4
Implementação da Integração

Este capítulo descreve como foram implementadas as linguagens e os elementos de integração na ferramenta protótipo que reutilizou o núcleo e estrutura da ferramenta de código aberto *Oryx*.

4.1. Ferramenta *Oryx*

A ferramenta *Oryx* é um framework acadêmico de código aberto inicialmente desenvolvido para oferecer a modelagem gráfica de processos de negócio (BPMo). Sua tecnologia é baseada em web, sendo executado através de *browser*, o que elimina a necessidade de instalação do software. A ferramenta oferece várias linguagens para modelagem, tais como BPMN (*Business Process Model and Notation*) e EPC (*Event-driven Process Chain*), além disso, é extensível a novas funções através da adição de *plugins* [*Oryx*12].

Sua arquitetura é bem definida e já foi reutilizada para o desenvolvimento de soluções comerciais como, por exemplo, a ferramenta Signavio [Kunze&Weske10]. Além disso, a ferramenta *Oryx* encontra-se em uso por grandes empresas, por exemplo, a organização Serpro [Serpro11], que realiza customizações para adequá-la às suas necessidades.

A Figura 50 apresenta alguns diagramas disponíveis para uso através da ferramenta *Oryx*. Atualmente a ferramenta encontra-se na versão Beta e pode ser acessada pelo site oficial do *Oryx*: “http://bpt.hpi.uni-potsdam.de/*Oryx*/Research”.

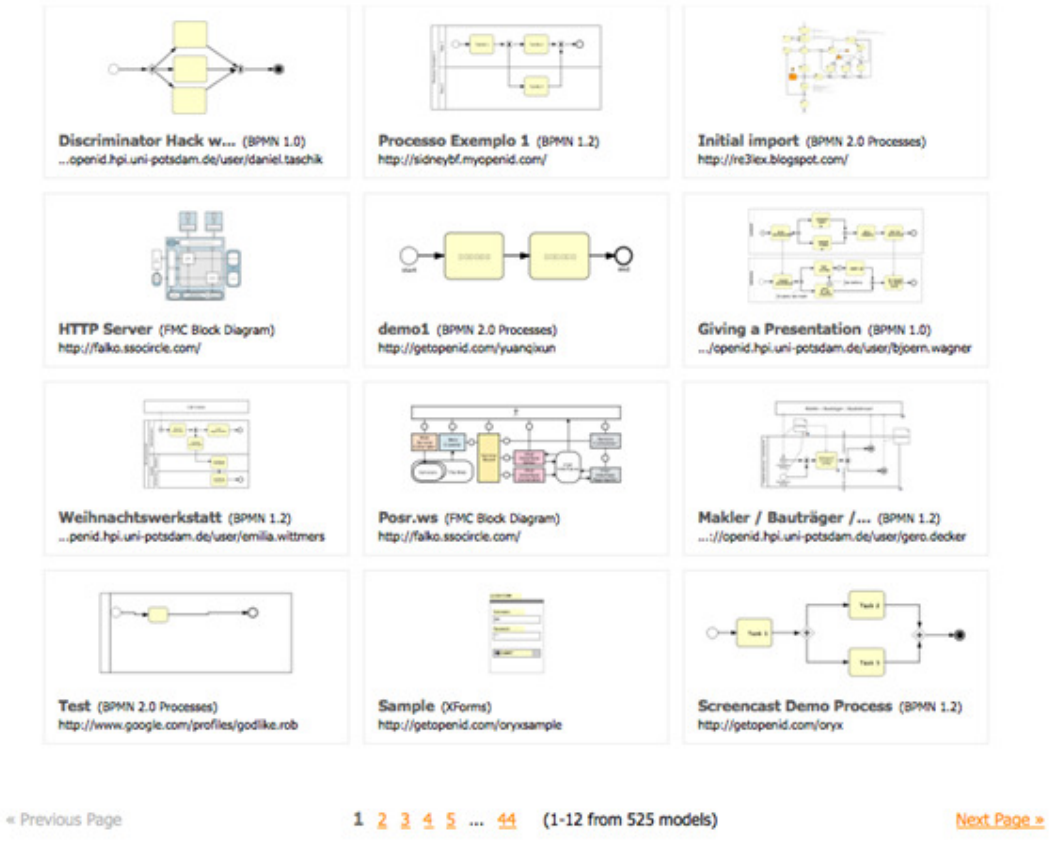


Figura 50 - Painel com diversos exemplos de tipos de modelos que podem ser modelados na ferramenta

4.2. Estudo da ferramenta

Apesar de possuir o código aberto, pouca documentação existe sobre a infraestrutura da ferramenta, além disso, a documentação disponível encontra-se em línguas estrangeiras como o inglês e alemão. Portanto, para adquirir o conhecimento necessário para customizar a ferramenta de acordo com as nossas necessidades foram demandados esforços de pesquisa nas bibliografias existentes, mas principalmente a partir do código fonte, utilizando o método de tentativa e erro.

Como forma de projetar um caminho estratégico para alcançar o conhecimento necessário sobre a ferramenta para posteriormente customizá-la, definiu-se o uso das seguintes técnicas da Engenharia de Software: Engenharia Reversa e Reengenharia de Software.

A Engenharia de Software utiliza-se dos princípios da Engenharia Reversa como uma coleção de teorias, metodologias e técnicas capazes de suportar a extração e abstração de informações de um software existente, produzindo documentos consistentes, quer seja a partir do código fonte, ou através da adição de conhecimento

e experiência, que não poderiam ser automaticamente reconstruídos a partir do código [Benedusi92].

