
# PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 10/11

# Nested Context Language 3.0
# Parte 14 - Suíte de Testes de Conformidade para o Ginga-NCL

**Eduardo Cruz Araújo**
**Luciana Rosa Redlich**
**Vinicius Lago**
**Marcelo Ferreira Moreno**
**Luiz Fernando Gomes Soares**

Departamento de Informática

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO**
**RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900**
**RIO DE JANEIRO - BRASIL**

Monografias em Ciência da Computação, No. 10/11 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Setembro, 2011

# Nested Context Language 3.0

# Parte 14: Suíte de Testes de Conformidade para o Ginga-NCL

Eduardo Cruz Araújo, Luciana R. Redlich, Vinicius Lago, Marcelo F. Moreno, Luiz Fernando G. Soares.

Laboratório TeleMídia DI – PUC-Rio

Rua Marquês de São Vicente, 225, Rio de Janeiro, RJ - 22451-900.

{edcaraujo,lrosa,vclago,moreno,lfgs@telemidia.puc-rio.br}

**Abstract.** This paper describes the development of the compliance test suite for the Ginga-NCL, an ISDB-TB standard for digital terrestrial TV and ITU-T H.761 Recommendation for IPTV services.Both the test suite specification, i.e., the set of its test cases, as the several possibilities for its application are discussed. The peculiarities with regards the development of conformance tests for systems designed for declarative languages are argued, in particular those found in developing tests for middleware systems aiming at XML-based declarative environments. In this respect, the paper points out what the proposed suite has brought as contributions to the state of the art. Because it is an extensible suite, the article also brings the rules, adopted by the ITU-T, for its extension.

**Keywords**: Nested Context Language, Ginga-NCL, compliance test suite.

**Resumo**. Essa monografia descreve a trajetória percorrida no desenvolvimento da suíte de testes de conformidade para o Ginga-NCL, padrão ISDB-TB para TV digital terrestre e Recomendação ITU-T H.761 para serviços IPTV. Tanto a especificação da suíte de testes, ou seja, o conjunto de seus casos de teste, quanto as diversas possibilidades para sua aplicação são discutidos. As peculiaridades do desenvolvimento de testes de conformidade para sistemas projetados para linguagens declarativas são tratadas, em particular aquelas encontradas no desenvolvimento de testes para sistemas de middlewares focados em ambientes declarativos baseados em XML. Nesse aspecto, a monografia ressalta o que a suíte proposta trouxe de contribuição para o estado da arte. Por ser uma suíte extensível, o artigo traz também as regras, adotadas pelo ITU-T, para sua extensão.

**Palavras-chave**: Nested Context Language, Ginga-NCL, suíte de testes de conformidade.

**Responsável por publicações**:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br

# Sumário

# 1 Introdução

O middleware Ginga é composto obrigatoriamente por um ambiente declarativo (Ginga-NCL), responsável pela execução de aplicações NCL (Nested Context Language) e sua linguagem de script Lua, e opcionalmente por um ambiente imperativo (Ginga-Imp).

No caso do sistema ISDB-TB (International Standard for Digital Broadcasting) [ABNT NBR 15606-2, 2007], adotado no Brasil, o ambiente Ginga-NCL é obrigatório tanto para receptores fixos quanto para receptores portáteis. O Ginga-Imp adota a linguagem Java (Ginga-J), sendo obrigatório apenas para receptores fixos. Hoje o ISDB abrange uma população de mais de 550 milhões potenciais usuários, incluindo aquela dos 11 países latino americanos que o adotaram.

Ginga-NCL e sua linguagem NCL são os únicos padrões que abrangem múltiplas plataformas de transmissão. Além de reconhecido pela Recomendação ITU-R BT-1699 [ITU-R Recommendation BT-1699, 2009] para TV digital (TVD) terrestre, é também Recomendação ITU-T H.761 [ITU-T Recommendation H.761, 2009] para serviços IPTV. Para IPTV, o Ginga se resume ao ambiente Ginga-NCL.

Proposto em 2006 para o Sistema Brasileiro de TV Digital (SBTVD), o Ginga já mencionava a necessidade de uma suíte de testes para garantir sua implementação conforme, para evitar que um legado de produtos não-conformes às normas chegasse ao mercado. No caso do SBTVD, a não obrigatoriedade de uma certificação (o sistema adotado é de auto-certificação) atrasou a especificação e implementação da suíte de testes, e hoje já existe um legado que, embora ainda pequeno, merece atenção. O enorme mercado já existente, mencionado nos parágrafos anteriores, aponta a urgência na definição de uma suíte de testes.

Visando resolver esse problema, e antecipando seu alcance nos serviços IPTV, a Questão 13 do Grupo de Estudos 16 do ITU-T ("Multimedia Application Platforms and End Systems for IPTV"), resolveu criar uma suíte de testes de conformidade para o Ginga-NCL, adotando o modelo proposto pelos autores deste artigo.

A monografia descreve a trajetória percorrida até o momento no desenvolvimento dessa suíte, ressaltando as peculiaridades do desenvolvimento de testes de conformidade para sistemas desenvolvidos sobre linguagens declarativas e apresentando o que a suíte do Ginga-NCL traz de contribuição para o estado da arte.

O restante da monografia segue a seguinte organização. A Seção 2 discute alguns conceitos sobre testes de conformidade, para salientar as particularidades exigidas pelas linguagens declarativas. Nessa seção são definidos vários conceitos, utilizados na proposta do artigo, tanto na especificação da suíte de testes de conformidade, quanto nas diversas formas de sua aplicação. A Seção 3 descreve alguns trabalhos relacionados, chamando atenção para as diferenças introduzidas na presente proposta. A Seção 4 trata da especificação da suíte de testes de conformidade. Em particular, ela traz uma discussão sobre os casos de teste de uma suíte para um sistema desenvolvido sobre uma linguagem XML, retomando a discussão iniciada na Seção 2. Na Seção 5 são apresentadas as várias opções de aplicações dos casos de teste. Finalmente, a Seção 6 é reservada às Conclusões

## 2 Testes de conformidade para linguagens declarativas

No desenvolvimento de um sistema de software, várias suítes de teste devem se fazer presentes, como por exemplo, aquelas que verificam o desempenho e a integridade (robustez) do sistema, entre outras.

No desenvolvimento de um sistema que deve seguir uma especificação padrão, é de suma importância uma suíte de testes que verifica a aderência de uma implementação do sistema à especificação. Tal conjunto de testes compõe a suíte de testes de conformidade. Testes de conformidade são testes do tipo caixa preta [MYERS04], ou seja, têm como finalidade verificar a funcionalidade especificada e o comportamento observável, sem que seja necessário saber como é realizada a implementação.

Quase todo estudo de testes de sistemas de software é voltado para sistemas desenvolvidos sobre linguagens imperativas. Apesar do paradigma de desenvolvimento ser completamente diferentes daqueles usados em linguagens declarativas, que são o foco deste artigo, o objetivo principal das diversas técnicas desenvolvidas continua a ser o mesmo, encontrar falhas no software: no caso deste artigo, falhas de conformidade. Entre as técnicas que mais nos interessam estão os testes unitários (ou de unidade), os testes de integração e os testes de sistemas. Esta seção tem como objetivo analisar essas técnicas, apresentando como contribuição suas adaptações a sistemas desenvolvidos sobre linguagens declarativas, em especial linguagens de marcação baseadas em XML, ou seja, *aplicações XML*.

