UNIVERSIDADE FEDERAL FLUMINENSE

GLAUCO FIOROTT AMORIM

LEIAUTES DINÂMICOS BASEADOS EM TEMPLATES PARA DOCUMENTOS MULTIMÍDIA

NITERÓI

2017

UNIVERSIDADE FEDERAL FLUMINENSE

GLAUCO FIOROTT AMORIM

LEIAUTES DINÂMICOS BASEADOS EM TEMPLATES PARA DOCUMENTOS MULTIMÍDIA

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Sistemas de Computação

Orientadora:
DÉBORA CHRISTINA MUCHALUAT SAADE

NITERÓI

2017

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

A524 Amorim, Glauco Fiorott
Leiautes dinâmicos baseados em templates para documentos multimídia / Glauco Fiorott Amorim. – Niterói, RJ : [s.n.], 2017.
131 f.

Tese (Doutorado em Computação) - Universidade Federal Fluminense, 2017.
Orientador: Débora Christina Muchaluat Saade.

1. Aplicação web. 2. Leiaute. 3. Multimídia interativa. 4. NCL (Linguagem de marcação de documento). I. Título.

CDD 005.3

GLAUCO FIOROTT AMORIM

LEIAUTES DINÂMICOS PARA DOCUMENTOS MULTIMÍDIA BASEADOS EM TEMPLATES

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Sistemas de Computação.

Aprovada em agosto de 2017.

BANCA EXAMINADORA

Profª. Débora Christina Muchaluat Saade, D.Sc.
Orientadora, Universidade Federal Fluminense (UFF)

Prof. Christiano de Oliveira Braga, D.Sc.
Universidade Federal Fluminense (UFF)

Prof. Esteban Walter Gonzalez Clua, D.Sc.
Universidade Federal Fluminense (UFF)

Profª. Claudia Maria Lima Werner, D.Sc.
Universidade Federal do Rio de Janeiro (UFRJ)

Profª. Simone Diniz Junqueira Barbosa, D.Sc.
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

Dedicatória(s): À minha família, em especial, à minha esposa Myrna e ao meu filho Miguel que abdicaram de várias horas de convívio para que eu pudesse chegar ao final dessa jornada. Sempre estiveram do meu lado, me apoiando nos momentos em que duvidei que era possível. Amo vocês do fundo do meu coração.

Agradecimentos

Não poderia deixar de agradecer à Profª. Débora Christina Muchaluat-Saade, minha orientadora e mentora. Sempre dedicada e acolhedora, fez do laboratório MídiaCom nossa segunda casa. Possui uma liderança que não se faz por imposição e sim por exemplos. Mestre na essência da palavra, tinha sempre um incentivo para dar mesmo quando não tínhamos avançado tanto. Foi uma honra poder trabalhar junto com você, minha querida Professora. Aproveito para estender esse agradecimento ao Profº Luiz Fernando Gomes Soares, que nos deixou prematuramente. Tive o privilégio de conviver, mesmo que por pouco tempo, com sua ilustre presença. Muito obrigado pelas valiosas dicas na qualificação, pelos ensinamentos transmitidos por intermédio da Profª Débora e pelas visitas ao laboratório Telemídia onde fiz vários amigos.

Essa tese não seria essa tese se não fosse pelo grande amigo Joel. Não bastasse me ajudar com conselhos, discussões sobre os temas, artigos escritos em conjunto, horas no telefone falando sobre regiões daqui e leiautes de lá, ainda arranjou tempo para fazer uma cerveja para celebrarmos. Muito obrigado meu amigo. Tenho certeza que construiremos outras coisas juntos, mas nunca me esquecerei do seu grande suporte nessa batalha.

A família MídiaCom nunca pode ser esquecida. Obrigado a todos os amigos que ajudaram, de alguma forma, direta ou indiretamente, nesse trabalho. Nunca esquecerei de nossas festas juninas, churrascos e encontros. Estaremos sempre juntos. Abro aqui um parênteses para nossa querida Marister. Se temos esse sentimento de família devemos um pouco disso a ela. Uma mãezona, sempre com um olhar materno e um conselho certeiro.

Obrigado a todos os professores que fizeram parte dessa caminhada. Aprendi muito com todos vocês.

Obrigado aos amigos do Cefet-RJ. Obrigado pelo apoio incondicional, pela troca de experiências e pelos ensinamentos.

"Nas grandes batalhas da vida, o primeiro passo para a vitória é o desejo de vencer."
(Mahatma Ghandhi)

Resumo

Visando melhorar a experiência do usuário, independentemente do dispositivo utilizado, aplicações multimídia interativas podem oferecer leiautes autoajustáveis que são modificados dependendo das condições de visualização apresentadas. Algumas linguagens de autoria, como HTML5 e CSS3, permitem que o autor do documento declare diretivas para que o leiaute da aplicação responda a mudanças no contexto do usuário. Entretanto, essas diretivas têm expressividade limitada oferecendo relações muito básicas como, por exemplo, a que especifica que o tamanho do objeto A deve ser igual ao tamanho de B. Outras linguagens para autoria multimídia, como NCL e SMIL, não fornecem facilidades para que o autor do documento possa desenvolver leiautes adaptáveis, obrigando o autor a definir detalhadamente a região em que cada objeto de mídia visual ocupará na tela de cada dispositivo, o que torna a autoria bastante trabalhosa.

Esta tese propõe STyLe, uma linguagem de *templates* baseada em XML e que utiliza restrições para especificar leiautes espaciais adaptativos e dinâmicos para aplicações multimídia. Um *leiaute espacial adaptativo* é uma abordagem que possibilita que autores possam criar características de apresentação genéricas que adaptam o leiaute espacial especificado ao número de objetos de mídia de um dado documento, diminuindo o esforço de autoria na criação de leiautes espaciais. Um *leiaute espacial adaptativo e dinâmico* é uma extensão de um leiaute adaptativo de tal forma que as características de apresentação dos objetos de mídia possam ser alteradas em tempo de execução e em resposta a ocorrência de eventos na apresentação do documento, tais como interação do usuário ou edição ao vivo de partes do documento. STyLe fornece esta facilidade definindo o leiaute espacial de um documento hipermídia através de restrições espaciais.

Como prova de conceito, a linguagem de autoria multimídia NCL e a linguagem de autoria de *templates XTemplate* foram estendidas para utilizar STyLe na concepção dos leiautes espaciais dos documentos. Além disso, uma arquitetura capaz de interpretar as restrições espaciais contidas em um *template* e construir ou atualizar corretamente o leiaute espacial de um documento é proposta e implementada para o uso de STyLe em conjunto com NCL.

Esta tese também avalia o desempenho da solução apresentada para adaptar dinamicamente o leiaute espacial em tempo de execução de um documento NCL, medindo o tempo de resposta para adaptação de leiautes em diferentes exemplos de uso de STyLe.

Palavras-chave: Leiautes Dinâmicos, Leiautes Adaptativos, STyLe, NCL, Restrições espaciais, XTemplate, Autoria de Templates.

Abstract

Focusing on improving the user experience regardless of the device used, interactive multimedia applications should provide self-tuning layouts. Some authoring languages as HTML5 and CSS3 allow the author to declare policies so that the application's layout can adapt itself according to changes in the application context. However, those polices have limited expressiveness providing very basic relations such as the size of objects A and B must be equal. Other languages used for multimedia authoring, such as NCL and SMIL, do not provide native facilities for the document author to develop adaptive layouts. So, authors must define in details where each media object will be presented in a device screen, making authoring very hard.

This thesis proposes STyLe, an XML-based template language that provides constraints for specifying adaptive and dynamic spatial layouts. An adaptive layout allows authors to create generic presentation characteristics that adapt a document presentation layout depending on the number of media objects it contains, reducing the authoring effort to specify spatial layouts. An adaptive and dynamic spatial layout is an extension of an adaptive layout in such a way that the presentation characteristics of the media objects can be changed at runtime in response to event occurrences in document presentation, such as user interaction or live edition. STyLe provides this facility by defining the spatial layout of a hypermedia document through spatial constraints.

As proof of concept, the NCL multimedia authoring language and the *XTemplate* template authoring language have been extended to use STyLe for the design of document spatial layouts. In addition, an architecture capable of interpreting this language and performing the necessary changes in order to dynamically update NCL media object presentation characteristics at runtime is proposed and implemented.

This thesis also evaluates the performance of the proposed solution for dynamically adapting layout during NCL document execution measuring the response time for layout adaptation in different STyLe template examples.

Palavras-chave: Dynamic Spatial Layout, Adaptive Layouts, STyLe, NCL, Spatial Constraints, XTemplate, Template Authoring.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

BBC *British Broadcasting Company* 98

CCSS *Constraint Cascading Style Sheets* 4

CSS *Cascading Style Sheets* 4

CSS3 *Cascading Style Sheets - versão 3* 4

CSVG *Constraint Scalable Vector Graphics* 23

DSL *Domain Specific Language* 7

EPG *Eletronic Program Guide* 21

GSS *Grid Style Sheets* 20

HTML *HyperText Markup Language* 19

HTML5 *HyperText Markup Language - versão 5* 4

NCL *Nested Context Language* 4

RWD *Responsive Web Design* 20

SMIL *Synchronized Multimedia Integration Language* 4

STAMP *Synchronized Template for Adaptable Multimedia Presentation* 15

STyLe *Spatial Template Language* 7

SVG *Scalable Vector Graphics* 23

TAL *Template Authoring Language* 6

XHTML *eXtensible Hypertext Markup Language* 12

XML *eXtensible Markup Language* 7

Sumário

Capítulo 1

Introdução

A forma como interagimos com os programas computacionais, sejam eles utilitários ou aplicações do usuário, têm sofrido mudanças ao longo do tempo. Nos primeiros sistemas, a interação se dava exclusivamente por linha de comando, mas atualmente existem várias possibilidades que vão desde interação através do mouse até interação por toque, gestos ou comandos de voz.

Essas mudanças impulsionaram a evolução das interfaces de comunicação entre a aplicação e o usuário. Hoje, o leiaute de uma aplicação é tão importante quanto seu conteúdo, tanto que as ferramentas desenvolvidas para facilitar a autoria das aplicações possuem elementos específicos para definição dos leiautes. Por exemplo, o *kit* de desenvolvimento *Android Studio* [1] utilizado para criação de aplicações móveis em aparelhos que executam o sistema operacional *Android* possui uma visão separada com diversas facilidades para o desenvolvimento do leiaute da aplicação.

Aplicações interativas multimídia como aplicações hipermídia, *slides* interativos, videoaulas, etc, têm seguido o mesmo caminho e enriqueceram a forma como o conteúdo é apresentado para o usuário. Isso tem sido feito para tentar melhorar cada vez mais a experiência do indivíduo ao interagir com a aplicação.

Aplicações hipermídia tem um papel fundamental na evolução da interface de comunicação com o usuário. Atualmente, é possível identificar aplicações hipermídia personalizadas. De acordo com [2], *personalização*, também chamada de *customização* ou *adaptação*, é o processo que consiste em mudar o comportamento da aplicação baseado no conhecimento que a aplicação tem do usuário ou da plataforma em que a aplicação executa. Aplicações que possuem essas características são chamadas de aplicações adaptativas. A

adaptação pode ocorrer em vários aspectos como navegação entre os elementos, conteúdo apresentado e visualização dos elementos na tela do exibidor.

Como aplicações multimídia interativas podem ser executadas em vários dispositivos com diferentes tipos de telas, por exemplo: TVs, *tablets* e *smartphones*, é interessante que o leiaute dessas aplicações seja adaptável para que a experiência dos usuários ao usar a aplicação possa ser tão boa quanto possível, independentemente do dispositivo usado. Um leiaute adaptativo é, portanto, capaz de se modificar dependendo das características de apresentação do dispositivo ou do contexto onde está sendo apresentado, como mostrado na Figura 1.1.

Figura 1.1: Exemplo de leiaute adaptativo

Como mencionado anteriormente, é possível estabelecer alguns aspectos como principais fatores para adaptação da aplicação, como por exemplo, características de apresentação da aplicação (posicionamento e tamanho) ou a quantidade de objetos de mídia que deverão ser apresentados.

A adaptação pode ser realizada tanto em tempo de autoria quanto em tempo de execução. Um exemplo de adaptação em tempo de autoria pode ser visualizado quando o autor não sabe antecipadamente quantos objetos de mídia participarão da aplicação. Isso pode acontecer quando um *template* é utilizado. Nesse caso, há a necessidade de se criar elementos que possam se adaptar para que todos os objetos de mídia possam ser apresentados. Quando a adaptação é realizada em tempo de execução, as alterações ocorrem enquanto a aplicação é apresentada, como por exemplo, alterar a posição de um objeto de mídia porque outro objeto foi inserido na aplicação.

O trabalho proposto nesta tese aborda a adaptação do leiaute espacial em tempo de autoria e em tempo de execução. Este trabalho define leiaute espacial adaptativo como aquele que possibilita ao autor criar características de apresentação genéricas que adaptam

o leiaute espacial ao número de objetos de mídia em um dado documento. Isso reduz o esforço de autoria na criação do leiaute espacial das aplicações, especialmente, quando os documentos possuem um grande número de componentes visuais [3].

A ideia é permitir que os leiautes das aplicações sejam definidos de maneira genérica, através de *templates*, utilizando modelos predefinidos. Os modelos deverão especificar comportamentos para as características de apresentação dos objetos de mídia que os referenciam, como: regiões onde serão apresentados, navegação entre os objetos e parâmetros como transparência do objeto, por exemplo.

Nesta tese, define-se *Leiaute Adaptativo* como uma abordagem que possibilita aos autores criarem características de apresentação genéricas que se adaptam ao número de objetos de mídia em um dado documento, diminuindo, portanto, o esforço de autoria em relação à especificação das características de apresentação dos objetos que serão mostrados na tela do dispositivo [4]. Essa diminuição é explicada porque o autor não precisa descrever o código completo. O documento que utiliza um *template* pode ser considerado um documento com lacunas a serem preenchidas. Basta que o autor as preencha para ter um documento completo.

De acordo com essa definição, um *Leiaute Adaptativo* não é necessariamente um leiaute dinâmico, isto é, um leiaute que se ajusta automaticamente quando um evento ocorre ou quando as características de apresentação se alteram. Um leiaute adaptativo é definido através de um *template* de leiaute, que pode se basear em diversos modelos de leiaute distintos.

Além de poder ser adaptativo, um leiaute também pode ser dinâmico. Este trabalho define um leiaute espacial adaptativo e dinâmico como uma extensão de um leiaute adaptativo onde características de apresentação dos objetos de mídia podem ser alteradas dinamicamente, em tempo de execução, em resposta a ocorrência de eventos na apresentação do documento, como por exemplo, o início ou o fim da apresentação de objetos de mídia, interação do usuário ou edição ao vivo do documento em exibição. Nesta tese, edição ao vivo de um documento multimídia refere-se a alterações na estrutura do documento, como por exemplo, adição ou remoção de objetos de mídia enquanto o documento multimídia estiver sendo apresentado pelo dispositivo de exibição do usuário.

1.1 Motivação

Algumas linguagens de autoria multimídia, como *HyperText Markup Language - versão 5* (HTML5) [5] em conjunto com *Cascading Style Sheets - versão 3* (CSS3) [6], permitem que o autor declare políticas para que o leiaute das aplicações possa se adaptar de acordo com mudanças no contexto da aplicação (características ou preferências do dispositivo). Embora essas linguagens apresentem elementos que facilitem a autoria na declaração de leiautes dinâmicos, as relações espaciais criadas entre esses elementos são básicas como, por exemplo, o tamanho de A é igual ao tamanho de B.

Para solucionar as limitações do *Cascading Style Sheets* (CSS), alguns trabalhos, tais como [7] e [8], estenderam a linguagem CSS introduzindo um sistema de restrições com novos elementos e facilidades para criar leiautes adaptativos. Esse sistema é chamado de *Constraint Cascading Style Sheets* (CCSS). Embora as contribuições do trabalho sejam significativas, as restrições ainda criam relações espaciais básicas, tomadas sempre duas a duas, como o topo da região X é igual ao topo da região Y. Em ambos os trabalhos [7] e [8], não existe uma forma de definir o leiaute usando restrições de mais alto nível como alinhamentos ou distribuições frequentemente encontradas em ferramentas de edição de apresentações ou edição de imagem.

Outras linguagens usadas para autoria multimídia, como *Nested Context Language* (NCL) [9, 10] e *Synchronized Multimedia Integration Language* (SMIL) [11], fornecem um conjunto de elementos que facilitam a sincronização espaço-temporal entre objetos de mídia, além de apresentarem uma separação clara entre o conteúdo e estrutura de um documento.

Por exemplo, em NCL e SMIL, para cada objeto de mídia exibido visualmente, é necessário definir as características de apresentação como coordenadas espaciais (x, y, z) e o tamanho de uma região que o objeto irá ocupar na tela do dispositivo. Como essa definição é feita individualmente para cada região ou componente visual de um documento multimídia, o processo pode se tornar custoso em termos de autoria. A Figura 1.2 apresenta um exemplo de um documento multimídia e suas regiões em uma TV.

Em aplicações mais elaboradas, a quantidade de objetos e a necessidade de se definir relacionamentos temporais e espaciais entre eles aumentam consideravelmente, o que acaba tornando a estrutura do documento e a sincronização entre os elementos complexa, dificultando a autoria.

Figura 1.2: Apresentação de objetos de mídia em regiões na tela da TV

A linguagem NCL dá suporte à adaptação do leiaute da aplicação em tempo de execução. Isso pode ser feito através da definição de elos que mapeiam as alterações desejadas e modificam o valor das características de apresentação, como por exemplo, tamanho ou posição de componentes visuais. O problema de tal abordagem é que o autor deverá estabelecer previamente todas as modificações que o documento irá realizar.

Nas linguagens multimídia, a falta de elementos de alto nível que facilitem a autoria de relacionamentos espaciais em uma aplicação resulta em dois problemas principais:

- aumento de relacionamentos espaço-temporais entre objetos de mídia tende a tornar a estrutura do documento e a sincronização entre os elementos mais complexa, elevando a probabilidade de erros na autoria;
- apesar de ser possível alterar as informações de apresentação do objeto de mídia em tempo de execução, o autor precisa prever todas as alterações possíveis o que torna a autoria extremamente difícil.

Esses problemas podem ser tratados a partir de duas soluções que podem ser combinadas:

- utilizar especificações genéricas de programas, como modelos pré-definidos, denominados *templates de composição* [12] na definição do sincronismo espacial, reduzindo o esforço de autoria;

- prover uma arquitetura que possa prestar serviços de adaptação do leiaute quando as características de apresentação dos objetos de mídia mudam ou quando o contexto em que essas mídias são exibidas se altera.

Resumidamente, *templates* de composição hipermídia [12], ou simplesmente *templates*, são utilizados para definir semântica espaço-temporal em composições multimídia. Um *template* de composição representa um conjunto de objetos de mídia como um componente e define relações genéricas entre esses componentes. Em tempo de processamento do *template*, essas relações são instanciadas como relacionamentos (por exemplo, um par elo-conector em NCL) para cada objeto de mídia que está associado a um componente do *template*.

Alguns ganhos fornecidos pelo uso de *templates* de composição são: reutilização de especificações entre os diferentes documentos, redução do tamanho do código de um documento que usa *templates* e redução de possíveis erros de autoria.

Exemplos dessa abordagem podem ser encontrados em linguagens como: XTemplate 3.0 [4, 12] e *Template Authoring Language* (TAL) [13], que são utilizadas para gerar modelos que servirão de base para criação de documentos multimídia. Isso reduz a quantidade e a complexidade do código de uma aplicação que usa *templates* e diminui a probabilidade de erros na confecção deste código.

Embora o uso de *templates* reduza o esforço de autoria na definição da estrutura e da sincronização de documentos multimídia, ainda há alguns pontos que precisam ser melhorados, principalmente no que diz respeito a aspectos relacionados às características de apresentação de componentes visuais de um documento multimídia.

Geralmente, linguagens de programação suportam modelos de leiaute que permitem posicionar automaticamente componentes de interface na tela do exibidor, como uma grade. Tais modelos, como *grid* ou *flow*, seriam muito úteis para a especificação de características de apresentação genéricas dos objetos de mídia.

Como mencionado anteriormente, aplicações multimídia envolvem a apresentação de vários tipos de mídia em exibidores que vão desde *smartphones* até aparelhos digitais de TV e que impõem algumas restrições de apresentação como o tamanho da tela do exibidor.

O formato da tela do exibidor é um requisito importante quando se constrói uma aplicação multimídia. Às vezes, aplicações que possuem uma boa apresentação em um tipo de dispositivo, perdem a qualidade de apresentação quando são exibidas em outros aparelhos diferentes. Esta facilidade, que torna o leiaute das aplicações mais agradável

ao usuário, independente do dispositivo, ou não é oferecida nativamente em linguagens declarativas para autoria multimídia ou o sistema usado para a especificação de leiautes dinâmicos oferece relações espaciais muito básicas, como por exemplo, o tamanho da mídia A é duas vezes maior que o tamanho da mídia B.

1.2 Objetivos

Esta tese propõe uma nova linguagem específica de domínio (*Domain Specific Language* (DSL)) baseada em *eXtensible Markup Language* (XML) para a definição de modelos de leiautes adaptativos e dinâmicos, denominada *Spatial Template Language* (STyLe). Esta linguagem visa permitir uma diminuição da complexidade e, consequentemente, do esforço de autoria de documentos multimídia declarativos, além de diminuir a possibilidade de erros do autor.

Para facilitar a declaração do leiaute espacial de um documento, STyLe apresenta um conjunto de restrições espaciais predefinidas e utiliza *templates* e modelos de leiaute [14]. A ideia é fornecer uma versão dinâmica dos modelos para que o leiaute espacial possa utilizá-los, mesmo quando mudanças ocorrerem. Adicionalmente às restrições predefinidas, STyLe ainda permite que o autor personalize tipos de restrições espaciais através de conectores de restrição que foram apresentados inicialmente em [12].

STyLe consegue adaptar o leiaute espacial de um documento multimídia através de restrições espaciais que consideram o estado dos objetos de mídia. Além disso, as restrições em STyLe podem também ser utilizadas para adaptar a aplicação aos diferentes contextos de apresentação como diferentes tamanhos de tela.

É necessário fornecer um conjunto de serviços capazes de interpretar as restrições espaciais e construir corretamente o leiaute espacial de acordo com o número de elementos do documento. Esse é outro objetivo importante desta tese.

É proposta uma arquitetura capaz de atualizar o leiaute espacial dinamicamente em tempo de execução. Ao realizar uma atualização em um leiaute dinâmico, o tempo entre a interação do usuário e a modificação do leiaute em resposta a esta interação é fundamental. Portanto, tem-se como objetivo criar uma arquitetura capaz de responder às intervenções com desempenho satisfatório para que o usuário tenha uma boa experiência ao interagir com a aplicação.

É importante ressaltar que o objetivo de StyLe é definir o leiaute espacial de apresentação do documento. Relações temporais entre objetos de mídia no documento devem ser estabelecidas diretamente na linguagem de autoria multimídia alvo ou geradas utilizando-se outras linguagens de *template*, tais como XTemplate [14] ou TAL [13]. Uma solução completa é possível se StyLe for integrada a uma dessas linguagens de *template*.

Como prova de conceito, esta tese propõe e implementa extensões na linguagem de autoria multimídia NCL e na linguagem de autoria de *templates XTemplate* para utilizar StyLe na concepção dos leiautes espaciais dos documentos. Embora utilizada com NCL e XTemplate, é possível estender outras linguagens de autoria multimídia como HTML e SMIL para utilizar StyLe e leiautes dinâmicos.

1.3 Contribuições Principais

Ao se atingir os dois objetivos principais destacados para esta tese que são: definição de uma linguagem para criação de leiautes espaciais adaptativos e dinâmicos e construção de uma arquitetura capaz de realizar os ajustes espaciais necessários quando o documento é alterado em tempo de exibição, as contribuições alcançadas são:

1. Linguagem StyLe para autoria de modelos de leiaute adaptativos e dinâmicos baseados em restrições espaciais para documentos multimídia;
2. Extensão da linguagem NCL para uso da linguagem StyLe;
3. Implementação do processador da linguagem StyLe para documentos NCL;
4. Versão 4.0 da linguagem XTemplate para uso da linguagem StyLe;
5. Implementação do processador XTemplate 4.0 para documentos NCL;
6. Especificação e implementação de uma arquitetura que possibilite a criação de leiautes adaptativos e dinâmicos a partir da linguagem StyLe e que responda às alterações das características de apresentação de documentos multimídia NCL em tempo de exibição.

1.4 Organização do Trabalho

O restante do texto está organizado da seguinte forma.

O Capítulo 2 apresenta os principais trabalhos relacionados, separando-os em subseções que apontam as principais características de cada trabalho: leiautes adaptativos, relações espaciais, linguagem de autoria baseada em *templates*, e alteração dinâmica de documentos.

O Capítulo 3 propõe a linguagem STyLe descrevendo em detalhes sua sintaxe. Neste capítulo, são apresentados os elementos da linguagem com enfoque no elemento básico *item*, o agrupamento de *itens* em *containers* e a definição de restrições espaciais predefinidas e restrições espaciais personalizadas pelo autor. Além disso, este capítulo apresenta a definição de leiautes adaptativos e modelos predefinidos de *containers* oferecidos pela linguagem.

O Capítulo 4 mostra as extensões necessárias na linguagem NCL para que possa utilizar a linguagem STyLe. Além de um exemplo dessa utilização, o processamento feito para que o documento NCL, que faz uso de um *template* em STyLe, se torne um documento NCL padrão é mostrado.

O Capítulo 5 apresenta a versão 4.0 da linguagem XTemplate focalizando nas facilidades adicionadas: aninhamento de *templates* e utilização de modelos de leiautes. Além disso, é mostrado como XTemplate 4.0 utiliza STyLe.

O Capítulo 6 discute a especificação e implementação de uma arquitetura para que documentos escritos na linguagem NCL que utilizam STyLe possam ser adaptados dinamicamente em tempo de execução.

Para que essa arquitetura seja avaliada, o Capítulo 7 apresenta alguns testes que levam em consideração o tempo de resposta da implementação da arquitetura proposta quando o documento que está em execução é alterado. Além disso, testes sobre a usabilidade da linguagem são apresentados.

Finalmente, o Capítulo 8 apresenta uma discussão sobre todos os pontos apresentados, realçando as contribuições da tese e sugerindo trabalhos futuros que visam uma evolução das propostas apresentadas.

Capítulo 2

Trabalhos Relacionados

Esta tese trata de diferentes temas abordados na literatura, tais como leiautes adaptativos, relações espaciais, linguagem de autoria baseada em *templates*, e alteração dinâmica de documentos. Por isso, este capítulo apresenta os trabalhos relacionados em seções separadas por tema.

2.1 Leiautes Adaptativos

Pesquisas como [15, 16, 17], apresentam modelos de leiautes dinâmicos e adaptativos.

Em [15], os autores propõem um sistema para criar e apresentar documentos em grade para acomodar várias condições de visualização. De acordo com os autores, um estilo de leiaute adaptativo é codificado como um conjunto de *templates* baseados em grade que sabe como se adaptar ao tamanho das páginas e outras condições de visualização. Os *templates* possuem vários tipos de visualização e definem, através de relacionamentos baseados em restrições, como os elementos (texto, figuras, etc) devem ser arranjados usando como base propriedades do conteúdo do elemento, como tamanho ou proporção de uma figura, e propriedades de visualização onde conteúdo será mostrado, como por exemplo o tamanho da tela do exibidor.

O tamanho e o local de cada elemento em um *template* é determinado pela avaliação de um conjunto interdependente de restrições que formam um dígrafo acíclico. O sistema de restrição compreende um conjunto de variáveis de restrição, cujos valores são determinados por uma expressão matemática em termos de outras variáveis de restrição. Esta configuração é conhecida como sistema de restrição de uma via (*one-way constraint system*). As variáveis de restrição de um *template* podem ser divididas em dois tipos:

entrada e saída. As variáveis de entrada são usadas para informar ao *template* sobre o contexto que será usado, por exemplo dimensões da página onde o *template* será usado. Já as variáveis de saída são usadas para indicar as regiões de cada elemento. Uma variável importante de saída é denominada *template*.score que permite a um *template* expressar sua adequação em termos de variáveis de entrada.

Dado o conteúdo a ser apresentado e mapeado pelas variáveis de entrada, vários *templates* são analisados e é escolhido aquele que se adequa mais.

Uma extensão desse trabalho é apresentada em [16] e traz algumas novas ideias com relação ao trabalho anterior, como: decisões sobre regiões onde os elementos deveriam ficar no leiaute passaram a ser individuais e não mais sobre a página inteira, foram apresentadas uma nova linguagem de mais alto nível para especificação de *template* e decisões mais inteligentes sobre sobreposição de janelas.

Esses dois trabalhos serviram de base para algumas ideias apresentadas nesta tese, como uma linguagem de mais alto nível para definir as restrições espaciais do *template*.

Já em [17], os autores apresentam uma linguagem de definição de leiautes baseados em *templates* para revistas digitais. O objetivo da abordagem é fazer com que o editor da revista possa definir suas preferências de leiautes com o mínimo esforço. Além disso, o sistema deveria ser apto a realizar escolhas razoáveis sobre a disposição do conteúdo sem intervenção do usuário.

Na solução proposta, o leiaute da página é dividido em colunas e os *templates* se referem sempre a uma ou mais colunas. A altura da coluna é fixa, baseada nas propriedades do dispositivo. O tamanho ideal da largura da coluna é calculado automaticamente baseado no tamanho do texto e se a tela do dispositivo pode ser rolada horizontalmente.

O leiaute é definido com a ajuda dos *templates* que são utilizados juntos com restrições que podem ser aplicadas tanto aos *templates* quanto ao leiaute propriamente dito. Os *templates* podem ser hierárquicos e podem conter substitutos, que são ordenados. Se um *template* não puder ser aplicado a um conteúdo específico, por conta por exemplo do tamanho da tela, o sistema aplica o *template* substituto melhor ranqueado. A solução, que obtém resultados satisfatórios, é simples e adequada a uma estrutura mais bem-comportada como sugere o leiaute de uma revista. Os conceitos propostos nesta tese são mais flexíveis no que diz respeito à estrutura visual do leiaute.

Existem ainda ferramentas que trabalham com a definição de características de leiaute e/ou autoria de documentos XML baseada em *templates* para documentos web, como:

CSS Regions Module [18] e AXEL [19]. O Módulo de Regiões CSS permite que o conteúdo de um ou mais elementos continue através de uma ou mais áreas denominadas regiões CSS. Além disso, permite que leiautes dinâmicos sejam instanciados. Já AXEL (*Adaptable XML Editing Library*) é uma biblioteca *JavaScript* para autoria de conteúdo XML baseado em *templates* na Web. Baseia-se no uso do XTiger XML, linguagem de definição de *template* baseada em *eXtensible Hypertext Markup Language* (XHTML), que é utilizada para expressar restrições estruturais no conteúdo editado. A construção do documento é feita sem uma clara divisão entre o *template* e o conteúdo do documento.

2.2 Relações Espaciais

Um documento multimídia interativo é uma composição de diferentes recursos de mídia (texto, imagem, vídeo e áudio). As relações temporais, espaciais e a definição de como se dará a interação do usuário com esses recursos definem uma apresentação multimídia interativa para o documento [20].

Alguns trabalhos tem um foco maior na discussão sobre as relações espaciais como em [21, 22, 23, 24].

Em [21], o autor define relações espaciais para documentos hipermídia. Essas relações podem ser estabelecidas entre janelas ou regiões. Embora o conjunto de relações seja interessante, sua utilização é desnecessariamente complexa. STyLe simplifica as relações definidas em [21] e fornece a mesma expressividade. Além disso, a linguagem proposta nesta tese é baseada em *template* e estabelece uma clara separação entre definições espaciais e temporais.

Os autores em [22] definem um modelo, baseado em um conjunto de relações espaçais e temporais entre os objetos, para especificar aplicações multimídia. Considerando relações espaciais, os autores apresentam três aspectos importantes para representação de composições definidas no espaço:

- relação topológica entre os objetos (são disjuntos, se interceptam, se sobrepõem, etc);
- relação direcional entre os objetos (esquerda, direita, abaixo, abaixo-direita, etc);
- relação de distância entre os objetos (5 cm para fora, 2 cm para dentro, etc).

Levando em consideração as relações topológicas, pode-se criar um conjunto, denominado *4-intersection model* e apresentado em [25], com todas as relações entre dois objetos do tipo: coincidem (*equals*), sobrepõem (*overlap*), interceptam externamente (*meet*), interceptam internamente (*covers* ou *covered_ by*), está dentro do outro (*inside* ou *contains*) ou são disjuntos (*disjoints*). Para relações topológicas, os autores em [25] provaram que este conjunto é completo.

Randell et al. [23] apresentam um conjunto parecido de relações espaciais: *disconnected*, *externally connected*, *partially overllaping*, *tangential proper part*, *Non-tangential proper part*, *equal*.

Pode-se fazer um paralelo entre os dois conjuntos, como mostrado na Tabela 2.1.

Tabela 2.1: Paralelo entre as relações espaciais

Trabalhos	Relações espaciais					
Egenhofer et al. [25]	equals	overlap	meet	covers	inside	disjoints
Randell et al. [23]	equal	partially overlla-ping	externally connec-ted	tangential proper part	Non-tangential proper part	disconnected

Essas relações podem ser consideradas como uma projeção no espaço para as relações temporais definidas por Allen em [26]. A Figura 2.1 representa essa projeção usando espaço 1D. Algumas relações podem ser unidas em uma só, como a R1 e R13, gerando um conjunto reduzido.

Adicionando as relações de direção e distância às relações topológicas, é possível estender esse conjunto para um conjunto muito maior, com 169 relações (13^2), como mostrado em [22] e [24]. Os autores provam que este conjunto é completo para essas relações.

Em [27], os autores apresentam uma abordagem para validação espaço-temporal de documentos multimídia. A validação espaço-temporal leva em consideração as informações de posicionamento dos objetos de mídia declarados no documento e que essas informações podem ser alteradas com o passar do tempo. Para que a validação possa ser realizada, é necessário que o autor do documento possa descrever suas expectativas com relação ao leiaute espaço-temporal. No eixo espacial, fórmulas que representam relações espaciais entre regiões foram disponibilizadas. Essas relações são as mesmas encontradas em [23], e serviram de conjunto base para que as relações espaciais oferecidas pela linguagem STyLe fosse desenvolvida.

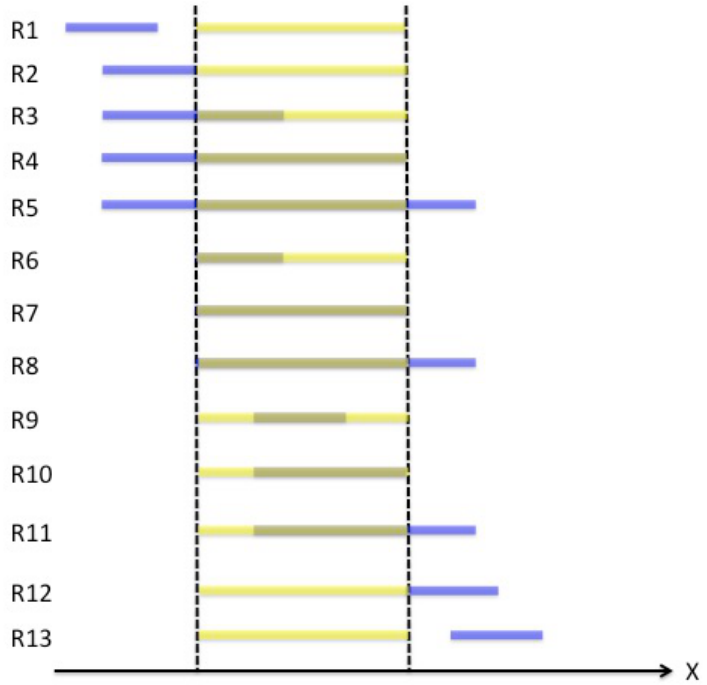

Figura 2.1: Projeção espacial das relações temporais de Allen

É importante notar que a expressividade da linguagem STyLe está fortemente ligada a possibilidade de se fornecer todas as relações espaciais determinadas pelos trabalhos anteriores. No Capítulo 3, será mostrado que STyLe consegue mapear todas essas relações espaciais.

STyLe utiliza uma extensão dos modelos de leiautes adaptativos introduzido em [14] como uma de suas características para criar as relações espaciais desejadas.

2.3 Linguagens de Autoria Baseadas em *Templates*

Alguns trabalhos que tratam da criação de documentos baseados em *templates* podem ser encontrados em [28, 29, 4, 13].

2.3.1 LimSee 3

Os autores em [28] apresentam o modelo LimSee3 que define uma linguagem de autoria, independente das linguagens existentes, concentrando na estrutura lógica do documento e não na semântica do elemento da linguagem alvo. Com esta abordagem, LimSee3 fornece flexibilidade e reúso. O *template* criado com esta linguagem descreve uma hierarquia

genérica entre seus componentes. A sincronização temporal em LimSee3 é feita usando elementos que fazem referência a *containers* SMIL.

Em LimSee 3, um *template* tem que ser editado para se tornar um documento completo ou outro *template*. Não existe diferença explícita entre um documento e um *template* em si. Caso seja necessária uma personalização para gerar um documento final, são definidas zonas para permitir que o autor possa incluir elementos específicos. Para evitar que partes importantes ou imutáveis sejam editadas por acaso por autores não experientes, partes do *template* podem ser bloqueadas usando um elemento criado com essa finalidade.

Esta abordagem abstrai a dependência entre os elementos do *template* e seus componentes externos, dando-lhes nomes simbólicos que são usados nas definições do leiaute e das relações temporais. Esta camada de abstração facilita a extração de um componente de seu contexto, aumentando a modularidade. A maior desvantagem desta abordagem é que visões globais do documento (visão temporal global e leiaute espacial global) não são diretamente acessíveis porque precisam ser processadas, o que pode ser custoso.