A engenharia Reversa atua no caminho contrário da engenharia Progressiva, que se define como àquela que segue a sequencia de desenvolvimento estabelecida por uma metodologia, visando à obtenção do sistema implementado. Na engenharia Reversa o sistema geralmente é o ponto inicial do processo [Chikofsky&Cross90].

Portanto utiliza-se a técnica de engenharia reversa, quando o produto existe juntamente com a necessidade realizar alterações de qualquer natureza, porém, não existem insumos suficientes que documentem a sua infraestrutura de forma a possibilitar o entendimento necessário para realizar as alterações.

Duas categorias dentro da engenharia reversa são apontadas por [Chikofsky&Cross90]: redocumentação e recuperação do projeto.

A **redocumentação** visa criar diferentes visões do sistema, em diferentes níveis de abstração através da análise do código fonte, para possibilitar maior compreensão do sistema. O desenvolvimento dessas visões pode gerar nova documentação (diferente da documentação preexistente), agregando mais conhecimento sobre o sistema.

A **recuperação** do projeto visa obter as informações necessárias para melhorar o entendimento sobre os objetivos do sistema, quais suas atividades e como ele as executa.

Por sua vez, a reengenharia de software também chamada de recuperação ou renovação, recupera informações de projeto de um software existente e usa essas informações para alterar ou reconstituir o sistema, preservando as funções existentes, ao mesmo tempo em que adiciona novas funções ao software, num esforço para melhorar sua qualidade global [Piekarski&Quinaia95].

Essa definição demonstra uma ligação direta entre a reengenharia de software e a engenharia reversa, uma vez que a reengenharia utiliza o conhecimento gerado a partir da aplicação da engenharia reversa para possibilitar as alterações no software durante a execução da reengenharia. Portanto existe distinção clara entre as duas técnicas, conforme destaca [Piekarski&Quinaia95]: “O objetivo da reengenharia é produzir um sistema novo com maior facilidade de manutenção e a engenharia reversa é usada como parte do processo de reengenharia, pois fornece o entendimento do sistema a ser reconstruído”.

Durante o trabalho no *Oryx*, foi recuperado o desenho da ferramenta a partir das descobertas possibilitadas pela engenharia reversa e concluída a reengenharia ao implementar um conjunto de *plugins* que possuem os elementos propostos neste trabalho. As próximas subseções detalham as atividades realizadas.

4.2.1. Aplicando a Engenharia Reversa

Como início da execução da engenharia reversa, foi definido um plano composto por 3 fases que, por sua vez, são compostas por um conjunto de atividades conforme descrito na Tabela 16:

Tabela 16 – Projeção de macroatividades da engenharia reversa da ferramenta *Oryx*

Fase 1	Fase 2	Fase 3
Identificar como obter o código fonte	Identificar documentação da ferramenta	Estudar o código fonte
Identificar os procedimentos para a instalação da base de codificação	Estudar a arquitetura da ferramenta	Identificar os conceitos arquiteturais na codificação
Configurar o ambiente de desenvolvimento	Identificar os procedimentos para desenvolvimento de *plugins*	Avaliar outras necessidades

Execução da Fase 1

A primeira fase do processo de engenharia reversa foi iniciada com uma pesquisa online para obter informações gerais sobre como proceder com a instalação do ambiente de desenvolvimento *Oryx*. Durante as buscas foram identificados muitos sites com o assunto "*Oryx*", porém o site oficial do projeto *Oryx* (http://code.google.com/p/*Oryx*-editor/) publicado no "*Google Codes*" ofereceu o conjunto de informações necessárias para realizar as atividades da primeira fase. Os procedimentos de instalação da ferramenta, acesso à codificação e configuração do ambiente de desenvolvimento são descritos em [Sousa&Leite12].

Após as execuções destas atividades, a fase 1 da engenharia reversa do software *Oryx* foi finalizada. A próxima subseção descreve a execução da segunda fase.

Execução da Fase 2

A segunda fase da engenharia reversa foi iniciada com a busca de documentos que descrevessem a arquitetura da ferramenta *Oryx*. Alguns documentos [Daniel07], [Peters07], [Tscheschner07], foram encontrados no site oficial *Oryx*-editor, no *Google Codes* (http://code.google.com/p/*Oryx*-editor/). Estes documentos são monografias desenvolvidas por alunos que estão envolvidos no grupo de pesquisa da ferramenta *Oryx*. Os trabalhos possuem um conteúdo extenso sobre a infraestrutura da ferramenta e o desenvolvimento de *plugins*, no entanto, todas as documentações estão em língua estrangeira como o inglês e o alemão.

Para solucionar parcialmente a dificuldade de leitura em alemão, os arquivos em PDF deveriam ser traduzidos. Então foi feita uma pesquisa para identificar ferramentas que realizassem esse trabalho. A maioria das ferramentas encontradas não traduziam arquivos com extensão PDF, e as que traduziam eram caras, além disso, as traduções em softwares gratuitos e demonstrativos eram limitadas em poucas linhas.

Outra busca foi realizada para verificar softwares de tradução de arquivos do tipo DOC, e diversos aplicativos foram encontrados. A partir disso foi iniciada outra busca por programas capazes de traduzir arquivos PDF para arquivos DOC. Vários aplicativos foram encontrados e a conversão foi feita, no entanto, o produto desta conversão deixou a desejar já que criou muitos caracteres aleatórios, deformando a formatação e afetando o documento, o que impossibilitou uma tradução minimamente satisfatória.

