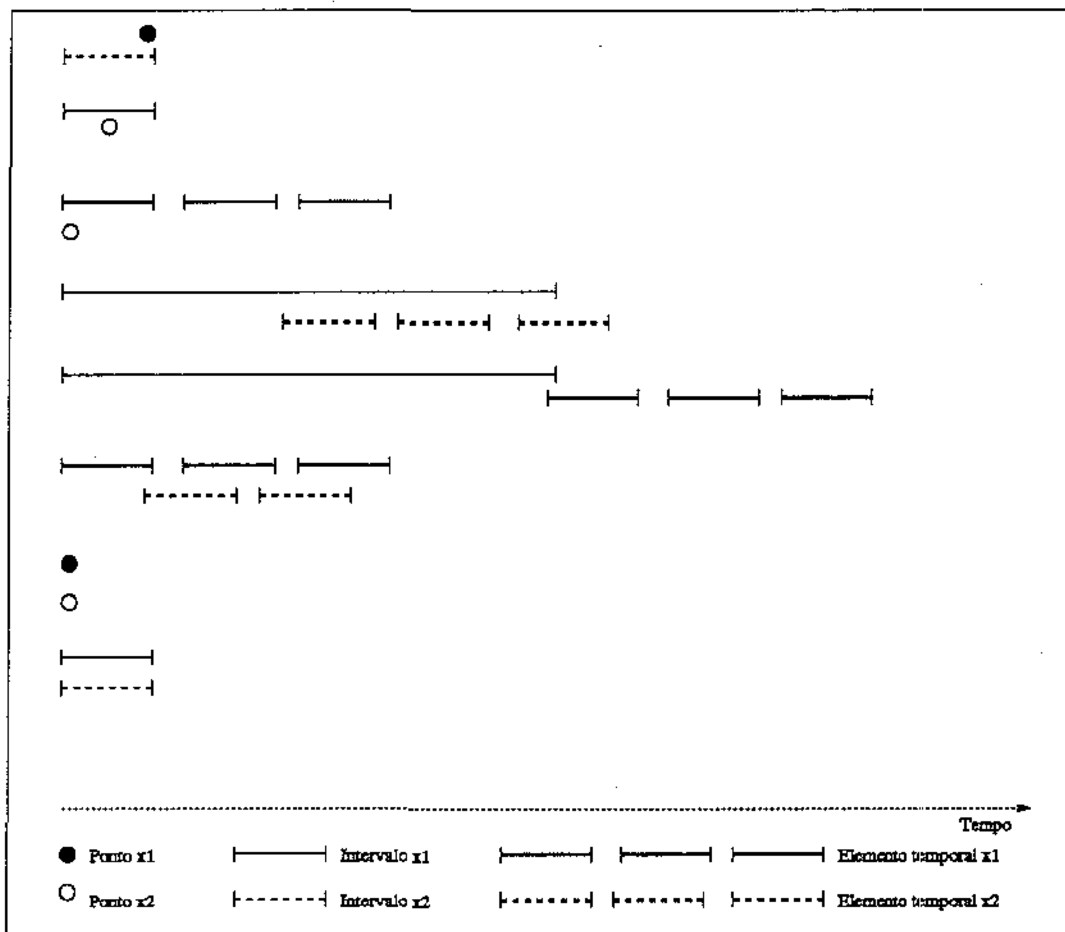


Figura 3.7: Situações ilustrando o relacionamento temporal *t_overlaps*

- O operador T_EQUAL (A,B) retorna o valor lógico verdadeiro quando A apresenta uma marca de tempo equivalente à marca de tempo apresentada por B, e falso caso contrário. Ou seja,

  T_EQUAL (A,B) $\rightarrow$ A = B

- O operador T_CONTAINS (A,B) retorna o valor lógico verdadeiro quando A contém todos os valores de tempo pertencentes a B, e falso caso contrário. Ou seja,

  T_CONTAINS (A,B) $\rightarrow \forall t \in B$, t $\in A$

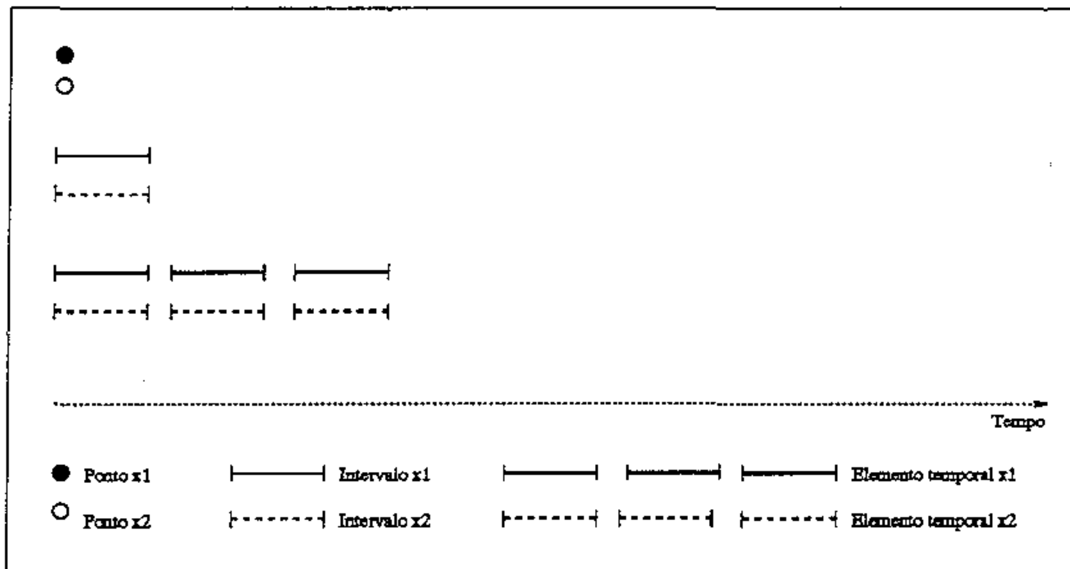


Figura 3.8: Situações ilustrando o relacionamento temporal *t_equal*

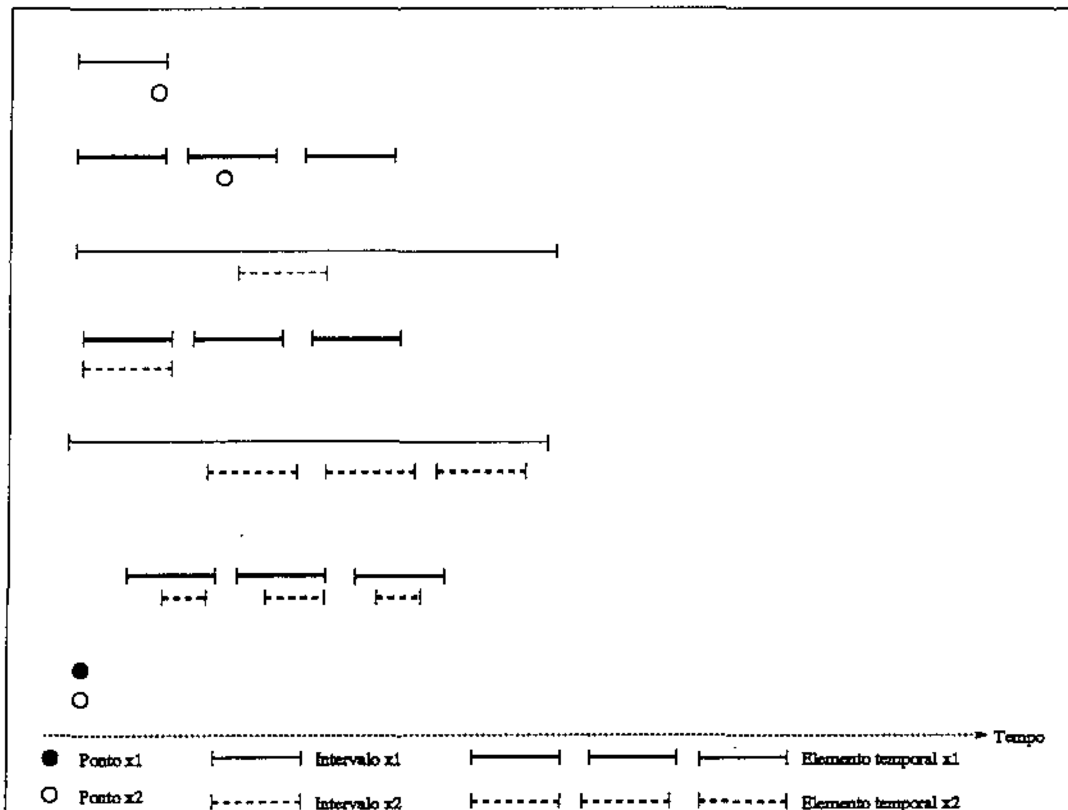


Figura 3.9: Situações ilustrando o relacionamento temporal *t_contains*

- O operador T_MEETS (A,B) retorna o valor lógico verdadeiro quando B inicia-se no instante seguinte ao instante final de A, e falso caso contrário. Ou seja,

  T_MEETS (A,B) $\rightarrow T_S$ (B) $- T_E$ (A) $= 1$, onde 1 representa um chronon.

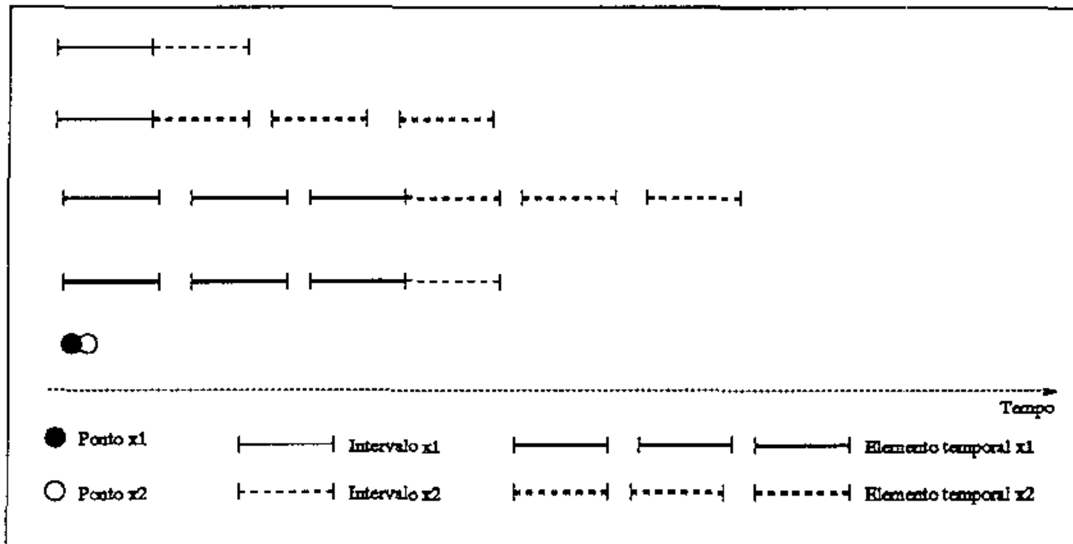


Figura 3.10: Situações ilustrando o relacionamento temporal *t_meets*

Dos operadores definidos acima, apenas T_EQUAL e T_OVERLAPS são simétricos. Outros operadores binários são os seguintes:

- O operador VSLICE (A,T) retorna o parâmetro A (objeto temporal ou espaço-temporal) reduzido apenas àqueles tempos especificados pelo parâmetro T. Em outras palavras, VSLICE retorna o objeto A apenas com os estados válidos no tempo T.

  Um *estado* de um banco de dados é definido pelo valor de seus atributos. Cada alteração efetuada no valor de um dos atributos determina um novo estado do banco de dados. Um estado tem, portanto, determinada duração, que é o intervalo de tempo durante o qual nenhum valor de atributo foi alterado [EO94].

- O operador T_INTER (A,B) retorna um valor de tempo que corresponde à interseção entre os valores de tempo A e B.

- O operador INTERVAL (s,e) retorna um intervalo de tempo tendo tempo de início s e tempo de fim e.

3.2.3 Operadores Espaço-temporais

Estes operadores estendem os operadores espaciais, definidos na seção 3.2.1 pela adição da dimensão temporal. Seus operandos são um ou dois objetos espaço-temporais; ou um objeto espaço-temporal e um objeto espacial. Os operadores espaço-temporais podem ser localização-temporal, direcionais-temporais, métrico-temporais, e topológico-temporais.

Operador Localização-temporal

O operador localização-temporal, denotado ST_SP (A,T), retorna as localizações do objeto A válidas no tempo T. Informalmente tem-se:

ST_SP (A,T) = $\{\langle Sp_i, T_i \rangle\}$

onde Sp_i é o conjunto de objetos geométricos que descrevem a geometria de A no tempo T_i; e todos T_i são disjuntos e contidos em T.

Se T for um instante de tempo, ST_SP (A,T) retorna apenas a localização de A no tempo T.

Operadores Direcionais-temporais

Os operadores direcionais-temporais, também chamados operadores espaço-temporais de orientação, retornam um valor lógico indicando se há um determinado relacionamento de orientação entre dois objetos A e B, para cada intervalo de interseção temporal (no domínio de tempo T).

Considera-se aqui os operadores direcionais-temporais: ST_NORTH e ST_EAST. De forma análoga aos operadores espaciais de orientação, outros relacionamentos direcionais temporais podem ser expressos a partir destes operadores.

Operadores Métrico-temporais

Os operadores métrico-temporais, também chamados espaço-temporais métricos, podem envolver um ou dois objetos. No primeiro caso o operador recebe como parâmetro um objeto (A) e um valor de tempo (T) sobre o qual este operador deve ser aplicado; retornando uma lista de valores numéricos associados aos respectivos valores de tempo em que a localização do objeto A é a mesma durante o tempo T. Em particular são definidos: ST_AREA (A,T), ST_LENGTH (A,T) e ST_PERIMETER (A,T). Estes operadores retornam, respectivamente, pares ⟨área, tempo⟩, ⟨comprimento, tempo⟩, ⟨perímetro, tempo⟩ para cada estado do componente espacial de A válido em T.

O segundo caso é representado pelo operador ST_DISTANCE (A,B,T). Ele retorna a distância entre os componentes espaciais de A e B (válidos em T) para todos intervalos de interseção temporal entre estes componentes.

Por exemplo, ST_DISTANCE (A,B,[t1,t20]) = {10,[t1,t5]; 15,[t6,t7]; 18,[t8,t20]} mostra que a distância entre A e B varia no tempo de 10 para 15 para 18 nos intervalos indicados, sendo esta distância calculada para determinada representação predefinida de A e de B.

Operadores Topológico-temporais

Os operadores topológico-temporais, também chamados operadores espaço-temporais topológicos, retornam uma lista de valores lógicos associados aos respectivos intervalos em que a localização de um objeto A e a localização de um objeto B são constantes durante um tempo T. Estes valores lógicos indicam se há um determinado relacionamento topológico entre as localizações no tempo considerado.

São propostos aqui os seguintes operadores topológico-temporais: ST_DISJOINT, ST_TOUCH, ST_OVERLAP, ST_INSIDE e ST_CROSS.

Por exemplo, ST_TOUCH (A,B) = {true,[t1,t2]; false,[t3,t5]; true,[t9,Now]} indica que A e B são adjacentes nos intervalos [t1,t2] e [t9,Now].

3.2.4 Outros Operadores

Foram definidos aqui, além dos operadores espaciais, temporais e espaço-temporais, outros operadores, também necessários para a elaboração de consultas espaço-temporais. São eles:

- IS_S_TRUE (A): retorna o valor lógico verdadeiro se existe ao menos um valor verdadeiro na lista de valores lógicos (e, possivelmente, outros atributos) que recebe como parâmetro, e falso caso contrário.
- IS_A_TRUE (A): retorna o valor lógico verdadeiro se todos os valores da lista de valores lógicos que recebe como parâmetro forem verdadeiros, e falso caso contrário.
- GT (A,v), GE(A,v), LT(A,v), LE(A,v), EQ(A,v), NE(A,v): recebem como parâmetro uma lista de valores numéricos (e, possivelmente, outros atributos) A, retornando um valor lógico verdadeiro se todos os elementos da lista são, respectivamente, maiores, maiores ou iguais, menores, menores ou iguais, iguais ou diferentes de v.

Outros operadores espaciais, temporais ou espaço-temporais podem ser definidos a partir dos operadores apresentados. Exemplos são:

- Operadores espaciais tendo como parâmetros objetos espaciais ou espaço-temporais. Seja, por exemplo, um operador DISTANCIA, cujos parâmetros são objetos espaciais. Este operador pode ser especificado a partir dos operadores definidos na seção 3.2.1, da seguinte forma: DISTANCIA (A,B) = DISTANCE (SP (A), SP (B)).

- Operadores temporais tendo como parâmetros objetos temporais ou espaço-temporais. Por exemplo, pode-se especificar um operador BEFORE (A,B), onde A e B objetos temporais, usando os operadores da seção 3.2.2, da seguinte maneira:

  BEFORE (A,B) = T_BEFORE (TV (A), TV (B)).

3.3 Consultas

Esta seção mostra como os operadores definidos anteriormente podem ser utilizados para expressar uma grande gama de consultas.

As consultas a bancos de dados geográficos podem ser espaciais, temporais ou espaço-temporais. Elas combinam predicados nas dimensões temporais com predicados nas dimensões espaciais.

3.3.1 Notação

Tsotras et al. [TJS97] propõem uma notação geral para consultas espaço-temporais. Apesar de ser simples e classificar as consultas, esta notação é restrita a um único conjunto de dados, além de suportar apenas consultas conjuntivas, e predicados de interseção e de continência, envolvendo intervalos e pontos.

É proposta aqui uma notação que permite a utilização de vários operadores (temporais, espaciais, espaço-temporais), além de um ou mais conjuntos de objetos, na elaboração de consultas espaciais, temporais e espaço-temporais típicas. Esta notação é apresentada a seguir, usando uma BNF [BNF] informal.

<consulta> ::= [<nome_consulta>] = <resultado> [| <predicado>] |
<consulta_basica>

<resultado> ::= <identificador> |
<identificador>.<atributo> |
<operacao> |
<resultado> {,<resultado>}*

<operacao> ::= <esp> |
<tem> |
<esp_tem>

<esp> ::= <c_espacial> |
<met> |
<top> |
<ori>

```
<c_espacial> ::=  SP (<identificador>) |
                  ST_SP (<identificador>, <data>)

<met> ::=  AREA (<c_espacial>) |
           LENGTH (<c_espacial>) |
           PERIMETER (<c_espacial>) |
           DISTANCE (<c_espacial>, <c_espacial>)

<top> ::=  DISJOINT (<c_espacial>, <c_espacial>) |
           TOUCH (<c_espacial>, <c_espacial>) |
           OVERLAP (<c_espacial>, <c_espacial>) |
           INSIDE (<c_espacial>, <c_espacial>) |
           CROSS (<c_espacial>, <c_espacial>)

<ori> ::=  NORTH (<c_espacial>, <c_espacial>) |
           EAST (<c_espacial>, <c_espacial>)

<tem>::=  <t_tem> |
          <i_tem> |
          <bool_tem> |
          VSLICE (<identificador>, <valor_tempo>)

<t_tem>::=  <c_temporal> |
            BEGIN (<valor_tempo>) |
            END (<valor_tempo>) |
            INTERVAL (<data>,<data>) |
            T_INTER (<valor_tempo>,<valor_tempo>)

<c_temporal>::=  TV (<identificador>) |
                 TWHEN (<bool_esp_tem>)

<bool_esp_tem>::=  <st_top> |
                   <st_ori>

<st_top>::=  ST_DISJOINT (<identificador>, <identificador>, <valor_tempo>) |
             ST_TOUCH (<identificador>, <identificador>, <valor_tempo>) |
             ST_OVERLAP (<identificador>, <identificador>, <valor_tempo>) |
             ST_INSIDE (<identificador>, <identificador>, <valor_tempo>) |
             ST_CROSS (<identificador>, <identificador>, <valor_tempo>)

<st_ori>::=  ST_NORTH (<identificador>,<identificador>, <valor_tempo>) |
             ST_EAST (<identificador>,<identificador>, <valor_tempo>)
```

```
<valor_tempo>::=  <data> |
                  Beginning |
                  Forever |
                  Now |
                  <t_tem>

<i_tem>::=  YEAR (<data>) |
            MONTH (<data>) |
            DAY (<data>))

<bool_tem>::=  T_BEFORE (<valor_tempo>,<valor_tempo>) |
               T_OVERLAPS (<valor_tempo>,<valor_tempo>) |
               T_EQUAL (<valor_tempo>,<valor_tempo>) |
               T_CONTAINS (<valor_tempo>,<valor_tempo>) |
               T_MEETS(<valor_tempo>,<valor_tempo>)

<esp_tem>::=  <c_espacial_temp> |
              <bool_esp_tem> |
              <st_met>

<c_espacial_temp> ::=  ST_SP (<identificador>, <valor_tempo>)

<st_met> ::=  ST_AREA (<identificador>,<valor_tempo>) |
              ST_LENGTH (<identificador>,<valor_tempo>) |
              ST_PERIMETER (<identificador>,<valor_tempo>) |
              ST_DISTANCE (<identificador>,<identificador>,<valor_tempo>)

<predicado> ::=  <consulta_basica> |
                 <identificador> ∈ <tipo> |
                 ¬ <predicado> |
                 (<predicado>) |
                 <predicado> ∧ <predicado> |
                 <predicado> ∨ <predicado> |
                 ∃ identificador (<predicado>) |
                 ∀ identificador (<predicado>)
```

```
<consulta_basica> ::=  <ori> |
                       <top> |
                       <met> <comparador> <valor_numerico> |
                       <i_tem> <comparador> <valor_numerico> |
                       IS_A_TRUE (<bool_esp_tem>) |
                       IS_S_TRUE (<bool_esp_tem>) |
                       <p_met_temporal> |
                       <bool_tem>

<comparador>::=  < |
                 ≤ |
                 = |
                 > |
                 ≥ |
                 ≠

<p_met_temporal> ::=  <comp> (<st_met>, <valor_numerico>)

<comp>::=  GT |
           GE |
           LT |
           LE |
           EQ |
           NE
```

3.3.2 Consultas Espaciais

As consultas espaciais buscam relacionamentos espaciais entre objetos. Elas podem recuperar atributos espaciais (ou valores calculados a partir deles) desses objetos. Estas consultas são formuladas em termos de operadores espaciais.

Dependendo do que retornam e dos operadores que utilizam, as consultas espaciais básicas podem ser classificadas em:

1. Consultas que retornam componentes espaciais – usam o operador de localização.

   Exemplo: "Onde se situa a fazenda A?".

   SP (A)

2. Consultas que retornam objetos:

   (a) Consultas de orientação – usam os operadores de orientação em seus predicados.
   Exemplo: "Selecione as fazendas ao norte da cidade de Campinas."

f | f ∈ Fazenda ∧ NORTH (SP(f), SP(Campinas))

(b) Consultas métricas - usam os operadores métricos em seus predicados.

Exemplo:

"Selecione as fazendas que estão a menos de 1000 m de uma rodovia."

f | f ∈ Fazenda ∧ ∃r (r ∈ Rodovia ∧ DISTANCE (SP(f), SP(r)) < 1000)

(c) Consultas topológicas - usam os operadores topológicos em seus predicados.

Exemplo: "Selecione os rios que estão contidos na fazenda X."

r | r ∈ Rio ∧ INSIDE (SP(r), SP(X))

3. Consultas que retornam valores (lógicos ou numéricos).

   (a) Consultas que retornam um valor lógico indicando se há ou não um determinado relacionamento espacial entre dois objetos.

   Utilizam operadores métricos, topológicos e de orientação.

   Exemplo: A fazenda X e o rio Y são adjacentes?

   TOUCH (SP(X),SP(Y))

   (b) Consultas que retornam valores numéricos, exigindo o cálculo de uma operação métrica.

   Exemplo: "Qual a distância entre Goiânia e Campinas?"

   DISTANCE (SP(Goiânia), SP(Campinas))

3.3.3 Consultas Temporais

Consultas temporais varrem estados de um objeto ao longo do tempo [Bot95]. Em outras palavras, são consultas onde o tempo é um parâmetro. Estas consultas podem buscar relacionamentos puramente temporais entre objetos, além de buscar informações relacionadas ao tempo de vida de um objeto em particular. Estas consultas contêm ao menos um operador temporal e nenhum operador espacial.

Considera-se aqui que cada objeto é identificado por um OID invariante no tempo, mesmo que sofra mudanças de estado.

As consultas temporais básicas são classificadas, conforme seu resultado, em:

1. Consultas que retornam valores de tempo.

   Utilizam os operadores: TV, BEGIN, END, T_INTER.

   Exemplos:

(a) "Qual o tempo de vida da fazenda A?"
TV (A)

(b) "Quando a fazenda A foi registrada?"
BEGIN (TV (A))

2. Consultas que retornam objetos satisfazendo a um predicado temporal.

   Utilizam os operadores temporais booleanos.

   Exemplos:

   (a) "Selecione as fazendas que foram registradas nos anos 60?"
   f | f $\in$ Fazenda $\wedge$ T_CONTAINS (INTERVAL (01/01/1960,31/12/1969), BEGIN (TV (f)))

   (b) "Quais postes foram fincados ao mesmo tempo que o poste A?"
   p | p $\in$ Poste $\wedge$ T_EQUAL (BEGIN (TV (p)), BEGIN (TV (A)))

   (c) "Quais fazendas existiam em 01/03/1990?"
   f | f $\in$ Fazenda $\wedge$ T_OVERLAPS (TV (f), 01/03/1990)

3. Consultas que retornam estados temporais.

   Utilizam o operador VSLICE.

   Exemplo: "Como a fazenda A era conhecida antes de 1970?"

   VSLICE (A, INTERVAL (Beginning, 31/12/1969))

   (*Beginning* representa o instante inicial da linha de tempo)

   Esta consulta retorna o objeto A reduzido apenas àqueles tempos especificados pelo predicado.

3.3.4 Consultas Espaço-Temporais

Seguindo Botelho [Bot95], as consultas espaço-temporais buscam relacionamentos espaciais entre objetos ao longo do tempo. Elas podem recuperar componentes espaciais ou valores métricos válidos em um determinado intervalo de tempo, ou o tempo de validade de relacionamentos espaço-temporais. Como citado anteriormente, as consultas espaço-temporais envolvem predicados espaciais e temporais, utilizando operadores espaço-temporais.

A seguir serão apresentados os casos básicos de consultas espaço-temporais considerados aqui.

Consultas que Retornam Componentes Espaciais

Retornam as localizações de um objeto que satisfazem a um predicado temporal. Através deste tipo de consulta pode-se derivar o histórico do componente espacial de um objeto.

Exemplo:

"Quais as localizações da fazenda A depois de 1970?"

ST_SP (A, INTERVAL (01/01/1971, Forever))

(*Forever* representa o último instante de tempo na linha de tempo.)

Esta consulta retorna as localizações de A (com seus atributos temporais), cujo tempo associado satisfaz ao predicado temporal.

Consultas que Retornam Valores Métricos

Estas consultas têm como resultado valores calculados por operações métrico-temporais (ST_AREA, ST_LENGTH, ST_PERIMETER, ST_DISTANCE), associados a componentes temporais.

Exemplo:

"Qual a área prevista para a fazenda X no ano de 1998?"

