Método de Validação Estrutural e Contextual de Documentos NCL

José Rios Cerqueira Neto, Rodrigo Costa Mesquita Santos, Carlos de Salles Soares Neto, Mário Meireles Teixeira

Laboratory of Advanced Web Systems - LAWS

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

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

ABSTRACT

This paper proposes a structural and contextual validation method for NCL (Nested Context Language) documents. As part of the proposed method, we define a declarative metalanguage to ensure low coupling between NCL structure and the code of the validator, which easy the validation of new language profiles. Requirements such as incremental processing and multilingual messages are also covered by this work. An implementation of this method using component architecture is presented as a concept proof.

RESUMO

Este artigo propõe um método de validação estrutural e contextual de documentos NCL (*Nested Context Language*). Como parte do método proposto, é definida uma metalinguagem declarativa visando assegurar baixo acoplamento entre a estrutura de NCL e o código do validador, o que facilita a validação de novos perfis da linguagem. Requisitos como validação incremental e mensagens multilíngües também são contemplados por este trabalho. Uma implementação deste método utilizando uma arquitetura de componentes é apresentada como prova de conceito.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *Incremental compilers*.

General Terms

Design, Languages, Verification.

Keywords

Nested Context Language, NCL, Incremental Compilers, Source Code Validation.

1. INTRODUÇÃO

Em qualquer ambiente integrado de desenvolvimento (IDE – *Integrated Development Environment*) atual, há a necessidade pela validação do código fonte instantaneamente, o que é indispensável para facilitar o processo de desenvolvimento de software. O programador precisa de uma resposta rápida de validação de seu código fonte para confirmar sua correção sintática e posteriormente checar a correção semântica com testes. Nos primórdios da computação, esse requisito não era atendido. Nesse caso, o programador tipicamente entregava seu código fonte em um centro de processamento de dados (CPD) para obter o resultado da compilação daquele código apenas algumas horas ou até dias depois. A mera falta de um mecanismo instantâneo de compilação e validação do código fonte, como se vê, conduzia a uma mudança completa na forma em que se desenvolvia *software*. Apenas ilustrando uma diferença importante, o código fonte precisava ser checado várias vezes antes de ser submetido ao CPD, o que exigia bem mais tempo para o desenvolvimento.

A validação sintática de código em linguagens de programação imperativas é um problema bem resolvido há anos. Linguagens imperativas são aquelas em que o código fonte é descrito algoritmicamente, como uma sequência passo a passo de comandos. Por outro lado, há também as linguagens declarativas, que descrevem a intenção final do código e não como fazê-lo, geralmente em maior nível de abstração. Linguagens declarativas normalmente não focam em uma descrição algorítmica e são mais descritivas.

Ferramentas de autoria para linguagens declarativas são menos numerosas quando comparadas a IDEs imperativas. Dentre essas IDEs, observa-se que uma considerável parcela não emprega mecanismos eficazes de validação de código. Quando se trata de aplicações XML (*eXtensible Markup Language*) [26], alguns desses mecanismos são reduzidos a uma verificação da instância do código escrito junto ao XML Schema [25] ou DTD [6] da linguagem. Uma possível explicação para a falta de pluralidade de ferramentas de validação para linguagens declarativas pode ser o fato de que, geralmente, ferramentas de autoria para essas linguagens tendem a fazer uso de abstrações gráficas em detrimento da autoria textual. Pressupõe-se que essas abstrações gráficas produzam documentos sintaticamente corretos. Outro problema recorrente é a falta de maior expressividade nas mensagens resultantes do processo de validação. Tipicamente a mudança central ao se projetar mecanismos de correção de código em linguagens declarativas é que o nível em que se quer dar as mensagens de validação também deve ser alto, para ser equivalente ao nível mais descritivo da linguagem. Mensagens de erro em ambientes declarativos precisam conduzir bem mais rapidamente ao entendimento do erro encontrado e, assim como em linguagens imperativas, possivelmente sugerir uma alternativa para a correção desse erro. A importância de mensagens de erro é

WebMedia'11: Proceedings of the 17th Brazilian Symposium on Multimedia and the Web. Full Papers.
October 3 -6, 2011, Florianópolis, SC, Brazil.
ISSN 2175-9642.
SBC - Brazilian Computer Society

identificada em diversos trabalhos [21] mas a sistematização ou uma metodologia para esse mecanismo ainda é um problema em aberto.

Há, também, outras diferenças na validação de código quando do emprego da abordagem declarativa. Vários estudos alinham a necessidade de validação sintática com a de correção semântica do código, o que ocorre normalmente pelo emprego de técnicas formais [17][18]. Em ambientes de autoria declarativos, essa mesma necessidade de validação se faz presente, pela mesma motivação de guiar o programador mais facilmente e rapidamente a um código semanticamente correto. Uma diferença importante é que o código declarativo é criado para ser executado por uma máquina abstrata de mais alto nível. Uma sentença declarativa pode equivaler a vários comandos imperativos. Como o código declarativo já descreve a intenção final, a checagem formal da semântica desse código tende a ser mais simples de ser aplicada.

Um exemplo particular de linguagem declarativa é NCL (*Nested Context Language*). NCL é uma linguagem XML modular que permite a especificação de documentos multimídia como um conjunto de nós (objetos de mídia ou de composição) e elos, que relacionam eventos de sincronismo espaço-temporais entre esses nós. NCL é a linguagem padrão do ISDB-Tb [1] para aplicações interativas de TV digital e ITU-T para IPTV [11].

Diversos requisitos não-funcionais são identificados por este trabalho como necessários de serem atendidos por uma proposta de validação em NCL. Como NCL foi adotada em diversos países da América Latina e África como padrão para a criação de aplicações interativas em TV aberta, um primeiro requisito, simples de ser atendido, é a necessidade das mensagens de erro do mecanismo de validação serem multilíngue. Adicionalmente, a validação de código precisa ser facilmente integrada a ferramentas de autoria, possibilitando seu reúso em diversos ambientes. Logo elucidamos nosso segundo requisito: suporte ao reúso e independência de tecnologia.