Teste unitário é implementado com base no menor elemento testável (unidade) do software. Em programação procedural, uma unidade pode ser uma função individual ou um procedimento. Em sistemas desenvolvidos sobre linguagens baseadas em XML (uma aplicação XML) uma unidade pode ser a atribuição de um valor a um atributo de um elemento da linguagem.

A fase de teste de integração sucede a de teste de unidade, e tem como objetivo encontrar falhas provenientes da integração interna dos componentes de um sistema. Nessa fase, os componentes são combinados e testados em grupo.

Idealmente, cada teste de unidade é independente dos demais, o que possibilita ao programador testar cada módulo (ou elemento no caso de uma aplicação XML) isoladamente. Isso, no entanto, é impossível na realização de testes de conformidade para sistemas desenvolvidos sobre linguagens declarativas baseadas em XML. A interdependência entre atributos de um mesmo elemento, ou mesmo a interdependência (hierárquica ou não) dos diversos elementos da linguagem, torna impossível um teste de unidade.

Tomemos como exemplo uma linguagem NCL, onde um elemento <body> pode conter como filhos elementos <context> e <media>.

```
<body>
     <media attr1="val1" attr2="val2" …/>
     <context>
          <media/>
     </context>
</body>
```

Em NCL, um elemento <media> necessariamente deve ser descendente de um elemento <body>. Assim, é impossível fazer um teste unitário para atribuição de valor a uma propriedade de um elemento <media>, sem, ao mesmo tempo, testar a pertinência desse elemento ao elemento <body>. Em outras palavras, é impossível definir

uma entrada do caso de teste (que como veremos na Seção 4 será uma aplicação NCL), sem incluir o elemento <body>. Em última análise, para esse caso de teste específico "assume-se" que a pertinência já foi verificada. Note que não é necessário assumir que o teste unitário de <body> foi bem sucedido (isto é, a pertinência de elementos <context> e <media>), mas apenas parcialmente atendido (a pertinência de elementos <media>).

A impossibilidade da realização de testes de unidade "puros" para conformidade de sistemas desenvolvidos sobre linguagens baseadas em XML, nos leva a buscar casos de teste tendo como alvo uma unidade, mas que sempre deverão levar em conta que outras unidades estarão presentes, e que elas possivelmente ainda não foram testadas e podem influir nos resultados obtidos. Note que esses testes embora bastante similares aos testes de integração, são diferentes, pois esses últimos assumem como pressupostos que testes unitários já foram realizados sobre todos os componentes envolvidos no teste. Devido à similaridade, chamaremos esses testes de *testes integrados*, mas salientando mais uma vez que eles não são de fato testes de integração.

Deve-se notar que para uma dada unidade alvo, pode haver mais de um (geralmente muitos) testes integrados, cada um deles tendo a mesma unidade alvo, mas diferentes unidades componentes. Quanto menos esses testes envolverem outras unidades que não a unidade alvo, mais próximos estaremos dos testes unitários. As entradas dos testes integrados são de fato pequenas aplicações especificadas na linguagem XML.

Dentro de um conjunto de testes integrados que tem como alvo o teste de uma unidade de uma aplicação XML, introduziremos o conceito de *teste integrado minimalista*, definindo-o como aquele caso de teste que contém o menor número possível de elementos e atributos, diferentes daquele atributo/elemento alvo do teste. Uma suíte de testes de conformidade para uma aplicação XML deve procurar identificar, para cada conjunto de testes integrados referentes a uma unidade, aquele que é considerado o minimalista.

Outro conceito torna-se também importante: uma *sequência de testes integrados* de conformidade verifica se vários elementos e atributos de uma especificação de sistema desenvolvido sobre uma linguagem declarativa XML estão conformes. A escolha de uma boa sequência de testes integrados é fundamental na aplicação de testes de conformidade. Dependendo da ordem escolhida, pode ser que as interdependências com os elementos que não são o alvo de testes não afetem o caso de teste em questão, uma vez que esses elementos podem já ter sido validados, na aplicação passo a passo da sequência.

Tomemos o seguinte exemplo. Suponha que o teste integrado T1 de uma unidade A depende também que uma unidade B e uma unidade C sejam assumidas, ao menos parcialmente, como corretas. Suponha também que outro teste T2 integrado para A depende que uma unidade D seja assumida, ao menos parcialmente, como correta. Suponha ainda que o teste integrado T3 da unidade D também depende que a unidade B e a unidade C sejam assumidas, ao menos parcialmente, como corretas. Finalmente, vamos assumir que seja possível testar as unidades B e C de forma unitária (testes T4 e T5 respectivamente). Se quisermos testar as unidades A, B, C e D, as seguintes sequências seriam possíveis:

SEQ1: T4→T5→T3→T1

SEQ2: T4→T5→T3→T2

SEQ3: T2 →T4→T5→T3

Note que T2 seria o teste integrado minimalista para A, comparado com T1, no entanto, dado que T4 e T5 foram realizados com sucesso, tanto T1 como T2 e T3, têm todas as pré-condições, sobre os componentes que não são o alvo de teste, satisfeitas nas sequências SEQ1 e SEQ2; ou seja, a utilização de casos de teste integrado minimalistas não seria importante nesse caso. Note também que SEQ1 utiliza um teste integrado não minimalista (T1), mas que a sequência é melhor que SEQ3, uma vez que se T2 falhar logo na primeira aplicação, é impossível saber se a falha se deu por causa da unidade A ou D.

Diferente dos casos de testes unitários, a sequência de aplicação de testes integrados é muito importante. Mais ainda, o fato de usarmos casos de teste de conformidade minimalistas em uma dada sequência não garante que ela seja a melhor para testes, como vimos no exemplo anterior. A única coisa que se pode garantir é que: se os casos de teste integrado minimalistas não formam uma interdependência circular (por exemplo, A depende de B, que depende de C, que depende de A) é sempre possível formar uma sequência de testes com o menor grau de interdependências não resolvidas na aplicação da sequência, usando apenas casos de teste integrados minimalistas.

Voltando às definições clássicas, seguindo a fase de testes de unidades e de integração, a fase de teste de sistema tem como objetivo executar o sistema sob o ponto de vista de seu usuário final, varrendo as funcionalidades em busca de falhas em relação aos objetivos originais. Os testes são executados em condições similares – de ambiente, interfaces sistêmicas e massas de dados – àquelas que um usuário utilizará no seu dia-a-dia de manipulação do sistema.

No caso específico de um sistema de TV digital terrestre desenvolvido sobre uma linguagem declarativa XML, todos os casos de teste integrados podem ter suas entradas aplicadas diretamente no sistema alvo de teste, sem a necessidade de codificá-los no formato final de transmissão que, no caso do SBTVD, são fluxos TS com as aplicações transportadas em carrosséis de objetos. No entanto, para testes de sistema isso se faz necessário. Assim, cada caso de teste de sistema gera um stream de teste que se caracteriza como uma aplicação, no nosso caso, declarativa, similar àquelas transmitidas pelos difusores de conteúdo, buscando testar ao máximo possível todas as funcionalidades do sistema. Note assim que todos os casos de teste integrado, uma vez codificados como fluxos para transmissão, podem ser considerados teste de sistema.

Temos usado, até o presente momento, os termos “caso de teste” e “suíte de testes” sem precisá-los. A Seção 4 trará uma definição específica para o foco deste artigo.