LimSee 3 incide sobre todo o documento definindo um *template* que pode ser exportado para qualquer linguagem cuja semântica está incluída no modelo do documento. Como mencionado anteriormente, a sincronização temporal é baseada em SMIL, fornecendo três tipos de semântica temporal embutidas nas composições (<par>, <seq> e <excl>). Além disso, LimSee 3 não fornece clara distinção entre o *template* e sua instância, já que o autor precisa editar o *template* diretamente para preencher as lacunas que restam. O *template* é aplicado sobre todo o documento, não permitindo que composições dentro do documento possam ter uma outra semântica embutida ou estendida.

2.3.2 STAMP

Em [29], os autores apresentam o modelo *Synchronized Template for Adaptable Multimedia Presentation* (STAMP), que propõe uma solução para adaptação de apresentações multimídia. Este modelo é aplicado a um conteúdo de apresentação que vem de uma base de dados, concentrando-se, então, em sistemas web. STAMP utiliza *templates* para criar apresentações automaticamente. O modelo e a estrutura da apresentação podem ser adaptados de acordo com o número de elementos recuperados da base de dados. Entretanto, STAMP não permite estender um *template* e não introduz semântica espaço-temporal dentro de uma composição.

STAMP descreve uma apresentação multimídia em seis dimensões. A dimensão espacial que se refere a organização espacial dos componentes na apresentação. A dimensão temporal que define o cenário da apresentação. A dimensão de navegação que define os elos de navegação entre os elementos e podem ser ativados pelo usuário durante a apresentação. A dimensão do dado que se refere ao conteúdo a ser inserido na apresentação. O conteúdo é obtido através de requisições que são avaliadas no momento em que são recebidas. Esse modelo leva em consideração o fato de que o conteúdo pode não ser conhecido quando descrito pelas outras dimensões. Já a dimensão de composição representa a apresentação como uma estrutura hierárquica construída a partir do conteúdo recebido e a dimensão decorativa permite especificar para qualquer componente da apresentação algumas propriedades adicionais relacionadas à aparência.

No modelo STAMP, o *template* é uma descrição fornecida por um autor de *template* e é o ponto de partida no processo de geração da apresentação pretendida. Um *template* permite definir uma apresentação de acordo com as dimensões: espacial, temporal e de navegação. As dimensões espaciais e temporais baseiam-se em relações qualitativas entre os componentes da apresentação, como por exemplo, A ficará a esquerda de B ou A aparece depois de B. A dimensão de navegação trabalha com definições de links para dentro ou fora do documento de apresentação.

A adaptação no STAMP é realizada através de restrições espaciais e temporais definidas no *template*. De acordo com os autores, situações que requerem adaptação são: *overflow* de dados, falta de espaço disponível e um tempo muito curto para as consultas.

A adaptação ao número de itens recuperados de uma origem pode ser realizada por três soluções diferentes: i) ignorar os itens adicionais, ii) apresentar os itens um por um e iii) modificar a estrutura inicial da apresentação. A solução de adaptação proposta nesta tese se aproxima da solução proposta em (ii). Os objetos de mídia adicionais serão relacionados ao leiaute como se pertencessem a uma nova *"página"* da apresentação.

A falta de espaço é detectada pelas restrições espaciais definidas para os elementos que estão sendo apresentados em um instante. Três soluções são propostas, i) excluir da apresentação um elemento que faz parte do conflito, ii) modificar o cenário temporal para acabar com o conflito e iii) criar uma apresentação complementar que será relacionada à apresentação inicial para acabar com o conflito.

A adaptação dinâmica de leiautes proposta no Capítulo 5 desta tese também se baseia em restrições, mas essas restrições são resolvidas em tempo de apresentação, diferentemente do STAMP onde as restrições são resolvidas em tempo de criação do documento.

Outra diferença é que as restrições espaciais propostas nesta tese têm níveis de prioridade que são avaliados para resolução de conflitos.

2.3.3 XTemplate 3.0

A versão 3.0 de XTemplate [4] foi definida para manter a simplicidade na especificação e no uso de um *template* de composição das versões anteriores XTemplate 2.1 [30] e XTemplate [12], enquanto algumas facilidades foram adicionadas, como por exemplo, a possibilidade de definição de características de apresentação para os componentes de *template* e definição de valores de variáveis em tempo de processamento. Embora a versão 3.0 permita que o autor do *template* defina características de apresentação em seus componentes, estas definições não se adaptam ao número de objetos de mídia.

É possível que vários *templates* sejam usados em conjunto para definirem semântica em diferentes contextos no documento NCL. Quando vários *templates* são usados em um mesmo documento NCL, o autor NCL deve acessar a descrição dos metadados de cada *template* para entender seu comportamento e como eles podem ser unidos.

Em XTemplate 3.0, a definição de *templates* de composição é composta de quatro partes: cabeçalho Ca, vocabulário V, corpo Co e restrições R, onde somente o vocabulário é obrigatório. A representação simplificada de um *template* é então: XTP = Ca; V; Co; R.

O cabeçalho pode importar especificações de apresentações através de elementos *descriptorBase*, que definem a forma como as mídias serão inicialmente apresentadas, e *connectorBase*, usados para criar as relações espaço-temporais entre os componentes do vocabulário de um *template*. Um *template* pode ainda estender definições de outros *templates* através do elemento *extends*.

O vocabulário define tipos de componentes e tipos de relacionamentos. Os tipos de componentes são especificados através do elemento *component*. Cada tipo de componente pode ter pontos de interface (elementos *port*) que podem representar âncoras, portas, propriedades ou outros componentes aninhados. Já os tipos de relacionamentos são definidos como conectores hipermídia. Todo elemento do vocabulário pode definir seu número máximo e mínimo de ocorrências.

No corpo, são declarados pontos de interface (portas) da composição que usa o *template*; instâncias de componentes do vocabulário e seus pontos de interface; especificação de relacionamentos entre os componentes usando conectores e de inclusão, referenciando

os tipos de componentes e conectores declarados no vocabulário ou instâncias de componentes. É possível a utilização de instruções para declaração de variáveis (elemento *variable*) e instruções para a realização de repetições (elemento *for-each ... selects*) especificadas pelo padrão XSLT [31] na especificação de pontos de interface da composição, dos componentes, relacionamentos e elementos de controle de conteúdo (*switches*).

As restrições, assim como nas versões anteriores, são responsáveis pela definição de restrições adicionais a serem aplicadas sobre elementos do vocabulário. As restrições são definidas separadamente (elemento *constraint*) contendo uma expressão na linguagem XPath [32], que retorna um valor booleano. A definição de uma restrição permite especificar uma mensagem de erro que pode ser indicada pelo processador de *templates* caso a restrição não seja satisfeita pelo documento que usa o *template*.

Apesar da inserção das facilidades citadas acima, é possível apontar algumas limitações da versão 3.0. A definição de regiões e descritores pode ser realizada no próprio *template*, mas é feita exatamente como em um documento NCL, isto é, é necessário determinar explicitamente todas as regiões e os descritores relacionados. Esta tese propõe uma solução para esta limitação da linguagem XTemplate.

Outro fator limitador é a impossibilidade de definir *templates* aninhados. Em geral, *templates* são utilizados para definir uma estrutura genérica dos nós e elos para uma única composição hipermídia. Portanto, sempre que se quer usar múltiplos *templates* dentro de um documento, é necessário importar explicitamente estes *templates* e associá-los às composições correspondentes no documento. A desvantagem de tal cenário para o autor que utiliza múltiplos *templates* é que, ao utilizá-los, o autor deverá ter conhecimento sobre cada *template* individualmente e como os múltiplos *templates* podem ser usados em conjunto. Esta tese também propõe uma solução para esta limitação de XTemplate.

2.3.4 TAL

TAL (Template Authoring Language) [13] suporta especificações de *templates* denominadas composições hipermídia incompletas. Como XTemplate, TAL pode definir um conjunto de documentos que compartilham a mesma estrutura de composição. Todavia, TAL não permite atribuir a seus componentes informações de leiaute, nem fornece qualquer facilidade para criar definições genéricas de características de apresentação. TAL possui aninhamento de *templates*, mas não são definidas interfaces específicas para este aninhamento, então a propriedade de composicionalidade pode ser violada. Um *template* em TAL pode ser visto como a especificação de uma composição em aberto. Uma

composição em aberto é definida por quatro elementos principais: Vocabulário, Restrições, Recursos e Relacionamentos hipermídia. Esses elementos se baseiam nos que foram originalmente definidos em XTemplate [12].

O Vocabulário define os tipos de seus componentes e interfaces. Na especificação do vocabulário, componentes são a representação de conjuntos de objetos de mídia ou de outras composições internas aos documentos finais. Interfaces são a representação de conjuntos de âncoras em objetos. Âncoras podem ser de conteúdo, representando parte do conteúdo de um objeto de mídia, ou uma propriedade (atributo) de um objeto de mídia ou de uma composição. As Restrições especificam regras sobre os tipos definidos no vocabulário. Podem também definir expressões algébricas correlacionando a cardinalidade de tipos diversos. Restrições são utilizadas para verificar a correção do processo de instanciação de *templates*.

Os Recursos são objetos não-editáveis da composição, isto é, não permitem qualquer tipo de edição. Se um elemento é declarado como Recurso, ele será copiado integralmente para o documento final. Um exemplo desse tipo de elemento pode ser um logo que deve aparecer sempre em uma aplicação. Já a especificação de relacionamentos hipermídia dá a uma composição sua semântica. Pode ocorrer entre tipos, ou entre recursos, ou entre tipos e recursos.

Em comparação com XTemplate 3.0, os autores descrevem que a linguagem TAL é mais simples pois evita o uso de notações externas ao modelo conceitual e fora do nível de abstração da linguagem como o emprego de expressões em XPath por exemplo. Entretanto, se for observada uma expressão típica em TAL como: $< tal : componentid =$ $"button" selects = "media[class = button]" / >$, pode-se notar uma semelhança muito próxima à expressão XPath utilizada em XTemplate, como por exemplo: $< componentid =$ $"button" selects = "child :: *[xlabel =' media'][class = button]" / >$.

2.4 Alterações Dinâmicas em Documentos Hipermídia

Normalmente, o acesso à informação pode se dar através de vários dispositivos, então alterações dinâmicas no leiaute de aplicações são comumente empregadas. Um dos exemplos mais utilizados hoje em dia são diretivas *HyperText Markup Language* (HTML)+CSS que permitem que páginas web possam se adaptar dinamicamente às alterações de apresentação do dispositivo exibidor. A criação de páginas web que respondem às alterações de características de exibição dos dispositivos, como tamanho de tela, é chamada de

Responsive Web Design (RWD) [33]. RWD é uma técnica que utiliza somente HTML e CSS sem usar linguagem procedural como *JavaScript*.

Foram adicionados inúmeros elementos XML tanto em HTML quanto em CSS que possibilitaram uma construção simples e eficaz de páginas que respondem às alterações de apresentação. Entretanto, o código que define a página precisa prever estas alterações e, normalmente, ele não é editado em tempo de apresentação. Em cenários como ambientes de TV Digital, alterações dinâmicas no código da aplicação em tempo de execução são possíveis e podem provocar alterações nas características de apresentação da aplicação. Este tema é uma área ainda pouco explorada.

Uma tentativa de tornar a linguagem CSS mais expressiva em relação à definição de leiautes dinâmicos pode ser encontrada em [8]. Neste trabalho, os autores propõem *Grid Style Sheets* (GSS). GSS é um pré-processador, em tempo de execução, de CSS e *JavaScript* que implementa o algoritmo *Cassowary*. GSS e *Cassowary* são baseados em *"Constraint Programming"*. O algoritmo *Cassowary* foi proposto em [7]. Nesse sistema, restrições podem expressar relações entre variáveis que podem ou não ser satisfeitos. Qualquer propriedade numérica de um elemento pode participar de uma relação de restrição, não somente posicionamento ou tamanho. Entretanto, GSS não possui elementos de mais alto nível que representem relações de restrição como alinhamentos ou distribuições. Esses elementos podem ser encontrados facilmente em ferramentas de edição gráfica, como o projeto *Inkscape*, ou ferramentas de construção de apresentações, com o *Keynote* da Apple. Além disso, o GSS não fornece modelos de leiautes como podem ser encontrados em [34].

Em [35] os autores propõem, de forma geral, uma arquitetura baseada em *templates* para aplicações hipermídia dinâmicas e em [36] os autores se baseiam nessa proposta para definir um *framework* capaz de prover os serviços para geração automática e dinâmica de aplicações hipermídia declarativas usando uma abordagem no lado do receptor da aplicação, ao invés do lado do servidor.

Os serviços são implementados por um processador de *templates*, um agente e um filtro. Os *templates* são responsáveis por garantir as especificações a serem seguidas e pelas restrições que o agente deve seguir. As restrições são necessárias para que as alterações sofridas pela aplicação não a tornem inconsistente.

Eventos gerados pela aplicação base (aquela que sofrerá alterações dinâmicas) iniciam o processamento do agente. O agente então começa a reconstruir a aplicação baseado em um *template*. Ao terminar a reconstrução, o agente envia para o processador de *template* o

documento de preenchimento gerado e o *template* utilizado. O processador de *template*, ao receber os documentos enviados pelo agente, processa o *template* junto com o documento de preenchimento e obtém o novo documento hipermídia. Esse documento é então enviado ao filtro que observa as diferenças entre a aplicação base e o novo documento, gera os comandos de edição necessários e aplica-os no documento base.

Apesar dos autores utilizarem uma linguagem baseada em *templates* para manter a consistência, nenhuma ferramenta de validação é utilizada para garanti-la.

Em [37] é apresentado um mecanismo de *Eletronic Program Guide* (EPG) adaptável que permite a personalização do leiaute da aplicação, associado ao oferecimento de um meta-serviço voltado para construção dinâmica dessas aplicações. O guia eletrônico é implementado como uma aplicação da linguagem alvo. A abordagem utilizada para implementar a construção dessa aplicação foi híbrida, isto é, uma parte declarativa e outra procedural. Além disso, os módulos que compõem o meta-serviço também são híbridos, pois parte é residente no receptor do usuário e parte é recebida pelo canal de difusão.

A arquitetura do meta-serviço (construção da aplicação) é dividida em duas partes: Alimentação e Produção. Na fase de Alimentação, as informações para montar o guia são obtidas e transformadas em estruturas de dados internas. Essas estruturas são repassadas para a fase de Produção que constrói as estruturas declarativas da aplicação EPG. O meta-serviço tem parte de seus componentes descritos de forma declarativa e parte de seus componentes escritos utilizando-se linguagem procedural.

O processo de Alimentação possui como módulos principais uma Máquina de Busca, que tem como objetivo reunir as informações que vêm multiplexadas no conteúdo audiovisual ou através de informações distribuídas na rede, e um módulo Gerenciador de Dados que controla a Máquina de Busca. Já o processo de Produção separa os aspectos de leiaute e da estrutura da aplicação através dos Gerenciadores de Estilo e de Conteúdo Estruturado.

O meta-serviço é agregado, como aplicação residente, à arquitetura modular do *middleware* declarativo Ginga-NCL e utiliza o Gerenciador de Base Privadas do *middleware* para processar os comandos de edição provenientes e alterar, quando necessário, as informações do EPG. Isso pode ser realizado em tempo de exibição do EPG. O meta-serviço utiliza o Gerenciador de Leiaute da máquina de apresentação Ginga-NCL para fazer a associação do conteúdo com as regiões definidas no dispositivo.

As alterações dinâmicas são restritas às informações apresentadas no Guia e não devem interferir no leiaute do EPG. Para alterar o leiaute do Guia, é necessário alterar as informações provenientes do Gerenciador de Estilo e isso não é dinâmico.

Para manter a consistência de documentos hipermídia que são alterados dinamicamente, a principal abordagem é a baseada em restrições [38, 39]. Uma das maiores vantagens em utilizar restrições é que elas permitem especificações parciais dos leiautes, as quais podem ser combinadas com outras especificações parciais, tornando a solução bastante flexível.

O trabalho de [38] apresenta um sistema onde tanto o autor da página web quanto o espectador podem declarar restrições de leiaute para a página, algumas requeridas outras preferenciais. O modelo de leiaute baseado em restrições proposto pelos autores possui três componentes principais: a ferramenta de autoria do documento, a ferramenta de visualização e o solver utilizado para processar as restrições.

A ferramenta de autoria permite que o autor possa editar o conteúdo do documento. O conteúdo pode consistir de textos, caixas invisíveis, imagens e tabelas. Cada componente, diferente de texto, possui uma caixa delimitadora e restrições entre essas caixas definidas pelo leiaute do documento. O texto do documento flui ao redor dessas caixas.

O sistema de restrição emprega dois algoritmos para solucionar as restrições impostas, *Cassowary* e *BAFSS*. Ambos permitem restrições exigidas e restrições preferenciais que podem ter diferentes prioridades. *Cassowary* é usado para restrições com igualdades e desigualdades lineares. A coleção de restrições pode incluir ciclos, restrições redundantes e incompatibilidade de preferências. Já o *BAFSS* é usado para restrições de domínio finito, mas não pode conter ciclos. O *BAFSS* resolve as restrições relacionadas ao tamanho de fonte e o *Cassowary* resolve as restrições de leiaute da página. A interação com cada solver pode ocorrer de três formas: uma restrição pode ser adicionada ao conjunto de restrições atual, uma restrição pode ser excluída do conjunto de restrições e novos valores podem ser aplicados às variáveis gerando uma nova solução para o leiaute.

Os autores apresentaram um protótipo, construído em Java, do sistema com quatro janelas. A janela principal mostra a visão espacial do documento utilizando a folha de estilo de restrições definida. A folha de estilo de restrições é mostrada como uma coleção textual de restrições em duas janelas: uma para as restrições de domínio finito que controlam o tamanho e tipo do texto e outra para as restrições aritméticas lineares. A última janela contém um editor para o texto do documento. A utilização de restrições com valores de prioridades diferentes foi usada na solução proposta nesta tese.

Em [40] os autores propõem um conjunto de extensões para linguagens baseadas em XML, como SMIL, para fornecer a capacidade de reagir a eventos gerados dinamicamente pelos usuários. As extensões capturam certos comportamentos dinâmicos nas sentenças descritas em XML e preveem o dinamismo da apresentação em resposta às entradas do usuário. As mudanças dinâmicas do comportamento podem acontecer de duas formas: baseada em eventos, que consiste de uma mudança discreta do estado da apresentação quando uma propriedade particular torna-se verdadeira; e baseada em dependência contínua, que permite que a saída se modifique continuamente em resposta a uma entrada dinâmica. Para fornecer suporte a essas alterações de comportamento, os autores propõem duas extensões:

1. Valores de atributos definidos como expressões avaliadas dinamicamente;
2. Eventos customizados baseados em expressões de predicado.

A sintaxe para a linguagem baseada em expressão proposta segue a sintaxe adotada para expressões de temporização em SMIL 2, combinando referências a propriedades da forma elemento.propriedade com operadores aritméticos e constantes. A linguagem é fortemente tipada e fornece três tipos possíveis: numérico, *string* e *boolean*.

Os autores fornecem um estudo simplificado sobre a frequência que uma determinada expressão deve ser calculada. O cenário pode gerar expressões dependentes e alterações em algumas expressões podem ocasionar alterações em outras expressões e assim por diante. Um estudo similar é feito na implementação da arquitetura proposta na tese para verificar em quais casos a solução ainda mantém um desempenho satisfatório.

Em [41], os autores propõem a adição de mecanismos declarativos de leiautes *Scalable Vector Graphics* (SVG) que permitem um comportamento adaptativo de elementos espaciais em relação a modificações, tais como: variação do tamanho da janela do *browser* ou diferentes tamanhos de fontes. Uma sugestão dada pelos autores é que valores de atributos sejam especificados através de expressões que devem ser avaliadas em tempo de execução para determinar o valor do atributo utilizado. Uma pequena extensão de SVG, chamada de *Constraint Scalable Vector Graphics* (CSVG), que fornece tal funcionalidade foi descrita. Um conjunto de restrições espaciais foi também proposto. Os autores demonstraram o uso de uma rede acíclica de restrições que foram declaradas como expressões que mapeavam as propriedades dos atributos entre componentes SVG. Por exemplo, uma definição do tipo *svg: rec id = " B " x = " A.x + A.width + 30 "* deveria definir B afastado 30 pixels do lado direito de A. Então, sempre que a propriedade A for atualizada, a

modificação é refletida em B. Os autores demonstraram também que a extensão proposta permite a criação de leiautes sofisticados.

Exemplos simples, como demonstrado no exemplo anterior, podem facilmente ser descritos com CSVG. Entretanto, quando os leiautes são mais sofisticados, as expressões tendem a aumentar, consideravelmente, a complexidade. StyLe tenta manter a simplicidade de autoria fornecendo restrições pré-definidas que determinam comportamentos mais expressivos.

2.5 Comparação entre os Trabalhos Relacionados

Como descrito anteriormente, esta tese envolve trabalhos das seguintes áreas: leiautes adaptativos, relações espaciais, linguagens de autoria baseadas em *templates* e alterações dinâmicas em documentos hipermídia. Nem todas as pesquisas que serviram de referência para o trabalho proposto possuem características de todas as áreas citadas e, por isso, não foram diretamente comparadas com a linguagem StyLe.

A Tabela 2.2 traz uma comparação entre os trabalhos que possuem características semelhantes, levando em consideração aspectos que se mostraram importantes no desenvolvimento da pesquisa com foco em leiautes adaptativos e dinâmicos. A última coluna da tabela apresenta as características de StyLe.

	STAMP[29]	XTemplate3[4]	TAL[13]	LimSee3[28]	GSS[8]	CSVG[41]	Schrier et. al.[17]	Borning et. al.[38]	de Moura[21]	*STyLe*
Baseado em templates	✓	✓	✓	✓	-	-	✓	-	-	✓
Baseado em restrições	✓	-	-	-	✓	✓	✓	✓	✓	✓
Leiautes Estáticos	✓	✓	-	-	✓	✓	✓	✓	✓	✓
Leiautes Adaptativos	✓	-	-	-	✓	✓	-	-	-	✓
Leiautes Dinâmicos	-	-	-	-	✓	✓	✓	✓	-	✓
Relações espaciais de alto nível	-	-	-	-	-	-	✓	-	-	✓
Edição em tempo de execução	-	-	-	-	-	-	-	-	-	✓

Tabela 2.2: Tabela de comparação entre os trabalhos relacionados

Como pode ser visualizado, StyLe oferece todas as características importantes para a criação de leiautes adaptativos e dinâmicos. É importante ressaltar que a solução proposta oferece a possibilidade de alteração do leiaute em tempo de execução, característica que não é vista em nenhum dos trabalhos citados.

O próximo capítulo traz a descrição detalhada da linguagem StyLe, o que facilitará a observação das características apontadas na tabela.

Capítulo 3

Linguagem SᵀʸLe

Esta tese propõe a linguagem SᵀʸLe, baseada em XML e usada para definição de *templates* de leiautes espaciais que serão instanciados em documentos multimídia. SᵀʸLe permite a definição de leiautes personalizados pelo próprio autor, através da especificação de relações de restrição entre elementos que compõem um leiaute. Essas relações manipulam atributos de posição ou tamanho dos elementos, tornando-se uma solução flexível e bastante expressiva. Permitir ao autor especificar novos modelos de leiaute é uma das principais vantagens da utilização de restrições em SᵀʸLe. Além disso, o autor pode utilizar modelos de leiautes predefinidos pela linguagem. O intuito é diminuir o esforço de autoria na criação de leiautes adaptativos e dinâmicos.

Quando um leiaute espacial é definido utilizando restrições e modelos de leiautes, ele pode se ajustar de acordo com a quantidade de objetos de mídia a serem apresentados, por exemplo, e pode ser alterado dinamicamente, em tempo de execução, quando acontecem eventos durante a apresentação do documento. Isto acontece porque as restrições deverão ser sempre satisfeitas independente das mudanças que ocorrerem no documento e os modelos de leiautes preveem mecanismos para adaptar o leiaute a quantidade de mídias. A ideia é manter o leiaute definido no *template* SᵀʸLe durante toda a execução do documento. Dessa forma, SᵀʸLe oferece leiautes adaptativos e dinâmicos.

As restrições espaciais SᵀʸLe podem ter tipos predefinidos ou podem ser definidas de forma personalizada pelo autor, utilizando-se conectores de restrição (*constraintConnector*), que são baseados na linguagem XConnector [12], como será detalhado adiante neste capítulo.

Ao se propor a linguagem SᵀʸLe, alguns aspectos foram considerados importantes: diminuir o esforço de autoria para descrever um leiaute e manter o leiaute consistente

mesmo na presença de modificações das características de apresentação dos objetos de mídia.

Como SᵀyLᵉ define *templates* de leiautes espaciais, é possível reutilizar esses modelos para criar uma família de aplicações que tenham características similares em relação ao leiaute, como por exemplo, aplicações que utilizam um menu de opções. Além disso, SᵀyLᵉ utiliza componentes de leiaute (*containers*) que podem representar genericamente arranjos espaciais, como uma grade. Com uma simples descrição é possível criar uma grade de qualquer tamanho. Como mostrado em [34], esta representação diminui consideravelmente o esforço de autoria se compararmos a representação utilizada para construir o mesmo leiaute sem usar o *template*.

Outro aspecto importante é garantir que o leiaute possa ser modificado em tempo de execução e ainda manter a consistência da descrição. Isso é realizado através do conjunto de restrições estabelecido para os componentes do *template*. Sempre que um evento que altera características de apresentação de um objeto de mídia é detectado, o conjunto de restrições é avaliado por um *solver* que verifica a satisfabilidade do mesmo e retorna os novos valores para que as alterações sejam empregadas. Esse aspecto está relacionado à execução da aplicação e não da linguagem em si, mas só é possível porque a linguagem é baseada em restrições espaciais.

Com o pensamento em manter a sintaxe da linguagem simples, SᵀyLᵉ permite que o autor utilize restrições espaciais predefinidas para criar relações espaciais entre os componentes da linguagem. As relações que podem ser criadas através das restrições predefinidas são: relações de alinhamento, relações de distribuição e relações de tamanho. As relações serão aplicadas sobre atributos (pontos de interface) dos componentes de leiaute. Esses atributos podem ser: topo, base, direita, esquerda, centro e meio. Com isso, é possível descrever uma relação do tipo: alinhar o topo do objeto A com a base do objeto B. Ainda é possível definir restrições customizadas através de conectores de restrição. Isso torna a solução expressiva o suficiente para definir todas as relações espaciais descritas em [22], como mostrado no final do capítulo.

A linguagem SᵀyLᵉ é especificada em três diferentes módulos, definidos com Schema XML [42]: módulo *Básico*, módulo *Container* e módulo *Transition*. Os módulos e elementos SᵀyLᵉ são apresentados na Tabela 3.1. A especificação completa de SᵀyLᵉ em XML Schema está disponível no Apêndice B.

O módulo *Básico* define o elemento raiz da linguagem, chamado *layout*, e seus elementos filhos, os elementos *head* e *body*.

O elemento *head* pode ter como elementos filhos o elemento *importBase* e o elemento *connectorBase*. O elemento *importBase* é utilizado para importar outras bases já definidas como, por exemplo, outros *templates* de leiautes Style, e o elemento *connectorBase* é usado para definir uma base de conectores de restrições espaciais que serão referenciados no elemento *body*. Essa definição é realizada através da declaração do elemento *constraintConnector*, feita dentro do elemento *connectorBase* ou importando uma base de conectores já definida em outro arquivo. O elemento *constraintConnector* é usado para construir restrições personalizadas pelo autor. Os elementos *importBase*, *connectorBase* e *constraintConnector*, assim como seus filhos e atributos foram trazidos da linguagem NCL.

Basic Module		
Elementos Principais	Atributos	Conteúdo
layout	id	head, body
head		ncl:connectorBase, ncl:transitionBase
ncl:connectorBase		ncl:constraintConnector, ncl:importBase
ncl:constraintConnector	id	ncl:connectorParam, ncl:compoundStatement, ncl:assessmentStatement, ncl:attributeAssessment
body	id	item, container, spatialConstraint
item	id, top, left, right, bottom, width, height, focusIndex	
spatialConstraint	id, xconnector, type, priority, offset, factor, master	bind
Container Module		
Elementos Principais	Atributos	Conteúdo
container	id, type, transIn, transOut, sort, rearrange, rearrangeDur	item, format, focus, container, spatialConstraint
format	top, left, right, bottom, width, height, align, vspace, hspace, columns, rows, orientation, step, zIndex	
focus	focusIndex, moveLeft, moveRight, moveUp, moveDown	
Transition Module		
Elementos Principais	Atributos	Conteúdo
ncl:transitionBase		ncl:transition, ncl:importBase
ncl:transition	id, type, dur	

Tabela 3.1: Elementos e atributos da linguagem STyLe

3.1 Elementos da Linguagem

A linguagem StyLe baseia-se em três componentes principais: *item*, *container* e *restrições espaciais*. Esse componentes são determinados na linguagem através dos elementos: *item*, *container* e *spatialConstraint*. Esta seção descreve detalhadamente cada um desses elementos.

3.1.1 Item

O elemento básico em um *template* StyLe é o *item*. Ele define uma área retangular no leiaute espacial do documento. A escolha de uma área retangular deve-se ao fato de que a maioria das linguagens de autoria para aplicações multimídia interativas, tais como NCL e SMIL, utilizam regiões retangulares para definição espacial de objetos de mídia. Um *item* representa características de apresentação que serão atribuídas a, possivelmente, muitos objetos de mídia.

O autor pode especificar todas características de apresentação de um *item* como: ponto inicial da área (através dos atributos *top* e *left*), largura (através do atributo *width*) e altura (através do atributo *height*); ou definir restrições espaciais que possam gerar esses valores. Exemplos dessas especificações podem ser observados na Listagem 3.1.

```
1 <item id="A" top="0" left="100" width="100" height="100"/>
2 <item id="B" left="250" width="50" height="20"/>
3
4 <spatialConstraint id="topAlign" type="align" priority="high">
5   <bind component="A" interface="top"/>
6   <bind component="B" interface="top"/>
7 </spatialConstraint>
```

Listing 3.1: Exemplo de itens e restrições

Um *item* pode ser declarado no corpo do leiaute ou dentro de um *container*. No exemplo apresentado na Listagem 3.2, três itens, A, B e C são declarados dentro do *container*. O item A tem sua posição e tamanho determinados pelos seus atributos, enquanto que o posicionamento dos itens B e C são definidos pelas restrições espaciais *dist1* e *align1*.

3.1.2 Container

Em StyLe, *itens* podem ser agrupados dentro de um elemento espacial, denominado *container*.

O elemento *container* estabelece uma área de exibição delimitadora contendo *itens* e/ou outros *containers* aninhados. A posição e o tamanho desses elementos, assim como o comportamento quando os elementos estão sendo apresentados ou não, podem ser especificados dentro do *container* através da declaração de atributos e/ou restrições espaciais, como pode ser visto na Listagem 3.2.

```
<container id="foo">
  <item id="A" top="0" left="100" width="260" height="195"/>
  <item id="B">
  <item id="C">

  <spatialConstraint id="same_size" type="size" factor="1" >
    <bind component="A" interface="size"/>
    <bind component="B" interface="size"/>
    <bind component="C" interface="size"/>
  </spatialConstraint>

  <spatialConstraint id="dist1" type="distribute" direction="horizontal" >
    <bind component="A"/>
    <bind component="B"/>
    <bind component="C"/>
  </spatialConstraint>

  <spatialConstraint id="align1" type="align">
    <bind component="A" interface="bottom"/>
    <bind component="B" interface="top"/>
    <bind component="C" interface="bottom"/>
  </spatialConstraint>
</container>
```

Listing 3.2: Exemplo de container

Este *container* pode ser visualizado na Figura 3.1.

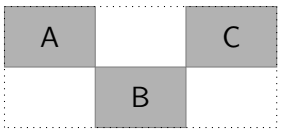

Figura 3.1: Container

Quando as características de apresentação dos elementos internos ao *container* são determinadas por restrições espaciais definidas dentro do próprio *container*, este é chamado de *container não tipado*. Outra opção para determinar o posicionamento e o tamanho dos elementos dentro de um *container* é seguindo uma regra definida por um modelo de leiaute, como descrito em [34]. Quatro modelos foram disponibilizados em StyLe: *gridLayout*, *flowLayout*, *stackLayout* e *carouselLayout*. Os modelos são especificados através do atributo *type* do elemento *container*. Quando isso é feito, o elemento é denominado *container* tipado.

Modelos de leiaute podem ser vistos como *templates* de *container* e serão definidos com mais detalhes a seguir, na Subseção 3.1.3. Ao se utilizar modelos de leiaute, o autor não precisa definir restrições espaciais para estabelecer as características de apresentação

dos elementos internos, pois já é predefinido no modelo. Isso reduz o esforço de autoria no que tange, por exemplo, a quantidade de linhas de código para descrever o *container*.

Todos os modelos de leiaute podem ser definidos diretamente com itens e restrições espaciais, mesmo assim, os modelos predefinidos são oferecidos para facilitar a autoria e diminuir o esforço do autor de *templates* de leiaute. A Listagem 3.3 apresenta um exemplo do *container flowLayout*.

```
1 <container id="f1" type="flowLayout" >
2   <format top="10" left="10" width="600" height="400" align="center"
3     vspace="10" hspace="10"/>
4   <item id="D" width="100" height="120"/>
5   <focus focusIndex="1" />
6 </container>
```

Listing 3.3: *Container do tipo flowLayout*

Um *container* pode descrever uma transição para os *itens* internos ao *container* e uma animação para rearranjar os elementos restantes de um *container*, quando ocorrer um evento de parada da execução de um *item*.

Para realizar uma transição ao se mostrar os *itens* pela primeira vez, se faz necessária a declaração de um elemento *transition* na base de transições localizada no elemento *head*. O elemento *transition* possui como atributos um identificador único, o tipo da transição (*barWipe*, *fade*, *diagonalWipe* e *clockWipe*) e a duração da transição. Para relacionar uma transição a um *container* e definir quando a transição irá ocorrer (início ou fim da apresentação do objeto de mídia), basta definir um atributo *transIn* ou *transOut* com o valor igual ao *id* da transição. Esse processo é o mesmo que é feito na linguagem NCL [9] para criar transições. Uma representação gráfica das transições pode ser visualizada na Figura 3.2.

Figura 3.2: Representação gráfica das transições

É possível animar a mudança de posição de *itens* em um *container* para que ela seja percebida de forma mais suave pelo usuário. Isso é definido através do atributo *rearrange* que espera receber dois valores: *animate* e *appear*. O valor *animate* anima a forma como os *itens* serão rearranjados quando um deles for retirado, e o valor *appear* posiciona os *itens* nas coordenadas corretas sem fazer uma animação. Para estabelecer a duração da animação, o atributo *rearrangeDur* deve ser declarado.

Itens e *containers* definem pontos de interface que representam coordenadas específicas em sua área retangular. Esses pontos de interface são atributos que podem ser utilizados nas restrições. A Figura 3.3 mostra os pontos de interface disponíveis para itens e *containers*.

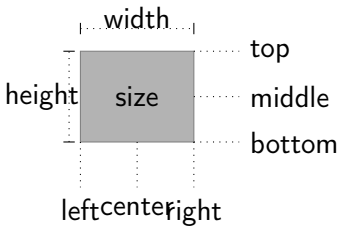

Figura 3.3: Pontos de Interface de *Item* e *Container*

3.1.3 Modelos de Leiaute

Um *container* pode especificar opcionalmente um modelo predefinido de leiaute. Um modelo de leiaute especifica diretivas genéricas para a apresentação de um conjunto de objetos de mídia. Por exemplo, um modelo de leiaute em grade especifica como apresentar objetos dispostos em uma grade.

Instâncias de modelos de leiaute são declaradas por *containers* tipados, que utilizam um modelo de leiaute predefinido. Um *container* é chamado então de componente de leiaute. Um componente de leiaute especifica a área na tela onde objetos de mídia a ele associados serão inseridos, além de definir informações específicas sobre o modelo de leiaute utilizado. Por exemplo, um componente de leiaute usando um modelo de leiaute em grade especifica o seu número de linhas e colunas.

A Figura 3.4 apresenta um componente de leiaute e um exemplo de documento que usa este componente. O item de leiaute pode ser identificado como a região azul mais escura da Figura 3.4a.

a) Componente b) Exemplo de utilização do componente

Figura 3.4: Representação de um componente de leiaute e um exemplo de uso

No exemplo da Figura 3.4a, o tamanho do item é o mesmo para todas as regiões definidas, mas pode-se definir itens de tamanhos diferentes como mostrado na Figura 3.5.

Figura 3.5: Itens com tamanhos diferentes em um componente de leiaute

Na linguagem XTemplate [12], *templates* de composição definem componentes que representam grupos de objetos de mídia. Na utilização de um *template* de composição, o autor de um documento associa objetos de mídia específicos a componentes do *template* rotulando-os (atributo *xlabel* em *XTemplate*) com a identificação de um dado componente do *template*.

Seguindo a mesma ideia, esta tese propõe que objetos de mídia possam ser associados a componentes de leiaute, rotulando-os (atributo *layout*) com a identificação de um componente de leiaute definido em um *template* de leiaute. Um *template* de leiaute define componentes e itens de leiaute.

Como será apresentado na próxima seção, StyLe oferece restrições para criar relacionamentos entre componentes e itens de leiaute. Usando restrições, o autor tem bastante liberdade para criar o modelo de leiaute que deseja. Entretanto, para facilitar a autoria, StyLe também oferece alguns modelos de leiaute predefinidos na linguagem.

Para determinar quais modelos de leiaute predefinidos seriam fornecidos pela linguagem StyLe proposta nesta tese, diferentes aplicações multimídia interativas criadas pela comunidade NCL e disponíveis na página do Clube NCL [43] foram analisadas. As aplicações avaliadas mostraram características de apresentação que se assemelhavam muito com gerenciadores de leiaute como: *FlowLayout* e *GridLayout* oferecidos em linguagens de programação, tal como Java por exemplo. Outros leiautes encontrados nas aplicações do Clube NCL ou não tinham uma estrutura bem definida ou podiam ser representados por meio da composição desses gerenciadores de leiaute. Com base nesse estudo das aplicações, foram definidos os modelos de leiaute a serem oferecidos. Futuramente, pode-se estender StyLe para que novos modelos possam ser implementados.