O fato de NCL ser uma linguagem modular permite que os módulos definidos sejam agrupados em perfis. A título de exemplo, citam-se dois perfis de NCL definidos no contexto de TV digital: EDTV (*Enhanced Digital TV*) e BDTV (*Basic Digital TV*). Um terceiro requisito, então, é o suporte a adequação do processo de validação a diferentes perfis da linguagem. Receptores de TV digital possuem recursos computacionais escassos, tanto de processamento quanto de memória. Logo, um *middleware* de TV digital, que executará em um desses receptores, não dispõe de ampla capacidade computacional para a execução das aplicações interativas. Uma abordagem que pode ser interessante nessas condições seria que o próprio *middleware* realizasse uma validação no código da aplicação interativa recebida, evitando a possibilidade de executar aplicações corrompidas que só iriam desperdiçar recursos. Porém, o próprio processo de validação necessita ser eficiente para que ele não seja o gargalo, consumindo mais recursos do que a própria tentativa de execução de uma aplicação defeituosa. Esse cenário apenas reforça um quarto requisito: eficiência.

Por fim, uma característica desejável do processo de validação em ferramentas de autoria é que ele possa ser parcial, ou seja, seria interessante que a validação pudesse acontecer em trechos específicos do código fonte (apenas no cabeçalho ou apenas no corpo do documento, por exemplo). O atendimento desse requisito pode dar subsídio para que tal validação parcial possa vir a ser, também, incremental, no sentido que só executa sobre trechos de código que tiverem sofrido alterações ou tiverem sido influenciados por essas mudanças desde a última validação efetuada.

Este trabalho apresenta uma proposta detalhada de validação sintática e contextual para documentos NCL. Por validação contextual, entende-se a verificação de escopo na linguagem, o qual é definido pelas perspectivas dos diversos nós de composição (contextos e nós de alternativa), conforme é mais bem detalhado adiante. O propósito desta proposta é facilitar a concepção de novas ferramentas de autoria para NCL com suporte abrangente para a validação de código. O método proposto nas seções subjacentes busca atender, além dos requisitos funcionais de uma ferramenta de validação, aos requisitos não funcionais citados nos parágrafos anteriores. Espera-se, também, que se estabeleça embasamento para que o próprio método seja estendido para a validação de outras linguagens de programação declarativas.

Para alcançar o objetivo de descrever tal proposta de validação, este artigo subdivide-se da seguinte forma. Na Seção 2 são levantados e analisados diversos trabalhos relacionados. A Seção 3 detalha o método proposto e a Seção 4 apresenta uma implementação de tal método. As considerações finais e as conclusões deste trabalho são feitas na Seção 5.

2. TRABALHOS RELACIONADOS

Ao se fazer uma análise dos diversos ambientes de autoria existentes para o desenvolvimento de aplicações, observa-se que grande parte deles possui algum mecanismo embutido de validação de código fonte, sendo que esses mecanismos variam desde os mais simples, que apenas fazem verificação léxica das palavras chave da linguagem [15] até os que implementam mecanismos mais complexos, como validação incremental [2] ou sugestão de correção de código [8] [19].

Dentro do escopo das linguagens imperativas, nota-se que as diferentes ferramentas de autoria implementam diversos modelos de validação de código. O Lua Eclipse [15] é um *plug-in* para a IDE Eclipse [9] voltado ao desenvolvimento de *scripts* Lua. Este ambiente disponibiliza poucos recursos de validação, limitando-se quase que somente a verificação de palavras chave da linguagem. Outras ferramentas, como o CDT [7] – *plug-in* de desenvolvimento C/C++ para o Eclipse – e o Visual Studio – ferramenta proprietária para o desenvolvimento dedicado à plataforma .NET – vão além dessa simples verificação e abordam também outros aspectos sintáticos e semânticos do código. O JDT [8] – também um *plug-in* para o Eclipse que oferece um ambiente de desenvolvimento para a linguagem Java – chega a implementar um mecanismo de sugestão e aplicação de correções no código fonte.

Na validação de código para linguagens declarativas derivadas de XML, o mais trivial é se pensar em uma validação utilizando XML Schema [25], uma vez que este especifica regras de validação para os elementos e os atributos das linguagens. Há diversas ferramentas que implementam essa modalidade de validação. O IKME (*Intelligent Knowledge Management Environment*) [20] é uma ferramenta de validação estática de código que identifica erros sintáticos baseado exclusivamente no XML Schema da linguagem. O XML Screamer [12] é outra ferramenta que utiliza o XML Schema para realizar a validação de documentos XML e possui uma arquitetura que busca oferecer otimizações neste processo. Tais otimizações são possíveis graças à síntese do processo de *parsing* do documento e da geração de

eventos SAX, pulando algumas etapas inerentes à maioria dos processos de validação que utilizam o SAX. Há também bibliotecas de validação que se utilizam do XML Schema, como por exemplo a biblioteca padrão Java para o tratamento de XML e a API (*Application Programming Interface*) Xerces [24]. Apesar de ser uma linguagem derivada de XML, a validação de código NCL utilizando essa abordagem não é suficiente, uma vez que NCL possui algumas especificidades que o XML Schema não contempla, como, por exemplo, o escopo baseado em perspectivas. Validar um código NCL utilizando o XML Schema abrange apenas a sintaxe e a estrutura de NCL, desprezando importantes aspectos referenciais e contextuais.

