PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 17/09

Nested Context Language 3.0

Aplicações Declarativas NCL com Objetos NCLua Imperativos Embutidos

Francisco Sant'Anna
Carlos de Salles Soares Neto
Simone Diniz Junqueira Barbosa
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. 17/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Junho/2009

Nested Context Language 3.0

Aplicações Declarativas NCL com Objetos NCLua Imperativos Embutidos

Francisco Sant'Anna
Carlos de Salles Soares Neto
Simone Diniz Junqueira Barbosa
Luiz Fernando Gomes Soares

Laboratório TeleMídia DI – PUC-Rio
Rua Marquês de São Vicente, 225, Rio de Janeiro, RJ - 22451-900.

{francisco, csalles}@telemidia.puc-rio.br, {simone, lfgs}@inf.puc-rio.br

Abstract. *The Nested Context Language (NCL) allows the development of multimedia applications with spatio-temporal synchronization among media objects, such as videos, sounds and images. In order to extend NCL with a general purpose programming language, objects written with the Lua language, the so called NCLua objects, are also supported. For common media objects, it is the role of the media player to interpret the actions commanded by the NCL formatter, such as start and abortion of media presentations. However, for NCLua objects, this role is attributed to the object author, which can give any semantics to the commanded actions. An NCLua object communicates with the NCL document in which it is embedded through events, in accordance with the transitions in its state machine. This technical report describes how NCLua objects are developed.*

Keywords: *Ginga-NCL, Middleware, DTV, NCL, SBTVD-T.*

Resumo. *A Linguagem de Contextos Aninhados (Nested Context Language-NCL) permite desenvolver aplicações multimídia com sincronismo espaço-temporal entre objetos de mídia, tais como vídeos, sons e imagens. De maneira a estender NCL com uma linguagem de programação de propósito geral, também são suportados objetos cujo conteúdo é composto por código escrito na linguagem Lua, conhecidos como objetos NCLua. Para objetos de mídia comuns, é dever do exibidor da mídia interpretar as ações comandadas pelo formatador NCL, tais como o início e interrompimento da apresentação da mídia. No caso de um NCLua, essa tarefa é atribuída ao autor do objeto, que pode dar uma semântica qualquer para as ações do formatador. Um objeto NCLua se comunica com o documento NCL no qual está inserido através de eventos, de acordo com as transições em sua máquina de estados. Este relatório técnico descreve como são desenvolvidos objetos NCLua embutidos em aplicações NCL. boas práticas de desenvolvimento de aplicações.*

Palavras chave: *Ginga-NCL, Middleware, DTV, NCL, SBTVD-T.*

Nested Context Language 3.0
Aplicações Declarativas NCL com Objetos NCLua Imperativos Embutidos

Laboratório TeleMídia

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rua Marquês de São Vicente, 225, Prédio ITS - Gávea

22451-900 – Rio de Janeiro – RJ – Brasil

http://www.telemidia.puc-rio.br

Sumário

Nested Context Language 3.0
Aplicações Declarativas NCL com
Objetos NCLua Imperativos Embutidos

Francisco Sant'Anna
Carlos de Salles Soares Neto
Simone Diniz Junqueira Barbosa
Luiz Fernando Gomes Soares

Laboratório TeleMídia DI – PUC-Rio
Rua Marquês de São Vicente, 225, Rio de Janeiro, RJ - 22451-900.

{francisco, csalles}@telemidia.puc-rio.br, {simone, lfgs }@inf.puc-rio.br

Resumo. *A Linguagem de Contextos Aninhados (Nested Context Language-NCL) permite desenvolver aplicações multimídia com sincronismo espaço-temporal entre objetos de mídia, tais como vídeos, sons e imagens. De maneira a estender NCL com uma linguagem de programação de propósito geral, também são suportados objetos cujo conteúdo é composto por código escrito na linguagem Lua, conhecidos como objetos NCLua. Para objetos de mídia comuns, é dever do exibidor da mídia interpretar as ações comandadas pelo formatador NCL, tais como o início e interrompimento da apresentação da mídia. No caso de um NCLua, essa tarefa é atribuída ao autor do objeto, que pode dar uma semântica qualquer para as ações do formatador. Um objeto NCLua se comunica com o documento NCL no qual está inserido através de eventos, de acordo com as transições em sua máquina de estados. Este relatório técnico descreve como são desenvolvidos objetos NCLua embutidos em aplicações NCL. boas práticas de desenvolvimento de aplicações.*

1 Introdução

Desde o início de seu desenvolvimento, no início dos anos 90, Lua foi projetada para ser usada em conjunto com outras linguagens, não sendo comum encontrar programas escritos puramente em Lua. Nesse sentido, Lua é normalmente usada para permitir que uma aplicação principal seja estendida ou adaptada através do uso de scripts. A aplicação principal pode ser um video game, onde um script Lua é usado para definir o comportamento de um personagem; um editor de textos, que permite que os textos sejam acessados e modificados por scripts Lua; ou, de maneira mais geral, aplicações que usam Lua em scripts de configuração. Esse tipo de uso caracteriza Lua como uma linguagem de scripts no seu sentido mais puro. O próprio nome da linguagem, Lua, remete à idéia de uma linguagem satélite.

Lua é uma linguagem de fácil aprendizado, que combina sintaxe procedural com declarativa, com poucos comandos primitivos. Dessa característica resulta uma implementação leve e muito eficiente quando comparada com linguagens de propósitos similares. Lua também apresenta um alto grau de portabilidade, podendo ser executada com

todas as suas funcionalidades em diversas plataformas, tais como computadores pessoais, celulares, sistemas embarcados e consoles de video games.

As características de Lua — simplicidade, eficiência e portabilidade—, além de sua licença livre, casam perfeitamente com o cenário de TV Digital. Uma linguagem simples é bem-vinda onde é comum equipes formadas não só por programadores, mas também por designers e produtores de conteúdo. A portabilidade é importante quando o middleware deve ser desenvolvido para dispositivos com características conflitantes tais como celulares e set-top boxes. A eficiência e tamanho da linguagem requerem menos custos com hardware. Já a licença livre de royalties reduz a custo zero a adoção do interpretador por unidade produzida. Um indicador de como a linguagem Lua se adapta bem a esse tipo de cenário é a liderança de Lua como linguagem de script em video games, nicho que compartilha as mesmas características descritas.

Uma apresentação mais detalhada da sintaxe e funcionalidades de Lua fica fora do escopo desta monografia. Explicaremos os conceitos da linguagem necessários conforme forem utilizados nos exemplos desta monografia.

A monografia está organizada como se segue. A Seção 2 discute o paradigma de programação orientada a eventos. Na Seção 3 são usados exemplos para detalhar como é feita a interação de código Lua com documentos NCL. Na Seção 4 é apresentada a interface de programação para o desenho na região do objeto NCLua. A Seção 5, por sua vez, ilustra também por exemplos o reúso de código Lua em objetos NCLua.

1.1 Extensões de NCLua

Para se adequar ao ambiente de TV Digital e se integrar à NCL, a linguagem Lua foi estendida com novas funcionalidades. Por exemplo, um NCLua precisa se comunicar com o documento NCL para saber quando o seu objeto <media> correspondente é iniciado por um elo. Um NCLua também pode responder a teclas do controle remoto, ou desenhar livremente dentro da região NCL a ele destinada. Essas funcionalidades são específicas da linguagem NCL e, obviamente, não fazem parte da biblioteca padrão de Lua. O que diferencia um NCLua de um programa Lua puro é o fato de ser controlado pelo documento NCL no qual está inserido, e utilizar as extensões descritas a seguir.

Além da biblioteca padrão de Lua, os seguintes módulos estão disponíveis para scripts NCLua:

- Módulo event: permite que objetos NCLua se comuniquem com o documento NCL e outras entidades externas (tais como controle remoto e canal de interatividade).
- Módulo canvas: oferece funcionalidades para desenhar objetos gráficos na região do NCLua.
- Módulo settings: oferece acesso às variáveis definidas no objeto settings do documento NCL (objeto do tipo “application/x-ncl-settings”).

- Módulo persistent: exporta uma tabela com variáveis persistentes entre execuções de objetos imperativos.

A norma ABNT NBR 15606-2:2007 [1] e H.761[4] lista detalhadamente todas as funções suportadas por cada módulo.

As seguintes funções da biblioteca padrão de Lua são dependentes de plataforma e por isso não estão disponíveis para scripts NCLua:

- No módulo package: a função loadlib.
- No módulo io: todas as funções.
- No módulo os: as funções clock, execute, exit, getenv, remove, rename, tmpname e setlocale.
- No módulo debug: todas as funções.

2 Programação Orientada a Eventos

O Middleware Ginga possui um modelo de execução e comunicação de objetos imperativos embutidos em documentos NCL. No caso de objetos NCLua, os mecanismos de integração com o documento NCL se fazem através do paradigma de programação orientada a eventos.