A linguagem StyLe para autoria de leiautes adaptativos fornece quatro modelos de leiautes:

- *FlowLayout*: os itens do leiaute são colocados da esquerda para a direita, linha por linha. Quando não existe mais espaço na linha, considerando o tamanho do item observado, outra linha é criada embaixo e utiliza-se o mesmo princípio novamente;
- *GridLayout*: os itens do leiaute são colocados da esquerda para a direita, linha por linha, no formato de uma grade. A grade é sempre construída considerando toda a área especificada pelo componente de leiaute;
- *CarouselLayout*: os itens do leiaute são colocados no formato de uma elipse. A elipse é construída tangenciando internamente uma região retangular definida pelo autor;
- *StackLayout*: os itens do leiaute são colocados sobrepostos uns aos outros formando uma pilha. O item de um nível seguinte pode ter um deslocamento na horizontal e na vertical com relação ao item do nível anterior. Os itens são colocados do nível mais interno para o nível mais externo em relação à tela do dispositivo.

Exemplos dos modelos podem ser visualizados na Figura 3.6.

Figura 3.6: Exemplos dos modelos de Leiautes Adaptativos

É importante ressaltar que os leiautes apresentados neste capítulo também podem ser usados em conjunto em um mesmo documento para formar um padrão de regiões mais elaborado.

Um componente de leiaute é definido pelo elemento *container*, que possui o atributo opcional *type* indicando o modelo de leiaute instanciado por ele. Um elemento *container* com o tipo *flowLayout* não precisa definir um número de linhas e colunas, mas, por outro lado, precisa definir o alinhamento dos itens. Já um elemento com tipo *gridLayout* define o número de linhas e colunas, mas não define o alinhamento nem o tamanho dos itens uma vez que são inerentes ao modelo. Um elemento com tipo *carouselLayout* define a região que circunda a elipse e com o tipo *stackLayout* define índices de sobreposição entre os elementos.

Os itens de um componente de leiaute são declarados pelo elemento *item*. Sempre que um componente de um leiaute instancia um modelo *flowLayout* ou um modelo *carouselLayout*, é possível incluir itens de tamanhos diferentes no mesmo componente. Para isso, cada elemento *item* deve definir seus próprios atributos *width* e *height*. Os itens do modelo *gridLayout* terão sempre o mesmo tamanho, que é uma razão entre a área em pixels do elemento *container* (dada pelas medidas de tamanho definidas) e o produto entre o número de linhas e colunas. Os itens do modelo *stackLayout* terão o mesmo tamanho, mas índices de sobreposição diferentes (atributo *zIndex*).

Qualquer componente de leiaute deve ser instanciado dentro do elemento raiz *layout*. Este elemento delimita todas as declarações para um *template* de leiaute adaptativo e possui, como atributo requerido, um identificador único (*id*).

A Tabela 3.2 apresenta as informações dos elementos e atributos XML para a especificação de componentes e itens de leiaute.

Tabela 3.2: Elementos e Atributos XML de Containers e Itens de Leiaute

Elemento	Atributo	Tipo do Valor
container	id	string
	type	string (flowLayout | gridLayout | carouselLayout | stackLayout)
format	width	double | porcentagem
	height	double | porcentagem
	top	double | porcentagem
	left	double | porcentagem
	right	double | porcentagem
	bottom	double | porcentagem
	align	center | left | right
	hspace	double
	vspace	double
	columns	inteiro
	rows	inteiro
	zIndex	inteiro
	orientation	diagonal | left_to_right | right_to_left | top_to_bottom | bottom_to_top
	step	double
item	id	string
	width	double | porcentagem
	height	double | porcentagem
focus	focusIndex	inteiro
	moveUp	inteiro
	moveDown	inteiro
	moveLeft	inteiro
	moveRight	inteiro

As próximas subseções detalham cada tipo de modelo de leiaute proposto, explicando os elementos e atributos utilizados para cada modelo de leiaute.

3.1.3.1 flowLayout

O leiaute definido pelo tipo *flowLayout* cria regiões da esquerda para a direita. A quantidade de regiões criadas em uma linha depende da quantidade de mídias que utilizam esse leiaute e foram declaradas no corpo do documento multimídia que utiliza o *template* de leiaute. Quando não existe mais espaço em uma linha, é criada outra linha abaixo dela e o mesmo critério é usado novamente. As mídias ocuparão as regiões obedecendo a ordem em que foram declaradas, isto é, primeira mídia declarada ocupará a primeira região criada por esse leiaute. Como descrito anteriormente, cada elemento do tipo *flowLayout* pode ainda definir elementos filhos para compor todo o comportamento do leiaute. Conforme a Tabela 1, os atributos possíveis para o elemento *format* são: *width, height, top, left, right, bottom, align, hspace e vspace.* Para item são: *id, width e height.* Já para focus os atributos possíveis são: *focusIndex, moveUp, moveDown, moveLeft e moveRight.*

Um exemplo de uso do *flowLayout* pode ser visualizado pelo código apresentado na Listagem 3.4, que define um *flow* com um item de tamanho 150 x 150 pixels. A região da tela que o modelo irá usar tem 600 pixels de largura e 400 pixels de altura e estará a 100 pixels do topo e 340 pixels da margem esquerda da tela do exibidor. A navegação entre os itens do leiaute será possível, pois o elemento filho *focus* foi declarado. Uma ilustração do leiaute gerado pelo código descrito pode ser visualizada na Figura 3.7.

```
1 <container id="F1" type="flowLayout">
2   <format align="center" top="100" left="340" width="600" height="400"
3     hspace="10" vspace="10" zIndex="3"/>
4   <item id="i1" width="150" height="150"/>
5   <focus focusIndex="1"/>
6 </container>
```

Listing 3.4: Exemplo de leiaute do tipo flowLayout

3.1.3.2 gridLayout

O leiaute definido pelo tipo *gridLayout* cria regiões em formato de grade. Para isso, são definidas as quantidades de linhas e colunas da grade. As mídias ocuparão as regiões obedecendo a ordem em que forem declaradas no documento que usa o *template* de leiaute. A primeira mídia declarada ocupará a primeira região criada por esse leiaute e assim sucessivamente. Mesmo que a quantidade de mídias seja inferior à quantidade de regiões estabelecidas, estas são criadas, mas não serão utilizadas. A grade será sempre

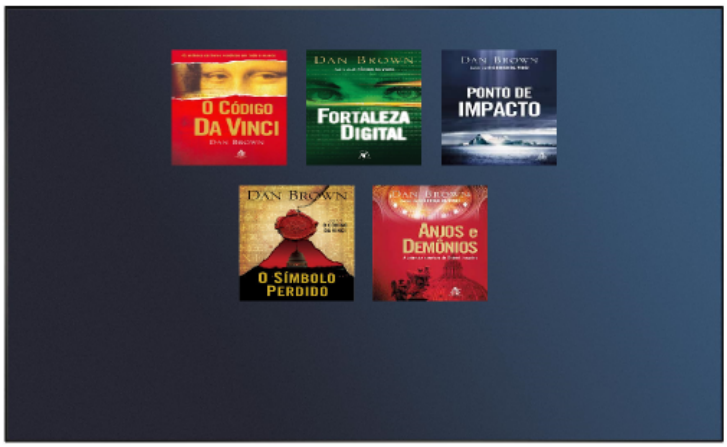

Figura 3.7: Exemplo de flowLayout criado a partir do código da Listagem 3.4

construída no centro da região determinada e, portanto, não terá o atributo *align* como no elemento *flowLayout*. Cada elemento *gridLayout* pode ainda definir elementos filhos para estabelecer o comportamento do leiaute. O elemento filho *item* não é necessário nesse leiaute, pois as características de cada célula da grade são derivadas da própria grade. Os atributos possíveis para o elemento *format* são: *width, height, top, left, right, bottom, hspace, vspace, columns e rows*. Já para o elemento focus os atributos possíveis são: *focusIndex, moveUp, moveDown, moveLeft e moveRight.*

Um exemplo de utilização do *gridLayout* pode ser visualizado pelo código apresentado na Listagem 3.5, que apresenta uma grade com 2 linhas e três colunas. A região utilizada pelo modelo terá 620 pixels de largura e 310 pixels de altura. Os itens estarão separados horizontalmente e verticalmente por 10 pixels e, novamente, será possível navegar entre os itens do modelo. Uma ilustração do leiaute gerado pelo código descrito pode ser visualizada na Figura 3.8.

```
1 <container id="G1" type="gridLayout">
2   <format top="100" left="340" width="620" height="310"
3     columns="3" rows="2" hspace="10" vspace="10"/>
4   <focus focusIndex="2"/>
5 </container>
```

Listing 3.5: Exemplo de leiaute do tipo gridLayout

3.1.3.3 carouselLayout

O leiaute definido pelo tipo *carouselLayout* cria regiões que serão arranjadas em formato de uma elipse. Para tal, o autor deve definir uma região retangular que irá circunscrever a elipse desejada. Dessa forma, as principais informações para se construir a elipse, semieixos maior e menor, podem ser concebidas através das informações dessa região.

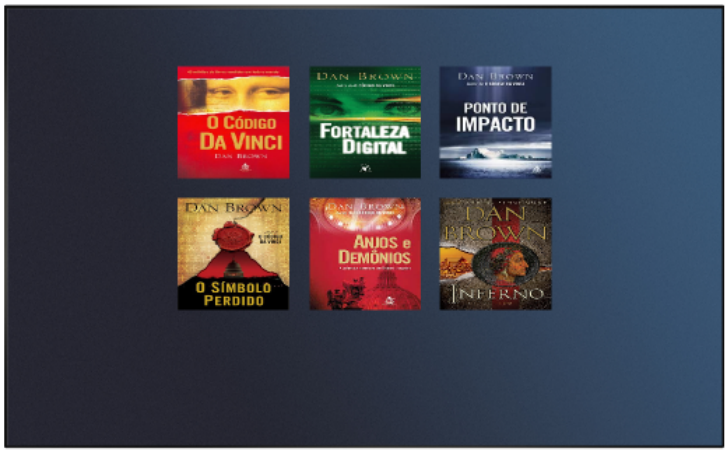

Figura 3.8: Exemplo de gridLayout criado a partir do código da Listagem 3.5

Todas as regiões criadas para formar esse modelo e que serão associadas aos objetos de mídia são internas à elipse, como é possível verificar na Figura 3.9.

Figura 3.9: Construção do modelo Carousel

Os pontos iniciais de cada região são distribuídos igualmente pela elipse, começando do ponto (0, semieixo menor) e seguindo no sentido horário. Assim como o modelo *flowLayout*, é possível determinar regiões de tamanhos diferentes através dos atributos *width e height* do elemento item. Cada elemento do tipo *carouselLayout* pode ainda definir elementos filhos para compor todo o comportamento do leiaute. Os atributos possíveis para o elemento *format* são: *width, height, top, left, right, bottom.* Para *item* são: *id, width e height.* Já para *focus*, os atributos possíveis são: *focusIndex, moveUp, moveDown, moveLeft e moveRight.*

A Figura 3.10 apresenta um exemplo de uso desse modelo. Este exemplo define um carrossel que será inscrito em uma região de 800 x 400 pixels e possuirá um tamanho único para o item de 152 x 40 pixels. Será possível navegar entre os itens do leiaute já que o elemento focus foi declarado. É possível visualizar o código XML na Listagem 3.6.

```
1 <container id="C1" type="carouselLayout">
2   <format top="200" left="240" width="800" height="400"/>
3   <item id="i1" width="40" height="152"/>
```

```
4   <focus focusIndex="3"/>
5 </container>
```

Listing 3.6: Exemplo de leiaute do tipo carouselLayout

3.1.3.4 stackLayout

O modelo de leiaute definido pelo tipo *stackLayout* cria regiões parcialmente sobrepostas. Estas regiões são dispostas dentro de uma região retangular e seguirão uma orientação, ambos atributos definidos pelo autor.

Para que a sobreposição seja corretamente interpretada, as regiões receberão um atributo zIndex, condizente com seu nível de sobreposição. Assim como em NCL, regiões com maiores valores para zIndex irão se sobrepor àquelas com valores menores. Este atributo será gerado automaticamente pelo processamento do leiaute, a partir de um valor inicial especificado pelo autor. Esse valor não é obrigatório e, se não indicado, terá como valor padrão 0.

Da mesma forma como *flowLayout e carouselLayout*, é necessário definir o tamanho do item a ser gerenciado pelo leiaute. Isso é feito declarando um elemento filho *item* com os atributos *width* e *height*.

As regiões serão criadas seguindo uma estrutura que se assemelha a uma pilha de elementos, isto é, a primeira será sobreposta pela segunda que será sobreposta pela terceira e assim por diante. O autor deverá indicar uma orientação (atributo *orientation*) e um alinhamento (atributo *align*) e a combinação desses atributos definirá como as regiões serão criadas. O atributo *orientation* poderá receber os valores: *diagonal, left_to_right, right_to_left, top_to_bottom e bottom_to_top*. Já o atributo *align* poderá receber os

Figura 3.10: Exemplo de carouselLayout criado a partir do código da Listagem 3.6

valores: *center, left e right*. Por exemplo, se os valores para *orientation e align* forem respectivamente *diagonal e right*, as regiões serão definidas da esquerda para a direita seguindo uma diagonal como mostra a Figura 3.11. O deslocamento entre cada região será determinado pelo atributo *step*. Se a orientação for em diagonal, ambos, atributos *top e left* da próxima região, serão incrementados igualmente com o valor de *step*. Se a orientação for da direita para esquerda ou vice-versa, o atributo *left* será decrementado/incrementado com *step*, mas o valor de *top* não será alterado; e se a orientação for de cima para baixo ou vice-versa, o atributo *top* será incrementado/decrementado com *step*, mas o valor de *left* não irá sofrer alterações.

Figura 3.11: Exemplo do modelo Stack

Cada elemento do tipo *stackLayout* pode ainda definir elementos filhos para compor todo o comportamento do leiaute. Os atributos possíveis para o elemento *format* são: *width, height, top, left, right, bottom, zIndex, orientation, align e step*. Para item são: *id, width e height*. Já para focus os atributos possíveis são: *focusIndex, moveUp, moveDown, moveLeft e moveRight*.

Um exemplo de código para declaração do modelo *stackLayout* pode ser visto na Listagem 3.7. Este código define um leiaute que ocupa uma região de 800 x 400 pixels. As regiões serão criadas a partir da esquerda para a direita, de cima para baixo e em diagonal. A região na base da "pilha" terá o atributo zIndex igual a 2. O atributo zIndex da próxima região um nível acima da "pilha" será incrementado de 1 e assim por diante. Uma ilustração do leiaute gerado pelo código descrito pode ser visualizada na Figura 3.12.

```
1 <container id="S1" type="stackLayout">
2   <format top="200" left="240" width="800" height="400"
3     zIndex="2" orientation="diagonal" align="right" step="50"/>
4   <item id="i1" width="40" height="152"/>
5   <focus focusIndex="4"/>
6 </container>
```

Listing 3.7: Exemplo do modelo de leiaute Stack

Figura 3.12: Exemplo de stackLayout criado a partir do código da Listagem 3.7

3.1.4 Navegação por Teclas

Além do tamanho e do posicionamento, um *container* pode descrever o comportamento de navegação por teclas entre seus itens e entre si e outros componentes de leiaute. Cada elemento *container* pode definir um elemento filho *focus* com um atributo *focusIndex* que determina um índice de navegação único para o componente de leiaute como um todo. Sempre que o elemento *focus* for definido, é necessário atribuir um valor inteiro positivo, que ainda não foi utilizado, ao atributo *focusIndex*. Se isso for efetuado, a navegação por teclas entre os itens de um componente de leiaute será ativada, portanto, objetos de mídia que referenciam um dado componente de leiaute serão associados à definição de navegação prevista neste componente. A Figura 3.13 apresenta um exemplo de navegação entre itens de um componente de leiaute.

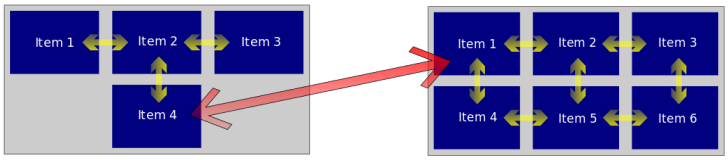

Figura 3.13: Navegação no leiaute

A navegação entre os componentes de leiaute como um todo é declarada pelos atributos *moveUp*, *moveDown*, *moveLeft* e *moveRight* do elemento focus. Esses atributos não são obrigatórios, mas ao menos um é necessário para estabelecer a navegação entre componentes. Cada atributo deve indicar o valor do *focusIndex*, definido no elemento *focus*, do componente de leiaute que receberá o foco quando a tecla do controle remoto correspondente for pressionada. Por exemplo, suponha que tenham sido definidos os componentes mostrados nas Listagens 3.4 e 3.5, e que se queira definir uma navegação entre

os componentes, isto é, deseja-se navegar do componente *flow* para o componente *grid* e vice-versa. Para isso, pode-se acrescentar os elementos *moveRight* e *moveLeft* como mostrado na Listagem 3.8.

```
<container id="F1" type="flowLayout">
  <format align="center" top="100" left="0" width="600" height="400"
    hspace="10" vspace="10" zIndex="3"/>
  <item id="i1" width="150" height="150"/>
  <focus focusIndex="1" moveRight="2"/>
</container>
...
<container id="G1" type="gridLayout">
  <format top="100" left="500" width="620" height="310"
    columns="3" rows="2" hspace="10" vspace="10"/>
  <focus focusIndex="2" moveLeft="1"/>
</container>
```

Listing 3.8: Exemplo de leiaute do tipo flowLayout

Sempre que houver navegação entre componentes de leiaute, itens localizados na borda de um componente irão herdar o atributo de navegação deste componente, permitindo a navegação entre itens de componentes de leiaute diferentes. Por exemplo, itens na borda inferior de um componente de leiaute irão herdar o atributo *moveDown* estabelecido para o componente. Aqueles que estão na borda à esquerda herdarão o atributo *moveLeft* e assim por diante. É importante ressaltar que, quando a navegação entre componentes não é declarada, o comportamento padrão é mantido, i.e., a navegação de um item de borda leva ao item na borda oposta do mesmo componente (navegação cíclica).

3.1.5 Restrições Espaciais

Uma restrição espacial é declarada em um *template* StyLe com o elemento *spatialConstraint*. É necessário definir os atributos *id* e *type*. O atributo *id* é usado para identificar unicamente a restrição espacial. O atributo *type*, quando requerida, identifica o tipo da restrição.

Para definir o nível de prioridade, é possível declarar o atributo *priority*, criando desta forma, uma hierarquia de restrições como mencionado em [38]. Este atributo é opcional e possui *high* como valor padrão. StyLe fornece quatro níveis de prioridade: *core*, *high*, *medium* e *low*. Uma restrição com o valor de prioridade igual a *core* deve sempre ser satisfeita pelo sistema. Os valores *high*, *medium* e *low* definem níveis preferidos, mas não requeridos. Então, quando o sistema tenta resolver as restrições, ele procura uma solução que atenda ao conjunto de restrições por completo. Se isso não é possível, uma nova tentativa é feita, removendo-se as restrições com prioridade *low* do conjunto de restrições. Esse processo continua até que o conjunto de restrições tenha somente restrições com

prioridade *core*. Se ainda assim o sistema não consegue encontrar uma solução, então um erro é gerado. Para definir a hierarquia das restrições dentro do mesmo nível de prioridade, a ordem de declaração dentro do *template* é usada.

Uma *spatialConstraint* pode relacionar dois ou mais componentes espaciais (*item* ou *container*). Os componentes que participam da restrição são identificados no elemento filho *bind*. Um elemento *bind* deve indicar o ponto de interface espacial do componente que será considerado. O componente participante e seu ponto de interface são determinados pelos atributos *component* e *interface*, respectivamente.

A linguagem StyLe fornece ao autor a possibilidade de definir uma restrição espacial personalizada. Isto é feito utilizando-se conectores de restrição, que são baseados na linguagem XConnector [12]. Este conector confere maior flexibilidade à linguagem, pois fornece ao autor a possibilidade de criar diferentes tipos de restrição.

Um conector de restrição é uma relação multiponto com semântica de restrição que define uma expressão assertiva e que será usado para criar relacionamentos de restrição entre elementos do leiaute. Ele especifica a semântica mas não os elementos que farão parte da relação espacial. Os elementos serão definidos nas restrições espaciais (*spatialConstraint*) que utilizarão esse conector. Essa expressão assertiva deve ser satisfeita durante a execução do documento.

Um conector define um conjunto de papeis (*role*), que especificam a função de um participante da relação, e como eles se relacionam. Cada *role* do conector define um *id*, um tipo do evento (*eventType*) e um tipo do atributo (*attributeType*). Papeis de restrição são especificados através de *statements*. Um *statement* contém uma expressão que avalia o valor de uma propriedade de um nó, como por exemplo, o valor da sua largura. A expressão pode ser simples ou composta. Uma expressão simples (*assessmentStatement*) compara papeis de um mesmo tipo ou um papel com um valor. Uma expressão composta (*compoundStatement*) consiste de uma expressão lógica com operadores *and* ou *or* envolvendo duas ou mais expressões.

A Listagem 3.9 apresenta um exemplo de um conector de restrição. Este conector de restrição determina que o elemento associado ao *role left1* e ao *role left2* devem ter seus pontos de interface com valores iguais e que o elemento associado ao *role base50* deve ter seu ponto de interface 50 pixels a mais do que o elemento associado ao *role base*.

```
1 <constraintConnector id="alignLeftBottomTop">
2   <compoundStatement operator="and">
3     <assessmentStatement comparator="eq">
4       <attributeAssessment role="left1" eventType="attribution" attributeType="nodeProperty"/>
5       <attributeAssessment role="left2" eventType="attribution" attributeType="nodeProperty"/>
6     </assessmentStatement>
```

```
7       <assessmentStatement comparator="eq">
8         <attributeAssessment role="base" eventType="attribution" attributeType="nodeProperty"/>
9         <attributeAssessment role="base50" eventType="attribution" attributeType="nodeProperty" offset="
          50"/>
10      </assessmentStatement>
11    </compoundStatement>
12  </constraintConnector>
```

Listing 3.9: Constraint Connector

Para utilizar um conector de restrição, uma restrição espacial deve declarar um atributo *xconnector* que relaciona esta restrição ao conector. O valor do atributo *xconnector* é o *id* do conector de restrição, que pode ter sido declarado no cabeçalho do *template* ou em uma base de conectores de restrição.

Cada elemento *bind* na restrição espacial deve ter um atributo *role* cujo valor é um dos papeis declarados no conector de restrição. As linhas 37-42 da Listagem 4.1 apresentam um exemplo da restrição espacial usando o conector exibido na Listagem 3.9.

Como é possível observar, a declaração do conector de restrição e sua utilização através do elemento *spatialConector* aumentam a verbosidade do código do *template*. Para facilitar a autoria do documento e diminuir o esforço do autor, StyLe propõe três tipos de restrições predefinidas para especificar um leiaute espacial, que são: *align*, *distribute* e *size*.

Todas as restrições predefinidas podem ser reescritas utilizando-se conectores de restrição. Entretanto, as restrições predefinidas diminuem a complexidade na autoria de restrições que relacionam vários componentes, por isso também são oferecidas por StyLe.

Uma restrição do tipo *align* serve para alinhar um ou mais pontos de interface de *itens* ou *containers*. Adicionalmente, pode-se definir um *offset* ou *factor* que será adicionado ou multiplicado entre cada par de pontos de interface participantes de uma dada restrição. O atributo *master* pode ser declarado para determinar se cada relação criada a partir do *align* terá como mestre o elemento imediatamente anterior (*previous*) ou o primeiro elemento declarado (*first*). No exemplo mostrado pela Listagem 3.10, o lado esquerdo do *item* B será alinhado com o lado esquerdo de A, acrescentando 10 pixels nesse alinhamento e o lado esquerdo de C será alinhado com o lado esquerdo de B, acrescentando mais 10 pixels nesse alinhamento. O valor *default* desse atributo é *first*.

```
1 <spatialConstraint id="leftAlign" type="align" offset="10" priority="core" master="previous">
2   <bind component="A" interface="left"/>
3   <bind component="B" interface="left"/>
4   <bind component="C" interface="left"/>
5 </spatialConstraint>
```

Listing 3.10: Exemplo da restrição espacial align

Uma restrição espacial do tipo *distribute* especifica que dois ou mais componentes espaciais deveriam ser distribuídos ao longo de uma parte da tela. Pode-se determinar um atributo *direction* com valores *horizontal*, *vertical* ou um valor representando um ângulo, que define a direção da distribuição. O ângulo entre dois itens é calculado como mostrado na Figura 3.14.

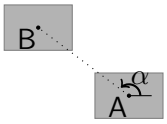

Figura 3.14: Distribute usando um ângulo

Já uma restrição espacial do tipo *size* cria relações de tamanho sobre os componentes, como por exemplo: o tamanho de A é duas vezes maior que o tamanho de B ou a altura de C é a altura de $D + 10$. Os atributos *factor* e *offset* definem se o valor vai ser multiplicado ou somado ao ponto de interface. Normalmente, os pontos de interface utilizados nessa restrição são pontos referentes à altura, largura ou tamanho de um item. O ponto de interface que relaciona o tamanho do elemento, isto é, manipula altura e largura ao mesmo tempo, denomina-se *size*. A Listagem 3.11 mostra um exemplo de utilização dessa restrição.

```
1 <spatialConstraint id="doubleSize" type="size" factor="2" priority="core">
2   <bind component="A" interface="size"/>
3   <bind component="B" interface="size"/>
4 </spatialConstraint>
```

Listing 3.11: Exemplo da restrição espacial size

3.2 Aninhamento de Leiautes

Para que o autor possa criar leiautes mais elaborados, StyLe fornece a possibilidade de aninhamento de leiautes. Por exemplo, é possível que o autor defina um componente de leiaute do tipo *flowLayout* e que um dos elementos internos desse leiaute seja um outro componente do tipo *gridLayout*, como apresentado na Figura 3.15.

Assim como um *container* inclui a declaração de um elemento *item* que estará contido dentro dele, também pode incluir a declaração de um *container* interno que fará parte do conjunto de seus elementos. A Listagem 3.12 mostra o código da declaração do componente de leiaute que foi apresentado na Figura 3.15.

```
1 <container id="f1" type="flowLayout" focusIndex="1">
2   <format top="10" left="10" width="600" height="400" align="center"
3     vspace="10" hspace="10"/>
```

```
4   <item id="D" width="100" height="120"/>
5   <container id="g1" type="gridLayout">
6       <format width="380" height="400" columns="3" rows="2"
7           hspace="10" vspace="10"/>
8     </container>
9 </container>
```

Listing 3.12: *Containers* aninhados

Uma questão importante a se definir é como arranjar os elementos dentro do *container* mais externo, também chamado de *pai*. Como mencionado anteriormente, o arranjo é definido pela ordem dos objetos de mídia que fazem referência ao leiaute e foram declaradas no documento que utiliza o *template* StyLe. A identificação do *container* interno deve ser feita utilizando o *id* do *container* pai, um ponto (.) como separador e o *id* do *container* interno. Essa identificação deixa explícita a hierarquia existente entre os *containers*.

Quando o leiaute for processado, o processador varre os objetos de mídia e verifica as referências feitas aos elementos do leiaute, criando as regiões adequadas para o elemento referenciado. Sempre que o processador encontrar o primeiro objeto de mídia que referencia um *container* interno, ele determinará uma região adequada levando em consideração a heurística definida pelo *container* mais externo (pai) e as informações contidas no elemento *format* do *container* em questão.

3.3 Relações Espaciais

As restrições espaciais (predefinidas ou personalizadas) criam relações espaciais entre os objetos de mídia do documento que utiliza um *template* StyLe. Relações espaciais em documentos multimídia já foram discutidas em vários trabalhos como [23, 24, 22, 27] e no Capítulo 2 desta tese.

Como discutido em [22], pode-se criar um conjunto, denominado *4-intersection model*, com todas as relações topológicas entre dois objetos do tipo: coincidem (*equals*), sobre-

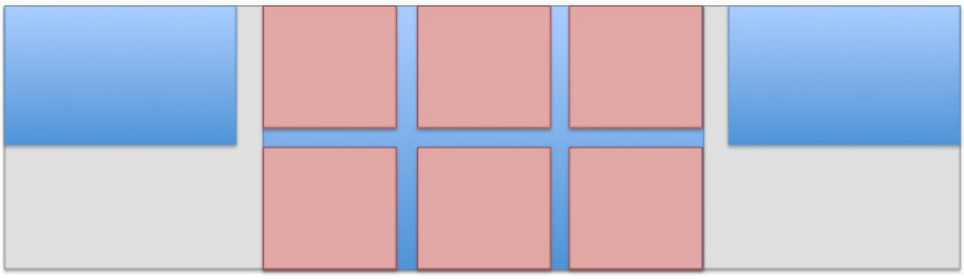

Figura 3.15: Exemplo de Leiautes Aninhados

põem (*overlap*), interceptam externamente (*meet*), interceptam internamente (*covers* ou *covered_by*), estão dentro do outro (*inside* ou *contains*) ou são disjuntos (*disjoints*). Quando se adicionam as relações de direção e distância às relações topológicas, é possível estender esse conjunto para um conjunto muito maior.

StyLe apresenta elementos que tornam possível representar qualquer relação apresentada nesses trabalhos [23, 24, 22, 21], o que demonstra a expressividade da linguagem. Para exemplificar esta afirmação, vamos considerar a apresentação das regiões somente no eixo x do espaço 2D (é possível repetir a operação para o eixo y sem perda de generalidade). A Figura 3.16 apresenta as representações gráficas das relações topológicas.

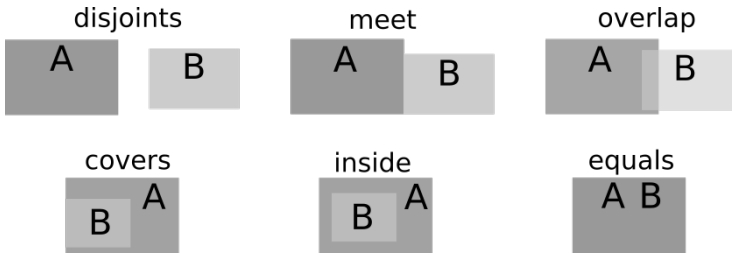

Figura 3.16: Representação gráfica das relações topológicas

Com os conectores de restrição, podemos criar qualquer relação espacial. Entretanto, relações espaciais complexas precisarão de conectores mais elaborados. Para tentar diminuir o esforço de autoria, StyLe fornece as restrições predefinidas. Com essas restrições, é possível criar as relações topológicas que foram descritas anteriormente, como se segue.

A relação espacial *disjoints* pode ser criada com uma restrição *distribute* ou com uma restrição *align* utilizando-se os pontos de interface *left* e *right* e um atributo *offset* positivo qualquer, como pode ser visto na Listagem 3.13.

```
<spatialConstraint id="relDisjoints" type="distribute" direction="horizontal" >
  <bind component="A"/>
  <bind component="B"/>
</spatialConstraint>
```

Listing 3.13: Representação da relação disjoints

A relação espacial *meet* pode ser criada com duas restrições *align*, a primeira utilizando-se os pontos de interface *left* e *right* e a segunda para alinhar os dois objetos pela base, como pode ser observado na Listagem 3.14.

```
<spatialConstraint id="leftRightAlign" type="align">
  <bind component="A" interface="left"/>
  <bind component="B" interface="right"/>
</spatialConstraint>

<spatialConstraint id="baseAlign" type="align">
  <bind component="A" interface="bottom"/>
  <bind component="B" interface="bottom"/>
```

```
</spatialConstraint>
```

Listing 3.14: Representação da relação meet

A relação espacial *overlap* pode ser criada com uma restrição espacial *align* utilizando-se os pontos de interface *center* e o atributo *offset* positivo, como pode ser observado na Listagem 3.15.

```
<spatialConstraint id="overlap" type="align" offset="50">
  <bind component="A" interface="center"/>
  <bind component="B" interface="center"/>
</spatialConstraint>
```

Listing 3.15: Representação da relação overlap

A relação espacial *covers* pode ser criada com uma restrição espacial *align* utilizando-se os pontos de interface *left* e outra restrição do tipo *size* com atributo *offset* ou *factor* para informar que o tamanho de um elemento é maior que o outro, como pode ser observado na Listagem 3.16.

```
<spatialConstraint id="covers1" type="size" factor="2">
  <bind component="B" interface="size"/>
  <bind component="A" interface="size"/>
</spatialConstraint>

<spatialConstraint id="covers2" type="align">
  <bind component="A" interface="left"/>
  <bind component="B" interface="left"/>
</spatialConstraint>
```

Listing 3.16: Representação da relação covers

A relação espacial *inside* pode ser criada com uma restrição espacial *align* utilizando-se os pontos de interface *left* e o atributo *offset* para desalinhar as arestas à esquerda e outra restrição do tipo *size* com atributo *offset* ou *factor* para garantir que o tamanho de um elemento é menor que o outro e que ele estará contido no primeiro, como pode ser observado na Listagem 3.17.

```
<spatialConstraint id="inside1" type="size" factor="2">
  <bind component="B" interface="size"/>
  <bind component="A" interface="size"/>
</spatialConstraint>

<spatialConstraint id="inside2" type="align" offset="10">
  <bind component="A" interface="left"/>
  <bind component="B" interface="left"/>
</spatialConstraint>
```

Listing 3.17: Representação da relação inside

A relação espacial *equals* pode ser criada com uma restrição espacial *size* utilizando-se o atributo *factor* igual a 1 e tendo como pontos de interface *size* e duas outras restrições

do tipo *align* com os pontos de interface *center* e *middle*, como pode ser observado na Listagem 3.18.

```
<spatialConstraint id="equals1" type="size" factor="1">
  <bind component="B" interface="size"/>
  <bind component="A" interface="size"/>
</spatialConstraint>

<spatialConstraint id="equals2" type="align">
  <bind component="A" interface="center"/>
  <bind component="B" interface="center"/>
</spatialConstraint>

<spatialConstraint id="equals3" type="align">
  <bind component="A" interface="middle"/>
  <bind component="B" interface="middle"/>
</spatialConstraint>
```

Listing 3.18: Representação da relação equals

Foi dado um exemplo de criação para cada relação espacial através das restrições espaciais definidas em STyLe, mas o exemplo dado não é único. Existem outras formas de criar as mesmas relações espaciais utilizando outras restrições ou combinações de restrições.

É possível relacionar, para cada aspecto da representação de composições espaciais descrito em [22], uma característica de STyLe que cobre o aspecto, como mostrado na Tabela 3.3.

Tabela 3.3: Relação entre os aspectos para representação espacial e características de STyLe

Aspectos	Características de STyLe
relação topológica	Restrições espaciais predefinidas: align, distribute e size
relação direcional	Pontos de interface: top, bottom, center, left, right, middle
relação de distância	Atributos: offset e factor

3.4 Redução da Linguagem STyLe para um problema SMT (Satisfiability Modulo Theories)

Uma das principais características da linguagem STyLe é a utilização de restrições para descrever as relações espaciais entre objetos de mídia em um leiaute. Além de serem utilizadas para definir a disposição dos objetos de mídia no interior do leiaute, restrições também são utilizadas para determinar a mudança do posicionamento dos objetos de mídia, quando acontecem modificações nas características de apresentação. Essas modificações podem ocorrer em tempo de execução, através de interações dos usuários.

Para manter a consistência do leiaute com a descrição das restrições espaciais, todas as restrições, inclusive as implícitas definidas pelos *containers* tipados, são representadas em

um conjunto de fórmulas SMT. Essas fórmulas são resolvidas por um *solver* que retorna um conjunto de valores. Esse conjunto de valores é tal que satisfaz todas as fórmulas propostas.

De acordo com [44], um problema SMT é um problema de decisão para fórmulas lógicas baseadas na combinação de teorias expressas em lógica de primeira ordem com igualdades. SMT pode ser pensado como uma forma de expressar problemas de satisfatibilidade de restrições e, portanto, ser definida como uma formalização para problemas que utilizam conjuntos de restrições. Já [45], define SMT como um problema de satisfatibilidade onde as fórmulas combinam conectivos lógicos, tais como conjunção, disjunção e negação, com fórmulas atômicas sob a forma de desigualdades aritméticas lineares. Um exemplo de fórmula SMT é apresentado da seguinte maneira.

$$(v_1 \leq v_2) \wedge (v_3 \vee (v_1 > v_2))$$

Como visto no exemplo, as fórmulas são compostas por variáveis (v_1, v_2 e v_3), representando valores booleanos ou aritméticos. Uma solução para tal fórmula é uma atribuição que mapeia as variáveis v_i para valores que tornam a fórmula verdadeira.

Em STyLe, para efeito sua representação em SMT, tanto um *item* quanto um *container* são mapeados em regiões retangulares. Uma região retangular é caracterizada pelas seguintes variáveis:

$$r{:}x^{ini}, r{:}x^{mid}, r{:}x^{end}, r{:}x^{wid}, r{:}y^{ini}, r{:}y^{cen}, r{:}y^{end}, r{:}y^{hei} \in \mathbb{R}_{\geq 0}$$
$$r{:}s^{occ} \in \mathit{Bool}$$

onde $r{:}x^{ini}$, $r{:}x^{mid}$, $r{:}x^{end}$ e $r{:}x^{wid}$ representam o início, o meio, o fim e a largura de uma região r no eixo x, enquanto $r{:}y^{ini}$, $r{:}y^{cen}$, $r{:}y^{end}$ e $r{:}y^{hei}$ representam o início, o centro, o fim e a altura da mesma região no eixo y. A variável $r{:}s^{occ}$ representa se um objeto de mídia é apresentado e está visível (verdadeiro ou falso).