Há outros processos de validação que não se baseiam no XML Schema. Um exemplo disso é o SMIL Builder [2], um ambiente de autoria gráfico para a linguagem SMIL [22]. Assim como NCL, SMIL é uma linguagem hipermídia focada na definição de relações de sincronismos entre as mídias que compõem a aplicação. Este ambiente utiliza como representação interna do documento SMIL uma estrutura de dados chamada H-SMIL-Net [3] (sendo esta uma extensão temporal da estrutura Petri Net [18]), que oferece um modelo estruturado de representação de dados e permite uma validação incremental deste modelo. Assim, a alteração em uma parte da aplicação não leva à validação de todo o modelo, apenas da parte afetada pela modificação (validação incremental). O uso dessa estrutura de dados se justifica pela necessidade em SMIL de checar a definição de sincronismos temporais inconsistentes [29]. A H-SMIL-Net facilita a percepção de inconsistências temporais, além de dar suporte a validação incremental.

A natureza visual do SMIL Builder reduz substancialmente o trabalho de validação do documento. Como todas as interações entre o autor da aplicação e o documento são intermediadas por abstrações visuais, a ferramenta de validação pode pressupor que o código fonte gerado pela ferramenta de autoria será sempre sintaticamente correto, limitando o processo de validação à verificação semântica dos sincronismos temporais definidos pelo autor. Apesar de parecer um trabalho reduzido, o processo de verificação desses sincronismos não é trivial. O paradigma de causalidade em que se baseia a NCL não permite o desenvolvimento de relações temporais inconsistentes, porém um modelo de validação incremental equivalente ao utilizado pelo SMIL Builder pode ser obtido utilizando o método de validação proposto por este trabalho.

O SMIL Author [28] é mais um ambiente de autoria que emprega técnicas de validação que não se baseiam no XML Schema e é capaz de identificar inconsistências temporais nos documentos. Utiliza como modelo interno de dados o RTSM (*Real-Time Synchronization Model*) [28], porém, diferentemente do SMIL Builder e do trabalho descrito neste artigo, não emprega técnicas de validação incremental ou parcial.

As duas principais ferramentas de autoria para NCL são o NCL Eclipse [19] e o Composer [14]. Ambas utilizam como componente de validação o NCL Validator [16]. O NCL Validator é um processo de validação de documentos NCL dividido em três etapas: i) validação léxica; ii) validação estrutural; e iii) validação de contextos e referências. A etapa (i) verifica aspectos inerentes à escrita de elementos e atributos em XML. A etapa (ii) valida aspectos estruturais, como, por exemplo, a presença ou ausência dos atributos ou filhos obrigatórios. Por último a etapa (iii) valida se as referências feitas no documento estão de acordo com o mecanismo de escopo baseado em perspectivas definidos para a NCL. Há, também, o suporte, nessa ferramenta, a mensagens de erros em diferentes idiomas.

Apesar de realizar um processo de validação eficaz de documentos NCL, alguns requisitos atendidos pela presente proposta não são atendidos pelo NCL Validator. Por não fazer uma clara separação entre a estrutura da linguagem e o código fonte da aplicação, as particularidades da NCL não são generalizadas, levando a validação de cada elemento da linguagem como um caso particular. Na prática, para oferecer a checagem semântica, há uma classe para cada elemento existente na linguagem. O escopo baseado em perspectivas é, talvez, o exemplo mais claro de uma das particularidades que não estão generalizadas no código fonte. Essa não-separação da estrutura da linguagem acarreta um segundo inconveniente: para a validação de diferentes perfis da linguagem há a necessidade de alteração do código fonte e realizar uma nova compilação da ferramenta.

O NCL Validator utiliza como estrutura de modelagem interna dos dados uma árvore DOM [5]. Isso implica que uma ferramenta de autoria utilizando o NCL Validator necessita fazer a conversão da instância do código NCL em questão para a respectiva árvore DOM que o represente a cada vez que o código for verificado, o que acarreta um custo computacional a mais no processo. Uma possível melhoria neste processo seria a ferramenta de autoria também trabalhar com uma árvore DOM para manter a estrutura do documento. Essa abordagem traz, porém, alguns inconvenientes. Em primeiro lugar, isso força a ferramenta de autoria a utilizar a mesma modelagem de dados que o NCL Validator utiliza. O ideal seria que a ferramenta de validação se ajustasse às ferramentas de autoria e não o contrário. Um segundo inconveniente desta abordagem é o custo em se manter uma árvore DOM na memória sempre atualizada. Esse problema é particularmente difícil de resolver quando lidamos com a autoria textual, caso em que construir uma árvore DOM é impraticável enquanto o documento não estiver bem formado, inviabilizando a validação.

Como terceira característica indesejável, o NCL Validator não emprega – ou mesmo possibilita – nenhum tipo de técnica que permita uma validação parcial ou incremental no documento NCL. Dessa forma, percebe-se que, mesmo tendo como foco a mesma linguagem alvo, o NCL Validator e o método de validação descrito neste artigo possuem diferenças substanciais no processo de validação. Tais diferenças são melhor detalhadas na Seção 3.

Por fim, cita-se também o NCL Inspector [10], sendo esta uma ferramenta de crítica de código NCL que possibilita ao autor de aplicações definir novas regras a serem validadas. Essas regras podem ser definidas utilizando a linguagem de programação Java ou utilizando documentos XSLT [27]. Essa ferramenta, no entanto, não realiza a validação do código NCL, apenas valida as regras definidas. Este trabalho não implementa mecanismos de definição de novas regras, ficando esta funcionalidade como trabalho futuro. Mais detalhes sobre a possibilidade de extensão da validação com novas regras são dados na Seção 5.

3. VALIDAÇÃO NCL

NCL é baseada em um modelo conceitual de dados próprio, o qual é chamado NCM (*Nested Context Model*) [13]. NCL herda do NCM o paradigma de causalidade e restrição no qual se baseia a especificação espaço-temporal dos relacionamentos definidos entre as mídias que compõem uma aplicação. Para a TV digital

interativa, os perfis usados de NCL dão suporte apenas ao paradigma de causalidade, onde determinada ação é executada quando certa condição é satisfeita.

