# Correção de Código Semi-Automática em Nested Context Language[1]

Rodrigo Costa Mesquita Santos[1], Thiago Alencar Gomes[1], Roberto Gerson Albuquerque Azevedo[2], Carlos de Salles Soares Neto[1,2], Mario Meireles Teixeira[1]

[1] Departamento de Informática – UFMA
Av. dos Portugueses, Campus do Bacanga
São Luís/MA – 65080-040 – Brasil

[2] Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225
Rio de Janeiro/RJ – 22453-900 – Brasil

{rodrim.c, thiago, roberto}@laws.deinf.ufma.br, {csalles, mario}@deinf.ufma.br

## RESUMO
O uso de ferramentas textuais para o desenvolvimento de aplicações hipermídia em NCL (*Nested Context Language*) aproxima o autor e o código fonte produzido. Por outro lado, essas ferramentas propiciam a criação de um ambiente mais sujeito a erros de programação, exigindo um autor de aplicações com um conhecimento moderado da linguagem e que seja capaz de identificar possíveis soluções para esses erros. Este trabalho apresenta técnicas de sugestão de correção semi-automática de erros, de forma a reduzir o tempo gasto na correção dos mesmos e, consequentemente, no desenvolvimento das aplicações, encorajando autores sem muita experiência a utilizarem ferramentas textuais. Também é discutida a implementação dessas técnicas na ferramenta NCL Eclipse.

## ABSTRACT
The use of textual authoring tools to NCL hypermedia application development brings the author closer to the source code produced. However, those tools tend to create an environment more subject to programming errors requiring an author with an intermediate knowledge of the language and capable of identifying possible solutions for those errors. This work presents techniques to suggest semi-automatic code correction in order to minimize the time spent in this task and consequently reduce application development time, thus encouraging less experienced authors to utilize textual authoring tools. Finally, the implementation of those techniques in the NCL Eclipse environment is also discussed.

## Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Hypertext/hypermedia, Markup Languages, *Standards, Corrections.*

## General Terms
Documentation, Design, Languages, Verification.

## Keywords
Nested Context Language, SBTVD, IPTV, Digital TV, Error Correction, NCL Eclipse.

## 1. INTRODUÇÃO
Este artigo apresenta um conjunto de técnicas voltadas para apoiar autores na tarefa de *correção de código* durante o desenvolvimento de aplicações em *Nested Context Language* (NCL), padrão do Sistema Brasileiro de TV Digital (SBTVD) [1] e recomendação internacional ITU-T para IPTV [2]. De forma a ilustrar o uso das técnicas fornecidas, uma implementação também é apresentada, a qual foi incorporada à ferramenta de autoria NCL Eclipse [3], *plug-in* textual desenvolvido para IDE Eclipse.

As técnicas propostas neste artigo visam dois objetivos principais: i) reduzir o tempo de desenvolvimento ao incorporar facilidades para a fase de depuração de código; ii) garantir maior qualidade final ao código gerado, ajudando o autor não só a identificar rapidamente erros mas também a corrigi-los de forma semi-automática.

A necessidade de criar aplicações livres de erros é um anseio desde os primórdios da ciência da computação [4]. Na prática, quanto mais cresce o tamanho do código fonte, cresce exponencialmente a expectativa pelo aparecimento de erros embutidos no código de forma não-proposital. Há um certo limiar em que uma aplicação cresce a tal ponto que a possibilidade de inexistência de erros, sem o emprego exaustivo de testes, é remota.

Apenas para citar duas grandes abordagens voltadas para a correção de erros, há, por um lado, o esforço centrado na *verificação formal* [5] de sistemas de software, onde a correção da intenção de algoritmos é feita com base em uma especificação formal ou em propriedades, usando métodos matemáticos formais. Por outro lado, há o crescente investimento em Ambientes Integrados de Desenvolvimento (IDEs) que guiem seu usuário não

[1] Semi-automatic Code Correction in Nested Context Language

apenas na tarefa de encontrar os erros mas também auxiliam na correção dos erros encontrados, sugerindo alternativas para corrigir seu código.

O emprego de verificação formal permite que, em certos casos, erros possam ser automaticamente corrigidos, de forma a atender determinadas propriedades. Uma vantagem dessa prática é a de assegurar formalmente a correção de seu código e assim garantir um alto grau de qualidade. Um exemplo do emprego dessa abordagem é [6], onde um sistema operacional foi desenvolvido com certas garantias de ausência de erros. Como desvantagem, o emprego de técnicas formais ainda requer um esforço de desenvolvimento maior, assim como uma maior especialidade da equipe de programadores.

O uso de IDEs, por outro lado, visa suavizar o esforço na fase de depuração de código tornando semi-automática a tarefa de corrigir erros. Em outros termos, a própria IDE detecta erros e aponta ao autor certas alternativas, onde uma delas deve ser escolhida para tornar seu código correto. Muita atenção tem sido dada a IDEs voltadas ao desenvolvimento de aplicações por meio de linguagens imperativas [7][8], mas pouco tem sido oferecido para linguagens declarativas. Linguagens declarativas enfatizam a descrição de um problema, em detrimento de sua implementação algorítmica, como é feito em linguagens imperativas. Dessa forma, elas são, em geral, mais legíveis por não programadores e, quando o propósito da linguagem é o mesmo (ou se assemelha) com o domínio de um problema, seu uso vai implicar em um programa mais simples de ser construído, com menos linhas de código [9].