As variáveis relativas a uma região se relacionam da seguinte maneira:

$$r{:}x^{ini} < r{:}x^{end} \tag{3.1}$$
$$r{:}x^{mid} = \left(r{:}x^{ini} + r{:}x^{end}\right)/2 \tag{3.2}$$
$$r{:}x^{end} = r{:}x^{ini} + r{:}x^{wid} \tag{3.3}$$
$$r{:}y^{ini} < r{:}y^{end} \tag{3.4}$$
$$r{:}y^{cen} = \left(r{:}y^{ini} + r{:}y^{end}\right)/2 \tag{3.5}$$
$$r{:}y^{end} = r{:}y^{ini} + r{:}y^{hei} \tag{3.6}$$

Todas as regiões, sejam elas originárias de itens ou *containers*, são determinadas dentro de um *canvas* que representa a tela do exibidor. Tal *canvas* é representando por uma região retangular, cuja posição e tamanho são definidas a priori da seguinte forma:

$$canvas{:}x^{ini} = 0 \tag{3.7}$$
$$canvas{:}y^{ini} = 0 \tag{3.8}$$
$$canvas{:}x^{wid} = screen.width \tag{3.9}$$
$$canvas{:}y^{hei} = screen.height \tag{3.10}$$

A definição de que uma região é interna ao *canvas* é feita através da aplicação das seguintes assertivas para cada região r.

$$r{:}x^{ini} \geq canvas{:}x^{ini} \ \wedge \ r{:}x^{end} \leq canvas{:}x^{end} \tag{3.11}$$
$$r{:}y^{ini} \geq canvas{:}y^{ini} \ \wedge \ r{:}y^{end} \leq canvas{:}y^{end} \tag{3.12}$$

As assertivas 3.1 a 3.12 definem regras básicas para a construção das regiões associadas a itens ou *containers*. Tais assertivas, portanto, fazem parte do conjunto de restrições com prioridade *core*.

Um *container* não tipado define uma coleção de itens ou *containers* internos a ele. Portanto, seja c um *container* podemos representá-lo como um conjunto de regiões r que representam os elementos internos ao *container*. Dessa forma, para todo r dentro de c, as assertivas a seguir são aplicadas:

$$c{:}x^{ini} \leq r{:}x^{ini} \tag{3.13}$$
$$c{:}y^{ini} \leq r{:}y^{ini} \tag{3.14}$$
$$c{:}x^{end} \geq r{:}x^{end} \tag{3.15}$$
$$c{:}y^{end} \geq r{:}y^{end} \tag{3.16}$$

Como mencionado anteriormente, StyLe provê três tipos de restrições espaciais predefinidas: *align*, *distribute* e *size*.

A restrição *align* é uma relação espacial de alinhamento dos elementos espaciais. Formalmente, uma restrição *align* é representada pela tupla (I, d, Δ, K), onde I é o conjunto de todos os pontos de interface, $d \in \{offset, factor\}$ indica se será usado um fator incremental ou multiplicativo usado para o distanciamento entre pontos de interface, $\Delta \in N$ é o valor do fator de deslocamento, e $K \in \{previous, first\}$ indica se a relação será aplicada tomando como base sempre o primeiro ponto de interface da relação ou o anterior. A forma geral da restrição *align*, juntamente com seu mapeamento para fórmulas SMT são como segue:

$$(\{i_1, i_2, \ldots, i_n\}, offset, \Delta, previous)$$
$$i_2 = i_1 + \Delta \wedge i_3 = i_2 + \Delta \wedge \ldots \wedge i_n = i_{n-1} + \Delta \tag{3.17}$$

$$(\{i_1, i_2, \ldots, i_n\}, offset, \Delta, first)$$
$$i_2 = i_1 + \Delta \wedge i_3 = i_1 + \Delta \wedge \ldots \wedge i_n = i_1 + \Delta \tag{3.18}$$

$$(\{i_1, i_2, \ldots, i_n\}, factor, \Delta, previous)$$
$$i_2 = i_1 * \Delta \wedge i_3 = i_2 * \Delta \wedge \ldots \wedge i_n = i_{n-1} * \Delta \tag{3.19}$$

$$(\{i_1, i_2, \ldots, i_n\}, factor, \Delta, first)$$
$$i_2 = i_1 * \Delta \wedge i_3 = i_1 * \Delta \wedge \ldots \wedge i_n = i_1 * \Delta \tag{3.20}$$

A restrição *distribute* é uma relação espacial de distribuição dos elementos espaciais em um espaço específico. Formalmente, uma restrição *distribute* é representada pela tripla (E, d, Δ), onde E é o conjunto de elementos espaciais que fazem parte da relação, $d \in \{horizontal, vertical\}$ indica a direção da distribuição, e $\Delta \in N$ é o fator de deslocamento.

A forma geral da restrição *distribute*, juntamente com seu mapeamento para fórmulas SMT são como segue:

$$(\{e_1, e_2, \ldots, e_n\}, horizontal, \Delta)$$
$$e_2{:}x^{ini} = e_1{:}x^{end} + \Delta \wedge e_3{:}x^{ini} = e_2{:}x^{end} + \Delta \wedge \ldots \wedge e_n{:}x^{ini} = e_{n-1}{:}x^{end} + \Delta \tag{3.21}$$

$$(\{e_1, e_2, \ldots, e_n\}, vertical, \Delta)$$
$$e_2{:}y^{ini} = e_1{:}y^{end} + \Delta \wedge e_3{:}y^{ini} = e_2{:}y^{end} + \Delta \wedge \ldots \wedge e_n{:}y^{ini} = e_{n-1}{:}y^{end} + \Delta \tag{3.22}$$

A restrição *size* é uma relação espacial aplicada sobre pontos de interface que representam as dimensões dos elementos. Formalmente, a restrição *size* é representada pela tupla (E, S, d, Δ), onde E é o conjunto de elementos espaciais que fazem parte da relação, $S \in \{width, height, size\}$ é o conjunto das dimensões possíveis, $d \in \{offset, factor\}$ indica um valor incremental ou multiplicativo, e $\Delta \in N$ é o fator a ser considerado na definição da dimensão. A forma geral da restrição *size*, juntamente com seu mapeamento para fórmulas SMT são como segue:

$$(\{e_1, e_2\}, width, offset, \Delta)$$
$$e_2{:}x^{wid} = e_1{:}x^{wid} + \Delta \tag{3.23}$$

$$(\{e_1, e_2\}, width, factor, \Delta)$$
$$e_2{:}x^{wid} = e_1{:}x^{wid} * \Delta \tag{3.24}$$

$$(\{e_1, e_2\}, height, offset, \Delta)$$
$$e_2{:}y^{hei} = e_1{:}y^{hei} + \Delta \tag{3.25}$$

$$(\{e_1, e_2\}, height, factor, \Delta)$$
$$e_2{:}y^{hei} = e_1{:}y^{hei} * \Delta \tag{3.26}$$

$$(\{e_1, e_2\}, size, offset, \Delta)$$
$$e_2{:}x^{wid} = e_1{:}x^{wid} + \Delta \wedge e_2{:}y^{hei} = e_1{:}y^{hei} + \Delta \tag{3.27}$$

$$(\{e_1, e_2\}, size, factor, \Delta)$$
$$e_2{:}x^{wid} = e_1{:}x^{wid} * \Delta \wedge e_2{:}y^{hei} = e_1{:}y^{hei} * \Delta \tag{3.28}$$

Além das restrições espaciais predefinidas, a linguagem STyLe também dá suporte à construção de restrições personalizadas através dos conectores de restrição. Os conectores de restrição são traduzidos para um conjunto de cláusulas que serão resolvidas pelo *solver*. O EBNF (extended Backus-Naur form) para o conector de restrição é como segue:

```
1  ⟨constraintConnector⟩ := ⟨assessmentStatement⟩ | ⟨compoundStatement⟩;
2  ⟨compoundStatement⟩ = ⟨operator⟩,⟨StatementList⟩;
3  ⟨StatementList⟩ = ⟨compoundStatement⟩ ⟨StatementList⟩ | ⟨assessmentStatement⟩ ⟨StatementList⟩ | ⟨
     compoundStatement⟩ | ⟨assessmentStatement⟩;
4  ⟨operator⟩ = ''and'' | ''or'';
5  ⟨assessmentStatement⟩ = ⟨attributeStatement⟩ ⟨Comparator⟩ ⟨attributeStatement⟩ | ⟨attributeStatement⟩ ⟨
     Comparator⟩ ⟨value⟩;
6  ⟨attributeStatement⟩ = ''top'' | ''left'' | ''right'' | ''bottom'' | ''center'' | ''middle'' | ''width
     '' | ''height'' | ''size'';
7  ⟨Comparator⟩ = ''eq'' | ''ne'' | ''lte'' | ''gte'' | ''lt'' | ''gt'';
8  ⟨value⟩ = ⟨digit⟩⟨value⟩ | ⟨digit⟩;
9  ⟨digit⟩ = ''0'' | ''1'' | ''2'' | ''3'' | ''4'' | ''5'' | ''6'' | ''7'' | ''8'' | ''9'';
```

O mapeamento de um conector de restrição para uma fórmula SMT é direto, representando cada ⟨assessmentStatement⟩ como uma fórmula atômica SMT, seja uma desigual-

dade ou fórmula booleana, e combinando tais fórmulas de acordo com o(s) operador(es) definido(s) pelo(s) elemento(s) ⟨compoundStatement⟩ presente(s) no conector.

Por fim, STyLe suporta a definição de *containers* tipados. Cada *container* com um tipo predefinido é mapeado para um conjunto de fórmulas SMT que definem a distribuição dos elementos internos ao *containers* de acordo com o tipo definido. As fórmulas a seguir apresentam um exemplo de *container* do tipo *grid* tendo os elementos e_1, e_2 e e_3 como elementos internos.

Inicialmente todos os elementos internos são dimensionados de acordo com o tamanho do *container* (representado pelo elemento `grid`), o número de linhas (nr), colunas (nc), espaçamento vertical (vs) e horizontal (hs) definidos para o *container* como segue:

$$e_i{:}x^{wid} = (grid{:}x^{wid} - (nc-1) * hs)/nc \wedge e_i{:}y^{hei} = (grid{:}y^{hei} - (nr-1) * vs)/nr \tag{3.29}$$

Para o primeiro elemento (e_1) é aplicada a fórmula a seguir:

$$pos(1) = 1 \wedge e_1{:}x^{ini} = grid{:}x^{ini} \wedge e_1{:}y^{ini} = grid{:}y^{ini} \tag{3.30}$$

onde a variável $pos(1)$ é usada para auxiliar no posicionamento dos elementos nas células do *grid*.

As fórmulas a seguir posicionam os elementos e_2 e e_3 de acordo com o elemento anterior, levando em consideração se o mesmo está sendo apresentado ou não. Caso um elemento anterior não esteja sendo apresentado, o elemento em questão ocupa o seu lugar.

$$
\begin{aligned}
&(e_1{:}s^{occ} \wedge \\
&\quad ((pos(1) < nc \wedge pos(2) = pos(1) + 1 \wedge e_2{:}x^{ini} = e_1{:}x^{end} + hs \wedge e_2{:}y^{ini} = e_1{:}y^{ini}) \vee \\
&\quad (pos(1) \geq nc \wedge pos(2) = 1 \wedge e_2{:}x^{ini} = grid{:}x^{ini} \wedge e_2{:}y^{ini} = e_1{:}y^{end} + vs))) \\
&\vee (\neg e_1{:}s^{occ} \wedge pos(2) = pos(1) \wedge e_2{:}x^{ini} = e_1{:}x^{ini} \wedge e_2{:}y^{ini} = e_1{:}y^{ini})
\end{aligned} \tag{3.31}
$$

$$
\begin{aligned}
&(e_2{:}s^{occ} \wedge \\
&\quad ((pos(2) < nc \wedge pos(3) = pos(2) + 1 \wedge e_3{:}x^{ini} = e_2{:}x^{end} + hs \wedge e_3{:}y^{ini} = e_2{:}y^{ini}) \vee \\
&\quad (pos(2) \geq nc \wedge pos(3) = 1 \wedge e_3{:}x^{ini} = grid{:}x^{ini} \wedge e_3{:}y^{ini} = e_2{:}y^{end} + vs))) \\
&\vee (\neg e_2{:}s^{occ} \wedge pos(3) = pos(2) \wedge e_3{:}x^{ini} = e_2{:}x^{ini} \wedge e_3{:}y^{ini} = e_2{:}y^{ini})
\end{aligned} \tag{3.32}
$$

A fórmula a seguir é uma generalização das anteriores para o *grid* dinâmico.

$$
\begin{aligned}
&(e_i{:}s^{occ} \wedge \\
&((pos(e_i) < nc \wedge pos(e_{i+1}) = pos(e_i) + 1 \wedge e_{i+1}{:}x^{ini} = e_i{:}x^{end} + hs \wedge e_{i+1}{:}y^{ini} = e_i{:}y^{ini}) \vee \\
&(pos(e_i) \geq nc \wedge pos(e_{i+1}) = 1 \wedge e_{i+1}{:}x^{ini} = grid{:}x^{ini} \wedge e_{i+1}{:}y^{ini} = e_i{:}y^{end} + vs))) \\
&\vee (\neg e_i{:}s^{occ} \wedge pos(e_{i+1}) = pos(e_i) \wedge e_{i+1}{:}x^{ini} = e_i{:}x^{ini} \wedge e_{i+1}{:}y^{ini} = e_i{:}y^{ini})
\end{aligned} \tag{3.33}
$$

Para validar a linguagem SᵀyLᵉ, a linguagem de autoria para aplicações multimídia interativas NCL foi estendida. O próximo capítulo descreve essa extensão e apresenta exemplos da utilização de SᵀyLᵉ com NCL.

Capítulo 4

Extensão de NCL para usar STyLe

Este capítulo expõe como documentos NCL devem ser estendidos para que possam usar *templates* STyLE para fornecer leiaute espacial adaptativo e dinâmico. Para mostrar o funcionamento da extensão proposta, um exemplo de um documento NCL utilizando um *template* STyLE foi especificado. A Figura 4.1 mostra o leiaute espacial criado para este documento.

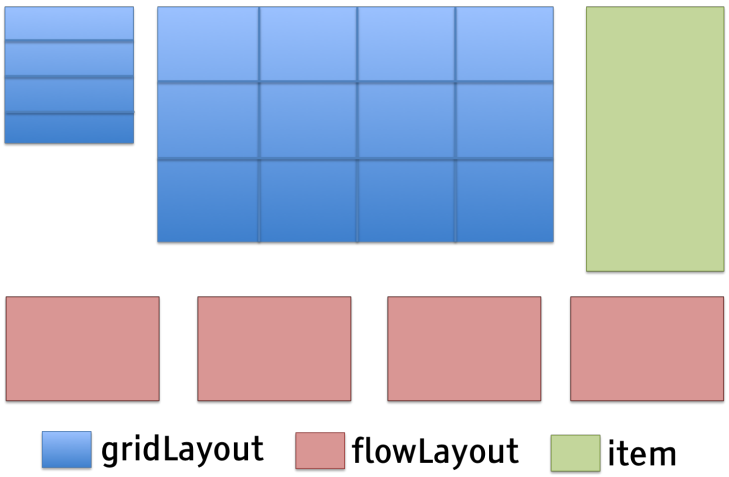

Figura 4.1: Leiaute Espacial do Exemplo

O código do *template* STyLE usado para criar o exemplo é apresentado na Listagem 4.1.

```
1  <layout id="LayoutAppExample">
2    <head>
3      <constraintConnectorBase>
4        <importBase alias="const" documentURI="constraint.xml"/>
5      </constraintConnectorBase>
6    </head>
7    <body>
8      <container id="menu" type="gridLayout" focusIndex="1">
9        <format top="0" left="0" width="120" height="240" columns="1" rows="4"
10         vspace="0" hspace="0"/>
11     </container>
12
13     <container id="grid" type="gridLayout" focusIndex="1">
```

```
14        <format width="490" height="300" columns="4"
15          rows="3" vspace="0" hspace="0"/>
16      </container>
17
18      <item id="info" width="190" height="380"/>
19
20      <container id="videos" type="flowLayout" focusIndex="1">
21        <format width="860" height="100" hspace="20"/>
22        <item id="item1" width="200" height="100"/>
23      </container>
24
25      <spatialConstraint id="dist1" type="distribute" direction="horizontal">
26        <bind component="menu"/>
27        <bind component="grid"/>
28        <bind component="info"/>
29      </spatialConstraint>
30
31      <spatialConstraint id="align1" type="align" >
32        <bind component="menu" interface="top"/>
33        <bind component="grid" interface="top"/>
34        <bind component="info" interface="top"/>
35      </spatialConstraint>
36
37      <spatialConstraint id="align2" xconnector="alignLeftBottomTop">
38        <bind role="left1" component="menu" interface="left"/>
39        <bind role="left2" component="videos" interface="left"/>
40        <bind role="base" component="videos" interface="top"/>
41        <bind role="base50" component="info" interface="bottom"/>
42      </spatialConstraint>
43    </body>
44  </layout>
```

Listing 4.1: Template de layout StyLe

Este código determina três *containers* e um *item*. O *container menu*, na esquerda, e o *container grid*, no centro, instanciam um modelo de leiaute do tipo *grid* e o *container vídeos*, na parte inferior, instancia um modelo do tipo *flow*. Como os *containers* possuem um tipo, não é necessário criar restrições espaciais adicionais dentro deles.

Duas restrições espaciais predefinidas e um conector de restrição foram usados para criar o leiaute resultante. A disposição dos *containers menu*, *grid* e do *item info* (esquerda, centro e direita, respectivamente) é determinada com o uso de uma restrição do tipo *distribute*. Em seguida, esses elementos são alinhados utilizando-se uma restrição do tipo *align*, usando como pontos de interface o atributo *top*. Outra restrição, usando o conector de restrição definido na Listagem 3.9, é utilizada para (i) alinhar os *containers menu* e *vídeo*, usando o valor *left* para o ponto de interface, e (ii) alinhar o ponto de interface *bottom* do *container vídeo* com o ponto de interface *top* do *item info*.

É importante notar que o conector de restrição define o atributo *offset* de 50 pixels ao valor definido no papel *base50*. Isso faz com que o topo do *container videos* seja igual a base do item *info* mais 50 pixels.

NCL [9] foi estendida para usar *templates* StyLe. A extensão proposta foi especificada em XML Schema e está disponível no Apêndice C.

Para que um documento NCL possa utilizar um *template* STyLe, é necessário declarar um elemento *layoutBase* dentro do elemento *head* que já faz parte de NCL. *layoutBase* é utilizado para definir informações importantes sobre o *template* utilizado, como sua localização. Seus atributos são: *alias*, utilizado para definir um pseudônimo que será utilizado no documento NCL como prefixo para se referir aos elementos importados (no formato "pseudônimo#id_do_elemento_importado") e *documentURI* que define a localização e o nome do arquivo que contém a declaração do *template* a ser importado.

O autor NCL pode estabelecer a relação entre os objetos de mídia e os elementos de leiaute declarados no *template* STyLe, sejam eles *itens* ou *containers*, de forma explícita ou implícita.

Para definir de forma explícita a relação entre objeto de mídia e um elemento de leiaute, é necessário definir um novo atributo, denominado *layout*, no elemento *mídia* de NCL. O valor desse novo atributo deve ser igual ao valor do atributo *alias*, definido no elemento *layoutBase*, concatenado com o valor do identificador de um elemento do *template* STyLe, como mostrado nas linhas 9 – 32 da Listagem 4.2. Para alguns componentes do leiaute, o objeto de mídia ainda precisa declarar um atributo *item*, que define a qual item, dentro do componente, aquele objeto está relacionado.

Para estabelecer a relação de maneira implícita, o atributo *layout* deve ser adicionado a um contexto em NCL (elemento *context*). Quando isso ocorre, todas os objetos de mídia declarados dentro do contexto herdarão o atributo *layout*, criando a relação implicitamente. O mesmo pode ser feito com elemento *body* da linguagem NCL, já que esse elemento é um caso particular de contexto. Um exemplo pode ser visualizado na Listagem 4.4.

A Listagem 4.2 mostra um fragmento do código NCL para o documento que utiliza o *template* STyLe mostrado na Figura 4.1.

```
1  <ncl>
2    <head>
3      <layoutBase>
4        <importBase alias="layBase" documentURI="LayoutAppExample.xml"/>
5      </layoutBase>
6      ...
7    </head>
8    <body>
9      <media id="menu1" src="menu1.png" layout="layBase#menu"/>
10     <media id="menu2" src="menu2.png" layout="layBase#menu"/>
11     <media id="menu3" src="menu3.png" layout="layBase#menu"/>
12     <media id="menu4" src="menu4.png" layout="layBase#menu"/>
13
14     <media id="image1" src="arrow.jpg" layout="layBase#grid"/>
15     <media id="image2" src="flash.jpg" layout="layBase#grid"/>
16     <media id="image3" src="gameofthrones.jpg" layout="layBase#grid"/>
17     <media id="image4" src="vikings.jpg" layout="layBase#grid"/>
18     <media id="image5" src="bettercallsoul.jpg" layout="layBase#grid"/>
19     <media id="image6" src="bigbang.jpg" layout="layBase#grid"/>
```

```
20      <media id="image7" src="twohalfman.jpg" layout="layBase#grid"/>
21      <media id="image8" src="mentalist.jpg" layout="layBase#grid"/>
22      <media id="image9" src="breakingbad.jpg" layout="layBase#grid"/>
23      <media id="image10" src="goodwife.jpg" layout="layBase#grid"/>
24      <media id="image11" src="houseofcards.jpg" layout="layBase#grid"/>
25      <media id="image12" src="lietome.jpg" layout="layBase#grid"/>
26
27      <media id="image13" src="info.jpg" layout="layBase#info"/>
28
29      <media id="video1" src="video1.mp4" layout="layBase#videos" item="item1"/>
30      <media id="video2" src="video2.mp4" layout="layBase#videos" item="item1"/>
31      <media id="video3" src="video3.mp4" layout="layBase#videos" item="item1"/>
32      <media id="vidoe4" src="video4.mp4" layout="layBase#videos" item="item1"/>
33      ...
34    </body>
35  </ncl>
```

Listing 4.2: Código do exemplo NCL

Nesse exemplo, é possível perceber que os quatro primeiros objetos de mídia (menu1 a menu4) estão relacionados com o *container menu* definido no *template* StyLe *LayoutAppExample.xml*. Os próximos 12 objetos de mídia (image1 - image12) estão relacionados com o *container grid*. Já o objeto de mídia identificado por *image13* faz referência não a um *container* mas a um *item* do *template*, denominado info, e os objetos de mídia video1, video2, video3 e video4 não somente definem o *container* ao qual estão relacionados, mas também o *item* dentro do *container*.

4.1 Processamento do Documento NCL que usa STyLe

Para executar o documento NCL estendido, é necessário processar o *template* junto com o documento NCL estendido para transformá-lo em um documento NCL padrão, como mostrado na Figura 4.2.

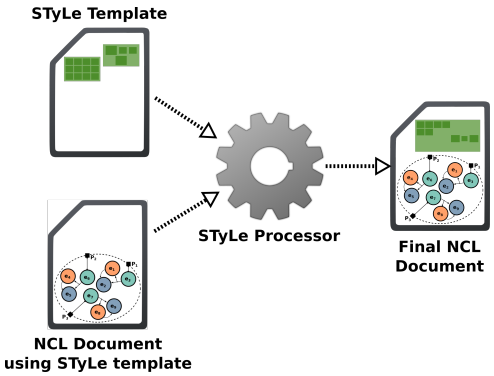

Figura 4.2: Processamento do documento NCL com leiautes StyLe

Um *template* StyLe define *itens* e/ou *containers* para representar regiões espaciais onde grupos de objetos de mídia serão posicionados. O documento que usa um dado *template* espacial associa seus objetos de mídia aos *itens* e/ou *containers* do *template* através

de rótulos (atributo *layout*). Relações de restrição que relacionam componentes espaciais são declaradas no *template* StyLe. Em tempo de processamento, esses componentes dão origem a regiões NCL que serão ocupadas por objetos de mídia declarados no documento que utiliza o *template* espacial.

Quando um documento NCL utiliza um *template* StyLe, o documento final NCL é gerado depois do processamento do *template*. Cada objeto de mídia no documento NCL possui um rótulo com a identificação de um elemento de leiaute (*container* ou *item*). Portanto, no processamento do leiaute, as características de apresentação são criadas de acordo com o número de objetos de mídia associados a cada elemento de leiaute. Depois desse passo, o documento final NCL tem o conteúdo original do documento mais a especificação do posicionamento, tamanho e navegação definidos no *template* de leiaute. A Figura 4.3 exibe o leiaute do documento NCL final gerado após o processamento do documento NCL estendido exibido na Listagem 4.2 e o *template* StyLe utilizado.

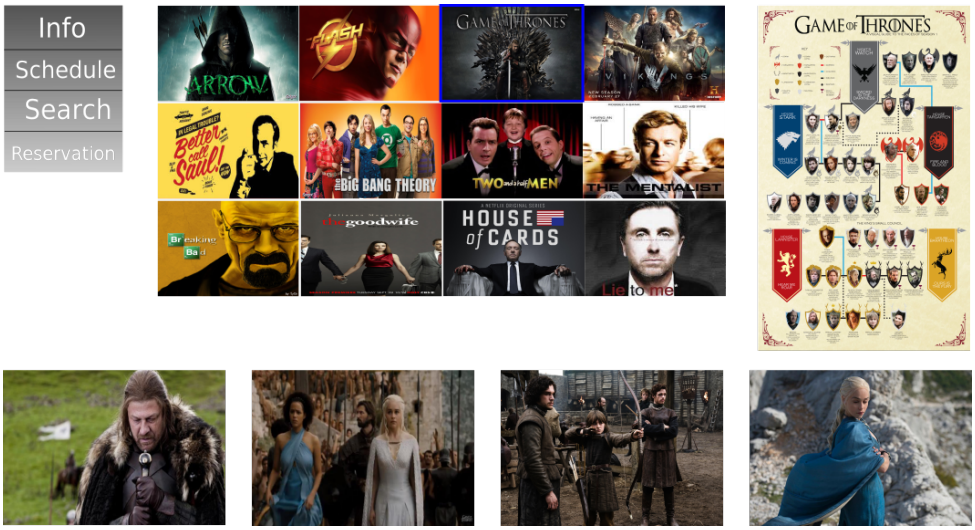

Figura 4.3: Leiaute Espacial do Exemplo

4.2 Uso de Leiautes StyLe em Contextos NCL

A utilização do elemento *contexto* na linguagem NCL sugere uma ideia de aninhamento de elementos. Este elemento é utilizado para agrupar objetos (de mídia ou de contexto), melhorando a estruturação do documento. Assim como *contextos* em NCL, StyLe fornece a possibilidade de aninhamento de leiautes.

A Listagem 4.3 mostra o fragmento de um código escrito em NCL, para ilustrar o uso de *containers* aninhados no documento multimídia que usa o *template* de leiaute StyLe.

```
<ncl>
  ...
  <media id="image1" src="menu1.jpg" layout="f1" item="D"/>
  <media id="image2" src="flash.jpg" layout="f1.g1"/>
  <media id="image3" src="gameofthrones.jpg" layout="f1.g1"/>
  <media id="image4" src="vikings.jpg" layout="f1.g1"/>
  <media id="image5" src="bettercallsoul.jpg" layout="f1.g1"/>
  <media id="image6" src="bigbang.jpg" layout="f1.g1"/>
  <media id="image7" src="twohalfman.jpg" layout="f1.g1"/>
  <media id="image1" src="menu2.jpg" layout="f1" item="D"/>
    ...
</ncl>
```

Listing 4.3: Código do exemplo NCL com aninhamento de leiaute

Esta listagem faz referência ao *template* de leiaute STyLe mostrado na Listagem 3.12. Neste exemplo, um *container* do tipo *gridLayout* é aninhado a um *container* do tipo *flowLayout*. Como mencionado anteriormente, sempre que um objeto de mídia faz referência a um elemento de um leiaute aninhado, é necessário definir explicitamente a hierarquia dos leiautes da seguinte forma: <id_container_externo>.<id_container_interno>.

Os exemplos utilizados até então mostram sempre um objeto de mídia referenciando diretamente o componente de leiaute. Entretanto, é possível que um elemento *contexto* da linguagem NCL, usado para criar uma estrutura mais adequada a algumas aplicações, referencie um componente do leiaute. Nesse caso, as mídias declaradas dentro desse contexto irão referenciar o componente do leiaute indiretamente e serão associadas a esse leiaute quando o mesmo for criado. A Listagem 4.4 mostra um fragmento de código NCL para representar essa facilidade.

```
<ncl>
  ...
  <context id="ctx_1" layout="f1">
      ...
    <media id="image1" src="menu1.jpg" item="D"/>
      <media id="image2" src="menu2.jpg" item="D"/>
    ...
  </context>
  ...
</ncl>
```

Listing 4.4: Contexto referenciando componente do leiaute

Caso o componente de leiaute referenciado pelo contexto tenha componentes internos, como o exemplo descrito na Seção 3.2 do Capítulo 3, então, as mídias relacionadas aos componentes internos deverão declarar essa associação explicitamente através do atributo *layout*, como mostrado na Listagem 4.5.

```
<ncl>
  ...
  <context id="ctx_1" layout="f1">
      ...
    <media id="image1" src="menu1.jpg" item="D"/>
    <media id="image2" src="flash.jpg" layout="g1"/>
      <media id="image3" src="gameofthrones.jpg" layout="g1"/>
```

```
8       <media id="image4" src="vikings.jpg" layout="g1"/>
9       <media id="image5" src="bettercallsoul.jpg" layout="g1"/>
10      <media id="image6" src="bigbang.jpg" layout="g1"/>
11      <media id="image7" src="twohalfman.jpg" layout="g1"/>
12      <media id="image8" src="menu2.jpg" item="D"/>
13    ...
14  </context>
15  ...
16 </ncl>
```

Listing 4.5: Contexto referenciando componentes aninhados no leiaute

Nesse caso, não há a necessidade de explicitar a hierarquia entre os *containers* (layout = "f1.g1") como mostrado na Listagem 4.3, porque a própria estrutura do contexto já o faz.

Embora NCL tenha usado StyLe diretamente para definir o leiaute espacial da aplicação, é possível que documentos NCL usem *templates* de composição, escritos em XTemplate por exemplo, e que esses *templates* de composição façam referência a leiautes StyLe. Neste caso, um objeto de mídia NCL deveria referenciar um componente genérico no *template* de composição através do atributo *xlabel* e o componente genérico referenciaria um *item* ou *container* no leiaute StyLe através do atributo *layout*. Documentos NCL que usam XTemplate ou StyLe devem ser processados para serem executados em uma implementação padrão NCL.

A linguagem XTemplate foi estendida para poder utilizar *templates* StyLe. O capítulo seguinte discorre sobre as características de XTemplate 4.0 e descreve como a extensão foi feita.

Capítulo 5

XTemplate 4.0

Como definido em [12], *templates* de composição especificam componentes genéricos e os relacionamentos entre estes sem identificar necessariamente quem são os objetos de mídia a serem relacionados. A composição hipermídia que utiliza um *template* fica então responsável por especificar tais objetos de mídia, referenciando os componentes genéricos do *template*. Em [4], os autores caracterizam um *template* de composição como um documento definido por meio de composições hipermídia com sua especificação feita de forma incompleta. O preenchimento dessas lacunas deve ser realizado pelo documento hipermídia que faz uso desse *template*. Por isso, esse documento pode ser chamado de documento de preenchimento. Este aspecto, confere as linguagens baseadas em *templates* a característica de diminuir o esforço de autoria e mitigar a ocorrência de erros.

A primeira linguagem proposta para criação de *templates* de composição hipermídia foi XTemplate. A linguagem possui 3 versões anteriores, a versão original [12], a versão 2.1 [30] e a versão 3.0 [4].

Um dos principais motivos para o desenvolvimento de XTemplate 3.0 foi o fato das duas versões anteriores não estarem atualizadas com a versão 3.0 da linguagem NCL. Algumas facilidades foram incorporadas à versão 3.0 como: a possibilidade de especificar definições de apresentações e a atribuição de valores a variáveis e parâmetros dos elos durante o processamento do *template*. Todas as outras facilidades encontradas nas versões anteriores da linguagem foram mantidas na versão 3.0.

Entretanto, algumas limitações ainda podiam ser encontradas. Por isso, esta tese propõe XTemplate 4.0 [14], onde são incluídas as facilidades de definição de leiautes adaptativos e de *templates* aninhados.

A Tabela 5.1 apresenta uma rápida comparação entre XTemplate 4.0 e os trabalhos relacionados que são baseados em *templates* apresentados no Capítulo 2, levando em consideração as novas facilidades apresentadas na versão 4.0.

Tabela 5.1: Comparação entre linguagens baseadas em *templates*

	XTemplate4	XTemplate3	TAL	LimSee 3	STAMP
Aninhamento de *templates*	X	-	X	-	-
Leiaute adaptativo	X	-	-	-	X
Leiaute dinâmico	X	-	-	-	-

Apesar da proposta de XTemplate ser independente da linguagem de autoria de documentos multimídia que usam *templates*, o processador XTemplate 4.0 foi implementado para a linguagem NCL.

5.1 Aninhamento de templates

Templates de composição definem estruturas genéricas que são herdadas por composições que os usam dentro de um documento hipermídia. Isso significa que a especificação de sincronização fornecida pelo *template* será utilizada pelo nós filhos de uma dada composição. Suponha que uma composição $C1$ declare três elementos $e1$, $e2$ e $e3$ dentro dela. A composição c1 usa um *template* $T1$, que define os componentes $G1$ e $G2$ juntamente com uma especificação de sincronização relacionando ambos os componentes, como ilustrado na Figura 5.1, onde círculos representam objetos de mídia ou componentes de um *template*, elipses tracejadas representam composições e elipses sólidas representam *templates*. Uma seta entre uma composição e um *template* indica o *template* usado por uma composição, enquanto linhas tracejadas indicam a associação de um objeto de mídia a um componente de um *template*.

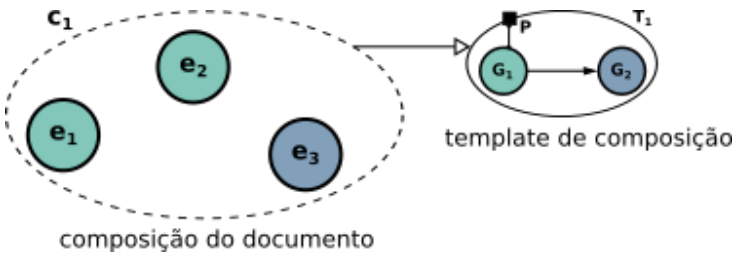

Figura 5.1: Composição C1 usando template T1

Suponha, por exemplo, que o elemento $e3$ na Figura 5.1 seja uma composição aninhada em $C1$. Se o autor do documento quer introduzir semântica dentro de $e3$, precisará relacionar e3 com outro *template* $T2$. Esta abordagem é mostrada na Figura 5.2.

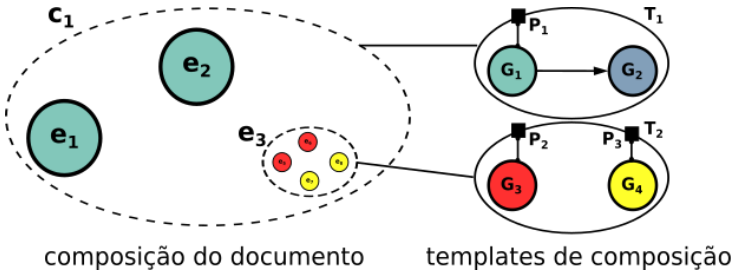

Figura 5.2: Aninhamento de composições usando templates

Seguindo essa abordagem, o autor do exemplo da Figura 5.2 deve associar explicitamente (um a um) os *templates* $T1$ e $T2$ às composições $C1$ e $C2$, respectivamente. Uma desvantagem deste cenário é que o autor deve conhecer os detalhes tanto de $T1$ quanto de $T2$, bem como a maneira de utilizá-los em conjunto. Levando-se em consideração que autores que utilizam *templates*, geralmente, não possuem grande conhecimento sobre as linguagens utilizadas para a especificação de *templates* (neste caso XTemplate), usar mais de um *template* no mesmo documento pode aumentar a complexidade do processo de autoria.

Neste trabalho, aninhamento de *templates* foi definida como uma facilidade onde os componentes de um *template* podem ser associados a outros *templates*. A Figura 5.3 estende o exemplo da Figura 5.2, onde o componente $G2$ é associado ao *template* $T2$. Dessa forma o autor não precisa mais fazer a referência explícita entre $e3$ e $T2$.

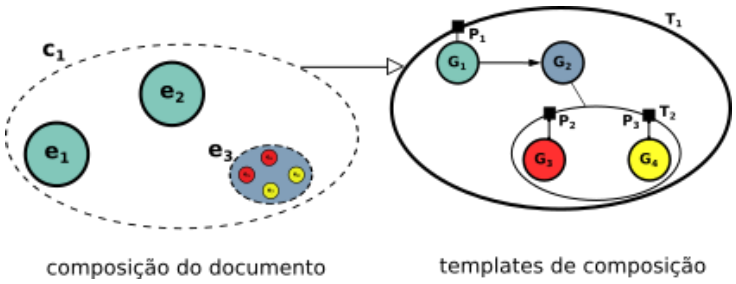

Figura 5.3: Componente de template usando templates aninhados

Para assegurar a comunicação entre dois *templates* (serão chamados de *template* externo e interno), é necessário estabelecer uma interface de comunicação. Essa interface é definida pelo elemento *port*. O *template* interno deve declarar em seu vocabulário um ou mais elementos *port* para funcionar como sua interface de comunicação. No *template* externo, o mesmo elemento *port* é declarado dentro do componente que usa o *template*

interno. Esta abordagem é ilustrada na Figura 5.4, onde quadrados preenchidos representam elementos *port*. Definir a interface dos *templates* aninhados é importante para manter a propriedade de composicionalidade para *templates*.