Buscando prover melhor estruturação e encapsulamento às aplicações, o modelo NCM introduz a noção de nós de composição. Nós de composição podem ser de dois tipos: contexto e alternativa. Nós de contexto agrupam nós que possuem alguma relação semântica entre si, bem como seus respectivos relacionamentos. O próprio corpo do documento é um nó de contexto. Já os nós de alternativa são os responsáveis pela adaptação de conteúdo de uma aplicação.

Os nós de composição, além de melhor estruturar uma aplicação, definem a perspectiva dos elementos internos a ele. Dessa forma, os elos contidos em uma composição apenas são capazes de referenciar elementos internos a mesma composição. A noção de perspectiva também pode ser entendida como a hierarquia aninhada de contextos na qual um elemento se encontra. Devido a essa noção, diz-se que NCL possui um escopo baseado em perspectivas. Adicionalmente, a verificação de código NCL precisa ser contextual, ou seja, levar em conta a perspectiva dos contextos.

Entre outras características de NCL, essa noção de perspectiva é um dos casos que inviabiliza a validação de documentos NCL baseada exclusivamente no XML Schema. Na verdade, o XML Schema seria suficiente apenas para validação sintática e estrutural dos documentos, uma vez que existem diversas particularidades em NCL que não podem ser expressas por ele. Como exemplo disso, ao checar a correção de um elo, é necessário assegurar se os nós que estão sendo associados são identificadores válidos e presentes na mesma composição (o que não pode ser verificado por uma especificação XML Schema).

Outra característica recorrentemente empregada no projeto de NCL são as múltiplas referências possíveis de serem feitas entre os elementos da linguagem. O projeto de NCL foi pensado para permitir alto grau de reúso [23]. Um nó de mídia, por exemplo, faz referência a um descritor, que por sua vez faz referência a uma região. Observa-se que tanto o descritor quanto a região (ou ambos) podem estar em arquivos separados, dificultando a validação. Um tipo especial de referência é a possibilitada pelo atributo *refer*. Se o nó A referencia o nó B por meio do atributo *refer*, então A vai herdar todas as características de B e eles necessariamente devem ser do mesmo tipo. Uma restrição a essa referência é que ela não pode acontecer se B for um dos nós da hierarquia de contextos de A ou se B também possuir o atributo *refer* referenciando outro nó C.

Uma terceira característica de NCL que dificulta a validação é o fato de que o valor de alguns atributos pode depender do valor de outro (como os atributos *type* e *subtype* do elemento *transition*). Em alguns casos, a própria cardinalidade dos elementos pode depender do elemento do qual ele é filho (como ocorrre nos elementos *compoundCondition* ou *compoundAction*, dependendo de seu elemento pai) ou do valor de algum atributo (como na cardinalidade dos elementos *bind* de um *link*).

As peculiaridades supracitadas tendem a aumentar a complexidade de sistematização do processo de validação, podendo levar à implementação de um validador que analisa cada elemento da linguagem individualmente, tornando-o pouco genérico. Um inconveniente de um validador que não generaliza seu método de validação é a dificuldade na manutenção do mesmo, podendo tornar inviável a adequação da validação a outros perfis ou versões da linguagem.

Este trabalho tem por objetivo principal elaborar um método que generalize o tratamento de parte das peculiaridades de NCL, permitindo o desenvolvimento de um processo de validação que lide com um número reduzido de casos particulares. Além de generalizar parte do processo, também busca-se dar suporte à validação incremental de documentos, o que tipicamente é um recurso desejável por ferramentas de autoria.

3.1 Metalinguagem de Validação NCL

Uma maneira de tornar um validador ciente da estrutura de NCL é embutir seus aspectos estruturais diretamente no código fonte, fazendo uso de alguma estrutura de dados abstrata. Essa abordagem, apesar de ser bem direta, apresenta o inconveniente de acoplar fortemente o código fonte à estrutura de NCL. Isso dificulta, por exemplo, modificar a validação para diferentes perfis (ou mesmo versões) de NCL.

Uma alternativa a esse forte acoplamento é descrever a linguagem a ser validada por meio de uma metalinguagem. Nesse caso, a validação deve interpretar uma especificação escrita nessa metalinguagem dinamicamente e executar de acordo com essa definição. Seguindo essa abordagem, fazer com que o validador consiga atuar em diferentes perfis de NCL equivale apenas a descrever esses perfis em função da metalinguagem, sem alterar o código fonte do validador. Outra importante vantagem dessa abordagem é o fato de que a generalização do processo de validação torna-se diretamente proporcional ao poder de expressão da metalinguagem em descrever particularidades de NCL. Em outras palavras, o uso de uma metalinguagem pode diminuir consideravelmente a quantidade de casos particulares que o validador deve tratar, minimizando o esforço de implementação e a margem para erros de codificação.

Este trabalho define uma metalinguagem para a descrição de NCL que é capaz de contemplar seu escopo baseado em perspectiva, uma das maiores limitações do emprego de XML Schema para validação NCL. Além dessa particularidade, diversos outros aspectos estruturais da NCL também são descritos por essa metalinguagem. O restante desta subseção tem a finalidade de descrevê-la, ressaltando como diversos aspectos de NCL podem ser descritos por ela.

A metalinguagem declarativa proposta é baseada em primitivas que descrevem certos aspectos da linguagem. Foram definidas quatro primitivas: ELEMENT, ATTRIBUTE, REFERENCE e DATATYPE. Cada uma dessas primitivas possui um conjunto de argumentos e informa aspectos específicos da NCL para o validador realizar a checagem de código. A Listagem 1 apresenta o esqueleto dessas primitivas, junto com seus argumentos.

```
ELEMENT   (name, parent, cardinality,
           define_scope)

ATTRIBUTE (element_name, name, is_required,
           datatype_id)

DATATYPE  (datatype_id, regex)

REFERENCE (element_name, attr_name,
           ref_element_name,
           ref_attr_name, perspective)
```

Listagem 1. Esqueleto das primitivas da metalinguagem

A primitiva ELEMENT define os elementos XML de NCL. Essa primitiva possui quatro argumentos: o nome do elemento, o nome do elemento pai, sua cardinalidade (a quantidade de ocorrências para esse elemento pai) e um valor booleano indicando se esse elemento define ou não um escopo.