Conforme foi previamente mencionado, este trabalho tem seu foco especificamente na identificação de técnicas que visam apoiar a correção de erros na autoria em NCL. NCL é uma linguagem declarativa para autoria de documentos hipermídia [10] e tem como foco a descrição do sincronismo entre as mídias através de relações de causalidade, onde uma sentença de causa faz disparar uma ação resultante. Ela é definida como uma linguagem de cola, fazendo uma relação espaço-temporal entre os objetos declarativos, imperativos e as mídias (vídeo, texto, áudio, imagem, etc) [11].

As técnicas propostas neste artigo visam explorar o próprio paradigma declarativo como apoio ao autor também na fase de correção de erros. Da mesma forma que as tarefas de autores usando linguagens declarativas são suavizadas ao fazer uso de metáforas de domínio, acreditamos que as técnicas propostas neste artigo também suavizam a *correção de código* ao aproximar essa tarefa do nível de abstração em que o código está escrito.

O artigo está organizado como segue. A Seção 2 apresenta alguns conceitos básicos. A Seção 3 fornece trabalhos relacionados à correção de erros. A Seção 4 discute as técnicas propostas especificamente em código NCL, descrevendo também a implementação proposta para o NCL Eclipse. A Seção 5 fornece conclusões e trabalhos futuros.

# 2. CONCEITOS BÁSICOS

Esta seção descreve os conceitos básicos utilizados no decorrer do artigo. A Seção 2.1 traz uma introdução à NCL, evidenciando os principais conceitos que serão alvos de técnicas de correção. A Seção 2.2 discute processos de validação e correção de código para NCL.

## 2.1 Linguagem NCL

NCL (*Nested Context Language*) é uma aplicação XML (*eXtensible Markup Language*) [12] que tem como base o modelo NCM (*Nested Context Model*) [13]. NCM é um modelo conceitual com foco na representação e manipulação de documentos hipermídia [14]. Sincronismo temporal e espacial, adaptação de conteúdo e exibição de mídias em múltiplos dispositivos de exibição são algumas funcionalidades que o NCM permite expressar.

Diferente de HTML ou XHTML, NCL define uma separação estrita entre o conteúdo das mídias e a estrutura do documento, o que possibilita um controle não invasivo da apresentação [15]. Dessa forma, NCL não define o conteúdo das mídias, mas como elas são estruturadas e relacionadas temporal e espacialmente, sendo, por isso, chamada de linguagem de cola.

Um documento NCL é dividido em dois elementos principais: o cabeçalho (*<head>*) e o corpo (*<body>*). No cabeçalho são definidas as bases de conectores (*<connectorBase>*), de regiões (*<regionBase>*), de descritores (*<descriptorBase>*), de regras (*<ruleBase>*), de transições (*<transitionBase>*), os documentos importados (*<importedDocumentBase>*) e informações de metadados das aplicações (*<meta>* e *<metadata>*). As bases agrupam e fornecem elementos que podem ser reutilizados no documento. O corpo do documento define os nós de mídia, os nós de composição e os relacionamentos entre os nós.

Os nós de mídia são os elementos da linguagem que representam os objetos de mídia em si (*<media>*). Além disso, também podem-se definir âncoras nesses objetos de mídia. As âncoras podem ser de dois tipos: de conteúdo, que representam uma porção do conteúdo da mídia; ou de atributos, que definem alguma característica dessa mídia, tais como dimensão, transparência etc. Em especial, a âncora de conteúdo que define todo o conteúdo da mídia não necessita ser definida e é interpretada como o próprio objeto de mídia.

Os nós de composição agrupam nós de mídia e outros nós de composição, recursivamente, e podem ser de dois tipos: contexto (*<context>*) ou alternativa (*<switch>*). Os nós de contexto permitem agrupar nós que possuem algum tipo de relacionamento entre si, o que permite estruturar e encapsular os elementos NCL. Os nós de alternativa dão suporte à adaptação de conteúdo, agrupando nós cuja execução depende da avaliação de regras definidas na base de regras. Para estabelecer interfaces para acesso a nós internos a uma composição, é necessária a definição de portas nesses nós (*<port>*). Nós de composição também atuam no encapsulamento, pois ajudam a definir a perspectiva de um elemento, que é a hierarquia de contextos dentro da qual esse elemento está definido e, dessa forma, seu escopo válido no documento.

NCL faz a separação das relações (conectores) e dos relacionamentos (elos). As relações utilizadas no documento devem ser definidas na base de conectores. Uma relação define um conjunto finito de papéis que podem ser exercidos pelas âncoras de nós. O relacionamento entre essas âncoras é definido por meio de um elo que utiliza uma das relações definida na base de conectores e deve explicitar qual âncora exerce qual papel na relação.

Visando aumentar o reuso em NCL, a forma como os objetos de mídia devem ser inicialmente executados também pode ser definida em elementos de primeira ordem: os descritores (*<descriptor>*). Da mesma forma, a posição da tela e a identificação da classe de dispositivo onde esses objetos serão apresentados inicialmente também podem ser definidos como elementos de primeira ordem, que são as regiões (*<region>*).

Como esses elementos são definidos separadamente, frequentemente, tem-se a necessidade de referenciá-los. Isso é feito por meio do identificador dos elementos, que deve ser único no documento. São alguns exemplos: uma mídia faz referência a um descritor, por meio do atributo *descriptor*; um elo faz referência a um conector pelo seu atributo *xconnector*, definindo que ele é um relacionamento daquele tipo; os descritores referenciam regiões através do atributo *region*. Percebe-se que NCL utiliza de forma exaustiva esses apontamentos entre elementos. Isso se justifica principalmente pelo aumento do reuso, visto que tais elementos são definidos uma única vez e podem, então, ser referenciados em outras partes do documento. Note que, com relação à correção de erros, o fato de os elementos estarem espacialmente separados (no código) com relação a onde eles são utilizados pode induzir a erros na autoria. Esse não é um problema específico de NCL, mas do mecanismo de referência presente em qualquer linguagem XML.