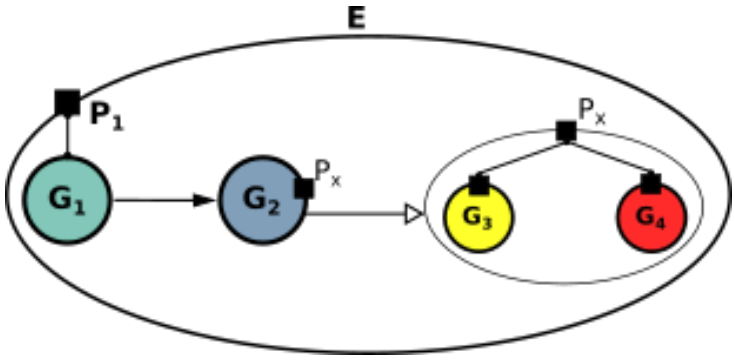

Figura 5.4: Interface de comunicação no aninhamento de templates

O aninhamento de *templates* é utilizado não só como uma forma de evidenciar o uso de *templates* em conjunto com um *template* principal, mas também como uma forma de explicitar a relação entre *templates* para ferramentas gráficas que usem *templates*. Com essa relação explicitamente indicada, uma dada ferramenta pode analisar um *template* e, no caso de algum(ns) de seus componentes ser(em) associado(s) a outro(s) *template*(s), a ferramenta pode automaticamente recuperar o *template* interno e apresentar ao autor informações de como preencher ambos os *templates* em conjunto. Essa ação torna os *templates* internos transparentes para o autor que utiliza a ferramenta. O editor NEXT [46] é um exemplo de ferramenta de autoria gráfica que suporta a utilização de *templates* criados com XTemplate 3.0. Como XTemplate 3.0 não permite o aninhamento de *templates*, o autor deve interagir com o editor para utilizar cada *template* interno ou externo separadamente. A atualização da ferramenta NEXT para suportar XTemplate 4.0 é um trabalho futuro.

Quando um elemento *component* define um atributo *type* com valor *template*, tem-se o aninhamento de *templates* em XTemplate 4.0. Deste modo, um elemento *component* com um tipo *xtemplate* contém a mesma estrutura do *template* interno ao qual ele referencia. O *template* interno é indicado junto com o seu tipo como *xtemplate/alias*, onde alias é uma identificação única de um determinado *template* na base de *templates*. Para possibilitar a utilização de *templates* aninhados, o elemento *component* declara elementos *port*, filhos de *component*, que são usados como interface de comunicação entre o *template* interno e o *template* externo. Um exemplo de *templates* aninhados é mostrado na Seção 5.3.

É possível definir mais de um nível de aninhamento entre os *templates*. O processador de *templates* irá varrer o aninhamento de forma recursiva. Cada componente definido com o tipo *xtemplate* é processado do elemento mais interno até o elemento mais externo.

5.2 Leiautes Adaptativos

Assim como na extensão para NCL, XTemplate 4.0 utiliza o elemento *layoutBase* para importar os *templates* de leiaute. Uma declaração de um *template* de leiaute é feita através do elemento *layout*. O *template* STyLE possui as mesmas definições já apresentadas no Capítulo 3.

É importante estabelecer como os objetos de mídia de um documento que usa o *template* do XTemplate serão relacionados às regiões construídas por cada leiaute. O documento que utiliza um *template*, gerado com a linguagem XTemplate 4.0, deverá ser processado para que possa ser transformado em um documento NCL 3.0 completo. Um componente de um *template* pode indicar qual leiaute irá utilizar através do atributo *layout*, como mostrado na Listagem 5.1. Um documento NCL que usa um *template* deverá definir seus objetos de mídia e cada um deles deverá especificar o atributo *xlabel*, indicando a qual componente do *template* se refere, como já era feito nas versões anteriores de XTemplate, conforme a Listagem 5.2. Por meio dessa referência ao componente do *template*, pode-se definir sua posição no leiaute.

```
1 ...
2 <component xlabel="imagens" layout="layBase#G1"/>
3 <component xlabel="botoes" layout="layBase#G2"/>
4 ...
```

Listing 5.1: Exemplo de utilização do elemento layout em XTemplate 4.0

```
1 ...
2 <media id="Fotos" src="foto1.jpg" xlabel="imagens"/>
3 <media id="Controle" src="botao1.jpg" xlabel="botoes"/>
4 ...
```

Listing 5.2: Exemplo de documento NCL que usa um template

As mídias serão associadas às regiões dos leiautes na ordem em que forem declaradas no documento NCL, isto é, a primeira mídia declarada no documento NCL ocupará a primeira região definida para o leiaute especificado pelo componente relacionado a ela. As regiões são criadas automaticamente pelo processador de leiautes, conforme a quantidade de objetos de mídia com um mesmo *xlabel* no documento que usa o *template*.

Continua sendo possível especificar regiões sem a definição de nenhum tipo de leiaute. Isso pode ser feito através da declaração de bases de descritores diretamente no *template*,

como ocorre na versão 3.0 da linguagem. Manter essa facilidade garante a flexibilidade de criar regiões de qualquer tamanho e em qualquer lugar, não diminuindo a expressividade da linguagem. Além disso, também é possível criar regiões independentes dos modelos de leiautes disponíveis no *template* de leiautes. Para isso, basta o autor declarar um elemento *region* no *template* de leiaute do mesmo modo como ocorre em NCL. Ao ser processado, esse elemento é exportado para o documento final sem qualquer alteração.

5.3 Exemplo de utilização do XTemplate 4.0

Para demonstrar as novas facilidades fornecidas por XTemplate 4.0, os *templates* "quiz.xml" e "screen.xml" apresentados em [4] foram modificados. A Figura 5.5 mostra a visão estrutural das versões originais dos *templates* "quiz.xml" e "screen.xml", retirada de [4].

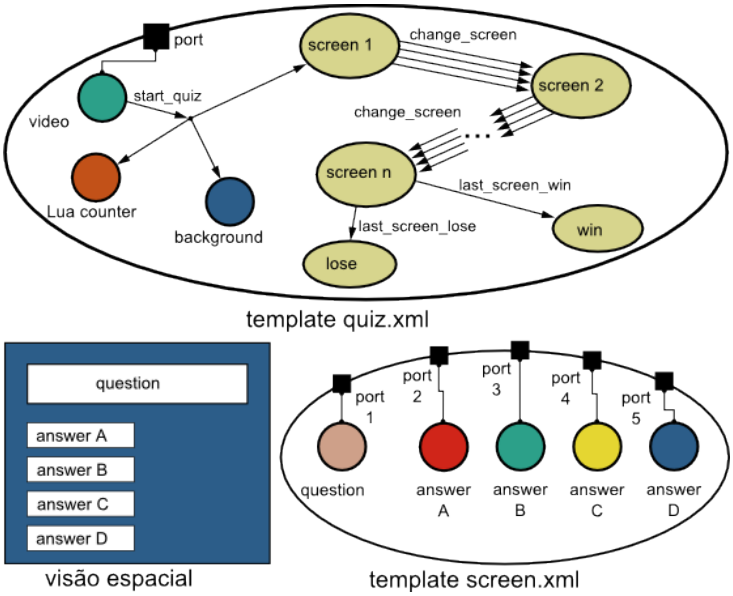

Figura 5.5: Versão original dos templates quiz.xml e screen.xml

O *template* "quiz.xml" ajuda a criar um quiz interativo que será apresentado durante a apresentação de um vídeo. Em conjunto com o *template* "quiz.xml", é usado o *template* "screen.xml" para apresentar as questões e as possíveis respostas em uma tela. Entre um componente *screen* e seu sucessor, no *template* quiz, existem elos "change_screen" que são responsáveis por verificar se uma das teclas coloridas do controle remoto foi pressionada (vermelha, verde, amarela e azul). Quando uma das teclas coloridas for pressionada, um elo interrompe a apresentação do componente *screen* atual, inicia seu sucessor e repassa o valor que representa a tecla pressionada para um programa em Lua (nó counter). Esse nó é responsável por testar se uma dada resposta é correta e contabilizar os pontos.

Usando XTemplate 3.0, onde componentes de leiaute não estão disponíveis, é necessário declarar no *template* "screen.xml" quatro tipos de componentes, um para cada resposta. Cada componente é associado a um par *region-descriptor*, representando a forma como serão apresentadas (uma embaixo da outra). Com XTemplate 4.0, somente um componente do tipo *answer* precisa ser definido. O componente *answer* é associado a um componente de leiaute do tipo *flowLayout* especificando que as respostas serão apresentadas uma embaixo da outra, independentemente da quantidade de respostas para cada questão. Desta forma, simplifica-se a definição do *template* "screen.xml", que agora fica independente da quantidade de respostas disponíveis para uma dada pergunta.

O *template* "quiz.xml" foi modificado da seguinte forma: (i) cada componente *screen* agora representa o *template* aninhado "screen.xml"; (ii) os elos "change_screen" foram atualizados para verificar se uma dada resposta foi selecionada e passa sua posição para o nó *counter*. A nova versão do "quiz.xml"e do "screen.xml" é mostrada na Figura 5.6.

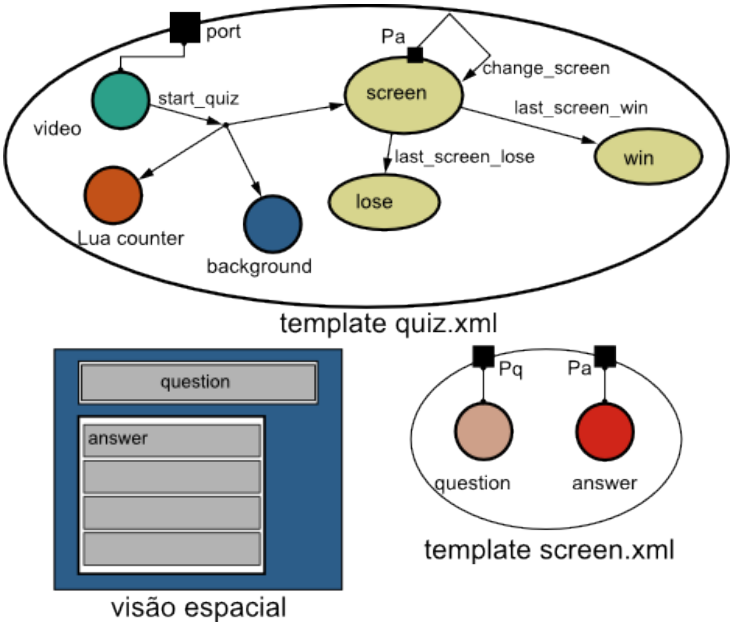

Figura 5.6: Nova versão dos templates quiz.xml e screen.xml

O *template* "layoutQuiz.xml" define dois componentes de leiaute que serão usados para a pergunta e as respostas relacionadas a essa pergunta. A pergunta é apresentada no topo da tela (centralizada) e as respostas serão apresentadas uma embaixo da outra (centralizadas). A Listagem 5.3 apresenta a definição dos componentes de leiaute (*container*).

```
1 <container id="qstF1" type="flowLayout">
2     <format align="center" height="240" hspace="0" left="10" top="20"
3         vspace="0" width="780" zIndex="3"/>
4     <item id="c1" height="100" width="780"/>
5 </container>
```

```
6
7  <container id="awsF1" type="flowLayout">
8      <format align="center" height="640" hspace="10" left="10" top="300"
9          vspace="0" width="200" zIndex="3"/>
10     <item id="c1" height="150" width="200"/>
11     <focus focusIndex="1"/>
12 </container>
13 ...
```

Listing 5.3: Componentes de leiaute de layoutQuiz.xml

O *template* "screen.xml" define como as questões e respostas serão apresentadas. Esse *template* é aninhado dentro do *template* "quiz.xml", assim ele define um elemento do tipo *port* (portAnswer) para funcionar como interface de comunicação com o *template* "quiz.xml". Um fragmento do vocabulário desse *template* é apresentado na Listagem 5.4. Como mencionado anteriormente, o *template* será processado de forma recursiva, do mais interno para o mais externo. Então, quando o "screen.xml" é processado, cada elemento *media* com o rótulo *answer* receberá um elemento *port*. Este elemento será associado com o rótulo *portAnswer* para identificar a interface de comunicação.

```
1  <vocabulary>
2      <port xlabel="portAnswer"/>
3      <component xlabel="question" layout="lay#qstF1"/>
4      <component xlabel="answer" layout="lay#ansF1"/>
5  <vocabulary>
6
7  <body>
8      ...
9      <variable name="i" select="1"/>
10     <for-each select="child::media[@xlabel='answeer']">
11         <port id="port" select="current()" xlabel="portMenu"/>
12         <variable name="i" select="$i+1"/>
13     </for-each>
14     ...
15 <body>
```

Listing 5.4: Fragmento do template screen.xml

O *template* "quiz.xml", por sua vez define o componente *screen* para representar o *template* interno. Da mesma forma como feito no *template* "screen.xml", o componente *screen* declara um elemento filho *port*, denominado *portAnswer*. Fragmentos do "quiz.xml" são apresentados na Listagem 5.5. Os códigos completos dos *templates*: quiz.xml, screen.xml e layoutQuiz.xml, podem ser encontrados no Apêndice A.

```
1  <templateBase>
2      <importBase alias="scn" documentURI="screen.xml"/>
3  </templateBase>
4  ...
5  <component xlabel="screen" xtype="xtemplate/scn"/>
6      <port xlabel="portAnswer"/>
7  </component>
8  ...
9  <link xtype="change_screen">
10     <bind role="onSelection"
11         select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][position()-1]"/>
12     <bind role="isPresenting" select="current()"/>
13     <bind role="stop" select="current()"/>
```

```
14      <bind role="set"
15          select="child::media[@xlabel='counter']/child::property[@xlabel='value_answer']">
16          <bindParam name="var" value="A"/>
17      </bind>
18      <bind role="start" select="child::*[@xlabel='screen'][position()=$i+1]"/>
19  </link>
20  ...
```

Listing 5.5: Fragmentos do template quiz.xml

Um exemplo de documento NCL usando o *template* "quiz.xml" é mostrado na Listagem 5.6. Note que, diferentemente do exemplo em [4], o autor NCL precisa usar somente o *template* "quiz.xml". O elemento *context*, que utiliza o *template* aninhado, o faz implicitamente através da referência ao *xlabel screen*.

```
1  <templateBase>
2      <importBase alias="quiz" documentURI="quiz.xml"/>
3  </templateBase>
4  <body xtemplate="quiz">
5      <context id="screen_01" xlabel="screen">
6          <media id="question_01" xlabel="question"/>
7          <media id="answer_01_A" xlabel="answer"/>
8          <media id="answer_01_B" xlabel="answer"/>
9          <media id="answer_01_C" xlabel="answer"/>
10         <media id="answer_01_D" xlabel="answer"/>
11     </context>
12     ...
13 </body>
```

Listing 5.6: Fragmento do documento NCL usando o template quiz.xml

5.4 Processamento de XTemplate 4.0 com Uso de StyLe

Quando um documento NCL utiliza um *template* de composição que usa um *template* StyLe, um objeto de mídia NCL deve referenciar o componente genérico XTemplate usando seu atributo *xlabel* e o componente genérico XTemplate deve referenciar um *item* ou container no *template* StyLe através do atributo *layout*, como mostrados na linha 6 da Listagem 5.6 e linha 3 da Listagem 5.4, respectivamente.

Documentos NCL que usam XTemplate com StyLe devem ser processados para que se transforme em um documento NCL padrão. Neste caso, o processamento do *template* é feito em dois passos. No primeiro passo, denominado **Processamento de Template**, o *template* é processado como um *template* de composição padrão, criando um documento NCL intermediário com o conteúdo do documento original e as relações de sincronização definidas no *template*. Este documento NCL intermediário segue a extensão de NCL para uso de StyLe, apresentada no capítulo anterior, onde cada objeto de mídia no documento intermediário é rotulado com a identificação do componente de leiaute ao qual está relacionado.

No segundo passo, denominado **Processamento do Leiaute**, as características de apresentação são criadas de acordo com o número dos objetos de mídia associados a cada elemento do leiaute. Depois do segundo passo, o documento final NCL tem o conteúdo do documento original, as sincronizações definidas no *template* de composição, e o posicionamento, tamanho e especificações de navegação definidas no *template* espacial StyLe. A Figura 5.7 mostra o processamento de um documento NCL em dois passos, **Processamento de Template** e **Processamento do Leiaute**.

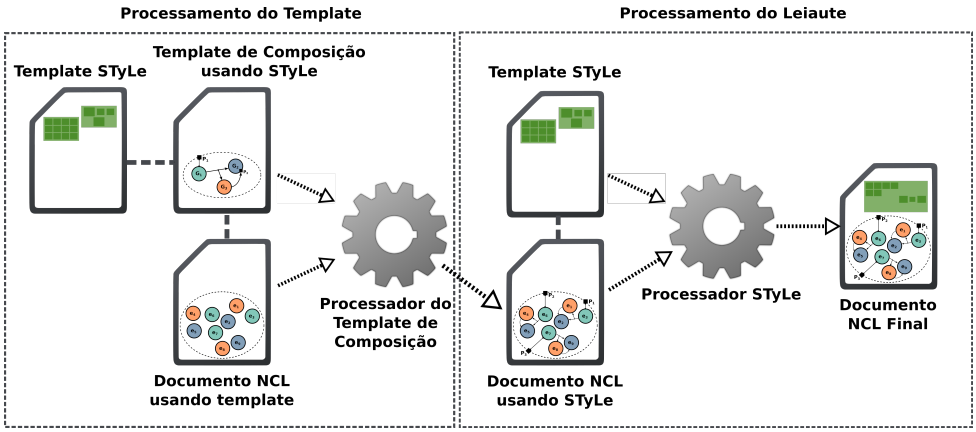

Figura 5.7: Processamento de Documentos NCL com *templates* de composição e leiautes StyLe

O XML Schema utilizado para definir os elementos da linguagem XTemplate 4.0, assim como a extensão realizada para que XTemplate 4.0 pudesse usar a linguagem StyLe, pode ser encontrado no Apêndice D.

Capítulo 6

Arquitetura para Uso de Leiautes Dinâmicos StyLe com NCL

A extensão de NCL para uso de leiautes adaptativos StyLe foi apresentada no Capítulo 4 e é oferecida através do processamento do leiaute StyLe, que gera um documento NCL 3.0 padrão. Entretanto, quando é especificado um leiaute StyLe dinâmico, com o uso de relações de restrição, não é possível gerar um documento NCL 3.0 com essas funcionalidades, já que a versão 3.0 de NCL não contempla conectores de restrição, como ocorria na versão 2.0 de NCL [12]. Por este motivo, esta tese propõe uma arquitetura e apresenta uma implementação que permitem o uso prático de leiautes dinâmicos StyLe com NCL 3.0, realizando a adaptação do leiaute do documento NCL em tempo de execução.

De acordo com os autores em [35, 36], as adaptações de conteúdo que podem causar alterações dinâmicas na aplicação podem se dar tanto no lado cliente quanto no lado servidor.

Quando o conteúdo é adaptado no lado servidor, a aplicação é alterada antes de ser enviada para o agente usuário, que recebe uma aplicação pronta para ser apresentada. Esta abordagem pode apresentar um problema de escalabilidade, pois, em cenários onde a aplicação é enviada para um conjunto de usuários e não para usuários individuais, o servidor precisa criar uma cópia do documento para cada usuário o que poderia provocar um desempenho insatisfatório.

Quando o conteúdo é adaptado no lado cliente, a escalabilidade não é mais um problema. Entretanto, esta abordagem requer do dispositivo cliente mais poder de processamento para executar o mecanismo de adaptação. A implementação de leiautes espaciais dinâmicos para documentos NCL que utilizam StyLe segue a segunda abordagem. Isso é realizado através da arquitetura apresentada na Figura 6.1, que se baseia na arquitetura

descrita em [36]. Sendo assim, o processador StyLe transforma um documento NCL que usa StyLe, chamado de aplicação base, em uma nova aplicação NCL enriquecida com um componente *controller*, como mostrado na Figura 6.1.

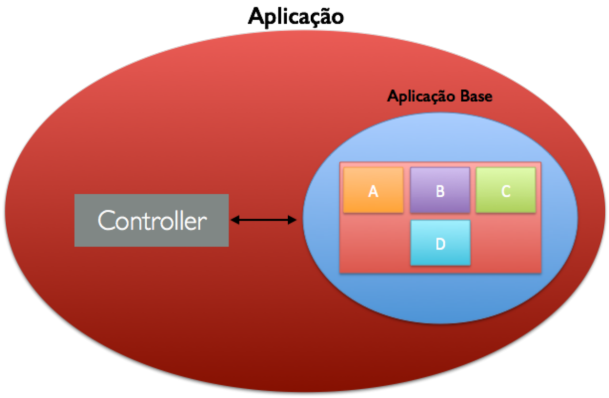

Figura 6.1: Arquitetura da Aplicação NCL para executar StyLe

A arquitetura proposta compreende um conjunto de serviços que tem como objetivo suportar a geração automática e dinâmica de leiautes para a aplicação. Esses serviços são implementados por um componente chamado *controller* e outros objetos com código imperativo, que farão parte da aplicação declarativa, como explicado nos próximos parágrafos.

Como visto na Figura 6.1, o documento NCL que terá seu leiaute espacial alterado dinamicamente é denominado *Aplicação Base*. Ele é enriquecido com um componente *controller*, responsável por capturar ocorrências de eventos e realizar as mudanças requeridas no leiaute da *Aplicação Base* em tempo de execução.

Eventos capturados pelo componente *controller* podem ser relacionados ao início ou término da apresentação de objetos de mídia, interações do usuário ou pela recepção de comandos de edição ao vivo enviados pelas transmissoras de TV. O componente *controller* é implementado em Lua [47], linguagem de *script* utilizada como auxiliar para aplicações NCL.

A sincronização espacial entre a *Aplicação Base* e o código de leiaute é construido a partir do *template* StyLe, considerando os objetos de mídia declarados pelo autor na *Aplicação Base*. Por exemplo, na Figura 6.1, a *Aplicação Base* definida pelo autor deveria especificar que os quatro objetos de mídia A, B, C e D deveriam ser apresentados seguindo o modelo *flowLayout*.

Baseado na aplicação original NCL e no *template* StyLe utilizado, o componente *controller* constrói um mapa de apresentação inicial para os objetos de mídia da *Aplicação*

Base. Sempre que o *controller* captura um evento, ele verifica se tal evento requer uma atualização do mapa de apresentação, uma mudança na estrutura do documento ou ambas. O início ou término da apresentação de um objeto de mídia é um caso típico de atualização do mapa de apresentação. A recepção de um comando de edição ao vivo é um caso típico de alteração da estrutura do documento, o que acarretará em uma modificação do mapa de apresentação.

Considere o caso onde o objeto de mídia B foi parado por intervenção do usuário. Esta interação irá disparar um evento (*Stop* B) que será capturado pelo *controller*, como visto na Figura 6.2.

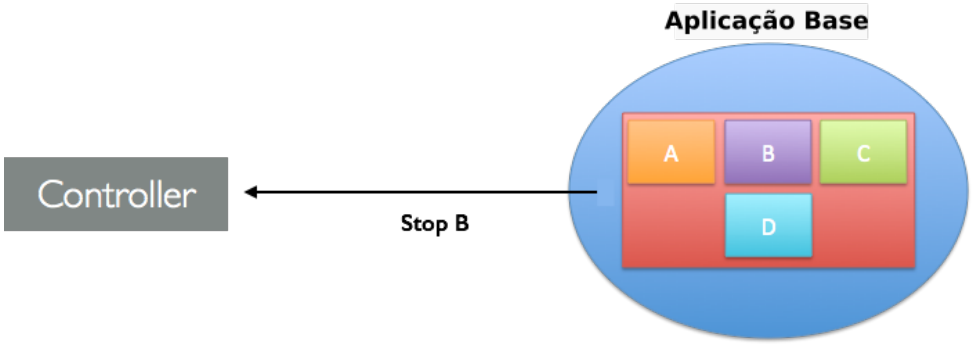

Figura 6.2: Exemplo de interação do usuário

Depois da recepção deste evento, o *controller* processará esta modificação no mapa de apresentação da *Aplicação Base*. Então, o *controller* enviará comandos de edição para alterar o leiaute espacial da apresentação de acordo com esse novo mapa.

Como o objeto de mídia B pertencia a um *container* do tipo *flowLayout*, as mudanças serão realizadas para reorganizar o restante dos objetos de mídia de tal modo que continuem sendo apresentados seguindo as regras de um modelo do tipo *flow*. Consequentemente, o objeto de mídia C será movido para a esquerda e o objeto de mídia D será movido para a primeira linha, ao lado do elemento C, como mostrado na Figura 6.3.

Figura 6.3: Mudança no leiaute de apresentação

O componente *controller* é composto por três elementos, que são: o processador STyLe, o componente *diff* e o componente *solver*. A arquitetura do *controller* é apresentada na Figura 6.4.

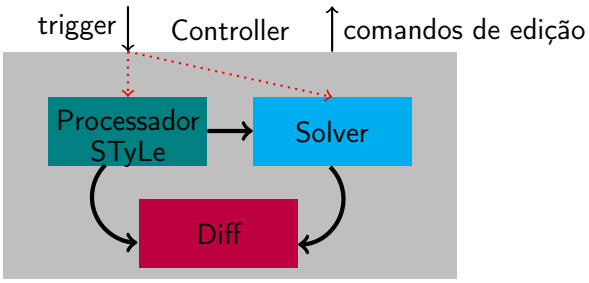

Figura 6.4: Arquitetura do *Controller*

Depois de receber uma ocorrência de um evento (*stop B*, na Figura 6.2), o *controller* decide se tal ocorrência de evento deve ser enviada para o processador STyLe ou para o componente *solver*. O processador STyLe é escolhido sempre que uma mudança na estrutura do documento é necessária e o componente *solver* é escolhido sempre que uma atualização no mapa de apresentação é requerida. Quando a estrutura do documento é alterada, o mapa de apresentação também é alterado. Portanto, o componente *solver* também é utilizado depois que o processador STyLe termina de reprocessar o *template*.

O componente *diff* calcula as alterações necessárias na estrutura do documento e/ou características de apresentação para que as modificações possam ser realizadas de forma incremental com o objetivo de se alterar suavemente o documento em execução, melhorando a QoE (*Quality of Experience*) do usuário.

O processador STyLe é usado para construir novas relações espaciais (elos NCL) para os novos objetos de mídia incluídos na *Aplicação Base* (pelos comandos de edição ao vivo) ou atualizar o conjunto de relações espaciais existentes quando objetos de mídia são removidos da aplicação. Isto é feito reprocessando o documento original, considerando as edições, com o *template* STyLe usado no documento original. Ambos, documento original e o novo documento processado, são utilizados como entrada para o componente *diff* que calcula as modificações necessárias.

O componente *solver* é usado para solucionar o conjunto de restrições de um *template* STyLe. Restrições espaciais são representadas e resolvidas usando um *solver SMT (Satisfiability Modulo Theories)* [44]. A redução da linguagem STyLe para um problema SMT foi discutida na Seção 3.4.

De acordo com [44], um problema SMT é um problema de decisão para fórmulas lógicas baseada na combinação de teorias expressas em lógica de primeira ordem com igualdades. SMT pode ser pensado como uma forma de expressar problemas de satisfabilidade de restrições e, portanto, ser definida como uma formalização de programas de restrição.

Uma instância SMT é uma fórmula escrita em lógica de primeira ordem, que engloba funções e símbolos de predicados. SMT é o problema de determinar se essa fórmula é satisfatória. Em outras palavras, imagine uma instância do problema de satisfatibilidade booleana (SAT) em que algumas variáveis binárias são substituídas por predicados obtidos sobre um conjunto adequado de variáveis não-binárias. Um predicado é basicamente uma função binária com valor de variáveis não-binárias. Exemplos de predicados incluem desigualdades lineares (por exemplo, $3x + 2y - z \geq 4$) ou igualdades, envolvendo termos não interpretados e símbolos de função (por exemplo, $f(f(u, v), v) = f(u, v)$ onde f é alguma função não especificada de dois argumentos não especificados.) Esses predicados são classificados de acordo com a respectiva teoria. Por exemplo, as desigualdades lineares sobre variáveis reais são avaliadas utilizando as regras da teoria da aritmética linear real, enquanto predicados que envolvem termos não interpretados e símbolos de função são avaliados utilizando as regras da teoria das funções não interpretadas com igualdade.

Considerando o exemplo NCL mostrado no Capítulo 4, foi simulado um evento de *stop* em um objeto de mídia que pertence a um *container grid* (no centro) (veja Figura 4.3). O objeto de mídia parado foi o que apresentava a figura *Viking*, Neste caso, o *controller* interceptou o evento. Ao processá-lo, verificou que o evento causou uma mudança no mapa de apresentação da aplicação. Deste modo, o *controller* enviou uma mensagem para o *solver*, que criou o novo mapa de apresentação. Comandos de edição foram enviados para alterar o leiaute da apresentação e o resultado pode ser visualizado na Figura 6.5.

Em um segundo experimento, foram usados comandos de edição ao vivo NCL para adicionar um novo objeto de mídia (botão *Options*) no *container menu*. Novamente, o evento é interceptado e processado pelo componente *controller*. Como o evento modifica a estrutura do documento original, é necessário processar o documento alterado com o *template* StyLe. Após identificar as diferenças entre o documento original e o novo documento através do componente *diff*, comandos de edição mudarão a estrutura do documento e seu leiaute como apresentado na Figura 6.5.

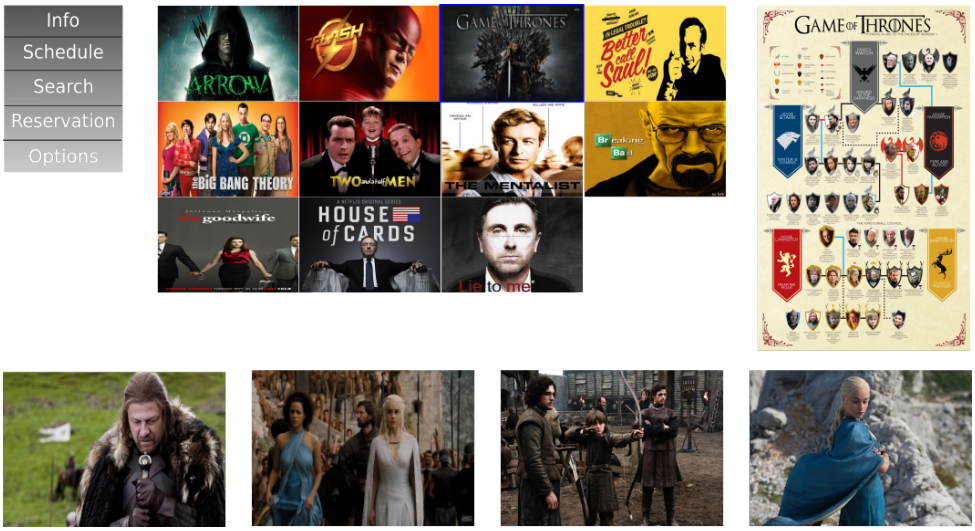

Figura 6.5: Parando e adicionando um objeto de mídia

6.1 Implementação da Arquitetura

Todos os componentes da arquitetura StyLe foram implementados na linguagem Lua [47], uma linguagem de programação funcional, leve e poderosa. Além de se levar em conta essas características para a escolha dessa linguagem, outro fator primordial foi a fácil comunicação existente entre NCL e Lua.

Como a abordagem escolhida para a arquitetura proposta foi a abordagem baseada no cliente, todos os componentes devem ser executados no dispositivo receptor do indivíduo. Como a integração de *scripts* Lua pode ser feita de forma simples em NCL, a solução encontrada foi encapsular toda a arquitetura StyLe em um nó de mídia Lua na aplicação NCL.

A ideia central na construção da implementação é estabelecer todas as situações necessárias para rearranjar os elementos do leiaute e inseri-las no código NCL que será executado. Como mencionado anteriormente, isso pode ser feito em tempo de autoria através da declaração de elos que estabeleçam a relação espacial entre os objetos. Entretanto, todas as alterações devem ser previstas, o que é extremamente complexo.

Nossa concepção usou a definição de elos, mas os momentos nos quais os elos são ativados dependem dos eventos que são disparados, ou por intermédio da interação do usuário, ou por eventos de edição ao vivo ou, até mesmo, por início ou término natural da apresentação da mídia.

Os dois principais arquivos necessários para a execução do leiaute dinâmico são: o documento NCL que utiliza o *template* StyLe e o documento que descreve esse *template*

espacial. Esses documentos servem de entrada para que o processador possa ter em memória as informações pertinentes de localização dos objetos de mídia e dos componentes StyLe que serão utilizados. Inicialmente, o documento NCL que usa StyLe precisa ser editado para que as informações de posicionamento possam ser acessadas e modificadas. Este passo é realizado antes de criarmos o componente *controller*.

Os dois documentos, ambos baseados em XML, são traduzidos em tabelas Lua através de uma biblioteca construída para manipular documentos XML. Existem várias bibliotecas para manipular documentos XML em Lua, mas, na revisão realizada, nenhuma utilizava somente códigos Lua. A maioria encapsulava códigos da linguagem C dentro dos métodos em Lua, o que é perfeitamente possível, já que Lua tem como base a linguagem C. O problema nessa abordagem é que seria necessário compilar esse código no receptor do cliente o que não é desejável.

Tabelas em Lua são *arrays* associativos [47]. Um *array* associativo é um *array* que pode ser indexado não apenas por números, mas também por cadeias ou qualquer outro valor da linguagem, exceto o valor nulo (*nil*). É a única estrutura de dados da linguagem e pode ser utilizada para representar conjuntos, registros, pacotes e até objetos. A Figura 6.6 mostra um exemplo simples de uma tabela em Lua.

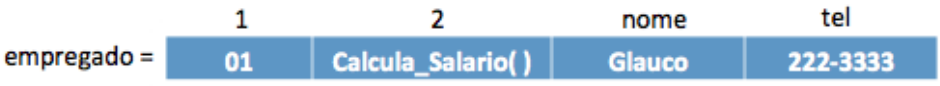

Figura 6.6: Exemplo simples de uma tabela em Lua

No documento NCL que utiliza StyLe, todo objeto de mídia que fizer referência ao leiaute espacial receberá uma propriedade que será utilizada para manipular as características de posicionamento, como indicado na Listagem 6.1.

```
<media id='imgRJ' src='videoRJ.mp4' descriptor='drj'>
    <property name='location'/>
</media>
```

Listing 6.1: Fragmento de código NCL mostrando definição das propriedades

Além disso, serão adicionados elos do tipo *get* e *set* para manipular as propriedades de cada objeto de mídia, como mostrado na Listagem 6.2.

```
...
<link xconnector='onEndAttributionGetSet'>
    <bind role='onEndAttribution' component='mlua' interface='locrj'/>
    <bind role='get' component='mlua' interface='locrj'/>
    <bind role='set' component='imgRJ' interface='location'>
        <bindParam name='var' value='$get'/>
    </bind>
</link>
```

```
9 ...
```

Listing 6.2: Fragmento de código NCL que define os elos

Essas alterações serão realizadas na tabela Lua que manipula o documento NCL através dos métodos denominados, respectivamente: *createProperites* e *createLinks*.

Outra alteração realizada no documento NCL que usa o *template* espacial é a inserção de um objeto de mídia do tipo Lua, que representará o componente *controller* na aplicação NCL. Este objeto de mídia, criado automaticamente, terá diversas âncoras de conteúdo (*area*) e de propriedade (*property*), uma para cada objeto de mídia que referenciar um leiaute espacial. As âncoras servirão como interface de comunicação entre o documento NCL e o *script* Lua que tratará todos os eventos gerados. Um exemplo desse objeto de mídia é mostrado na Listagem 6.3. Essa alteração é realizada pelo método *createMediaLua*.

```
1  ...
2  <media id='controller' src='controller.lua'>
3      <area id='arj'/>
4      <area id='apr'/>
5      <area id='atr'/>
6      <property name='locrj'/>
7      <property name='locpr'/>
8      <property name='loctr'/>
9  </media>
10 ...
```

Listing 6.3: Fragmento de código NCL que declara a mídia Lua

O próximo passo é criar o *script* Lua que controlará a recepção dos eventos e determinará qual método deverá ser chamado: o processador de *template*, o *solver* ou ambos. Esse *script* deverá representar todos os componentes declarados no *template* StyLe utilizado. Por isso, é gerado automaticamente utilizando como base o processamento do *template* em conjunto com o documento NCL que o referencia. Este passo é criado pelo arquivo *createController.lua*. Ele é a representação do componente *controller* do lado da linguagem Lua.

Neste momento, os objetos que representam os *itens* e *containers*, sejam eles tipados ou não, são instanciados e carregados na memória na forma de tabelas Lua. A Figura 6.7 mostra um diagrama de classes dos elementos espaciais.

A Listagem 6.4 apresenta um trecho de código utilizado para criar um objeto que representa o modelo *flowLayout*. Além do construtor, linhas 16 - 21, todos os métodos para obter e alterar as informações dos atributos foram implementados.

```
1  Flow = {  id = "",
2            focusIndex = 0,
```

```
3           top = 0,
4           left = 0,
5           bottom = 0,
6           right = 0,
7           width = 0,
8           height = 0,
9           hspace = 0,
10          vspace = 0,
11          align = "center",
12          itens = {},
13          medias = {}
14 }
15
16 function Flow:new(o)
17   o = o or {}
18   setmetatable(o, self)
19   self.__index = self
20   return o
21 end
22 ...
```

Listing 6.4: Fragmento de código de criação do objeto Flow

Seguindo o mesmo raciocínio, as restrições espaciais também são mapeadas em objetos (tabelas Lua) para que as informações pertinentes sejam manipuladas. A Figura 6.8 mostra o diagrama de classes utilizado para as restrições espaciais.

Esses objetos ficam carregados em memória e guardam as informações necessárias para que o componente *solver* possa calcular os posicionamentos dos objetos levando em consideração as restrições declaradas.

A base da arquitetura é o componente *solver*, pois é o componente que processa as restrições espaciais impostas sobre os elementos e retorna os valores de posicionamentos que satisfazem essas restrições.