ST_AREA (X, INTERVAL (01/01/1998, 31/12/1998))

Consultas que Retornam Tempos de Relacionamentos

Estas consultas recuperam atributos temporais associados a relacionamentos espaço-temporais topológicos ou de orientação, ou seja, valores de tempo provenientes da avaliação de relacionamentos espaço-temporais. Geralmente têm a seguinte forma básica (a seção 3.3.1 detalha esta notação):

TWHEN (<bool_esp_tem>)

onde <bool_esp_tem> representa uma operação espaço-temporal (topológica ou de orientação), cujos operandos são pelo menos um objeto espaço-temporal e um valor de tempo (limita o domínio temporal da consulta).

Exemplo:

"Em que período, de 1980 a 1995, a fazenda A era adjacente à estrada B?"

TWHEN (ST_TOUCH (A, B, INTERVAL (01/01/1980, 31/12/1995)))

Consultas sobre Existência de Relacionamentos Espaço-temporais

Estas consultas verificam a existência de um relacionamento espacial entre dois objetos em um determinado intervalo de tempo (T).

Geralmente têm uma das seguintes formas:

- IS_A_TRUE (<bool_esp_tem>)
- IS_S_TRUE (<bool_esp_tem>)
- <comp> (<st_met>, <valor_numerico>)

onde <bool_esp_tem> representa uma operação espaço-temporal topológica ou de orientação; <st_met> uma operação métrica-temporal; e <comp> um dos operadores de comparação LT, GE, EQ, etc.

Exemplo:
"A fazenda A alguma vez foi adjacente à estrada B?"
IS_S_TRUE (ST_TOUCH (A, B, INTERVAL (Beginning, Now)))
(*Now* denota o tempo corrente.)

Se T for um instante de tempo, estas consultas também podem ser escritas utilizando operações espaciais (<ori>, <top>, <met>), e o operador ST_SP para obtenção do componente espacial (parâmetro dos operadores espaciais) no tempo T.

Exemplo:
"A fazenda A era adjacente à estrada B em 01/12/1996?"
TOUCH (ST_SP (A,01/12/1996), ST_SP (B,01/12/1996))

Consultas que Retornam Objetos

As consultas anteriores retornam valores ou partes de objetos. Aqui são mostradas consultas que retornam objetos e que têm predicados espaciais e temporais. Estes predicados utilizam operadores espaço-temporais, e geralmente têm uma das formas apresentadas na seção anterior.

Exemplos:

1. "Selecione as fazendas atravessadas por rodovia em 01/12/1996."

   f | f ∈ Fazenda ∧ ∃r (r ∈ Rodovia ∧ IS_A_TRUE (ST_CROSS (f,r,01/02/1996)))

2. "Selecione as fazendas que tiveram área maior que 10 ha entre 01/01/1996 e 01/01/1998."

   f | f ∈ Fazenda ∧ GT (ST_AREA (f, INTERVAL (01/01/1996,01/01/1998)), 10)

3.4 Outras Consultas

A partir das consultas típicas apresentadas, podem ser obtidas consultas mais complexas, tanto pela combinação dos casos apresentados quanto pela construção de predicados mais complexos usando os conectivos AND (∧), OR (∨) e NOT (¬). Exemplos:

1. "Apresente a localização das fazendas adjacentes a um rio, não cruzadas por estrada e com área acima de 100 ha." (Consulta espacial)

   Con1 = f | f ∈ Fazenda ∧ ∃r (r ∈ Rio ∧ TOUCH (SP(f),SP(r))) ∧ ¬ ∃e (e ∈ Estrada ∧ CROSS (SP(f),SP(e))) ∧ AREA (SP(f)) > 100

   Resultado = f, SP (f) | f ∈ Con1

2. "Retorne o tempo de existência das linhas telefônicas instaladas no ano de 1996." (Consulta Temporal)

   l, TV (l) | l ∈ Linha Telefônica ∧ T_CONTAINS (INTERVAL (01/01/1995,31/01/1995), BEGIN (TV (l)))

3. "Selecione as fazendas adjacentes que nunca tiveram área < 100 ha." (Consulta espaço-temporal)

   f_1, f_2 | f_1 ∈ Fazenda ∧ f_2 ∈ Fazenda ∧ IS_A_TRUE (ST_TOUCH (f_1, f_2, Now)) ∧ ¬ LT (ST_AREA (INTERVAL (Beginning, Now)), 100)

Outras consultas espaço-temporais não muito comuns, mas que também podem ser submetidas a bancos de dados geográficos temporais, são as consultas sobre relacionamentos espaciais entre entidades em estados temporais distintos.

Estas consultas têm similaridade com a operação de união espaço-temporal de mapas [Bot95], quando há uma união de temas de estados temporais distintos, por exemplo, união do mapa das fazendas há 10 anos com o mapa hidrográfico atual – vide fig. 3.11.

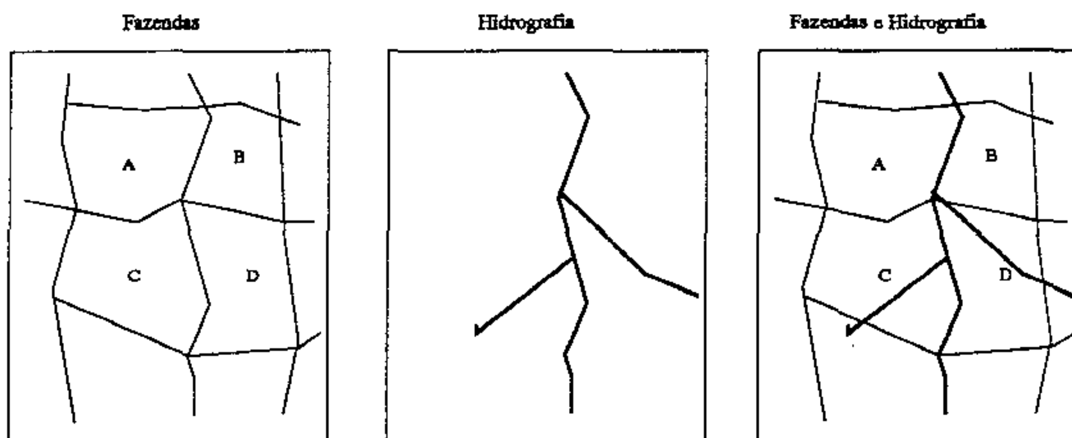


Figura 3.11: União de mapas

Exemplo:
"Que rios cruzariam a fazenda X se ela tivesse hoje o formato que tinha em 01/10/1987?"

r | r $\in$ Rio $\wedge$ CROSS (ST_SP (r, Now), ST_SP (X, 01/10/1987))

A consulta retorna os rios que, no tempo *Now*, têm um relacionamento CROSS com a fazenda X (em 1987).

3.5 Resumo

Este capítulo apresentou um conjunto básico de operadores espaciais, temporais e espaço-temporais e mostrou como usá-los em algumas consultas. O próximo capítulo discute a implementação destes operadores no SGBD O_2.

Os operadores espaciais manipulam atributos espaciais, recebendo como parâmetros objetos geográficos ou componentes espaciais. Podem ser de localização (SP), métricos (AREA, LENGTH, PERIMETER, DISTANCE), topológicos (DISJOINT, TOUCH, OVERLAP, INSIDE, CROSS) ou de orientação (NORTH, EAST).

Os operadores temporais manipulam atributos temporais. Recebem como parâmetros objetos (espaço-temporais ou temporais) ou valores de tempo. Retornam valores de tempo (TV, BEGIN, END, TWHEN, INTERVAL, DAY, MONTH, YEAR, T_INTER), valores booleanos (T_BEFORE, T_OVERLAPS, T_EQUAL, T_CONTAINS, T_MEETS) ou objetos (VSLICE).

Os operadores espaço-temporais manipulam os componentes espaciais de objetos espaço-temporais. Recebem como parâmetros objetos e tempo, retornando valores ou objetos, e tempo. Estes operadores estendem os operadores espaciais pela adição da dimensão temporal.

Capítulo 4

Implementação do Sistema – Estruturas e Algoritmos

Este capítulo apresenta as principais estruturas definidas nesta dissertação para implementação de consultas espaço-temporais. Estas estruturas são baseadas na adaptação do modelo de dados adotado [Bot95] às necessidades constatadas de implementação, tendo em vista os operadores especificados no capítulo anterior.

4.1 Classes Implementadas – Visão Geral

O modelo de Botelho [Bot95] considera a existência de objetos de classes Convencionais ou Geo-Classes. Estes, por sua vez, podem ser temporais ou atemporais. Seguindo ainda [Bot95], esta dissertação trata apenas os objetos geográficos representados pelos geo-objetos, instâncias de classes de uma hierarquia cuja raiz é a classe GeoObject, com subclasses SpatialObject e SpatioTempObject.

A figura 4.1 apresenta as principais classes definidas na dissertação para permitir facilidades espaço-temporais, e algumas das relações (composição e herança) entre elas usando a notação OMT [R$^+$91]. As dimensões espacial e temporal são representadas respectivamente pelas hierarquias com raiz nas classes Location e Time.

Os objetos do tipo Time (Event, Interval e TempElement) são utilizados para representar as marcas de tempo associadas aos objetos temporais. A classe TempElement é composta por uma lista de objetos da classe Interval. A classe Interval é composta por uma tupla de objetos do tipo Event. Objetos do tipo Event representam instantes (pontos) de tempo.

Objetos temporais (instâncias de TempObject) são aqueles que têm um componente temporal associado (ou seja, objeto da classe Time). Os objetos invariantes no tempo (ou objetos atemporais) não têm nenhum objeto componente do tipo Time.

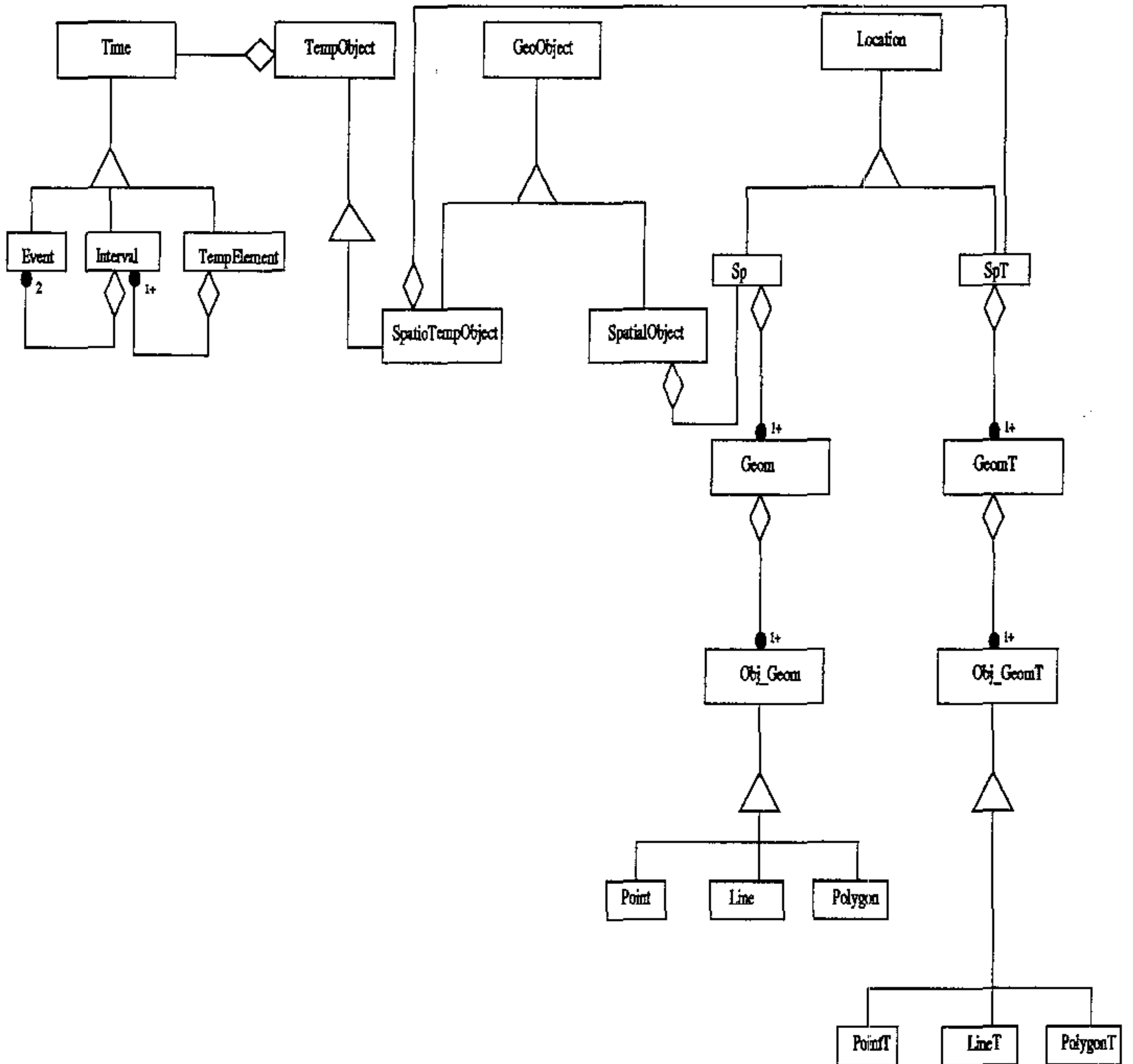


Figura 4.1: Modelo básico de dados

A classe SpatialObject representa os geo-objetos atemporais, ou seja, objetos que têm um componente espacial – objeto do tipo Sp, vide capítulo 3 seção 3.2 – associado e nenhum objeto componente do tipo Time.

O componente espacial (ou localização) de um geo-objeto do tipo SpatialObject é definido por uma lista de objetos da classe Geom. Cada instância de Geom corresponde a uma representação espacial do objeto geográfico, sendo composta por atributos não espaciais para guardar informações específicas da representação (como tipo de projeção, escala, dentre outras) e um atributo espacial para guardar objetos geométricos que definem a geometria da representação. Usando uma notação informal,

Sp: list (Geom)
Geom: ⟨atributos convencionais, list (Obj_Geom)⟩

O componente espacial da classe Geom é uma lista de objetos da classe abstrata Obj_Geom, que é uma generalização das classes geométricas básicas (Point, Line, Polygon). A classe Polygon é composta por uma lista ordenada de coordenadas geográficas, que descrevem a fronteira do polígono. A classe Line é composta por uma tupla de coordenadas, que representam os extremos da linha.

Enquanto a classe SpatialObject descreve geo-objetos atemporais, a classe SpatioTempObject representa geo-objetos temporais (aqui também chamados objetos espaço-temporais), que são os objetos cujos componentes variam com o tempo (tanto componente convencional quanto espacial). Nesta dissertação, a preocupação é com a variação temporal do componente espacial e, desta forma, não serão discutidos aspectos relativos à implementação do tempo para componentes convencionais, já que este é um caso amplamente considerado na literatura.

O componente espacial de um geo-objeto do tipo SpatioTempObject – objeto do tipo SpT, vide seção 3.2 – é definido por uma lista de objetos da classe GeomT. Assim como a classe Geom, GeomT corresponde a uma representação espacial do objeto geográfico. O que a torna diferente de Geom é fato de seus atributos serem temporalizados. Seguindo [Bot95], a temporalização consiste em anexar à classe os atributos necessários para armazenar toda a história do objeto e dos seus componentes. A classe GeomT tem como componente espacial uma lista de objetos da classe abstrata Obj_GeomT, que é uma generalização das classes geométricas temporalizadas (PointT, LineT, PolygonT). Note que o componente espacial de um objeto do tipo SpatioTempObject não pode ser obtido pela simples composição de objetos das classes Sp e Time. Definida desta forma, a localização de um geo-objeto temporal ficaria assim:

⟨Sp, T⟩
Sp: list (Geom)

Além de não corresponder ao modelo proposto por [Bot95], esta alternativa não per-

mite que a evolução da localização de um geo-objeto seja tratada isoladamente do geo-objeto em si, o que contraria o objetivo da dissertação. Assim, a classe SpatioTempObject teria que ser especificada da seguinte maneira (informal):

```
<l: list (<Sp, T>),
 nome_atributo: list (<valor_nome_atributo: tipo,
                        t: Time>),
 lst_geoobjs: list (<lst_objs: list (SpatioTempObject),
                      t: Time>)>
```

Da mesma forma, é necessária a definição da classe GeomT, pois, caso contrário, além de ter-se que acrescentar um atributo de tempo à lista que compõe a classe Sp (list((Geom,T))), que não se aplica à localização dos objetos espaciais, não se poderia ter uma história para cada representação espacial do objeto espaço-temporal.

A seguir são detalhadas as hierarquias relativas às dimensões temporal e espacial (com raízes em Time e Location).

4.2 Classes de Tempo

Esta seção detalha as classes da hierarquia Time, apresentando seus atributos e operações. Além disso, descreve a representação de Tempo utilizada para incorporar o contexto temporal no O_2.

4.2.1 Representação do Tempo

O capítulo 2 apresentou as características através das quais é possível representar-se o Tempo. Aqui é utilizado um modelo *linear* e *discreto* de tempo, apresentando a dimensão temporal *tempo de validade*.

Optou-se pelo modelo linear por facilidade na representação e, conseqüentemente, na implementação. Esta opção implica, portanto, que valores de tempo ocorrerão de forma linear e ordenada, sempre no sentido *passado* $\rightarrow$ *futuro*.

Apesar do conceito de tempo aproximar-se mais de uma representação contínua, o modelo discreto foi escolhido devido aos seguintes aspectos:

1. impossiblidade de medição de eventos instantâneos, pois a medição de eventos é feita com base em *chronons* que apresentam uma duração;

2. a necessidade de também se representar eventos não instantâneos, caso no qual o modelo discreto é mais adequado;

3. por último, qualquer implementação de um modelo temporal de dados sempre implicará na utilização de uma codificação discreta para tempo.

Para a definição da granularidade das marcas de tempo, considerou-se a existência de dois níveis de abstração:

1. nível do usuário (nível de abstração mais alto);

2. nível interno (nível de abstração mais baixo).

Para o nível de abstração mais baixo, definiu-se que as marcas de tempo são representadas por uma granularidade do tipo *simples* e que utiliza a métrica *dia*. A vantagem desta padronização de granularidade é evitar a incompatibilidade de granularidades entre objetos do banco de dados. Os valores de tempo neste nível de abstração serão definidos como sendo do tipo *inteiro*.

Para representar a granularidade no nível mais alto de abstração (o nível do usuário) foi utilizado o conceito de *granularidade virtual*. Granularidade virtual é a propriedade do usuário "enxergar" uma granularidade distinta da granularidade interna [Bra94]. Em outras palavras, com base na propriedade da granularidade virtual, permite-se que o usuário manipule uma granularidade diferente da granularidade interna. Note, no entanto, que há limitações. Por exemplo, se a granularidade interna é *dia*, o usuário não pode recuperar o valor de um atributo em uma granularidade mais fina, por exemplo, *segundo*.

O princípio básico da propriedade da granularidade virtual consiste na transformação do tipo de granularidade de uma marca de tempo. Por exemplo, considere que um eixo temporal apresenta como origem a seguinte marca de tempo: "01/01/1997" (1 de janeiro de 1997). Considere, ainda, a marca de tempo "64", que representa a quantidade de dias após a origem do eixo. Pode-se transformar o tipo da granularidade desta marca de tempo para uma granularidade do tipo composta, formada pelos seguintes elementos: "04/03/1997".

A conversão da granularidade do tipo simples para granularidade do tipo composta é realizada através do método *date* da classe Event, enquanto que a conversão da granularidade do tipo composta para o tipo simples é realizada através da função *event*. Assim, a forma de representação externa do tempo, aqui, é convertida para uma forma interna do tipo inteiro para fácil manipulação e menor espaço de armazenamento. Desta forma, as operações entre tempos são na verdade operações entre valores inteiros, obtidos a partir dos tempos dados. A função *date* converte um dado valor inteiro para uma forma de tempo compreensível para o usuário.

Para representar marcas de tempo foram definidos três formatos: eventos, intervalos e elementos temporais. Estes formatos são representados, respectivamente, por objetos das classes Event, Interval e TempElement.

Para efeito de implementação convencionou-se como origem do eixo temporal, ou valor de *Beginning* a seguinte marca de tempo: "01/01/1997". Este ponto de referência pode ser mudado com algumas modificações na rotinas de conversão de tempo. Tendo em vista a implementação de inteiros no sistema O_2, o fim do eixo temporal, ou valor de *Forever*, é a seguinte marca de tempo: "01/01/2500".

4.2.2 Estrutura Interna das Classes *Time*

Esta seção descreve a especificação das classes da hierarquia Time – Event, Interval, TempElement – usando a linguagem do sistema O_2 [O2T95a]. O tratamento dos operadores temporais YEAR, MONTH, DAY, BEGIN, END, T_BEFORE, T_EQUAL, T_OVERLAPS, T_CONTAINS, T_MEETS e T_INTER, definidos no capítulo 3, é realizado através de métodos destas classes.

O esquema das classes de Tempo fica assim:

```
class Time  /* classe abstrata */
method
    begin: Time, /* métodos básicos, abstratos */
    end: Time,
    t_before (t: Time): boolean,
    t_equal (t: Time): boolean,
    t_overlaps (t: Time): boolean,
    t_contains (t: Time): boolean,
    t_meets (t: Time): boolean,
    t_inter (t: Time): Time
end;

class Event inherit Time type
    integer
method
    date: string,
    year: integer,
    month: integer,
    day: integer
    /* redefine begin, end, t_contains e t_inter */
end;

class Interval inherit Time type
    tuple (t_inicio: Event,
```

```
          t_fim: Event)
method
    /* redefine begin, end, t_contains e t_inter */
end;

class TempElement inherit Time type
    list (Interval)
method
    /* redefine begin, end, t_overlaps, t_contains e t_inter */
end;
```

Observe, por exemplo, que alguns métodos de Time (classe abstrata) são redefinidos em suas subclasses. A definição de métodos é realizada nesta classe para facilitar a especificação da hierarquia.

Os métodos temporais foram implementados tendo em vista as diversas combinações possíveis de objetos de Tempo em uma comparação (Event e Interval, Interval e TempElement, etc.).

Sejam, por exemplo, os objetos *t1* do tipo TempElement e *t2* do tipo Interval, tendo como conteúdo (em notação informal), respectivamente, {⟨1970, 1982⟩, ⟨1985, 1990⟩} e ⟨1980, 1988⟩.

O método *t_overlaps* da classe TempElement aplicado a *t1* retorna true no caso *t_overlaps (t2)*. A implementação deste método verifica qual o tipo do objeto *t* passado como parâmetro, executando um conjunto de instruções diferentes segundo *t* seja Event/Interval ou TempElement. No caso do exemplo, o código executado se baseia em percorrer os intervalos de TempElement (retornados pelo método *intervals*), verificando se para algum *intervalo* o resultado de *intervalo -> t_overlaps (t2)* é igual a true.

A seguir, a implementação de *t_overlaps*:

```
------------------------------------------------------------------
method body t_overlaps (t: Time): boolean in class Time
{
 o2 boolean resultado = false;
 o2 Event e = new Event;
 o2 Interval i = new Interval;
 o2 TempElement telem = new TempElement;
 if ((t->type_of == e->type_of) || (t->type_of == i->type_of)) {
   o2 integer sb,se,tb,te;
   sb = self -> begin -> get_value;
   se = self -> end -> get_value;
   tb = t -> begin -> get_value;
   te = t -> end -> get_value;
   if ((sb <= te) && (tb <= se)) resultado = true;
```

```
 }
 else {
  if (t->type_of == telem->type_of){
    o2 list (Interval) intervalos;
    o2 Interval intervalo;
    intervalos = t -> intervals;
    for (intervalo in intervalos) {
      /* Verifica se self "overlaps" intervalo */
      if (self -> t_overlaps (intervalo)){
        resultado = true;
        break;
      }
    }
  }
 }
 return resultado;
};
------------------------------------------------------------------
method body t_overlaps (t: Time): boolean in class TempElement
{
o2 boolean resultado = false;
o2 Event e = new Event;
o2 Interval i = new Interval, intervalo;
o2 list(Interval) intervalos;
o2 TempElement te = new TempElement;
if ((t->type_of == e->type_of) || (t->type_of == i->type_of)) {
  intervalos = self->intervals;
  for (intervalo in intervalos) {
    /* Verifica se intervalo "overlaps" t */
    if (intervalo -> t_overlaps (t)){
      resultado = true;
      break;
    }
  }
}
else {
  if (t->type_of == te->type_of){
    o2 integer n,m,i,j;
    o2 Interval is, it;
    n = self -> num_intervals; /* Numero de intervalos na lista */
    m = t -> num_intervals;
    for (i=0; i < n; i++){
       for (j=0; j < m; j++){
         is = self -> get_interval(i);
         it = t -> get_interval(j);
         if (is -> t_overlaps (it)){
```

```
                resultado = true;
                break;
            }
        }
    }
    /* Se nao existe nenhum par de intervalos que tenha interseção,
    a funcao retorna falso */
  }
}
return resultado;
};
------------------------------------------------------------------------
```

4.3 Classes de Localização e Geometria

Esta seção apresenta a especificação das classes do tipo Location (Sp e SpT), utilizadas para representar o componente espacial de um objeto geográfico, temporalizado ou não. O modelo definido permite a representação múltipla, restrita ao componente espacial de um geo-objeto.

A independência de um geo-objeto de sua localização (que é um outro objeto do banco de dados) permite que mais de um geo-objeto tenha o mesmo componente espacial. Por exemplo, um geo-objeto Hidrovia pode possuir a mesma localização que um geo-objeto Rio.

Vale a pena ressaltar que a temporalização da classe Geom, resultando na classe GeomT, permite que a evolução temporal da localização de um objeto espaço-temporal seja independente da evolução temporal dos objetos geométricos componentes. Com issso, a quantidade de objetos geométricos de uma representação pode mudar sem que esses objetos tenham sido modificados. Por exemplo, uma fazenda representada por uma lista de polígonos pode ter sua representação alterada pela inclusão de mais um polígono. Além disto, a evolução temporal da localização de um objeto composto é independente da evolução temporal dos geo-objetos componentes. Ou seja, uma alteração na localização de geo-objetos componentes não afeta necessariamente a localização do geo-objeto composto que os contém. De novo, no caso de Fazenda, um dos polígonos pode ser a casa central da Fazenda, que pode mudar de geometria sem afetar os limites da Fazenda propriamente ditos.

O esquema no banco de dados das classes de localização e geometria fica assim:

```
class Location
end;
```

```
class Sp inherit Location type
   list (Geom)
method
   select_geometry: list (Obj_Geom)
end;

class SpT inherit Location type
 list (GeomT)
method
 t_select_geometry(t:Time):list(tuple(lst_objs:list(Obj_GeomT),t:Time))
end;

class Geom type
   tuple (atribs: tipo,
          objs: list (Obj_Geom))
method
   select_lst_objs: list (Obj_Geom)
end;

class GeomT type
 tuple (atribs: [atributos convencionais temporalizados],
          objs: list (tuple (lst_objs: list (Obj_GeomT),
                             t: Time))
method
 t_select_lst_objs(t:Time):list(tuple(lst_objs:list(Obj_GeomT),t:Time))
end;
```

Os métodos *select_geometry* e *select_lst_objs* são ativados pelo método *sp* da classe SpatialObject. O método *select_geometry* seleciona dentre as representações espaciais da localização (Sp), o objeto geometria (Geom) adequado à escala e projeção do mapa atual. O método *select_lst_ojs* retorna a lista de objetos geométricos (Obj_Geom) que definem a geometria da representação.