## 2.2 Detecção de Erros em Código NCL

Há várias razões pelas quais se faz necessário algum processo de validação durante a concepção e execução de documentos hipermídia. É fácil perceber que há um ganho significativo de tempo no processo de autoria quando uma ferramenta oferece ao autor um retorno descritivo sobre erros no documento antes mesmo de executá-lo [3]. No ambiente de execução, por exemplo, essa detecção de erros é necessária para evitar que a execução do documento resulte em apresentações inconsistentes. Identificados os erros, é possível tanto que a apresentação seja abortada, como também que a aplicação seja executada em partes, ignorando os trechos de código com erros.

Os erros de programação em linguagens imperativas são geralmente divididos em erros sintáticos, semânticos e lógicos [16]. Erros sintáticos estão relacionados a erros na grafia dos comandos ou pontuação. Erros semânticos referem-se ao significado do código. Erros lógicos, por sua vez, estão relacionados com a intenção do autor e não a particularidades da linguagem utilizada. A identificação dos erros sintáticos e semânticos se faz presente na maioria dos ambientes de autoria, mas muitas vezes não é possível identificar erros lógicos, já que inferir a lógica, ou intenção, do autor a partir dos comandos não é uma tarefa trivial. Em linguagens declarativas, por outro lado, os erros lógicos e semânticos não possuem uma barreira clara de separação, já que o que se expressa na linguagem é o próprio objetivo final, ou seja, a lógica do programa. Assim sendo, em ambientes declarativos um erro semântico é, em geral, equivalente a um erro lógico.

Em linguagens declarativas baseadas em XML, a utilização do XML Schema [12] constitui-se como uma boa abordagem inicial para validação do documento, uma vez que existem diversas bibliotecas disponíveis para esse fim. Porém, NCL possui várias particularidades que não podem ser descritas através de um XML Schema, tais como escopo baseado em perspectiva e relações semânticas de referência. O NCL Validator [17] é uma proposta de validação sintática e semântica específica para documentos NCL e que será utilizada como base para as técnicas de correção de código propostas. O NCL Validator possui uma implementação em Java que pode ser utilizada em sua versão *stand-alone* ou como uma biblioteca a ser utilizada por ferramentas de autoria (o NCL Eclipse e o Composer [14] são exemplos de ferramentas que a utilizam). O NCL Validator divide o processo de validação em quatro etapas principais: léxica e sintática; estrutural; contextual e de referências; e semântica.

A etapa léxica e sintática consiste na leitura do documento XML e geração da árvore sintática que será utilizada pelas próximas etapas de validação. Essa é a etapa mais genérica, uma vez que não são analisadas nenhumas das particularidades da linguagem NCL, e sim, se o documento está seguindo as regras de XML.

A etapa estrutural valida os elementos e atributos da linguagem, verifica a presença dos atributos obrigatórios, a cardinalidade dos elementos e os filhos desses elementos. Até essa etapa a validação é bastante similar a uma validação convencional por XML Schema e segue a norma ABNT de NCL [1].

A etapa de validação contextual e de referências é responsável por validar se os atributos que apontam para outros elementos do documento são consistentes. Ser consistente, nesse caso, significa apontar para um tipo de elemento válido para aquele atributo (por exemplo, um atributo *component* de um *<bind>* deve apontar para um elemento *<context>*, *<media>* ou *<switch>*) e ainda o elemento apontado deve estar em um contexto acessível a partir de quem o aponta (na mesma perspectiva ou acessível por interfaces).

A etapa de validação semântica é responsável por identificar informações acerca da apresentação do documento, tais como trechos do documento inalcançáveis por elos. Embora essa validação seja chamada de semântica, observa-se claramente um problema de nomenclatura, já que as duas etapas anteriores também identificam erros semânticos, segundo a definição apresentada anteriormente.

Atualmente, o NCL Validator é capaz de identificar erros sintáticos e semânticos em documentos NCL, inclusive com mensagens de erros que auxiliam a sua correção. Entretanto, não existe nenhum mecanismo sistemático de correção de erros, seja de forma automática ou semi-automática. Este artigo tem como um dos focos de sua implementação estender o NCL Validator e propor melhorias de forma que seja possível a correção de código semi-automática. A correção semi-automática, em um ambiente de autoria é muito bem-vinda já que pode ser utilizada para auxiliar o autor, sugerindo correções que podem ser validadas (se estão logicamente corretas, ou seja, segundo o que o autor espera) antes de serem aplicadas.

# 3. TRABALHOS RELACIONADOS

Há um conjunto extenso de ferramentas de desenvolvimento de aplicações que utilizam técnicas de sugestão de correção semi-automática de erros para linguagens imperativas, contrastando com o cenário declarativo, onde não observamos a mesma variedade de ferramentas. Nesta seção apresentaremos algumas dessas ferramentas e as técnicas empregadas por elas.