## 3 Trabalhos Relacionados

Em geral, testes de conformidade são utilizados para aumentar o nível de confiança na qualidade final do produto. Mais especificamente, quando se fala de padrões abertos, a interoperabilidade tem um papel essencial. Isso vale tanto para sistemas desenvolvidos sobre linguagens imperativas, quanto para aqueles desenvolvidos sobre linguagens declarativas.

No caso específico de sistemas desenvolvidos sobre linguagens imperativas, a avaliação de conformidade tira proveito do próprio paradigma para automatização de seus testes. Esse paradigma trabalha em um nível de instrução, descrevendo uma lista de comandos que o computador deve executar. Uma vez que cada comando possui uma função bem definida e independente (ao menos é o que se espera), os testes de conformidade podem focar na avaliação de cada um desses comandos. Assim, é pos-

sível definir testes unitários para sistemas desenvolvidos sobre esse tipo de linguagem, ao contrário do que acontece com sistemas desenvolvidos sobre linguagens declarativas, especialmente aquelas baseadas em XML, que exigem testes integrados, como introduzido e discutido na Seção 2. Neste artigo estamos focando apenas nos testes de conformidade para os executores (*players*) de linguagens declarativas baseadas em XML.

Em alguns casos, linguagens imperativas de scripts são utilizadas como suporte às declarativas. Por servirem de extensão a outra linguagem, é natural se pensar que antes mesmo de avaliar a conformidade de executores da linguagem principal, no caso a declarativa, deve-se verificar a conformidade dos executores (*engines*) da linguagem auxiliar. Para a NCL, a linguagem de extensão é Lua [IERUSALIMSCHY06], em sua API NCLua [SSC08]. A maioria das linguagens citadas nesta seção utilizam a linguagem ECMAScript. Os detalhes sobre a avaliação da máquina NCLua fogem, no entanto, do foco desta monografia.

Sempre que existe um esforço de padronização, existe uma preocupação com relação à conformidade e à interoperabilidade das implementações. Vários padrões de linguagens declarativas, como, por exemplo, XML, HTML, SMIL e SVG, definem um espaço, em cada uma de suas versões, para especificações dos casos de teste de conformidade de seus executores. De modo geral, essas especificações são bem semelhantes, incluindo, para cada caso de teste, uma referência para a recomendação, a descrição do caso de teste, a suas entradas e os comportamentos esperados. Um manual para o desenvolvimento de novos testes e para contribuição pública é também comum.

A suíte de teste para parsers XML [W3C XML] é composta por cerca de 2000 arquivos de testes, acompanhado de um relatório contendo informações gerais sobre a suíte e a descrição de cada arquivo. Os testes são divididos em dois grandes tipos: testes binários e testes de saída.

Os testes binários devem ser tratados por cada tipo de *parser* XML de forma consistente, seja aceitando o arquivo de entrada do teste (teste positivo) ou rejeitando (teste negativo). A recomendação XML 1.0 (Segunda Edição) define dois tipos de *parsers*: os *validating*, que checam produções especiais e requerem documentos XML bem formados; e os *novalidating*, que não obrigam a inserção de entidades externas referenciadas aos arquivos de entrada de teste, não reportando, assim, erros que seriam detectados a partir da leitura dessas entidades.

Os testes de saída são aqueles responsáveis por verificar o retorno de informações dos *parsers* XML para as aplicações que utilizam o parser.

Diferente do XML, o HTML, SMIL e SVG realizam a divisão de seus casos de teste por áreas funcionais da linguagem, facilitando a navegação e fechando o escopo de um determinado grupo de casos de teste.

A especificação da suíte de teste de conformidade para (*players*) HTML4 [HTML4] consiste de uma documentação geral e dos diversos casos de teste. Cada caso de teste consiste de uma assertiva da especificação da linguagem e de um arquivo que contém a entrada do teste e a descrição do comportamento esperado. Um detalhe especial é que alguns desses casos de teste são dependentes de elementos que contém código ECMAScript 262, linguagem de script do HTML 4. Isso sugere que para realizar uma avaliação mais precisa é preciso antes executar a suíte de teste para (*engines*) ECMAScript 262, eliminando, teoricamente, mais um ponto de falha para o teste. A suíte de teste para (*engines*) ECMAScript 262 [ECMAScript] pode ser realizada de forma completamente automatizada, ao contrário do que acontece com a suíte para (*players*)

HTML 4, que foi prevista para ser realizada de modo manual, como apresentado na Seção 5.

A suíte para (*players*) SVG [SVG] possui detalhes específicos na divisão e organização dos seus casos de teste de conformidade. Na suíte [13 e 14], além dos mesmos detalhes encontrados na suíte para o (*player*) HTML4, existe a especificação de uma ordem de execução para os testes. Assim, recomenda-se que as aplicações dos testes avaliem inicialmente os testes mais básicos e simples, e só depois os testes mais complexos e de funcionalidades mais avançadas. A suíte divide os casos de teste em dois grandes grupos, os de renderização estática, relacionados à alta qualidade dos desenhos, e os de dinâmica, relacionados a animações e *scriping*. Ambos os grupos possuem ordem de execução definidas para cada um dos seus módulos. Adicionalmente, são definidos ainda quatro tipos genéricos de teste, aplicáveis igualmente, tanto para a parte estática, quanto para a parte dinâmica. São eles:

- *Basic effectivity* (BE) – verifica de forma rudimentar a capacidade de uma dada área funcional;
- *Detailed* (DT) – abrange todos os requisitos de uma área funcional;
- *Error* (ER) – em que o interpretador deve identificar condições de erro na especificação do SVG;
- *Demo* (DM) – casos de SVG do "mundo real", gerados a partir de produtos, idealmente complexos.

A especificação desses quatros tipos de teste tem a intenção de fornecer um *checklist* de alto nível para um conjunto de casos de teste de uma dada área funcional.

Mais simples que a suíte para (*players*) SVG, a especificação de testes de conformidade para (*players*) SMIL [15 e 16] realiza uma organização bem mais próxima da suíte para (*players*) HTML4. A especificação é também dividida em áreas funcionais, onde é possível navegar entre os casos de teste e obter os arquivos a eles relacionados, além de suas respectivas descrições e referencia à especificação da linguagem. Assim como as outras suítes discutidas nesta seção, ela é especificada para aplicação manual, dependente do olhar humano para avaliação e iteração entre os testes.

Uma característica comum entre as suítes é a disponibilização de um relatório de interoperabilidade. Este relatório contém o resultado da execução da suíte por diferentes implementações dos executores das linguagens. O objetivo do relatório é apresentar que ao menos dois implementadores, de modo independente, produziram implementações interoperáveis para uma dada funcionalidade, partindo da mesma especificação, testando assim tanto a clareza da especificação, como a sua implementabilidade.

Nenhuma das suítes de teste mencionadas nesta seção discute a possibilidade de automatização da execução de seus casos de teste, projetados sempre visando uma aplicação de forma manual e exigindo uma avaliação humana. Ao contrário, a suíte proposta neste artigo também permite uma aplicação semiautomatizada, como discutido na Seção 5.

## 4 A especificação da Suíte de Testes para o Ginga-NCL

Um caso de teste de conformidade geralmente consiste de uma referência a um identificador ou requisito de uma especificação, pré-condições, uma série de passos a se seguir, uma entrada de dados e uma saída de dados (resultado esperado).

Uma coleção de casos de teste é chamada de suíte de teste. Geralmente, ela também contém instruções detalhadas, ou objetivos, para cada coleção de casos de teste, além de uma seção para descrição da configuração do sistema usado.