Os métodos *t_select_geometry* e *t_select_lst_objs* são ativados pelo método *st_sp* da classe SpatioTempObject. O método *t_select_geometry* seleciona, dentre as representações espaciais da localização (SpT), o objeto geometria (GeomT) adequado à escala e projeção do mapa atual, usando a função *right_scale*. O método *t_select_lst_objs* retorna uma lista de objetos geométricos válidos no tempo t especificado, reduzidos aos estados válidos nesse tempo. Para isto são utilizados os métodos *vslice* das classes do tipo Obj_GeomT, definidos na próxima seção.

4.4 Classes Geométricas

Esta seção descreve a especificação das classes geométricas - Point, Line, Polygon. As operações espaciais LENGTH, AREA, DISTANCE, DISJOINT, TOUCH, INSIDE, OVERLAP, CROSS, NORTH e EAST, definidas no capítulo 3 e aplicadas a listas de objetos geométricos, acionam os métodos de mesmo nome destas classes para a obtenção do resultado da operação sobre um objeto – no caso de operações unárias – ou dois objetos – operações binárias.

Uma instância de Point é um ponto no espaço bidimensional $\mathbb{R}^2$. Usa-se a tupla *(a,b)* para denotar um ponto com a coordenada x igual a a e a coordenada y igual a b. Linhas e polígonos são definidos pressupondo-se direcionamento, o que facilita o processamento das operações topológicas.

Uma instância l de Line é um segmento de reta dirigido no plano bidimensional, ou seja, um segmento *[pt_inicio, pt_fim]* conectando dois pontos *pt_inicio*, *pt_fim* $\in \mathbb{R}^2$. *pt_inicio* é denominado o ponto inicial de l , e *pt_fim*, o ponto final de l.

Uma instância p de Polygon é um polígono simples no plano definido pela seqüência de seus vértices ordenada no sentido anti-horário a partir do vértice inferior esquerdo. Por exemplo, definição de um quadrado de dimensões 10 x 10, com vértice inferior esquerdo posicionado na origem (0,0): list (tuple (x:0, y:0), tuple (x:10, y:0), tuple (x:10, y:10), tuple (x:0, y:10)).

O esquema das classes geométricas fica assim:

```
class Obj_Geom  /* classe abstrata */
method
     distance (o: Obj_Geom): real,
     disjoint (o: Obj_Geom): boolean,
     touch (o: Obj_Geom): boolean,
     inside (o: Obj_Geom): boolean,
     overlap (o: Obj_Geom): boolean,
     cross (o: Obj_Geom): boolean,
     north (o: Obj_Geom): boolean,
     east (o: Obj_Geom): boolean
end;

class Point inherit Obj_Geom type
     tuple (x:real, y:real)
method
     /* redefine inside, touch, north e east */
end;
```

```
class Line inherit Obj_Geom type
     tuple (pt_inicio: tuple (x:real, y:real),
            pt_fim: tuple (x:real, y:real))
method
     length: real
     /* redefine overlap, inside, touch, cross, north, east */
end;

class Polygon inherit Obj_Geom type
     list (tuple (x:real, y:real))
method
     perimeter: real,
     area: real
     /* redefine overlap, inside, touch, cross, north, east */
end;
```

4.4.1 Exemplo de Implementação de Operador Espacial

Os métodos topológicos e métricos foram implementados usando algoritmos de O'Rourke [O'R94], e Preparata e Shamos [PS85]. A seguir é apresentado um exemplo que ilustra parte da implementação.

Seja a figura 4.2, que mostra duas figuras poligonais (por exemplo, a representação da fazenda A e da fazenda B). O método *touch* da classe Polygon aplicado a A retorna *true* no caso *touch (B)*, e o método *disjoint* de Polygon aplicado a A retorna *false* no caso *disjoint (B)*.

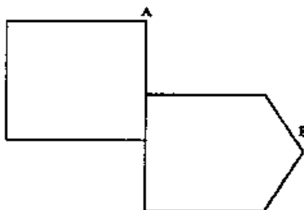


Figura 4.2: Representação de dois geo-objetos do tipo Fazenda

A implementação do método *disjoint* utiliza o método *intersect*, definido para cada subclasse de Obj_Geom, retornando *true* se o resultado de *intersect* for igual a *false*. A implementação do método *intersect* para a classe Polygon, que verifica a interseção entre dois polígonos, utiliza a função *PolygonIntersectionTest*, cuja idéia básica é a seguinte:

"Dados dois polígonos simples P e Q, se algum lado de P intersecta um lado de Q, ou P contém Q, ou Q contém P, então P e Q se intersectam." [PS85].

A implementação do método *touch* verifica qual o tipo do objeto geométrico retornado pela função *ConvexIntersect* – *polinter*, que calcula a interseção entre dois polígonos – A e B, e, se o objeto resultante não for nem A nem B, ou não formar um polígono, o resultado da operação é verdadeiro.

O método *touch* é implementado da seguinte forma:

```
-------------------------------------------------------------------
method body touch (o: Polygon): boolean in class Polygon
{
  o2 Polygon polinter = new Polygon;
  if ( ConvexIntersect (self, o, polinter) ) {
     /* verify if intersection is diferent from A (self) and B (o) */
     if ( ! ( polinter->deep_equal (self) || polinter -> deep_equal (o) ) ) {
       /* verify if polinter is not a polygon */
       if (polinter -> num_vertices < 3)
         return true; /* polinter is not a polygon */
       else {
          o2 coord a, b, c;
          a = polinter->vertice(0);
          b = polinter->vertice(1);
          c = polinter->vertice(2);
          if ( Collinear (a,b,c) )
             return true;
          else return false;
       }
     }
     else return false;
  }
  else return false;
};
-------------------------------------------------------------------
```

A implementação de *disjoint* é a seguinte:

```
-------------------------------------------------------------------
method body disjoint (o: Obj_Geom): boolean in class Obj_Geom
{
  return ! self->intersect (o);
};
-------------------------------------------------------------------
```

Os demais métodos espaciais não serão descritos aqui para evitar fatigar o leitor. O código completo está no apêndice.

4.4.2 Classes Geométricas Temporalizadas

Esta seção descreve a especificação das classes geométricas temporalizadas - PointT, LineT, PolygonT. Os objetos destas classes armazenam a história de um objeto geométrico ao longo do tempo.

Uma das estruturas possíveis para a representação dos objetos geométricos temporalizados (objetos cuja raiz é a classe Obj_GeomT), baseada no modelo de [Bot95], é a seguinte:

```
class Obj_GeomT  /* classe abstrata */

end;

class PointT inherit Obj_GeomT type
 tuple (coord: tuple (x:real, y:real),
        t: Time)
end;

class LineT inherit Obj_GeomT type
  list (tuple (pts: list (PointT),
               t: Time))
end;

class PolygonT inherit Obj_GeomT type
  list (tuple (lns: list (LineT),
               t: Time))
end;
```

Usando as estruturas acima, um polígono (PolygonT), e uma linha (LineT) podem ter seu conjunto de componentes alterados sem que qualquer ponto (PointT) tenha sido modificado. Apesar destas estruturas serem vantajosas quanto ao espaço de armazenamento, diminuindo a redundância de dados armazenados, elas dificultam o processamento de consultas, principalmente das que exigem a recuperação da localização de um objeto espaço-temporal em um determinado intervalo de tempo. Isto se deve ao elevado nível de indireção das estruturas, e ao fato de que cada objeto varia de forma independente no tempo, assim como cada um de seus componentes.

Assim, optando por melhor desempenho na execução de consultas, com prejuízo no espaço de armazenamento, foram adotadas as seguintes estruturas para representar a evolução temporal de objetos geométricos:

```
class Obj_GeomT  /* classe abstrata */
method
   vslice (t:Time): Obj_GeomT
end;

class PointT inherit Obj_GeomT type
   tuple (coord:tuple (x:real, y:real),
          t: Time)
method
   /* redefine vslice */
end;

class LineT inherit Obj_GeomT type
   list (tuple (pts: tuple (pt_inicio: tuple (x:real, y:real),
                            pt_fim: tuple (x:real, y:real)),
                t: Time))
method
   /* redefine vslice */
end;

class PolygonT inherit Obj_GeomT type
   list (tuple (lst_coord: list (tuple (x:real, y:real)),
                t: Time))
method
   /* redefine vslice */
end;
```

Com esta representação, cada vez que muda uma das coordenadas de um polígono, nova versão do polígono é criada, copiando as demais coordenadas.

O método *vslice* recupera uma "fatia temporal" de um objeto válida no valor de tempo passado como parâmetro (evento, intervalo, ou elemento temporal). Esta "fatia temporal" corresponde ao estado ou conjunto de estados do objeto válidos no valor de tempo desejado.

Seja, por exemplo, o polígono p definido informalmente da seguinte maneira:

⟨50,200⟩,⟨100,200⟩,⟨100,800⟩,⟨50,800⟩,[1950,1975]
⟨50,200⟩,⟨100,200⟩,⟨50,800⟩, [1976,1994]
⟨55,200⟩,⟨100,200⟩,⟨50,800⟩, [1995,1997]

O método *vslice* da classe Polygon aplicado a p retorna o seguinte objeto p' no caso de *p->vslice(t)*, t = [1970,1976]:

⟨50,200⟩,⟨100,200⟩,⟨100,800⟩,⟨50,800⟩,[1970,1975]
⟨50,200⟩,⟨100,200⟩,⟨50,800⟩, 1976

4.5 Classes Geográficas e Temporais

Conforme citado anteriormente, os objetos no banco de dados podem ser temporais ou atemporais considerando a dimensão temporal; e espaciais ou convencionais na dimensão espacial. Na figura 4.1, a classe TempObject representa os objetos temporais e a classe abstrata GeoObject representa os geo-objetos, que podem ser temporais (SpatioTempObject) ou atemporais (SpatialObject).

Os geo-objetos podem ser elementares, compostos ou fracos. O geo-objeto elementar corresponde a uma entidade que não é composta por outros geo-objetos. O seu estado (atributos) é constituído apenas por atributos não espaciais e por um componente espacial. O geo-objeto composto é construído a partir da composição de outros geo-objetos. O geo-objeto fraco contém apenas o componente espacial (Location) e existe somente enquanto faz parte de um geo-objeto composto.

A seguir a especificação das classes:

```
class TempObject type
    tuple (t: Time,
           nome_atributo: list (tuple (valor_nome_atributo: tipo,
                                       t: Time)))
method
    vslice (t:Time): TempObject,
    tv: Time
end;

class GeoObject /* Classe Abstrata */
method
    sp: list (Obj_Geom),
    st_sp (t: Time): list (tuple (cs:list(Obj_Geom),t:TempElement))
end;

class SpatialObject inherit GeoObject type
    tuple (l: Sp,
           nome_atributo: tipo,
           lst_objs: list (SpatialObject))
```

```
method
    sp: list (Obj_Geom)
end;

class SpatioTempObject inherit GeoObject, TempObject type
  tuple (l: SpT,
         nome_atributo: list (tuple (valor_nome_atributo: tipo,
                                     t: Time)),
         lst_geoobjs: list (tuple (lst_objs: list (SpatioTempObject),
                                   t: Time)))
method
  vslice (t: Time):SpatioTempObject,
  sp: list (Obj_Geom),
  st_sp (t: Time):list (tuple (cs:list(Obj_Geom),t:TempElement)),
  loc (t: Event):list(Obj_Geom),
  st_length (t:Time):list (tuple (res: real, t: Time)),
  st_perimeter (t: Time):list (tuple (res: real, t: Time)),
  st_area (t: Time):list (tuple (res: real, t: Time)),
  st_distance(obj:GeoObject,t:Time):list(tuple(res:real,t:Time)),
  st_disjoint(obj:GeoObject,t:Time):list(tuple(res:boolean,t:Time)),
  st_overlap(obj:GeoObject,t:Time):list(tuple(res:boolean,t:Time)),
  st_inside(obj:GeoObject,t:Time):list(tuple(res:boolean,t: Time)),
  st_touch(obj:GeoObject,t:Time):list(tuple(res:boolean,t:Time)),
  st_cross(obj:GeoObject,t:Time):list(tuple(res:boolean,t:Time)),
  st_north(obj:GeoObject, t:Time):list(tuple(res:boolean,t:Time)),
  st_east(obj:GeoObject, t:Time):list(tuple(res:boolean,t:Time)),
end;
```

Os operadores VSLICE e TV, definidos no capítulo 3 são especificados, respectivamente, como os métodos *vslice* and *tv* de TempObject, podendo ser redefinidos nas suas subclasses.

O operador SP do capítulo 3 é tratado como um método da classe GeoObject, sendo redefinido nas suas subclasses. A implementação de *sp* para a classe SpatialObject retorna a lista de objetos geométricos da localização (Sp) de um geo-objeto, enquanto que a implementação do mesmo operador para a classe SpatioTempObject retorna a lista de objetos geométricos da localização de um geo-objeto, válida no tempo presente (retornado pela função Now()).

O operador ST_SP é mapeado para os métodos *st_sp* e *loc* de SpatioTempObject. *loc* retorna a localização de uma instância de SpatioTempObject num instante de tempo t

(Event), enquanto que *st_sp* retorna o histórico da área ocupada por uma instância de SpatioTempObject em um tempo *t* (Event, Interval, TempElement). Sendo o método *st_sp* mais geral, ele é utilizado na implementação de *loc*. Vale a pena ressaltar que o método *loc* retorna apenas uma localização, enquanto que o método *st_sp* retorna localização e tempo, sendo este mais apropriado para consultas com intervalos de tempo.

Note que, apesar de serem equivalentes os resultados de *sp* e *loc (Now)* quando aplicados a um objeto espaço-temporal, optou-se por definir o método *sp* para a classe SpatioTempObject de modo a facilitar a elaboração de consultas espaciais. Desta maneira, pode-se verificar relacionamentos espaciais ou localização atual de objetos sem a preocupação de aplicar o método correto (*sp* ou *loc (Now)*) para objetos espaciais ou espaço-temporais.

A implementação do método *st_sp* é composta de três partes principais. A primeira determina se o geo-objeto temporal é válido no tempo especificado, utilizando o método temporal *t_overlaps*. Em caso afirmativo, a segunda parte envia uma mensagem para o componente localização (SpT) desse geo-objeto acionando o seu método *t_select_geometry*, descrito na seção 4.3. A lista de objetos geométricos temporalizados retornada pelo método *t_select_geometry* tem a seguinte estrutura:

list (tuple (lst_objs: list (Obj_GeomT), t: Time))

Cada elemento da lista tem dois componentes: um atributo temporal (t) e um atributo espacial (lst_objs), que é a representação espacial da localização do geo-objeto válida nos intervalos de tempo contidos em t. Note que o componente lst_objs é representado por uma lista de objetos, indicando que num determinado tempo um geo-objeto pode estar espacialmente representado por uma combinação de objetos geométricos.

A terceira parte da implementação de *st_sp* organiza o resultado de *t_select_geometry* em termos de objetos geométricos (Obj_Geom) e tempo. O objetivo desta manipulação é determinar exatamente quais objetos geométricos definem a geometria de um geo-objeto em um tempo específico, ou seja, quais as versões dos Obj_GeomT retornados por *t_select_geometry* realmente definem a geometria do geo-objeto em um valor de tempo t. Esta informação não poderia ser obtida diretamente do resultado de *t_select_geometry*, já que a história de cada objeto geométrico é independente da sua associação com um geo-objeto específico. O resultado de *st_sp* tem, então a seguinte forma:

list (tuple (cs: list(Obj_Geom), t:TempElement))

sendo que cada elemento da lista corresponde ao conjunto de objetos geométricos que representam a localização de um geo-objeto no tempo t, contido no tempo que é parâmetro do operador.

Os operadores espaço-temporais do capítulo 3 - ST_LENGTH, ST_DISJOINT, etc. - são definidos como métodos *st_length, st_disjoint, etc.* de SpatioTempObject. Eles utilizam os operadores espaciais (definidos como funções que recebem como parâmetros listas de objetos geométricos), e os operadores temporais (métodos da classe Time).

Os métodos espaço-temporais que envolvem apenas um objeto espaço-temporal – *st_length, st_perimeter* e *st_area*, invocados com um parâmetro de tempo (t), são implementados em três passos principais:

1. O método *st_sp* cria uma lista de tuplas $\langle G,T\rangle$, onde G é um conjunto de objetos geométricos (Obj_Geom) que compõem a geometria do objeto, e T é o tempo de validade associado, que deve estar contido no parâmetro de tempo especificado (t).

2. Um valor (comprimento/perímetro/área) é calculado para o componente G de cada tupla.

3. O resultado final é uma lista de tuplas $\langle valor, T\rangle$, onde a cada *valor* é associado o tempo de validade correspondente T.

A implementação dos métodos que envolvem dois geo-objetos – *st_disjoint, st_distance, etc.*, invocados com um parâmetro de tempo t, compreende basicamente as seguintes etapas:

1. Para cada geo-objeto é criada uma lista de tuplas $\langle G,T\rangle$, do mesmo modo que para os métodos que envolvem apenas um geo-objeto. Assume-se que os geo-objetos do tipo SpatialObject – que também podem ser parâmetros deste tipo de método espaço-temporal – são válidos em qualquer tempo.

2. Os tempos de validade de cada elemento destas listas são processados para a obtenção dos intervalos de interseção temporal, usando o método *t_inter* da hierarquia Time.

3. Para cada intervalo de interseção temporal, os objetos geométricos G válidos nesse intervalo são processados pela função espacial correspondente (*disjoint, distance, etc.*), sendo calculado um resultado.

4. Uma lista com os resultados finais é retornada.

Seja, por exemplo, a seguinte variação de estados da localização de dois geo-objetos no intervalo de tempo [t0,t11], colocadas nas listas cs1 e cs2:

geom1 = cs1[1][t1,t2], cs1[2][t3,t5], cs1[3][t6,t10]
geom2 = cs2[1][t0,t1], cs2[2][t2,t4], cs2[3][t5,t7], cs2[4][t9,t11]

As fatias (ou estados) temporais sobre as quais se deve aplicar a operação espacial são:

t1: cs1[1] e cs2[1]; t2: cs1[1] e cs2[2]; [t3,t4]: cs1[2] e cs2[2]; t5: cs1[2] e cs2[3]; [t6,t7]: cs1[3] e cs2[3]; [t9,t10]: cs1[3] e cs2[4].

Considere de novo o exemplo do método espacial *disjoint* cujo código foi descrito anteriormente. No caso da verificação do mesmo relacionamento topológico, no tempo, entre as duas fazendas (A e B), o operador é utilizado agora repetidas vezes, uma para cada intervalo de interseção temporal da localização dos objetos.

Seja, por exemplo, a seqüência temporal apresentada na figura 4.3.

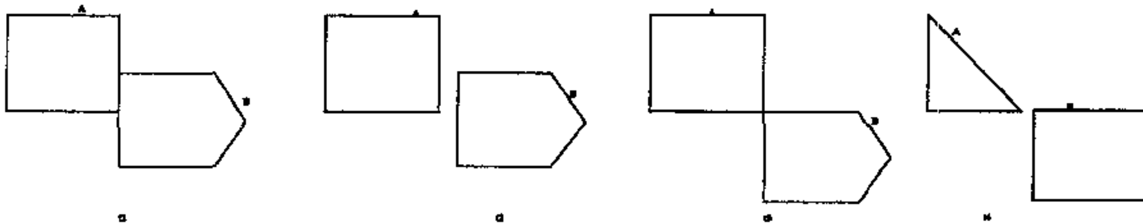


Figura 4.3: Estados da geometria das fazendas A e B entre t1 e t4

st_disjoint (B,[t1,t3]) aplicado a A terá os valores ⟨false, t1⟩, ⟨true, t2⟩, ⟨false, t3⟩.

A seguir, a implementação do método *st_disjoint*. Note que ele é baseado na aplicação repetida da função *disjoint* para cada intervalo *intervalo* de interseção temporal da localização dos objetos (válidas no parâmetro de tempo *t*). A função *disjoint*, por sua vez, é baseada na aplicação do método *disjoint* (descrito na seção 4.4.1) para cada par de objetos geométricos que compõe a geometria de A e B no tempo *intervalo*.

```
--------------------------------------------------------------------------------
method body st_disjoint (obj: GeoObject, t: Time) in class SpatioTempObject
 {   /* Operador ST_DISJOINT */
  o2 list (tuple (res: boolean, t: Time)) resultado;
  o2 list (tuple (cs: list (Obj_Geom), t: TempElement)) geom1;
  o2 Time intersecao;
  o2 list (Interval) intervalos;
  o2 Interval intervalo;
  o2 boolean disj;
  o2 list (Obj_Geom) g1,g2;
  o2 TempElement tgeom1;

  resultado = list ();
  geom1 = self -> st_sp (t);
  if ( geom1 != list() ) {
    tgeom1 = tvg (geom1);  /* Obtem tv de geom1 */
    if (obj->type_of == self->type_of) {
      o2 list (tuple (cs: list (Obj_Geom), t: TempElement)) geom2;
      geom2 = obj -> st_sp (t);
      if ( geom2 != list() ) {
        o2 TempElement tgeom2;
        tgeom2 = tvg (geom2);
```

```
            intersecao = tgeom1 -> t_inter (tgeom2);
            intervalos = intersecao -> intervals;
            for (intervalo in intervalos) {
               g1 = get_temporal_version (geom1, intervalo);
               /* Verifica qual das listas intersecta o intervalo */
               g2 = get_temporal_version (geom2, intervalo);
               disj = disjoint (g1, g2);
               resultado += list (tuple (res: disj, t: intervalo));
            }
         }
      }
      else {
         o2 tuple (cs: list (Obj_Geom), t: TempElement) geom_el;
         g2 = obj -> sp;
         for (geom_el in geom1) {
            disj = disjoint (geom_el.cs, g2);
            resultado += list (tuple (res:disj, t:geom_el.t));
         }
      }
   }
   return resultado;
};
------------------------------------------------------------------
```

Esta solução é baseada em inicialmente obter o histórico espacial de cada objeto (no domínio de tempo t), fazer interseções temporais e, finalmente, interseções espaciais.

Existem outras alternativas para a implementação dos métodos espaço-temporais descritos acima (métodos que envolvem dois geo-objetos A e B). Uma delas é a seguinte:

1. Para cada objeto, criar uma lista de tuplas $\langle G,T \rangle$ correspondendo à história da localização do objeto.

2. Determinar os pares de tuplas $\langle\langle G_A,T_A\rangle, \langle G_B,T_B\rangle\rangle$, da história da localização de A e de B que têm interseção temporal.

3. Para cada par obtido no passo anterior, retornar o resultado da função espacial associado à interseção temporal entre os componentes do par.

4. Determinar os pares obtidos no passo anterior cujo tempo associado tem interseção com t (satisfazem ao predicado temporal), substituindo estes tempos pela sua interseção com t.

5. Retornar o resultado.

Esta alternativa é similar à solução adotada, com a exceção de que só resolve o predicado temporal no fim do processamento.

Outra opção é a seguinte:

1. Para cada objeto, criar uma lista de tuplas $\langle G, T \rangle$ correspondendo à história da localização do objeto.

2. Calcular o resultado da função espacial correspondente para cada par de tuplas da história da localização de A e de B, produzindo tuplas $\langle valor, T_A, T_B \rangle$, onde *valor* é o resultado da função, e T_A, T_B são, respectivamente, o tempo T de uma tupla da história de A e B.

3. Determinar os elementos da lista obtida no passo anterior com interseção temporal, retornando pares $\langle valor, T \rangle$, sendo T o tempo de interseção entre T_A e T_B.

4. Determinar os pares obtidos no passo anterior cujo tempo associado tem interseção com t (satisfazem ao predicado temporal), substituindo estes tempos pela sua interseção com t.

5. Retornar o resultado.

Esta solução é baseada em inicialmente obter o histórico espacial de cada objeto, resolver a operação espacial, e, finalmente, fazer interseções temporais.

A escolha da melhor alternativa de implementação depende de uma estimativa do custo de cada uma, discussão que está fora do escopo deste texto.

4.6 Funções

Alguns dos operadores apresentados no capítulo 3 foram especificados como funções: os de orientação NORTH e SOUTH; métricos AREA, LENGTH, PERIMETER e DISTANCE; topológicos DISJOINT, INSIDE, TOUCH, CROSS e OVERLAP; temporais TWHEN e INTERVAL, booleanos IS_A_TRUE e IS_S_TRUE; e de comparação GT, GE, EQ, NE, LT e LE.

As funções que implementam os operadores espaciais invocam os métodos das classes geométricas (Obj_Geom), de mesmo nome. As funções espaciais – *area, length, disjoint, etc.* – recebem como parâmetro uma – operadores unários – ou duas – operadores binários – listas de objetos geométricos (list (Obj_Geom)), e retornam um valor numérico – operadores métricos – ou um valor lógico – operadores topológicos e direcionais.

Um exemplo de função espacial importante é

inside (A: list (Obj_Geom), B: list (Obj_Geom))

que verifica para todo objeto geométrico *objA* da lista *A* se existe um objeto *objB* em *B* tal que

(objA -> inside (objB)) = true

ativando assim o método *inside* de cada objeto *objA* de *A*.

A seguir a implementação da função *inside*:

```
-----------------------------------------------------------------------
function body inside (A: list (Obj_Geom), B: list (Obj_Geom)): boolean
{
 o2 boolean resultado, found;
 o2 Obj_Geom objA, objB;
 if ((A == list ()) || (B == list ()) )
    return false;
 else {
   resultado = true;
   for ( objA in A ) {
     found = false;
       for ( objB in B ) {
         if ( objA -> inside (objB) ) {
           found = true; /* Existe um objeto em B que contem objA */
           break;
         }
       }
       if (!found) {
          resultado = false;
          break;
       }
   }
   return resultado;
 }
};
-----------------------------------------------------------------------
```

Outro exemplo é *disjoint (A: list (Obj_Geom), B: list (Obj_Geom))*, que verifica se para todo objeto geométrico *objA* da lista *A* e para todo objeto *objB* da lista *B*,

(objA -> disjoint (objB)) = true

A seguir a implementação da função *disjoint*:

```
-----------------------------------------------------------------------
function body disjoint (A: list (Obj_Geom), B: list (Obj_Geom)): boolean
{
 o2 boolean resultado;
 o2 Obj_Geom objA, objB;
 if ((A == list ()) || (B == list ()) ) return true;
```

```
 resultado = true;
 for ( objA in A ){
   for ( objB in B ){
     if ( ! ( objA -> disjoint (objB) ) ) {
        resultado = false;
        break;
     }
   }
 }
 return resultado;
};
```

A função *twhen* recebe como parâmetro uma lista de valores lógicos associados a tempo (list (tuple(res: boolean, t: Time))), e retorna um objeto do tipo TempElement que representa os intervalos de tempo em que o relacionamento topológico é verdadeiro.

A função *interval* recebe como parâmetro duas datas (s e e) e retorna um objeto do tipo Interval com tempo inicial s e tempo final e; enquanto que a função *event* recebe uma data como parâmetro, e retorna um objeto Event correspondente. Como mencionado anteriormente, a função *event* é usada principalmente para converter a granularidade de tempo virtual "enxergada" pelo usuário em uma granularidade interna, manipulada pelas operações temporais.

As funções *is_a_true* e *is_s_true* são utilizadas para converter o resultado de uma operação espaço-temporal booleana (que retorna um conjunto de valores lógicos e tempo associado - list (tuple(res: boolean, t: Time))), em um único valor lógico. A primeira função retorna *true* se todos os valores lógicos retornados pela operação forem verdadeiros, enquanto que a última retorna *true* se algum válor lógico for verdadeiro.