Não somente a comunicação com o documento NCL, mas também qualquer interação com entidades externas à aplicação, tais como o canal de interatividade, controle remoto e temporizadores, se faz pela difusão e recepção de eventos. O módulo `event` de NCLua, usado para esse fim, é a mais importante extensão à linguagem Lua, e seu entendimento é essencial para desenvolver qualquer aplicação que utilize objetos NCLua.

A Figura 1 exibe ao centro um NCLua, envolto por diversas entidades com as quais ele pode interagir. Para se comunicar com um NCLua, uma entidade externa deve inserir um evento na fila indicada na figura, que é então redirecionado às funções tratadoras de eventos, definidas pelo programador do script NCLua. Enquanto cada tratador, um de cada vez, processa um evento, nenhum outro evento da fila é tratado. Sendo assim, fica a cargo do programador escrever tratadores que executem o mais rápido possível, de maneira a evitar o congestionamento da fila. Um NCLua também pode se comunicar com entidades externas postando eventos dentro de seus tratadores, como mostram as setas saindo do NCLua.[1]

[1] A fila de eventos é controlada pelo sistema e não é visível a um NCLua.

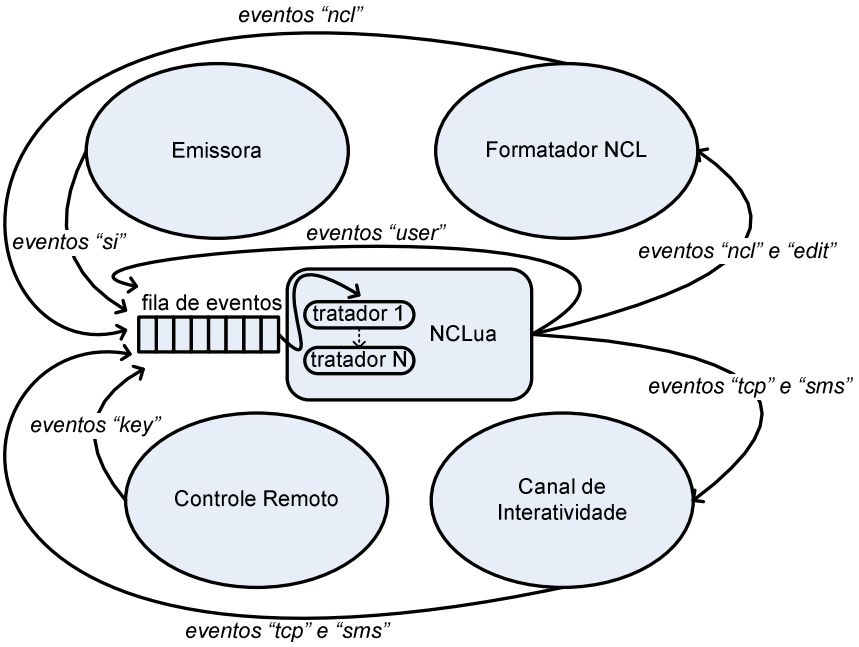

Figura 1: Paradigma de programação orientado a eventos.

A principal vantagem do uso do paradigma de eventos é a sua característica de acoplamento fraco entre as entidades do sistema. Como se pode observar pela figura, a remoção ou adição de uma entidade não acarreta mudanças internas em nenhuma outra entidade. Essa característica vai ao encontro dos requisitos de mínima intrusão do ambiente de autoria.

Para ser informado quando eventos externos são recebidos, um NCLua deve registrar pelo menos uma função de tratamento em seu corpo através de uma chamada à função event.register (o nome da função a ser registrada é qualquer). O código de um NCLua segue uma estrutura comum a todos os scripts, como a seguir:

```
...                          -- código de inicialização
function tratador (evt)
   ...                       -- código de um tratador
end
event.register(tratador)     -- registro de pelo menos um tratador
```

O código de inicialização, a definição do tratador e seu registro são executados antes que o documento NCL (ou qualquer entidade externa ao script) sinalize qualquer evento ao NCLua, inclusive o de início de apresentação do objeto. Após esse processo de carga do script, efetuado pelo sistema, apenas o código do tratador é chamado, toda vez que ocorre um evento externo. O código de inicialização pode ser utilizado para criar objetos e funções auxiliares que serão usadas pelos tratadores.

Eventos são representados por tabelas Lua com chaves e valores descrevendo seus atributos. Como exemplo, a função tratadora pode receber um evento (parâmetro evt do

tratador, na Listagem 1) indicando que a tecla vermelha do controle remoto foi pressionada pelo telespectador.

Listagem 1: Representação de evento em NCLua.

```
evt = {
  class  = 'key',
  type   = 'press',
  key    = 'RED',
}
```

A função event.post é utilizada para que um NCLua poste eventos e possa, por exemplo, enviar dados pelo canal de interatividade ou sinalizar seu estado ao documento NCL. No exemplo a seguir (Listagem 2), o NCLua sinaliza ao documento o seu fim natural.

Listagem 2: Exemplo de evento postado por um NCLua para sinalizar ao documento NCL o seu fim natural.

```
event.post {
    class  = 'ncl',
    type   = 'presentation',
    action = 'stop',
}
```

Como um NCLua (por meio de seus tratadores) deve executar rapidamente, a função de envio de eventos nunca aguarda o retorno de um valor. Caso o destino necessite retornar uma informação ao NCLua, o fará através do envio de um novo evento.

2.1 Classes de Evento

O campo class de uma tabela representando um evento é obrigatório e tem a finalidade de separar os eventos em categorias. A classe identifica não somente a origem de eventos passados aos tratadores, mas também o seu destino, caso o evento seja gerado e postado por um script NCLua.

As seguintes classes de eventos estão definidas:

- Classe ncl: Usada na comunicação entre um NCLua e o documento NCL que contém o objeto de mídia.
- Classe key: Representa o pressionamento de teclas do controle remoto pelo usuário.
- Classe tcp: Permite acesso ao canal de interatividade por meio do protocolo tcp.
- Classe sms: Usada para envio e recebimento de mensagens SMS em dispositivos móveis.
- Classe edit: Permite que os comandos de edição ao vivo sejam disparados a partir de scripts NCLua.

- Classe si: Provê acesso a um conjunto de informações multiplexadas em um fluxo de transporte e transmitidas periodicamente por difusão.

- Classe user: Através dessa classe, aplicações podem estender sua funcionalidade criando seus próprios eventos.

Observe, pela Figura 1, que há eventos apenas de entrada, apenas de saída, e eventos que são usados em ambos os sentidos.

O modelo orientado a eventos de NCLua foi projetado para suportar outras entidades externas estendendo o modelo básico, bastando para isso definir novas classes de eventos [3].

3 Interagindo com o Documento NCL

Assim como qualquer objeto de mídia, um NCLua interage com o documento NCL através de elos. Em elos que acionam um NCLua, a condição satisfeita faz com que o NCLua receba um evento da classe ncl descrevendo a ação a ser tomada. No trecho da Listagem 3, quando o elo for disparado com o início de "videoId", o tratador de eventos de NCLua receberá o evento no código do objeto NCLua.

Listagem 3: Exemplo de códigos NCL e NCLua que tratam um evento de apresentação de um objeto NCL.

arquivo NCL que contém o objeto NCLua

```
<link xconnector="onBeginStart">
    <bind role="onBegin" component="videoId"/>
    <bind role="start"   component="luaId"/>
</link>
```

arquivo NCLua

```
-- Evento recebido pelo tratador do NCLua no
--   disparo do elo:
evt = {
    class  = 'ncl',
    type   = 'presentation',
    action = 'start',
}
```

Já em elos cuja condição depende de um NCLua, a ação do elo será disparada quando o NCLua sinalizar o evento que casa com a condição esperada. No trecho da Listagem 4, quando o NCLua sinalizar o evento indicado, o elo será disparado e a imagem exibida.

Listagem 4: Elo disparado pelo código do objeto NCLua.

```
<link xconnector="onBeginStart">
    <bind role="onEnd" component="luaId"/>
    <bind role="start" component="imagemId"/>
</link>
```

arquivo NCL que contém o objeto NCLua

arquivo NCLua

```
-- O elo acima será disparado quando o evento a seguir
-- for postado pelo NCLua 'luaId':
event.post {
    class  = 'ncl',
    type   = 'presentation',
    action = 'stop',
}
```

O tipo "presentation" indica que os eventos se referem à apresentação do NCLua. Como nenhuma âncora foi especificada, é assumida a âncora de conteúdo principal.

Como os dois exemplos indicam, a interação de um NCLua com o documento NCL se dá sempre através da classe de eventos ncl, seja para receber instruções do formatador, seja para notificar o estado de suas âncoras..

Exemplo 1 – Ciclo de Vida de Objetos NCLua

Este exemplo apresenta uma aplicação com o fim de ilustrar o modelo de execução de objetos imperativos NCLua em documentos NCL.

Ao iniciar a aplicação, três objetos NCLua são iniciados, cada qual com um comportamento interno diferente:

- O primeiro NCLua executa indefinidamente.
- O segundo NCLua termina a si próprio assim que é iniciado.
- O terceiro NCLua termina a si próprio três segundos após ser iniciado.