Em NCL, um mesmo elemento pode ser filho de diferentes elementos pai. Esse é o caso, por exemplo, do elemento *<media>*, que pode ser filho de *<context>*, *<body>* ou *<switch>*. Outro interessante caso particular acontece com o elemento *<compoundCondition>*. Esse elemento pode ser tanto filho do elemento *<causalConnector>* quanto dele mesmo. O elemento *<causalConnector>* pode possuir 0 ou 1 elemento *<compoundCondition>*, porém é permitido um *<compoundCondition>* ter tantos filhos *<compoundCondition>* quanto se queira. Essa particularidade demonstra que para um mesmo elemento, podem-se ter diferentes cardinalidades para diferentes elementos pai. Dessa forma, de acordo com a metalinguagem proposta, é necessária a declaração de uma primitiva ELEMENT para cada diferente par de elemento pai e elemento filho.

A cardinalidade dos elementos pode ser um número inteiro ou um dos seguintes caracteres especiais: *, +, ?, #. Também existe o operador *A* seguido de índices numéricos (*A*1, *A*2...). Se a cardinalidade de um elemento for um número, esse valor indicará o número exato de vezes que o elemento em questão deve aparecer. O * indica que o elemento pode aparecer zero ou mais vezes, já o + indica que o elemento deve aparecer pelo menos uma vez e o ? indica que o elemento deve aparecer zero ou uma vez.

O símbolo # tem o seguinte significado: digamos que o elemento X tenha os elementos Y e Z como filhos com cardinalidades #, então, pelo menos uma vez Y ou Z devem aparecer como filho de X, podendo ambos aparecer simultaneamente. É o caso, por exemplo, do elemento *<regionBase>*, que deve ter no mínimo um elemento *<region>* ou um elemento *<importBase>* como filho.

O operador *A* seguido de índices delimita elementos dentre os quais apenas um deles deve aparecer exatamente uma vez e os outros não devem aparecer. Digamos que os elementos Y e Z, filhos do elemento X, tenham a cardinalidade *A*1. Isso significa que o elemento X deve possuir obrigatoriamente Y ou Z como filho, sendo que só será permitida a existência de um elemento Y ou Z como filho de X. Ainda seguindo esse exemplo, se os elementos P e Q também forem filhos de X, porém com cardinalidade *A*2, então obrigatoriamente, além de Y ou Z, o elemento X também deve ter como filho P ou Q, novamente, sendo possível eles aparecerem apenas uma vez.

A primitiva ATTRIBUTE especifica todos os atributos dos elementos XML. Para cada atributo de um determinado elemento é necessária uma nova entrada ATTRIBUTE, mesmo que diferentes elementos possuam o mesmo atributo. Exemplo disso é o atributo *id*, compartilhado pela maior parte dos elementos, o qual requer várias declarações. O primeiro argumento da primitiva é o nome do atributo, seguido do nome do elemento ao qual esse atributo se aplica. O terceiro argumento é um valor booleano que indica se esse atributo é ou não é obrigatório. Por fim, o quarto argumento deve ser o identificador de um dos tipos de dados definidos pela primitiva DATATYPE.

A primitiva DATATYPE define todos os tipos de dados que os atributos dos elementos podem possuir. Essa primitiva possui dois argumentos, sendo o primeiro o identificador do tipo de dado e o segundo uma expressão regular que é capaz de validá-lo. Assim, se quisermos criar um novo tipo de dado (booleano, por exemplo) poderíamos criar uma entrada DATATYPE com o identificador desse tipo (BOOLEAN) e uma expressão regular capaz de validar esse novo tipo de dado (^(*true*|*false*)$).

Por fim, há a primitiva REFERENCE. Essa primitiva é capaz de expressar referências que podem ser feitas entre os elementos. Em NCL, dizer que o elemento A referencia um elemento B significa dizer que há um atributo de A cujo valor equivale a um atributo que identifica o elemento B (alguns exemplos em NCL de atributos que identificam um elemento são *id, alias* e *name*). Os argumentos da primitiva REFERENCE são: o nome do elemento que faz a referência; o atributo do elemento que faz a referência; o nome do elemento que será referenciado; o atributo do elemento que será referenciado; e o escopo em que deve se encontrar o elemento que será referenciado.

Em linhas gerais, há três formas em que uma referência pode acontecer. Há elementos que podem ser referenciados a partir de qualquer escopo do código NCL. O atributo *id* de um elemento *<descriptor>*, por exemplo, pode ser referenciado pelo atributo *descriptor* de qualquer elemento *<media>* independente do escopo em que o elemento *<media>* se encontra. Nesse caso, dizemos que o escopo dessa referência é ANY.

Em outro caso, uma referência só é válida se ambos os elementos estiverem no mesmo escopo (ou perspectiva, usando o jargão NCL). O atributo *component* de um elemento *<port>,* por exemplo, deve referenciar o atributo *id* de um dos seguintes elementos: *<media>, <context>* ou *<switch>*. Essa referência só será válida, porém, se o elemento referenciado estiver na mesma perspectiva do elemento *<port>* (ou seja, se ambos os elementos forem filhos do mesmo elemento *<context>* ou *<body>*). O escopo dessa referência será SAME.