Uma suíte de testes de conformidade para o Ginga-NCL deve verificar a conformidade a cada assertiva encontrada na Recomendação H.761 do ITU-T [ITU-T Recommendation H.761, 2009], que coincidem com as assertivas presentes nas Normas [ABNT NBR 15606-2, 2007] e Guia operacional do ISDB-$T_B$ [ABNT NBR 15606-7, 2011]. Todas as assertivas especificadas deverão fazer parte da especificação e receberão um identificador "X". Cada assertiva pode gerar uma ou mais instruções gerais de como aplicar o teste e qual o resultado esperado. Cada instrução será identificada por "X.Y", onde X é o identificador da assertiva. Por sua vez, cada instrução pode ser satisfeita por uma ou mais entradas do teste, que no caso do Ginga-NCL são aplicações especificadas em NCL. Cada uma dessas entradas também será identificada. Na verdade, o identificador da entrada é o mesmo identificador do caso de testes "X.Y.Z", onde X é o identificador da assertiva e Y o identificador da instrução.

Um caso de teste para o Ginga-NCL possui a seguinte estrutura:

- O identificador da assertiva (*Name*);
- A referência para a Seção da Recomendação H.761 do ITU onde a assertiva pode ser encontrada (*Reference*);
- A assertiva (*Normative statement*);
- O alvo de testes (*Target*);
- A obrigatoriedade ou não do atendimento da assertiva (*Prescription Level*);
- O identificador da instrução (*Instruction Name*);
- O tipo da validação (*Validadtion Type*), detalhando se é um teste em que o esperado é a falha ou o sucesso;
- A instrução (*Instruction*);
- A identificação do caso de teste, especificado no atributo *id* do elemento <ncl> da entrada para o teste;
- O documento NCL que é a entrada para o teste;
- O comportamento esperado de saída, especificado em comentários colocados imediatamente depois do elemento <ncl> da entrada para o teste.
- A identificação do autor do caso de teste, especificada em um elemento <meta> a ser colocado como primeiro filho do elemento <head> do documento NCL de entrada do teste (<meta name="author" content="string com o nome do autor (entidade responsável) do teste"/>)

Deve-se notar que o formato acima refina o formato padrão IEEE 829 [IEEE Std 829,1998 e MYERS04] adaptando-o para testes de conformidade de sistemas desenvolvidos sobre linguagens baseadas em XML. Note também que esse formato estende todos os outros formatos apresentados na Seção 3, tornando mais rica a especificação dos casos de teste.

É importante ressaltar que o conjunto de entradas para o elemento ou atributo da linguagem alvo de teste pode ser muito grande, ou mesmo infinito e, portanto, é desejável que a suíte de testes seja extensível, e que seja permitido a outras pessoas contribuir com mais casos de testes.

Quanto às saídas do teste, normalmente elas não são valores, mas comportamentos observáveis, ou por um se humano, ou automaticamente, como será discutido na Seção 5.

Para um atributo alvo de teste, quanto mais entradas são fornecidas, mais rico será o teste. Numa situação ideal todas as entradas possíveis deveriam ser testadas, mas na ampla maioria dos casos isso é impossível, pois geraria um número absurdo, senão infinito, de casos de teste, como já mencionado. Uma abordagem mais realista para o teste de caixa-preta é escolher um subconjunto de entradas que maximize a qualidade do teste. Pode-se agrupar subconjuntos de entradas possíveis que são processadas similarmente, de forma que o teste de somente um elemento desse subconjunto serve para averiguar a qualidade de todo o subconjunto. Vamos ver três casos representativos em NCL, aproveitando para ilustrar a especificação dos casos de teste. Outros casos específicos podem, no entanto, existir, em que todas as entradas possíveis devem ser usadas para a geração de casos de testes.

No primeiro caso (CASO1), a propriedade *background* de um elemento mídia deve aceitar uma cor como valor, o que pode gerar um grande número de casos possíveis. Podemos, no entanto, definir os valores "transparent", "white", "black", "red", "green"e "blue" como representativos para todos os valores.

No segundo caso, o atributo *src* de um elemento <media> pode receber como valor uma URI. Inúmeros casos de teste podem ser derivados, podemos, no entanto, definir os casos usando um esquema específico (file:, http:, etc, entre todos aqueles possíveis) e a parte específica do sistema identificando um tipo de mídia diferente (video/mp4, audio/mp4, text/txt, image/png, etc.) para cada esquema escolhido. Uma especificação para cada tipo de esquema seria necessária, duas delas (CASO2a e CASO2b) sendo apresentadas a seguir.

CASO 1
#Name: property11
#Reference: ITU-T H.761 - 7.2.5
#Prescription level: Mandatory
#Validation type: Positive
#Target: Element <property name="background" value="?"…/>,
#Normative Statement:
The <body>, <context>, and <media> elements may have several embedded properties. Examples of these properties can be found among those that define the media object placement during a presentation, the presentation duration, and others that define additional presentation characteristics.

Reserved color names: "white", "black", "silver", "gray", red", "maroon", fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal". The background value may also be the reserved value "transparent". This can be helpful to present transparent images, like transparent GIFs, superposed on other images or videos.
#Instruction Name: property11.01
#Instruction:
Create a document containing a <media> element with a child <property> element whose name attribute is set to "background" and the value attribute is set to a valid color value ("white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", ou "teal"). The region in which the media object is rendered must present the background color set to the value. At least the "white", "black", "red", "green", "blue" and "transparent" color must be tested.
#NCL input documents: URI to the file of the test case 01; URI to the file of the test case 02, etc.

CASO 2a
#Name: media01
#Reference: ITU-T H.761 - 7.2.4
#Prescription level: Mandatory
#Validation type: Positive
#Target: Element <media src="?"/>
#Normative Statement:
Each media object has two main attributes, besides its *id* attribute: *src*, which defines a URI of the object content, and *type*, which defines the object type.
#Instruction Name: media01.01
Create a document containing a <media> element whose *src* attribute is set to a valid location for a local media file. Test for each possible media type (at least one type of media object different from the types used in other URI schemas must be tested). The <media> element must be associated to a region on the screen. The media object must be presented in the defined region.
#NCL input documents: URI to the file of the test case 01; URI to the file of the test case 02, etc.

CASO 2a
#Name: media01
#Reference: ITU-T H.761 - 7.2.4
#Prescription level: Mandatory
#Validation type: Positive
#Target: Element <media src="?"/>
#Normative Statement:
Each media object has two main attributes, besides its *id* attribute: *src*, which defines a URI of the object content, and *type*, which defines the object type.
#Instruction Name: media01.01
Create a document containing a <media> element whose *src* attribute is set to a valid location for a local media file. Test for each possible media type (at least one type of media object different from the types used in other URI schemas must be tested). The <media> element must be associated to a region on the screen. The media object must be presented in the defined region.
#NCL input documents: URI to the file of the test case 01; URI to the file of the test case 02, etc.

No terceiro caso, o atributo *left* de um elemento <region> pode receber uma percentagem na faixa [0,100] ou um valor inteiro especificando o atributo em pixels. Neste caso específico, podemos definir a especificação de uma percentagem qualquer como um caso único de teste, e de um valor qualquer em pixel como outro caso único de teste. Duas especificações se fazem então necessárias.