Novamente iniciou-se a busca por tradutores de PDF, e desta vez foi identificada uma ferramenta online de tradução baseada no tradutor do Google (*Google Translate* - http://translate.google.com.br/), disponível em http://www.onlinedoctranslator.com/translator.html. Este ferramenta traduziu o documento integralmente, mas também inseriu muitos erros, no entanto, foi a melhor solução descoberta e o produto da tradução era aceitável.

Com a leitura destes documentos foi possível obter o conjunto de informações necessárias sobre a arquitetura da ferramenta (Figura 51), compreender no nível teórico como é a organização de arquivos do *Oryx* e identificar os arquivos chaves para a construção de *plugins*. Essas informações descobertas a partir do estudo dos documentos identificados na pesquisa serão descritas a seguir, de forma resumida.

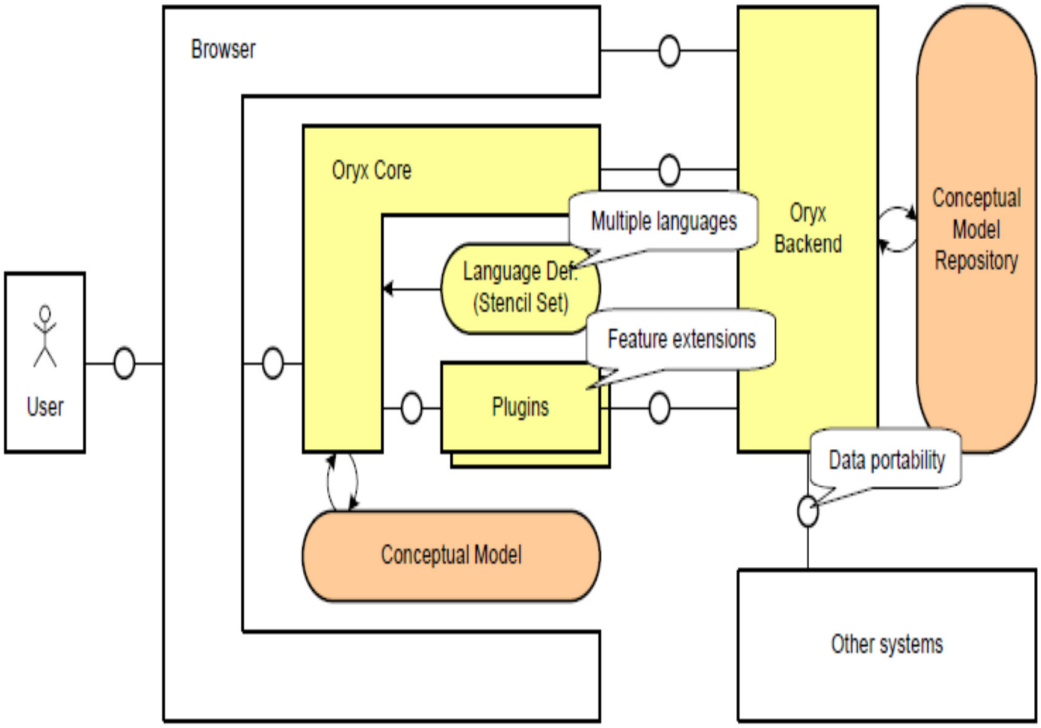


Figura 51 – Arquitetura da ferramenta *Oryx* [Daniel07]

A ferramenta tem um projeto típico "Cliente-Servidor" onde mantém a codificação principal sendo processada no lado do servidor, enquanto parte da codificação permanece no lado cliente. A codificação desenvolvida para o lado servidor não foi estudada neste trabalho, já que não é necessária para o desenvolvimento de *plugins*. A estrutura da ferramenta foi desenvolvida para ser capaz de interpretar arquivos que são compostos pela definição de elementos de modelagem e suas regras de interação. O desenvolvimento destes arquivos independe da complexidade do núcleo da ferramenta *Oryx*, o que demonstra a modularidade dos componentes "núcleo de sistema e *plugin*".

O foco do trabalho será na parte do cliente, mais especificamente no desenvolvimento dos "*Stencils*". Um *stencil* é a descrição de um objeto gráfico e suas regras, que definem como esse objeto irá se relacionar com os outros no diagrama. Um conjunto de *stencils* podem ser carregados para uso no editor *Oryx* para construir modelos [Peters07]. No entanto os *stencils* não serão carregados em separado, eles serão agrupados em um mesmo arquivo formando um *stencil set*.

Uma importante característica do *stencil* é que ele não define a representação gráfica dos elementos, mas somente as propriedades referentes ao comportamento do objeto no diagrama. O desenvolvimento da parte gráfica dos componentes será abordado mais a frente.

Os *stencils set* são armazenados em arquivos do tipo JSON (JavaScript Object Notation - http://JSON.org/). O arquivo JSON que define um *plugin* para o *Oryx* é composto por um cabeçalho, conjunto de *stencils* e conjunto de regras.

A Figura 52 exemplifica o código de um *stencil set*. Esse fragmento é composto por um título, que nomeia o *stencil set*, um *namespace* que é um identificador único, uma descrição, e o conjunto de *stencils* e regras.

```
{
  "title":"Workflow Nets",
  "namespace":"http://www.example.org/workflownets",
  "description":"Simple stencil set for Workflow Nets."
  "stencils": [/*...*/],
  "rules": {/*...*/}
}
```

Figura 52 – Fragmento de um *stencil set*

No interior da *tag* de *stencils* são inseridas as definições dos elementos de modelagem. Não há restrição de numero de *stencils* a serem inseridos no conjunto. A Figura 53 exemplifica o código de um *stencil*, presente dentro da *tag* "*Stencils*": [ ], ele inicia e termina entre as chaves ({ }).