Expresso[18] é uma ferramenta desenvolvida com fins educacionais que busca apoiar programadores iniciantes na linguagem Java. Tem o seguinte padrão de funcionamento: o autor da aplicação submete o código fonte à ferramenta e ela retorna mensagens explicitando os erros e alertas encontrados e as sugestões de possíveis correções para erros sintáticos, semânticos e lógicos. O mecanismo de correção dessa ferramenta se baseia em uma lista composta dos erros mais comuns na linguagem, que foram obtidos através de um estudo prévio realizado com programadores, onde foram observados seus padrões de programação e os erros cometidos no processo de autoria. Dessa forma, há um conjunto de sugestões previamente estabelecidas para os erros que o programa é capaz de identificar. Por não se tratar de um ambiente de autoria, e nem de um compilador, mas sim de uma ferramenta de validação de código fonte, o Expresso não oferece nenhum tipo de suporte à aplicação de correções no código, diferindo da abordagem aqui proposta que visa modificar diretamente o código fonte, aplicando as sugestões escolhidas pelo autor.

O JDT (Java Development Tool) [19] é um *plug-in* para a IDE Eclipse que oferece um conjunto de ferramentas para programadores da linguagem Java. Algumas das funcionalidades dessa ferramenta são a coloração das palavras reservadas e a sugestão de conteúdo (*autocomplete*) de métodos. Essa ferramenta também é capaz de reconhecer erros sintáticos e semânticos no código, oferece uma lista de sugestão para alguns dos erros encontrados e, além disso, realiza a correção escolhida pelo usuário para os erros. Nossa proposta tem como base o funcionamento do JDT, uma vez que ambos anseiam os mesmos objetivos que são oferecer sugestões de correção ao autor de aplicação e realizar as devidas alterações no código fonte de acordo com a escolha do autor, caracterizando uma abordagem semi-automática de correção. Um segundo fator que leva a tomar essa ferramenta como base é o fato dela ser implementada como um *plug-in* para o Eclipse, da mesma forma que o NCL Eclipse, que reutiliza o NCL Validator. A diferença fundamental entre essas duas propostas é que o JDT é direcionado à linguagem imperativa Java, com todas as restrições relacionadas.

Visual Studio[8] é uma ferramenta proprietária que contém um conjunto de editores para diversas linguagens. O editor da linguagem Visual Basic contém um mecanismo de correção de erros capaz de sugerir correções para erros comuns como, por exemplo, conversões inválidas entre tipos de dados e parênteses desbalanceados. Da mesma forma que o JDT, possui funcionamento semelhante à presente proposta, porém é voltada à linguagem imperativa Visual Basic.

JECA (Java Error Correcting Algorithm) [20] é um algoritmo desenvolvido para correção de erros sintáticos em pequenos programas Java. Esse algoritmo implementa uma proposta de correção de código baseado na tentativa de predizer a intenção do usuário a cada erro encontrado. Como auxílio a esse mecanismo de correção, é mantida uma estrutura de dados que armazena algumas das variações errôneas de cada uma das palavras-chave da linguagem, utilizando o algoritmo distância de edição [22] para guardar informações capazes de expressar o quão diferente essa variação é da palavra-chave correspondente. O processo de correção segue duas etapas: a etapa 1 envolve examinar os identificadores do programa, comparando-os com as palavras-chave da linguagem; no passo 2, para cada erro encontrado, o programa oferece no máximo uma sugestão de correção e questiona ao autor se tal alteração deve ou não ser realizada no código, realizando-a caso autorizado. Uma das deficiências da implementação desse algoritmo é a aceitação apenas de pequenos trechos de código (no máximo 50 linhas). Esse algoritmo difere deste trabalho primeiramente pelo paradigma da linguagem foco. Em segundo lugar, por fazer apenas uma única sugestão por erro, sendo que alguns erros são passíveis de serem corrigidos de formas distintas. Por último, JECA se limita a correção de erros sintáticos.

Mecanismos de correção de erros em linguagens declarativas são pouco explorados por ferramentas de autoria, o que não implica na ausência de técnicas para este fim. O IKME (Intelligent Knowledge Management Environment) [21] é um projeto da Universidade de Kansas cujo foco é o desenvolvimento de um ambiente capaz de validar documentos XML em geral, identificar os erros sintáticos através do uso do *XML Schema* da linguagem em questão, oferecer sugestões de correção e realizar as correções escolhidas pelo usuário. Seu funcionamento é composto de três fases: na fase 1 é construída uma árvore sintática do documento; na fase 2 é feita a interpretação do XML Schema da linguagem em questão (que deve ser explicitamente carregado para a ferramenta) e a validação do documento segundo esse XML Schema; na fase 3 são realizadas as mudanças que o autor escolher. O fato do IKME utilizar o XML Schema como base para realizar a validação e as sugestões de correções impossibilitam a utilização desse ambiente para oferecer sugestões de correção semi-automática em NCL. Isso porque, como discutido na Seção 2, NCL possui especificidades que não podem ser validadas somente por meio do XML Schema.

# 4. CORREÇÃO DE CÓDIGO SEMI-AUTOMÁTICA EM NCL

Esta seção apresenta a proposta de sugestão de correção de código semi-automática para NCL. A Seção 4.1 descreve as etapas envolvidas no processo de sugestão e aplicação de correção de código. A Seção 4.2 discute esse processo aplicado à linguagem NCL. E a Seção 4.3 descreve os detalhes de implementação e integração ao NCL Eclipse.

## 4.1 As Etapas da Correção de Código

O processo de correção de código fonte semi-automático tem sido preferencialmente usado, em vez de um processo de correção automática, pois diminui consideravelmente a possibilidade de inserção de novos erros pelo próprio processo de correção. A correção automática de erros semânticos, por mais que seja possível, está sujeita a inserir novos erros lógicos (ou seja, resultando em uma aplicação com propósito diferente daquele que o autor inicialmente desejava). Isso é particularmente verdade quando existe mais de uma possibilidade de correção de erro, o que pode resultar em códigos fonte "certos", mas com significados diferentes. Para linguagens imperativas, é possível, embora pouco provável, que mesmo diferentes ações de correção para um mesmo erro semântico resultem em uma mesma lógica. No caso específico de linguagens declarativas, contudo, isso não se aplica, já que a múltipla possibilidade de correção de erros semânticos já está relacionado a múltiplas lógicas da aplicação resultantes.