As funções *gt, ge, lt, le, eq, ne* recebem como parâmetro uma lista de tuplas $\langle$valor, tempo$\rangle$ - list (tuple(res: real, t: Time)) - e um valor v, retornando *true* se todos os valores da lista são, respectivamente, maiores, maiores ou iguais, menores, menores ou iguais, iguais ou diferentes de v.

O código para as funções descritas e outras funções auxiliares se encontra no apêndice.

4.7 Análise das Classes e sua Implementação

O modelo de classes especificado é baseado no trabalho de [Bot95]. No entanto, aquele trabalho é teórico e não se ateve a problemas que poderiam ocorrer em uma implementação. Desta forma, tendo em vista a implementação no O_2 e as limitações decorrentes, as classes descritas contêm algumas modificações em relação ao modelo original.

A principal modificação ocorreu nas marcas de tempo (ou atributos temporais) associadas a objetos ou atributos. A estrutura proposta por [Bot95] para estes atributos temporais é uma lista de intervalos de tempo, descrita da seguinte forma:

atributo temporal: list (tuple (tv_inicio:integer, tv_fim:integer))

Este modelo foi aqui estendido com a inclusão das classes do tipo Time (Event, Interval e TempElement), sendo a forma de definição dos atributos temporais alterada para a seguinte:

atributo temporal: Time

Esta estrutura permite que marcas de tempo sejam de qualquer tipo (evento, intervalo ou elemento temporal), e não apenas do tipo elemento temporal, como foi proposto originalmente por [Bot95]. Além disso, a definição de objetos Time permite modelar abstrações de tempo mais adequadamente. Operações temporais podem ser definidas como métodos da classe Time e redefinidas nas suas subclasses, sendo implementadas de acordo com as propriedades de cada uma destas classes.

A segunda modificação foi a possibilidade de existência simultânea no banco de dados de geo-objetos apenas espaciais e geo-objetos espaço-temporais. Esta modificação acarretou mudanças na nomenclatura adotada por [Bot95] para as classes geográficas, geométricas, de localização e geometria.

A terceira modificação ocorreu nas classes do tipo Obj_GeomT, conforme discutido na seção 4.4.2.

A quarta modificação ocorreu nas classes do tipo Obj_Geom. O modelo original previa que um polígono fosse composto por uma lista de linhas, e que uma linha fosse composta por uma lista de pontos. No entanto, como mencionado previamente, a especificação das classes Obj_GeomT utilizando o conceito de [Bot95] dificulta o processamento de consultas. Assim, para manter a consistência entre a definição de objetos geométricos (Obj_Geom) e objetos geométricos temporalizados (Obj_GeomT), um polígono é definido por uma lista ordenada de coordenadas geográficas, e uma linha, por uma tupla de coordenadas. Esta definição é comumente encontrada na literatura. Para simplificar os algoritmos, não foram considerados os multi-polígonos (MultiPolygon[1]) definidos em [Bot95].

As estruturas para armazenar a evolução temporal de dados espaciais foram implementadas a partir dos construtores de tipos *list* e *tuple* existentes no O_2 [O2T95a]. Estes construtores foram utilizados pela sua eficiência na redução de espaço de armazenamento e tempo de consulta, sendo otimizados internamente pelo O_2.

Para utilizar os operadores básicos implementados, as classes espaço-temporais de uma

[1]A classe MultiPolygon é composta por uma lista de polígonos para representar regiões desconexas, e uma lista de polígonos para representar buracos desconexos.

aplicação SIG devem ser definidas como subclasses de SpatioTempObject.

O apêndice contém o código da implementação sobre o sistema O_2 das classes e da maioria dos métodos e funções descritos neste capítulo, além de outros operadores auxiliares. Foram implementadas os métodos da hierarquia Time e Location; de Geom e GeomT; das classes geométricas (Obj_Geom), com a exceção de *north, east, cross*, e alguns casos de *touch* e *inside*; de Obj_GeomT; de TempObject, exceto *vslice*; de SpatialObject; de SpatioTempObject, com a exceção de *vslice, st_north, st_east, st_cross* e alguns casos de *st_touch* e st_inside; e todas as funções, exceto *north, east, cross.* Deve ser mencionado aqui que o código da função *right_scale* invoca funções que devem ser implementadas por aplicações.

4.8 Outras Alternativas de Modelagem

Existem várias formas de implementar estruturas para navegação espaço-temporal. A estrutura escolhida é um compromisso entre espaço de armazenamento e facilidade no processamento de consultas, conforme citado anteriormente.

Uma primeira alternativa é definir a classe SpatioTempObject da seguinte forma:

```
class SpatioTempObject inherit GeoObject, TempObject type
   list (tuple (l: list (tuple (x:real, y:real)),
                nome_atributo: tipo,
                lst_geoobjs: list (SpatioTempObject),
                t: Time))
end;
```

Esta estrutura otimiza o processamento de consultas que exigem a recuperação do estado do objeto em um tempo específico, mas é extremamente redundante, já que cada mudança em algum dos componentes ocasiona a criação de uma nova versão para o objeto como um todo.

Uma segunda alternativa é definir a classe SpatioTempObject de modo que seu componente espacial seja uma lista de coordenadas (com um (ponto), dois (linha), ou mais elementos (polígono)):

```
class SpatioTempObject inherit GeoObject, TempObject type
   tuple (l: list (tuple (coord: tuple (x:real, y:real),
                          t: Time)),
          nome_atributo: list (tuple (valor_nome_atributo: tipo,
                                      t: Time)),
          lst_geoobjs: list (tuple (lst_objs: list (SpatioTempObject),
```

```
                                   t: Time)))
end;
```

Apesar de facilitar a implementação dos operadores espaço-temporais, a utilização desta segunda estrutura ocasiona grande redundância de dados, além de permitir apenas uma representação para o geo-objeto.

Uma outra opção seria definir a classe GeomT de forma que os objetos que compõem a sua geometria sejam os objetos geométricos básicos (Polygon, Line ou Point):

```
class GeomT type
  tuple (atribs: [atributos convencionais temporalizados],
         objs: list (tuple (lst_objs: list (Obj_Geom),
                            t: Time))
end;
```

A utilização desta estrutura também facilita o processamento de consultas espaço-temporais, mas ainda causa certa redundância de dados, já que a mudança de uma coordenada na geometria de um geo-objeto provoca a definição de um novo objeto geométrico, além do acréscimo de um elemento à lista de versões da localização (GeomT).

4.9 Resumo

Este capítulo apresentou as classes implementadas para navegação espaço-temporal: classes de tempo (Event, Interval e TempElement); localização (Sp e SpT); de geometria (Geom e GeomT); geométricas (Point, Line e Polygon); geométricas temporalizadas (PointT, LineT e PolygonT); temporais (TempObject e SpatioTempObject); e geográficas (SpatialObject e SpatioTempObject).

Os operadores do capítulo 3 foram implementados em sua maioria como métodos destas classes, usando algoritmos de geometria computacional para cálculo de relacionamentos topológicos.

Capítulo 5

Consultas Espaço-temporais

As classes e métodos do capítulo 4 formam a base do sistema espaço-temporal que permite análise integrada dos dados nas dimensões de espaço e tempo. Aplicações que desejam realizar consultas podem fazê-lo agregando suas classes às classes propostas.

Este capítulo descreve um exemplo da utilização do banco de dados implementado para uma aplicação geográfica. A especificação dos problemas e consultas da aplicação foi elaborada a partir de exemplos fornecidos por pesquisadores da FEAGRI - UNICAMP.

5.1 Visão Geral do Problema

O objetivo da aplicação é analisar a ocupação do território brasileiro por áreas de cultivo, aqui também chamadas divisões agrícolas. Isto pode ser usado para, por exemplo, simplificar o processo de reforma agrária, detectando fazendas improdutivas, ou monitorar o desmatamento. Através de consultas aos dados coletados pode-se determinar fazendas com infra-estrutura propícia ao cultivo (próximas ou cortadas por estradas, abastecidas por rede elétrica, etc), locais onde construir estradas, e assim por diante.

Classifica-se as consultas típicas da aplicação em:

a) Consultas sobre evolução do plantio.

   Utilizadas para o estudo da evolução do cultivo e produtividade ao longo do tempo, visando, por exemplo, planejamento de rotação de culturas, detecção de técnicas impróprias de manejo ou indícios de destruição ambiental.

   Exemplos:

   a.1) Quais as divisões agrícolas ocupadas por plantações de cana antes de 01/07/1997?

   a.2) Qual a produção (toneladas/produto) da fazenda A entre 1990 e 1995?

a.3) O que foi cultivado na fazenda A entre 1990 e 1995?

a.4) Selecione as plantações de cana adjacentes em 01/02/1997.

b) Consultas sobre infra-estrutura de propriedades.

Utilizadas para verificar a situação (por exemplo, localização, proximidade de vias de acesso, disponibilidade de instalações) para planejamento de escoamento de colheita, melhoria de infra-estrutura, etc.

Exemplos:

b.1) Qual a localização da fazenda A?

b.2) Selecione as fazendas distantes até 10km da estrada X.

b.3) Selecione as fazendas que contêm postes de luz.

b.4) Quais fazendas existiam em 07/09/1997?

c) Consultas sobre evolução espacial.

Utilizadas para acompanhamento da evolução espacial de uma ou várias propriedades, visando detectar, por exemplo, tendências de ocupação de uma região. Se, por exemplo, as propriedades apresentam retalhamento ao longo do tempo, isto indica um aumento no número de pequenos agricultores, o que pode demandar estabelecimento de cooperativas agrícolas. Outro exemplo ocorre quando há redução constante da área plantada das propriedades agrícolas, o que pode indicar êxodo rural.

Exemplos:

c.1) Qual a evolução da área ocupada pela fazenda A entre 1992 e 1996?

c.2) Qual a área ocupada por fazendas entre 01/01/97 e 01/01/98?

c.3) Quando a fazenda A esteve incluída no retângulo delimitado pelas coordenadas X1,Y1,X2,Y2?

d) Consultas sobre evolução global (cultivo, infra-estrutura, espaço)

Visam planejamento integrado ao longo do tempo, a partir do estudo de tendências e análise de alternativas.

Exemplos:

d.1) Qual o estado[1] da fazenda A entre 1992 e 1997?

[1] Estado é entendido, aqui, pela configuração (valor dos atributos) do objeto em um tempo específico.

d.2) Que estradas intersectariam a fazenda A se ela tivesse hoje o formato que tinha em 01/03/1997?

d.3) Apresente a localização atual das fazendas que cultivam cana e que tiveram área maior que 1000 ha entre 1990 e 1992.

Usando a classificação apresentada no capítulo 3, as consultas acima também podem ser divididas em: espaciais (b.1, b.2, b.3), temporais (b.4, d.1, a.1, a.2, a.3), e espaço-temporais (c.1, c.2, c.3, a.4, d.2, d.3).

5.2 Definições de Classes

O primeiro passo no desenvolvimento de uma aplicação é a definição das classes dependentes da aplicação. No caso desta dissertação, estas classes são acrescentadas às classes e operações já existentes no sistema espaço-temporal. O exemplo usado aqui é uma aplicação SIG simples envolvendo fazendas (compostas por divisões agrícolas), estradas e postes. Como estes objetos são entidades do mundo real e podem variar no tempo, eles são descritos por classes derivadas de SpatioTempObject. A especificação das classes correspondentes é mostrada na figura 5.1, usando a sintaxe do O_2.

Deve-se lembrar aqui que a localização de um geo-objeto pode ter múltiplas representações (objetos Geometria), cada qual tendo informações sobre escala ou projeção. A representação adequada é recuperada pelo operador *select_geometry* (*t_select_geometry*) no processamento de uma consulta .

Para efeito dos exemplos apresentados no decorrer do capítulo, pressupõe-se que a representação utilizada para uma fazenda é uma lista de polígonos; para uma divisão agrícola, um polígono; para uma estrada, uma lista de linhas; e para um poste, um ponto.

A figura 5.1 mostra que uma fazenda é um objeto formado por divisões (lst_divisoes) que por sua vez têm um histórico. Observe que a evolução temporal da localização da fazenda é independente da evolução temporal da localização das divisões componentes. Ou seja, os limites de uma divisão agrícola podem mudar sem alterar os limites da fazenda a qual ela pertence, assim como os limites da fazenda podem mudar sem alterar os limites das suas divisões. Além disso, a área coberta pelas plantações pode não cobrir a área total da fazenda. Assim, a evolução da cultura se avalia a partir de cada divisão e a evolução do espaço ocupado, a partir do componente espacial da fazenda.

5.3 Exemplo de Modelagem de Geo-objetos

O exemplo (tabelas 5.1, 5.2, 5.3, 5.4, 5.5, 5.6 e 5.7) mostra um geo-objeto fazenda *faz1* e suas divisões agrícolas, modelados usando as classes especificadas na figura 5.1 e no

```
class Fazenda inherit SpatioTempObject type
  tuple (nome: string,
         historico_divisoes: list (tuple (lst_divisoes: list (Divisao_Agricola),
                                          t: Time)))
end;
```

```
class Divisao_Agricola inherit SpatioTempObject type
  tuple (nome: string,
         historico_cultura: list (tuple (cultura: string,
                                         t: Time)),
         historico_producao: list (tuple (producao: real,
                                          t: Time)))
end;
```

```
class Estrada inherit SpatioTempObject type
  tuple (nome: string,
         historico_tipo_pavimentacao: list (tuple (tipo_pavimentacao: string,
                                                   t: Time)))
end;
```

```
class Poste inherit SpatioTempObject type
  tuple (numero: integer,
         tipo: string)
end;
```

```
name Fazendas: set (Fazenda);
name Divisoes: set (Divisao_Agricola);
name Estradas: set (Estrada);
name Postes: set (Poste);
```

Figura 5.1: Esquema da Aplicação

capítulo 4, onde i_j são intervalos de tempo. A figura 5.2 ilustra a evolução espacial da fazenda.

Os geo-objetos são representados por instâncias das classes Fazenda e Divisao_Agricola. As instâncias de Divisao_Agricola *div1* e *div2* representam as divisões da fazenda. A instância de Fazenda *faz1* representa a fazenda, sendo que a sua lista de geo-objetos contém *div1* e *div2* no tempo *i1*, e *div1* no tempo *i2*. As localizações de *faz1, div1* e *div2* referenciam apenas uma representação espacial cada uma: G1, G2 e G3, respectivamente. G1, G2 e G3 têm como o atributo não espacial (Escala), respectivamente, EscalaG1, EscalaG2, EscalaG3. A representação de *faz1* – G1 – tem como componentes espaciais os polígonos Pol1 e Pol2, no intervalo i1; e o polígono Pol1, no intervalo i2. As representações de *div1* e *div2* têm como componente espacial Pol1 (no intervalo i3), e Pol2 (no intervalo i1), respectivamente. Note que cada polígono tem sua própria história, que independe da associação deste a uma representação específica (instância de GeomT). e_j são instantes de tempo, e T_Forever, a instância de Event que representa o último instante da linha de tempo.

As tabelas a seguir mostram o conteúdo dos objetos desse exemplo.

Fazenda Id	**Nome**	**Histórico Divisões**	**Localização**	**Tempo de Validade**
faz1	A	{div1,div2}, i1 {div1}, i2	loc_faz1	i3

Tabela 5.1: Fazenda

Div Id	**Nome**	**Cultura**	**Produção**	**Localização**	**Tempo de Validade**
div1	Divisao-1	trigo, i4 cana, i5	100, i4 200, i5	loc_div1	i3
div2	Divisao-2	cafe, i1	300, i1	loc_div2	i1

Tabela 5.2: Divisao_Agricola

SpT Id	Lista de Geometria
loc_faz1	G1
loc_div1	G2
loc_div2	G3

Tabela 5.3: SpT

GeomT Id	Lista de Objetos Geométricos	Escala
G1	{Pol1, Pol2}, i1 {Pol1}, i2	EscalaG1
G2	{Pol1}, i3	EscalaG2
G3	{Pol2}, i1	EscalaG3

Tabela 5.4: GeomT

PolygonT Id	Lista de Coordenadas
Pol1	{(50,200),(100,200),(100,800),(50,800)}, i1 {(50,200),(150,200),(150,800),(50,800)}, i2
Pol2	{(100,200),(300,200),(300,800),(100,800)}, i3

Tabela 5.5: PolygonT

Interval Id	Início	Fim
i1	e1	e2
i2	e3	T_Forever
i3	e1	T_Forever
i4	e1	e4
i5	e5	e6

Tabela 5.6: Interval

Event Id	Valor
e1	1
e2	400
e3	401
e4	430
e5	500
e6	800

Tabela 5.7: Event

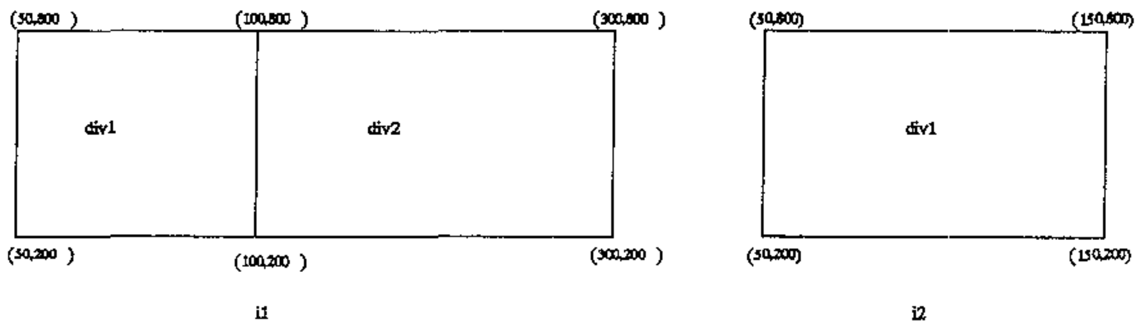


Figura 5.2: Histórico da localização da fazenda A

5.4 Exemplos de Consultas

Esta seção usa a OQL [O2T95b], linguagem de consulta do O_2 para mostrar como algumas das consultas enumeradas na seção 5.1 são resolvidas usando os operadores implementados. É importante notar que o sistema implementado não gera código OQL automaticamente dada uma consulta do usuário (em linguagem natural) – ou seja, não provê uma interface amigável ao usuário. A geração automática de código OQL dada uma linguagem de consultas espaço-temporal é, no entanto, um tópico de pesquisa interessante, fazendo parte das extensões a este trabalho.

5.4.1 Consultas Espaciais

As consultas espaciais utilizam basicamente o método *sp* de GeoObject (redefinido nas suas subclasses). Na realidade, o próprio *sp* pode representar uma consulta básica sobre um único objeto do banco de dados.

Um exemplo de consulta que retorna a localização de um geo-objeto, e usa *sp* é:

- "Qual a localização da fazenda A?" (Consulta b.1)

→ SP (A)

→ Consulta OQL gerada:

```
select f->sp
from f in Fazendas
where f.nome = "A"
```

As consultas espaciais que buscam relacionamentos espaciais entre objetos ou retornam valores numéricos utilizam as funções *length, area, distance, disjoint, overlap, inside, touch, cross, north, east.*

Um exemplo de consulta que busca o relacionamento espacial *inside* entre objetos é:

- "Selecione as fazendas que contêm postes de luz." (Consulta b.3)

  → f | f ∈ Fazenda ∧ p ∈ Poste ∧ p.tipo = "luz" ∧ INSIDE (SP(p),SP(f))

  → Consulta OQL gerada:

```
select distinct f
from f in Fazendas,
     p in Postes
where p.tipo = "luz" and inside(p->sp, f->sp)
```

Esta consulta exige a execução da função *inside* para objetos Fazenda do conjunto Fazendas e objetos Poste do conjunto Postes cujo tipo é "luz".

5.4.2 Consultas Temporais

As consultas temporais são compostas basicamente por predicados contendo os métodos temporais *t_before, t_meets, t_contains, t_equal e t_overlaps* da hierarquia Time. Estes métodos recebem como parâmetros objetos Time, que podem representar um valor de tempo específico (fornecido pela consulta), o tempo de validade de um objeto temporal ou uma marca de tempo associada a um atributo temporal.

Exemplos de consultas com operadores temporais booleanos são:

1. "Quais fazendas existiam em 07/09/1997?" (Consulta b.4)

   → f | f ∈ Fazenda ∧ T_OVERLAPS (TV (f), 07/09/1997)

   → Consulta OQL gerada:

```
select f
from f in Fazendas
where f->tv -> t_overlaps (event ("07/09/1997"))
```

Esta consulta exige a execução dos métodos *t_overlaps* e *tv* para cada fazenda (f) em Fazendas, e a execução da função *event.* O objeto do tipo Event, retornado pela função *event*, e o tempo de validade (f->tv) de cada fazenda são os objetos de tempo manipulados pelo método *t_overlaps.*

2. "Quais as divisões agrícolas ocupadas por plantações de cana antes de 01/07/1997?" (Consulta a.1)

   $\rightarrow$ d | d $\in$ Divisao_Agricola $\wedge$ e $\in$ d.historico_cultura $\wedge$ d.cultura = "cana" $\wedge$ T_BEFORE (BEGIN (e.t), 01/07/1997)

   $\rightarrow$ Consulta OQL gerada:

```
select distinct d
from d in Divisoes,
     e in d.historico_cultura
where e.cultura = "cana" and
      e.t->begin->t_before(event("1/7/1997"))
```

   Esta consulta seleciona os objetos do tipo Divisao_Agricola armazenados no banco de dados - no conjunto Divisoes - tais que, em seu histórico de cultura (historico_cultura), exista ao menos uma cultura (e) de cana cujo início da marca de tempo associada (e.t) tenha o relacionamento temporal *t_before* com o instante de tempo "1/7/1997".

Outras consultas temporais podem retornar valores de tempo, usando os métodos *begin, end, day, month, year* da hierarquia Time, ou retornar objetos reduzidos a seus estados válidos em um tempo específico, usando o operador *vslice*, definido para as classes temporalizadas (TempObject, SpT, GeomT, etc.).

Exemplo:

- "Qual o estado da fazenda A entre 1992 e 1997?" (Consulta d.1)

  $\rightarrow$ VSLICE (A, INTERVAL (01/01/1992, 31/12/1997))

  $\rightarrow$ Consulta OQL gerada:

```
select f-> vslice (interval ("01/01/1992","31/12/1997"))
from f in Fazendas
where f.nome = "A"
```

Esta consulta retorna os estados da Fazenda A válidos no intervalo de tempo especificado. Ela utiliza o método *vslice*, cuja execução exige conhecimento do esquema das classes envolvidas na operação. O protótipo desenvolvido contém o operador *vslice* para as classes pré-definidas SpT, GeomT e Obj_GeomT (PointT, LineT, PolygonT), já que estas fazem parte da infra-estrutura espaço-temporal. Por outro lado, a implementação de *vslice* sobre classes definidas pelo "usuário" não pode ser feita a priori, pois depende do esquema destas classes.

Desta forma, esta consulta não pode ser executada diretamente, exigindo a implementação de um código especial. Uma solução seria executar um conjunto de consultas do O_2 sobre metadados (por exemplo, a consulta *attributes*, que retorna os atributos de uma classe), que retornariam o esquema da classe, que por sua vez seria passado como parâmetro para *vslice*. Supondo que o esquema já esteja disponível, esta consulta seria processada da seguinte forma:

- Obtenção do identificador do objeto Fazenda cujo nome é A (f);
- Obtenção do objeto Time (neste caso, um Interval *t*, retornado pela função *interval*) correspondente ao intervalo de tempo [01/01/1992, 31/12/1997].
- Execução da operação f -> vslice (t).

5.4.3 Consultas Espaço-Temporais

As consultas espaço-temporais utilizam basicamente os métodos *st_sp, st_disjoint, st_overlap, etc.* de SpatioTempObject, cuja implementação combina funções espaciais e métodos temporais. Estes métodos recebem como parâmetro 0 ou 1 geo-objeto e um objeto do tipo Time, e retornam valores (lógicos, númericos) ou objetos geométricos associados a tempos de validade.

Alguns exemplos de consultas espaço-temporais são:

1. "Qual a área ocupada por fazendas entre 01/01/97 e 01/01/98?" (Consulta c.2)

   → ST_AREA (f, INTERVAL (01/01/97,01/01/98)) | f ∈ Fazenda

   → Consulta OQL gerada:

```
select tuple (fazenda: f,
              area:f->st_area(interval("01/01/97", "01/01/98")))
from f in Fazendas
```

   Esta consulta executa o método espaço-temporal *st_area* para cada instância *f* de Fazendas no banco de dados, retornando as áreas válidas no intervalo de tempo

especificado associadas a uma marca de tempo (que indica seu tempo de validade dentro do intervalo fornecido).

2. "Quando a fazenda A esteve incluída no retângulo delimitado pelas coordenadas (1,1),(15,40)?" (Consulta c.3)

   → TWHEN (ST_INSIDE (X, TO_OBJ ((1,1), (15,1), (15,40), (1,40))), INTERVAL (Beginning, Now))

   → Consulta OQL gerada:

   A consulta é executada em três etapas:

   (a) Seleção do objeto fazenda cujo nome é "A".

   ```
   define q1 as element (select f
                           from f in Fazendas
                           where f.nome = "A")
   ```

   (b) Obtenção do geo-objeto fraco correspondente ao retângulo fornecido.

   ```
   define q2 as to_obj (list (tuple(x:1.0,y:1.0),
          tuple(x:15.0,y:1.0),tuple(x:15.0,y:40.0),
          tuple(x:1.0,y:40.0)))
   ```

   (c) Obtenção do tempo em que a fazenda esteve incluída no retângulo.

   ```
   twhen (q1->st_inside(q2, interval(Beginning,Now())))
   ```

   Esta consulta utiliza as funções temporais *twhen, interval*, o método espaço-temporal *st_inside* da classe SpatioTempObject, e a função *to_obj*, que retorna um geo-objeto (SpatialObject) com uma geometria constante delimitada pelas coordenadas indicadas. *twhen* retorna o objeto Time correspondente aos intervalos de tempo, no domínio de tempo da consulta ([Beginning, Now]), em que o objeto *f* esteve contido no retângulo. *Beginning* é um tempo constante, enquanto que *Now* é uma função que retorna o tempo atual.

3. "Selecione as plantações de cana adjacentes em 01/02/1997." (Consulta a.4)

   → q1 = d | d ∈ Divisoes ∧ T_OVERLAPS (TV(d), 01/02/1997) ∧ ∃c (c ∈ d.historico_cultura ∧ T_OVERLAPS (c.t,01/02/1997) ∧ c.cultura = "cana")

   q2 = d1, d2 | d1 ∈ q1 ∧ d2 ∈ q1 ∧ IS_A_TRUE (ST_TOUCH (d1,d2,01/02/1997))

   → Consulta OQL gerada:

   Uma das formas de se processar esta consulta é a seguinte:

(a) seleção das divisões agrícolas que existiam em 01/02/1997.

```
define q1 as select d
             from d in Divisoes
             where d->tv->t_overlaps(event("01/02/1997"))
```

(b) seleção das divisões em q1 que cultivavam cana em 01/02/1997.

```
define q2 as select distinct d
             from d in q1,
                  c in d.historico_cultura
             where d.historico_cultura != list() and
                   c.t->t_overlaps (event("01/02/1997"))
                   and c.cultura = "cana"
```

(c) seleção das divisões resultantes da consulta q2 que eram adjacentes em 01/02/1997.

```
select tuple (Divisao1: d1, Divisao2: d2)
from d1 in q2,
     d2 in q2
where d1 != d2 and
      is_a_true(d1->st_touch(d2,event("01/02/1997")))
```

Esta consulta utiliza o método temporal *t_overlaps*, a função *event*, o método espaço-temporal *st_touch*, e a função *is_a_true*. *is_a_true* retorna verdadeiro se o resultado de *st_touch* for verdadeiro para todos os intervalos de interseção temporal entre d1 e d2 no domínio de tempo especificado pela consulta (no caso, 01/02/1997).