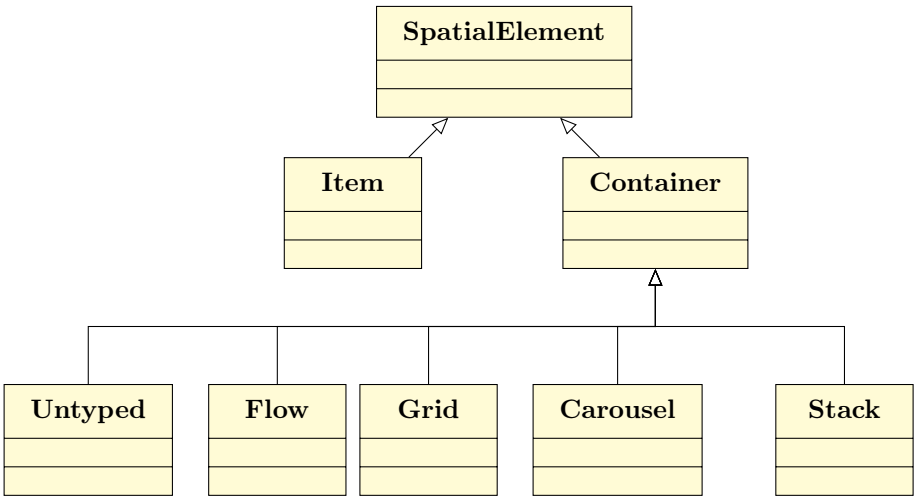

Figura 6.7: Diagrama de Classes dos Elementos Espaciais

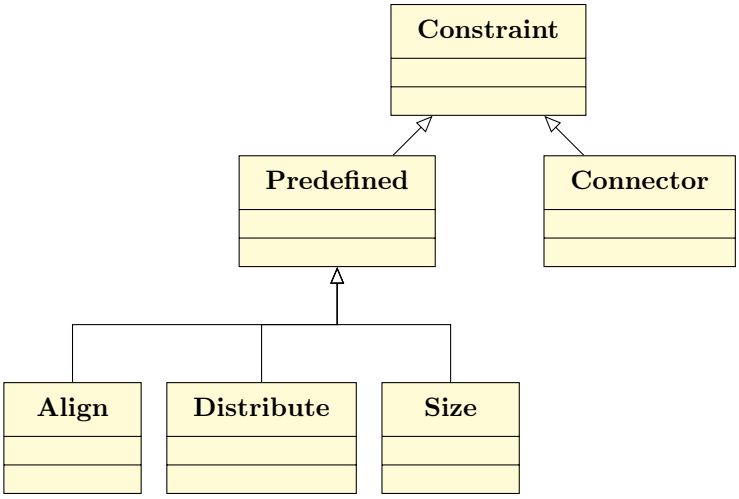

Figura 6.8: Diagrama de Classes das Restrições Espaciais

Portanto, todas as relações espaciais declaradas no documento de *template* são mapeadas em métodos no componente *solver*. Sempre que algum evento for detectado, o método correspondente à relação espacial é invocado para calcular e fornecer os novos valores de posicionamento ou tamanho.

Os modelos espaciais *flowLayout*, *gridLayout*, *carouselLayout* e *stackLayout* são representados pelo elemento *container* e trazem restrições embutidas em suas relações. Portanto, os *containers* também serão mapeados por métodos no código.

O *solver* SMT escolhido foi o Yices2 [48] por possuir características adequadas para esse ambiente e por se tratar de um poderoso solver, como indicam os resultados obtidos em [49]. A lógica do Yices suporta os tipos primitivos *int* e *real* para os tipos aritméticos, *bool* para o tipo booleano entre outros. Termos em Yices podem representar constantes ou fórmulas. Os termos primitivos suportados são os aritméticos e os booleanos.

O Yices mantém uma base global de tipos e termos. Sua API fornece funções para a construção de tipos, termos, termos de impressão e etc. A estrutura central do Yices é o contexto. Um contexto armazena afirmações que serão verificadas para saber se são satisfeitas ou não. Da mesma forma que para tipos e termos, sua API fornece funções para criar, destruir e verificar contextos, além de adicionar e remover afirmações em contextos.

Se as afirmações em um contexto são satisfatórias, o *solver* pode construir um modelo do contexto. Uma vez construído, um modelo pode ser consultado e examinado independentemente do contexto. Além disso, mudanças adicionais no contexto não afetam um modelo já criado.

É possível ainda, criar um ponto de retrocesso e voltar para um ponto de *backtracking* criado anteriormente. Esse recurso é interessante quando algumas fórmulas no contexto podem ser removidas, se não forem satisfatórias.

A declaração de tipos e termos em Yices2 pode ser realizada usando a sintaxe SMT-LIB 2.0. Mais informações sobre a ferramenta estão disponíveis no manual técnico [48].

O Yices2 possui uma biblioteca escrita na linguagem C. Essa biblioteca foi compilada e utilizado dentro do código Lua do *solver*. Um trecho de código da função que manipula informações referentes ao *container gridLayout* pode ser visualizado na Listagem 6.5.

```
function model:grid_dyn(p, items, nrow, ncol, hspace, vspace)
...
for i = 2, #items do
    local item_a = items[i-1]
    local item_b = items[i]
    local pos_a = smt.int(i-1)
    local pos_b = smt.int(i)

    smt.assert(self,
        smt.ite(item_a._oc,
            smt.ite(smt.lt(smt.apply(pos, {nf, pos_a}), nc),
                smt.land{
                    smt.eq(smt.apply(pos, {nf, pos_b}), smt.sum{smt.apply(pos, {nf, pos_a}), smt.int(1)
    }),
                    smt.eq(item_b._xi, smt.sum{item_a._xe, hs}),
                    smt.eq(item_b._yi, item_a._yi)
                },
                smt.land{
                    smt.eq(smt.apply(pos, {nf, pos_b}), smt.int(1)),
                    smt.eq(item_b._xi, canvas._xi),
                    smt.eq(item_b._yi, smt.sum{item_a._ye, vs})
                }
            ),
            smt.land{
                smt.eq(smt.apply(pos, {nf, pos_b}), smt.apply(pos, {nf, pos_a})),
                smt.eq(item_b._xi, item_a._xi),
                smt.eq(item_b._yi, item_a._yi)
            }))

    smt.assert(self,
        smt.land{
            smt.eq(item_b._xs, cell_xs),
            smt.eq(item_b._ys, cell_ys)
        })
end
...
```

Listing 6.5: Fragmento de código da função que manipula gridLayout

Esse código ilustra a implementação em Lua das Fórmulas SMT 3.29, 3.30, 3.31 e 3.32 apresentadas na Seção 3.4 do Capítulo 3.

De acordo com a hierarquia definida para as restrições, inicialmente o sistema deve tentar resolver o conjunto de restrições por completo. Se isso não é possível, uma nova tentativa é feita, removendo-se as restrições com prioridade *low* do conjunto de restrições. Esse processo continua até que o conjunto de restrições tenha somente restrições

com prioridade *core*. Para definir a hierarquia das restrições dentro do mesmo nível de prioridade, a ordem de declaração dentro do *template* é usada.

Para implementar esse modelo de hierarquia, varre-se o conjunto de maior prioridade e, levando em conta a ordem de declaração do documento, coloca-se um ponto de retrocesso depois de cada definição da restrição. Isso é feito para os outros níveis de prioridade até que todos os conjuntos tenham sido varridos. Quando uma restrição não é satisfeita, o sistema volta ao ponto de retrocesso imediatamente anterior, retira-se a restrição que não foi satisfeita e executa-se novamente com o conjunto de restrições restante.

Vale a pena ressaltar que, apesar do sistema de hierarquia de prioridades ser extremamente importante para aumentar o conjunto de soluções possíveis para um leiaute proposto, esta técnica pode influenciar diretamente na relação entre leiaute esperado e leiaute obtido. Se as restrições que pertencem a um certo nível de prioridade não forem utilizadas na resolução do leiaute, o leiaute apresentado pode diferir consideravelmente do leiaute especificado. O autor deve ser alertado para esse possível imprevisto. No Capítulo 8, será apresentado uma proposta futura para tentar amenizar esse problema.

O diagrama de atividades e o diagrama de componentes podem ser vistos, respectivamente, nas Figuras 6.9 e 6.10.

O diagrama de atividades determina o fluxo de controle da implementação proposta para a arquitetura. As primeiras ações a serem tomadas são ler o documento NCL que utiliza o *template* StyLe e o próprio *template*.

De posse do documento NCL, é possível criar as propriedades que serão utilizadas para manipular as informações espaciais, o objeto de mídia Lua que servirá como intermediário para que se possa editar as propriedades espaciais no documento NCL com os valores obtidos pelo *solver* e os elos necessários para realizar essas alterações.

Através do processamento dos dois documentos (NCL e StyLe), é criado o mapa de apresentações, isto é, a tabela que contém informações sobre as propriedades espaciais utilizadas (nomes e valores) e estado de cada objeto de mídia (apresentando ou não). Com o mapa estabelecido, pode-se criar automaticamente o script Lua, denominado *Controller*, para interceptar qualquer evento que altere as informações espaciais dos objetos de mídia.

Em seguida, cria-se uma instância do *solver* que contém um objeto para cada componente de leiaute e restrição declarado no *template* StyLe e aguarda-se por um evento.

Quando o evento ocorre, o *Controller* verifica se o mesmo é um início ou pausa de um objeto de mídia. Se for, o mapa é atualizado e os eventos necessários para modificar

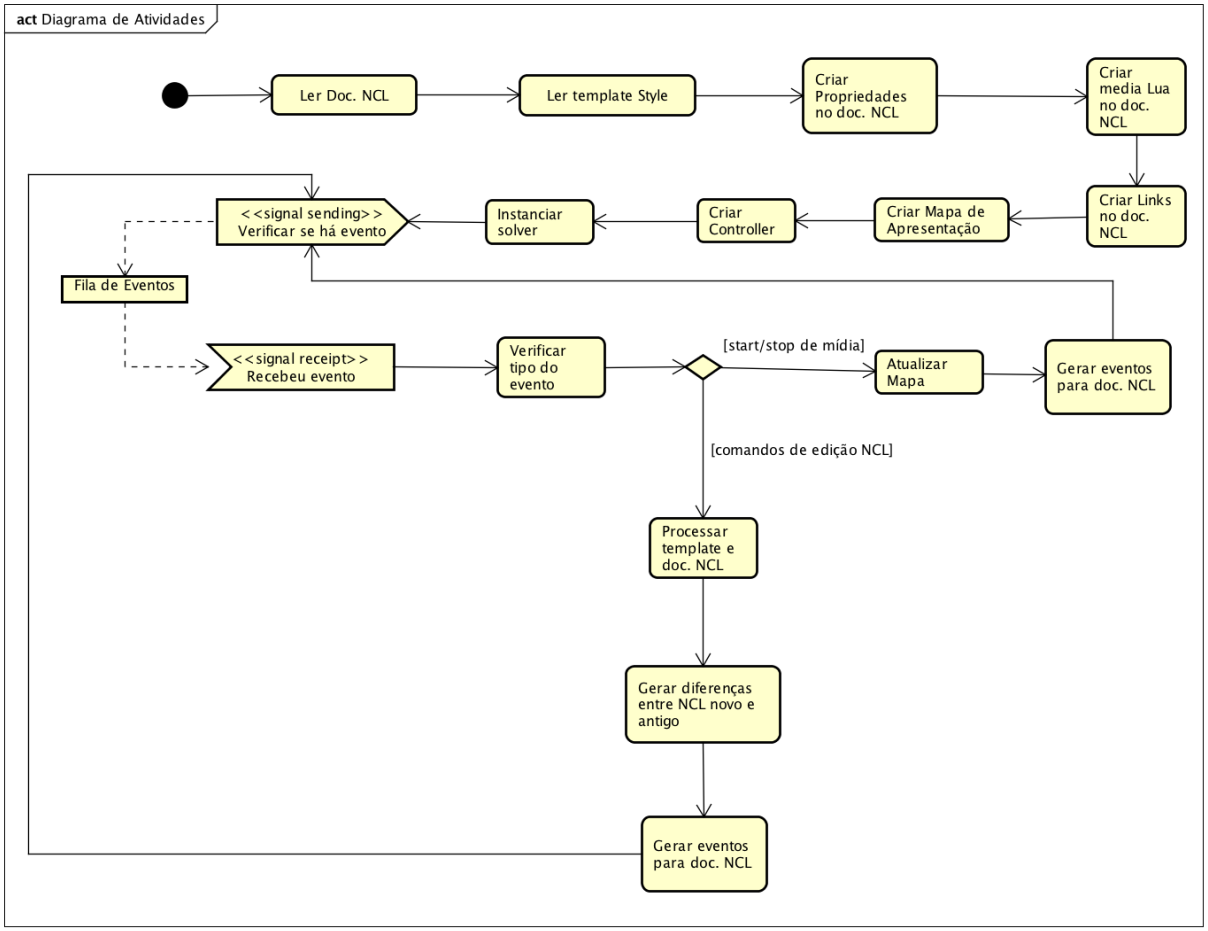

Figura 6.9: Diagrama de Atividades da Implementação

as informações espaciais dos objetos de mídia envolvidos são enviados para o documento NCL. Se o evento for ocasionado por comandos de edição NCL, é necessário processar novamente o *template* e o documento NCL, gerar as diferenças entre o documento NCL novo e o documento NCL antes de receber o evento de edição e gerar ações em eventos NCL para que essas diferenças se reflitam na aplicação.

O diagrama de componentes, ilustrado na Figura 6.10, apresenta uma visão estática de como a arquitetura será implementada, incluindo algumas bibliotecas utilizadas como auxílio para a implementação (por exemplo, tableToXML.lua).

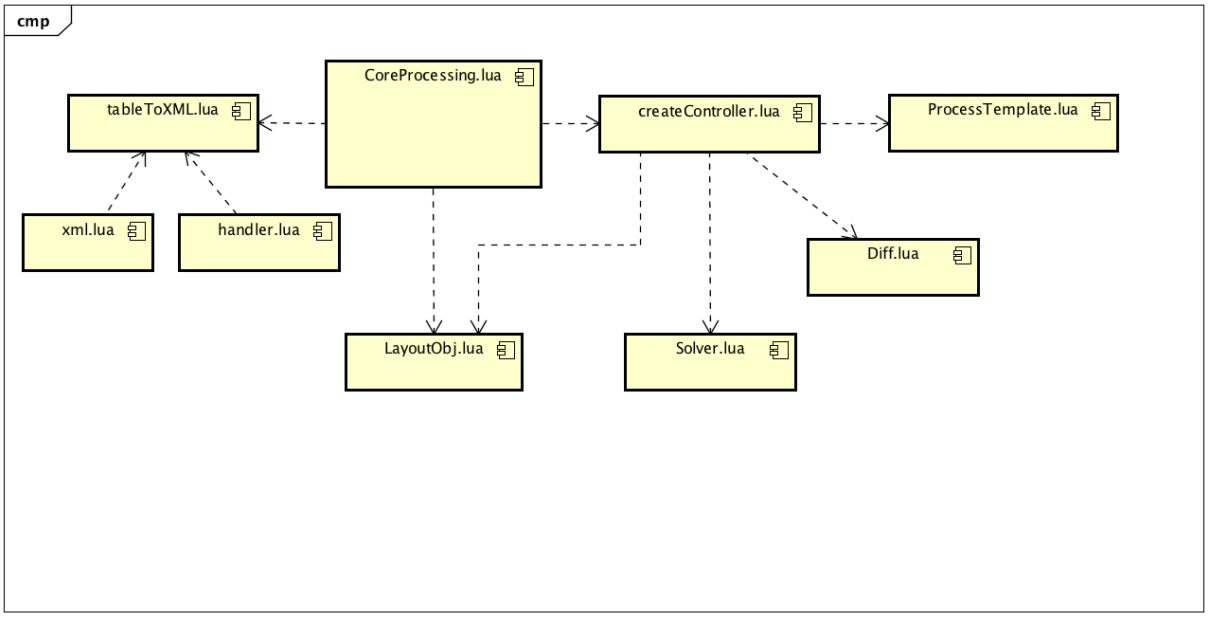

Figura 6.10: Diagrama de Componentes da Implementação

Capítulo 7

Testes e Avaliação

Este capítulo apresenta testes de usabilidade da linguagem SᵀyLᵉ para a especificação de leiautes adaptativos, testes de desempenho considerando a execução de documentos NCL com leiautes dinâmicos em SᵀyLᵉ e avaliações dos resultados obtidos em ambos os testes.

7.1 Testes dos Leiautes Adaptativos oferecidos por SᵀyLᵉ

Para avaliar a usabilidade da linguagem SᵀyLᵉ ao se criar leiautes adaptativos, foi realizado um teste com cinco atividades. Em cada atividade, os autores deveriam apenas escrever o código para um ou mais componentes de leiaute gerando o *template* de leiaute. Este *template* de leiaute SᵀyLᵉ é utilizado por um *template* de composição XTemplate, que é instanciado em um documento NCL. Os autores que realizaram o teste não tiveram contato com o *template* de composição, pois o objetivo do teste era avaliar apenas a linguagem SᵀyLᵉ para descrição de leiautes adaptativos.

Um total de 20 alunos participaram do teste: sete estudantes do curso Técnico de Telecomunicações/TVDigital (Grupo 1) e 13 estudantes do Bacharelado em Ciência da Computação (Grupo 2), ambos do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET - RJ).

Todos os estudantes do Grupo 1 estavam cursando a disciplina de aplicações para TV Digital e, portanto, tinham bom conhecimento sobre NCL. Já os estudantes do Grupo 2 não possuíam nenhum conhecimento sobre NCL, mas já tinham trabalhado com HTML, que também é baseada em XML.

Antes de começar as atividades, os autores passaram por um breve treinamento onde foram apresentados os principais elementos da linguagem STyLe (*item*, *container* e *restrições espaciais*). Após esse treinamento, os autores passaram a realizar as atividades.

Na Atividade 1, os autores deveriam colocar objetos de mídia em uma grade, com três colunas e duas linhas, no centro da tela do exibidor. Na Atividade 2, os autores deveriam colocar objetos de mídia em um *flow*, com itens do mesmo tamanho, na parte inferior da tela do exibidor. Essas duas atividades possuem complexidade baixa, pois o autor pode utilizar os modelos de leiaute definidos nos *containers* tipados para especificar diretamente o leiaute. Entretanto, o autor pode descrever o leiaute através de *itens* e restrições espaciais. Embora possível, essa solução não foi verificada em nenhuma resposta dos grupos avaliados.

Na Atividade 3, os autores deveriam colocar objetos de mídia em um *flow*, com itens de dois tamanhos diferentes, no topo da tela e uma grade, com quatro colunas e uma linha, na parte inferior da tela do exibidor. Na Atividade 4, os autores deveriam colocar os objetos de mídia em duas grades, uma do lado esquerdo e outra do lado direito da tela, ambas com quatro linhas e uma coluna e um *flow*, com itens do mesmo tamanho, na parte inferior da tela do exibidor. Nessas atividades, os leiautes propostos eram um pouco mais complexos que o anterior, visto que mesclavam dois modelos de leiautes com tamanho variado para os itens. Novamente, embora fosse possível a criação dos leiautes com a declaração de itens e restrições espaciais, os autores preferiram utilizar os *containers* tipados.

Para as quatro atividades sugeridas, foram apresentados desenhos ilustrativos dos leiautes para que não houvesse dúvida na interpretação do texto que propunha a atividade.

Finalmente, na Atividade 5, os autores deveriam criar qualquer leiaute utilizando os modelos disponíveis. Foi possível verificar que, apesar da flexibilidade proposta para a atividade, os autores ainda utilizaram os modelos de leiautes predefinidos para criarem os leiautes. A combinação dos modelos de leiautes foi a técnica mais utilizada pelos autores.

Todos os autores, de ambos os grupos, conseguiram terminar as tarefas apesar de erros cometidos no decorrer do teste. O tempo que cada autor levou para realizar cada teste não foi medido.

Depois de completarem as cinco atividades, os autores preencheram um questionário com nove questões relacionadas às seguintes dimensões cognitivas [50]:

- Visibilidade - habilidade para visualizar facilmente os componentes da linguagem;

- Expressividade dos papéis - a utilidade de qualquer entidade é facilmente percebida;
- Proximidade do mapeamento - proximidade da representação com os elementos do domínio;
- Verbosidade - algumas notações podem ser extensas;
- Comprometimento prematuro - restringe uma ordem para realizar as tarefas;
- Dependências ocultas - vínculos importantes entre as entidades não são visíveis;
- Propensão a erros - a notação induz a erros e o sistema fornece pouca proteção;
- Consistência - semânticas similares são expressas com sintaxes similares;
- Viscosidade - resistência à mudança.

O questionário pode ser encontrado no Apêndice E.

Cada questão deveria ser respondida com um valor de um a dez. O resultado de cada dimensão é uma média entre as notas dada por cada aluno. Além das nove questões, foi solicitado que o aluno desse uma nota final de um a dez para o uso de STyLe ao se desenvolver as atividades. O valor correspondente a este quesito pode ser observado no gráfico com título Final. Todos os resultados podem ser observados na Figura 7.1.

É importante ressaltar que algumas dimensões cognitivas relacionam fatores positivos da linguagem, como: visibilidade, expressividade dos papéis, proximidade do mapeamento e consistência. Já outras dimensões estão ligadas a fatores negativos da linguagem, como: verbosidade, comprometimento prematuro, dependências ocultas, propensão a erros e viscosidade. Espera-se de uma boa avaliação de usabilidade da linguagem que as dimensões cognitivas que estão ligadas a fatores positivos da linguagem sejam bem avaliados e que as dimensões cognitivas atreladas a fatores negativos da linguagem recebam valores baixos no teste.

A primeira conclusão interessante que podemos obter é que, embora os dois grupos tenham conhecimentos diferentes sobre NCL, os resultados foram muito similares em ambos os grupos. Este resultado mostra que a experiência do autor com leiautes adaptativos não é influenciada pela experiência do autor com a linguagem alvo, no nosso caso NCL.

Uma das principais preocupações ao propor a linguagem de autoria com leiautes adaptativos era manter a verbosidade baixa. Como pode ser visto nos resultados do teste, este objetivo foi alcançado. Outro importante resultado está nas dimensões expressividade

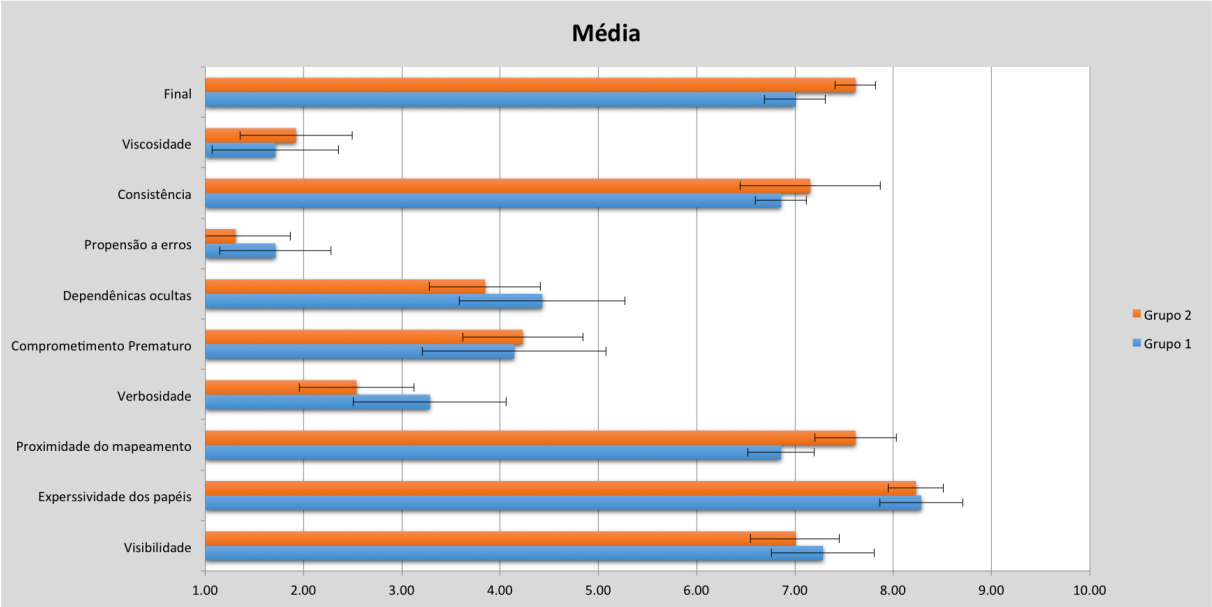

Figura 7.1: Resultado dos testes de usabilidade

dos papéis e visibilidade. O resultado indica que a linguagem é percebida como autoexplicativa e fácil de usar. Combinado com os resultados para as dimensões propensão a erros e viscosidade, pode-se concluir que a funcionalidade de leiautes adaptativos não traz dificuldade para quem quer usá-la, mesmo para autores inexperientes.

Embora a avaliação dos resultados para as dimensões dependências ocultas e comprometimento prematuro não tenham sido muito altos, não estão ruins. Dependência oculta é explicada por causa da forma como a navegação fornecida pelos componentes de leiaute é criada. Ambos os atributos *id* e *focus* do componente do leiaute são usados para definir o índice de foco no descritor NCL. O valor de ambos os atributos é fornecido pelo autor. Um erro na atribuição destes valores, por exemplo no índice de foco, pode ocasionar um erro na navegação. Esta dependência oculta pode ser resolvida se o índice de foco não tiver relação direta com estes atributos, o que é feito na atual implementação do processador de *templates*.

Já comprometimento prematuro é explicado porque os autores não sabiam que a posição relativa entre os componentes do leiaute no *template* não interfere no posicionamento final dos objetos de mídia na tela do dispositivo, o que não configura em uma característica negativa da linguagem, mas um equívoco na abordagem do teste. É importante refazer testes que evidenciam essas dimensões, como alternar as posições dos componentes de leiaute em testes consecutivos, e verificar a resposta dos usuários.

É importante ressaltar que os usuários que participaram do teste foram encorajados a deixarem sugestões e impressões sobre a linguagem. A utilização de uma ferramenta gráfica para a criação dos leiautes foi uma sugestão encontrada com frequência. A ferramenta gráfica é um trabalho em andamento.

Outra sugestão encontrada foi desenvolvimento de elementos que pudessem dar a flexibilidade ao autor de criar leiautes onde os objetos de mídia pudessem ser rotacionados ou aumentar o objeto de mídia selecionado. O desenvolvimento de modelos mais sofisticados é um trabalho futuro.

Em geral, o recurso de leiautes adaptativos foi muito bem avaliado. Entretanto, como as sugestões propuseram, alguns ajustes podem ser realizados para melhorá-lo.

Algumas impressões deixadas pelos participantes do teste foram bem positivas como: *"É uma boa ferramenta, muito prática e de fácil entendimento"* e *"O programa nos traz uma liberdade e uma facilidade para a programação"*. Embora positivas, as impressões foram encontradas somente nos participantes do Grupo 1 (estudantes com conhecimento da linguagem NCL). Possivelmente, porque esse grupo consegue compreender melhor as vantagens fornecidas pela linguagem StyLe.

7.2 Avaliação de XTemplate 4.0

Como discutido na Seção 5.1 do Capítulo 5, a facilidade de aninhamento de *templates* representa uma relação explícita entre os *templates* internos e o externo, o que é útil para editores gráficos com suporte a *templates*. Quando o aninhamento de *templates* está disponível, um editor gráfico pode tornar o uso de vários *templates* aninhados transparente para o autor do documento. Utilizando somente edição textual, o autor do documento poderá não perceber claramente a diferença entre usar *templates* separados ou *templates* aninhados. Por isso, ainda não foi aplicado um teste de usabilidade para avaliar este recurso como foi feito para a facilidade de leiautes adaptativos. Tais testes serão realizados assim que o editor gráfico com suporte a *templates* aninhados estiver disponível.

No intuito de analisar as outras funcionalidades de XTemplate 4.0, foi feita uma avaliação para verificar a diminuição do esforço do autor do *template* ao utilizar as facilidades disponibilizadas. Para tal, foram verificados quantos elementos da linguagem são necessários para criar um cenário utilizando ou não a definição genérica de leiautes e o aninhamento de *templates*. A avaliação das facilidades foi feita separadamente, para que se possa identificar o quanto cada recurso impactou a diminuição do esforço de autoria.

Para avaliar a definição genérica de leiautes, os cenários utilizados foram os exemplos descritos nas atividades 1, 2, 3 e 4 e um último exemplo com um leiaute mais complexo. O quinto exemplo simula uma biblioteca de séries de TV. No canto esquerdo da tela, existe um menu com quatro opções. No centro, aparecem os títulos das séries ordenados em uma grade 4 x 3. No lado direito, existe uma área com dados referentes ao menu da esquerda e, na parte inferior, ordenados em quatro regiões alinhadas, são mostrados os episódios da série selecionada pelo usuário. Este exemplo foi construído usando dois elementos *gridLayout*: grade central e menu à esquerda; e dois elementos *flowLayout*: vídeos na parte inferior e região com informação à direita, como pode ser visualizado na Figura 4.3 do Capítulo 4.

Foram contabilizados todos os elementos relacionados somente às características de apresentação necessárias para criar os cenários propostos. Os outros elementos da linguagem não foram contados, pois o número permanece o mesmo quando se utiliza os leiautes adaptativos e quando não se utiliza a facilidade. Por exemplo, quando STyLe não é utilizado, é necessário que o autor declare todos os elementos relacionados com a apresentação, portanto, todos os elementos *region* foram contabilizados, assim como os elementos *descriptor*. Quando STyLe é utilizado, o autor tem que instanciar os componentes de leiaute, então todos os elementos utilizados para realizar essa instanciação foram contabilizados.

O resultado da avaliação pode ser visualizado na Figura 7.2.

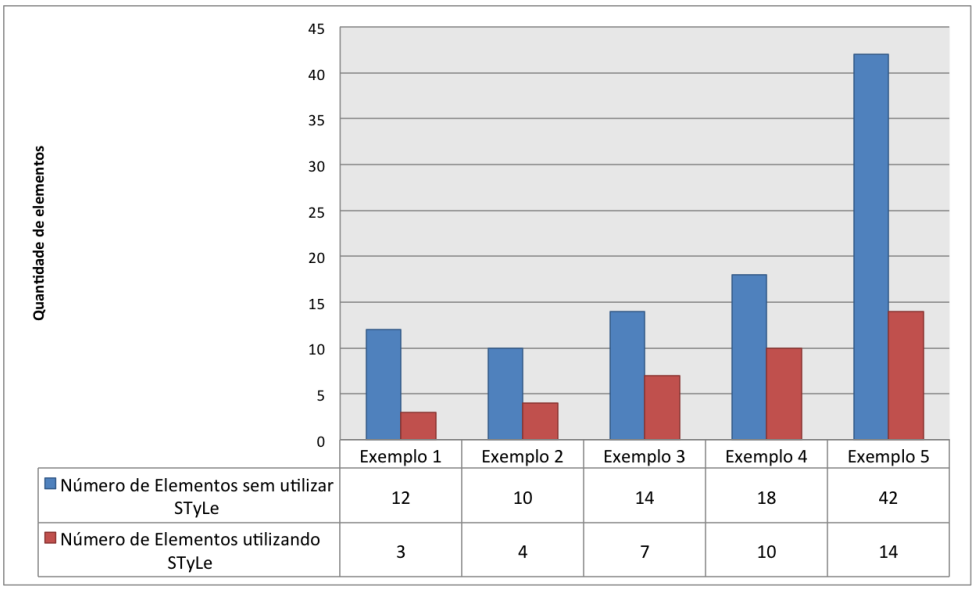

Figura 7.2: Gráfico com o número de elementos necessários em relação a utilização ou não de STyLe

Como pode ser visto, sem a utilização dos modelos de leiaute, a quantidade de elementos da linguagem cresce consideravelmente com o aumento da complexidade dos exemplos. Esse resultado é esperado, pois os exemplos mais complexos implicam a criação de mais pares *region-descriptor* em NCL. Com a utilização dos modelos, esse esforço é mitigado porque o autor precisa definir somente o modelo adequado e, para cada modelo, alguns poucos elementos filhos obrigatórios. Vale a pena ressaltar que esse esforço inicial para declarar o modelo não está relacionado à quantidade de itens de leiaute. Por exemplo, o esforço para criar uma grade com nove itens (3x3) é o mesmo para criar uma grade com 49 itens (7x7). Obviamente, isso não acontece sem a utilização do modelo de leiaute, porque para cada objeto de mídia definido é necessário criar, pelo menos, uma região (elemento *region*) e um descritor (elemento *descriptor*). Sendo assim, podemos considerar que, se o leiaute criado for apenas adaptativo, ou seja, não contiver elementos necessários para alterar as características de posicionamento dos objetos de mídia, pode-se calcular a quantidade de elementos necessários através da fórmula: $elementos = 2n$, onde n é a quantidade de objetos de mídia no documento.

Quando utiliza-se os *containers* tipados da linguagem STyLe, a quantidade de elementos é fixa e determinada pelo tipo do *container* (2 elementos para o *gridLayout* e 3 elementos para os demais). Esse valor não se altera, mesmo quando a quantidade de objetos de mídia dentro do *container* cresce. Portanto, quando o leiaute é mais complexo, a utilização da linguagem STyLe reduz consideravelmente o esforço de autoria se comparado a criação de um documento sem a utilização da linguagem STyLe.

Se for considerado o cenário dinâmico, a diferença entre utilizar a linguagem STyLe e não utilizar impacta tanto na quantidade de elementos que torna a autoria proibitiva. Para que fosse possível construir um documento NCL que reagisse às alterações feitas pela interação do usuário em tempo de execução, o autor deveria prever todas as modificações possíveis.

Suponha que se queira construir uma grade dinâmica com n elementos. Seria necessário criar um elo para testar o estado do objeto de mídia (ocorrendo ou não) e $(n - 1)$ elos, com as informações corretas de posicionamento, para alterar o posicionamento dos elementos caso o primeiro objeto de mídia não fosse apresentado. Isso deveria ser feito para todos os objetos de mídia na grade.

Com a linguagem STyLe, os elos são gerados automaticamente pelo processador da linguagem. Além disso, as restrições espaciais são solucionadas pelo solver, o que garante a consistência do leiaute.

Para analisar o aninhamento de *templates*, vários exemplos XTemplate que usam mais de um *template* foram redefinidos para utilizar o recurso de aninhamento de *templates*, incluindo o exemplo do quiz mostrado na Seção 5.3 do Capítulo 5. Pode-se perceber, através dos resultados obtidos, um aumento de elementos de linguagem ao se utilizar o aninhamento de *templates* proposto. Entretanto, esse aumento é muito pequeno e deve-se ao fato da obrigatoriedade de se definir um elemento *port*, tanto no *template* interno quanto no *template* externo, para prover a comunicação entre os mesmos. Considerando as pequenas modificações necessárias para utilizar o recurso de aninhamento de *templates* em todos os exemplos analisados, pode-se afirmar que não houve aumento de complexidade na autoria dos exemplos.

Ao combinar os resultados obtidos sobre as duas facilidades, é possível notar uma diminuição relevante no número de elementos de linguagem, que é um dos fatores que impacta diretamente o esforço de autoria do *template*.

7.3 Avaliação de Desempenho da Adaptação Dinâmica de Documentos NCL com Uso de StyLe

Como mencionado anteriormente, a implementação da arquitetura, proposta no Capítulo 6 para suportar leiautes dinâmicos em documentos NCL, utiliza Yices como componente para o *solver*. Desta forma, testes foram realizados para observar o tempo de resposta do *solver*, considerando diferentes tipos de alterações no leiaute da aplicação NCL durante sua execução.

Já que um dos principais objetivos da arquitetura é suportar alterações em tempo de execução, o tempo de resposta do *solver* e da implementação da arquitetura é crucial para a qualidade de experiência (QoE) do usuário. Nesta tese, o tempo de resposta é definido como o intervalo temporal entre o instante em que um evento que altera as características do leiaute é capturado pelo *controller* e o momento em que o *controller* emite uma resposta adequada.

Para executar os testes de desempenho, a implementação da arquitetura do StyLe foi levemente alterada para emular o envio dos eventos de início e término da apresentação dos objetos de mídia. Para medir o tempo de resposta, sempre que um evento é recebido pelo componente *controller*, dá-se o início do cálculo do tempo até que o *solver* responda com a nova informação de posicionamento.

Depois de se analisar muitos leiautes de aplicações interativas, disponíveis em sites como o Clube NCL*, percebeu-se que o número de itens em cada leiaute é normalmente em torno de 10, já que se fossem em maior quantidade o tamanho resultante de cada item seria muito pequeno. Além disso, como já foi descrito anteriormente na Seção 3.1.3 do Capítulo 3, a maioria dos leiautes dispõem os objetos de mídia na forma de uma *grid* ou de um *flow*.

Considerando estas características, dois cenários de avaliação foram concebidos. Ambos os cenários consideram um arranjo do tipo grade com uma quantidade variável de itens no leiaute. Esta grade é construída usando-se os modelos *flowLayout* e *gridLayout* em momentos distintos. Quando um modelo *flowLayout* é utilizado, o tamanho dos itens também varia. Em todas as simulações, o tamanho da grade construída no leiaute varia de 2x2 (4 itens) até 15x15 (225 itens). A ideia é avaliar o impacto do número crescente de elementos no tempo de resposta do *solver*.

Todos os testes foram realizados em um computador com as seguintes características: MacBook Pro, Processador 2.5 GHz Intel Core I5, 8 GB 1600 MHz DDR3 RAM e Intel HD Graphics 4000 1536 MB.

7.3.1 Cenário 1

No primeiro cenário, alguns itens específicos foram escolhidos para não serem apresentados. São eles: o primeiro, um item do meio da grade e o último. A Figura 7.3 descreve a ideia, onde os itens A, E e I são os elementos escolhidos. Quando itens não são apresentados, os outros itens no leiaute precisam ser rearranjados, induzindo a mudanças na apresentação.

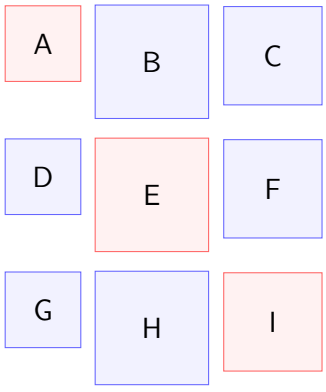

Figura 7.3: Exemplo do Primeiro Cenário

*http://clube.ncl.org.br

Como mencionado, quando a simulação é realizada usando o modelo *flowLayout*, o tamanho dos itens varia de acordo com o tamanho do leiaute e a quantidade de itens na grade. O tamanho usado para o leiaute foi 1920x1080 (full HDTV). Para cada configuração diferente do número de itens, foram calculados valores para as seguintes variáveis: *baseWidth* e *baseHeight*. Por exemplo, se a grade tem tamanho 3x3, a variável *baseWidth* receberia o valor 640 (1920/3) e a variável *baseHeight* receberia o valor 360 (1080/3). Cada item da primeira linha da grade recebeu um tamanho (*width* e *height*) diferente, calculado em função dos valores base, da seguinte forma:

$$width = \lfloor 0.7 * basewidth + random(0.3 * basewidth) \rfloor$$

$$height = \lfloor 0.7 * baseheight + random(0.3 * baseheight) \rfloor$$

Neste cenário, os tamanhos dos itens da primeria linha são repetidos nas outras linha, como mostrado na Figura 7.3.