O processo de correção de código de forma semi-automática pode ser dividido em seis passos gerais (Figura 1): (i) identificação dos erros presentes no código fonte, por meio de um compilador (para linguagens imperativas) ou de um validador (para linguagens declarativas); (ii) computação e (iii) sugestão das possíveis correções (observe que, dependendo do erro, é possível corrigi-lo com mais do que uma ação de correção); (iv) escolha de uma das correções propostas, pelo usuário; (v) execução da ação de correção escolhida; (vi) correção de qualquer outra parte do código afetada pela aplicação da correção. A Figura 1 apresenta esse processo de forma esquemática, representando cada uma dessas etapas e as interdependências entre elas.

O usuário geralmente é o responsável por disparar o processo de correção, por meio da mensagem (1) da Figura 1, seja explicitamente, ou por meio de alguma mudança no código fonte. A etapa (i), validação do código fonte, tem então início, resultando em mensagens de erro ou de alertas sobre aquele código fonte. As mensagens de erros, sejam elas informadas como cadeias de caracteres ou estruturas de dados mais elaboradas (que podem incluir outras informações, como o elemento relacionado àquele erro, a posição desse elemento etc.) são então enviadas, através da mensagem (2) – para que seja possível determinar as correções. A identificação unívoca das mensagens de erros se faz necessária para que não ocorram ambiguidades no processo de determinar essas correções.

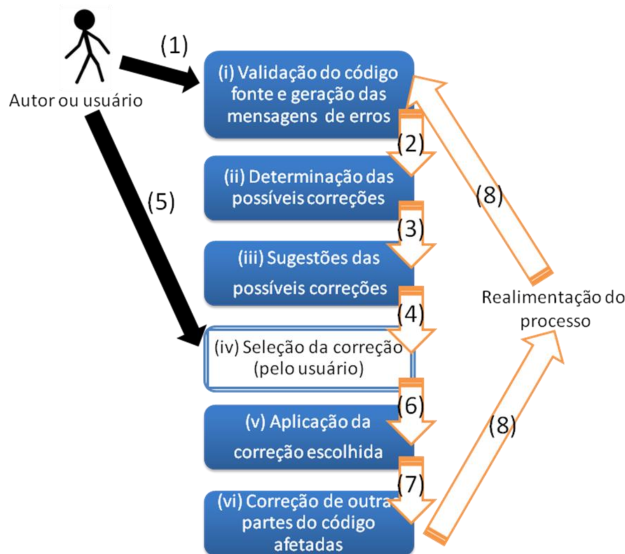


**Figura 1: Processo de correção de código semi-automático.**

A etapa (ii), de determinar as correções de erros possíveis, está intimamente relacionada com a sugestão de conteúdo contextual (*autocomplete*), ou seja, de predizer uma frase ou texto que o usuário deseja inserir. É muito provável que o usuário tenha feito algo próximo daquilo que ele realmente intenta e tenha cometido algum erro simples, como por exemplo, erro de digitação . Em linguagens baseadas em XML, que frequentemente utilizam apontamentos para identificadores de elementos, um erro comum de se cometer é digitar erroneamente o nome de um elemento declarado anteriormente. Contudo, o próprio nome digitado já pode dar uma pista de qual a solução para aquele erro.

A Figura 2 apresenta um exemplo simples e didático. Em NCL, por exemplo, o atributo *region* de um elemento <descriptor> deve ter o valor do atributo *id* de um elemento <region>. No exemplo da figura, o usuário digitou “RGtexto”, sendo que os valores possíveis são “rgTela”, “rgVideo”, “rgImage” e “rgTexto1”. É fácil perceber, que, dentre esses, muito provavelmente, o usuário pretendia digitar “rgTexto1” e não os outros valores. Outras correções possíveis seriam: adicionar uma nova região com identificador “RGtexto”; e ainda remover o atributo *region* do elemento <descriptor>. Fica claro, assim, que determinar os possíveis valores de um atributo e filtrá-lo de forma correta (evitando, no exemplo, três sugestões de correção pouco ou sequer prováveis: “rgTela”, “rgVideo” e “rgImage” ) também fazem parte da etapa (ii).

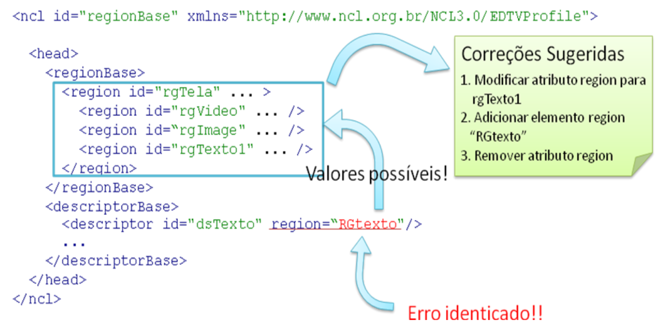


**Figura 2: Possíveis correções para um código NCL.**

A etapa (ii) resulta em uma lista de ações de correção que podem ser aplicadas ao código, reunidas na mensagem (3). A etapa (iii) recebe essa lista de ações e a transforma em uma lista de mensagens que podem ser entendidas pelo usuário, e são então apresentadas pela interface, deixando-o apto a escolher uma delas. O conjunto dessas mensagens amigáveis ao usuário estão reunidas na mensagem (4).