O *stencil* é formado pelos atributos: *type*, que define qual é o tipo de elemento do *stencil*, por exemplo, um nó ou relacionamento; *id*, que é um identificador único para o elemento; *title*, que nomeia o objeto no diagrama; *description*, que define uma descrição do elemento que será mostrada no gráfico; *view*, que é o *path* para o arquivo gráfico que será desenhado no diagrama; *icon*, que é o *path* para o arquivo gráfico que possuirá o pequeno ícone que representará o elemento; *roles*, que especificam regras definidas que podem ser aplicadas a elementos específicos; *properties*, que oferecem ao usuário informações e campos para serem preenchidos referentes a propriedades do elemento. Não existe restrição ao numero de propriedades inseridas no *stencil*.

```
/*...*/
"stencils":
[
  {
    "type":"node",
    "id":"Activity",
    "title":"Activity",
    "description":"An atomic activity.",
    "view":"activity.svg",
    "icon":"activity.png",
    "roles": ["activitySource", "activityTarget"],
    "properties": [/*...*/]
  }/*,
  ...*/
]/*,
...*/
```

Figura 53 – Fragmento de um *stencil*

As propriedades de um *stencil* podem conter diversos parâmetros. Cada propriedade descrita irá gerar um campo no gráfico seja para permitir a inserção de texto, seleção de opção ou conteúdo pré-definido. A Figura 54 exemplifica o código das *properties* contendo apenas uma propriedade e seus parâmetros, limitados pelos "{ }".

```
"properties": [
  {
    "id":"caption",
    "type":"String",
    "title":"Caption",
    "value":"",
    "description":"",
    "readonly":false,
    "optional":true,
    "refToView":"caption",
    "wrapLines":true
  }/*,
  ...*/
]
```

Figura 54 – Fragmento das propriedades de um *stencil*

Por fim, as *rules* definem um conjunto de regras referente a relacionamentos entre os elementos no diagrama. A Figura 55 mostra um fragmento de código contendo 3 tipos de regras: *connectionRules*, *cardinalityRules* e *containmentRules*. Ainda existem outros tipos de regras, porém somente essas 3 serão detalhadas já que são as principais regras e os *stencils* set produzidos neste trabalho as utilizam predominantemente.

```
"rules":
{
  "connectionRules":  [/*...*/],
  "cardinalityRules": [/*...*/],
  "containmentRules": [/*...*/]
}
```

Figura 55 – Fragmento de código das regras de um *stencil set*

O tipo de regra "*connectionRules*" consiste em definir quais elementos podem ser interligados a partir de um relacionamento, ou seja, mais especificamente é a configuração de um determinado objeto do tipo relacionamento, onde define-se os elementos em que ele pode interligar.

A Figura 56 apresenta um fragmento de código que define regras para relacionamento entre elementos. Neste exemplo, o elemento de relacionamento "*controlflow*" está configurado com 2 regras de conexão. A primeira define que pode existir uma conexão saindo do elemento "*activitySource*" em direção ao elemento "*conditionTarget*". A segunda regra define a conexão saindo do elemento "*conditionSource*" para o elemento "*activityTarget*".

```
"connectionRules": [
  {
    "role":"controlflow",
    "connects": [
      {
        "from":"activitySource",
        "to":"conditionTarget"
      },
      {
        "from":"conditionSource",
        "to":"activityTarget"
      }
    ]
  }
]
```

Figura 56 – Fragmento de código das regras de conexão

O tipo de regra "*cardinalityRules*" define o número de conexões de entrada/saída que um elemento pode possuir. A Figura 57 exemplifica um fragmento contendo 2 regras de cardinalidade. Essa regra não está definida para um elemento específico como descrito nas regras de conexão, exemplificada anteriormente, mas define um papel (*role*) que deve ser configurado no *stencil* para adquirir a regra. No exemplo, 2 roles são definidas: a *inputcondition* e a *outputcondition*. A *role inputcondition* define que o objeto o qual está configurado para obedecer esta regra deverá obrigatoriamente ter um elemento de relacionamento "entrando". A *role outputcondition* também especifica que é obrigatória 1 relacionamento de saída do elemento.

```
"cardinalityRules": [
  {
    "role":"inputcondition",
    "minimumOccurrence":1,
    "maximumOccurrence":1
  },
  {
    "role":"outputcondition",
    "minimumOccurence":1,
    "maximumOccurrence":1
  },
]
```

Figura 57 – Fragmento de código de regras de cardinalidade

O tipo de regra "*containmentRules*" são regras que definem quais elementos podem estar contidos dentro do elemento configurado. Esta regra é mais adequada a raias (pools) onde os elementos são inseridos. A Figura 58 exemplifica um fragmento de código contendo regras de conteúdo. Esta regra cria uma *role "diagram*" que contém dos elementos que possuem o id "*activity*", "*condition*" e as regras definidas de conexão definidas anteriormente "*inputcondition*" e "*outputcondition*".

```
"containmentRules": [
  {
   "role":"diagram",
   "contains": [
     "activity",
     "condition",
     "inputcondition",
     "outputcondition"
   ]
  }
]
```

Figura 58 – Fragmento de código de regras de conteúdo

Uma vez definido o *stencil set*, todas as informações e regras sobre os elementos estarão disponíveis para interpretação do *Oryx*. O próximo passo é definir os elementos gráficos que representarão os *stencils* definidos no *stencil set*. Dois elementos gráficos são necessários para representação no modelo: o ícone e o objeto gráfico.