CASO 3a
#Name: region07
#Reference: ITU-T H.761 - 7.2.3
#Prescription level: Mandatory
#Validation type: Positive
#Target: Element <region left="?"/>.
#Normative Statement:
Attribute values may be non-negative "percentage" values, or integer pixel units. For pixel values, the author may omit the "px" unit qualifier (e.g. "100"). For percentage values, on the other hand, the "%" symbol shall be indicated (e.g. "50%"). The percentage is always relative to the parent's width, in the case of right, left and width definitions, and parent's height, in the case of bottom, top and height definitions.
#Instruction Name: region07.01
#Instruction:
Create a document containing a <region> element with its left attribute set to a valid value expressed in pixels. The region must be rendered in the position defined by the attribute value.
#NCL input documents: URI to the file of the test case 01; URI to the file of the test case 02, etc.

CASO 3b
***Idêntico ao Caso 3a, mudando apenas a parte abaixo***

#Instruction Name: region07.02
#Instruction:
Create a document containing a <region> element with its left attribute set to a valid value expressed in percentage of the parent region, which also has its positioning attributes expressed in percentage. The region must be rendered in the position defined by the attribute value. At least one test must be made having the full screen of a device as a parent region and another having another region as parent.
#NCL input documents: URI to the file of the test case 01; URI to the file of the test case 02, etc.

É importante ainda salientar que os casos de teste devem verificar não somente as entradas válidas para execução, como também as entradas inválidas. Para testar uma unidade, deve-se obter casos de teste suficientes para verificar se:

- para cada entrada válida, um comportamento apropriado foi retornado pela operação;
- para cada entrada não válida, somente um comportamento apropriado foi retornado pela operação;

## 4.1 Exemplos de Entradas de Casos de Testes

Para exemplificar o formato dos dados de entrada de casos de teste, retomemos o CASO 2, mencionado nesta Seção 4.

Um documento de entrada, gerando um caso de teste, é apresentado na Listagem 1, para o CASO2a. Note que o identificador da aplicação NCL é o identificador do caso de teste. Note também o comentário explicitando o comportamento esperado de saída, e o elemento <meta> identificando o autor do caso de teste.

```
<ncl id="media01.01.01"
       xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
<!-- Display the jpeg image, loaded from the local file system, on
     the whole device screen, as specified by the <property> el-
     ements -->
  <head>
     <meta name="author" content="TeleMidia PUC-Rio"/>
  </head>
  <body>
            <port id="port" component="media"/>
            <media id="media01" src="images/image01.jpg">
                        <property name="left" value="0%"/>
                        <property name="right" value="0%"/>
                        <property name="height" value="100%"/>
                        <property name="width" value="100%"/>
            </media>
  </body>
</ncl>
```

**Listagem 1. Caso de teste media01.01.01.**

Um caso de testes semelhante é apresentado na Listagem 2 para o CASO2b, mas agora para uma figura carregada através do protocolo HTTP.

```
<ncl id="media01.02.01"
       xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
<!-- Display the jpeg image, loaded from the remote file system
     by using HTTP protocol, on the whole device screen, as
     specified by the <property> elements -->
  <head>
     <meta name="author" content="Eduardo Cruz"/>
  </head>
  <body>
            <port id="port" component="media"/>
     <media id="media01"
          src="http://testsuite.gingancl.org/monomedias/image
          s/image01.jpg">
                        <property name="left" value="0%"/>
                        <property name="right" value="0%"/>
                        <property name="height" value="100%"/>
                        <property name="width" value="100%"/>
            </media>
  </body>
</ncl>
```

**Listagem 2. Caso de teste media01.02.01.**

Note agora um outro caso de teste para o mesmo CASO2b. A diferença está na interdependência de outros elementos NCL.

```
<ncl id="media01.02.02"
       xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
<!-- Display the jpeg image, loaded from the remote file system
     by using HTTP protocol, on the whole device screen, as
     specified by the <property> elements -->
  <head>
     <meta name="author" content="Vinicius Lago"/>
           <regionBase>
        <region id="r01" left="0%" top="0%" width="100%"
                                                 height="100%"/>
           </regionBase>
                    <descriptorBase>
        <descriptor id="d01" region="r01"/>
           </descriptorBase>
            </head>
  <body>
                    <port id="port" component="media"/>
     <media   id="media01"   src="http://testsuite.ginganc l.org/
       monomedias/images/image01.jpg" descriptor="d01"/>
  </body>
</ncl>
```

**Listagem 3. Caso de teste media01.02.02.**

Comparando as Listagens 2 e 3, note que o caso da Listagem 2 tem dependência dos elementos <property> além de outros elementos comuns aos dois casos; ao passo que a Listagem 3 traz a dependência dos elementos <region> e <descriptor> além dos elementos comuns. Assim, pela definição da Seção 4, a Listagem 2 representa o caso de teste integrado minimalista.

## 5 Ambientes de teste

Um ponto importante quando se fala na construção de uma suíte de testes é seu custo. Para a análise do custo, deve-se levar em conta: o esforço requerido para o desenho dos testes, ou seja, para a construção dos casos de testes; a aplicação dos casos de teste; e a análise dos resultados. Esses dois últimos pontos tornam-se menos preponderantes se a suíte for aplicada várias vezes.

É possível definir duas formas diferentes para aplicação dos testes e análise dos resultados. A primeira e mais simples, é a execução manual. Nessa, os testes são executados um a um, por um ser humano, chamado, nesse cenário, de testador. Nesse cenário, é importante que a descrição dos resultados do caso de teste seja feita de forma bem clara para que qualquer pessoa possa assumir o papel do testador, sem que seja necessário que ela conheça a implementação do caso de teste e a especificação do software, ou seja, a forma como ele processa.

A segunda forma é a execução automática. Nessa, os testes são realizados e o processamento dos resultados é feito de forma automática, sem a necessidade da presença de um testador. Nesse cenário, é necessário que seja desenvolvida uma infraestrutura para a execução dos testes e para a comparação dos resultados obtidos com os resultados esperados.

Para a suíte de testes de conformidade do Ginga-NCL, proposta neste documento, foram pensados três ambientes diferentes de execução de testes: manual, semiautomático e automático.

No ambiente manual todos os casos de testes são executados de forma manual, por um testador, que é responsável por aplicar cada um dos casos de teste e observar sua execução, verificando se o resultado obtido está de acordo com a descrição dos resultados esperados do caso de teste. Nesse ambiente, a construção de cada caso de teste tem um custo baixo, já que não é necessária a especificação de nenhum ponto além dos já definidos na Seção 4 para cada caso de teste. O custo para a "construção" do ambiente de testes é zero, já que não é necessário nada além dos casos de testes e do dispositivo a ser testado. Entretanto, a aplicação dos testes e a análise dos resultados é bastante custosa, levando-se em consideração o tempo para a execução de todos os testes de forma manual, e o custo de um testador, para fazer a análise dos resultados. Não deve ser desprezado, também, que testes manuais podem ter uma baixa confiabilidade, já que depende de um olhar humano para obtenção dos resultados.

No ambiente semiautomático alguns dos casos de testes são executados de forma automática (cerca de 50% dos casos para o Ginga-NCL são passíveis de semiautomatização), e o restante executado de forma manual. A automatização nesse ambiente é uma automatização por transições de estados, como descrito a seguir.