A escolha de uma dessas ações, etapa (iv), é realizada pela mensagem (5) vinda do usuário. Essa ação então é enviada pela mensagem (6), resultando na sua execução, pela etapa (v), e consequente modificação do código fonte. É possível, entretanto, que tal mudança não seja atômica, necessitando assim da etapa (vi), que ajusta o código à ação de correção aplicada. Um exemplo simples da necessidade dessa etapa (vi) é a formatação do código após a aplicação da correção.

Estender o processo da Figura 1 para um processo de correção automática equivale a remover a mensagem (5) e utilizar alguma heurística para inferir a correção de erro a ser utilizada. A abordagem semi-automática foi utilizada neste artigo para evitar que novos erros fossem inseridos pelo próprio processo, o que culmina em confundir ainda mais o autor. Essa abordagem semi-automática também permite que o autor tenha total controle e seja informado sobre todas as alterações que porventura ocorram no texto. Manter o autor ciente de todas as implicações sobre o código de suas decisões é uma premissa importante.

O processo de sugestão semi-automática, até a etapa (iii), pode ser realizado dinamicamente em tempo de autoria, ou seja, enquanto o usuário interage e modifica o código fonte, refletindo as mudanças mais recentes. Em um processo automático, uma vez que ele é disparado, não é possível interrompê-lo em um momento específico, resultando sempre que necessário, em modificações no

código fonte. Modificar o código fonte enquanto o usuário o edita costuma ser um inconveniente ainda maior, o que impossibilita o uso da abordagem automática em tempo de autoria, ao mesmo tempo em que o autor interage com o código.

## 4.2 Correção de código em NCL

É possível especializar o processo de correção de código fonte acima discutido para linguagens baseadas em XML. Em especial, nesta seção, discutir-se-á como realizar essa especialização para a linguagem NCL. Esta seção tem como foco as etapas (ii), (iii), (v) e (vi), que são as inovações propostas nesse artigo. As etapas (i) e (iv) serão discutidas na Seção 4.3, pois estão mais relacionadas com a implementação.Os erros semânticos em NCL podem ser divididos em quatro categorias principais:

(a) **Elementos com atributos obrigatórios ausentes ou inválidos**.

(b) **Elementos com filhos obrigatórios ausentes ou inválidos.**

(c) **Elementos que possuem inconsistências entre seus atributos.** Alguns atributos em NCL devem levar em consideração a presença ou o valor de outros. Por exemplo, um atributo *refer* não deve coexistir com o *src* em um elemento *<media>*.

(d) **Erros de referência.** Diferente de outras linguagens XML, em NCL as perspectivas devem ser consideradas.

Especializar o processo discutido na Seção anterior para NCL consiste em, além de criar mecanismos para a identificação dos erros, agrupando-os de forma coerente, também definir um conjunto de transformações (ou ações) que podem ser aplicadas no texto com o objetivo de corrigi-lo. Tais transformações devem ser tidas como atômicas, no sentido de corrigirem um erro específico em um único passo. As transformações não devem se limitar a alterações de texto simples, mas também algumas abstratas, que tratem erros específicos no nível de abstração da linguagem. As seguintes primitivas são propostas como transformações possíveis em um documento: *addString, removeString, addStartTag, removeStartTag, addEndTag, removeEndTag, addElement, removeElement, removeAttribute, setAttribute.* Alguns comentários devem ser feitos a respeito da primitiva *addEndTag*. Quando ela for chamada, deve-se fazer uma busca em todos os filhos do elemento que se deseja corrigir, em busca de um elemento inválido, ou seja, que não pode ser filho daquele elemento segundo a definição formal da linguagem NCL, para só então ser inserida a *tag* de fechamento. No caso de elementos que não podem ter filhos, o fechamento deve ocorrer na mesma linha de abertura.

A Tabela 1 apresenta uma lista de erros comuns e como resolvê-los. Seu objetivo principal é demonstrar alguns dos tipos de erros mais comuns em NCL e possíveis soluções. As informações contidas na Tabela 1 são um indício de que, com base em um pequeno conjunto de transformações, é possível sistematizar o processo de correção de código em NCL.

Sugerir correções também passa pelo processo de sugerir valores possíveis, ou seja, sugestão de conteúdo contextual. Particularmente em NCL, essa sugestão de conteúdo também deve ser baseada na perspectiva onde o usuário está editando naquele instante. Além de descobrir os valores possíveis que o usuário pode inserir em uma determinada posição, o processo de filtrar esses valores para torná-lo mais próximo do que o usuário realmente intenta pode ser realizado através de algoritmos relativamente comuns. Neste trabalho propomos o uso do algoritmo de distância de edição [22] para que, baseado no que o usuário digitou, seja possível sugerir correções mais direcionadas.