O ícone é uma versão reduzida do objeto gráfico que serve para a ferramenta apresentar ao usuário para que ele selecione no painel e inclua no diagrama. Os ícones devem estar no formato de arquivo PNG com dimensões de 16 x 16 pixels. Essas figuras podem ser facilmente desenvolvidas utilizando o *MsPaint*, no *Windows*.

O elemento gráfico que representa o objeto que será desenhado no diagrama é definido a partir do formato SVG (*Scalable Vetor Graphics*).

Neste formato os desenhos são criados a partir de códigos do padrão SVG. Existem vários comandos que podem ser utilizados para gerar diversos tipos de grafos. Esses comandos podem ser estudados no site do W3C (http://www.w3.org/Graphics/SVG/). Existem vários tipos de arquivo SVG, por exemplo, é possível criar um arquivo SVG a partir de outras figuras existentes, criando apenas uma ancora para o outro arquivo gráfico, no entanto, este tipo de arquivo SVG não funciona corretamente no *Oryx*, sendo recomendado apenas o uso dos arquivos SVG descritivos, em que as imagens são geradas a partir da especificação de código na linguagem oferecida pelo padrão.

Além da codificação do padrão SVG, devem ser inseridos no arquivo outras *tags* que serão utilizadas pelo *Oryx*. Por exemplo, devem ser definidos no arquivo SVG a localização do campo de texto do elemento, cor e tamanho da fonte e outros elementos, como os *Oryx Magnets*, que definem pontos no desenho onde os relacionamentos podem se ligar ao objeto.

A Figura 59 demonstra um exemplo de codificação de arquivo SVG contendo *tags* do *Oryx*. Neste exemplo, nota-se que o cabeçalho possui a definição do *namespace* do *Oryx*, necessário para reconhecer as *tags*: “xmlns:*Oryx*="http://www.b3mn.org/*Oryx*".

A *tag* <*Oryx*:magnets> demarca o inicio da configuração dos pontos que estarão disponíveis no objeto para a ligação de relacionamentos. Cada ponto é definido pela sua coordenada x e y.

Entre as *tags* <g> </g> estão configurados os elementos que definirão o visual gráfico do elemento. A *tag* <radialGradient> </radialGradient> define um fundo para ser utilizado no objeto.

Esta imagem é composta por uma reta <rect> que está ancorada ao elemento (*Oryx*:anchors) pelo topo, esquerda e direita, o que significa que ao ampliar o objeto no diagrama, a reta crescerá para estas direções, exceto para baixo. O mesmo ocorre para o círculo <circle> que também compõem a imagem. Na *tag* <text> é definido o campo para inserção de texto no elemento, incluindo o tamanho da fonte e sua coordenada x, y e seu alinhamento (align) em relação ao objeto. O texto é encaixado na reta que é desenhada no gráfico (fittoelem).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
   xmlns="http://www.w3.org/2000/svg"
   xmlns:oryx="http://www.b3mn.org/oryx"
   width="100"
   height="100"
   version="1.0">

  <oryx:magnets>
    <oryx:magnet oryx:cx="0" oryx:cy="60" oryx:anchors="left" />
    <oryx:magnet oryx:cx="75" oryx:cy="80" oryx:anchors="bottom" />
    <oryx:magnet oryx:cx="100" oryx:cy="60" oryx:anchors="right" />
    <oryx:magnet oryx:cx="75" oryx:cy="0" oryx:anchors="top" />
    <oryx:magnet oryx:cx="50" oryx:cy="40" oryx:default="yes" />
  </oryx:magnets>

  <g pointer-events="fill">

    <defs>
        <radialGradient id="background" cx="50%" cy="50%" r="100%" fx="50%" fy="50%">
            <stop offset="0%" stop-color="lightgray" stop-opacity="1"/>
            <stop id="fill_el" offset="100%" stop-color="lightgray" stop-opacity="1"/>
        </radialGradient>
    </defs>

    <rect id="text_frame" oryx:anchors="bottom top right left" x="25" y="25"
    width="70" height="50" rx="10" ry="10" stroke="black" stroke-width="0" fill="none" />

    <circle id="bg_frame" oryx:resize="vertical horizontal" cx="50" cy="40" r="45"
    stroke="black" fill="url(#background) white" stroke-width="2"/>