Outra opção seria processar a consulta da etapa (c) da seguinte forma:

```
select tuple (Divisao1: d1, Divisao2: d2)
from d1 in q2,
     d2 in q2
where d1 != d2 and
     touch (d1->loc(event("01/02/1997")),d2->loc(event("01/02/1997")))
```

Esta solução utiliza o método espaço-temporal *loc* para recuperar a localização de cada divisão no tempo especificado (01/02/1997), e a função espacial *touch* para verificar o relacionamento *touch* entre as localizações retornadas por *loc*. Esta opção de processamento é preferível sempre que o tempo especificado na consulta for um

instante; não podendo ser usada com parâmetros de tempo do tipo intervalo, já que a função *touch* verifica um relacionamento espacial apenas entre dois conjuntos de objetos geométricos.

4. "Que estradas intersectariam a fazenda A se ela tivesse hoje o formato que tinha em 01/03/1997?" (Consulta d.2)

   $\rightarrow$ e | e $\in$ Estrada $\wedge$ $\neg$ DISJOINT (SP(e), ST_SP (A,01/03/97)).

   $\rightarrow$ Consulta OQL gerada:

   Uma das formas de se processar esta consulta é a seguinte:

   (a) obtenção do objeto Fazenda cujo nome é "A".

   ```
   define a as element (select f
                        from f in Fazendas
                        where f.nome = "A")
   ```

   (b) obtenção das estradas que satisfazem ao predicado.

   ```
   select e
   from e in Estradas
   where not disjoint (e->sp, a->loc(event("01/03/97")))
   ```

   Esta consulta executa a função *disjoint* para cada estrada do banco de dados. Ela verifica se há o relacionamento topológico *intersect* (ou seja, *not disjoint*) entre a localização de cada estrada no tempo presente (e->sp) e a localização da fazenda A no tempo 01/03/97 (a->loc(event("01/03/97"))). O método *loc* é usado aqui em vez do método *st_sp*, porque retorna apenas uma localização, tendo como parâmetro um instante de tempo. O método *st_sp* retorna localização e tempo, sendo mais apropriado para consultas com intervalos de tempo.

5. "Apresente a localização atual das fazendas que cultivam cana e que tiveram área maior que 1000 ha entre 1990 e 1992." (Consulta d.3)

   $\rightarrow$ q1 = f | f $\in$ Fazenda $\wedge$ e $\in$ f.historico_divisoes $\wedge$ d $\in$ e.lst_divisoes $\wedge$ c $\in$ d.historico_cultura $\wedge$ T_OVERLAPS (e.t, EVENT(Now)) $\wedge$ T_OVERLAPS (c.t, EVENT(Now)) $\wedge$ c.cultura = "cana"

   q2 = f, SP (f) | f $\in$ q1 $\wedge$ GE (ST_AREA (f, INTERVAL (01/01/1990, 31/12/1992)), 1000)

   $\rightarrow$ Consulta OQL gerada:

   Uma das formas de se processar esta consulta é a seguinte:

(a) Seleção das fazendas que cultivam cana

```
define q1 as select f
             from f in Fazendas,
                  e in f.historico_divisoes,
                  d in e.lst_divisoes,
                  c in d.historico_cultura
             where e.t->t_overlaps (event(now())) and
                   c.t->t_overlaps (event(now())) and
                   c.cultura = "cana"
```

(b) Seleção das fazendas que tinham área maior que 1000 ha entre 1990 e 1992, e apresentação da localização atual destas.

```
select tuple (Fazenda: f, Localizacao: f->sp)
from f in q1
where gt (f->st_area(interval("01/01/1990","31/12/1992")),1000)
```

Esta consulta combina métodos espaciais (*sp*), temporais (*t_overlaps, event, interval*) e espaço-temporais (*st_area*). Além disto é utilizado o método *ge*, que verifica, para cada área retornada pelo método *st_area*, se esta é maior que 1000.

A implementação das consultas enumeradas na seção 5.1 e não descritas neste capítulo se encontra no apêndice.

5.5 Resumo

Este capítulo mostra como consultas de uma aplicação SIG podem ser implementadas utilizando as classes e funções definidas no sistema espaço-temporal. Além disso, mostra como consultas espaciais, temporais e espaço-temporais podem ser executadas sobre o O_2, usando os operadores apresentados nos capítulos 3 e 4 e os comandos da linguagem OQL (linguagem de consulta orientada a objetos do sistema O_2) .

Capítulo 6

Conclusões e Extensões

Esta dissertação discutiu a implementação da base de um banco de dados espaço-temporal, baseado no modelo de Botelho [Bot95], sobre o SGBD OO O_2.

As principais contribuições da dissertação são:

- Especificação de um conjunto de operadores básicos espaciais, temporais e espaço-temporais que podem ser usados para a formulação de uma grande gama de consultas (Capítulo 3). A literatura correlata trata de operadores espaciais ou operadores temporais, mas não de sua combinação.

- Especificação de um conjunto básico de classes do banco de dados para dar apoio ao gerenciamento espaço-temporal, e algoritmos para implementar os operadores nestas classes (Capítulo 4). Esta especificação foi acompanhada de uma descrição das modificações feitas no trabalho de [Bot95] visando viabilizar a implementação.

- Implementação de classes e algoritmos, validando a proposta com implementação de consultas típicas de aplicações geográficas (Capítulo 5).

Até hoje, há apenas dois trabalhos publicados sobre implementação efetiva de modelos espaço-temporais [PW94, SB97], ficando a maioria dos trabalhos em propostas de modelo. Uma análise cuidadosa de algumas destas propostas mostra que sua implementação não é factível, exigindo modificações dos modelos (como, aliás, foi necessário nesta dissertação).

Os trabalhos de implementação [PW94, SB97] não contemplam processamento integrado de espaço e tempo e nem a possibilidade de múltiplas representações espaciais e temporais, sendo assim menos gerais que o protótipo implementado na dissertação.

As extensões a este trabalho podem ser tanto teóricas quanto práticas. As seguintes modificações poderiam ser consideradas:

- Adoção de índices espaciais e temporais para melhorar a performance de consultas sobre dados espaço-temporais.

- Aperfeiçoamento e definição de novos algoritmos para processamento de consultas espaço-temporais.

- Estudo sobre a complexidade dos algoritmos propostos.

- Implementação dos operadores especificados e não implementados (a seção 4.7 lista os operadores implementados), além de operadores para atualização de dados espaço-temporais.

- Implementação de um Gerenciador de Consultas que construa consultas a serem processadas pelo O_2query (processador de consultas do O_2) com base em especificações fornecidas pelo usuário.

- Melhoria da interface do sistema, permitindo, por exemplo, a apresentação gráfica de geometrias.

- Trabalho com dados tridimensionais.

- Uso de dados no formato raster (GeoCampo).

- Tratamento de indeterminância temporal.

- Permissão para que o usuário a escolha da granularidade temporal mais adequada às suas aplicações. A granularidade do sistema consiste na duração de um *chronon* – dia. No entanto, dependendo da aplicação considerada, às vezes é necessário considerar diferentes granularidades (dias, meses, anos) para permitir uma melhor representação da realidade.

Bibliografia

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

[ATSS93] K.K. Al-Taha, R. T. Snodgrass, and M. D. Soo. Bibliography on Spatiotemporal Databases. *ACM SIGMOD Record*, 22(1):59–67, March 1993.

[BBFS96] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. A Temporal Access Control Mechanism for Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):67–80, 1996.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-oriented Database System*. Data Management Systems. Morgan Kaufmann Publishers, 1992.

[BJS95] M. Bohlen, C. S. Jensen, and R. T. Snodgrass. Evaluating and Enhancing the Completeness of TSQL2. Technical Report TR 95-05, Department of Computer Science, University of Arizona, Tucson, AZ85721, June 1995.

[BJS97] M. Bohlen, C. S. Jensen, and B. Skjellaug. Spatio-Temporal Database Support for Legacy Applications. Technical Report TR-20, TimeCenter, July 1997.

[BM94] A. R. Brayner and C. B. Medeiros. Incorporação do Tempo em um SGBD Orientado a Objetos. In *Anais do IX Simpósio Brasileiro de Banco de Dados*, pages 16–29, São Carlos, Brazil, September 1994.

[BNF] Bnf. http://www.cs.man.ac.uk/ pjj/bnf/bnf.html.

[Boh95] M. H. Bohlen. Temporal Database System Implementations. *Sigmod Record*, 24(4):53–60, 1995.

[Bot95] M. A. Botelho. Incorporação de Facilidades Espaço-Temporais em Bancos de Dados Orientados a Objetos. Master's thesis, Universidade Estadual de Campinas, 1995.

[Bra94] A. R. Brayner. Implementação de um Sistema Temporal em um Banco de Dados Orientado a Objetos. Master's thesis, Universidade Estadual de Campinas, 1994.

[BVH96] L. Becker, A. Voigtmann, and K. H. Hinrichs. Temporal Support for Geo-Data in Object-Oriented Databases. In *Proc. of DEXA'96*, pages 79–93. Springer, 1996.

[CCH+96] G. Câmara, M. A. Casanova, A. S. Hemerly, G. C. Magalhães, and C. M. Bauzer Medeiros. *Anatomia de Sistemas de Informação Geográfica.* X Escola de Computação, Instituto de Computação, UNICAMP, 1996.

[CdFvO93] E. Clementini, P. di Felice, and P. van Oosterom. A Small Set of Formal Topological Relationships Suitable for End-User Interaction. In *Proceedings of the 3rd International Symposium on Large Spatial Databases*, pages 277–295, 1993.

[Cil96] M. A. Cilia. Banco de Dados Ativos como Suporte a Restrições Topológicas em Sistemas de Informação Geográfica. Master's thesis, Unicamp, March 1996.

[CS94] A. E. Cavalcanti and A. C. Salgado. Um Estudo para Tratar a Dimensão de Tempo em Sistemas de Banco de Dados. *IX Simpósio Brasileiro de Banco de Dados*, pages 357–381, 1994.

[CT95a] C. Claramunt and M. Thériault. Managing Time in GIS - An Event-Oriented Approach. In *Proc. of the International Workshop on Temporal Databases*, Zurich, Switzerland, September 1995. Springer.

[CT95b] J. Clifford and A. Tuzhilin, editors. *Recent Advances in Temporal Databases: Proceedings of the International Workshop on Temporal Databases*, Berlin, 1995. Springer-Verlag.

[EO94] N. Edelweiss and J. P. M. Oliveira. *Modelagem de Aspectos Temporais de Sistemas de Informação.* IX Escola de Computação, Recife (PE), 1994.

[FM91] A. Frank and D. Mark. Language Issues for GIS. In *Geographical Information Systems*, volume I, pages 147–161. John Wiley and Sons, 1991.

[GS89] H. Gunadhi and A. Segev. Query Optimization in Temporal Databases. In *Proc. of the Fifth International Conference on Statistical and Scientific Database Management Systems*, pages 131–147, 1989.

[JCE+94] C. S. Jensen, J. Clifford, R. Elmasri, S. Gadia, P. Hayes, and S. Jajodia. Consensus Glossary of Temporal Database Concepts. *SIGMOD Record*, 23(1):52–64, 1994.

[Jen93] C. S. Jensen. A Consensus Test Suite of Temporal Database Queries. Technical Report TR 93-2034, Department of Mathematics and Computer, Institute for Electronic Systems, Aalborg University, Denmark, 1993.

[Jen97] C. S. Jensen. Tutorial on Temporal Databases. SBBD'97 - Fortaleza, CE, October 1997.

[JS96] C. S. Jensen and R. T. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4):311–352, 1996.

[JS97] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. Technical Report TR-17, TimeCenter, June 1997.

[JSS94] C. J. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying temporal models via a conceptual model. *Information Systems*, 19(7):513–547, 1994.

[Kli93] N. Kline. An Update of the Temporal Database Bibliography. *ACM Sigmod Record*, 22(1):59–67, March 1993.

[Lan89] G. Langran. A Review of Temporal Database Research and its Use in GIS Applications. *International Journal of Geographic Information Systems*, 3(3):215–232, 1989.

[Lan93a] G. Langran. Issues of Implementing a Spatiotemporal System. *International Journal of Geographic Information Systems*, 7(4):305–314, 1993.

[Lan93b] G. Langran. *Time in Geographic Information Systems.* Taylor & Francis Ltda, 1993.

[Lim76] E. L. Lima. *Elementos de Topologia Geral.* Livros Técnicos e Científicos, 1976.

[MB96] C. B. Medeiros and M. Botelho. Tratamento do Tempo em SIG. In *Anais do GIS Brasil'96*, May 1996.

[McK86] E. McKenzie. Bibliography: Temporal databases. *SIGMOD Record*, 15(4):40–52, December 1986.

[Mon95] L. D. Montgomery. Temporal Geographic Information Systems Technology and Requirements: Where We Are Today. Master's thesis, The Ohio Sate University, 1995.

[MP94] C. B. Medeiros and F. Pires. Databases for GIS. *Sigmod Record*, 23(1):107–115, 1994.

[NTE92] R. G. Newell, D. Theriault, and M. Easterfield. Temporal GIS - Modeling the Evolution of Spatial Data in Time. *Computers and Geosciences*, 18(4):427–433, 1992.

[O2T92] O2Technology. *The O2 User's Manual*, 1992. Version 3.3.

[O2T95a] O2Technology. *O2C Reference Manual*, March 1995.

[O2T95b] O2Technology. *Object Query Language - OQL Manual*, March 1995. Version 4.5.

[Oli93] L. C. Medina Oliveira. Incorporação da Dimensão Temporal em Bancos de Dados Orientados a Objetos. Master's thesis, Universidade Estadual de Campinas, 1993.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[OS95] G. Ozsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction.* Springer-Verlag, 1985.

[PS93] D. Papadias and T. Sellis. The Semantics of Relations in 2D Space Using Representative Points: Spatial Indexes. In A. U. Frank and I. Campari, editors, *Proc. of the European Conference on Spatial Information Theory*, pages 234–247, 1993.

[PTS94] D. Papadias, Y. Theodoridis, and T. Sellis. The Retrieval of Direction Relations Using R-trees. In *Proc. of the 5th International Conference on Database and Expert Systems Applications*, pages 173–182, 1994.

[PW94] D. Peuquet and E. A. Wentz. An Approach for Time-based Spatial Analysis of Spatio-Temporal Data. In *Advances in GIS Research*, pages 489–504, 1994.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[RP92] J. F. Roddick and J. D. Patrick. Temporal Semantics in Information Systems - A Survey. *Information Systems*, 17(3):249–267, 1992.

[RS94] P. J. Resende and J. Stolfi. *Fundamentos de Geometria Computacional.* IX Escola de Computação, Recife (PE), July 1994.

[RSS+96] K. Ramamritham, R. Sivasankaran, J. A. Stankovic, D. T. Towsley, and M. Xiong. Integrating Temporal, Real-time, and Active Databases. *Sigmod Record*, 25(1):8–12, 1996.

[RY94] H. Raafat and Z. Yang. Relational Spatial Topologies for Historical Geografical Information. *International Journal of Geographic Information Systems*, 8(2):163–173, 1994.

[SA86] R.T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[SAA+94a] R. T. Snodgrass, I. Ahn, G. Ariav, P. Bayer, J. Clifford, C. Dyreson, F. Grandi, L. Hermosilla, C. Jensen, W. Käfer, N. Kline, T. Leung, and N. Lorentzos. An Evaluation of TSQL2. Commentary, University of Arizona, Department of Computer Science, October 1994. In: The TSQL2 Language Specification.

[SAA+94b] R. T. Snodgrass, I. Ahn, G. Ariav, et al. A TSQL2 Tutorial. *SIGMOD Record*, 23(3):27–33, September 1994.

[SB97] B. Skjellaug and A-J Berre. Multi-dimensional Time Support for Spatial Data Models. Technical Report 253, Institutt for Informatikk, Universitetet i Oslo, May 1997.

[SJS95] A. Segev, C. S. Jensen, and R. T. Snodgrass. Report on The 1995 International Workshop on Temporal Databases. *Sigmod Record*, 24(4):46–52, 1995.

[Skj96] B. Skjellaug. Time and Temporal Data Management - Operationalization in a Temporal GIS. Technical Report STF 40 A 96063, Department of Informatics, University of Oslo, November 1996.

[Skj97a] B. Skjellaug. Temporal Data: Time and Object Databases. Technical Report 245, Department of Informatics, University of Oslo, April 1997.

[Skj97b] B. Skjellaug. Temporal Data: Time and Relational Databases. Technical Report 246, Department of Informatics, University of Oslo, April 1997.

[Sno92] R. T. Snodgrass. Temporal Databases. In A. U. Frank et al., editor, *Proc. of the Int. Conf. on GIS: From Space to Territory*, pages 22–65, 1992.

[Sno93] R. T. Snodgrass, editor. *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, June 1993.

[Sno95a] R. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems: the object model, interoperability and beyond*, pages 386–408. ACM Press, 1995.

[Sno95b] R. T. Snodgrass. Language specification. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 32, pages 599–603. Kluwer Academic Publishers, 1995.

[Sno95c] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[Soo91] M.D. Soo. Bibliography on temporal databases. *SIGMOD Record*, 20(1):14–23, March 1991.

[TCG+94] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Database Systems and Applications Series. Benjamin/Cummings, Redwood City, CA, 1994.

[TJS97] V. J. Tsotras, C. S. Jensen, and R. T. Snodgrass. A Notation for SpatioTemporal Queries. Technical Report TR-10, TimeCenter, April 1997.

[TK96] V.J. Tsotras and A. Kumar. Temporal Database Bibliography Update. *SIGMOD Record*, 25(1):41–51, March 1996.

[TP95] Y. Theodoridis and D. Papadias. Range Queries Involving Spatial Relations: A Performance Analysis. In *Lecture Notes in Computer Science*, volume 988, pages 537–551, 1995.

[VBH96] A. Voigtmann, L. Becker, and K. H. Hinrichs. Temporal extensions for an Object-Oriented Geo-Data-Model. Technical Report Bericht Nr. 6/96-I, Institut für Informatik, Münster, Germany, 1996.