Associados a cada NCLua há um botão verde e um vermelho, indicando se o NCLua está ocorrendo ou terminado, respectivamente. As visões temporal e espacial da aplicação, mostradas na Figura 2, refletem o comportamento descrito.

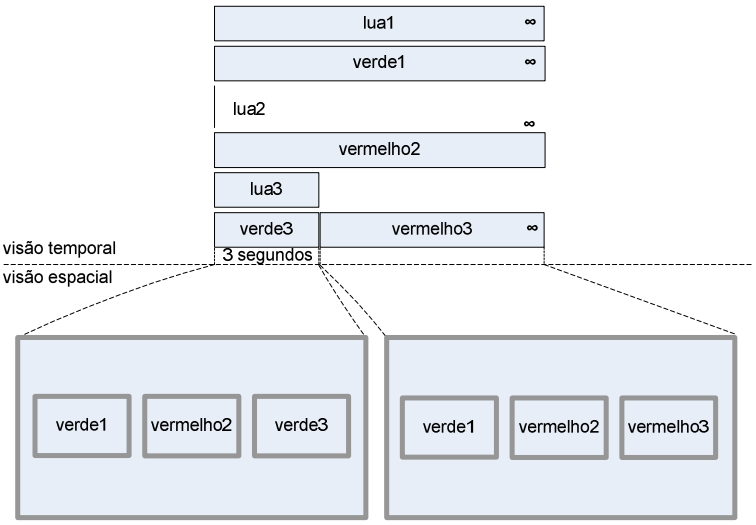

Figura 2: Visão temporal e espacial do Exemplo 1.

A Figura 3 exibe a visão estrutural do documento NCL. O primeiro NCLua é ligado aos outros por meio de um elo "onBeginStart". Como o primeiro NCLua está ligado à porta de entrada do documento, os três objetos iniciam juntamente com a aplicação. Cada NCLua se liga por meio de um elo "onBeginStart" com seu respectivo botão verde, para que eles sejam exibidos com o início de cada NCLua. Para esconder seu respectivo botão verde e exibir o vermelho, cada NCLua também utiliza um elo "onEndStopStart" com seus botões, conforme a figura mostra.

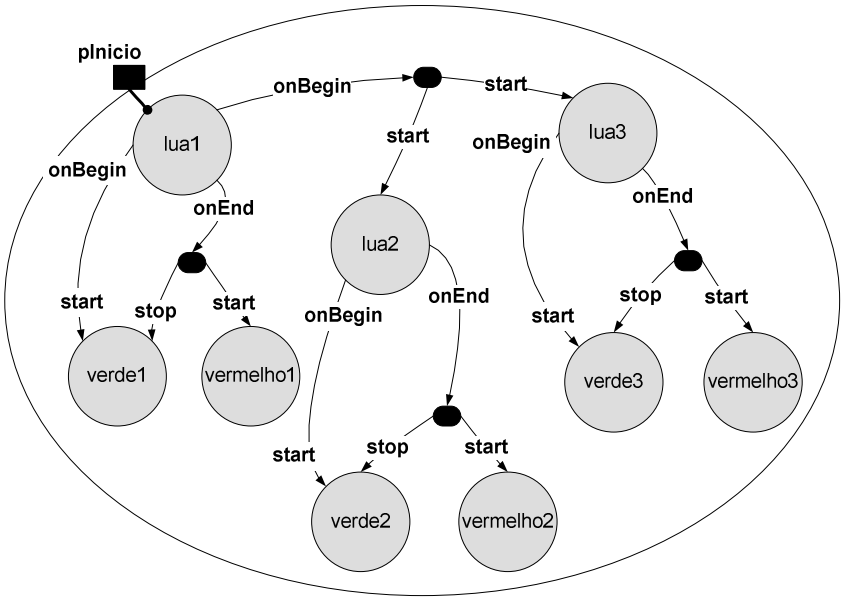

Figura 3: Visão estrutural do Exemplo 1.

O documento NCL é responsável por definir e iniciar os três objetos NCLua, assim como pela *cola lógica* entre cada NCLua e seus botões correspondentes, conforme definido na Listagem 5.

Listagem 5: Código NCL parcial do Exemplo 18.1.

```
<body>
   <!-- INICIA primeiro o NCLua -->
   <port id="pInicio" component="lua1"/>

   <!-- MIDIAS -->
   <media id="lua1"  src="1.lua"/>
   <media id="lua2"  src="2.lua"/>
   <media id="lua3"  src="3.lua"/>
   <media id="verde1"    src="verde1.png" descriptor="ds1"/>
   <media id="vermelho1" src="verm1.png"  descriptor="ds1"/>
   <media id="verde2"    src="verde2.png" descriptor="ds2"/>
   <media id="vermelho2" src="verm2.png"  descriptor="ds2"/>
   <media id="verde3"    src="verde3.png" descriptor="ds3"/>
   <media id="vermelho3" src="verm3.png"  descriptor="ds3"/>

   <!-- BEGIN lua1 -> START lua2/lua3 -->
   <link xconnector="onBeginStart">
      <bind role="onBegin" component="lua1"/>
      <bind role="start"   component="lua2"/>
      <bind role="start"   component="lua3"/>
   </link>

   <!-- BEGIN luaN -> START greenN -->
   <link xconnector="onBeginStart">
      <bind role="onBegin" component="lua1"/>
      <bind role="start"   component="verde1"/>
   </link>
   ...  <!-- Código idêntico para os outros NCLua -->

   <!-- END luaN -> STOP greenN | START redN -->
   <link xconnector="onEndStopStart">
      <bind role="onEnd" component="lua1"/>
      <bind role="stop"  component="verde1"/>
      <bind role="start" component="vermelho1"/>
   </link>
   ...  <!-- Código idêntico para os outros NCLua -->
</body>
```

Note como neste exemplo o NCL não aciona o término de nenhum objeto NCLua. Esse papel ficou a cargo de cada script NCLua, como mostrado a seguir.

- O primeiro NCLua é um script vazio (sem nenhuma linha de código). Em particular, como não possui um tratador de eventos, nunca sinaliza o seu término para o documento NCL. O efeito visual é a exibição permanente do primeiro botão verde.

Listagem 6: Código do arquivo 1.lua do Exemplo 1.

```
-- 1.lua
-- vazio
```

- O segundo NCLua registra um tratador de eventos que gera seu fim natural ao receber um "start" do documento NCL. Visualmente, instantaneamente após a exibição do segundo botão verde, é exibido o botão vermelho correspondente (o botão verde pode nem ser visto).

Listagem 7: Código do arquivo 2.lua do Exemplo 18.1.

```
-- 2.lua:
function tratador (evt)
  if (evt.class == 'ncl') and (evt.type == 'presentation')
     and (evt.action == 'start') then
    event.post {
        class  = 'ncl',
        type   = 'presentation',
        action = 'stop' }
  end
end
event.register(tratador)
```

Tão logo o evento indicando seu início é recebido, o NCLua posta um evento para sinalizar o seu fim natural.

- O terceiro NCLua registra um tratador de eventos que cria um temporizador de três segundos que, por sua vez, gera seu fim natural ao expirar. Como efeito visual, temos a exibição do terceiro botão verde e após três segundos a mudança para vermelho.

Listagem 8: Código do arquivo 3.lua do Exemplo 1.

```
-- 3.lua:
function tratador (evt)
  if (evt.class == 'ncl') and
     (evt.type == 'presentation') and
     (evt.action == 'start') then
    event.timer(3000,
        function()
            event.post {
                class  = 'ncl',
                type   = 'presentation',
                action = 'stop'
            }
        end)
  end
end
event.register(tratador)
```

O temporizador de três segundos (3000 milissegundos) é criado assim que o evento de início é recebido, passando a função que deve ser executada quando o temporizador expira. Essa função posta um evento idêntico ao do segundo NCLua para sinalizar seu fim natural.

Enquanto executa indefinidamente, o primeiro NCLua não consome recursos e poderia responder a eventos (apesar de não o fazer por não possuir um tratador para tal fim). O mesmo ocorre com o terceiro NCLua enquanto aguarda os três segundos para terminar.

3.1 Eventos em Âncoras de Conteúdo e Propriedades

No exemplo anterior, apenas a âncora de conteúdo principal de cada NCLua foi acionada. No entanto, âncoras de conteúdo específicas e propriedades também podem ser relacionadas entre o documento NCL e o objeto NCLua.

Dados os tipos de eventos NCL suportados, o campo type de um evento da classe ncl pode assumir os valores “presentation” ou “attribution”, conforme o atributo *eventType* dos conectores NCL. Eventos do tipo “selection” são tratados pela classe key.

3.1.1 Eventos do Tipo “presentation”

Eventos de apresentação estão associados à apresentação de âncoras de conteúdo, sendo identificadas pelo campo label do evento. O campo action indica a ação a ser realizada ou sinalizada pelo NCLua, dependendo se este está recebendo ou gerando o evento.

Um evento de apresentação possui a seguinte estrutura:

- `class: 'ncl'`

- `type: 'presentation'`
- `label: [string]` – Rótulo da âncora associada ao evento.
- `action: [string]` – Pode assumir os seguintes valores: `'start'`, `'stop'`, `'abort'`, `'pause'` e `'resume'`.