    <text font-size="12"
        id="text_name"
        x="50" y="40"
        oryx:align="middle center"
        oryx:fittoelem="text_frame"
        stroke="black">
    </text>
  </g>
</svg>
```

Figura 59 – Exemplo de código de imagem SVG

Em suma, para desenvolver o *plugin* desejado, é necessário definir o *stencil set* contendo todos os elementos que se deseja modelar e suas regras, e os ícones e arquivos SVG referentes a cada um dos elementos. Esse conhecimento é suficiente para finalizar a Fase 2 (Tabela 16) do processo de engenharia reversa.

Na próxima subseção, são descritos os procedimentos realizados na terceira fase do processo de engenharia reversa onde foram estudados os conceitos descobertos na segunda fase diretamente na codificação da ferramenta.

Execução da Fase 3

Na terceira fase do processo de engenharia reversa, o alvo de estudo é a codificação e o ambiente de programação do *Oryx*. Grande parte do código já foi estudada nos documentos, na segunda fase, porém neste momento todos os conceitos serão revistos de forma prática nos arquivos do projeto do *Oryx*.

Dentre o conjunto de pastas e arquivos do projeto do *Oryx*, estão codificações referentes ao lado servidor e cliente da aplicação. Como dito antes, o estudo realizado tem foco no lado do cliente. Os arquivos dos stencils, os quais serão de fato editados, estão presentes no seguinte path: "*Oryx*/editor/data/stencilsets" (Figura 60).

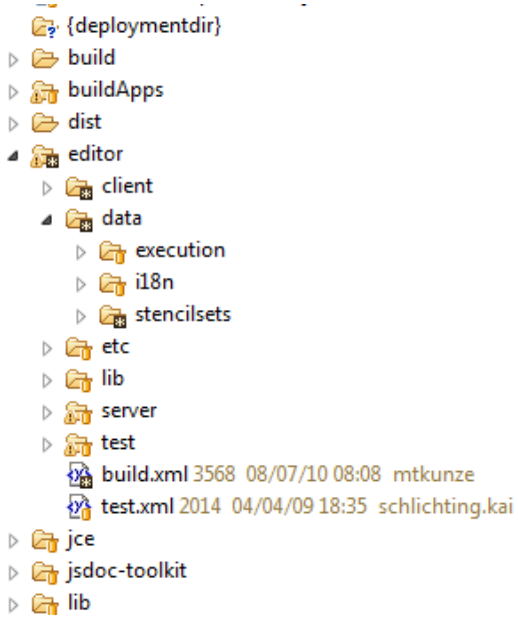


Figura 60 – Infraestrutura de pastas do ambiente de desenvolvimento do *Oryx*

Abrindo a pasta *stencilset* (Figura 61) é possível encontrar um grande conjunto de *stencils* presentes nas subpastas e também o arquivo "*stencils*ets.json". Neste arquivo são registrados todos os *stencils sets* presentes na pasta *stencilsets*. Portanto é necessário registrar o novo *stencil* neste arquivo. A Figura 62 exemplifica o registro de um *stencil set* no arquivo *stencilsets.json*. As informações que compõem o registro são título do *stencil* (*title)*, *namespace,* descrição (*description)*, *path* onde está o arquivo JSON do *stencilset* que se deseja registrar (*uri)*, *path* do ícone para representar o *stencil set* (*icon_url*) e a opção que torna o *stencil set* visível na ferramenta ou não (*visible).*

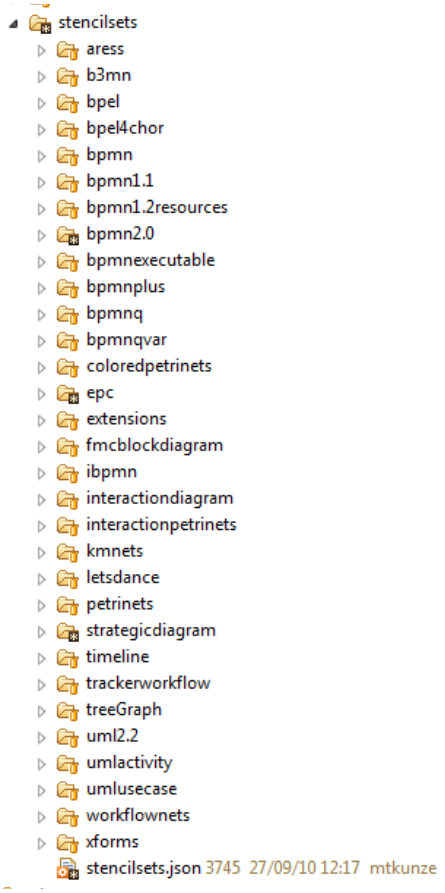


Figura 61 – Detalhamento da pasta *stencilsets*