Apêndice A

Código das Classes e Operadores

```
/**************************************************************************
0. Classes Importadas
**************************************************************************/
import Date from o2kit
import Box from o2kit

/**************************************************************************
 1.  Constantes e tipos
**************************************************************************/
name Infinity :Date; /* Constante auxiliar */
name O :Date; /* Constante auxiliar */
name Beginning :string;  /* Utilizada em consultas - referencia inicio da linha
de tempo */
name T_Beginning: Event; /* Objeto de tempo - representa inicio da linha tempora
l */
name Forever :string; /* Utilizada em consultas - referencia fim da linha de tem
po */
name T_Forever: Event; /* Objeto de tempo - representa fim da linha temporal */
name Dialoguer :Box;

run body{
  /* Inicializacao das constantes de tempo */
  O = new Date (1,1,1997);
  Infinity = new Date (1,1,2500);
  /*  183717 -  01/01/2500 */
  Beginning: "01/01/1997";
  Forever: "01/01/2500";
  T_Beginning = event(Beginning);
  T_Forever = event(Forever);
}

type coord :tuple(x: real,
                  y: real);

/**********************************************************************/
/* Classes da infra-estrutura                                         */
/**********************************************************************/

/**********************************************************************
2  Classes Time
**********************************************************************/
class Time inherit Object public
method
  public get_value: integer,
  public num_intervals: integer,
  public get_interval (i:integer): Interval,
  public intervals: list (Interval),
  public begin: Time,
  public end: Time,
  public t_before (t: Time): boolean,
  public t_equal (t: Time): boolean,
  public t_overlaps (t: Time): boolean,
  public t_contains (t: Time): boolean,
  public t_meets (t: Time): boolean,
  public t_inter (t: Time): Time  /* ativado somente quando existe interseoao en
tre os valores de tempo */
end;

class Event inherit Time public type
  integer
```



```
method
  public get_value: integer,
  public begin: Event,
  public end: Event,
  public t_contains (t: Time): boolean,
  public t_inter (t: Time): Time,
  public date_view,
  public get_date: Date,
  public date: string,
  public year: integer,
  public month: integer,
  public day: integer
end;

class Interval inherit Time public type
  tuple (t_inicio: Event, t_fim: Event)
method
  public begin: Event,
  public end: Event,
  public t_contains (t: Time): boolean,
  public t_inter (t: Time): Time
end;

class TempElement inherit Time public type
  list (Interval)
method
  public num_intervals: integer,
  public get_interval (i: integer): Interval,
  public intervals: list (Interval),
  public begin: Event,
  public end: Event,
  public t_overlaps (t: Time): boolean,
  public t_contains (t: Time): boolean,
  public t_inter (t: Time): Time
end;

/**************************************************************************
2.1.  Classes Time - Metodos
**************************************************************************/

/**************************************************************************
2.1.1  Metodos de Time
**************************************************************************/
method body t_before (t: Time): boolean in class Time
{
 o2 boolean resultado;
 o2 integer tb, se;
 se = self -> end -> get_value;
 tb = t -> begin -> get_value;
 if (se < tb) resultado = true;
 else resultado = false;
 return resultado;
};

method body t_equal (t: Time): boolean in class Time
{
 o2 boolean resultado;
 resultado = self -> deep_equal (t);
 return resultado;
};
```

```
method body t_meets (t: Time): boolean in class Time
{
 o2 boolean resultado;
 o2 integer se,tb;
 se = self -> end -> get_value;
 tb = t -> begin -> get_value;
 if (tb - se == 1) resultado = true;
 else resultado = false;
 return resultado;
};

method body t_overlaps (t: Time): boolean in class Time
{
 o2 boolean resultado = false;
 o2 Event e = new Event;
 o2 Interval i = new Interval;
 o2 TempElement telem = new TempElement;
 if ((t->type_of == e->type_of) || (t->type_of == i->type_of)) {
   o2 integer sb,se,tb,te;
   sb = self -> begin -> get_value;
   se = self -> end -> get_value;
   tb = t -> begin -> get_value;
   te = t -> end -> get_value;
   if ((sb <= te) && (tb <= se)) resultado = true;
 }
 else {
  if (t->type_of == telem->type_of){
    o2 list (Interval) intervalos;
    o2 Interval intervalo;
    intervalos = t -> intervals;
    for (intervalo in intervalos) {
      /* Verifica se self "overlaps" intervalo */
      if (self -> t_overlaps (intervalo)){
        resultado = true;
        break;
      }
    }
  }
 }
 return resultado;
};

/*******************************************************************************
2.1.2  Metodos de Event
*******************************************************************************/
method body begin in class Event
{
return self;
};

method body end in class Event
{
return self;
};

method body get_value: integer in class Event
{
  return *self;
```



```
};

method body get_date: Date in class Event
{
  o2 Date data;
  o2 integer dia, mes, ano, dias;
  dia = 0 -> get_day;
  mes = 0 -> get_month;
  ano = 0 -> get_year;
  dias = self -> get_value;
  data = new Date (dia, mes, ano);
  data -> add_days (dias);
  return data;
 };

method body date: string in class Event
{
 o2 string resultado;
 o2 Date data;
 data = self->get_date;
 resultado = data->to_string (tuple (mode:'e'));
 return resultado;
};

method body date_view in class Event
{
 o2 string data;
 data = self -> date;
 display (data);
};

method body year in class Event
{
 o2 Date data;
 data = self -> get_date;
 return data -> get_year;
};

method body month in class Event
{
 o2 Date data;
 data = self -> get_date;
 return data -> get_month;
};

method body day in class Event
{
 o2 Date data;
 data = self -> get_date;
 return data -> get_day;
};

method body t_contains (t: Time): boolean in class Event
{
 /* Um Event so pode conter um outro Event, e isto acontece quando eles sao igua
is */
 return self -> deep_equal (t);
};

method body t_inter (t:Time): Time in class Event
```

```
{
 if (self -> t_overlaps (t)) {
   return self;
 }
 else {
   return (o2 Time) nil;}
};
```

```
/******************************************************************************
2.1.3  Metodos de Interval
******************************************************************************/
method body begin in class Interval
{
return self->t_inicio;
};

method body end in class Interval
{
return self -> t_fim;
};

method body t_contains (t: Time): boolean in class Interval
{
  o2 integer sb,se,tb,te;
  o2 boolean resultado;
  sb = self -> begin -> get_value;
  se = self -> end -> get_value;
  tb = t -> begin -> get_value;
  te = t -> end -> get_value;
  if ((tb >= sb) && (te <= se)) resultado = true;
  else resultado = false;
  return resultado;
};

method body t_inter (t: Time): Time in class Interval
{
 o2 Event e = new Event;
 o2 Interval i = new Interval;
 o2 TempElement telem = new TempElement;
 if (self -> t_overlaps (t)) {
  if (t->type_of == e->type_of) {
    return t;}
  else {
    if (t->type_of == i->type_of) {
      o2 Interval resultado = new Interval;
      o2 Time as, ae, bs, be;
      as = self->begin;
      ae = self->end;
      bs = t->begin;
      be = t->end;
      /* Calcula o limite inferior do intervalo de interseccao: max (as,bs) */
      if (as->get_value > bs->get_value)
         resultado->t_inicio = (o2 Event) as;
      else
         resultado->t_inicio = (o2 Event) bs;
      /* Calcula o limite superior do intervalo de interseccao: min (ae,be) */
      if (ae->get_value < be->get_value)
         resultado -> t_fim = (o2 Event) ae;
      else
         resultado -> t_fim = (o2 Event) be;
```



```
      return resultado;
    }
    else {
      if (t->type_of == telem->type_of) {
        o2 TempElement resultado = new TempElement;
        o2 Interval intervalo, inter=new Interval;
        o2 list (Interval) resultin;
        resultin = list ();
        for (intervalo in t->intervals) {
          inter = (o2 Interval ) self -> t_inter (intervalo);
          if (inter != nil) {
            resultin += list (inter);}
        }
        *resultado = resultin;
        return resultado;
      }
    }
  }
 }
 else {
   return (o2 Time) nil;}
};
```

```
/******************************************************************************
2.1.4  Metodos de TempElement
******************************************************************************/
method body begin in class TempElement
{
 return (*self)[0] -> begin;
};

method body end in class TempElement
{
 o2 integer ult_interval;
 ult_interval = count (*self) - 1;
 return (*self)[ult_interval] -> end;
};

method body num_intervals: integer in class TempElement
{
 o2 integer num;
 num = count (*self);
 return num;
};

method body get_interval (i:integer): Interval in class TempElement
{
 o2 Interval intervalo;
 intervalo = (*self)[i];
 return intervalo;
};

method body intervals: list (Interval) in class TempElement
{
 return *self;
};

method body t_overlaps (t: Time): boolean in class TempElement
{
o2 boolean resultado = false;
```

```
o2 Event e = new Event;
o2 Interval i = new Interval, intervalo;
o2 list(Interval) intervalos;
o2 TempElement te = new TempElement;
if ((t->type_of == e->type_of) || (t->type_of == i->type_of)) {
  intervalos = self->intervals;
  for (intervalo in intervalos) {
    /* Verifica se intervalo "overlaps" t */
    if (intervalo -> t_overlaps (t)){
      resultado = true;
      break;
    }
  }
}
else {
  if (t->type_of == te->type_of){
    o2 integer n,m,i,j;
    o2 Interval is, it;
    n = self -> num_intervals; /* Numero de intervalos na lista */
    m = t -> num_intervals;
    for (i=0; i < n; i++){
       for (j=0; j < m; j++){
         is = self -> get_interval(i);
         it = t -> get_interval(j);
         if (is -> t_overlaps (it)){
            resultado = true;
            break;
         }
       }
    }
    /* Se nao existe nenhum par de intervalos que tenha interseção,a funcao reto
rna falso */
  }
}
return resultado;
};

method body t_contains (t: Time): boolean in class TempElement
{
o2 boolean resultado = false;
o2 Event e = new Event;
o2 Interval i = new Interval, intervalo;
o2 TempElement te = new TempElement;
o2 list (Interval) intervalos;
if ((t->type_of == e->type_of) || (t->type_of == i->type_of)) {
  intervalos = *self;
  for (intervalo in intervalos) {
    /* Verifica se existe intervalo que "contains" t */
    if (intervalo -> t_contains (t)){
      resultado = true;
      break;
    }
  }
}
else {
  if (t->type_of == te->type_of){
    o2 integer n,m,i,j;
    o2 Interval is, it;
    o2 boolean found;
    resultado = true;
```



```
    n = self -> num_intervals; /* Numero de intervalos na lista */
    m = t -> num_intervals;
    for (j=0; j < m; j++){
      it = t -> get_interval (j);
      found = false;
      i = 0;
      /*Verifica se o intervalo it considerado está contido em algum dos interv
alos i do objeto*/
      while ((i < n) && (!found)){
         is = self -> get_interval (i);
         if (is -> t_contains (it)){
           found = true;
         }
         i++;
      }
      if (!found){
        resultado = false;
        break;
      }
    }
    /* Se todos os intervalos de t estiverem contidos em algum intervalo do obje
to, t_contains retorna verdadeiro */
  }
}
return resultado;
};

method body t_inter (t: Time): Time in class TempElement
{
 o2 Event e = new Event;
 o2 Interval i = new Interval;
 o2 TempElement telem = new TempElement;
 if (self -> t_overlaps (t)){
   if (t->type_of == e->type_of) {
    return t;
   }
   else{
     if (t->type_of == i->type_of) {
       o2 TempElement resultado = new TempElement;
       resultado = (o2 TempElement) t -> t_inter (self);
       return resultado;
     }
     else {
      if (t->type_of == telem->type_of) {
        o2 TempElement resultado = new TempElement;
        o2 list (Interval) resultin;
        o2 Interval is, it, ii;
        resultin = list ();
        for (is in self->intervals) {
          for (it in t->intervals){
            if (is -> t_overlaps (it)) {
               ii = (o2 Interval) is -> t_inter (it);
               resultin += list (ii);
            }
          }
        }
        *resultado = resultin;
        return resultado;
      }
    }
```

```
    }
  }
  else {
    return (o2 Time) nil;
  }
};

/*******************************************************************************
 3  Classes Geometricas (Obj_Geom)
*******************************************************************************/
class Obj_Geom public
method
   public vertice (i: integer): tuple (x:real, y:real),
   public centroid: Point,
   public distance (o: Obj_Geom): real,
   public intersect (o: Obj_Geom): boolean,
   public disjoint (o: Obj_Geom): boolean,
   public overlap (o: Obj_Geom): boolean,
   public inside (o: Obj_Geom): boolean,
   public touch (o: Obj_Geom): boolean
end;

class Point inherit Obj_Geom type
 tuple (x:real,
        y:real)
method
   public centroid: Point,
   public inside (o: Polygon): boolean,
   public intersect (o: Obj_Geom): boolean
end;

class Line inherit Obj_Geom type
   tuple (pt_inicio: tuple (x:real, y:real),
          pt_fim: tuple (x:real, y:real))
method
   public vertice (i: integer): tuple (x:real, y:real),
   public horizontal: boolean,
   public vertical: boolean,
   public max_x: real,
   public max_y: real,
   public min_x: real,
   public min_y: real,
   public x_proj (y: real): real,
   public y_proj (x: real): real,
   public centroid: Point,
   public length: real,
   public intersect (o: Obj_Geom): boolean,
   public contains (p: Point): boolean,
   public overlap (o: Line): boolean
end;

class Polygon inherit Obj_Geom type
   list (tuple (x:real, y:real))
method
   public vertice (i: integer): tuple (x:real, y:real),
   public vertices: list (tuple (x:real, y: real)),
   public num_vertices: integer,
   public y (I: integer): real,
   public x (i: integer): real,
   public edge (i: integer): Line,
```



```
   public edges: list(Line),
   public centroid: Point,
   public perimeter: real,
   public area: real,
   public intersect (o: Obj_Geom): boolean,
   public overlap (o: Polygon): boolean,
   public touch (o: Polygon): boolean,
   public inside (o: Polygon): boolean
end;

/*******************************************************************************
3.1  Classes Geometricas (Obj_Geom) - Metodos
*******************************************************************************/

/*******************************************************************************
3.1.1 Metodos de Obj_Geom
*******************************************************************************/
method body disjoint (o: Obj_Geom): boolean in class Obj_Geom
{
  return ! self->intersect (o);
};

method body distance (o: Obj_Geom): real in class Obj_Geom
{
  o2 Point cs, co;
  cs = self -> centroid;
  co = o -> centroid;
  return pp_dist (*cs, *co);
};

/*******************************************************************************
3.1.2 Metodos de Point
*******************************************************************************/
method body inside (o: Polygon): boolean in class Point
{
/* Code from Wm. Randolph Franklin, wrf@ecse.rpi.edu with some minor modificatio
ns */
  int i, j, npol;
  o2 boolean c = false;
  npol = o -> num_vertices;
  for (i=0,j=npol-1; i < npol; j = i++) {
     if ((((o->y(i) <= self->y) && (self->y < o->y(j))) ||
           ((o->y(j) <= self->y) && (self->y < o->y(i)))) &&
          (self->x < (o->x(j) - o->x(i)) * (self->y - o->y(i)) / (o->y(j) - o->y(
i)) + o->x(i))) {
          c = !c;
      }
  }
  return c;
};

method body intersect (o: Obj_Geom): boolean in class Point
{
 o2 Point p = new Point;
 if (o->type_of == p->type_of) {
   if (self -> deep_equal (o) )
      return true;
   else
      return false;
 }
```

```
 else
   return o -> intersect (self);
};

method body centroid: Point in class Point
{
  return self;
};

/*************************************************************************
3.1.3 Metodos de Line
*************************************************************************/
method body horizontal: boolean in class Line
{
 return self->pt_inicio.y == self->pt_fim.y;
};

method body vertical: boolean in class Line
{
 return self->pt_inicio.x == self->pt_fim.x;
};

method body max_x: real in class Line
{
 o2 real x1,x2,resultado;
 x1 = self->pt_inicio.x;
 x2 = self->pt_fim.x;
 o2query (resultado, "max(set($1,$2))",x1,x2);
 return resultado;
};

method body max_y: real in class Line
{
 o2 real y1,y2,resultado;
 y1 = self->pt_inicio.y;
 y2 = self->pt_fim.y;
 o2query (resultado, "max(set($1,$2))",y1,y2);
 return resultado;
};

method body min_x: real in class Line
{
 o2 real x1,x2,resultado;
 x1 = self->pt_inicio.x;
 x2 = self->pt_fim.x;
 o2query (resultado, "min(set($1,$2))",x1,x2);
 return resultado;
};

method body min_y: real in class Line
{
 o2 real y1,y2,resultado;
 y1 = self->pt_inicio.y;
 y2 = self->pt_fim.y;
 o2query (resultado, "min(set($1,$2))",y1,y2);
 return resultado;
};

method body y_proj (x: real): real in class Line
{
```



```
 /* Calculates y, given x */
 o2 real resultado;
 resultado = (self->pt_inicio.y - self->pt_fim.y) /
             (self->pt_inicio.x - self->pt_fim.x) *
             (x - self->pt_inicio.x) +
             self->pt_inicio.y;
 return resultado;
};

method body x_proj (y: real): real in class Line
{
/* Calcula x, dado y (nao pode ser y qualquer), a partir dos pontos inicial e fi
nal da linha considerada.
Formula derivada de: y - y1 = ((y1 - y2)/(x1 - x2)) * (x - x1),
 onde (x1,y1) e (x2,y2) sao pontos da reta. */

 if (self->vertical) return self->pt_inicio.x;

 return (y - self->pt_inicio.y)*
        ((self->pt_inicio.x - self->pt_fim.x)/
         (self->pt_inicio.y - self->pt_fim.y)) +
         self->pt_inicio.x;
};

method body contains (p: Point): boolean in class Line
{
 /* Verify if p satisfies the equation of the line, that is, it's inside the lin
e segment */
 /* It's not necessary that p is an interior point of Line */

 if ( ( (p->x >= self->min_x) && (p->x <= self->max_x) ) &&
      ( (p->y >= self->min_y) && (p->x <= self->max_y) )
    )
    if (!((self->horizontal) || (self->vertical))) {
       /* Verificar se ponto satisfaz equacao da reta */
       if (self -> y_proj (p->x) == p->y) return true;
       else return false;
    }
    else return true;
 else
    return false;
};

method body intersect (o: Obj_Geom): boolean in class Line
{
 o2 Point p = new Point;
 o2 Line l = new Line;
 o2 Polygon pol = new Polygon;


 if (o->type_of == p->type_of) {
   /* Verify intersection between a line and a point */
  if ( self->contains ((o2 Point) o) ) return true;
  else return false;
 };


 if (o->type_of == l->type_of) {
   return LineIntersectionTest (self,(o2 Line) o);
 }
```

```
 if (o->type_of == pol->type_of) {
    /* Verify intersection between a line and a polygon */
    return o -> intersect (self);
 }
};

method body vertice in class Line
{
 if (i == 0)
   return self -> pt_inicio;
 else return self -> pt_fim;
};

method body centroid: Point in class Line
{
 o2 Point c = new Point;
 c -> x = (self->pt_inicio.x + self->pt_fim.x) / 2;
 c -> y = (self->pt_inicio.y + self->pt_fim.y) / 2;
 return c;
};

method body length in class Line
{
 o2 real resultado;
 resultado = pp_dist (self->pt_inicio,self->pt_fim);
 return resultado;
};

method body overlap (o: Line): boolean in class Line
{
  o2 coord a,b,c,d;
  double  s, t;   /* The two parameters of the parametric eqns. */
  double  denom;  /* Denominator of solutions. */
  a = self -> pt_inicio;
  b = self -> pt_fim;
  c = o -> pt_inicio;
  d = o -> pt_fim;

  denom = a.x * ( d.y - c.y ) +
          b.x * ( c.y - d.y ) +
          d.x * ( b.y - a.y ) +
          c.x * ( a.y - b.y );


  if (denom == 0.0) {
           /* the segments are parallel */
           /* verificar se sao colineares, e se se intersectam */
           if ( Collinear (a,b,c) && Collinear (a,b,d) ) {
              if ( self -> vertical )  {
                /* It's a vertical segment */
                if ( ((self->min_y < o->max_y) && (o->min_y < self->max_y)) ||
                     ((o->min_y < self->max_y) && (self->min_y < o->max_y))
 )
                     return true;
                else
                     return false;
              }
              else  {
                if ( ((self->min_x < o->max_x) && (o->min_x < self->max_x)) ||
```



```
                     ((o->min_x < self->max_x) && (self->min_x < o->max_x))
 )
                     return true;
                else return false;
              }
           }
           else
              return false;
  }
  else
       return false;
};

/**************************************************************************
3.1.4 Metodos de Polygon
**************************************************************************/
method body vertices in class Polygon
{
 return *self;
};

method body num_vertices in class Polygon
{
 return count(*self);
};

method body y (i:integer): real in class Polygon
{
  return (*self)[i].y;
};

method body x(i:integer): real in class Polygon
{
  return (*self)[i].x;
};

method body edge (i:integer): Line in class Polygon
{
 o2 list(tuple(x:real,y:real)) pontos;
 o2 Line resultado = new Line;
 o2 integer n_pontos;
 pontos = self->vertices;
 n_pontos = self->num_vertices;
 resultado -> pt_inicio = pontos[i];
 if (i == n_pontos - 1) {
   resultado -> pt_fim = pontos[0];}
 else {
   resultado -> pt_fim = pontos[i+1];}
 return resultado;
};

method body edges: list (Line) in class Polygon
{
 o2 integer n_edges,i;
 o2 list(Line) resultado;
 n_edges = self->num_vertices;
 resultado = list();
 for (i=0;i<n_edges;i++) {
   o2 Line l = new Line;
   l = self -> edge(i);
```

```
   resultado += list(1);
 }
 return resultado;
};

method body vertice (i:integer): tuple(x:real,y:real)  in class Polygon
{
 o2 list(tuple(x:real,y:real)) pontos;
 o2 tuple(x:real,y:real) resultado;
 pontos = self->vertices;
 resultado = pontos[i];
 return resultado;
};

method body intersect (o: Obj_Geom): boolean in class Polygon
{
 /* Return true if there is any intersection between the objects - boundary or i
nterior */
 o2 list (Line) sedges;  /* polygon's edges */
 o2 Line es;
 o2 Point ponto = new Point;

 o2 Line linha = new Line;
 o2 Polygon poligono = new Polygon;
 o2 boolean inter = false;

 sedges = self->edges;

 if ( o->type_of == ponto->type_of ) {
   for ( es in sedges ) {
     if ( es -> contains ((o2 Point) o) ) {
       inter = true;
       break;
     }
   }
   if (! inter) {
     if ( o -> inside (self) ) inter = true;
   }
   return inter;
 }

 if ( o->type_of == linha->type_of ) {
    o2 Point p = new Point;
    for ( es in sedges ) {
      if ( es -> intersect (o) ) {
        inter = true;
        break;
      }
    }
    if (! inter) {
      p -> x = o->vertice(0).x;
      p -> y = o->vertice(0).y;
      if ( p -> inside (self) ) inter = true;  /* inter = true if line lies insi
de Polygon    */
    }
    return inter;
 }

 if (o->type_of == poligono->type_of)
  {
```



```
   return PolygonIntersectionTest (self, (o2 Polygon) o);
  }
};

method body centroid: Point in class Polygon
{
/*
       Written by Joseph O'Rourke
       orourke@cs.smith.edu
       October 27, 1995

       Computes the centroid (center of gravity) of an arbitrary
       simple polygon via a weighted sum of signed triangle areas,
       weighted by the centroid of each triangle.
       Reads x,y coordinates from stdin.
       NB: Assumes points are entered in ccw order!
       E.g., input for square:
               0       0
               10      0
               10      10
               0       10


       Returns the cg in CG.  Computes the weighted sum of
       each triangle's area times its centroid.  Twice area
       and three times centroid is used to avoid division
       until the last moment.
*/
    o2 integer  i, n;
    o2 real  A2, Areasum2 = 0;         /* Partial area sum */
    o2 Point Cent3 = new Point, CG = new Point;
    o2 coord a, b, c;
    CG->x = 0;
    CG->y = 0;
    n = self -> num_vertices;
    a = self->vertice(0);
    for (i = 1; i < n-1; i++) {
        b =  self->vertice(i);
        c =  self->vertice(i+1);
        *Cent3 = Centroid3( a, b, c);
        A2 =  Area2(a, b, c);
        CG->x += A2 * Cent3->x;
        CG->y += A2 * Cent3->y;
        Areasum2 += A2;
    }
    CG->x /= 3 * Areasum2;
    CG->y /= 3 * Areasum2;
    return CG;
};

method body perimeter in class Polygon
{
 o2 real resultado, n_coord;
 o2 integer i;
 o2 Line ed;

 resultado = 0;
 n_coord = self -> num_vertices;

 for (i=0; i < n_coord; i++) {
```

```
   ed = self -> edge (i);
   resultado +=  ed -> length;
 }
 return resultado;
};

method body overlap (o: Polygon): boolean in class Polygon
{
  o2 Polygon polinter = new Polygon;
  if ( ConvexIntersect (self, o, polinter) ) {
     /* verify if intersection is diferent from A (self) and B (o) */
     if ( ! ( polinter->deep_equal (self) || polinter -> deep_equal (o) ) ) {
       /* verify if polinter is,in fact, a polygon */
       if (polinter -> num_vertices < 3)
         return false; /* polinter is not a polygon */
       else {
          o2 coord a, b, c;
          a = polinter->vertice(0);
          b = polinter->vertice(1);
          c = polinter->vertice(2);
          if ( Collinear (a,b,c) )
             return false;
          else return true;
       }
     }
     else return false;
  }
  else return false;
};

method body area in class Polygon
{
  o2 integer i, n;
  o2 real result;
  o2 coord a, b, c;
  result = 0.0;
  n = self -> num_vertices;
  a = self->vertice(0);
  for (i = 1; i < n-1; i++) {
     b =  self->vertice(i);
     c =  self->vertice(i+1);
     result += Area2 (a,b,c);
  }
  result /= 2;
  return result;
};

method body touch (o: Polygon): boolean in class Polygon
{
  o2 Polygon polinter = new Polygon;
  if ( ConvexIntersect (self, o, polinter) ) {
     /* verify if intersection is diferent from A (self) and B (o) */
     if ( ! ( polinter->deep_equal (self) || polinter -> deep_equal (o) ) ) {
       /* verify if polinter is not a polygon */
       if (polinter -> num_vertices < 3)
         return true; /* polinter is not a polygon */
       else {
          o2 coord a, b, c;
          a = polinter->vertice(0);
```



```
          b = polinter->vertice(1);
          c = polinter->vertice(2);
          if ( Collinear (a,b,c) )
             return true;
          else return false;
       }
     }
     else return false;
  }
  else return false;
};

method body inside (o: Polygon): boolean in class Polygon
{
  o2 Polygon polinter = new Polygon;
  if ( ConvexIntersect (self, o, polinter) )
     if ( polinter->deep_equal (self) ) /* verify if intersection is equal A (se
lf) */
        return true;
     else return false;
  else return false;
};

/*******************************************************************************
 4 Classes Geometricas Temporalizadas (Obj_GeomT)
*******************************************************************************/
class Obj_GeomT public
 method
   public valice (t:Time): Obj_GeomT,
   public valice_s (t:Event): Obj_Geom
end;

class PointT inherit Obj_GeomT type
  tuple (coord: tuple (x:real, y:real),
         t: Time)
method
  public valice (t:Time): PointT,
  public valice_s (t:Event): Point
end;

class LineT inherit Obj_GeomT type
  list (tuple (pts: tuple (pt_inicio: tuple (x:real, y:real),
                           pt_fim: tuple (x:real, y:real)),
               t:Time))
method
  public valice (t:Time): LineT,
  public valice_s (t:Event): Line
end;

class PolygonT inherit Obj_GeomT type
  list (tuple (lst_coord: list (tuple (x:real, y:real)),
               t: Time))
method
  public valice (t:Time): PolygonT,
  public valice_s (t:Event): Polygon
end;

/*******************************************************************************
 4.1 Classes Geometricas Temporalizadas (Obj_GeomT) - Metodos
*******************************************************************************/
```

```
/*****************************************************************************
 4.1.1  Metodos de PointT
*****************************************************************************/
method body valice (t: Time): PointT in class PointT
{
 o2 PointT resultado;
 if (self -> t -> t_overlaps (t)) { /* se PointT existe no tempo t */
   resultado = new PointT;
   resultado -> t = self -> t  -> t_inter (t);
   resultado -> coord = self -> coord;
 }
 return resultado;
};

method body valice_s (t: Event): Point in class PointT
{
   o2 Point resultado;
   if (self -> t -> t_overlaps (t)) {
     resultado = new Point;
     *resultado = self -> coord;
   }
   return resultado;
};

/*****************************************************************************
 4.1.2  Metodos de LineT
*****************************************************************************/
method body valice (t: Time): LineT in class LineT
{
  o2 LineT resultado;
  o2 tuple (pts: tuple (pt_inicio:tuple (x:real, y:real), pt_fim: tuple (x:real,
 y:real)), t: Time) versao;
  o2 list (tuple (pts: tuple (pt_inicio:tuple (x:real, y:real), pt_fim: tuple (x
:real, y:real)), t: Time)) lpts;
  lpts = list();
  for (versao in *self) {
    if (versao.t -> t_overlaps (t)) {
      versao.t = versao.t -> t_inter (t);
      lpts += list (versao);
    }
  }
  if (lpts != list()) {
    resultado = new LineT;
    *resultado = lpts;
  }
  return resultado;
};

method body valice_s (t: Event): Line in class LineT
{
  o2  tuple (pts: tuple (pt_inicio: tuple (x:real, y:real), pt_fim: tuple (x:rea
l, y:real)),t:Time) estado;