3.1.2 Eventos do Tipo "attribution"

Eventos de atribuição estão associados às propriedades do objeto NCLua, que são identificados pelo campo name.

O campo `value` é preenchido com o valor a ser atribuído à propriedade e é sempre uma string, uma vez que vem de um atributo NCL. A ação de "`start`" em um evento de atribuição corresponde ao papel "`set`" de um elo NCL.

Um evento de atribuição possui a seguinte estrutura:

- `class: 'ncl'`
- `type: 'attribution'`
- `name: [string]` – Nome da propriedade associada ao evento.
- `action: [string]` – Pode assumir os seguintes valores: `'start'`, `'stop'`, `'abort'`, `'pause'` e `'resume'`.
- `value: [string]` – Novo valor a ser atribuído à propriedade.

O campo `action` de um evento `ncl` (seja ele de apresentação ou atribuição) pode assumir os valores correspondentes aos seus tipos, como mostrado em [1]. No entanto, o nome das transições são usadas sem o "s" final ("starts" vira "start"), de maneira a unificar a sintaxe para eventos recebidos ou sinalizados pelo NCLua.

Exemplo 2 – Contador de Cliques

Vamos supor uma aplicação NCL que exibe um botão *Clique Aqui* em quatro momentos diferentes. Se o usuário selecioná-lo com o controle remoto por pelo menos três vezes, ao final da apresentação é exibido o botão *Você Ganhou*, caso contrário é exibido o botão *Você Perdeu*. A Figura 4 mostra as visões temporal e espacial da aplicação.

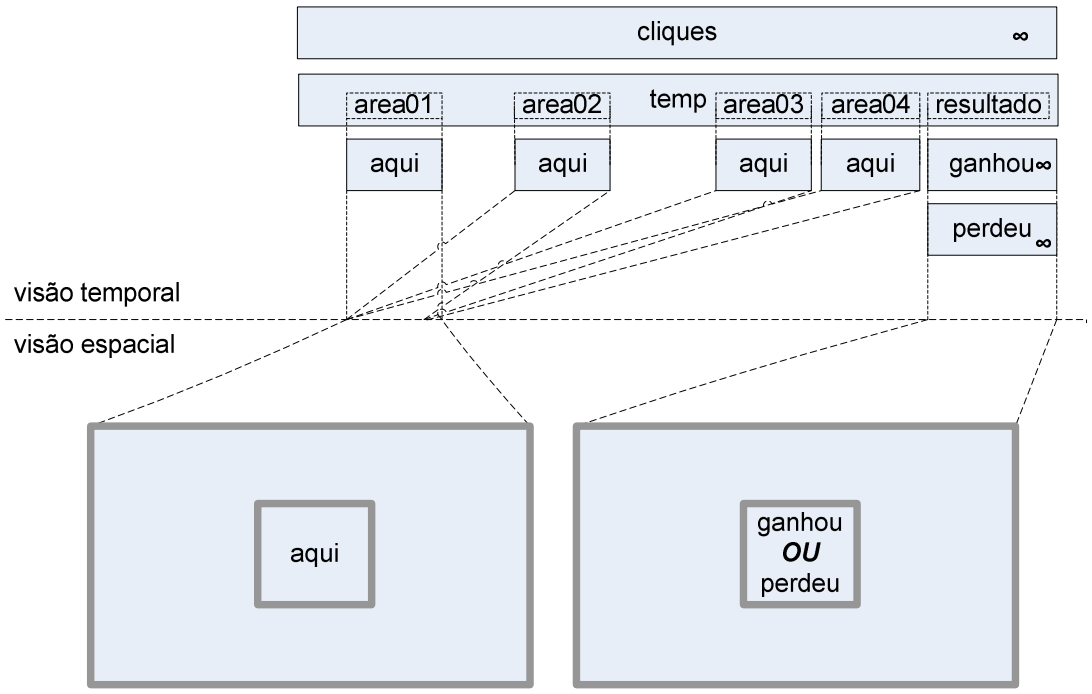

Figura 4: Visões temporal e espacial do Exemplo 2.

Não é possível fazer a contagem de cliques puramente em NCL de forma simples, uma vez que não há suporte a expressões aritméticas na linguagem. Neste exemplo, usaremos um NCLua para contar e armazenar o número de cliques em uma propriedade, que será consultada ao final para determinar o resultado.

O documento NCL mostrado a seguir define um temporizador ("*temp*") com quatro âncoras temporais ("*area01*" até "*area04*") durante as quais o botão *Clique aqui* é exibido. O temporizador também define uma âncora temporal para exibir o resultado após o botão ser exibido quatro vezes. Além do temporizador e dos botões, o documento define um NCLua responsável pela contagem dos cliques e exporta a propriedade "*contador*" para manter esse valor. A Figura 5 mostra a visão estrutural da aplicação.

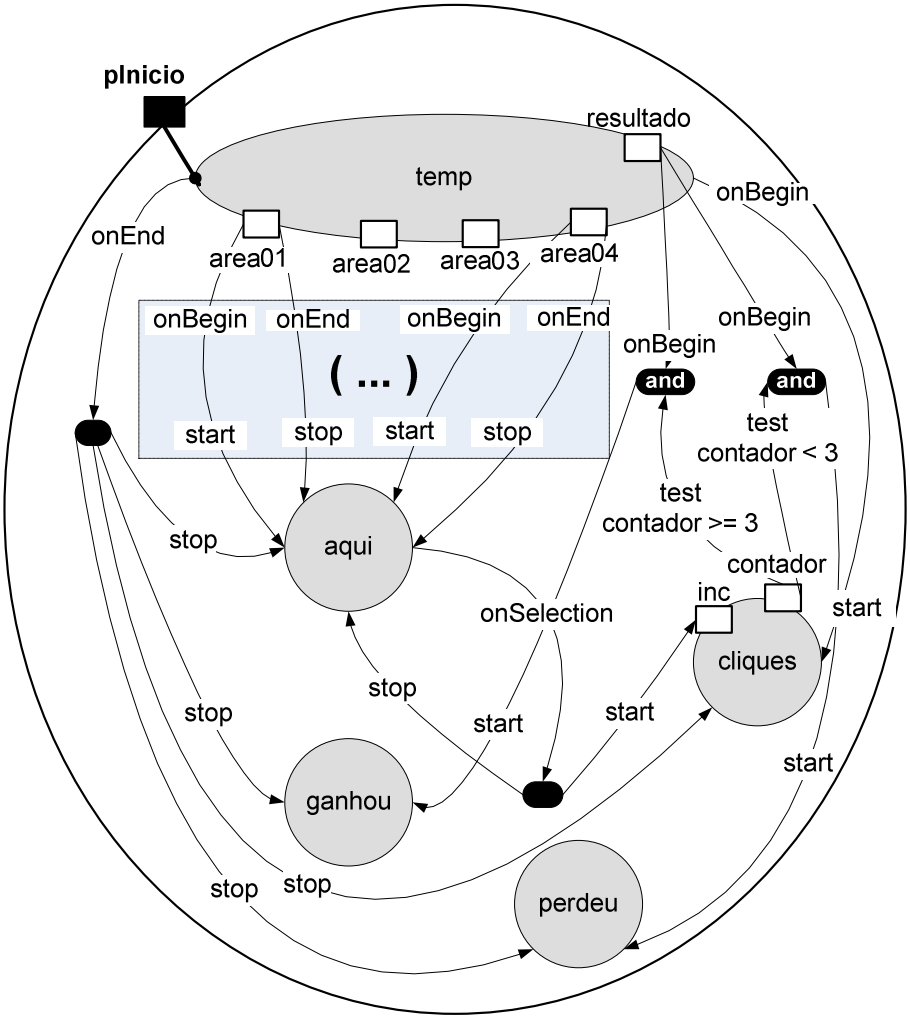

Figura 5: Visão estrutural do Exemplo 2.

O porta de entrada da aplicação é o temporizador, que também dispara o NCLua por meio de um elo "*onBeginStart*". Cada âncora de exibição do botão *Clique Aqui* possui um elo "*onBeginStart*" e "*onEndStop*" para o botão. Toda vez que o botão é selecionado pelo usuário (o que só pode acontecer enquanto está sendo exibido), a âncora "*inc*" do NCLua é iniciada e o botão é escondido, conforme o elo "*onSelectionStopSet*" saindo do próprio botão. O tratamento dado à âncora "*inc*" e à propriedade "*contador*" são especificados no código do NCLua. Ao final da âncora "*resultado*" do temporizador, dois elos testam se o valor da propriedade "*contador*" é maior ou menor que três cliques. Dependendo do resultado, a imagem *Você ganhou* ou *Você perdeu* é exibida.

O documento NCL para a aplicação é exibido na Listagem 9.