Em NCL, as relações são baseadas em eventos. Um evento é uma ocorrência no tempo que pode ser instantânea ou ter uma duração mensurável. Quatro tipos de eventos são definidos: evento de apresentação (definidos sobre de objetos de mídia − <media> − ou sobre objetos de composição − <body>, <context> ou <switch>), evento de seleção (definidos sobre um objeto de mídia − <media>), evento de atribuição (correspondendo a uma atribuição de valor a uma propriedade de um objeto − <media>, <body>, <context> ou <switch>) e evento de composição (definido pela apresentação da estrutura de um objeto de composição − <body>, <context> ou <switch>). A partir de relacionamentos entre eventos (representados em NCL pelos elementos <link>), os eventos de objetos podem ser detectados de forma automática, fazendo com que o estado do objeto, sob teste, possa ser comparado com seu estado esperado em um determinado momento. A automatização por transição de estados tem esse pressuposto por base.

Infelizmente, nem todos os casos de testes podem ser testados de forma semiautomática, por meio de transição de estados. Em geral, essa forma é válida somente para os casos de teste que têm seu foco no estado de um componente (<media>, <contexto> ou <switch>) em um dado momento da execução da aplicação. Testes dos elementos <conector>, <link>, <port> e <context> são exemplos onde essa forma de teste pode ser empregada.

Para a execução semiautomática dos testes, foi desenvolvida uma infraestrutura (um núcleo) que utiliza a própria NCL, com sua linguagem de script LUA [IERUSALIMSCHY06]. A utilização da própria NCL é necessária, já que o Ginga-NCL pode ser colocado em diferentes plataformas de hardware e sistemas operacionais, sendo, portanto, o middleware a única coisa que podemos afirmar que estará disponível para ser utilizada em todas elas. Pode-se assim dizer que tem-se o Ginga-NCL reflexivamente usado para seu próprio teste de conformidade.

O núcleo consiste, de forma simplificada, de uma aplicação NCL responsável por inicializar e parar todas as implementações dos casos de testes, e também responsável por verificar o estado dos componentes presentes em cada caso de teste, anotando em uma ata o estado desses componentes e os tempos em que ocorreram mudanças entre esses estados.

Como para a construção do núcleo descrito acima é utilizada a linguagem NCL, a própria aplicação NCL é também um caso de teste integrado, caracterizando a reflexi-

bilidade da proposta. Mais do que isto, esse caso de teste deve ser considerado como um teste de integração, pois deve-se assumir que todos os elementos e atributos (unidades) envolvidos já tenham sido testados anteriormente (possivelmente manualmente). Sendo assim, a execução do núcleo não comprometerá a execução dos casos de testes. Dessa forma, é necessário que seja testada, antes da execução semiautomática, uma sequência que testes manuais que garantam a conformidade de todos os elementos e atributos utilizados pelo núcleo para a automatização. Se essa sequencia de testes não produzir um resultado positivo, ou seja, se algo não estiver conforme a norma, os testes executados semiautomaticamente provavelmente darão o mesmo resultado.

No ambiente semiautomatizado, o custo do esforço de desenho dos casos de testes que podem ser automatizados é maior do que no ambiente manual, tendo em vista que é necessário que o criador dos casos de teste gere *também* a descrição do comportamento esperado de saída de acordo com um modelo que possa ser processado pelo núcleo de automatização. O custo da aplicação e da análise dos resultados dos casos de testes que são feitos de forma automática são bem pequenos, se não computarmos o custo da construção do núcleo de automatização (que é dividido entre as várias aplicações da sequência de testes). A confiabilidade dos resultados obtidos é maior que a dos testes feitos manualmente, já que todo o processamento dos resultados é feito pela máquina.

Finalmente, no ambiente totalmente automático, todos os casos de testes podem ser executados de forma automatizada. Para tanto esses ambientes exigem uma infraestrutura grande (e muitas vezes tecnologicamente ainda inexistente, como, por exemplo, o reconhecimento de imagens em vídeo) no ambiente para execução dos testes. Para ser possível que todos os casos de testes tenham seus resultados processados automaticamente, é necessário uma infraestrutura que faça a captura de imagens, sons e vídeos gerados pela aplicação dos testes, e que realize um procedimento de comparação os dados obtidos com as imagens, vídeos e sons que representam os resultados esperados dos casos de teste. Também é necessário um dispositivo que simule a interação do usuário com o middleware. Nesse ambiente, o custo para o esforço do desenho dos casos de teste é alto, uma vez que é necessário que o desenvolvedor do caso de teste "construa" as imagens, vídeos e sons que servirão como "modelo" do resultado da aplicação dos testes. O custo do desenvolvimento da infraestrutura de execução e processamento dos resultados também é bastante alto, levando em consideração a necessidade de plataformas de hardware específicas e algoritmos eficientes para o processamento de imagens, vídeos e áudios. Porém o custo da aplicação dos casos de teste e análise dos resultados é baixo e mais confiável que os manuais.

Um ambiente que automatize a execução dos testes é importante, já que possibilita menos chances de falhas humanas. Entretanto, a execução dos testes de forma semiautomatizada ou automatizada deve ser opcional, no caso da suíte de teste de conformidade para o Ginga-NCL. Para a aplicação semiautomática dos casos de teste anteriormente descritos, um núcleo foi desenvolvido e padronizado. No entanto, sua utilização exige que a plataforma sob teste tenha recursos que muitas vezes não são encontrados nos receptores de baixo custo para TV terrestre e IPTV. A Seção 5.1 descreve em mais detalhes a forma de semiautomatização proposta nesta monografia. Já a aplicação totalmente automatizada exige custos muito elevados de implementação. A aplicação de testes manuais, apesar de seus problemas, determinou, assim, um foco obrigatório na especificação da suíte de testes para o Ginga-NCL.

## 5.1 Semiautomatização

Como já discutido anteriormente, dois pontos são importantes quando se fala em testes de software: custo (de construção e execução dos casos de testes e análise dos resultados) e confiabilidade dos resultados.

A execução semiautomática dos testes do Ginga-NCL foi uma maneira encontrada para diminuir os custos e aumentar a confiabilidade da construção e execução dos casos de teste utilizando uma infraestrutura simples que utiliza a própria linguagem NCL com sua linguagem de script LUA [IERUSALIMSCHY06] para a automatização, não sendo necessário que haja um ambiente montado especialmente para a execução dos testes.

Uma infraestrutura composta por um núcleo de automatização, uma base de conectores e regras para a construção dos casos de testes foi criada para possibilitar a execução semiautomática dos casos de teste.

A seguir, cada um dos itens que compõe a infraestrutura de semiautomatização será detalhado, entretanto, nos primeiros parágrafos alguns conceitos importantes sobre a linguagem NCL serão discutidos para que a infraestrutura desenvolvida possa ser compreendida plenamente.

Uma aplicação NCL pode possuir vários objetos de mídia. Esses objetos de mídia podem ter conteúdos de tipos diferentes: texto, imagem, áudio, vídeo, etc. Entre os tipos de conteúdo de mídia permitidos em uma aplicação NCL, dois deles merecem destaque por serem à base do núcleo de automatização da suíte de testes do Ginga-NCL: os objetos com códigos imperativos, em especial código LUA e os objetos com códigos declarativos, em especial código NCL.

Objetos hipermídia declarativos podem definir âncoras de conteúdo (através do elemento <area>) e propriedades (através dos elementos <property>), assim como qualquer outro objeto de mídia NCL. Entretanto, âncoras de conteúdo de objetos com código NCL possuem uma função especial, elas externalizam conteúdos da aplicação NCL embutida. Para a externalização dos componentes de uma aplicação é necessário que a própria aplicação ofereça um ponto de acesso (uma interface) para os seus componentes. Isso pode ser feito com o elemento <port> que oferece acesso externo ao conteúdo da aplicação.