O maior objetivo deste cenário é verificar se o *solver* é rápido quando as mudanças são simples. As poucas mudanças aparentes são: o tamanho dos itens (no caso do *flowLayout*) e qual item não será apresentado. Já que o tamanho dos itens e os elementos não mostrados são relativamente controlados para todas as rodadas, considera-se que as mudanças são relativamente simples no primeiro experimento.

Não apresentar o último item não deveria interferir no cálculo do leiaute resultante, já que não é necessário rearranjar os outros elementos. Mas, quando o primeiro ou o item do meio não são apresentados, todos os itens que são posicionados depois desses elementos na grade deverão sofrer modificações. Neste caso, o componente *solver* precisa recalcular as posições dos itens para obter as novas informações de posicionamento e isso causa um efeito em cascata.

Para cada modelo de leiaute (*flow* e *grid*) e um número de itens determinado, 50 rodadas do teste foram realizadas. O resultado utilizando o modelo *flowLayout* pode ser visualizado na Figura 7.4, usando-se um intervalo de confiança de 95%.

A primeira informação importante extraída do gráfico é que o tempo de resposta cresce com o número de itens no leiaute, o que já era esperado. Outra observação importante é que identificamos o maior número de itens para o qual um tempo de resposta razoável é encontrado.

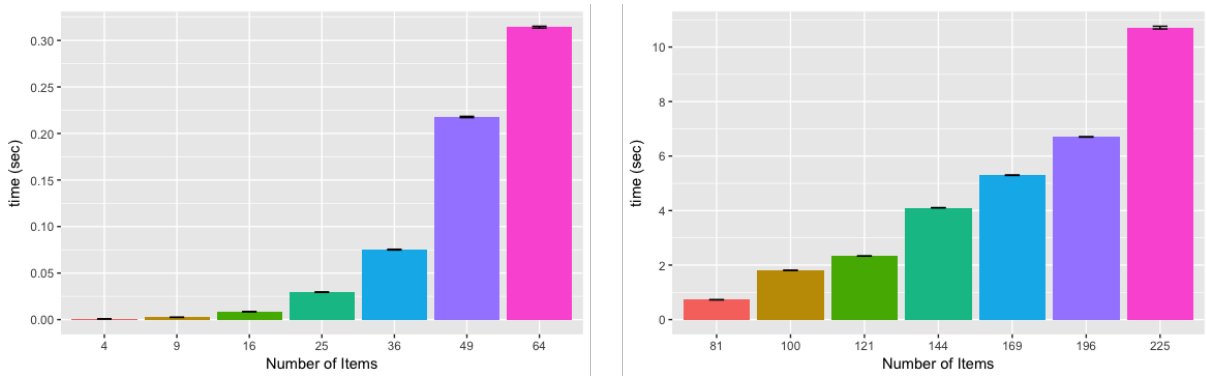

Figura 7.4: Modelo *flowLayout* - Tempo x Número de Itens

De acordo com [51] os tempos de espera para comandos de usuário em aplicações interativas para TV podem ser classificados da seguinte forma:

- menor ou igual a 0.1 s – impressão de continuidade. O usuário tem a sensação de que o sistema reage instantaneamente a seus comandos;
- menor ou igual a 1 s – o processo de pensamento do usuário permanece ininterrupto. O atraso será notado mas não irá interferir na sensação de interação direta com o sistema;
- menor ou igual a 10 s – os usuários ainda vão esperar e prestar atenção no que se passa;
- maior do que 10 s – usuário provavelmente irá desistir, a menos que esteja esperando por uma informação realmente importante, como no caso do resultado de uma transação comercial.

A *British Broadcasting Company* (BBC) adota como tempo de resposta máximo para um comando do usuário o valor de 8 segundos.

Seguindo [51], consideramos um tempo de resposta razoável para uma boa qualidade de experiência do usuário o valor em torno de 1 segundo. Nesse caso, a grade 9x9 (81 itens) é o limite superior, já que grades com uma quantidade maior de elementos obtêm um tempo de resposta maior do que 2 segundos. O resultado é interessante se levarmos em consideração que a quantidade de elementos em aplicações interativas comerciais, tais como guias de informação de programas, é por volta de 10 a 15 itens (grade 3x5).

A Figura 7.5 mostra os resultados de desempenho para leiautes criados com o modelo *gridLayout*, também utilizando um intervalo de confiança de 95%. Como mencionado

anteriormente, o modelo *gridLayout* define itens com o mesmo tamanho. Este tamanho depende do número de linhas, de colunas e do tamanho da grade. Por causa desta característica, a computação realizada pelo modelo *gridLayout* para reajustar o posicionamento dos itens é mais simples que do modelo *flowLayout*. Isto reflete no resultado do tempo de resposta que é consideravelmente menor do que o modelo *flowLayout*. Para o modelo *gridLayout*, todos os tempos são razoáveis, independente do tamanho da grade. É importante notar que a curva de crescimento do tempo de resposta é ainda relacionada ao número de itens na grade, assim como no modelo *flowLayout*.

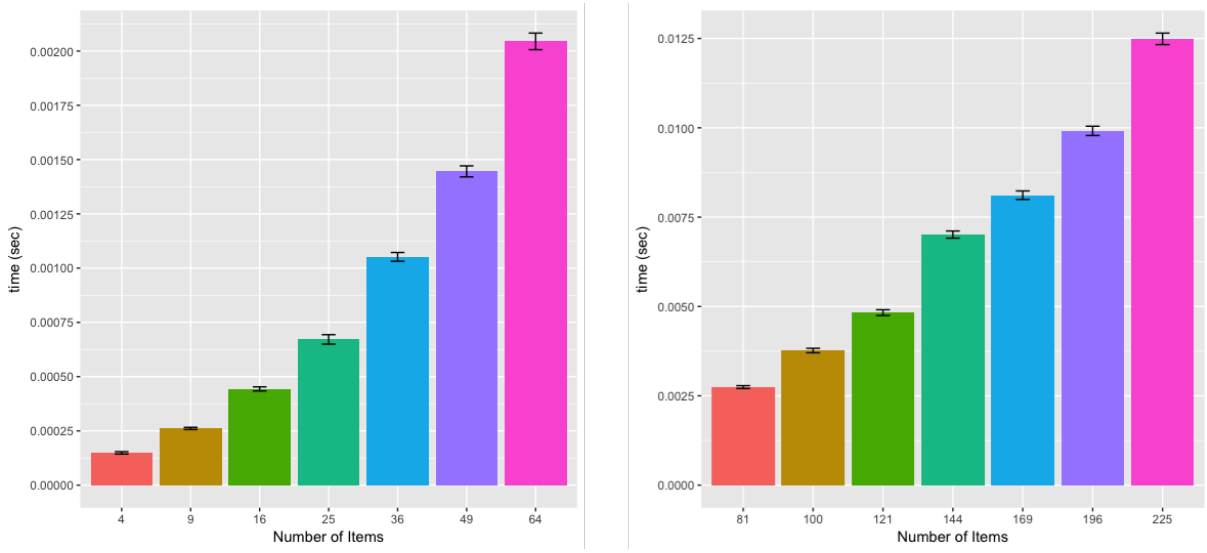

Figura 7.5: Modelo *gridLayout* - Tempo x Número de Itens

7.3.2 Cenário 2

No segundo caso, tentou-se identificar o pior cenário para o tempo de resposta. Os experimentos foram realizados com as seguintes características:

- Os valores para largura e altura de cada item são calculados em cada rodada, seguindo a fórmula especificada anteriormente, e nenhum valor é repetido;
- O número de itens que são apresentados variam de 0% (nenhum item é apresentado) até 100% (todos os itens são apresentados);
- Para um dado número de itens não apresentados, a escolha de qual item não será apresentado é aleatória e feita a cada rodada;

Por exemplo, em uma grade 4x4 (16 itens), todos os itens podem ter tamanhos diferentes e o tempo de resposta é calculado quando: nenhum item é apresentado (0%), 1 item é apresentado (10%), 3 itens são apresentados (20%) e assim por diante, até que todos os itens sejam apresentados (100%). Os itens que não serão apresentados são escolhidos aleatoriamente, rodada a rodada. No final, calcula-se a média das 50 rodadas para cada quantidade de itens apresentados.

O gráfico mostrado no Figura 7.6 apresenta o tempo de resposta para diferentes leiautes de acordo com o número de itens apresentados. Os resultados foram normalizados para que fossem apresentados juntos. A normalização realizada consiste em dividir os tempos de resposta, para um dado leiaute, pelo maior valor. A normalização nos ajuda a tentar identificar o comportamento da curva do tempo de resposta.

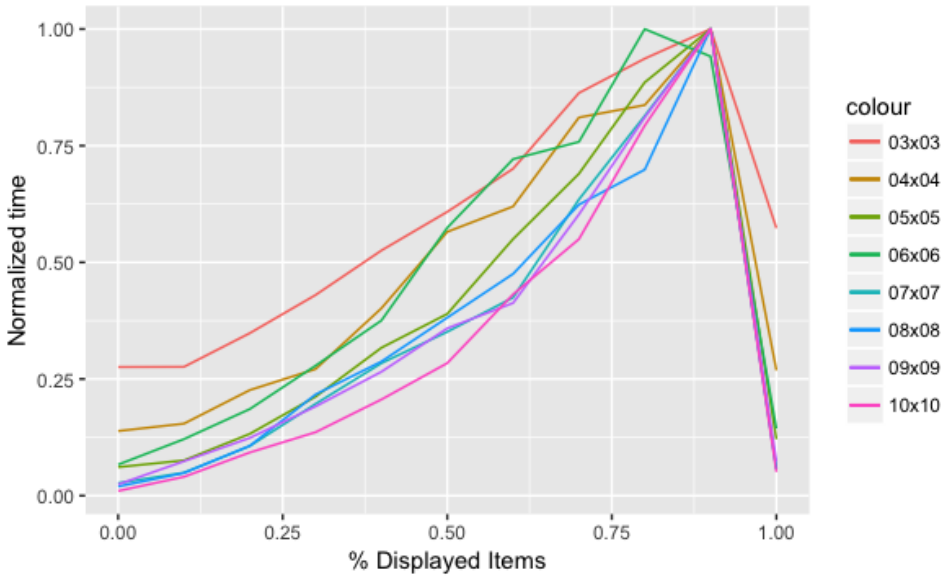

Figura 7.6: Modelo *flowLayout* - Tempo(seg) x % Itens apresentados

Pode-se observar que o comportamento das curvas do tempo de resposta para diferentes tamanhos de grade seguem o mesmo padrão. Ela cresce quando o número de itens apresentados cresce até 90% e depois cai. Este comportamento pode ser explicado porque, com 90% dos itens sendo apresentados, a quantidade de cálculos que devem ser realizados quando um certo número de itens é interrompido é grande. Então, é possível considerar que, para o modelo *flowLayout*, a taxa de 90% de itens apresentados é o pior caso, cujos valores para o tempo de resposta são apresentados na Tabela 7.1.

Como pode ser visto, mesmos no cenário de pior caso, a implementação alcança um resultado muito bom já que a grade 8x8 com 64 itens pode ser rearranjada em menos de 1 segundo. A grade com 81 itens leva aproximadamente 2 segundos.

Num. de Itens	9	16	25	36	49	64	81	100
Tempo em seg	0.0056	0.0138	0.0557	0.1802	0.3213	0.7231	2.0794	2.8377

Tabela 7.1: Valores de tempo de resposta para o Modelo *flowLayout* com 90% de itens apresentados

Os mesmos testes foram realizados utilizando-se o modelo *gridLayout* para construir o leiaute. Os valores são mostrados na Figura 7.7. Pode-se notar que as curvas do tempo de resposta apresentam o mesmo padrão identificado anteriormente. Para comparação, a Tabela 7.2 apresenta os valores de tempo de resposta referentes a 90% dos itens apresentados.

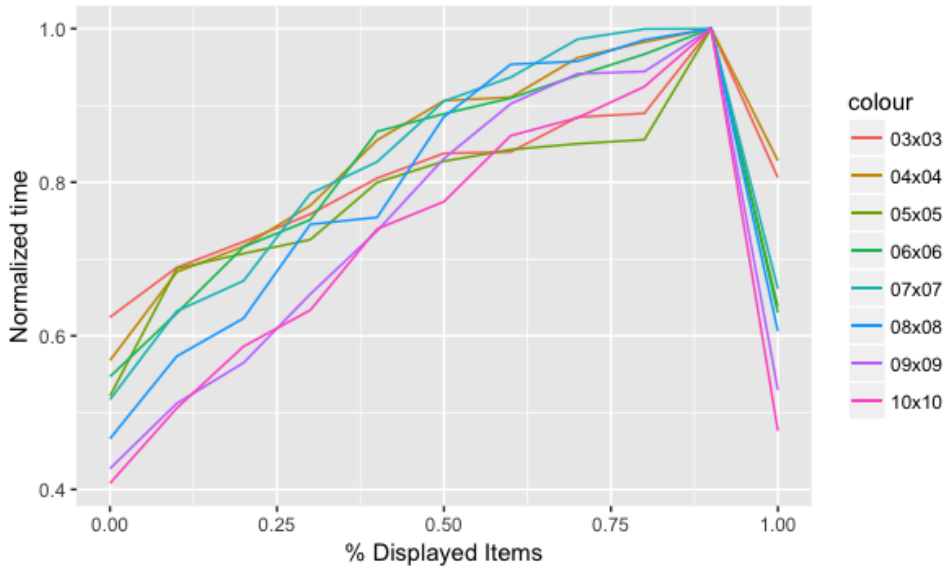

Figura 7.7: Modelo *gridLayout* - Tempo(seg) x % Itens Apresentados

Num. de Itens	9	16	25	36	49	64	81	100
Tempo em seg	0.00033	0.00059	0.00102	0.00170	0.00242	0.00365	0.00548	0.00725

Tabela 7.2: Valores de tempo de resposta para o Modelo *gridLayout* com 90% de itens apresentados

Os valores do tempo de resposta para o *gridLayout* são sempre muito pequenos. Isto é explicado porque o modelo é bem comportado o que facilita os cálculos realizados pelo *solver*.

Os resultados de desempenho da implementação da arquitetura proposta para uso de StyLe com NCL são promissores e mostram que a arquitetura tem potencial para ser usado na prática por aplicações NCL. Ambos os modelos, *flowLayout* e *gridLayout* podem ser utilizados para determinar o leiaute da aplicação, mesmo quando vários objetos de mídia são apresentados.

Apesar dos ótimos resultados iniciais, os testes foram realizados em um dispositivo com um alto poder computacional, o que não será necessariamente o cenário encontrado. Por isso, como trabalhos futuros, será preparado um outro ambiente de teste que se aproxima da realidade comum dos dispositivos de apresentação. O *middleware* Ginga será instalado em um dispositivo Raspberry Pi junto com a arquitetura STyLe. Além disso, serão avaliados os outros modelos de leiautes disponíveis na linguagem.

Capítulo 8

Conclusão

Geralmente, linguagens de programação suportam gerenciadores de leiaute para arrumar componentes gráficos da interface do usuário automaticamente e adaptar a apresentação dos elementos quando alterações ocorrem em tempo de execução, tais como mudança de tamanho de uma janela. Essa facilidade permite que o usuário tenha uma experiência mais agradável quando interage com a interface.

Algumas linguagens declarativas multimídia, tais como HTML5 e CSS3, apresentam elementos que facilitam a autoria na declaração de leiautes dinâmicos, mas as relações espaciais criadas entre esses elementos são básicas, geralmente associando posicionamentos ou tamanhos entre objetos de mídia, o que limita a expressividade.

Outras linguagens como NCL e SMIL não fornecem esta facilidade e os autores precisam declarar as características de apresentação para cada objeto de mídia separadamente, além de ter que construir diferentes leiautes para diferentes tipos de dispositivos. Nessas linguagens, a falta de elementos de alto nível que facilitem a autoria da aplicação, principalmente para especificação de relacionamentos espaciais, contribuem para que a estrutura do documento seja mais complexa, levando a um possível aumento de erros na autoria.

Apesar de ser possível alterar as informações de apresentação de objetos de mídia em tempo de execução em NCL e SMIL, o autor deve prever todas as alterações possíveis, o que torna a autoria extremamente trabalhosa.

Esta tese propôs resolver essas questões utilizando uma abordagem que combina duas técnicas, que são: utilização de especificações genéricas de programas na definição do sincronismo espacial e desenvolvimento e implementação de uma arquitetura para prestar serviços de adaptação de leiautes em tempo de execução.

Especificações genéricas de programas, como modelos pré-definidos, podem ser utilizados para criação de *templates*. Esses *templates* formam uma base de documentos que possuem as mesmas características e que podem ser reutilizados em diferentes aplicações multimídia, gerando redução no tamanho do código e redução em possíveis erros de autoria.

Embora existam algumas linguagens baseadas em *templates* para criação de documentos multimídia interativos, ainda há espaço para melhorias, principalmente no que diz respeito a aspectos relacionados às características de apresentação de componentes visuais de um documento multimídia. Esta tese preencheu essa lacuna propondo uma solução que une definição de características visuais e linguagens baseadas em *templates*.

Nos dias atuais, os usuários consomem informação nas mais variadas plataformas. Isso implicou uma evolução em como se constrói uma aplicação voltada para o usuário. Essa evolução impulsionou a transformação dos modos de interação entre o usuário e a aplicação e, por consequência, as interfaces que proveem essa interação. Um usuário pode inserir ou remover objetos da aplicação enquanto a executa. Para que a inserção e remoção de elementos não afete a experiência do usuário ao interagir com a aplicação, é necessário criar um conjunto de serviços que possa reagir a essas alterações mantendo a qualidade de apresentação da aplicação. Esta facilidade é muito interessante em ambientes que dão suporte a interatividade como sistemas para TV Digital interativa.

Foi desenvolvida nesta tese uma nova DSL, baseada em XML, para a definição de modelos de leiautes adaptativos e dinâmicos, denominada STyLe. Esta linguagem permite uma diminuição da complexidade e, consequentemente, do esforço de autoria de documentos multimídia declarativos, além de diminuir a possibilidade de erros do autor, pois usa uma abordagem baseada em *templates* que utilizam modelos predefinidos para definição dos leiautes. Os modelos especificam comportamentos para as características de apresentação dos objetos de mídia que os referenciam, como: regiões onde serão apresentados, navegação entre os objetos e parâmetros como transparência do objeto.

Para prover as características de adaptação, tanto em relação a quantidade de objetos de mídia quanto em relação às modificações que ocorrem em tempo de execução, STyLe define o leiaute espacial de um documento multimídia através de restrições espaciais que consideram o estado dos objetos de mídia. Além disso, o uso de restrições em STyLe pode também ser utilizado para adaptar a aplicação aos diferentes contextos de apresentação como diferentes tamanhos de tela.

Para facilitar a declaração do leiaute espacial de um documento, STyLe apresentou um conjunto de restrições espaciais predefinidas e utilizou *templates* e modelos de leiaute. A ideia é fornecer uma versão dinâmica dos modelos para que o leiaute espacial possa utilizá-los, mesmo quando mudanças ocorrerem na aplicação em tempo de execução. Adicionalmente às restrições predefinidas, STyLe ainda permite que o autor personalize tipos de restrições espaciais através de conectores de restrição, o que confere uma maior flexibilidade e expressividade à linguagem.

Vale ressaltar que STyLe é uma linguagem baseada em *templates*. Por isso, STyLe possui como características a diminuição do esforço de autoria e da quantidade de erros. Como mencionado, o autor precisa identificar quais componentes do documento farão parte dos relacionamentos espaciais sem a necessidade de descrevê-los por completo.

Com o intuito de fornecer um conjunto de serviços capazes de interpretar as restrições espaciais e construir corretamente o leiaute espacial de acordo com o número de elementos do documento, foi apresentada uma arquitetura capaz de realizar tal tarefa. A arquitetura proposta é capaz de atualizar o leiaute espacial em tempo de execução.

É importante lembrar que, apesar do objetivo de STyLe ser definir somente o leiaute espacial de apresentação do documento, é possível agregar STyLe com uma linguagem de *templates* de composição para que relações temporais entre objetos de mídia no documento também sejam estabelecidos provendo uma solução completa.

Como prova de conceito, a linguagem de autoria multimídia NCL e a linguagem de *templates XTemplate* foram estendidas para utilizarem STyLe na concepção dos leiautes espaciais dos documentos e alguns exemplos de utilização de STyLe nessas linguagens foram apresentados.

Além disso, testes para medir o desempenho da arquitetura proposta para STyLe, medir a usabilidade da linguagem e medir o esforço de autoria foram realizados. Foi avaliado o tempo de resposta do componente *solver*, essencial para o bom desempenho da arquitetura. Através dos testes foi possível notar que a arquitetura tem potencial para ser usada na prática por aplicações NCL. Os modelos testados *flowLayout* e *gridLayout* podem ser utilizados para especificar o leiaute de uma aplicação na prática, mesmo quando vários objetos de mídia são apresentados. Além disso, os testes de usabilidade e de esforço mostraram que a linguagem tem potencial para ser utilizada por autores com qualquer nível de experiência em programação multimídia.

Em resumo, ao se atingir os objetivos principais destacados nesta tese, as contribuições alcançadas foram:

1. Linguagem StyLe para autoria de modelos de leiaute adaptativos e dinâmicos baseados em restrições espaciais para documentos multimídia;
2. Extensão de NCL para uso da linguagem StyLe;
3. Implementação do processador da linguagem StyLe;
4. Extensão de XTemplate para uso da linguagem StyLe;
5. Definição de uma arquitetura para a linguagem StyLe que possibilite a criação de leiautes dinâmicos que respondem às alterações das características de apresentação de documentos NCL em tempo de exibição e a implementação da arquitetura proposta.

8.1 Trabalhos Futuros

Um dos primeiros trabalhos futuros é avaliar a arquitetura proposta em um ambiente de execução mais realista, como em um dispositivo com poder computacional e memória restrito, tal como são os receptores de TV digital. Para isso, pode-se instalar o *middleware* Ginga e a arquitetura StyLe em um dispositivo Raspberry Pi com um processador Quad-Core ARM 1.2 GHz e 1 GB de memória RAM. A memória utilizada pela implementação também é um quesito que é importante de ser avaliado, pois o dispositivo de exibição normalmente possui uma capacidade de armazenamento limitada.

Além disso, se faz necessário avaliar o desempenho da arquitetura com os outros modelos propostos *stackLayout* e *carouselLayout* e leiautes mais complexos como composições com vários modelos, além da ordem em que os objetos de mídia reaparecem depois de terem sido parados uma vez. Atualmente, essa ordem segue a ordem inicial, isto é, as mídias voltam na mesma posição de antes da parada. Outro aspecto importante para se avaliar é a interpretação de um leiaute que usa StyLe pelo usuário. Essa avaliação indicará se o autor consegue entender bem um leiaute a partir de sua especificação em StyLe.

Outro trabalho futuro é usar StyLe para estender outras linguagens de autoria, tais como HTML5 e SMIL, além de NCL. Outra ideia a ser seguida é a simplificação da de-

claração de conectores de restrição espacial, talvez usando açúcar sintático*, como nomes predefinidos para os papeis.

Ainda é possível melhorar a implementação da arquitetura, facilitando a proposta de outros modelos de leiautes. Isso pode ser feito criando uma API que receba o novo modelo como um *plugin*. A vantagem é que uma inserção de um novo modelo de leiaute pode ser feito sem alterar muito o código.

Outro fator importante é avaliar a usabilidade da linguagem proposta em sua totalidade. Para facilitar o uso de StyLe por programadores inexperientes, uma ferramenta gráfica deve ser produzida para que o programador possa criar um *template* de leiaute arrastando e posicionando elementos gráficos. Essa ferramenta deverá ser capaz de realizar pré-visualizações sobre a não satisfiabilidade de restrições de um certo nível em tempo de edição. Isso ajudará o autor da leiaute a prever situações inesperadas.

Ainda outro trabalho futuro é atualizar as ferramentas de autoria gráfica de documentos NCL e de templates de composição, como por exemplo o editor EDITEC, que permite a edição gráfica de templates, atualizando-o para a versão 4.0 de XTemplate; o editor STeVE, que permite a autoria gráfica da visão temporal de documentos NCL e HTML5, incluindo a facilidade de uso de templates StyLe e o editor gráfico NEXT para edição de documentos NCL com XTemplate 4.0 e StyLe.

Um outro trabalho futuro interessante é estender StyLe para a especificação de leiautes em três dimensões, e também para sua utilização em linguagens de autoria multimídia que permitem a definição de efeitos sensoriais, permitindo o uso de StyLe para determinar a posição em que os efeitos sensoriais devem ser executados em um ambiente físico.

*reescrever (código de computador) de uma forma mais refinada e concisa; remover todos os elementos sintáticos desnecessários do (código do computador) - The Free Dictionary by Farlex

Referências

[1] Google, "Android studio," https://developer.android.com/studio/index.html, 2016, visitado em Março.

[2] N. P. de Koch, "Software engineering for adaptive hypermedia systems: Reference model, modeling techniques and development process," Ph.D. dissertation, Ludwig-Maximilian University of Munich, 2001.

[3] G. F. Amorim, J. A. F. dos Santos, and D. C. Muchaluat-Saade, "Adaptive layouts and nesting templates for hypermedia composite templates," in *Proceedings of the 21st Brazilian Symposium on Multimedia and the Web*, ser. WebMedia '15, 2015.

[4] J. A. F. dos Santos and D. C. Muchaluat-Saade, "Xtemplate 3.0: Spatio-temporal semantics and structure reuse for hypermedia compositions," *Multimedia Tools Appl.*, vol. 61, no. 3, pp. 645–673, dec 2012.

[5] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, and S. Pfeiffer, "HTML5: A vocabulary and associated APIs for HTML and XHTML," http://www.w3.org/TR/html5/, October 2014.

[6] B. Bos and E. A. Mayer, "Introduction to css3," http://www.w3.org/TR/2001/WD-css3-roadmap-20010523, May 2001.

[7] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey, "Constraint cascading style sheets for the web," in *UIST*, 1999, pp. 73–82.

[8] D. Tocchini, G. Brados, A. Russel, H. Bergious, T. Aylott, Y. Fedin, Y. Paul, and C. Chouinard, "Grid Style Sheets 2.0," http://gridstylesheets.org/guides/ccss/, 2016, Grid Style Sheets Organization.

[9] ABNT, *Data coding and transmission specification for digital broadcasting Part 2: Ginga-NCL for fixed and mobile receivers — XML application language for application coding*, ABNT 15606-2:2007, 2011.

[10] ITU, "Nested Context Language (NCL) and Ginga-NCL for IPTV services," http://www.itu.int/rec/T-REC-H.761-200904-P, 2009, iTU-T Recommendation H.761.

[11] W3C, "Synchronized multimedia integration language - smil 3.0 specification," http://www.w3c.org/TR/SMIL3, 2008, w3C Recommendation.

[12] D. C. Muchaluat-Saade and L. F. G. Soares, "Xconnector & xtemplate: Improving the expressiveness and reuse in web authoring languages," *The New Review of Hypermedia and Multimedia Journal*, vol. 8, no. 1, pp. 139–169, 2002.

[13] C. S. Neto, H. F. Pinto, and L. F. G. Soares, "TAL Processor of Hypermedia Applications," in *Proceedings of the 2012 ACM Symposium on Document Engineering*, ser. DocEng '12. ACM, 2012, pp. 69–78.

[14] G. F. Amorim, J. A. F. dos Santos, and D. C. Muchaluat-Saade, "Xtemplate 4.0: Providing adaptive layouts and nested templates for hypermedia documents," in *MultiMedia Modeling: 22nd International Conference*, ser. MMM '16, 2016, pp. 642–653.

[15] C. Jacobs, W. Li, E. Schirier, D. Bargeron, and D. Salesin, "Adaptive grid-based document layout," *ACM Transactions on Graphics*, vol. 22, no. 3, July 2003.

[16] E. Schrier, M. Dontcheva, C. Jacobs, G. Wade, and D. Salesin, "Adaptive layout for dynamically aggregated documents," in *IUI '08*. ACM, 2008, pp. 99–108.

[17] I. Albert, H. Charae, and L. Lengyel, "Layout definition considerations for a content-driven template-based layout system," in *EUROCOM 2013*. IEEE, 2013.

[18] R. Atanassov and A. Stearns, "CSS Regions Module Level 1," http://www.w3.org/TR/css-regions-1/, 2014, w3C Working Draft.

[19] S. S. C. Vanoirbeek, V. Quint and C. Roisin, "A lightweight framework for authoring XML multimedia content on the web," *MTAP*, vol. 70, pp. 1229–1250, 2014.

[20] E. Spyrou, D. Iakovidis, and P. Mylonas, *Semantic Multimedia Analysis and Processing*. CRC Press, 2014.

[21] M. S. A. de Moura, "Relações espaciais em documentos hipermídia," Master's thesis, PUC-Rio (in portuguese), 2001.

[22] M. Vazirgiannis, Y. Theodoridis, and T. Sellis, "Spatio-temporal composition and indexing for large multimedia applications," *Multimedia Systems*, vol. 6, no. 4, pp. 284–298, 1998.

[23] D. A. Randell, Z. Cui, and A. G. Cohn, "A spatial logic based on regions and connection," in *Proceedings 3RD Iinternational Conference on Knowledge Representation and Reasoning*, 1992.

[24] D. Papadias, T. Sellis, Y. Theodoridis, and M. J. Egenhofer, "Topological relations in the world of minimum bounding rectangles: A study with r-trees," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '95. ACM, 1995, pp. 92–103.

[25] M. J. Egenhofer and R. D. Franzosa, "Point-set topological spatial relations," *International Journal of Geographical Information System*, vol. 5, no. 2, pp. 161–174, 1991.

[26] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, nov 1983.

[27] J. A. F. dos Santos, C. Braga, D. C. Muchaluat-Saade, C. Roisin, and N. Layaïda, "Spatio-temporal validation of multimedia documents," in *Proceedings of the 2015 ACM Symposium on Document Engineering*, ser. DocEng '15. ACM, 2015, pp. 133–142.

[28] R. Deltour and C. Roisin, "The limsee3 multimedia authoring model," in *Proceedings of the 2006 ACM Symposium on Document Engineering*, ser. DocEng '06. ACM, 2006, pp. 173–175.

[29] I. M. Bilasco, J. Gensel, and M. Villanova-Oliver, "STAMP: a model for generating adaptable multimedia presentations," *Journal of Multimedia Tools and Applications*, vol. 25, no. 3, March 2005.

[30] H. V. de O. e Silva, "X-smil: Aumentando reuso e expressividade em linguagens de autoria hipermidia," Master's thesis, PUC-Rio (in portuguese), Rio de Janeiro, Brasil, April 2005.

[31] J. Clark, "XSL Transformations (XSLT) version 1.0," http://www.w3.org/TR/xslt, 1999, W3C Recommendation.

[32] J. Clark and S. DeRose, "XML Path language (XPath) version 1.0," http://www.w3.org/TR/xpath, 1999, W3C Recommendation.

[33] W3Schools, "Responsive Web Design Introduction," http://www.w3schools.com/css/css_rwd_intro.asp, 2015.

[34] G. F. Amorim, J. A. F. dos Santos, and D. C. Muchaluat-Saade, "Style: Extending ncl for providing dynamic layouts," in *Proceedings of the 22Nd Brazilian Symposium on Multimedia and the Web*, ser. Webmedia '16. ACM, 2016, pp. 71–78.

[35] L. F. G. Soares, C. S. S. Neto, and J. G. S. Junior, "Architecture for hypermedia dynamic applications with content and behavior constraints," in *Proceedings of the 2012 ACM Symposium on Document Engineering*, ser. DocEng '12. ACM, 2012.

[36] ——, "Framework for automatic generation of hypermedia applications in runtime," in *Proceedings of the 20Nd Brazilian Symposium on Multimedia and the Web*, ser. WebMedia '14, 2014.

[37] M. F. Moreno, C. D. S. S. Neto, F. Nagato, and L. F. G. Soares, "Uma abordagem declarativa para geração e adaptação de aplicações de guias eletrônicos de programação," in *Proceedings of the 14 Brazilian Symposium on Multimedia and the Web*, ser. WebMedia '08, 2008.

[38] A. Borning, R. K.-H. Lin, and K. Marriott, "Constraint-based document layout for the web," *Multimedia Systems*, vol. 8, no. 3, pp. 177–189, 2000.

[39] J. P. T. M. Geurts, J. R. van Ossenbruggen, and H. L. Hardman, *Application-specific constraints for multimedia presentation generation.* Centrum voor Wiskunde en Informatica, 2001.

[40] P. King, P. Schmitz, and S. Thompson, "Behavioral reactivity and real time programming in xml: Functional programming meets smil animation," in *Proceedings of the 2004 ACM Symposium on Document Engineering*, ser. DocEng '04. ACM, 2004, pp. 71–78.

[41] C. McCormack, K. Marriott, and B. Meyer, "Adaptive layout using one-way constraints in svg," in *SVG Open*, 2004.

[42] W3C, "Xml schema part 0: Primer second edition," http://www.w3.org/TR/xmlschema-0/, 2004, w3C Recommendation.

[43] C. NCL, "Clube NCL: a liberdade de desenvolver e compartilhar conteúdo interativo," http://clube.ncl.org.br/.

[44] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories." *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[45] L. D. Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[46] D. P. de Mattos, J. V. da Silva, and D. C. Muchaluat-Saade, "Next: Graphical editor for authoring ncl documents supporting composite templates," in *Proceedings of the 11th European Conference on Interactive TV and Video*, ser. EuroITV '13, 2013, pp. 89–98.

[47] R. Ierusalimschy, *Programming in Lua, 3ª Edition.* Lua.org, 2013.

[48] B. Dutertre, *Yices Manual Version 2.4*, SRI International, December 2015.

[49] ——, *Yices 2.2.* Springer International Publishing, 2014.

[50] A. Blackwell and T. Green, *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science.* Morgan Kaufmann, 2003.

[51] M. Gawlinski, *Interactive television production.* Taylor & Francis, 2003.