Listagem 9: Código NCL parcial do Exemplo 2.

```
<body>
  <!-- INÍCIO PELO TEMPORIZADOR -->
  <port id="pInicio" component="temp"/>
   <!-- TEMPORIZADOR -->
  <media id="temp">
    <!-- ÂNCORAS PARA EXIBIR O BOTÃO "CLIQUE AQUI" -->
    <area id="area01" begin="3s"  end="6s"/>
    <area id="area02" begin="10s" end="13s"/>
    <area id="area03" begin="17s" end="20s"/>
    <area id="area04" begin="24s" end="27s"/>
    <!-- ÂNCORA PARA EXIBIR O RESULTADO -->
    <area id="resultado" begin="35s"/>
  </media>

  <!-- NCLua -->
  <media id="cliques" src="cliques.lua">
    <area label="inc"/>
    <property name="contador"/>
  </media>

  <!-- "CLIQUE AQUI"/"VOCÊ GANHOU"/"VOCÊ PERDEU" -->
  <media id="aqui"   descriptor="dsBotao" src="aqui.jpg"/>
  <media id="ganhou" descriptor="dsBotao"
         src="ganhou.jpg"/>
  <media id="perdeu" descriptor="dsBotao"
         src="perdeu.jpg"/>

  <!-- TEMPORIZADOR -> NCLua -->
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="temp"/>
    <bind role="start"   component="cliques"/>
  </link>

  <!-- AREA[N] -> BOTÃO "CLIQUE AQUI" -->
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="temp"
          interface="area01"/>
    <bind role="start" component="aqui"/>
  </link>
  ... <!-- Código idêntico para as outras âncoras -->

  <link xconnector="onEndStop">
    <bind role="onEnd" component="temp"
          interface="area01"/>
    <bind role="stop" component="aqui"/>
  </link>
```

```
  ... <!-- Código idêntico para as outras âncoras -->

  <!-- SELECT "CLIQUE AQUI" -> CLIQUES.INCREMENTA -->
  <link xconnector="onSelectionStopSet">
    <bind role="onSelection" component="aqui"/>
    <bind role="stop"        component="aqui"/>
    <bind role="start"       component="cliques"
                             interface="inc"/>
  </link>

  <!-- TESTE DO RESULTADO -->
  <link xconnector="onCondGteBeginStart">
  <linkParam name="var" value="3"/>
    <bind role="onBegin  component="temp"
                         interface="resultado"/>
    <bind role="attNodeTest" component="cliques"
                             interface="contador"/>
    <bind role="start" component="ganhou"/>
  </link>
  <link xconnector="onCondLtBeginStart">
    <linkParam name="var" value="3"/>
    <bind role="onBegin" component="temp"
                         interface="resultado"/>
    <bind role="attNodeTest" component="cliques"
                             interface="contador"/>
    <bind role="start" component="perdeu"/>
  </link>
</body>
```

O código do NCLua deve lidar em sua função tratadora de eventos com as duas interfaces com o documento NCL – a âncora "*inc*" e a propriedade "*contador*". A Listagem 10 apresenta o código do script.

Listagem 10: Código NCLua do Exemplo 2.

```
local contador = 0
function tratador (evt)
   contador = contador + 1
   local evtContador = {
      class = 'ncl',
      type  = 'attribution',
      name  = 'contador',
      value = contador,
   }
   evtContador.action = 'start'
   event.post(evtContador)
   evtContador.action = 'stop'
   event.post(evtContador)
   event.post {
      class  = 'ncl',
      type   = 'presentation',
      label  = 'inc',
      action = 'stop',
   }
end
event.register(tratador,'ncl','presentation','inc','start')
```

O script começa declarando uma variável para guardar a contagem de cliques. A variável não possui relação direta com a propriedade, de mesmo nome, do NCLua definida no documento NCL. O manuseio da propriedade é feito através da postagem de eventos, conforme apresentado a seguir.

A função "*tratador*" é então definida e registrada (na última linha do código). Os parâmetros extras na chamada à função *register* servem para filtrar apenas os eventos desejados, no caso, eventos *ncl* de início de apresentação da âncora "inc". Quando o botão é selecionado, iniciando a âncora "inc", a função *tratador* é executada. A função incrementa o,contador interno e posta um evento de início de atribuição (*action*='start'*)*, para indicar a mudança na propriedade "*contador*".

Note que é necessário sinalizar também o fim da atribuição, fazendo com que a máquina de estados da propriedade "*contador*" volte ao estado "sleeping" e futuras atribuições surtam efeito. Pelo mesmo motivo, é necessário sinalizar o término da âncora "*inc*", postando o evento de "*stop*" da apresentação, no fim da função.

O NCLua não tem como saber o momento exato em que a propriedade "*contador*" será consultada pelo documento NCL — por isso, sempre que incrementa o valor, também atualiza a propriedade.

4 Desenhando na Região do Objeto

Quando um NCLua é carregado, o formatador NCL cria um objeto gráfico para representar a região associada à mídia NCLua no documento. Esse objeto gráfico é pré-carregado na variável global "*canvas*" do script, e é através dela que todas as operações gráficas são efetuadas. Caso o objeto de mídia NCLua não esteja associado a nenhuma região, então o valor do "*canvas*" será igual a "*nil*".

Como exemplo, caso a região a seguir esteja associada ao objeto NCLua, a variável "*canvas*" do script irá representá-la, com tamanho 300x100 e posicionada em (20,200).

```
<region id="luaRegion" width="300" height="100" top="200" left="20"/>
```

Existem diversas operações gráficas suportadas pelo módulo de canvas, tais como desenho de linhas, textos e imagens. A lista completa de operações pode ser consultada em [1]. O exemplo a seguir cria um novo canvas representando a imagem passada, desenhando-a centralizada na região e acompanhada de uma legenda. A Figura 6 exibe o resultado visual da execução do script da Listagem 12.

Figura 6: Resultado da execução do script da Listagem 12.

Listagem 11: Script ilustrando o uso do canvas.

```
local regLarg, regAlt = canvas:attrSize()

local img = canvas:new('ginga.png')
local imgLarg, imgAlt = img:attrSize()
local imgX = (regLarg - imgLarg) / 2
local imgY = (regAlt - imgAlt) / 2
canvas:compose(imgX, imgY, img)

local txt = 'TV Digital se faz com Ginga'
local txtLarg, txtAlt = canvas:measureText(txt)
local txtX = (regLarg - txtLarg) / 2
local txtY = imgY + imgAlt + 2
canvas:attrColor('white')
canvas:drawText(txtX, txtY, txt)

canvas:flush()
```

O script da Listagem 12 começa guardando as dimensões da região inteira do NCLua nas variáveis `regLarg` e `regAlt` com a chamada à função `canvas:attrSize()`, que retorna a largura e altura do canvas. Em seguida, o método `canvas:new()` recebe uma imagem e retorna um novo canvas, "*img*", que a representa. As dimensões da imagem são então recuperadas e utilizadas para calcular a posição centralizada na qual a imagem é desenhada na região (`imgX` e `imgY`). A chamada `canvas:compose(imgX,imgY,img)` sobrepõe a imagem do *Ginga* à região do NCLua na posição informada.2

Para desenhar a legenda do texto, primeiramente é medido o tamanho que o texto ocupa no canvas, com a chamada à `canvas:measureText()`. A posição horizontal centralizada do texto é calculada de maneira similar a da imagem. Já sua posição vertical é calculada um pouco abaixo da imagem. Por fim, a chamada à `canvas:attrColor('white')` altera o atributo de cor para branco, em futuras operações sobre o canvas, para então desenhar o texto na posição calculada. Apenas com a chamada à função `canvas:flush()`, as operações gráficas descritas são de fato efetuadas.

Como se pode deduzir ao observar o exemplo, um objeto `canvas` guarda em seu estado diversos atributos sobre os quais as primitivas gráficas devem operar. Os atributos são acessados através de métodos de prefixo `attr` acompanhado do nome do atributo (e.g. `attrColor`), e servem tanto para leitura, quando chamados sem parâmetros (como é o caso da chamada à `attrSize()`), como para escrita, quando chamados com os novos parâmetros (como é o caso da chamada à `attrColor('white')`).

---

2 As coordenadas passadas para todos os métodos gráficos são sempre relativas ao ponto mais à esquerda e ao topo do canvas (0,0), como é comum em sistemas gráficos.

A referência completa do módulo *canvas*, com todos os atributos e operações gráficas suportados, pode ser encontrada em [1].

Exemplo 3 – Gráficos e Controle Remoto

Este exemplo apresenta um jogo bastante simples, onde são exibidos os logotipos das linguagens Lua e NCL. O usuário deve mover com o controle remoto o logotipo de Lua até o de NCL, quando é então exibida uma imagem indicando o fim do jogo. O exemplo explora os eventos da classe `key` e o pacote gráfico `canvas`. A Figura 7 mostra as visões temporal e espacial da aplicação.