Assim, quando se tem uma aplicação NCL com um objeto NCL embutido (objeto de media NCL) e deseja-se ter acesso aos componentes internos do NCL embutido, deve-se ter uma âncora de conteúdo ligada a cada uma das interfaces do objeto NCL. Vejamos os exemplos a seguir:

NCL embutido (embutido.ncl):

```
<body>
    <port id="interface1" component="media1" />

    <media id="media1" src="imagem.png">
          <property name="bounds" value="0,0,100%, 100%"/>
    </media>
    <media id="media2 src="video.mp4">
          <property name="bounds" value="0,0,100%, 100%"/>
    </media>
</body>
```

Neste exemplo somente a imagem (media1) possui uma interface e por isso só ela pode ser acessada externamente.

NCL principal:

```
<body>
```

```
    <port id="port" component="mediaNCL" />

    <media id="mediaNCL" src="embutido.ncl">
          <area id=”ancora” label=”interface1”>
          <property name="bounds" value="0,0,100%, 100%"/>
    </media>

</body>
```

O NCL principal através do elemento <area> com o atributo *label* preenchido com o *id* do elemento <port> que externaliza a media1 do NCL embutido, consegue agora acessá-la e verificar seu estado.

Na suíte de testes cada caso de teste é uma aplicação NCL. A automatização é feita comparando os estados dos componentes da aplicação NCL de testes com os estados esperados. Para isso, é necessário que todos os casos de teste possuam um ponto de interface (elemento <port>) para cada um dos seus componentes a serem testados.

A aplicação Main (o núcleo) é uma aplicação NCL responsável por iniciar e parar todos os casos de teste e verificar os estados dos componentes de cada um deles. Esta aplicação é construída automaticamente a partir de um arquivo de configuração que é passado como entrada e contém o nome de cada caso de teste, o caminho e a duração máxima de cada teste. Este último parâmetro deve ser informado pelo autor do caso de teste, e é importante para que o núcleo possa forçar o término de um dos testes sem interferir no seu resultado.

A aplicação Main possui então n objetos de mídia NCL que embutem as aplicações NCL de caso de teste. Cada um desses objetos possui uma âncora de conteúdo (elemento <area>) externalizando as interfaces abertas em cada caso de teste. Além disso esse núcleo possui um objeto de mídia LUA (chamado de controle).

O controle é responsável por sinalizar o momento em que um caso de teste (objeto de mídia NCL) deve ser inicializado (realizar ação de *start*) e sinalizar o momento que o caso de teste deve ser encerrado (realizar ação de *stop*) caso não tenha encerrado naturalmente. Este controle é feito com um *timer* para cada caso de teste a partir da duração máxima previamente informada. O final forçado de um caso de teste é necessário, já que nem todos os casos de teste terminam naturalmente, seja por vontade do autor do teste ou por uma não conformidade no *middleware*, fazendo com que a execução da aplicação Main fique “presa” no caso doe teste em questão, e por isso é necessário que todos os casos de teste terminem.

O controle também é responsável por gerar uma ata de execução de cada caso de teste, ou seja, ele recebe uma sinalização dos eventos ocorridos nos componentes de cada caso de teste e escreve em um arquivo esses eventos. A ata gerada é comparada com os roteiros de execução dos casos de teste feitos pelos autores e com isso são gerados os resultado finais dos testes.

A sinalização para início e término dos casos de teste e dos eventos ocorridos nos componentes é feita através de elos (elemento <link>). No primeiro caso, o script de controle faz uso do *timer*, já citado anteriormente, para mandar o sinal para que os eventos de *start* e *stop* sejam disparados pelos elos sobre os casos de teste. Esse sinal é um evento de apresentação feito pelo script LUA em uma âncora específica do objeto de mídia controle. No segundo caso, os elos utilizam uma base de conectores específica, que faz parte da infraestrutura de semiautomatização. Os links esperam os eventos de *stop*, *start*, *abort*, *pause*, *resume*, *selection*, *beginAttribution* e *endAttribution* de cada um dos componentes externalizados pelo elemento <area> e quando eles acontecem, esses <links> disparam uma ação de set sobre uma propriedade do script LUA controle passando o nome do componente que mudou de estado. Quando o controle recebe a

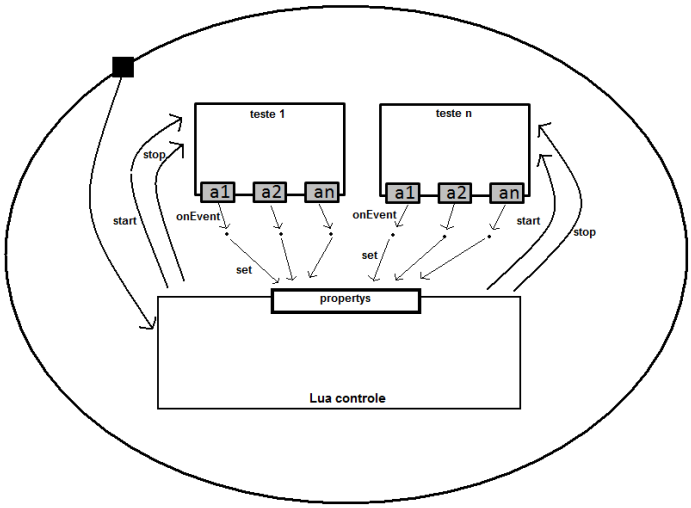
ação, ele escreve no log o nome do caso de teste, o nome do componente e o tempo (em relação ao início do caso de teste) que a ação ocorreu.

Abaixo podemos ver um exemplo de conector e um exemplo de um link que utiliza esse conector para esperar uma ação de start e sinalizá-la para o controle.

```
<causalConnector id="startTest">
    <connectorParam name="id" type="String"/>
    <simpleCondition role="onBegin"/>
    <simpleAction role="set" value="$id" max="2" min="1"/>
</causalConnector>
```

```
<link xconnector="con#startTest">
    <bind role="onBegin" component="v001" interface="av001_1"/>
    <bind role="set" component="luaControlador1"
interface="transaction">
          <bindParam name="id" value="start"/>
    </bind>
    <bind role="set" component="luaControlador1"
interface="idComponent">
          <bindParam name="id" value="av001_1"/>
    </bind>
</link>
```

A figura a baixo sintetiza o funcionamento do núcleo.




**Núcleo para a semiautomatização**

Para que o núcleo funcione adequadamente, é necessário que o autor dos testes siga um padrão para a nomenclatura das interfaces dos componentes de cada teste. Cada interface (elemento <port>) possui um atributo *id*. Para cada caso de teste o *id* das interfaces deve ser o nome do caso de teste seguido do caracter "_", seguido de um número, começando de 1 e indo até o número máximo de elementos <port>. É importante que o elemento com número 1 seja a interface de entrada do caso de teste. Esse padrão é importante para que o núcleo funcione corretamente. Por exemplo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="area06.01.01" xlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
      .
      .
      .