```
{
    "title":"EPC",
    "namespace":"http://b3mn.org/stencilset/epc#",
    "description":"Event-driven Process Chain specification.",
    "uri":"/epc/epc.json",
    "icon_url":"/epc/epc.png",
    "visible":true
},
```

Figura 62 – Exemplo de registro de *stencil set*

Ao entrar em uma pasta de um *stencil set*, identifica-se uma estrutura comum a todos as pastas dos *stencils set* que é composta por uma subpasta dedicada aos ícones dos elementos (*icons)*, uma subpasta dedicada aos arquivos SVG que representam os elementos no modelo (*view),* o arquivo JSON que contem todas as configurações dos *stencils* e suas regras, e opcionalmente um arquivo do tipo PNG contendo o ícone que representa o *stencil set*.

Figura 63 – Estrutura de pastas do *stencil set* "*petrinets*"

É justamente essa estrutura que foi implementado neste projeto a criação dos novos *stencils set*. Todo o conhecimento sobre a infraestrutura da ferramenta *Oryx* necessário para a continuação do projeto foi identificado, finalizando a atividade de engenharia reversa.

A partir do conhecimento extraído com a aplicação da engenharia reversa, foi possível modelar um diagrama conceitual, representado na Figura 64. Neste diagrama estão contidos os elementos básicos de um *stencil set*, conforme apresentado nos capítulos anteriores. O *stencil set* pode conter zero ou mais *stencils* e *rules*. Um *stencil* contém zero ou mais *properties* e *roles*. Uma *rule* pode ser de 3 tipos: C*onnectionRules, CardinalityRules* e *ContainmentRules.* Cada *rule* define uma *role* que pode ser referenciada pelo *stencil* que passa a respeitar a regra.

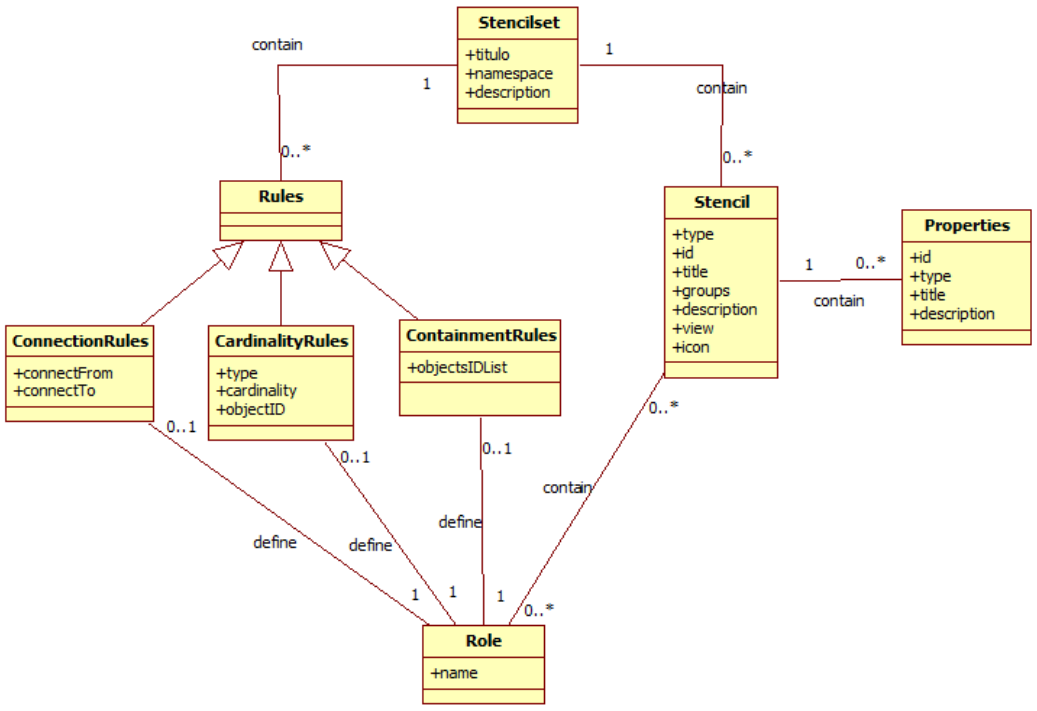


Figura 64 – Diagrama conceitual contendo elementos básicos de um *stencil set*

Com esse conhecimento adquirido, é possível descrever uma lista de requisitos mais detalhada, por exemplo, para poder implementar na ferramenta a linguagem do *framework* i* o *check list* de requisitos poderia ser*:*

- *Criar plugin contendo i**
    - *Criar elementos SVG com formato gráfico compatível ao i**
    - *Criar ícones para cada elemento SVG*
    - *Implementar o arquivo stencil set*
        - *Criar stencil para cada elemento do i**
        - *Criar regras de acordo com as regras do i**
        - *Aplicar as regras aos elementos de acordo com as regras do i**

Utilizando os novos conhecimentos que possibilitaram visualizar as atividades necessárias para poder implementar as novas demandas na ferramenta *Oryx* (i* alinhado à BPMN), é possível aplicar a reengenharia reversa na ferramenta *Oryx* para customizá-la inserindo novo código, alterando e reutilizando a codificação existente. A próxima seção descreve como foi implementados os novos requisitos e as novas alterações incluídas após gerar a versão 1.0 da ferramenta.

4.2.2. Reengenharia de software

Esta seção apresenta como foi executado o processo de reengenharia da ferramenta *Oryx.* A estrutura de pastas da ferramenta é explicada, mostrando onde se encontram as codificações no ambiente de desenvolvimento, além disso, são apresentados diagramas que demonstram as diferentes definições do *Oryx* e como os seus arquivos se relacionam.

Este capítulo possui um enfoque na apresentação dos conceitos e infraestrutura do ambiente e seus arquivos. Na fase de reengenharia, o objetivo era desenvolver os *plugins* do i* e da BPMN de forma a permitir a modelagem conjunta de seus elementos. No entanto, o código do *Oryx* disponibilizado para download já possui uma implementação do BPMN 2.0, permitindo o seu reuso.