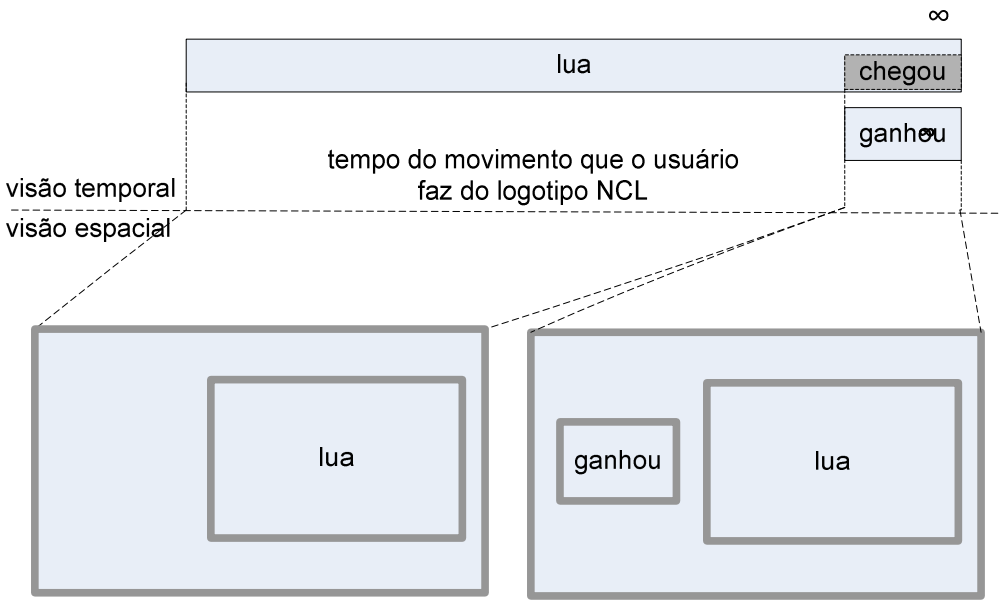

Figura 7: Visões temporal e espacial do Exemplo 3.

O documento NCL possui como mídias apenas o objeto NCLua ("*lua*") com o jogo e a imagem "Você ganhou" ("*ganhou*"), que é exibida quando o logotipo de Lua encontra o de NCL (acontecimento representado pela âncora "*chegou*" do NCLua). O jogo inicia assim que o documento é carregado. A Figura 8 mostra a visão estrutural da aplicação.

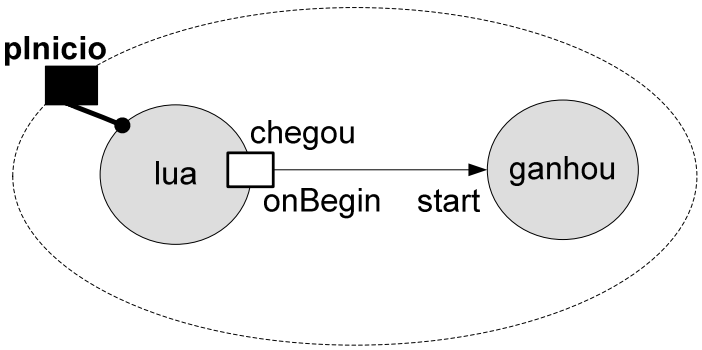

Figura 8: Visão estrutural do Exemplo 3.

O código NCL é bem simples, como definido na Listagem 12.

Listagem 12: Código NCL parcial do Exemplo 3.

```
<body>
  <port id="pInicio" component="lua"/>
  <media id="lua" src="jogo.lua" descriptor="dsLua">
    <area id="chegou"/>
  </media>
     <media id="ganhou" src="ganhou.jpg"
            descriptor="dsGanhou"/>

  <link xconnector="onBeginStart">
    <bind role="onBegin" component="lua"
                         interface="chegou"/>
    <bind role="start" component="ganhou"/>
    </link>
</body>
```

Nos exemplos anteriores, o código da aplicação estava concentrado no documento NCL, agora praticamente todo o código está dentro do NCLua. As primeiras linhas do código NCLua criam duas tabelas para representar os logotipos de Lua e NCL (Listagem 13).

Listagem 13: Primeira parte do código NCLua do Exemplo 3.

```
local img = canvas:new('lua.png')
local larg, alt = img:attrSize()
local logoLua = { canvas=img, x=10,y=10,
                  larg=larg,alt=alt }

local img = canvas:new('ncl.png')
local larg, alt = img:attrSize()
local logoNcl = { canvas=img, x=10,y=10,
                  larg=larg,alt=alt }
```

Conforme a Listagem 13, o logotipo de Lua é representado pela tabela *logoLua*, guardando o *canvas* com a imagem *lua.png*, as posições *x* e *y* onde deve ser desenhado, e sua largura e altura. A tabela *logoNcl* guarda os mesmos atributos (mas para a imagem *ncl.png*) para representar o logotipo de NCL. Em seguida, definimos na Listagem 14 a função de redesenho da tela, chamada durante a execução do NCLua toda vez que o usuário

movimenta o logotipo de Lua. Sempre que chamada, a função `redraw` desenha um retângulo preto ocupando a região inteira (para limpá-la), e em seguida compõe os canvas das duas tabelas `logoNcl` e `logoLua` sobre o canvas principal, em suas posições correntes.

Listagem 14: Segunda parte do código NCLua do Exemplo 3.

```
function redraw ()
    canvas:attrColor('black')
    canvas:drawRect('fill', 0,0, canvas:attrSize())
    canvas:compose(logoNcl.x, logoNcl.y, logoNcl.canvas)
    canvas:compose(logoLua.x, logoLua.y, logoLua.canvas)
    canvas:flush()
end
```

Sempre que chamada, a função Por fim, conforme a Listagem 15, definimos a função tratadora de eventos, responsável por responder às teclas do controle remoto e por sinalizar, ao documento NCL, o momento em que os logotipos se sobrepõem.

Listagem 15: Terceira parte do código NCLua do Exemplo 3.

```
function tratador (evt)
  -- apenas eventos de tecla interessam
   if evt.class == 'key' and evt.type == 'press'
   then
     -- apenas as setas que movem o logotipo Lua interessam
     if evt.key == 'CURSOR_UP' then
       logoLua.y = logoLua.y - 10
     elseif evt.key == 'CURSOR_DOWN' then
       logoLua.y = logoLua.y + 10
     elseif evt.key == 'CURSOR_LEFT' then
       logoLua.x = logoLua.x - 10
     elseif evt.key == 'CURSOR_RIGHT' then
       logoLua.x = logoLua.x + 10
     end
    -- testa se os logotipos estão sobrepostos
    if sobrepondo(logoLua, logoNcl) then
       -- sinaliza que a ancora "chegou" esta ocorrendo
       event.post {
        class  = 'ncl',
        type   = 'presentation',
        label  = 'chegou',
        action = 'start',
      }
    end
    redraw()  -- redesenha a tela inteira
  end
end
event.register(tratador)
```

Na Listagem 15, o script inicia testando se alguma das teclas direcionais foi pressionada (campo `key` do evento de tecla), alterando a posição do logotipo de Lua de acordo com a tecla. Caso os logotipos estejam se sobrepondo, é postado o evento para sinalizar que a âncora "*chegou*" iniciou. A função "*sobrepondo*" foi omitida sem prejuízo de

entendimento. Por fim, a tela é redesenhada, qualquer que seja a tecla que tenha sido pressionada.

Em um evento da classe *key*, o valor da tecla é uma *string*, guardada no campo *key* . O campo *type* pode ser *'press'* ou *'release'*, dependendo se a tecla foi pressionada ou solta.

4.1 Programando com Animações

Imaginemos agora que o logotipo de Lua fosse movimentado com o passar do tempo, como em uma animação, em vez de ser guiado pelo controle remoto. Ao receber a instrução de "*start*", o trecho hipotético da Listagem 16 atualiza a posição do logotipo a cada 30 milissegundos, até que sua posição chegue a 100.

Listagem 16: Exemplo (ruim) de animação em NCLua.

```
function tratador (evt)
   if evt.action == 'start' then
      while logoLua.x < 100 do
         logoLua.x = logoLua.x + 5
         redraw()
         sleep(30)
      end
   end
end
event.register(tratador, 'ncl', 'presentation')
```

O problema com esse código é que a chamada à função `sleep` bloqueia o tratador, não permitindo que outros eventos sejam processados, inviabilizando essa proposta.

Uma possível solução seria criar uma função de `update` a ser chamada a cada 30 milissegundos por um temporizador (Listagem 17).

Listagem 17: Exemplo de animação em NCLua que utiliza um temporizador.

```
function update ()
   logoLua.x = logoLua.x + 5
   redraw()
   if logoLua.x < 100 then
      event.timer(update, 30)
   end
end
function tratador (evt)
   if evt.action == 'start' then
      update()
   end
end
event.register(tratador, 'ncl', 'presentation')
```

Pela listagem anterior, ao ser chamado, o tratador ativa a função `update`, que atualiza a posição do logotipo de Lua, chama a função de redesenho e, caso o logotipo ainda não

tenha alcançado a posição 100, se programa para ser chamada novamente após 30 milissegundos de espera.

Essa solução não é tão legível quanto um *loop* localizado, mas mantém o NCLua reativo a possíveis eventos que possam chegar durante os 30 milissegundos.

4.2 Co-rotinas de Lua

O mecanismo de co-rotinas de Lua simplifica muito a programação de aplicações em que muitos objetos interagem e devem estar permanentemente sincronizados.