  o2 Line resultado;
  for (estado in *self) {
    if (estado.t -> t_overlaps (t)) {
      resultado = new Line;
      *resultado = estado.pts;
      break;
    }
```



```
  }
  return resultado;
};

/*****************************************************************************
4.1.3  Metodos de PolygonT
*****************************************************************************/
method body valice (t:Time): PolygonT in class PolygonT
{
 o2 PolygonT resultado;
 o2 list (tuple (lst_coord: list (tuple (x:real, y:real)),t: Time)) llns;
 o2 tuple (lst_coord: list (tuple (x:real, y:real)),t: Time) versao;
 llns = list ();
 for (versao in *self) {
   if (versao.t -> t_overlaps (t)) {
     versao.t = versao.t -> t_inter (t);
     llns += list (versao);
   }
 }
 if (llns != list()) {
   resultado = new PolygonT;
   *resultado = llns;
 }
 return resultado;
};

method body valice_s (t: Event): Polygon in class PolygonT
{
  o2 Polygon resultado;
  o2 tuple (lst_coord: list (tuple (x:real, y:real)), t: Time) estado;
  for (estado in *self) {
   if (estado.t -> t_overlaps (t)) {
     resultado = new Polygon;
     *resultado = estado.lst_coord;
     break;
   }
  }
  return resultado;
};

/*****************************************************************************
5  Classes de Geometria
*****************************************************************************/
class Geom inherit Object public type
   tuple (atribs: real,
          objs: list (Obj_Geom))
method
   public select_lst_objs: list (Obj_Geom)
end;

class GeomT inherit Object public type
   tuple (atribs: real,
          objs: list (tuple (lst_objs: list (Obj_GeomT),
                             t: Time)))
method
   public t_select_lst_objs (t: Time): list (tuple(lst_objs:list(Obj_GeomT),t:Ti
me)),
   public valice (t:Time): GeomT
end;
```

```
/******************************************************************************
 5.1  Classes de Geometria - Metodos
******************************************************************************/
method body select_lst_objs in class Geom
{
 return self -> objs;
};

method body t_select_lst_objs (t: Time): list (tuple (lst_objs: list(Obj_GeomT),
 t: Time)) in class GeomT
{
   o2 list (tuple(lst_objs: list (Obj_GeomT),t: Time)) resultado;
   o2 tuple (lst_objs: list (Obj_GeomT),t:Time) lista_obj;
   o2 Obj_GeomT obj;
   o2 list (Obj_GeomT) vobjs;
   resultado = list();
   /* seleciona as versoes da geometria que estao em t */
   for (lista_obj in self->objs) {
     if (lista_obj.t -> t_overlaps (t)) {
       o2 Time ti;
       ti = lista_obj.t -> t_inter (t);
       vobjs = list ();
       for (obj in lista_obj.lst_objs){
         o2 Obj_GeomT vobj;
         vobj = (o2 Obj_GeomT) obj -> valice (ti); /* reduz estados de obj */
         if (vobj != nil) {
            vobjs += list (vobj);
         }
       }
       if (vobjs != list()) {
         /* Teste feito pra evitar problemas de inconsistencia */
         resultado += list(tuple(lst_objs:vobjs, t:ti));}
     }
   }
   return resultado;
};

method body valice (t:Time): GeomT in class GeomT
{
 o2 GeomT resultado;
 o2 list (tuple (lst_objs: list (Obj_GeomT), t: Time)) lobjs;
 lobjs = self -> t_select_lst_objs (t);
 if (lobjs != list()) {
   resultado = new GeomT;
   resultado -> atribs = self -> atribs;
   resultado -> objs = lobjs;
 }
 return resultado;
};

/******************************************************************************
 6  Classes de Localizacao
******************************************************************************/
class Location inherit Object public
end;

class Sp inherit Location type
   list (Geom)
method
   public select_geometry: list (Obj_Geom)
```



```
end;

class SpT inherit Location type
  list (GeomT)
method
  public t_select_geometry (t:Time): list (tuple (lst_objs:list(Obj_GeomT),t:Tim
e)),
  public valice (t:Time): SpT
end;

/******************************************************************************
6.1  Classes de Localizacao - Metodos
******************************************************************************/
method body select_geometry in class Sp
{
 /* Seleciona a representacao adequada */
 o2 list (Obj_Geom) resultado;
 o2 Geom geo;
 resultado = list();
 for (geo in *self) {
    if (right_scale (geo->atribs)) {
        resultado = geo -> select_lst_objs;
        /*So existe uma escala correta */
    }
 }
 return resultado;
};

method body t_select_geometry (t: Time): list (tuple(lst_objs: list (Obj_GeomT),
 t: Time)) in class SpT
{
  /* Seleciona a representacao adequada */
  o2 list (tuple(lst_objs: list (Obj_GeomT),t: Time)) resultado;
  o2 GeomT geo;
  resultado = list();
  for (geo in *self) {
    if (right_scale(geo->atribs)){
      /* Se geo.atribs - aqui a escala - for um atributo temporalizado, ver qual
 o atributo valido em t */
        resultado += geo -> t_select_lst_objs (t);
        break;
        /*So existe uma escala correta */
    }
  }
  return resultado;
};

method body valice (t:Time): SpT in class SpT
{
 o2 SpT resultado;
 o2 list (GeomT) resultin;
 o2 GeomT rep;
 resultin = list();
 for (rep in *self) {
   o2 GeomT vrep;
   vrep = rep -> valice (t);
   if (vrep != nil)
      resultin += list (vrep);
 }
 if (resultin != list()) {
```

```
    resultado = new SpT;
    *resultado = resultin;
 }
 return resultado;
};

/******************************************************************************
7 Funcoes de proposito geral
******************************************************************************/

function lst_deep_equal (l1: list (Object), l2: list (Object)): boolean;

function body lst_deep_equal
{
 /* Compara duas listas de objetos e retorna verdadeiro se o conteudo de cada pa
r de objetos na mesma posicao da lista for igual */
 o2 integer n,m,k;
 o2 Object objl1, objl2;
 o2 boolean resultado;
 resultado = true;
 n = count (l1); /* Numero de objetos na lista */
 m = count (l2);
 if (n == m){
   for (k=0; k < n; k++){
     objl1 = l1[k];
     objl2 = l2[k];
     if (objl1 -> deep_equal (objl2));
     else {
        resultado = false;
        break;
     }
   }
 }
 else resultado = false;
 return resultado;
};

function is_a_true (A: list (tuple (res:boolean, t:Time))): boolean;

function body is_a_true
{
  o2 tuple (res:boolean, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res == false) {
          resultado = false;
          break;
       }
     }
  }
  else resultado = false;
  return resultado;
};

function is_s_true (A: list (tuple (res:boolean, t:Time))): boolean;

function body is_s_true
```



```
{
  o2 tuple (res:boolean, t:Time) elemres;
  o2 boolean resultado;
  resultado = false;
  if ( A != list() ) {
     /* O resultado e' falso inicialmente */
     for (elemres in A) {
       if (elemres.res == true) {
          resultado = true;
          break;
       }
     }
  }
  return resultado;
};

function gt (A: list (tuple (res:real, t:Time)), valor: real): boolean;

function body gt
{
  o2 tuple (res:real, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res <= valor) {
         resultado = false;
         break;
       }
     }
  }
  else resultado = false;
  return resultado;
};

function ge (A: list (tuple (res:real, t:Time)), valor: real): boolean;

function body ge
{
  o2 tuple (res:real, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res < valor) {
         resultado = false;
         break;
       }
     }
  }
  else resultado = false;
  return resultado;
};

function lt (A: list (tuple (res:real, t:Time)), valor: real): boolean;

function body lt
{
```

```
  o2 tuple (res:real, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res >= valor) {
         resultado = false;
         break;
       }
     }
  }
  else resultado = false;
  return resultado;
};

function le (A: list (tuple (res:real, t:Time)), valor: real): boolean;

function body le
{
  o2 tuple (res:real, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res > valor) {
         resultado = false;
         break;
       }
     }
  }
  else resultado = false;
  return resultado;
};

function eq (A: list (tuple (res:real, t:Time)), valor: real): boolean;

function body eq
{
  o2 tuple (res:real, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res != valor) {
         resultado = false;
         break;
       }
     }
  }
  else resultado = false;
  return resultado;
};

function ne (A: list (tuple (res:real, t:Time)), valor: real): boolean;

function body ne
{
```



```
  o2 tuple (res:real, t:Time) elemres;
  o2 boolean resultado;
  if ( A != list() ) {
     /* O resultado e' verdadeiro inicialmente */
     resultado = true;
     for (elemres in A) {
       if (elemres.res == valor) {
         resultado = false;
         break;
       }
     }
  }
  else resultado = false;
  return resultado;
};
function right_scale (atrib: real): boolean;

function body right_scale
 {
 /* map_scale deve ser fornecida pela aplicacao */
 if (atrib == map_scale())
   return true;
 else return false;
 };

function map_scale: real;

function body map_scale
{
 return 0.0;  /* dummy */
};

function to_obj_geom (c: list(coord)): Obj_Geom;

function body to_obj_geom
{
  if (count(c) == 0) {
     return (o2 Obj_Geom) nil;
     display ("Invalid number of coordinates");
  }
  else if (count(c) == 1) {
    o2 Point ponto = new Point;
    *ponto = c[0];
    return ponto;
  }
  else if (count(c) == 2) {
     o2 Line linha = new Line;
     linha->pt_inicio = c[0];
     linha->pt_fim = c[1];
     return linha;
  }
  else {
      o2 Polygon poligono = new Polygon;
      *poligono = c;
      return poligono;
  }
};

function to_obj (c: list (coord)): SpatialObject;
```

```
function body to_obj
{
  /* Retorna geo-objeto espacial fraco - com apenas o componente espacial */
  o2 Obj_Geom geometrico;
  o2 SpatialObject resultado;
  geometrico = to_obj_geom (c);
  if (geometrico != nil) {
    o2 Geom g = new Geom;
    o2 Sp s = new Sp;
    g -> objs = list (geometrico);
    *s = list (g);
    resultado = new SpatialObject;
    resultado -> l = s;
  }
  return resultado;
};

/******************************************************************************
8 Funcoes Espaciais
******************************************************************************/
/**********************definitions********************************************/
function pp_dist (coord1: coord, coord2: coord): real;

function Collinear (a: coord, b: coord, c: coord ): boolean;  /* Verify se 3 poi
nts are collinears */

function Area2( a: coord, b: coord, c: coord): integer;
/*
        Returns twice the signed area of the triangle determined by a,b,c,
        positive if a,b,c are oriented ccw, and negative if cw.
*/

function LineIntersectionTest (l1: Line, l2: Line): boolean;

function Intersection (a: coord, b: coord, c: coord, d: coord, p: Point): boolea
n;

function SubVec (a: coord, b: coord): coord;

function Left ( a: coord, b: coord, c: coord ): boolean;
/*
        Returns true iff c is strictly to the left of the directed
        line through a to b.
*/

function LeftOn ( a: coord, b: coord, c: coord ): boolean;

function ConvexIntersect (P: Polygon, Q: Polygon, result: Polygon): boolean;

function PolygonIntersectionTest (pol1: Polygon, pol2: Polygon): boolean;

function disjoint (A: list (Obj_Geom), B: list (Obj_Geom)): boolean;

function overlap (A: list (Obj_Geom), B: list (Obj_Geom)): boolean;

function length ( A: list (Line) ): real;

function Centroid3 ( p1: coord, p2: coord, p3: coord ): coord;

function area ( A: list (Polygon) ): real;
```



```
function perimeter ( A: list (Polygon) ): real;

function distance (A: list (Obj_Geom), B: list (Obj_Geom)): real;

function touch (A: list (Obj_Geom), B: list (Obj_Geom)): boolean;

function inside (A: list (Obj_Geom), B: list (Obj_Geom)): boolean;

/*********************************bodies*****************************/
function body pp_dist
{
#include <math.h>
  o2 real resultado,x,y,x2,y2;
  x = coord1.x - coord2.x;
  y = coord1.y - coord2.y;
  x2 = x * x;
  y2 = y * y;
  resultado = sqrt(x2 + y2);
  return resultado;
};

function body Collinear
{
   return  Area2( a, b, c ) == 0;
};


function body Area2
{
        return
                a.x * b.y - a.y * b.x +
                a.y * c.x - a.x * c.y +
                b.x * c.y - c.x * b.y;
};

function body Intersection
{
 /* Verify if there exists an intersection between two line segments
 +  Code from O'Rourke with minor modifications                          */
        double  s, t;   /* The two parameters of the parametric eqns. */
        double  denom;  /* Denominator of solutions. */

        denom = a.x * ( d.y - c.y ) +
                b.x * ( c.y - d.y ) +
                d.x * ( b.y - a.y ) +
                c.x * ( a.y - b.y );

        /* If denom is zero, then the line segments are parallel. */
        /* In this case, return false even though the segments might overlap. */

        /* testar extremidades de segmentos coincidentes */
        if (denom == 0.0) {
           if (a == c) {
              p->x = a.x;
              p->y = a.y;
              return true;
           }
           if (a == d) {
              p->x = a.x;
```

```
            p->y = a.y;
            return true;
         }
         if (b == c) {
            p->x = b.x;
            p->y = b.y;
            return true;
         }
         if (b == d) {
            p->x = b.x;
            p->y = b.y;
            return true;
         }
         return false;
      }

      s = (
            a.x * ( d.y - c.y ) +
            c.x * ( a.y - d.y ) +
            d.x * ( c.y - a.y )
         ) / denom;
      t = -(
            a.x * ( c.y - b.y ) +
            b.x * ( a.y - c.y ) +
            c.x * ( b.y - a.y )
         ) / denom;

      p->x = a.x + s * ( b.x - a.x );
      p->y = a.y + s * ( b.y - a.y );


      if (    (0.0 <= s) && (s <= 1.0) &&
              (0.0 <= t) && (t <= 1.0)
         )
              return  true;
      else    return  false;
};

function body SubVec (a: coord, b: coord): coord
{
 o2 coord result;
 result.x = a.x - b.x;
 result.y = a.y - b.y;
 return result;
};

function body Left ( a: coord, b: coord, c: coord ): boolean
{
        return  Area2( a, b, c ) > 0;
};

function body LeftOn ( a: coord, b: coord, c: coord ): boolean
{
        return  Area2( a, b, c ) >= 0;
};

function body ConvexIntersect (P: Polygon, Q: Polygon, result: Polygon): boolean

{
typedef enum { Pin, Qin, Unknown } tInFlag;
```



```
        o2 integer     n, m;             /* P has n vertices, Q has m vertices. */
        o2 integer     a, b;             /* indices on P and Q (resp.) */
        o2 integer     a1, b1;           /* a-1, b-1 (resp.) */
        o2 coord   A, B;    /* directed edges on P and Q (resp.) */
        o2 integer     cross;            /* A x B */
        o2 boolean    bHA, aHB;         /* b in H(A); a in H(b). */
        o2 coord Origin = tuple (x:0.0,y:0.0);          /* (0,0) */
        o2 Point p = new Point; /* point of intersection */
        tInFlag inflag;              /* {Pin, Qin, Unknown}: which polygon is known t
o be inside */
        int      i;                  /* loop counter */
        int      aa, ba;             /* # advances on a & b indices
                                        (from first intersection) */
        o2 boolean Reset = false;  /* have the advance counters ever been reset
*/
        o2 list (coord) resultin;
        o2 Point paux = new Point;  /* auxiliar object */

        /* Initialize variables. */
        a = 0;
        b = 0;
        aa = 0;
        ba = 0;
        i = 0;
        n = P -> num_vertices;
        m = Q -> num_vertices;
        inflag = Unknown;
        resultin = list();
        do {
                /* Computations of key variables. */
                a1 = (a + n - 1) % n;
                b1 = (b + m - 1) % m;

                A = SubVec( (*P)[a], (*P)[a1] );
                B = SubVec( (*Q)[b], (*Q)[b1] );
                cross = Area2 ( Origin, A, B );
                bHA = Left( (*P)[a1], (*P)[a], (*Q)[b] );
                aHB = Left( (*Q)[b1], (*Q)[b], (*P)[a] );
                /* If A & B intersect, update inflag. */
                if ( Intersection( (*P)[a1], (*P)[a], (*Q)[b1], (*Q)[b], p ) ) {
                        if ( (inflag == Unknown)  && (!Reset) ) {
                                aa = ba = 0;
                                Reset = true;
                        }

                        /* inflag = InOut( p, inflag, aHB, bHA ); */
                        if ((*p in resultin) < 0)  /* Verify if coordinate in re
sultin */

                            resultin += list (*p);
                        /* Update inflag. */
                        if ( aHB )
                           inflag = Pin;
                        else {
                          if ( bHA )
                               inflag = Qin;
                        }
                }
                /* Advance rules. */
                if ( (cross == 0) && !bHA && !aHB ) {
                        if ( inflag == Pin ) {
```

```
                    /* b = Advance( b, &ba, m, inflag == Qin, (*Q)[b] );
*/
                    if (inflag == Qin) {
                       resultin += list ((*Q)[b]);
                    }
                    ba++;
                    b = (b+1) % m;
                 }
                 else {
                       /*  a = Advance( a, &aa, n, inflag == Pin, (*P)[a]
 ); */
                       if (inflag == Pin) {
                          resultin += list ((*P)[a]);
                       }
                       aa++;
                       a = (a+1) % n;
                 }

            }
            else if ( cross >= 0 ) {
                    if ( bHA ) {
                        /* a = Advance( a, &aa, n, inflag == Pin, (*P)[a]
 ); */
                       if (inflag == Pin) {
                          resultin += list ((*P)[a]);
                       }
                       aa++;
                       a = (a+1) % n;
                    }
                    else {
                      /* b = Advance( b, &ba, m, inflag == Qin, (*Q)[b] );
*/
                      if (inflag == Qin) {
                         resultin += list ((*Q)[b]);
                      }
                      ba++;
                      b = (b+1) % m;
                     }
            }
            else /* if ( cross < 0 ) */ {
                    if ( aHB ) {
                        /*   b = Advance( b, &ba, m, inflag == Qin, (*Q)[b]
 ); */
                       if (inflag == Qin) {
                          resultin += list ((*Q)[b]);
                       }
                       ba++;
                       b = (b+1) % m;
                    }
                    else {
                       /*  a = Advance( a, &aa, n, inflag == Pin, (*P)[a]
 ); */
                       if (inflag == Pin) {
                          resultin += list ((*P)[a]);
                       }
                       aa++;
                       a = (a+1) % n;
                    }
            }
      } while ( ((aa < n) || (ba < m)) && (aa < 2*n) && (ba < 2*m) );
```



```
      /* Quit when both adv. indices have cycled, or one has cycled twice. */

      /* Deal with special cases: not implemented here. */
      if ( inflag == Unknown) {
        /* The boundaries of P and Q do not cross */
        if (resultin != list()) {
             /* the boundaries have some intersection */
             *result = resultin;
             return true;
        }
        else {
          /* Verify if P in Q */
          paux -> x = P -> vertice(0).x; /* Obtem 1o vertice de A */
          paux -> y = P -> vertice(0).y;
          if (  paux -> inside (Q) ) {
             *result = *P;
             return true;
             /* p in Q */
          }
          else { /* Verify if Q in P */
             paux -> x = Q -> vertice(0).x; /* Obtem 1o vertice de A */
             paux -> y = Q -> vertice(0).y;
             if ( paux -> inside (P) ) {
                *result = *Q;
                return true;
                /* there is point in Q such as point in P*/
             }
             else
               return false;
          }
        }
      }
      else {
             *result = resultin;
              return true;
      }
};

function body LineIntersectionTest (l1: Line, l2: Line): boolean
{
  o2 tuple(x:real, y:real) a,b,c,d;
  double  s, t;   /* The two parameters of the parametric eqns. */
  double  denom;  /* Denominator of solutions. */
  o2 Point p = new Point;
  a = l1 -> pt_inicio;
  b = l1 -> pt_fim;
  c = l2 -> pt_inicio;
  d = l2 -> pt_fim;

  denom = a.x * ( d.y - c.y ) +
          b.x * ( c.y - d.y ) +
          d.x * ( b.y - a.y ) +
          c.x * ( a.y - b.y );


  if (denom == 0.0) {
          /* verificar se sao colineares, e se se intersectam */
          if ( Collinear (a,b,c) && Collinear (a,b,d) ) {
             if ( l1 -> vertical )  {
```

```
            /* It's a vertical segment */
            if ( (l1->min_y <= l2->max_y) && (l2->min_y <= l1->max_y) )
                return true;
            else return false;
          }
          else  {
            if ( (l1->min_x <= l2->max_x) && (l2->min_x <= l1->max_x) )
                return true;
            else return false;
          }
        }
        else
          return false;
  }
  else
      return Intersection (a,b,c,d,p);

};

function body PolygonIntersectionTest (pol1: Polygon, pol2: Polygon): boolean
{
   /* Based on Preparata & Shamos's algorithm: Polygon Intersection Test */
    o2 Line eo1, eo2;
    o2 Point p = new Point;
    o2 list (Line) pol1edges, pol2edges;
    o2 boolean inter = false;

    pol1edges = pol1 -> edges;
    pol2edges = pol2 -> edges;

    /* Verify if there is intersection between edges of the polygons */
    for ( eo1 in pol1edges ) {
      for ( eo2 in pol2edges ) {
        if ( LineIntersectionTest (eo1,eo2) ) {
          inter = true;
          break;
        }
      }
    }

  /* If there isn't intersection between edges, verify if pol1 lies inside pol2
or vice-versa */

    if (!inter) {
      p -> x = pol1->vertice(0).x; /* Obtem 1o vertice de pol1 */
      p -> y = pol1->vertice(0).y;
      if (  p -> inside (pol2) ) {
        inter = true;
        /* p esta contido em pol2 */
      }
      else {
        p -> x = pol2 -> vertice(0).x; /* Obtem 1o vertice de pol2 */
        p -> y = pol2 -> vertice(0).y;
        if ( p -> inside (pol1) ) {
          inter = true;
          /* existe ponto de pol2 contido contido em pol1*/
        }
      }

      /* Else there isn't intersection */
```



```
    }
    return inter;
};

function body disjoint (A: list (Obj_Geom), B: list (Obj_Geom)): boolean
{
 o2 boolean resultado;
 o2 Obj_Geom objA, objB;
 if ((A == list()) || (B == list())) return true; /* A ou B indefinidos */
 resultado = true;
 for ( objA in A ){
    for ( objB in B ){
       if ( ! ( objA -> disjoint (objB) ) ) {
         resultado = false;
         break;
       }
    }
 }
 return resultado;
};

function body overlap (A: list (Obj_Geom), B: list (Obj_Geom)): boolean
{
 o2 boolean resultado;
 o2 Obj_Geom objA, objB;
 if ((A == list()) || (B == list())) return false; /* A ou B indefinidos */
 resultado = false;
 for ( objA in A ){
   for ( objB in B ){
      if ( objA -> overlap (objB) ) {
        resultado = true;
        break;
      }
   }
 }
 return resultado;
};

function body length ( A: list (Line) ): real
{
  o2 real resultado;
  o2 Line obj;
  resultado = 0.0;
  for ( obj in A ) {
     resultado += obj -> length;
  }
  return resultado;
};

function body Centroid3 ( p1: coord, p2: coord, p3: coord ): coord
{
  o2 coord c;
  c.x = p1.x + p2.x + p3.x;
  c.y = p1.y + p2.y + p3.y;
  return c;
};

function body area ( A: list (Polygon) ): real
{
  o2 real resultado;
```

```
  o2 Polygon obj;
  resultado = 0.0;
  for ( obj in A ) {
     resultado += obj -> area;
  }
  return resultado;
};

function body perimeter ( A: list (Polygon) ): real
{
  o2 real resultado;
  o2 Polygon obj;
  resultado = 0.0;
  for ( obj in A ) {
     resultado += obj -> perimeter;
  }
  return resultado;
};

function body distance (A: list (Obj_Geom), B: list (Obj_Geom)): real
{
  o2 set (real) distancias, dist_A;
  o2 Obj_Geom objA, objB;
  o2 real dist;
  if ((A == list ()) || (B == list ()) )
    return dist;
  else {
    for ( objA in A ){
      dist_A = set ();
      for ( objB in B ){
        dist = objA -> distance (objB);
        dist_A += set (dist);
      }
      o2query (dist,"min($1)", dist_A);
      distancias += set (dist);
    }
    o2query (dist,"max($1)", distancias);
  }
  return dist;
};

function body inside (A: list (Obj_Geom), B: list (Obj_Geom)): boolean
{
 o2 boolean resultado, found;
 o2 Obj_Geom objA, objB;
 if ((A == list ()) || (B == list ()) )
    return false;
 else {
   resultado = true;
   for ( objA in A ) {
     found = false;
       for ( objB in B ) {
         if ( objA -> inside (objB) ) {
           found = true; /* Existe um objeto em B que contem objA */
           break;
         }
       }
       if (!found) {
          resultado = false;
          break;
```



```
       }
    }
    return resultado;
 }
};

function body touch (A: list (Obj_Geom), B: list (Obj_Geom)): boolean
{
 o2 boolean resultado, found;
 o2 Obj_Geom objA, objB;
 if ((A == list ()) || (B == list ()) )
    return false;
 resultado = false;
 for ( objA in A ) {
   for ( objB in B ) {
      if ( objA -> touch (objB) ) {
        resultado = true;
        break;
      }
   }
 }
 return resultado;
};

/*************************************************************************
9 Funcoes Temporais
*************************************************************************/
function Event_to_TempElement (le: list (Event)): TempElement;

function body Event_to_TempElement
{
 /* transforma le (lista de eventos) em TempElement */
 o2 integer ult,qtde,i,k;
 o2 TempElement resultado = new TempElement;
 o2 Event inicio;
 qtde = count (le);
 ult = qtde - 1;
 i = 0;
 while (i < qtde) {
   inicio = le[i];
   while ((i < ult) && (le[i+1]->get_value - le[i]->get_value == 1)) {
    i++;
   }
   if (i < qtde){
     o2 Interval intervalo = new Interval;
     intervalo -> t_inicio = inicio;
     intervalo -> t_fim = le[i];
     (*resultado) += list (intervalo);
   }
   i++;
 }
 return resultado;
};


function Date_to_Integer (data: Date): integer;

function body Date_to_Integer
{
 /* Converte uma granularidade composta (DD/MM/YYYY) para uma granularidade simp
```

```
les do tipo integer, que representa a quantidade de dias a partir da origem do e
ixo temporal */
  o2 integer resultado;
  resultado = data -> diff (0);
  return resultado;
};

function event (data: string): Event;

function body event
{
 /* Implementa a operacao EVENT (data) */
 /* Funcao usada para converter datas fornecidas pelo usuario de uma aplicacao,
em um objeto do tipo Time (parametro dos operadores temporais) - Event  */
  o2 Event resultado = new Event;
  o2 integer conteudo;
  o2 Date dd = new Date(01,01,1997);
  while (! dd->to_date(tuple(mode:'e',s_date:data)) ){
      data = Dialoguer->dialog("Data invalida! Forneca novo valor (dd/mm/aaaa).",
"");
  }
  while ((dd->diff(0) < 0) || (dd->diff(Infinity) > 0)) {
      data = Dialoguer->dialog("Data fora do limite estabelecido (01/01/1997-01/0
1/2500)! Forneca novo valor (dd/mm/aaaa).","");
      dd->to_date(tuple(mode:'e',s_data:data));
  }
  conteudo = Date_to_Integer (dd);
  *resultado = conteudo;
  return resultado;
};

function Now: string;

function body Now
{
 /* Funcao que implementa a variavel now - retorna a data atual */
  o2 Date curr_date = new Date (01,01,1997);
  o2 string resultado;
  curr_date -> set_to_current_date; /* Obtem data atual */
  resultado =  curr_date -> to_string (tuple (mode:'e'));
  return resultado;
};

function twhen (A: list (tuple (res:boolean, t:Time))): TempElement;

function body twhen
{
  o2 tuple (res:boolean, t:Time) elemres;
  o2 TempElement resultado;
  o2 list (Interval) tmpres;
  o2 Event evento = new Event;
  o2 Interval intervalo = new Interval;
  /* O resultado e' nulo inicialmente */
  tmpres = list ();
  for (elemres in A) {
    if (elemres.res == true) {
       if (elemres.t->type_of ==  evento->type_of) {
          /* Pela implementacao atual isto nao vai acontecer...*/
          o2 Interval i = new Interval;
          i->t_inicio = i->t_fim = (o2 Event) elemres.t;
```



```
          tmpres += list (i);
       }
       else if (elemres.t->type_of == intervalo->type_of)
            tmpres += list ((o2 Interval) elemres.t);  /* Adiciona t ao result
ado final */
       else
            tmpres += elemres.t->intervals;
    }
  }
  if (tmpres != list()) {
     resultado = new TempElement;
     *resultado = tmpres;
  }
  return resultado;
};

function interval (s: string, e: string): Interval;

function body interval
{
 /** Implementacao do operador INTERVAL (s,e) *****/
 o2 Event e_inicio, e_fim;
 o2 Interval intervalo = new Interval;
 e_inicio = event (s);
 e_fim = event (e);
 *intervalo = tuple (t_inicio: e_inicio, t_fim: e_fim);
 return intervalo;
};

function tvg (g: list (tuple (os: list (Obj_Geom), t: TempElement)) ): TempEleme
nt;

function body tvg
{
 o2 list (Interval) tg;
 o2 tuple (os: list (Obj_Geom), t: TempElement) versao;
 o2 TempElement resultado = new TempElement;
 tg = list ();
 for (versao in g) {
    tg += versao.t -> intervals;
 }
 *resultado = tg;
 return resultado;
};

function get_temporal_version (g: list (tuple (os: list (Obj_Geom), t: TempEleme
nt)), t: Time): list (Obj_Geom);

function body get_temporal_version
{
  o2 tuple (os: list (Obj_Geom), t: TempElement) versao;
  o2 list (Obj_Geom) resultado;
  /* Retorna o versao da geometria no tempo "intervalo" */
  for (versao in g) {
    if (versao.t -> t_overlaps (t)) {
      resultado = versao.os;
      break;
    }
  }
  return resultado;
```

```
}