   <body>
      <port id="area06.01.01_1" component="media1"/>
```

```
      <port id="area06.01.01_2" component="media1" interface="area1"/>
      <media id="media1" src="../monomidias/Imagens/azul.jpg">
            <area id="area1" end="3s"/>
            <property name="bounds" value="0,0,100%,100%"/>
            <property name="explicitDur" value="6s"/>
      </media>
      <media id="media2" src="../monomidias/Imagens/vermelho.jpg">
            <property name="bounds" value="0,0,100%,100%"/>
            <property name="explicitDur" value="3s"/>
      </media>
      <link xconnector="onEndStart">
.
.
.
   </body>
</ncl>
```

Além disso, o autor dos testes deve fornecer junto com os casos de teste um arquivo com o roteiro de execução da aplicação. Ao final da execução dos casos de teste, um outro script LUA é executado e esse compara a execução do teste, ou seja, a ata de execução de cada caso de teste gerado pelo controle, com a execução esperada, fornecida pelo autor do teste. Uma ata final é gerada com o resultado de todos os testes que foram executados pelo núcleo.

## 6 Conclusões

A importância de uma suíte de testes para o Ginga-NCL é claramente evidenciada pelo enorme mercado internacional que adotou essa solução como middleware para TV digital terrestre e IPTV. Somente com testes de conformidade poder-se-á garantir uma implementação interoperável e que não deixará legados.

Em geral, as técnicas usadas para definição de testes de conformidade não se adequam à verificação de sistemas de middleware para ambientes declarativos XML. Uma adaptação se faz necessária, conforme proposto neste artigo.

Outros trabalhos já evidenciaram a necessidade da adequação das técnicas de testes desenvolvidas, em geral, para verificação de unidades imperativas. Este artigo, no entanto, tenta deixar claros os conceitos utilizados e os estende tanto com relação à especificação dos casos de teste quanto a suas aplicações.

Como discutido ao longo do texto, o conjunto de entradas para teste de elemento ou atributo alvo de uma linguagem XML pode ser muito grande, ou mesmo infinito. Portanto, é importante que uma suíte de testes para esse domínio seja extensível, e que seja permitido que outras pessoas possam contribuir com mais casos de testes. Esse é o caso da suíte proposta para o Ginga-NCL.

A suíte encontra-se disponível em http://testsuite.gingancl.org.br e qualquer pessoa física ou jurídica pode contribuir com novos casos de teste. Para tanto, basta seguir as normas ITU para contribuição, que apenas exigem que o formato de casos de teste, conforme discutido na Seção 4, seja seguido. Caso o autor do teste queira que seu teste seja promovido a teste integrado minimalista, ele deve fazer um pedido formal ao ITU-T, mostrando o porquê de sua requisição.

Embora a especificação da suíte de testes de conformidade para o Ginga-NCL atenda a todas as assertivas para os elementos e atributos NCL, ainda restam pelo menos três passos a cumprir:

1. uma suíte de testes para a API NCLua [ABNT NBR 15606-7, 2011 e [IEEE Std 829,1998]
2. uma suíte de testes para os comandos de edição NCL [SRCM06].
3. uma suíte com streams de teste de sistema.

O primeiro trabalho deve envolver uma solução muito semelhante àquela abordada em [ECMAScript]. Para os segundo e terceiro trabalhos a solução deve seguir a abordagem proposta neste artigo.

A grande maioria de suíte de testes para exibidores de linguagens baseadas em XML são para aplicação manual, ou seja, com a aplicação e verificação dos resultados feita pelo ser humano. A Seção 5 apresenta como sequências de teste podem ter sua aplicação semiautomatizada e automatizada. Muito trabalho nessa direção ainda resta ser feito. Por exemplo, pesquisadores do LIFIA estão desenvolvendo sequências automatizadas cuja verificação dos resultados é realizada pelo computador [19], pela comparação entre os comportamentos esperados e aqueles obtidos por captura das telas após a aplicação dos testes. Note que essa técnica exige reconhecimento de imagens, mas em casos bem simples de teste, é factível e de simples realização.

## Agradecimentos

Os autores gostariam de agradecer a toda a equipe atual do Laboratório TeleMídia, que tem proporcionado uma discussão profunda desse trabalho, baseada na aplicação dos casos de teste desenvolvidos na implementação de referência do ambiente Ginga-NCL.

## Referências

[ABNT NBR 15606-2, 2007] ABNT NBR 15606-2, 2007. Digital Terrestrial Television - Data Coding and Transmission Specification for Digital Broadcasting - Part 2: Ginga-NCL for fixed and mobile receivers - XML application language for application coding. Available: http://www.dtv.org.br/download/en-en/ABNTNBR15606_2D2_2007Ing_2008Vc2_2009.pdf

[ITU-R Recommendation BT-1699, 2009] ITU-R Recommendation BT-1699, 2009. Harmonization of declarative content format for interactive TV applications. Geneva, 2009.

[ITU-T Recommendation H.761, 2009] ITU-T Recommendation H.761, 2009. Nested Context Language (NCL) and Ginga-NCL for IPTV Services. Geneva, April, 2009.

[MYERS04] MYERS, G. J. The Art of Software Testing. John Wiley & Sons, Inc. 2a. Edition. 2004. ISBN 0-471-46912-2

[ABNT NBR 15606-7, 2011] ABNT NBR 15606-7, 2011. Televisão digital terrestre — Codificação de dados e especificações de transmissão para radiofusão digital Parte 7: Ginga-NCL – Diretrizes operacionais para as ABNT NBR 15606-2 e ABNT NBR 15606-5. Available: http://www.dtv.org.br/download/pt-br/ABNTNBR15606-7_2011Ed1.pdf

[IEEE Std 829,1998] IEEE Std 829. IEEE Standard for Software Test Documentation. 1998. ISBN 0-7381-1443-X

[IERUSALIMSCHY06] IERUSALIMSCHY,R. "Programming in Lua", Second Edition, Copyrighted Material, 2006.

[SSC08] SOARES L.F.G., SANT'ANNA F.F., CERQUEIRA R. F. G. 2008. Nested Context Language 3.0 Part 10 – Imperative Objects in NCL: The NCLua Scripting Language. Monografias em Ciência da Computação do Departamento de Informática da PUC-Rio, MCC 02/08. Rio de Janeiro. Janeiro de 2008. ISSN 0103-9741.

[W3C XML] Extensible Markup Language (XML) Conformance Test Suites – Disponível em <http://www.w3.org/XML/Test/>

[HTML4] HTML4 Test Suite - Disponível em <http://www.w3.org/MarkUp/Test/HTML401/current/>

[ECMAScript] ECMAScript test262 - Disponível em <http://test262.ecmascript.org/http://test262.ecmascript.org/>

[SVG] Scalable Vector Graphics (SVG) 1.1 (Second Edition) - Disponível em <http://www.w3.org/TR/SVG/>

[W3C SVG] W3C Scalable Vector Graphics (SVG) 1.1 Test Suite - Disponível em <http://www.w3.org/Graphics/SVG/Test/20061213/>

[BDR09] BULTERMAN, DICK C.A., RUTLEDGE, L.W.: SMIL 3.0 - Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books. 2nd ed. Springer, 2009. ISBN: 978-3-540-78546-0

[SMIL3.0] SMIL 3.0 Test Suite - Disponível em <http://www.w3.org/2007/SMIL30/testsuite/>

[SRCM06] SOARES L.F.G., RODRIGUES R.F., COSTA R.R., MORENO M.F.2006. Nested Context Language 3.0 Part 9 – NCL Live Editing Commands. Monografias em Ciência da Computação do Departamento de Informática, PUC-Rio, No. 36/06. Rio de Janeiro. Dezembro de 2006. ISSN 0103-9741.