Apesar de serem comumente comparadas a *threads*, co-rotinas são, na verdade, muito mais próximas ao conceito comum de função (ou rotina). Da mesma forma que funções, chamadas a co-rotinas são síncronas, isto é, o código que chama uma co-rotina sempre aguarda o seu retorno. No entanto, uma co-rotina pode explicitamente suspender sua própria execução, *preservando o seu estado corrente* (i.e. variáveis locais, contador de instruções), antes do seu término completo. Ao se suspender, uma co-rotina retorna imediatamente o controle a quem a chamou. Nesse caso, a co-rotina pode, a partir de outro trecho do código, ter sua execução continuada do ponto onde parou, executando até o seu término ou uma nova suspensão. Note que uma co-rotina que não se suspende antes de terminar é exatamente uma rotina comum.

O uso de co-rotinas em Lua é feito através das seguintes primitivas:

- coroutine.create (f): Retorna uma nova co-rotina a partir da função passada como parâmetro. Cria os meios pelos quais o estado de uma co-rotina é preservado entre suspensões.
- coroutine.resume (co): Recomeça a execução da co-rotina do ponto onde parou. Também serve para iniciar a co-rotina.
- coroutine.yield (): Suspende a execução da co-rotina em execução.
- coroutine.status (co): Retorna o estado da co-rotina passada, que pode ser *running*, *suspended*, *normal* ou *dead*.

Voltando ao exemplo da animação do logotipo de Lua, agora podemos usar o *loop* problemático da Listagem 16, bastando colocá-lo dentro de uma co-rotina e trocando a chamada *sleep()* por *coroutine.yield()* (Listagem 18).

Na listagem, a função *update* acorda a co-rotina a cada 30 milissegundos, até que ela termine, o que acontece quando o logotipo chega à posição 100 quando, então, o *loop* é encerrado.

O uso de co-rotinas simula a execução sequencial de código em situações em que, na verdade, uma entidade externa à aplicação controla sua execução (o temporizador, no exemplo apresentado).

É possível ainda trocar valores entre chamadas e suspensões de co-rotinas, analogamente à passagem de parâmetros de funções. Para uma descrição mais detalhada do suporte a co-rotinas de Lua, o manual da linguagem deve ser consultado [2].

Listagem 18: Exemplo de uso de co-rotinas para realizar uma animação.

```
function animaLogoLua ()
   while lua.x < 100 do
      lua.x = lua.x + 5
      redraw()
          coroutine.yield() -- sleep(30)
       end
    end
    coAnimaLogoLua = coroutine.create(animaLogoLua)

function update ()
   coroutine.resume(coAnimaLogoLua)
   if coroutine.status(coAnimaLogoLua) ~= 'dead' then
      event.timer(30, update)
   end
end

function tratador (evt)
   if evt.action == 'start' then
          update()
       end
end
event.register(tratador, 'ncl', 'presentation')
```

Exemplo 4 – Corrida de Cavalos (Parte I)

Imaginemos a simulação de uma corrida de cavalos. Neste exemplo, a primeira parte da simulação, o script para a animação de apenas um cavalo é criado. A segunda parte, apresentada no Exemplo 5, reúsa esse script na definição de todos os cavalos.

A Figura 9 apresenta uma tira de imagens, com o cavalo em diversas posições. A cada intervalo de tempo, uma imagem diferente do cavalo será mostrada na tela, dando uma sensação de realidade à animação.

Figura 9: Imagem dos cavalos do Exemplo 4.

A tabela representando o cavalo é um pouco diferente da usada para representar os logotipos do exemplo anterior:

```
local img = canvas:new('cavalo.png')
local larg, alt = img:attrSize()
local cavalo = { canvas=img,  quadro=0, y=0,
                 larg=larg/5, alt=alt }
```

A largura do cavalo é igual à largura da imagem dividida pelo número de quadros da tira. Além disso, a tabela também guarda o quadro a ser exibido, que varia durante a animação.

A função de redesenho deve exibir apenas a parte da imagem dos cavalos que representa o quadro corrente. A função *compose* aceita parâmetros extras para indicar qual região do canvas de origem (cavalo.img) deve ser composta. Os parâmetros são as posições x,y da origem e a largura e altura a partir desse ponto.

```
function redraw ()
   canvas:attrColor('black')
   canvas:drawRect('fill', 0,0, canvas:attrSize())
   canvas:compose( cavalo.x, cavalo.y, cavalo.canvas,
                   cavalo.quadro*cavalo.larg,0,
                   cavalo.larg,cavalo.alt) )
   canvas:flush()
end
```

Para animar o cavalo, usaremos uma co-rotina, como no exemplo anterior:

```
function animaCavalo ()
   local partida = 10     -- posição da linha de partida
   local chegada = 500    -- posição da linha de chegada
   local mudaQuadro = 20  -- muda de quadro a cada 20px
   cavalo.x = partida
   while cavalo.x < chegada do
      coroutine.yield()
      local deslocamento = 3 + math.random(1,3)
      cavalo.x = cavalo.x + deslocamento
      mudaQuadro = mudaQuadro - deslocamento
      if mudaQuadro < 0 then
         quadro = (quadro + 1) % 5
         mudaQuadro = 20
      end
      redraw()
   end
end
coAnimaCavalo = coroutine.create(animaCavalo)
```

O deslocamento do cavalo a cada 30 milissegundos varia de 4 a 6 pixels, de acordo com a chamada à função *math.random* (para que a aplicação tenha alguma imprevisibilidade). O código para a função *update* e o tratador de eventos são iguais ao do exemplo anterior, bastando trocar o nome da co-rotina de animação.

Novamente, o uso de uma co-rotina permite que o código do cavalo seja escrito sequencialmente, facilitando o seu desenvolvimento e entendimento.

5 Reúso de Código Lua

Conforme mencionado, um documento NCL não contém código Lua diretamente, mas referencia um objeto contendo o código Lua. Isso cria dois ambientes isolados de

programação e também permite que um mesmo código Lua possa ser reusado em diversos objetos de mídia NCL. Para tornar ainda mais versátil essa abordagem, elos de atribuição em NCL podem ser usados para informar parâmetros a scripts Lua. Âncoras em objetos de mídia NCLua permitem que o código Lua tanto sirva como condição para o disparo de elos como trate ações provenientes de elos, como já vimos.

O reúso de código Lua permite, por exemplo, que se definam componentes gráficos (ou *widgets*), tais como caixas de texto ou painéis de opção, uma única vez. Cada componente gráfico poderia ser reusado e parametrizado em documentos NCL como um objeto de mídia orquestrado, tal como qualquer outro objeto de mídia.

Exemplo 5 – Corrida de Cavalos (Parte II)

Este exemplo estende o Exemplo 4 para ilustrar uma corrida entre cavalos. O mesmo script NCLua responsável pela animação de um único cavalo é reusado na especificação de quatro objetos de mídia, demonstrando o uso de NCL como um orquestrador entre objetos de código imperativo de igual conteúdo.

A Figura 10 apresenta a visão estrutural do exemplo. O objeto de mídia "cavalo1" é a porta de início da exibição do documento. Quando "cavalo1" é iniciado, os outros objetos de mídia "cavalo2", "cavalo3" e "cavalo4" também são iniciados. Os quatro objetos de mídia referenciam o mesmo arquivo de extensão .lua.

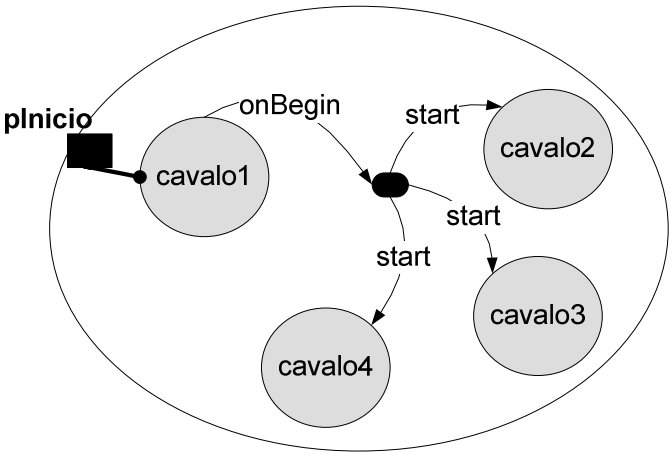

Figura 10: Visão Estrutural do Exemplo 5.

A Figura 11 ilustra as visões temporal e espacial do exemplo. Para simplificar,na visão temporal, as quatro animações em NCLua aparecem com o mesmo tempo total de exibição. Na prática, conforme é explicado no Exemplo 4, o tempo total de cada animação é imprevisível, já que cada cavalo possui um fator de deslocamento que varia de 4 a 6 pixels a cada 30 milissegundos.