Por fim, em uma terceira forma de referência, o escopo é definido pelo valor de outro atributo. Tomando ainda o elemento *<port>* do exemplo anterior, seu atributo *interface* deve referenciar alguma das interfaces do elemento referenciado pelo atributo *component.* Assim, se o atributo *component* de um elemento *<port>* referenciar o atributo *id* de um elemento *<media>,* então o valor do atributo *interface* deverá ser igual a: (i) o atributo *id* de um elemento *<area>* filho do *<media>* referenciado; ou (ii) o atributo *name* de um elemento *<property>* filho do *<media>* referenciado. Este tipo de referência, no qual o escopo do elemento depende do valor de outro atributo, é indicada pela seguinte sintaxe: *$ELEMENT.ATTRIBUTE*. Para o caso do *<port>* citado, esse escopo ficaria *$THIS.component*. THIS é um açúcar sintático que representa o próprio elemento referenciador. Da mesma forma, PARENT é um açúcar sintático que representa o pai do elemento referenciador. Há também o açúcar sintático GRANDPARENT, que representa o pai do pai do elemento referenciador.

Para exemplificar o uso da metalinguagem descrita nesta subseção, tomaremos o elemento *<regionBase>* como caso de uso. De acordo com o definido em [1] este elemento possui três atributos, não sendo nenhum deles obrigatórios: *id*, *device* e *region*. São definidos dois possíveis filhos: *<importBase>*, e *<region>*, sendo que no mínimo um desses filhos deve ser definido pelo menos uma vez. O elemento *<regionBase>* tem como pai o elemento *<head>,* sendo que um documento pode conter vários ou nenhum elemento *<regionBase>*. A Tabela 1 extraída de [1] faz um resumo dos atributos e dos elementos filho do elemento *<regionBase>*.

Tabela 1. Atributos e filhos do elemento *<regionBase>*.

Elemento	Atributo	Conteúdo
regionBase	*id,device, region*	*(importBase | region)**

A Listagem 2 ilustra parte do documento que representa na metalinguagem o elemento *<regionBase>* em um perfil de NCL. São exibidas as entradas que definem o elemento *<regionBase>*, seus atributos, seus elementos filho e a referência que o atributo *region* faz a um dos elementos *<region>* definidos no documento.

```
1. ELEMENT(regionBase,head,*,false)
2. ATTRIBUTE(regionBase,device,false,DEVICE)
3. ATTRIBUTE(regionBase,id,false,ID)
4. ATTRIBUTE(regionBase,region,false,STRING)
5. ELEMENT(region,regionBase,#,false)
6. ELEMENT(importBase,regionBase,#,false)
7. REFERENCE(regionBase,region,region,id,ANY)
```

Listagem 2. Parte da metalinguagem que define o elemento *<regionBase>*

Como só há um pai possível (*<head>*) para o elemento *<regionBase>*, uma entrada da primitiva ELEMENT é suficiente para definí-lo (linha 1). Em um documento NCL podem existir zero ou mais elementos desse tipo, logo sua cardinalidade é expressa pelo operador *. O conceito de perspectiva não se aplica a este elemento, o que justifica seu último argumento como sendo *false.* Três entradas ATTRIBUTE são necessárias para definição dos atributos (linhas 2 a 4). Para economizar espaço, foram suprimidas as primitivas DATATYPE que definem as expressões regulares que validam os tipos dos atributos. Os possíveis filhos do elemento *<regionBase>* são definidos por duas entradas da primitiva ELEMENT (linhas 5 e 6). Como ao menos um desses filhos deve aparecer no mínimo uma vez, a cardinalidade de ambos é expressa pelo operado #. Por fim, é definida uma referência (primitiva REFERENCE) que pode ser feita por meio do atributo *region*, que referencia o atributo *id* de outro elemento *region*. Como o elemento *<region>* pode ser referenciado de qualquer parte do documento, o último argumento dessa primitiva é ANY (linha 7).

3.2 Validação incremental

Validar incrementalmente uma instância de código requer inicialmente identificar partes do documento que sofreram modificações desde a última validação e também partes possivelmente afetadas por essas modificações. Uma vez identificadas tais partes, a validação incremental se aplica apenas a elas e não ao documento como um todo. Nesse sentido, para validar incrementalmente um documento é preciso ter como premissa a possibilidade de validar apenas parte de um documento, o qual é tratado neste trabalho como validação parcial.

Quando se projeta uma ferramenta de validação para ser reutilizada por diversas ferramentas, vários possíveis cenários devem ser levados em conta. O processo de validação deve ser robusto para validar desde pequenos documentos até instâncias de código com um elevado número de elementos da linguagem.

Diversas estruturas de dados abstratas podem ser usadas para armazenar de forma eficaz uma instância de código NCL. A título de exemplificação, citam-se três delas: árvores XML; tabelas de elementos e atributos; e o grafo temporal hipermídia [4]. Para possibilitar a validação incremental, é fundamental que a estrutura de dados escolhida para representação do código possibilite acesso não-linear aos elementos, caso contrário, o simples fato de percorrer todos os elementos da estrutura de dados para encontrar o elemento buscado tornaria a validação incremental ineficiente.

Durante o desenvolvimento de aplicações, é comum que o procedimento de validação de código seja acionado em diversos momentos distintos da autoria (algumas ferramentas acionam a validação a cada modificação no documento mesmo que tal modificação seja mínima, como, por exemplo, a inclusão de um caractere no código). No intervalo entre uma validação e outra, o usuário realiza diversas edições no documento. A validação, portanto, deveria ser realizada apenas nas partes modificadas pelo usuário, bem como nas que poderiam ter sido influenciadas por tais modificações, descartando a necessidade de checagem de trechos do documento que não sofreram qualquer influência derivada dessas mudanças. Percebe-se que parte da dificuldade em se implementar uma validação incremental é a determinação de quais trechos do código devem ser validados em uma dada execução.

Este trabalho levanta duas possíveis abordagens que dão suporte à necessidade de avaliação de quais partes do documento NCL realmente necessitam ser validadas: uma baseada em um registro de operações e outra baseada na marcação de elementos. A primeira alternativa guarda um conjunto de operações realizadas no documento desde a última validação. Há basicamente três possíveis operações: adição, edição e remoção de elementos. É possível manter um registro dessas operações à medida que elas ocorrem. De posse desse registro, a avaliação de quais elementos devem ser validados é intuitiva: os elementos que devem ser checados são aqueles que participaram de alguma das operações presentes no registro. Tal abordagem tem a vantagem de permitir a realização de otimizações no registro, retirando operações excludentes (por exemplo, uma operação de adição de um elemento seguida da remoção do mesmo).