/*****************************************************************************
10  Classes Geograficas e Temporais
******************************************************************************/
class TempObject inherit Object public type
  tuple (t: Time)
method
  tv: Time
end;

class GeoObject inherit Object public
method
   public st_sp (t:Time):list (tuple(os:list(Obj_Geom),t:TempElement)),
   public sp: list (Obj_Geom)
end;

class SpatialObject inherit GeoObject public type
    tuple (l: Sp)
method
    public sp: list (Obj_Geom)
end;

class SpatioTempObject inherit GeoObject, TempObject public type
 tuple (l: SpT)
method
      public valice_sp (t:Time): list (tuple(lst_objs:list(Obj_GeomT),t:Time)),
      public st_sp (t:Time):list (tuple(os:list(Obj_Geom),t:TempElement)),
      public sp: list (Obj_Geom),
      public loc (t: Event): list (Obj_Geom),
      public st_length (t:Time): list (tuple (res: real, t: Time)),
      public st_perimeter (t: Time): list (tuple (res: real, t: Time)),
      public st_area (t: Time): list (tuple (res: real, t: Time)),
      public st_disjoint (obj:GeoObject, t:Time): list (tuple (res: boolean, t:
Time)),
      public st_overlap (obj:GeoObject, t:Time): list (tuple (res: boolean, t: T
ime)),
      public st_distance (obj:GeoObject, t:Time): list (tuple (res: real, t: Tim
e)),
      public st_inside (obj:GeoObject, t:Time): list (tuple (res: boolean, t: Ti
me)),
      public st_touch (obj:GeoObject, t:Time): list (tupla (res: boolean, t: Tim
e))
end;

/*****************************************************************************
10.1  Classes Geograficas e Temporais   - Metodos
******************************************************************************/
method body tv in class TempObject
{
 return self->t;
};

method body sp in class SpatialObject /* Operador SP */
{
  o2 list (Obj_Geom) resultado;
  resultado = list();
  resultado = (self -> l) -> select_geometry;
  return resultado;
};
```



```
method body sp in class SpatioTempObject
{
  o2 list (Obj_Geom) resultado;
  o2 Event e;
  o2 string agora;
  resultado = list();
  agora = now(); /* Retorna uma string correspondendo a data atual */
  e = event (agora); /* Obtem objeto Event correspondente ao parametro */
  resultado = self -> loc (e); /* Retorna localizacao do geo-objeto valida no in
stante e */
  return resultado;
};

method body loc (t: Event): list (Obj_Geom) in class SpatioTempObject
{
 /* Retorna a localizacao de um objeto em um tempo especifico */
 o2 list (Obj_Geom) resultado;
 o2 list (tuple(os:list(Obj_Geom),t: TempElement)) localiz;
 resultado = list ();
 localiz = self -> st_sp (t);
 if (localiz != list()) {
   resultado = localiz[0].os;
 }
 return resultado;
};

method body valice_sp in class SpatioTempObject
{
  o2 list(tuple(lst_objs: list (Obj_GeomT),t:Time)) resultado;
  resultado = list();
  if (self -> t -> t_overlaps (t)) {
    resultado += (self->l) -> t_select_geometry (t);
  }
  return resultado;
};

method body st_sp in class SpatioTempObject
{
 /*** Declaracao de variaveis ******************************/
 o2 list (tuple(lst_objs: list (Obj_GeomT),t:Time)) geometria;
 o2 list (tuple(os:list(Obj_Geom),t: TempElement)) resultado;
 o2 Obj_GeomT objeto = new Obj_GeomT;
 o2 tuple (lst_objs:list(Obj_GeomT),t:Time) versao;
 o2 list (Obj_Geom) lst_objs_versao, lista_objs;
 o2 list (tuple(tempos:list(Event), objs: list(Obj_Geom))) resultin;
 o2 tuple (tempos:list(Event), objs: list(Obj_Geom))componente;
 o2 TempElement tem = new TempElement;
 o2 Event temp = new Event; /* Tempo usado em calculos intermediarios*/
 o2 integer fim,n,numero_de_componentes,iterator;
 o2 boolean found;
/*************************************************************/
if (self->t->t_overlaps (t)) {
 geometria = self -> l -> t_select_geometry (t);
 resultado = list ();
 resultin = list ();
 for (versao in geometria) {
   fim = versao.t -> end -> get_value;
   iterator = versao.t -> begin -> get_value;
   while ( iterator <= fim ) {
```

```
        lst_objs_versao = list();
        for (objeto in versao.lst_objs) {
          o2 Obj_Geom estado = new Obj_Geom;
          *temp = iterator;
          estado = objeto -> valice_s (temp);
          lst_objs_versao += list (estado);
        }
        /*  insert_lobjs_geom (resultin,lst_objs_versao,i); */
        /* se estado ja estiver na lista, so anexar i aos tempos */
        found = false;
        n = 0;
        numero_de_componentes = count (resultin);
        while ((!found) && (n < numero_de_componentes)) {
          lista_objs = resultin[n].objs;
          if (lst_deep_equal (lista_objs,lst_objs_versao)) {
            o2 Event ti = new Event;
            *ti = iterator;
            found = true;
            resultin[n].tempos += list (ti);
          }
          n++;
        }
        if (!found) {
          o2 Event ti = new Event;
          *ti = iterator;
           resultin += list(tuple(tempos:list(ti),objs:lst_objs_versao));
        }
        /* Obtem instante seguinte a "i" */
        iterator++;
        *temp = iterator;
        while (!(versao.t -> t_contains (temp)) && (iterator <= fim)) {
          /* Estado da componentes em "i" nao deve ser calculado se i nao estiver o
ntido na marca de tempo associada 'a versao considerada */
          iterator++;
        }
      }
    }
  for (componente in resultin) {
    o2 TempElement tem = new TempElement;
    tem = Event_to_TempElement (componente.tempos); /* Transforma lista de evento
s em TempElement (porque engloba todos os casos) */
    resultado += list(tuple(os:componente.objs, t:tem));
  }
  return resultado;
}
else return list(); /* Objeto nao existe durante o tempo especificado */
};

method body st_length in class SpatioTempObject /* Operador ST_LENGTH */
{
  o2 list (tuple (res: real, t: Time)) resultado;
  o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geometria;
  o2 Line l = new Line;
  o2 tuple (os: list (Obj_Geom), t: TempElement) versao;
  o2 real compr;

  resultado = list();
  geometria = self -> st_sp (t); /* Lista objetos componentes e tempo associado
*/
  if ( (geometria != list()) && ( (geometria[0].os)[0]->type_of == l->type_of )
```



```
) {
    for (versao in geometria) {
      compr = length ((o2 list (Line)) versao.os);
      resultado += list (tuple (res: compr, t: versao.t));
    }
  }
  /* se resultado = list (), objeto nao existe no tempo especificado */
  return resultado;
};

method body st_perimeter in class SpatioTempObject /* Operador ST_PERIMETER */
{
  o2 list (tuple (res: real, t: Time)) resultado;
  o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geometria;
  o2 Polygon pol = new Polygon;
  o2 tuple (os: list (Obj_Geom), t: TempElement) versao;
  o2 real per;

  resultado = list();
  geometria = self -> st_sp (t); /* Lista objetos componentes e tempo associado
*/
  if ( (geometria != list()) && ( (geometria[0].os)[0]->type_of == pol->type_of
) ){
    for (versao in geometria) {
      per = perimeter ((o2 list (Polygon)) versao.os);
      resultado += list (tuple (res: per, t: versao.t));
    }
  }
  return resultado;
};

method body st_area in class SpatioTempObject /* Operador ST_AREA */
{
  o2 list (tuple (res: real, t: Time)) resultado;
  o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geometria;
  o2 Polygon pol = new Polygon;
  o2 tuple (os: list (Obj_Geom), t: TempElement) versao;
  o2 real ar;

  resultado = list();
  geometria = self -> st_sp (t); /* Lista objetos componentes e tempo associado
*/
  if ( (geometria != list()) && ((geometria[0].os)[0]->type_of == pol->type_of)
) {
    for (versao in geometria) {
      ar = area ((o2 list (Polygon)) versao.os);
      resultado += list (tuple (res: ar, t: versao.t));
    }
  }
  return resultado;
};

method body st_disjoint in class SpatioTempObject /* Operador ST_DISJOINT */
{
  o2 list (tuple (res: boolean, t: Time)) resultado;
  o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geom1;
  o2 Time interseccao;
  o2 list (Interval) intervalos;
  o2 Interval intervalo;
  o2 boolean disj;
```

```
  o2 list (Obj_Geom) g1,g2;
  o2 TempElement tgeom1;

  resultado = list ();
  geom1 = self -> st_sp (t);
  if ( geom1 != list() ) {
    tgeom1 = tvg (geom1);  /* Obtem tv de geom1 */
    if (obj->type_of == self->type_of) {
      o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geom2;
      geom2 = obj -> st_sp (t);
      if ( geom2 != list() ) {
        o2 TempElement tgeom2;
        tgeom2 = tvg (geom2);
        intersecao = tgeom1 -> t_inter (tgeom2);
        intervalos = intersecao -> intervals;
        for (intervalo in intervalos) {
           g1 = get_temporal_version (geom1, intervalo); /* ve qual das listas i
ntersecta o intervalo */
           g2 = get_temporal_version (geom2, intervalo);
           disj = disjoint (g1, g2);
           resultado += list (tuple (res: disj, t: intervalo));
        }
      }
    }
    else {
       o2 tuple (os: list (Obj_Geom), t: TempElement) geom_el;
       g2 = obj -> sp;
       for (geom_el in geom1) {
          disj = disjoint (geom_el.os, g2);
          resultado += list (tuple (res:disj, t:geom_el.t));
       }
    }
  }
  return resultado;
};

method body st_overlap in class SpatioTempObject /* Operador ST_OVERLAP */
{
  o2 list (tuple (res: boolean, t: Time)) resultado;
  o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geom1;
  o2 Time intersecao;
  o2 list (Interval) intervalos;
  o2 Interval intervalo;
  o2 boolean over;
  o2 list (Obj_Geom) g1,g2;
  o2 TempElement tgeom1;

  resultado = list ();
  geom1 = self -> st_sp (t);
  if ( geom1 != list() ) {
    tgeom1 = tvg (geom1);  /* Obtem tv de geom1 */
    if (obj->type_of == self->type_of) {
      o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geom2;
      geom2 = obj -> st_sp (t);
      if ( geom2 != list() ) {
        o2 TempElement tgeom2;
        tgeom2 = tvg (geom2);
        intersecao = tgeom1 -> t_inter (tgeom2);
        intervalos = intersecao -> intervals;
        for (intervalo in intervalos) {
```



```
           g1 = get_temporal_version (geom1, intervalo); /* ve qual das listas i
ntersecta o intervalo */
           g2 = get_temporal_version (geom2, intervalo);
           over = overlap (g1, g2);
           resultado += list (tuple (res: over, t: intervalo));
        }
      }
    }
    else {
       o2 tuple (os: list (Obj_Geom), t: TempElement) geom_el;
       g2 = obj -> sp;
       for (geom_el in geom1) {
          over = overlap (geom_el.os, g2);
          resultado += list (tuple (res:over, t:geom_el.t));
       }
    }
  }
  return resultado;
};

method body st_distance in class SpatioTempObject /* Operador ST_DISTANCE */
{
  o2 list (tuple (res: real, t: Time)) resultado;
  o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geom1;
  o2 Time intersecao;
  o2 list (Interval) intervalos;
  o2 Interval intervalo;
  o2 real dist;
  o2 list (Obj_Geom) g1,g2;
  o2 TempElement tgeom1;

  resultado = list ();
  geom1 = self -> st_sp (t);
  if ( geom1 != list() ) {
    tgeom1 = tvg (geom1);  /* Obtem tv de geom1 */
    if (obj->type_of == self->type_of) {
      o2 list (tuple (os: list (Obj_Geom), t: TempElement)) geom2;
      geom2 = obj -> st_sp (t);
      if ( geom2 != list() ) {
        o2 TempElement tgeom2;
        tgeom2 = tvg (geom2);
        intersecao = tgeom1 -> t_inter (tgeom2);
        intervalos = intersecao -> intervals;
        for (intervalo in intervalos) {
           g1 = get_temporal_version (geom1, intervalo); /* ve qual das listas i
ntersecta o intervalo */
           g2 = get_temporal_version (geom2, intervalo);
           dist = distance (g1, g2);
           resultado += list (tuple (res: dist, t: intervalo));
        }
      }
    }
    else {
       o2 tuple (os: list (Obj_Geom), t: TempElement) geom_el;
       g2 = obj -> sp;
       for (geom_el in geom1) {
          dist = distance (geom_el.os, g2);
          resultado += list (tuple (res:dist, t:geom_el.t));
       }
    }
```

```
  }
  return resultado;

 };

/**************** Incompletas************************************/
method body st_inside in class SpatioTempObject /* Operador ST_IN */
 {
  o2 list (tuple (res: boolean, t: Time)) resultado;
  o2 list (tuple (cs: list (Obj_Geom), t: TempElement)) geoml;
  o2 Time interseoao;
  o2 list (Interval) intervalos;
  o2 Interval intervalo;
  o2 boolean ins;
  o2 Point pt = new Point;
  o2 Polygon pol = new Polygon;
  o2 list (Obj_Geom) g1,g2;
  o2 TempElement tgeoml;

  resultado = list ();
  geoml = self -> st_sp (t);
  if ( (geoml != list()) && ( ((geoml[0].cs)[0]->type_of == pt->type_of) || ( (g
eoml[0].cs)[0]->type_of == pol->type_of) ) ) {
    tgeoml = tvg (geoml);  /* Obtem tv de geoml */
    if (obj->type_of == self->type_of) {
      o2 list (tuple (cs: list (Obj_Geom), t: TempElement)) geom2;
      geom2 = obj -> st_sp (t);
      if ( geom2 != list()) {
       if ((geom2[0].cs)[0]->type_of == pol->type_of )  {
        o2 TempElement tgeom2;
        tgeom2 = tvg (geom2);
        interseoao = tgeoml -> t_inter (tgeom2);
        intervalos = interseoao -> intervals;
        for (intervalo in intervalos) {
           g1 = get_temporal_version (geoml, intervalo); /* ve qual das listas i
ntersecta o intervalo */
           g2 = get_temporal_version (geom2, intervalo);
           ins  = inside (g1, g2);
           resultado += list (tuple (res: ins, t: intervalo));
        }
       }
       else display("Operation not implemented");
      }
    }
    else {
      o2 tuple (cs: list (Obj_Geom), t: TempElement) geom_el;
      g2 = obj -> sp;
      if (g2 != list ()) {
       if (g2[0]->type_of == pol->type_of) {
         for (geom_el in geoml) {
          ins  = inside (geom_el.cs, g2);
          resultado += list (tuple (res:ins, t:geom_el.t));
         }
       }
       else  {
         display("Operation not implemented");
         return resultado;
       }
      }
    }
```



```
  }
  else {
     if (geoml != list ()) {
      display("Operation not implemented");
      return resultado;
     }
  }
  return resultado;
};

method body st_touch in class SpatioTempObject /* Operador ST_TOUCH */
 {
  o2 list (tuple (res: boolean, t: Time)) resultado;
  o2 list (tuple (cs: list (Obj_Geom), t: TempElement)) geoml;
  o2 Time interseoao;
  o2 list (Interval) intervalos;
  o2 Interval intervalo;
  o2 boolean tou;
  o2 Polygon pol = new Polygon;
  o2 list (Obj_Geom) g1,g2;
  o2 TempElement tgeoml;

  resultado = list ();
  geoml = self -> st_sp (t);
  if ( (geoml != list()) && ( (geoml[0].cs)[0]->type_of == pol->type_of ) ) {
    tgeoml = tvg (geoml);  /* Obtem tv de geoml */
    if (obj->type_of == self->type_of) {
      o2 list (tuple (cs: list (Obj_Geom), t: TempElement)) geom2;
      geom2 = obj -> st_sp (t);
      if ( geom2 != list() ) {
       if ( (geom2[0].cs)[0]->type_of == pol->type_of ) {
        o2 TempElement tgeom2;
        tgeom2 = tvg (geom2);
        interseoao = tgeoml -> t_inter (tgeom2);
        intervalos = interseoao -> intervals;
        for (intervalo in intervalos) {
           g1 = get_temporal_version (geoml, intervalo); /* ve qual das listas i
ntersecta o intervalo */
           g2 = get_temporal_version (geom2, intervalo);
           tou  = touch (g1, g2);
           resultado += list (tuple (res: tou, t: intervalo));
        }
       }
       else display("Operation not implemented");
      }
    }
    else {
      o2 tuple (cs: list (Obj_Geom), t: TempElement) geom_el;
      g2 = obj -> sp;
      if  (g2 != list ()) {
       if (g2[0]->type_of == pol->type_of)  {
        for (geom_el in geoml) {
          tou = touch (geom_el.cs, g2);
          resultado += list (tuple (res:tou, t:geom_el.t));
        }
       }
       else {
         display("Operation not implemented");
         return resultado;
```

```
        }
      }
    }
  }
  else {
    if (geoml != list ()) {
      display('Operation not implemented');
      return resultado;
    }
  }
  return resultado;
};
```

```
application Consultas
program
  public Con_a_1,
  public Con_a_1_a,
  public Con_a_2,
  public Con_a_3,
  public Con_a_4,
  public Con_a_4_1,
  public Con_a_4_2,
  public Con_b_1,
  public Con_b_2,
  public Con_b_2_1,
  public Con_b_3,
  public Con_b_4,
  public Con_c_1,
  public Con_c_2,
  public Con_c_3,
  public Con_c_3_a,
  public Con_d_2,
  public Con_d_3,
  public Con_d_4
end application;

/*********************************************************************/
program body Con_a_1 in application Consultas {
  o2 unique set (Divisao_Agricola) resultado;
  o2 string data;
  o2 string nome;
  nome = "cana";
  data = "1/7/1997";
  display ("Quais as divisoes agricolas ocupadas por plantacoes de cana antes de
 01/07/1997?");
  o2query (resultado, "select distinct d \
                        from d in Divisoes, \
                             e in d.historico_cultura \
                        where e.cultura = $1 and e.t->begin->t_before (event($2))
", nome,data);
  display (resultado);
};
/*********************************************************************/
program body Con_a_1_a in application Consultas {
  o2 unique set (Divisao_Agricola) resultado;
  o2 string data;
  o2 string nome;
  nome = "cana";
  data = "1/12/1999";
  display ("Quais as divisoes agricolas ocupadas por plantacoes de cana depois d
e 01/12/1999?");

  o2query (resultado, "select distinct d \
                        from d in Divisoes, \
                             e in d.historico_cultura \
                        where e.cultura = $1 and event($2)->t_before(e.t->end)",
nome,data);
  display (resultado);
};

/*********************************************************************/
program body Con_a_2 in application Consultas
{
```



```
 o2 set(tuple(lst_divisoes:list (Divisao_Agricola),t:Time)) q1;
 o2 set(tuple(divisao:Divisao_Agricola,producao:real,t:Time)) q2;
 o2 set(tuple(divisao:Divisao_Agricola,cultura:string,t:Time)) q3;
 o2 set(tuple(divisao:Divisao_Agricola,t:Time, cultura:string, producao:real)) q
4;
 o2 string nome, data1, data2;

 display ("Qual a producao (toneladas/produto) da fazenda D entre 05/12/1998 e 0
1/12/1999?");
 nome = "D";


 data1 = "05/12/1998";
 data2 = "01/12/1999";
 /* seleciona as listas de divisoes e tempo valido's no intervalo especificado */

 o2query(q1,"select tuple (lst_divisoes:e.lst_divisoes, t:e.t->t_inter(interval(
$2,$3))) \
              from f in Fazendas, \
                   e in f.historico_divisoes \
              where f.nome = $1 and e.t->t_overlaps(interval($2,$3))",nome,data1,
data2);

 o2query(q2,"select tuple(divisao:d,producao:c.producao,t:c.t->t_inter(l.t)) \
              from l in $1, \
                   d in l.lst_divisoes, \
                   c in d.historico_producao \
              where c.t->t_overlaps (l.t)",q1);

 o2query(q3,"select tuple(divisao:d,cultura:c.cultura,t:c.t->t_inter(l.t)) \
              from l in $1, \
                   d in l.lst_divisoes, \
                   c in d.historico_cultura \
              where c.t->t_overlaps (l.t)",q1);

 o2query(q4,"select tuple(divisao:c.divisao,t:c.t->t_inter(p.t),cultura:c.cultur
a,producao:p.producao)\
              from c in $1,\
                   p in $2 \
              where c.t->t_overlaps(p.t) and c.divisao = p.divisao",q3,q2);

 display (q4);
};
/*********************************************************************/
program body Con_a_3 in application Consultas {
 o2 set (tuple(lst_divisoes:list(Divisao_Agricola), t:Time)) q1;
 o2 set (tuple(divisao: Divisao_Agricola,cultura:string,t:Time)) q2;
 o2 string nome, data1, data2;

 display ("O que foi cultivado na fazenda A entre 01/01/1997 e 01/01/1998?");
 nome = "A";

 data1 = "01/01/1997";
 data2 = "01/01/1998";
 /* seleciona as listas de divisoes e tempo validos no intervalo especificado */

 o2query(q1,"select tuple (lst_divisoes:e.lst_divisoes, t:e.t->t_inter(interval(
$2,$3))) \
              from f in Fazendas, \
                   e in f.historico_divisoes \
```

```
             where f.nome = $1 and e.t->t_overlaps(interval($2,$3))",nome,data1,
data2);

 display (q1);

 o2query(q2,"select tuple(divisao:d, cultura:c.cultura,t:c.t->t_inter(l.t)) \
             from l in $1, \
                  d in l.lst_divisoes, \
                  c in d.historico_cultura \
             where c.t->t_overlaps (l.t)",q1);

 display (q2);
};
/**************************************************************************/
program body Con_a_4 in application Consultas {
  o2 set (Divisao_Agricola) q1;
  o2 unique set (Divisao_Agricola) q2;
  o2 set (tuple(Divisao1: Divisao_Agricola, Divisao2: Divisao_Agricola)) q3;
  o2 string data, cultura;
  data = "04/07/1999";
  cultura = "cana";
  display("Selecione as plantacoes de cana adjacentes em 04/07/1999"); /* ver tb
 com 01/02/97 */
  o2query (q1,"select d \
               from d in Divisoes \
               where d->tv->t_overlaps(event($1))",data);
  display (q1);

  o2query (q2,"select distinct d \
               from d in $3, \
                    c in d.historico_cultura \
               where d.historico_cultura != list() and c.t->t_overlaps (event($1
)) and c.cultura = $2",data,cultura,q1);
  display (q2);

  o2query (q3, "select tuple (Divisao1: d1, Divisao2: d2) \
                from d1 in $2, \
                     d2 in $2 \
                where d1 != d2 and touch (d1->loc(event($1)), d2->loc(event($1))
)", data,q2);
  display (q3);
};

/**************************************************************************/
program body Con_a_4_1 in application Consultas {
  o2 set (tuple(Divisao1: Divisao_Agricola, Divisao2: Divisao_Agricola)) q1;
  o2 string data1, data2;
  data1 = "01/10/1999";
  data2 = "1/12/99";
  display("Selecione as divisoes agricolas adjacentes alguma vez entre 01/10/99
e 1/12/99");
  o2query (q1, "select tuple(Divisao1:d1, Divisao2:d2) \
                from d1 in Divisoes, \
                     d2 in Divisoes \
                where d1 != d2 and is_s_true (d1->st_touch(d2,interval($1,$2)))"
,data1,data2);
  display (q1);
};

/**************************************************************************/
```



```
program body Con_a_4_2 in application Consultas {
  o2 set(list (tuple(res:boolean, t: Time))) q1;
  o2 string data1, data2,nome1,nome2;
  data1 = "03/08/1997";
  data2 = Now();
  nome1 = "DIV1";
  nome2 = "DIV2";
  /* usada para testar operadores espaco-temporais */
  o2query (q1, "select d1->st_disjoint(d2,interval($1,$2)) \
                from d1 in Divisoes, \
                     d2 in Divisoes \
                where d1.nome = $3 and d2.nome = $4",data1,data2,nome1,nome2);
  display (q1);
};

/**************************************************************************/
program body Con_b_1 in application Consultas {
   o2 Fazenda q1;
  o2 list (Obj_Geom) q2;
  o2 string nome;
  display ("Qual a localizacao da fazenda A?");
  nome = "A";
  o2query(q1,"element(select f \
                      from f in Fazendas \
                      where f.nome = $1)",nome);

  o2query (q2, "$1->sp",q1);
  display (q2);
};

/* Opcao:

select f->sp
from f in Fazendas
where f.nome = "A" */

/**************************************************************************/
program body Con_b_2 in application Consultas {
  o2 set (Fazenda) resultado;
  o2 string nome;
  display ("Selecione as fazendas distantes ate 10km da estrada E3");
  nome = "E3";
  o2query (resultado, "select f from f in Fazendas, e in Estradas where e.nome =
 $1 and le(f->st_distance (e,event(now())),10.0)", nome);
  display (resultado);
};

/**************************************************************************/

program body Con_b_2_1 in application Consultas {
  o2 real resultado;
  o2 string nome1, nome2,data;
  o2 Fazenda q1;
  o2 Estrada q2;
  display ("Qual a distancia entre a fazenda D e a estrada E1 em 01/07/97?");
  nome2 = "E1";
  nome1 = "D";
  data = "01/07/97";
  o2query (q1,"element(select f \
                       from f in Fazendas \
```

```
                        where f.nome = $1)",nome1);
  display (q1);

  o2query (q2,"element(select e \
                        from e in Estradas \
                        where e.nome = $1)",nome2);

  o2query (resultado, "distance($1->loc(event($3)), $2->loc(event($3)))",q1,q2,d
ata);
  display (q2);
  display (resultado);
};

/***********************************************************************/
program body Con_b_3 in application Consultas {
  o2 unique set (Fazenda) resultado;
  o2 string tipo;
  display("Selecione as fazendas que contem postes de luz");
  tipo = "luz";
  o2query (resultado, "select distinct f \
                        from f in Fazendas, \
                             p in Postes \
                        where p.tipo = $1 and inside (p->sp, f->sp)",tipo);
  display (resultado);
};
/* Opcao:
select f
from f in Fazendas,
     p in Postes
where exists p in Postes: p.tipo = "luz" and inside (p->sp, f->sp) */
/***********************************************************************/
program body Con_b_4 in application Consultas {
  o2 set (Fazenda) resultado;
  o2 string data;
  display("Quais fazendas existiam em 07/09/1997");
  data = "07/09/1997";
  o2query (resultado, "select f \
                        from f in Fazendas \
                        where f->tv->t_overlaps (event($1))", data);
  display (resultado);
};
/***********************************************************************/
program body Con_c_1 in application Consultas {
 o2 string nome, data1, data2;
 o2 set (list (tuple(os: list(Obj_Geom),t:TempElement))) q1;
 display ("Qual a evolucao do espaco ocupado pela fazenda A entre 1/5/1997 e 1/3
/1998?");
 nome = "A";
 data1 = "1/5/1997";
 data2 = "1/3/1998";
 o2query(q1,"select f->st_sp (interval($2,$3))\
             from f in Fazendas \
             where f.nome = $1",nome,data1,data2);
 display(q1);
};
/***********************************************************************/
program body Con_c_2 in application Consultas
{
 o2 set(tuple(fazenda:Fazenda, area: list(tuple(res:real,t:Time)))) q1;
 o2 string data1, data2;
```



```
 display("Qual a área ocupada por fazendas entre 01/01/97 e 01/01/98?");
 data1 = "01/01/97";
 data2 = "01/01/98";
 o2query(q1, "select tuple(fazenda: f, area: f->st_area(interval($1,$2)))\
             from f in Fazendas",data1,data2);
 display(q1);
};
/***********************************************************************/
program body Con_c_3 in application Consultas
{
 o2 Fazenda q1;
 o2 SpatialObject q2;
 o2 list(tuple(res:boolean,t:Time)) q3;
 o2 TempElement q4;
 o2 string nome = "A";
 display("Quando a fazenda A esteve incluida no retangulo delimitado pelas coord
enadas 1,1,15,40?");
 o2query(q1,"element(select f \
                      from f in Fazendas \
                      where f.nome = $1)",nome);
 display (q1);

 o2query(q2, "to_obj(list(tuple(x:1.0,y:1.0),tuple(x:15.0,y:1.0),tuple(x:15.0,y:
40.0),tuple(x:1.0,y:40.0)))");

 display (q2);

 o2query(q3,"$1->st_inside ($2,interval(Beginning,Now()))",q1,q2);

 display (q3);

 o2query(q4,"twhen($1)",q3);
 display(q4);
};
/***********************************************************************/
program body Con_c_3_a in application Consultas
{
 o2 Fazenda q1;
 o2 SpatialObject q2;
 o2 list(tuple(res:boolean,t:Time)) q3;
 o2 TempElement q4;
 o2 string nome = "A";
 display("Quando a fazenda A esteve incluida no retangulo delimitado pelas coord
enadas 1,3,24,40?");
 o2query(q1,"element(select f \
                      from f in Fazendas \
                      where f.nome = $1)",nome);
 display (q1);

 o2query(q2, "to_obj(list(tuple(x:1.0,y:3.0),tuple(x:24.0,y:3.0),tuple(x:24.0,y:
40.0),tuple(x:1.0,y:40.0)))");

 display (q2);

 o2query(q3,"$1->st_inside ($2,interval(Beginning,Now()))",q1,q2);

 display (q3);

 o2query(q4,"twhen($1)",q3);
 display(q4);
```

```
};

/*************************************************************************/
program body Con_d_2 in application Consultas
{
 o2 string nome, data;
 o2 Fazenda q1;
 o2 set (Estrada) q2;
 nome = "A";
 data = "01/07/97";
 display("Que estradas intersectariam a fazenda A se ela tivesse hoje o formato
que tinha em 01/07/1997?");

 /* ver com varias combinacoes de datas e fazendas diferentes */

 o2query(q1,"element(select f \
             from f in Fazendas \
             where f.nome = $1)",nome,data);

 o2query(q2,"select e \
             from e in Estradas \
             where not disjoint (e->sp, $1->loc(event($2)))",q1,data);
 display(q2);
};
/*************************************************************************/
program body Con_d_3 in application Consultas
{
 o2 unique set (Fazenda) q1;
 o2 set (tuple(fazenda: Fazenda, localizacao: list (Obj_Geom))) q2;
 o2 string cultura, data1, data2;
 display(" Apresente a localização atual das fazendas que cultivam cana e que ti
veram área maior que 1000 entre 01/01/1997 e 31/12/1998.");

/* Seleciona as fazendas que cultivam cana */
 cultura = "cana";
 o2query(q1,"select distinct f \
             from f in Fazendas, \
                  e in f.historico_divisoes, \
                  d in e.lst_divisoes, \
                  c in d.historico_cultura \
             where e.t->t_overlaps (event(now())) and c.t->t_overlaps (event(now
())) and c.cultura = $1",cultura);
 display (q1);

 /* Seleciona as fazendas com area maior que 1000 entre "01/01/1997" e "31/12/19
98".*/
 data1 = "01/01/1997";
 data2 = "31/12/1998";
 o2query(q2,"select tuple (fazenda: f, localizacao: f->sp) \
             from f in $1 \
             where gt(f->st_area(interval($2,$3)),1000.0)",q1,data1,data2);

 display(q2);
};

/*************************************************************************/
program body Con_d_4 in application Consultas
{
 o2 set (tuple (poste: Poste, perimetro: list(tuple(res:real,t:Time)))) q2;
 o2 string data1, data2;
```



```
 data1 = "01/01/97";
 data2 = "01/01/98";
 o2query(q2, "select tuple(poste: p, perimetro: p->st_perimeter(interval($1,$2))
)\
             from p in Postes",data1,data2);
 display(q2);
};
```