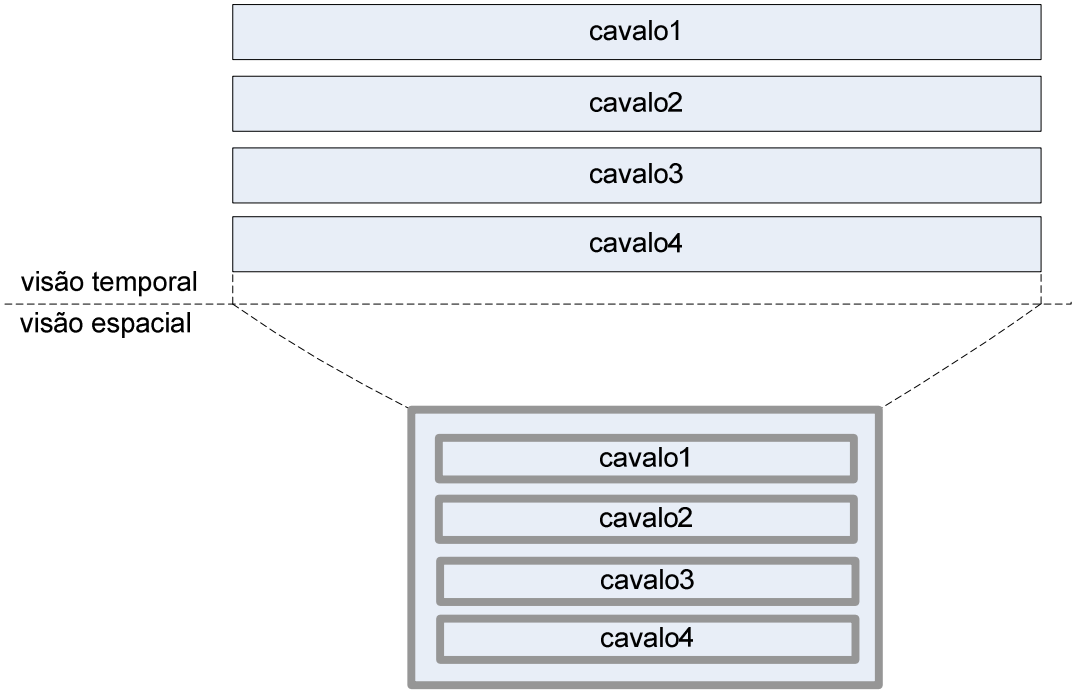

Figura 11: Visões temporal e espacial do Exemplo 5.

Exemplo 6 – Passagem de Valores

Esta seção apresenta uma aplicação que ilustra o reúso de código Lua em documentos NCL e a passagem de valores de propriedades entre objetos NCLua.

Quando inicia a aplicação, um campo de entrada e dois campos de saída são exibidos na tela. Enquanto o usuário preenche o campo de entrada, o primeiro campo de saída é automaticamente atualizado com o texto que vai sendo digitado. Apenas quando o usuário encerra a digitação por meio do botão OK do controle remoto, ou através do ENTER no teclado alfanumérico, o texto digitado até o momento é copiado para o segundo campo de saída.

A Figura 12 ilustra a visão estrutural deste exemplo. O objeto NCLua "*entrada*" é disparado no início do documento, o que faz iniciar também os outros dois objetos NCLua "*saida1*" e "*saida2*". O foco para digitação inicia no campo de entrada, cuja propriedade "*texto*" armazena o valor atual digitado pelo usuário. A cada mudança no valor da propriedade "*texto*", seu valor é copiado para a propriedade "*texto*" do objeto "*saida1*", o que atualiza o que é exibido pelo primeiro campo de saída. Por outro lado, apenas quando a âncora "*selecao*" do objeto "*entrada*" é iniciada, seu valor é copiado para a propriedade "*texto*" do objeto "*saida2*".

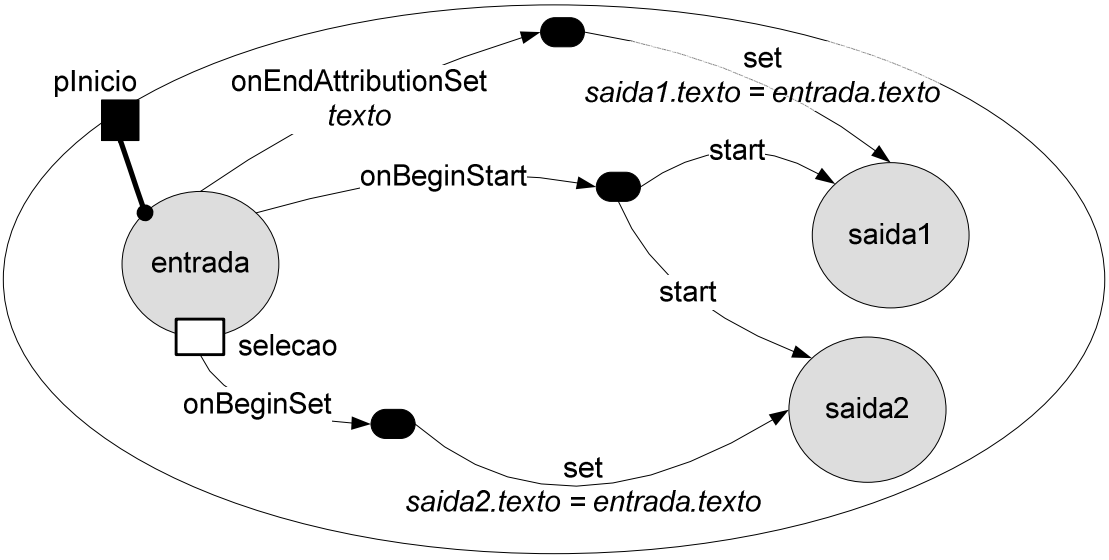

Figura 12: Visão estrutural do Exemplo 5.

Os campos de entrada e saída são implementados em dois scripts Lua diferentes. Ambos os códigos tratam a propriedade "*texto*" mantendo-a com o texto visualizado. No caso do campo de entrada, a propriedade "*texto*" é controlada pelo próprio NCLua, e é alterada toda vez que uma nova tecla é pressionada. No caso do campo de saída, a propriedade "*texto*" deve ser controlada pelo documento NCL, através de elos de atribuição.

Neste exemplo, os objetos NCLua podem ser vistos como caixas pretas, não importando o conteúdo dos arquivos .lua. Para o autor do documento NCL basta saber a interface que cada um dos NCLua oferece com suas propriedades "*texto*" e a âncora "*selecao*", especificamente no campo de entrada. Dessa forma, os arquivos .lua podem ser reusados em outras aplicações.

O campo de entrada é representado em NCL pelo seguinte código:

```
<media id="entrada" src="input.lua"  descriptor="dsInput">
    <area id="selecao"/>
    <property name="texto"/>
</media>
```

O objeto possui a âncora "*selecao*" que é iniciada sempre que o usuário pressiona OK no controle remoto ou ENTER no teclado alfanumérico.

Os campos de saída são representados com o seguinte código:

```
<media id="saida1" src="output.lua" descriptor="dsOutput1">
    <property name="texto"/>
</media>
<media id="saida2" src="output.lua" descriptor="dsOutput2">
    <property name="texto"/>
</media>
```

A parte mais importante do documento é a que contém os elos responsáveis em copiar o conteúdo do campo de entrada para os campos de saída.

O primeiro campo de saída é atualizado sempre que a propriedade "*texto*" do campo de entrada é alterada:

```
<link xconnector="onEndAttributionSet">
    <bind role="onEndAttribution" component="entrada"
             interface="texto"/>
    <bind role="set" component="saida1" interface="texto">
        <bindParam name="var" value="$get"/>
    </bind>
    <bind role="get" component="entrada" interface="texto"/>
</link>
```

A associação com o papel "*get*", definido no próprio elo, permite consultar o valor de uma propriedade de qualquer objeto, guardando-o na variável "$get". No exemplo acima, a propriedade consultada é a "*texto*" do objeto "*entrada*". A variável $get, por sua vez, é utilizada na associação com o papel de atribuição "*set*", fazendo com que o valor da entrada seja copiado para a propriedade "*texto*" do objeto "*saida1*".

O segundo campo de saída é atualizado somente quando a âncora "*selecao*" do objeto de "*entrada*" é iniciada:

```
<link xconnector="onBeginSet">
    <bind role="onBegin" component="entrada"
             interface="selecao"/>
    <bind role="set" component="saida2" interface="texto">
        <bindParam name="var" value="$get"/>
    </bind>
    <bind role="get" component="entrada" interface="texto"/>
</link>
```

O mesmo mecanismo de cópia de valores de uma propriedade para outra também é usado para a cópia do valor digitado no campo de entrada para o segundo campo de saída.

Referências

[1] ABNT NBR 15606-2:2007. 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. Setembro 2007.

[2] Soares, L. F. G., Rodrigues, R. F., Moreno, M. F. Ginga-NCL: the Declarative Environment of the Brazilian Digital TV System. Journal of the Brazilian Computer Society. n.4, v. 12, Março, 2007.

[3] Tratamento de Variáveis em Linguagens Declarativas XML Baseadas no Tempo. XIII Simpósio Brasileiro de Sistemas Multimídia e Web - WebMedia2007. Gramado, Brasil - Outubro de 2007.

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