Na segunda abordagem, ao invés de se manter o registro, é feita uma marcação em todos os elementos que sofreram alterações. Isso gera criada uma lista de elementos que necessitam ser validados. De posse dessa lista, o processo de validação tem seu trabalho reduzido, limitando-se a percorrê-la, realizando o procedimento de validação sobre cada um dos elementos. Tal método tem a vantagem de não necessitar da gravação dos registros de operações, porém dificulta a implementação de rotinas de otimização na escolha de quais elementos validar.

Por NCL trabalhar com a idéia de referências, em ambas as abordagens deve existir um mecanismo que identifique partes do documento afetadas pela mudança de um determinado elemento. Este trabalho propõe que cada elemento passível de ser referenciado tenha associado uma lista de elementos que o aponte, permitindo a descoberta direta de quais partes do documento são influenciadas por mudanças sofridas naquele elemento.

4. ARQUITETURA PROPOSTA

A partir deste ponto adotaremos a seguinte nomenclatura: Validador será o componente que implementa o método descrito por este trabalho; Entidade será o componente que faz uso do Validador para realizar a validação de código NCL. A Figura 1

detalha a arquitetura proposta para validação NCL. Nota-se que o Validador não é utilizado diretamente pela Entidade, mas sim por um componente lógico Adaptador, que mantém um modelo de dados próprio para representar o documento NCL. A Entidade, por sua vez, comunica-se com este componente para conhecer o resultado da validação. Tais componentes são descritos nas subseções a seguir.

Para aprimorar a portabilidade da ferramenta, a mesma foi desenvolvida na linguagem de programação C, de acordo com o padrão ANSI-C, que é utilizado por diversos sistemas operacionais. O objetivo é permitir que o Validador possa ser utilizado por diferentes Entidades, que podem variar de ambientes de autoria de alto nível, como o Composer, até os receptores com o padrão Ginga-NCL e recursos limitados.

Figura 1. Arquitetura Proposta

4.1 Modelo de Dados

O modelo de dados representa um documento NCL. O modelo assume importante papel na validação parcial. Ele é responsável por indicar quais elementos devem ser validados, em especial os elementos que não foram alterados, mas que devem ser validados por dependerem de outro elemento que foi alterado.

Este mecanismo utiliza a abordagem de marcação de elementos modificados. Para isso são utilizados uma fila de validação e um mapa de referências. A fila de validação armazena os elementos que foram alterados desde a última validação. Esses elementos devem ser validados na mesma ordem seqüencial em que foram alterados, para garantir que o estado interno do modelo esteja fiel ao da entidade. O mapa de referências associa um elemento e a lista de elementos que o apontam. Nesse caso, quando um elemento chave desse mapa for alterado, a lista de elementos que o aponta deverá ser validada para garantir a consistência do documento após a alteração.

4.2 Adaptador

O componente Adaptador atua como um elo lógico entre o Validador e a Entidade. Seu papel é manter o modelo atualizado com as mudanças realizadas no documento. Isto permite ao modelo retratar o estado atual do documento NCL e marcar os elementos que precisam ser validados de acordo com as mudanças feitas. Caso a Entidade trabalhe diretamente com o modelo, a existência desse componente é desnecessária.

4.3 Validador

O componente Validador é o núcleo da validação. Nele são implementadas as idéias de representação da linguagem alvo extraídas da especificação feita na metalinguagem proposta, é feito suporte a validação parcial e a mensagens multilíngüe. Sua arquitetura é modularizada visando baixo acoplamento. A Figura 2 detalha essa arquitetura.

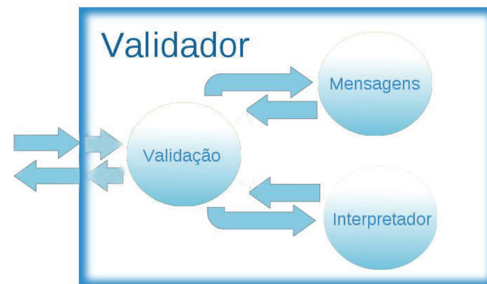

Figura 2. Arquitetura do Componente Validator

Para cada elemento a ser validado, é feito um conjunto de testes contextuais e referenciais. O módulo de Validação desconhece particularidades de NCL, ele executa testes genéricos e através do módulo Interpretador obtém a resposta de cada caso de teste particular. Esta divisão lógica permite que o código da validação seja mantido, caso a representação da linguagem seja alterada.

O módulo Interpretador atua extraindo a representação de NCL da metalinguagem. De posse da representação da linguagem, conhece detalhes da mesma e fornece funções que permitem tratar casos de testes particulares, como, por exemplo, saber se um determinado atributo de um elemento é obrigatório.

O módulo Mensagens armazena as mensagens que são utilizadas pela validação. Cada possível incoerência possui um registro e uma respectiva mensagem textual amistosa. Dessa maneira, para cada incoerência encontrada na validação, o módulo Validação busca a respectiva mensagem e a devolve no final da validação. Esta abordagem permite que as mensagens retornadas sejam alteradas ou adaptadas a outras línguas sem interferir no algoritmo da validação.

5. CONCLUSÃO E TRABALHOS FUTUROS

A validação de código fonte realiza uma importante tarefa em diversos cenários na computação, variando desde a resposta direta aos programadores dos erros cometidos em tempo de autoria até o impedimento da execução de programas maliciosos ou corrompidos. Diversos são os trabalhos que contemplam a validação de linguagens imperativas. Para as linguagens declarativas, observa-se pouca pluralidade.