APÊNDICE A – Códigos completos dos exemplos apresentados no texto

A listagem A.1 mostra o código completo do exemplo apresentado no Capítulo XTemplate 4.0.

```
<!-- Codigo do template quiz.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<xt:xtemplate id="quiz" description="Quiz template"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xt="http://www.midiacom.uff.br/gtvd/XTemplate40/XTemplateENCL"
 xsi:schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/XTemplateENCL http://www.midiacom.uff.
    br/gtvd/XTemplate40/profiles/XTPENCL.xsd">

<head>
    <connectorBase>
        <importBase alias="connectors" documentURI="connectorBase.ncl"/>
    </connectorBase>
    <descriptorBase >
        <importBase alias="descriptors" documentURI="descriptorBase.ncl"/>
    </descriptorBase>
    <templateBase>
        <importBase alias="scn" documentURI="screen.xml"/>
    </templateBase>
</head>

<vocabulary>
    <component xlabel="video" descriptor="descriptors#dp_video"/>
    <component xlabel="background" minOccurs="1" maxOccurs="1" descriptor="descriptors#dp_quiz"/>
    <component xlabel="screen" xtype="xtemplate/scn">
        <port xlabel="portAnswer"/>
    </component>
    <component xlabel="counter" xtype="application/x-ginga-NCLua" minOccurs="1" maxOccurs="1"
     descriptor="descriptors#dp_lua">
            <port xlabel="answer"/>
            <port xlabel="correct"/>
    </component>
    <component xlabel="win"/>
    <component xlabel="lose"/>

    <connector src="connectors#onFirstKeySelectionStart" xlabel="start_quiz"/>
    <connector src="connectors#onKeySelectionPresentingStopSetStart" xlabel="change_screen"/>
    <connector src="connectors#onEndTestGTEStopStart" xlabel="last_screen_win"/>
    <connector src="connectors#onEndTestLTStopStart" xlabel="last_screen_lose"/>
</vocabulary>

<body>
    <port id="port" select="child::*[@xlabel='video']"/>

        <media id="bkg" type="image/png" src="bkg.png" xlabel="background"/>

    <media id="ctrlua" src="main.lua" type="application/x-ginga-NCLua" xlabel="counter">
            <property name="answer" xlabel="answer"/>
            <property name="correct" xlabel="correct"/>
```

```
48      </media>
49
50      <!-- First Link -->
51      <link xtype="start_quiz">
52          <bind role="onSelection" select="child::*[@xlabel='video']">
53              <bindParam name="keyCode" value="YELLOW"/>
54          </bind>
55          <bind role="occurrences" select="child::*[@xlabel='screen'][position()=1]"/>
56                  <bind role="start" select="child::media[@xlabel='background']"/>
57          <bind role="start" select="child::media[@xlabel='counter']"/>
58          <bind role="start" select="child::*[@xlabel='screen'][position()=1]"/>
59      </link>
60
61
62      <!-- ########## Changing screens ########## -->
63      <variable name="i" select="1"/>
64      <for-each select="child::*[@xlabel='screen'][position()!=last()]">
65
66          <!-- link for the RED key -->
67          <link xtype="change_screen">
68              <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
    position()=1]"/>
69              <bind role="isPresenting" select="current()"/>
70              <bind role="stop" select="current()"/>
71              <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']"
    >
72                  <bindParam name="var" value="A"/>
73              </bind>
74              <bind role="start" select="child::*[@xlabel='screen'][position()=$i+1]"/>
75          </link>
76
77                  <!-- link for the GREEN key -->
78          <link xtype="change_screen">
79              <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
    position()=2]"/>
80              <bind role="isPresenting" select="current()"/>
81              <bind role="stop" select="current()"/>
82              <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']"
    >
83                  <bindParam name="var" value="B"/>
84              </bind>
85              <bind role="start" select="child::*[@xlabel='screen'][position()=$i+1]"/>
86          </link>
87
88                  <!-- link for the YELLOW key -->
89          <link xtype="change_screen">
90              <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
    position()=3]"/>
91              <bind role="isPresenting" select="current()"/>
92              <bind role="stop" select="current()"/>
93              <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']"
    >
94                  <bindParam name="var" value="C"/>
95              </bind>
96              <bind role="start" select="child::*[@xlabel='screen'][position()=$i+1]"/>
97          </link>
98
99                  <!-- link for the BLUE key -->
100         <link xtype="change_screen">
101             <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
    position()=4]"/>
102             <bind role="isPresenting" select="current()"/>
103             <bind role="stop" select="current()"/>
104             <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']"
    >
105                 <bindParam name="var" value="D"/>
106             </bind>
107             <bind role="start" select="child::*[@xlabel='screen'][position()=$i+1]"/>
108         </link>
109
110         <variable name="i" select="$i+1"/>
111     </for-each>
112
```

```
113     <!-- # # # # # # # # # #  Last Screen links  # # # # # # # # # # -->
114         <!-- link for the RED key -->
115     <link xtype="change_screen">
116         <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
        position()=1]"/>
117         <bind role="isPresenting" select="child::*[@xlabel='screen'][position()=last()]"/>
118         <bind role="stop" select="child::*[@xlabel='screen'][position()=last()]"/>
119                 <bind role="stop" select="child::media[@xlabel='background']"/>
120         <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']">
121             <bindParam name="var" value="A"/>
122         </bind>
123     </link>
124 
125         <!-- link for the GREEN key -->
126     <link xtype="change_screen">
127         <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
        position()=2]"/>
128         <bind role="isPresenting" select="child::*[@xlabel='screen'][position()=last()]"/>
129         <bind role="stop" select="child::*[@xlabel='screen'][position()=last()]"/>
130                 <bind role="stop" select="child::media[@xlabel='background']"/>
131         <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']">
132             <bindParam name="var" value="B"/>
133         </bind>
134     </link>
135 
136         <!-- link for the YELLOW key -->
137     <link xtype="change_screen">
138         <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
        position()=3]"/>
139         <bind role="isPresenting" select="child::*[@xlabel='screen'][position()=last()]"/>
140         <bind role="stop" select="child::*[@xlabel='screen'][position()=last()]"/>
141                 <bind role="stop" select="child::media[@xlabel='background']"/>
142         <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']">
143             <bindParam name="var" value="C"/>
144         </bind>
145     </link>
146 
147         <!-- link for the BLUE key -->
148     <link xtype="change_screen">
149         <bind role="onSelection" select="child::*[@xlabel='screen']/child::*[@xlabel='portAnswer'][
        position()=4]"/>
150         <bind role="isPresenting" select="child::*[@xlabel='screen'][position()=last()]"/>
151         <bind role="stop" select="child::*[@xlabel='screen'][position()=last()]"/>
152                 <bind role="stop" select="child::media[@xlabel='background']"/>
153         <bind role="set" select="child::media[@xlabel='counter']/child::property[@xlabel='answer']">
154             <bindParam name="var" value="D"/>
155         </bind>
156     </link>
157 
158 
159         <!-- # # # # # # # # # #  Closing the quiz  # # # # # # # # # # -->
160     <link xtype="last_screen_win">
161         <bind role="onEnd" select="child::*[@xlabel='screen'][position()=last()]"/>
162         <bind role="test" select="child::media[@xlabel='counter']/child::property[@xlabel='correct']">
163             <bindParam name="var" select="count(child::*[@xlabel='screen'])*0.7"/>
164         </bind>
165         <bind role="stop" select="child::*[@xlabel='screen']"/>
166                 <bind role="stop" select="child::media[@xlabel='counter']"/>
167         <bind role="start" select="child::*[@xlabel='win']"/>
168     </link>
169 
170     <link xtype="last_screen_lose">
171         <bind role="onEnd" select="child::*[@xlabel='screen'][position()=last()]"/>
172         <bind role="test" select="child::media[@xlabel='counter']/child::property[@xlabel='correct']">
173             <bindParam name="var" select="count(child::*[@xlabel='screen'])*0.7"/>
174         </bind>
175         <bind role="stop" select="child::*[@xlabel='screen']"/>
176                 <bind role="stop" select="child::media[@xlabel='counter']"/>
177         <bind role="start" select="child::*[@xlabel='lose']"/>
178     </link>
179 
180 </body>
181 
```

```
</xt:xtemplate>

<!-- Codigo do Template screen.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xt:xtemplate id="screen" description="screen template"
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
   xmlns:xt='http://www.midiacom.uff.br/gtvd/XTemplate40/XTemplateENCL'
   xsi:schemaLocation='http://www.midiacom.uff.br/gtvd/XTemplate40/XTemplateENCL http://www.midiacom.
     uff.br/gtvd/XTemplate40/profiles/XTPENCL.xsd'>
<head>
    <layoutBase>
        <importBase alias="lay" documentURI="quizLayout.xml"/>
    </layoutBase>
</head>
<vocabulary>
    <port xlabel="portAnswer"/>
    <component xlabel="question" layout="lay#qstFl"/>
    <component xlabel="answer" layout="lay#ansFl"/>
</vocabulary>

<body>
    <variable name="i" select="1"/>
    <for-each select="child::media[@xlabel='answer']">
        <port id="port" select="current()" xlabel="portAnswer"/>
        <variable name="i" select="$i+1"/>
    </for-each>
</body>

</xt:xtemplate>

<!-- Codigo do Template de Leiaute quizLayout.xml -->
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<layout id="quizLayout" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:layout="http://www.
    midiacom.uff.br/gtvd/LTemplate/Layout" xsi:schemaLocation=" http://www.midiacom.uff.
    br/gtvd/LTemplate/Layout http://www.midiacom.uff.br/gtvd/LTemplate/profiles/LTPLayout.xsd">
    <container id="qstFl" type="flowLayout">
        <format align="center" height="240" hspace="0" left="10" top="20" vspace="0" width="780"
    zIndex="3"/>
        <item id="c1" height="100" width="780" />
    </layout>
    <container id="aswFl" type="flowLayout">
        <format align="center" height="640" hspace="10" left="10" top="300" vspace="0" width="200"
    zIndex="3"/>
        <item id="c1" height="150" width="200" />
        <focus focusIndex="1"/>
    </layout>
</layout>
```

Listing A.1: Código Completo dos Exemplos do Capítulo XTemplate 4.0

As Listagens A.2 e A.3 mostram os códigos utilizados para criar o exemplo apresentado primeiramente no Capítulo 3 e aproveitado em outros capítulos.

```
<layout id="LayoutAppExample">
  <head>
    <constraintConnectorBase>
      <constraintConnector id="alignLeftBottomTop">
        <compoundStatement operator="and">
          <assessmentStatement comparator="eq">
            <attributeAssessment role="left1" eventType="attribution" attributeType="nodeProperty"/>
            <attributeAssessment role="left2" eventType="attribution" attributeType="nodeProperty"/>
          </assessmentStatement>
          <assessmentStatement comparator="eq">
            <attributeAssessment role="base" eventType="attribution" attributeType="nodeProperty"/>
            <attributeAssessment role="base50" eventType="attribution" attributeType="nodeProperty"
    offset="50"/>
          </assessmentStatement>
        </compoundStatement>
      </constraintConnector>
    </constraintConnectorBase>
```

```
  </head>
  <body>
    <container id="menu" type="gridLayout" focusIndex="1">
      <format top="0" left="0" width="120" height="240" columns="1" rows="4"
        vspace="0" hspace="0"/>
    </container>

    <container id="grid" type="gridLayout" focusIndex="1">
      <format width="500" height="300" columns="4"
        rows="3" vspace="0" hspace="0"/>
    </container>

    <item id="info" width="190" height="380"/>

    <container id="videos" type="flowLayout" focusIndex="1">
      <format width="860" height="100" hspace="20"/>
      <item id="item1" width="200" height="100"/>
    </container>

    <spatialConstraint id="dist1" type="distribute" direction="horizontal">
      <bind component="menu"/>
      <bind component="grid"/>
      <bind component="info"/>
    </spatialConstraint>

    <spatialConstraint id="align1" type="align" >
      <bind component="menu" interface="top"/>
      <bind component="grid" interface="top"/>
      <bind component="info" interface="top"/>
    </spatialConstraint>

    <spatialConstraint id="align2" xconnector="alignLeftBottomTop">
      <bind role="left1" component="menu" interface="left"/>
      <bind role="left2" component="videos" interface="left"/>
      <bind role="base" component="videos" interface="top"/>
      <bind role="base50" component="info" interface="bottom"/>
    </spatialConstraint>
  </body>
</layout>
```

Listing A.2: Template StyLe

```
<ncl>
  <head>
    <layoutBase>
      <importBase alias="layBase" documentURI="LayoutAppExample.xml"/>
    </layoutBase>
  </head>
  <body>
    <media id="menu1" src="menu1.png" layout="layBase#menu"/>
    <media id="menu2" src="menu2.png" layout="layBase#menu"/>
    <media id="menu3" src="menu3.png" layout="layBase#menu"/>
    <media id="menu4" src="menu4.png" layout="layBase#menu"/>

    <media id="image1" src="arrow.jpg" layout="layBase#grid"/>
    <media id="image2" src="flash.jpg" layout="layBase#grid"/>
    <media id="image3" src="gameofthrones.jpg" layout="layBase#grid"/>
    <media id="image4" src="vikings.jpg" layout="layBase#grid"/>
    <media id="image5" src="bettercallsoul.jpg" layout="layBase#grid"/>
    <media id="image6" src="bigbang.jpg" layout="layBase#grid"/>
    <media id="image7" src="twohalfman.jpg" layout="layBase#grid"/>
    <media id="image8" src="mentalist.jpg" layout="layBase#grid"/>
    <media id="image9" src="breakingbad.jpg" layout="layBase#grid"/>
    <media id="image10" src="goodwife.jpg" layout="layBase#grid"/>
    <media id="image11" src="houseofcards.jpg" layout="layBase#grid"/>
    <media id="image12" src="lietome.jpg" layout="layBase#grid"/>

    <media id="image13" src="info.jpg" layout="layBase#info"/>

    <media id="video1" src="video1.mp4" layout="layBase#videos" item="item1"/>
    <media id="vidoe2" src="video2.mp4" layout="layBase#videos" item="item1"/>
    <media id="video3" src="video3.mp4" layout="layBase#videos" item="item1"/>
```

```
31      <media id="vidoe4" src="video4.mp4" layout="layBase#videos" item="item1"/>
32    </body>
33  </ncl>
```

Listing A.3: Código do exemplo NCL que usa STyLe

APÊNDICE B – Schema XML da Linguagem STyLe

A listagem B.1 mostra o XML Schema da linguagem STyLe.

```xml
<!-- ======================================== -->
<!-- Layout                                   -->
<!-- ======================================== -->

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/Layout"
    xmlns:layout="http://www.midiacom.uff.br/gtvd/STyLe/Layout"
    xmlns:head="http://www.midiacom.uff.br/gtvd/STyLe/Head"
    xmlns:body="http://www.midiacom.uff.br/gtvd/STyLe/Body"
    elementFormDefault="unqualified">

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/Head"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLHead.xsd"/>

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/Body"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLBody.xsd"/>

    <complexType name="layoutPrototype">
      <sequence>
        <element ref="head:head" minOccurs="1" maxOccurs="1"/>
        <element ref="body:body" minOccurs="1" maxOccurs="1"/>
      </sequence>
    </complexType>
</schema>

<!-- ======================================== -->
<!-- Head                                     -->
<!-- ======================================== -->

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/Head"
    xmlns:head="http://www.midiacom.uff.br/gtvd/STyLe/Head"
    xmlns:transitionBase="http://www.ncl.org.br/NCL3.0/TransitionBase"
    elementFormDefault="unqualified">

    <import namespace="http://www.ncl.org.br/NCL3.0/TransitionBase"
        schemaLocation="http://www.ncl.org.br/NCL3.0/modules/NCL30TransitionBase.xsd"/>

    <complexType name="constraintConnectorBasePrototype">
      <attribute name="alias" type="string" use="required"/>
      <attribute name="URI" type="anyURI" use="required"/>
    </complexType>

    <complexType name="headPrototype">
      <sequence minOccurs="1" maxOccurs="1">
        <element ref="transitionBase:transitionBase"/>
        <element name="constraintConnectorBase" type="head:constraintConnectorBasePrototype"/>
      </sequence>
    </complexType>

    <element name="head" type="head:headPrototype"/>
</schema>
```

```
<!-- ========================================= -->
<!-- Body                                      -->
<!-- ========================================= -->

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/Body"
    xmlns:body="http://www.midiacom.uff.br/gtvd/STyLe/Body"
    xmlns:item="http://www.midiacom.uff.br/gtvd/STyLe/Item"
    xmlns:container="http://www.midiacom.uff.br/gtvd/STyLe/Container"
    xmlns:spatialConstraint="http://www.midiacom.uff.br/gtvd/STyLe/SpatialConstraint"
    elementFormDefault="unqualified">

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/SpatialConstraint"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLSpatialConstraint.xsd"/>

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/Item"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLItem.xsd"/>

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/Container"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLContainer.xsd"/>

    <complexType name="bodyPrototype">
      <choice minOccurs="0" maxOccurs="unbounded">
        <element ref="item:item" />
        <element ref="container:container" />
        <element ref="spatialConstraint:spatialConstraint" />
      </choice>
      <attribute name="id" type="string" use="optional"/>
    </complexType>

    <element name="body" type="body:bodyPrototype"/>
</schema>

<!-- ========================================== -->
<!-- Item                                       -->
<!-- ========================================== -->

<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/Item"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:item="http://www.midiacom.uff.br/gtvd/STyLe/Item"
        elementFormDefault="unqualified">


    <complexType name="itemType">
        <attribute name="id" type="ID" use="required"/>
        <attribute name="top" type="string" use="optional"/>
        <attribute name="left" type="string" use="optional"/>
        <attribute name="right" type="string" use="optional"/>
        <attribute name="bottom" type="string" use="optional"/>
        <attribute name="width" type="string" use="optional"/>
        <attribute name="height" type="string" use="optional"/>
        <attribute name="zIndex" type="integer" use="optional"/>
    </complexType>

    <element name="item" type="item:itemType"/>

</schema>

<!-- ============================================ -->
<!-- Container                                    -->
<!-- ============================================ -->

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/Container"
    xmlns:container="http://www.midiacom.uff.br/gtvd/STyLe/Container"
    xmlns:spatialConstraint="http://www.midiacom.uff.br/gtvd/STyLe/SpatialConstraint"
    xmlns:item="http://www.midiacom.uff.br/gtvd/STyLe/Item"
    elementFormDefault="unqualified">

```

```
    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/SpatialConstraint"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLSpatialConstraint.xsd"/>

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/Item"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLItem.xsd"/>

    <simpleType name="layoutType">
      <restriction base="string">
        <enumeration value="flowLayout"/>
        <enumeration value="gridLayout"/>
        <enumeration value="stackLayout"/>
        <enumeration value="carouselLayout"/>
      </restriction>
    </simpleType>

    <simpleType name="sortType">
      <restriction base="string">
        <enumeration value="size_increase"/>
        <enumeration value="size_decrease"/>
        <enumeration value="lexical_increase"/>
        <enumeration value="lexical_decrease"/>
      </restriction>
    </simpleType>

    <attributeGroup name="formatAttr">
      <attribute name="top" type="string" use="optional"/>
      <attribute name="left" type="string" use="optional"/>
      <attribute name="bottom" type="string" use="optional"/>
      <attribute name="right" type="string" use="optional"/>
      <attribute name="width" type="string" use="optional"/>
      <attribute name="height" type="string" use="optional"/>
      <attribute name="vspace" type="string" use="optional"/>
      <attribute name="hspace" type="string" use="optional"/>
      <attribute name="align" type="string" use="optional"/>
      <attribute name="zIndex" type="string" use="optional"/>
      <attribute name="columns" type="string" use="optional"/>
      <attribute name="rows" type="string" use="optional"/>
      <attribute name="orientation" type="string" use="optional"/>
      <attribute name="step" type="string" use="optional"/>
    </attributeGroup>

    <complexType name="formatType">
      <attributeGroup ref="container:formatAttr"></attributeGroup>
    </complexType>

    <complexType name="containerType">
      <choice minOccurs="0" maxOccurs="unbounded">
        <element ref="item:item"/>
        <element ref="spatialConstraint:spatialConstraint"/>
        <element name="format" type="container:formatType" minOccurs="0" maxOccurs="1"/>
        <element ref="container:container"/>
      </choice>
      <attribute name="id" type="ID" use="required"/>
      <attribute name="type" type="container:layoutType" use="optional"/>
      <attribute name="focusIndex" type="positiveInteger" use="optional"/>
      <attribute name="sortBy" type="container:sortType" use="optional"/>
    </complexType>

    <element name="container" type="container:containerType"/>
</schema>

<!-- ========================================= -->
<!-- SpatialConstraint                         -->
<!-- ========================================= -->

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/SpatialConstraint"
     xmlns:spatialConstraint="http://www.midiacom.uff.br/gtvd/STyLe/SpatialConstraint"
     elementFormDefault="unqualified">

    <simpleType name="protType">
      <restriction base="string">
```

```
        <enumeration value="align"/>
        <enumeration value="distribute"/>
        <enumeration value="size"/>
      </restriction>
    </simpleType>

    <simpleType name="protPriority">
      <restriction base="string">
        <enumeration value="core"/>
        <enumeration value="high"/>
        <enumeration value="medium"/>
        <enumeration value="low"/>
      </restriction>
    </simpleType>

    <simpleType name="protInterface">
      <restriction base="string">
        <enumeration value="top"/>
        <enumeration value="left"/>
        <enumeration value="right"/>
        <enumeration value="bottom"/>
        <enumeration value="middle"/>
        <enumeration value="center"/>
        <enumeration value="height"/>
        <enumeration value="width"/>
        <enumeration value="size"/>
      </restriction>
    </simpleType>

    <complexType name="bindParam">
      <attribute name="name" type="string"/>
      <attribute name="value" type="string"/>
    </complexType>

    <complexType name="bind">
      <sequence minOccurs="0" maxOccurs="unbounded">
        <element name="bindParam" type="spatialConstraint:bindParam"/>
      </sequence>
      <attribute name="component" type="string" use="required"/>
      <attribute name="interface" type="spatialConstraint:protInterface" use="required"/>
    </complexType>

    <complexType name="spatialConstraintType">
      <sequence minOccurs="2" maxOccurs="unbounded">
        <element name="bind" type="spatialConstraint:bind"/>
      </sequence>
      <attribute name="id" type="ID" use="required"/>
      <attribute name="direction" type="string" use="optional"/>
      <attribute name="offset" type="integer" use="optional"/>
      <attribute name="factor" type="decimal" use="optional"/>
      <attribute name="xconnector" type="string" use="optional"/>
      <attribute name="type" type="spatialConstraint:protType" use="optional"/>
      <attribute name="priority" type="spatialConstraint:protPriority" use="optional"/>
    </complexType>

    <element name="spatialConstraint" type="spatialConstraint:spatialConstraintType"/>

</schema>

<!-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = -->
<!-- ConstraintConnector                                                                      -->
<!-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = -->

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/ConstraintConnector"
  xmlns:constraintConnector="http://www.midiacom.uff.br/gtvd/STyLe/ConstraintConnector"
  xmlns:connectorAssessmentExpression="http://www.ncl.org.br/NCL3.0/ConnectorAssessmentExpression"
    xmlns:causalConnectorFunctionality="http://www.ncl.org.br/NCL3.0/CausalConnectorFunctionality"
  elementFormDefault="unqualified">

    <import namespace="http://www.ncl.org.br/NCL3.0/CausalConnectorFunctionality"
        schemaLocation="http://www.ncl.org.br/NCL3.0/modules/NCL30CausalConnectorFunctionality.xsd"/>
```

```

    <import namespace="http://www.ncl.org.br/NCL3.0/ConnectorAssessmentExpression"
        schemaLocation="http://www.ncl.org.br/NCL3.0/modules/NCL30ConnectorAssessmentExpression.xsd"/>

    <redefine schemaLocation="http://www.ncl.org.br/NCL3.0/modules/NCL30ConnectorCommonPart.xsd">
      <simpleType name="eventPrototype">
          <restriction base="eventPrototype">
            <enumeration value="attribution" />
          </restriction>
      </simpleType>
    </redefine>

    <redefine schemaLocation="http://www.ncl.org.br/NCL3.0/modules/NCL30ConnectorAssessmentExpression.
     xsd">
      <simpleType name="attributePrototype">
        <restriction base="attributePrototype">
          <enumeration value="nodeProperty" />
        </restriction>
      </simpleType>
    </redefine>

    <complexType name="constraintConnectorType">
      <choice minOccurs="0" maxOccurs="unbounded">
        <element ref="causalConnectorFunctionality:connectorParam"/>
        <element ref="connectorAssessmentExpression:assessmentStatement" />
        <element ref="connectorAssessmentExpression:compoundStatement" />
        <element ref="connectorAssessmentExpression:valueAssessment" />
        <element ref="connectorAssessmentExpression:attributeAssessment" />
      </choice>
      <attribute name="id" type="string" use="required"/>
      <attribute name="interface" type="string" use="required"/>
    </complexType>

    <element name="constraintConnector" type="constraintConnector:constraintConnectorType"/>
</schema>

<!-- ============================================ -->
<!-- STyLe Use                                    -->
<!-- ============================================ -->

<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/STLUse"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ncl="http://www.ncl.org.br/NCL3.0/EDTVProfile"
        xmlns:layoutuse="http://www.midiacom.uff.br/gtvd/STyLe/STLUse"
        elementFormDefault="unqualified">

    <import namespace="http://www.ncl.org.br/NCL3.0/EDTVProfile"
        schemaLocation="http://www.ncl.org.br/NCL3.0/profiles/NCL30EDTV.xsd"/>


    <!-- Layout Base -->
    <complexType name="layoutbaseType">
        <sequence minOccurs="0" maxOccurs="unbounded">
            <element ref="ncl:importBase"/>
        </sequence>
        <attribute name="id" type="ID" use="optional"/>
        <attribute name="name" type="string" use="optional"/>
    </complexType>

    <!-- declare global element in this module -->
    <element name="layoutBase" type="layoutuse:layoutbaseType"/>

    <!-- declare global attributes in this module -->
    <attributeGroup name="nodeAttribute">
        <attribute name="layout" type="string" use="optional"/>
    </attributeGroup>
</schema>
```

Listing B.1: Schema XML da Linguagem STyLe

APÊNDICE C – Schema XML da Extensão da Linguagem NCL para utilização da linguagem SᴛʏLe

A listagem C.1 mostra o XML Schema da extensão da linguagem NCL para utilizar a linguagem SᴛʏLe.

```
<!-- ================================================================= -->
<!-- NCL Extension                                                     -->
<!-- ================================================================= -->

<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.midiacom.uff.br/gtvd/STyLe/STLProfile"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:styleuse="http://www.midiacom.uff.br/gtvd/STyLe/STLUse"
        xmlns:profile="http://www.midiacom.uff.br/gtvd/STyLe/STLProfile"
        xmlns:ncl="http://www.ncl.org.br/NCL3.0/EDTVProfile"
        xmlns:nclstructure="http://www.ncl.org.br/NCL3.0/Structure"
        elementFormDefault="unqualified">

    <import namespace="http://www.midiacom.uff.br/gtvd/STyLe/STLUse"
        schemaLocation="http://www.midiacom.uff.br/gtvd/STyLe/modules/STLUse.xsd"/>
    <import namespace="http://www.ncl.org.br/NCL3.0/EDTVProfile"
        schemaLocation="http://www.ncl.org.br/NCL3.0/profiles/NCL30EDTV.xsd"/>
    <import namespace="http://www.ncl.org.br/NCL3.0/Structure"
        schemaLocation="http://www.ncl.org.br/NCL3.0/modules/NCL30Structure.xsd"/>


<!-- ================================================================= -->
<!-- Structure -->
<!-- ================================================================= -->
<!-- extends ncl element -->

    <element name="ncl" substitutionGroup="ncl:ncl">
        <complexType>
            <complexContent>
                <extension base="nclstructure:nclPrototype">
                    <sequence>

                    </sequence>
                </extension>
            </complexContent>
        </complexType>
    </element>

<!-- extends head element -->

    <complexType name="headType">
       <complexContent>
          <extension base="ncl:headType">
```

```
            <sequence>
               <element ref="layoutuse:layoutBase" minOccurs="0" maxOccurs="1"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <element name="head" type="profile:headType" substitutionGroup="ncl:head"/>

<!-- extends body element -->

   <complexType name="bodyType">
      <complexContent>
         <extension base="ncl:bodyType">
            <attributeGroup ref="layoutuse:contextAttribute"/>
         </extension>
      </complexContent>
   </complexType>

<element name="body" type="profile:bodyType" substitutionGroup="ncl:body"/>

<!-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = -->
<!-- Media -->
<!-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = -->
<!-- extends Media elements -->

   <complexType name="mediaType">
      <complexContent>
         <extension base="ncl:mediaType">
            <attributeGroup ref="layoutuse:nodeAttribute"/>
         </extension>
      </complexContent>
   </complexType>

<!-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = -->
<!-- Context -->
<!-- = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = -->
<!-- extends context element -->

<complexType name="contextType">
   <complexContent>
      <extension base="ncl:contextType">
         <attributeGroup ref="layoutuse:contextAttribute"/>
      </extension>
   </complexContent>
</complexType>

<element name="context" type="profile:contextType" substitutionGroup="ncl:context"/>
```

Listing C.1: Schema XML da Extensão da Linguagem NCL

APÊNDICE D – Schema XML da Linguagem XTemplate 4.0

A listagem D.1 mostra o XML Schema da linguagem XTemplate 4.0.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3  <schema targetNamespace="http://www.midiacom.uff.br/gtvd/XTemplate40/XTemplateESTyLe"
4          xmlns="http://www.w3.org/2001/XMLSchema"
5          xmlns:component="http://www.midiacom.uff.br/gtvd/XTemplate40/Component"
6          xmlns:connector="http://www.midiacom.uff.br/gtvd/XTemplate40/Connector"
7          xmlns:constraint="http://www.midiacom.uff.br/gtvd/XTemplate40/Constraint"
8          xmlns:impbase="http://www.midiacom.uff.br/gtvd/XTemplate40/ImportBase"
9          xmlns:iteration="http://www.midiacom.uff.br/gtvd/XTemplate40/Iteration"
10         xmlns:link="http://www.midiacom.uff.br/gtvd/XTemplate40/Link"
11         xmlns:media="http://www.midiacom.uff.br/gtvd/XTemplate40/Media"
12         xmlns:port="http://www.midiacom.uff.br/gtvd/XTemplate40/Port"
13         xmlns:structure="http://www.midiacom.uff.br/gtvd/XTemplate40/Structure"
14         xmlns:xtemplate="http://www.midiacom.uff.br/gtvd/XTemplate40/XTemplateENCL"
15         xmlns:layoutuse="http://www.midiacom.uff.br/gtvd/STyLe/STLUse"
16         elementFormDefault="unqualified">
17
18 <!-- import the definitions in the modules namespaces -->
19 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Component"
20         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPComponent.xsd"/>
21
22 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Connector"
23         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPConnector.xsd"/>
24
25 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Constraint"
26         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPConstraint.xsd"/>
27
28 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/ImportBase"
29         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPImportBase.xsd"/>
30
31 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Iteration"
32         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPIteration.xsd"/>
33
34 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Link"
35         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPLink.xsd"/>
36
37 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Media"
38         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPMedia.xsd"/>
39
40 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Port"
41         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPPort.xsd"/>
42
43 <import namespace="http://www.midiacom.uff.br/gtvd/XTemplate40/Structure"
44         schemaLocation="http://www.midiacom.uff.br/gtvd/XTemplate40/modules/XTPStructure.xsd"/>
45
46 <import namespace=""http://www.midiacom.uff.br/gtvd/STyLe/STLUse"
47         schemaLocation=""http://www.midiacom.uff.br/gtvd/STyLe/modules/STLUse.xsd"/>
48
49 <!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
50     Structure
51 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - ->
```

```
52 <!-- extends xtemplate element -->
53 <complexType name="xtemplateType">
54    <complexContent>
55       <extension base="structure:xtemplatePrototype">
56          <sequence>
57             <element name="head" type="xtemplate:headType" minOccurs="0" maxOccurs="1"/>
58             <element name="vocabulary" type="xtemplate:vocabularyType" minOccurs="1" maxOccurs="1"/>
59             <element name="body" type="xtemplate:bodyType" minOccurs="0" maxOccurs="1"/>
60             <element name="constraints" type="xtemplate:constraintsType" minOccurs="0" maxOccurs="1"/>
61          </sequence>
62       </extension>
63    </complexContent>
64 </complexType>
65
66 <element name="xtemplate" type="xtemplate:xtemplateType"/>
67
68
69 <!-- extends head element -->
70 <complexType name="headType">
71     <complexContent>
72         <extension base="structure:headPrototype">
73             <sequence>
74                 <element name="connectorBase" type="impbase:connectorbaseType" minOccurs="0" maxOccurs=
       "unbounded"/>
75                 <element name="descriptorBase" type="impbase:descriptorbaseType" minOccurs="0"
       maxOccurs="unbounded"/>
76         <element name="templateBase" type="impbase:templatebaseType" minOccurs="0" maxOccurs="unbounded
       "/>
77         <element name="layoutBase" type="impbase:layoutbaseType" minOccurs="0" maxOccurs="unbounded"/>
78             </sequence>
79         </extension>
80     </complexContent>
81 </complexType>
82
83
84 <!-- extends vocabulary element -->
85 <complexType name="vocabularyType">
86    <complexContent>
87       <extension base="structure:vocabularyPrototype">
88          <sequence>
89             <element name="component" type="component:componentType" minOccurs="0" maxOccurs="unbounded
       "/>
90             <element name="connector" type="connector:connectorType" minOccurs="0" maxOccurs="unbounded
       "/>
91          </sequence>
92       </extension>
93    </complexContent>
94 </complexType>
95
96
97 <!-- extends body element -->
98 <complexType name="bodyType">
99    <complexContent>
100      <extension base="structure:vocabularyPrototype">
101         <choice minOccurs="0" maxOccurs="unbounded">
102            <element name="port" type="port:portType"/>
103            <element name="media" type="media:mediaType"/>
104            <element name="link" type="link:linkType"/>
105            <element name="variable" type="iteration:VariableType"/>
106            <element name="for-each" type="iteration:forEachType"/>
107         </choice>
108      </extension>
109   </complexContent>
110 </complexType>
111
112
113 <!-- extends constraints element -->
114 <complexType name="constraintsType">
115    <complexContent>
116       <extension base="structure:constraintsPrototype">
117          <sequence>
118             <element name="constraint" type="constraint:constraintType" minOccurs="0" maxOccurs="
       unbounded"/>
```

```
          </sequence>
        </extension>
    </complexContent>
</complexType>

<!-- extends Component elements -->

<complexType name="componentType">
    <complexContent>
        <extension base="component:componentType">
            <attributeGroup ref="layoutuse:nodeAttribute"/>
        </extension>
    </complexContent>
</complexType>

<!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Template Base
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->

<element name="templates">
    <complexType>
        <sequence>
            <element ref="xtemplate:xtemplate" minOccurs="1" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="id" type="ID" use="optional"/>
        <attribute name="name" type="string" use="optional"/>
        <attribute name="description" type="string" use="optional"/>
    </complexType>
</element>

</schema>
```

Listing D.1: Schema XML da Linguagem XTemplate 4.0

APÊNDICE E – Formulário utilizado para realização do Teste de Usabilidade da Linguagem StyLe

De acordo com [50], o ponto principal da estrutura de dimensões cognitivas é considerar as notações ou interações das linguagens e o quão bem elas suportam as atividades pretendidas, dado o ambiente, meio e possíveis sub-dispositivos. Cada dimensão descreve um aspecto de uma estrutura de informação que é razoavelmente geral.

As dimensões cognitivas utilizadas nesse trabalho foram:

- Visibilidade - habilidade para visualizar facilmente os componentes da linguagem;
- Expressividade dos papéis - a utilidade de qualquer entidade é facilmente percebida;
- Proximidade do mapeamento - proximidade da representação com os elementos do domínio;
- Verbosidade - algumas notações podem ser extensas;
- Comprometimento prematuro - restringe uma ordem para realizar as tarefas;
- Dependências ocultas - vínculos importantes entre as entidades não são visíveis;
- Propensão a erros - a notação induz a erros e o sistema fornece pouca proteção;
- Consistência - semânticas similares são expressas com sintaxes similares;
- Viscosidade - resistência à mudança.

As Figuras E.1 e E.2 apresentam o formulário utilizado para o teste. Cada dimensão cognitiva é representada por uma pergunta. A relação entre perguntas e dimensões cognitivas são apresentadas logo a seguir.

- Visibilidade - Pergunta 1;
- Expressividade dos papéis - Pergunta 2;
- Proximidade do mapeamento - Pergunta 3;
- Verbosidade - Pergunta 4;
- Comprometimento prematuro - Pergunta 5;
- Dependências ocultas - Pergunta 6;
- Propensão a erros - Pergunta 7;
- Consistência - Pergunta 8;
- Viscosidade - Pergunta 9.

A questão 10 não representa uma dimensão cognitiva. A ideia é que os autores pudessem dar uma nota final para a linguagem.

Editar este formulário

Teste de Usabilidade da Linguagem STyLe - Dimensões Cognitivas

*Obrigatório

1. Você consegue visualizar claramente os componentes da sua aplicação? *

1 2 3 4 5 6 7 8 9 10

Visualizei pouco ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Visualizei muito

2. Você entendeu a função dos elementos da linguagem? *

1 2 3 4 5 6 7 8 9 10

Entendi pouco ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Entendi muito

3. O quão difícil foi implementar o teste com os elementos da linguagem? *

1 2 3 4 5 6 7 8 9 10

Muito Difícil ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Muito Fácil

4. Você considera a linguagem verbosa? *

1 2 3 4 5 6 7 8 9 10

Pouco verbosa ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Muito verbosa

5. Você considera que a linguagem restringe muito a ordem de criação dos elementos da sua aplicação? *

1 2 3 4 5 6 7 8 9 10

Restringe pouco ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Restringe muito

6. Você considera que a linguagem possui muitas dependências ocultas entre seus elementos? *

1 2 3 4 5 6 7 8 9 10

Figura E.1: Formulário usado para o teste de usabilidade da linguagem STyLe (página 1)

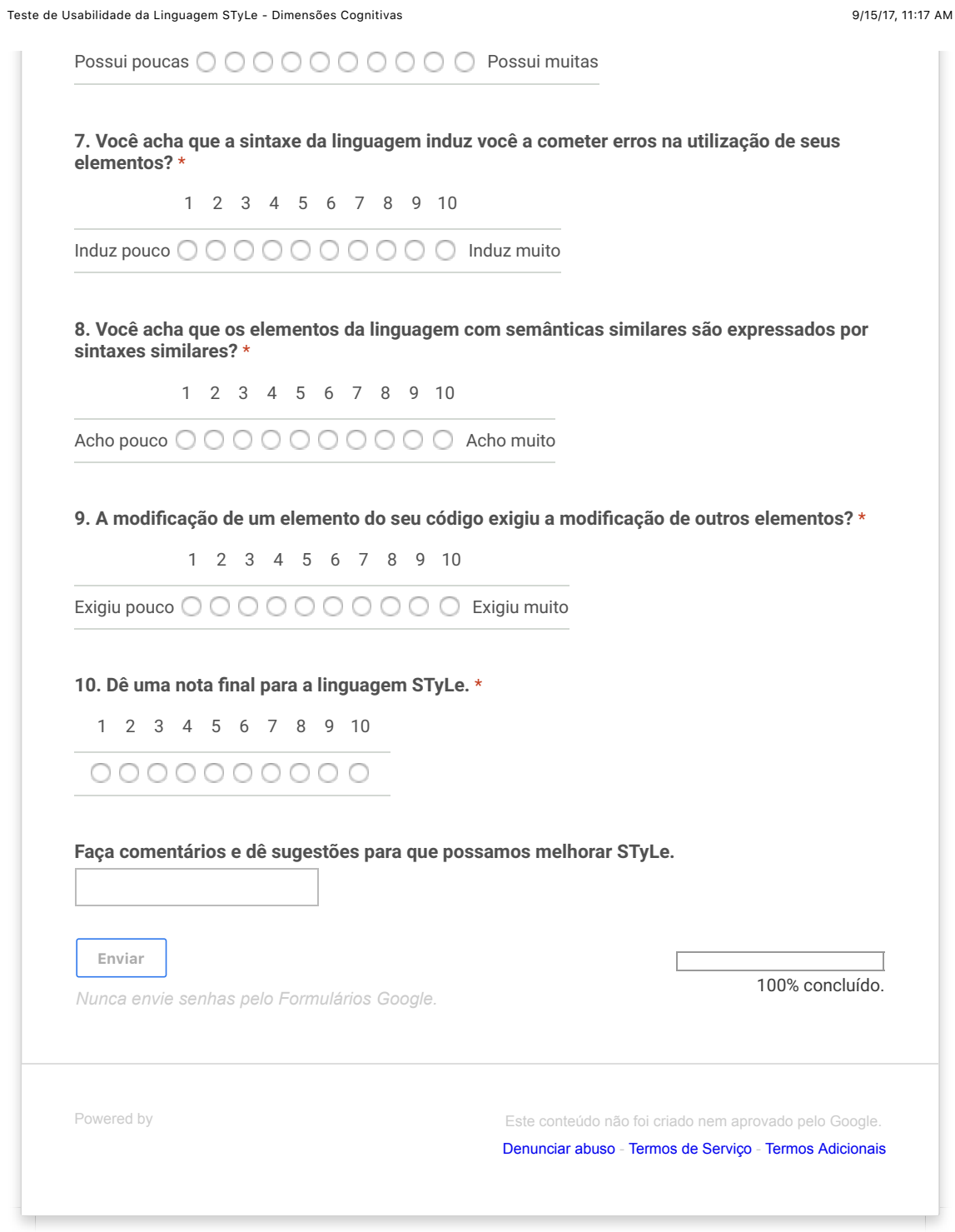

Figura E.2: Formulário usado para o teste de usabilidade da linguagem STyLe (página 2)