**Tabela 1: Exemplos de erros na autoria NCL e sugestões de soluções**

| Natureza | Erro | Sugestões de Soluções | Casos de Erro |
|---|---|---|---|
| Sintática | Elemento sem *tag* de fechamento. | 1. Sugerir a *tag* de fechamento; (*addEndTag*)<br>2. Remover *tag* de abertura; (*removeStartTag*) | Elemento <body> sem </body> correspondente. |
| Sintática | Elemento não pertencente à linguagem. | 1. Remover elemento; (*removeElement*) | Elemento <x/> como filho de <body>. |
| Semântica | Erros de referência | 1. Modificar referência para um elemento que exista. (*setAttribute*)<br>2. Adicionar elemento com esse id; (*addElement*)<br>3. Remover elemento que faz essa referencia; (*removeElement*) | O atributo *region* de um elemento *<descriptor>* referencia o id de uma região que não existe. |
| Semântica | Elementos com atributos obrigatórios ausentes | 1. Adicionar o atributo; (*setAtribute*) | Um elemento *<region>* sem o atributo *id*. |
| Semântica | Elementos com filhos obrigatórios ausentes | 1. Adicionar elemento; (*addElement*) | Um elo sem o elemento filho <bind>. |
| Semântica | Elementos que possuem inconsistências entre seus atributos | 1. Remover o primeiro atributo; (*removeAttribute*)<br>2. Modificar o valor do atributo, tornando-o consistente; (*setAttribute*) | Um elemento <media> com atributo src e refer. |

## 4.3 Implementação no NCL Eclipse

Visando validar a presente proposta, foi realizada uma implementação, acoplada ao NCL Eclipse. A Figura 3 mostra quais os principais componentes da arquitetura do NCL Eclipse, onde o componente "*NCL Error Correction*", adicionado por este trabalho, aparece em destaque. Esse componente é responsável por implementar o processo de correção de código semi-automático. Para isso, necessita comunicar-se com: o "*NCL Validator*", para receber as notificações de mensagens de erro; o "*NCL Content Proposer*", para, receber a sugestão de conteúdo, e, juntamente com as mensagens de erro, sugerir as possíveis ações de correção; a "*Problem View*", onde, além de apresentar os erros existentes no documento, é possível selecionar a correção desse erro; e o núcleo do NCL Eclipse ("*NCL Eclipse Core*"), para onde as solicitações de alteração no documento são enviadas.

O *NCL Validator* necessitou de pequenas adaptações, principalmente no que se refere ao formato da mensagem de erro retornada. Novas informações foram adicionadas à classe *Message* retornada por ele, visando facilitar o processo de correção. Todas as mensagens de erros do NCL Validator já são previamente identificadas de forma unívoca por um valor numérico (MSG_ID) e uma melhor classificação das mensagens, segundo a proposta deste trabalho, também foi necessária.

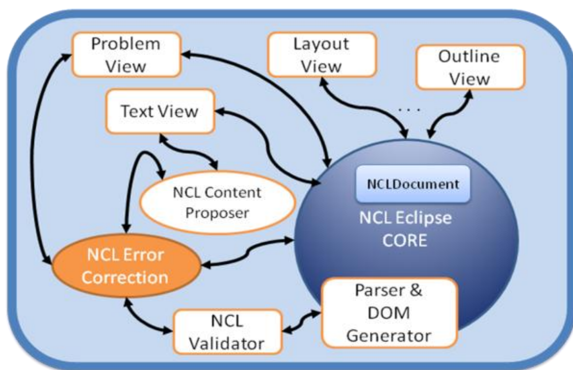


**Figura 3: Arquitetura destacando o NCL Error Correction entre os componentes do NCL Eclipse.**

Para cada uma das categorias de erros definidas acima foi implementada uma ou mais ações de correção, que agrupam um conjunto das transformações. Essas ações são parametrizadas, de tal forma que seja possível definir exatamente onde o erro ocorreu, qual a ação a ser aplicada e onde (em qual posição do texto) esta ação deve acontecer. A Figura 4demonstra o NCL Eclipse com o processo de correção de código NCL em execução.

## 5. CONCLUSÃO

O uso de mecanismos capazes de fornecer ao programador de aplicações correções de código, seja de forma automática ou semi-automática, é comumente utilizado em ambientes de autoria textual para linguagens imperativas, podendo atingir o nível de correção de erros sintático, semânticos ou mesmo lógicos. Porém, o uso desse mecanismo em ambiente de autoria de documentos declarativos hipermídia é pouco explorado. Este trabalho apresenta uma proposta de implementação desse mecanismo em ferramentas de autoria de documentos declarativos NCL, tomando como base a ferramenta de validação NCL Validator e estendendo-a de forma a propiciar a sugestão de correção semi-automática de erros.

A abordagem proposta visa suprir necessidades comuns aos diversos perfis de programadores, desde os iniciantes aos mais experientes, de forma que a atenção do autor de aplicações se focalize na construção da estrutura do documento e o tempo e esforço gasto na correção de erros se tornem mínimo. Na implementação dessa proposta, os erros mais comuns da linguagem foram listados e categorizados segundo suas sugestões de correção, tornando o processo de correção em si mais viável de ser implementado por ferramentas que reutilizem o validador.

No futuro, uma pesquisa com programadores de diferentes perfis será conduzida para verificar, de forma qualitativa, a eficácia dessa abordagem num ambiente de autoria, avaliando o quanto, de fato, o tempo gasto na autoria diminui. A pesquisa visa, também, avaliar se programadores com pouco conhecimento de NCL se sentiram mais confortáveis utilizando ambientes textuais que ofereçam o suporte ao desenvolvimento proposto por este artigo. Como exemplo específico, pretendemos avaliar a eficácia da aplicação do algoritmo de distância de edição como base para a oferta de sugestões na correção de código.