Este trabalho descreve uma nova abordagem de validação de documentos NCL. Tal abordagem apóia-se no uso de uma metalinguagem capaz de expressar parte das peculiaridades de NCL, o que pode tornar o processo de validação mais genérico. Espera-se que a metalinguagem proposta possa ser aplicada para validação de outras linguagens declarativas, mas, no geral, estender um validador que use a abordagem proposta por este trabalho para a validação de outras linguagens declarativas equivale a estender a metalinguagem proposta para atender às novas necessidades. Outra contribuição deste trabalho é o suporte à validação NCL incremental. Apesar do estudo de caso ter se baseado na linguagem NCL, tal funcionalidade é passível de ser implementada por ferramentas de validação que se proponham a validar outras linguagens. Como última contribuição cita-se a internacionalização das mensagens oriundas do processo de validação, buscando atender a um número maior de usuários.

Alguns trabalhos futuros são identificados, dentre os quais se destacam a inclusão de técnicas que possibilitem a um autor de aplicação definir novas regras de validação, permitindo a

checagem de aspectos da linguagem que não se resumam às regras sintáticas e estruturais de NCL. Outro trabalho futuro é a necessidade de testes de desempenho, buscando medir o quanto a validação incremental é de fato superior à validação convencional, além de tentar fazer um levantamento qualitativo sobre a ordem de grandeza em número de elementos que um documento pode conter de forma que nossa proposta aplique a validação em tempo de processamento razoável.

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] Bouyakoub, S. and Belkhir, A. 2011. SMIL Builder: an incremental authoring tool for SMIL documents. ACM Trans. Multimedia Comput. Commum. Appl. 7, 1, Article 2 (January 2011), 30 pages.

[3] Bouyakoub,S. and Belkhir, A. 2008. H-SMIL-Net: A hierarchical Petri net model for SMIL documents. In *Proceedings of the 10 th Conference on Computer Modeling and Simulation* (UKSIM'08).

[4] Costa, R. M. R., Soares, L. F. G. Modelo Temporal Hipermídia para Suporte a Apresentação em Ambientes Interativos. XIII Simpósio Brasileiro de Sistemas Multimídia e Web. WebMedia 2007, Outubro 2007.

[5] Document Object Model (DOM) Level 1 Specification Version 1.0. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 1998. W3C Recommendation.

[6] DTD. http://www.w3schools.com/DTD/

[7] Eclipse CDT. http://www.eclipse.org/cdt/

[8] Eclipse JDT. http://www.eclipse.org/jdt/

[9] Eclipse. http://www.eclipse.org/

[10] Honorato, G. S. C., e Barbosa, S. D. J.. 2010. NCL-inspector: towards improving NCL code. In Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)

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

[12] Kostoulas, M. G., Matsa, M, Mendelsohn, N., Perkins, E., Heifets, A. and Mercaldi, M. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization, Proceedings of the 15th international conference on World Wide Web, May 23-26, 2006, Edinburgh, Scotland

[13] L.F.G Soares; R.F Rodrigues. Nested Context Model 3.0: Part 1 - NCM Core, Technical Report, Departamento de Informática PUC-Rio, May 2005, ISSN: 0103-9741. Ding, W. and Marchionini, G. 1997. *A Study on Video Browsing Strategies*. Technical Report. University of Maryland at College Park.

[14] Lima, B. S., Azevedo, R. G. A., Moreno, M. F. e Soares, L. F. G. Composer 3: Ambiente de autoria extensível, adaptável e multiplataforma. II Workshop de TV Digital Interativa - WTVDI (WebMedia 2010). Belo Horizonte, Brasil – Outubro de 2010.

[15] LuaEclipse. http://luaeclipse.luaforge.net/.

[16] NCL Validator. http://http://laws.deinf.ufma.br/nclvalidator/.

[17] Newman, R. M. and Gaura, E. I. 2003. Formal design of SMIL presentation. In *Proceedings of the 21st Annual International Conference on Documentation (SIGDOC'03)*

[18] Peterson, J. L. Petri Net Theory and the Modeling of Systems, Prentice Hall PTR, Upper Saddle River, NJ, 1981

[19] Santos, R. C. M., Gomes, T. A., Azevedo, R. G. A., Soares Neto, C. S. e Teixeira, M. M. Correção de Código Semi-Automática em Nested Context Language. XVI Simpósio Brasileiro de Sistemas Multimídia e Web. WebMedia2010. Belo Horizonte, Brasil - Outubro de 2010.

[20] 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.

[21] Shneiderman, B. 1982. Designing computer system messages. Commun. ACM 25, 9 (September 1982)

[22] Synchronized Multimedia Integration Language (SMIL 3.0), 2008. http://www.w3.org/TR/REC-smil/ W3C Recommendation

[23] Soares Neto, C. S., e Soares, L. F. Reúso e Importação em Nested Context Language. . XV Simpósio Brasileiro de Sistemas Multimídia e Web. WebMedia2009. Fortaleza, Brasil - Outubro de 2009.

[24] Xerces. http://xerces.apache.org/.

[25] XML Schema Definition Language (XSD) 1.1 Part 1: Structures. 2009. http://www.w3.org/TR/xmlschema11-1/ W3C Recommendation.

[26] XML. Extensible Markup Language (XML) 1.0 (Fifth Edition). 2008. http://www.w3.org/TR/xml/ W3C Recommendation

[27] XSL Transformations (XSLT). Version 1.0. http://www.w3.org/TR/xslt. W3C Recommendation

[28] Yang, C. C., and Yang, Y. Z. SMILAuthor: An Authoring System for SMIL-Based Multimedia Presentations, Multimedia Tools and Applications, v.21 n.3, p.243-260, Dezembro 2003

[29] Yang, C. C., 2000. Detection of the time conflicts for SMIL-based multimedia presentations. In *Proceedings of the International Computer Symposium Workshop on Computer Networks, Internet and Multimedia.*