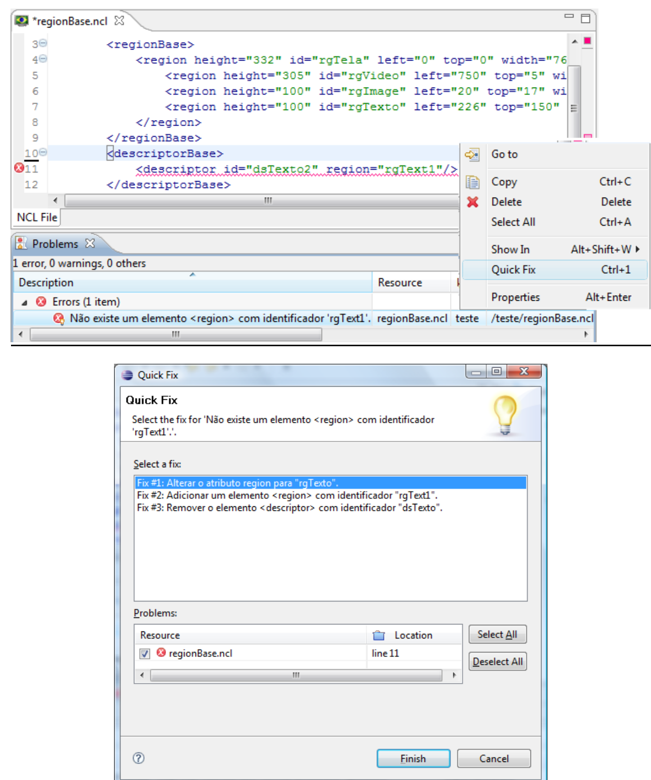


**Figura 4: NCL Eclipse com correção de código semi-automática em execução.**

## 6. REFERÊNCIAS


[1] ABNT - Associação Brasileira de Normas Técnicas (2007) "Televisão Digital Terrestre- Codificação de dados e especificações de transmissão para radiodifusão digital. Parte 2: Ginga -NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações" .

[2] ITU - International Telecommunication Union (2009) "Nested context language (NCL) and Ginga-NCL for IPTV services".

[3] Azevedo, R. G. A. NCL Eclipse: editor textual para desenvolvimento de programas Hipermídia Interativos em NCL. 2008.

[4] Bentley, J. L. 1982 *Writing Efficient Programs*. Prentice-Hall, Inc.

[5] Vardi, M. Y., and Wilke, T. "Automata - from logic to algorithms," Logic and Automata - History and Perspectives, 2007.

[6] Elkaduwe, D., Klein, G. and Elphinstone, K. Verified Protection Model of the seL4 Microkernel. Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments, Toronto, Canada. October 06-09, 2008.

[7] Eclipse SDK. Disponível em: http://www.eclipse.org.

[8] Visual Studio. Disponível em: http://www.microsoft.com/visualstudio.

[9] Soares, L. F. G. and Barbosa, S. D. J. Programando em NCL: Desenvolvimento de aplicações para o middleware Ginga, TV digital e Web. Rio de Janeiro: Elsevier, 2009.

[10] Soares, L. F. G., and Rodrigues, R. F. Nested Context Language 3.0 Part 8 – NCL Digital TV Profiles. Technical Report. Departamento de Informática da PUC-Rio, MCC 35/06. Disponível em *http://www.ncl.org.br/NCL3.0-DTV.pdf*

[11] Soares, L. G., Moreno, M. F., and Sant'Anna, F. 2009. Relating declarative hypermedia objects and imperative objects through the NCL glue language. In *Proceedings of the 9th ACM Symposium on Document Engineering* (Munich, Germany, September 16 - 18, 2009).

[12] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium, 10 February 1998, revised 6 October 2000. This version of the XML 1.0 Recommendation is http://www.w3.org/TR/2000/REC-xml-20001006. The latest version of XML 1.0 is available at http://www.w3.org/TR/REC-xml.

[13] Casanova, M. A., Tucherman, L., Lima, M. J. D., Netto, J. L. R., Rodriguez, N., and Soares, L.F.G. 1991. The nested context model for hyperdocuments. In *Proceedings of the third annual ACM conference on Hypertext*. ( San Antonio, Texas, United States, 1991) .

[14] Guimarães, R. L., Costa, R. M. R., and Soares, L. F. G. 2007. Composer: Ambiente de Autoria de Aplicações Declarativas para TV Digital. XIII Simpósio Brasileiro de Sistemas Multimídia e Web WebMedia2007. Gramado, Brasil - Outubro de 2007.

[15] Soares, L. F. G. e Souza Filho, G. L. Interactive Television in Brazil: System Software and the Digital Divide. 2007. European Interactive TV Conference - EuroITV2007 Amsterdam, The Netherlands - Maio de 2007.

[16] Youngs, E.A. Human errors in programming. Int. J. Man-Machine Studies 6 (1974).

[17] NCL Validator. Disponível em: http://www.laws.deinf.ufma.br/nclvalidator.

[18] Hristova, M., Misra, A., Rutter and M., Mercuri, R. Identifying and correcting Java programming errors for introductory computer science students, Proceedings of the 34th SIGCSE technical symposium on Computer science education, Fevereiro, 2003, Reno, Navada, USA.

[19] Eclipse Java Development Tools. Disponível em: http://www.eclipse.org/jdt.

[20] Sykes, E. R., and Franek, F. Presenting JECA: A Java Error Correcting Algorithm for the Java Intelligent Tutoring System. Proceedings of the IASTED Conference on Advances in Computer science and Technology. St. Thomas, US Virgin Islands. 2004.

[21] Shivadas, A. Intelligent Correction and Validation Tool for XML. Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science. 2001.

[22] William J. Masek, Michael S. Paterson, A faster algorithm computing string edit distances, Journal of Computer and System Sciences, Volume 20, Issue 1, February 1980, Pages 18-31.