Retornando ao assunto da codificação, conforme aprendido nos estudos anteriores, é possível definir as atividades necessárias para alcançar os objetivos desta fase. O reuso do *stencil set* BPMN irá reduzir bastante trabalho, sendo apenas necessário incluir os novos *stencils* do i* e suas regras, além de definir os elementos gráficos e realizar as alterações necessárias para adaptar o i* ao *stencil set* do BPMN.

O *stencil set* BPMN + i* foi projetado conforme o esquema apresentado na Figura 65. A maior dificuldade encontrada na implementação foi a ausência de ferramenta de *debug*, já que não há como acompanhar a execução do código de um *stencil set*, e um pequeno erro de uma virgula ausente já faz com que a ferramenta não funcione corretamente.

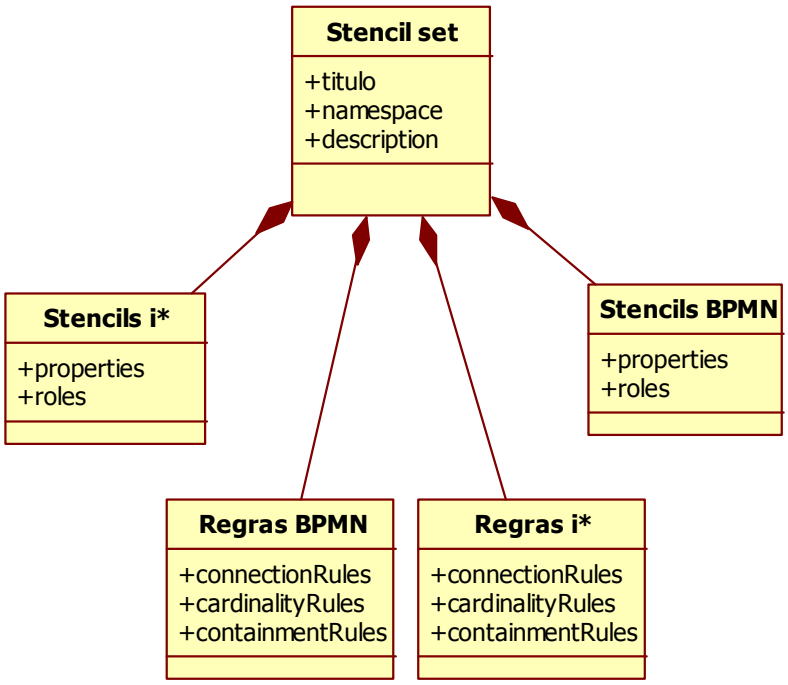


Figura 65 – Esquema de formação do *Stencil set* BPMN + i*

Diversas vezes foi necessário apagar grande parte da codificação desenvolvida e reinserir novamente por partes e realizando testes de execução em paralelo para garantir que nada estivesse faltando. Neste tipo de programação, onde se trabalha com um arquivo simples de *tags* e sem um sistema de *debug*, não é seguro aplicar um grande número de modificações e posteriormente realizar testes, já que um erro inserido em algum ponto do código (mesmo que sem querer) será dificilmente rastreado e identificado.

Os sintomas que ocorrem por problemas de codificação são os mesmos para qualquer problema inserido no *stencil set*. O sistema não apresenta os elementos do *stencil set* no ambiente *Oryx* ou a ferramenta simplesmente não carrega integralmente, independente de carregar os elementos do *stencil set*.

Após a finalização do código, os stencils sofreram novas alterações para oferecer ao usuário a possibilidade de trabalhar nos diagramas separadamente, por exemplo, atuar no modelo SD ou SR e utilizar unicamente a BPMN. Na parte de integração das linguagens, incluir as visões geradas a partir da união da BPMN com SD e da BPMN com SR.

Para implementar esses alterações, foram extraídos todos os elementos do *stencil set* implementado até o momento. Cada diagrama (i* SD, i* SR e BPMN) foi transformado em um único *stencil set* e foram criadas “perspectivas” na ferramenta para que o usuário possa selecionar e facilmente trocar entre os diferentes *plugins*.

A separação dos novos *stencil sets* não trazem nenhuma novidade ao trabalho, a não ser pela mudança do esquema de implementação (Figura 65), no entanto, a criação de perspectivas demandou alterações em um arquivo que define as perspectivas na ferramenta e na estrutura de pastas.

O *Oryx* possui um arquivo chamado *extension.json* (Figura 66), onde são configurados *stencil sets* que estendem a capacidade de outros *stencil set* considerados como principais. Estas extensões são aplicadas ao modelo quando o usuário solicita a inclusão do *plugin* manualmente ou quando seleciona a perspectiva predefinida que contém *plugins* de extensão.

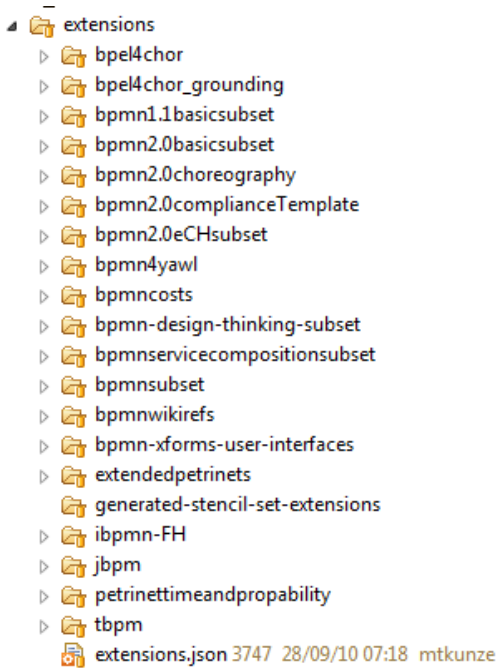


Figura 66 – Pasta "*extensions*" contendo o arquivo "*extensions.json*"

Esta estrutura de extensões foi utilizada como estratégia para a implementação das visões. Um *stencil set* genérico foi desenvolvido contendo o *stencil* básico do diagrama e o conjunto de regras genéricas que são usadas pelos *stencils set* que irão estendê-lo. Esse *stencil set* principal é iniciado na ferramenta, e as perspectivas definem os elementos que vão ser apresentados neste *stencil set* principal. Portanto cada *stencil set* implementado a partir da separação mencionada anteriormente (i* SD, i* SR e BPMN) foi configurado como uma extensão. A Figura 67 ilustra o esquema de formação do *stencil set* principal i*BPMN e seu relacionamento com suas extensões.

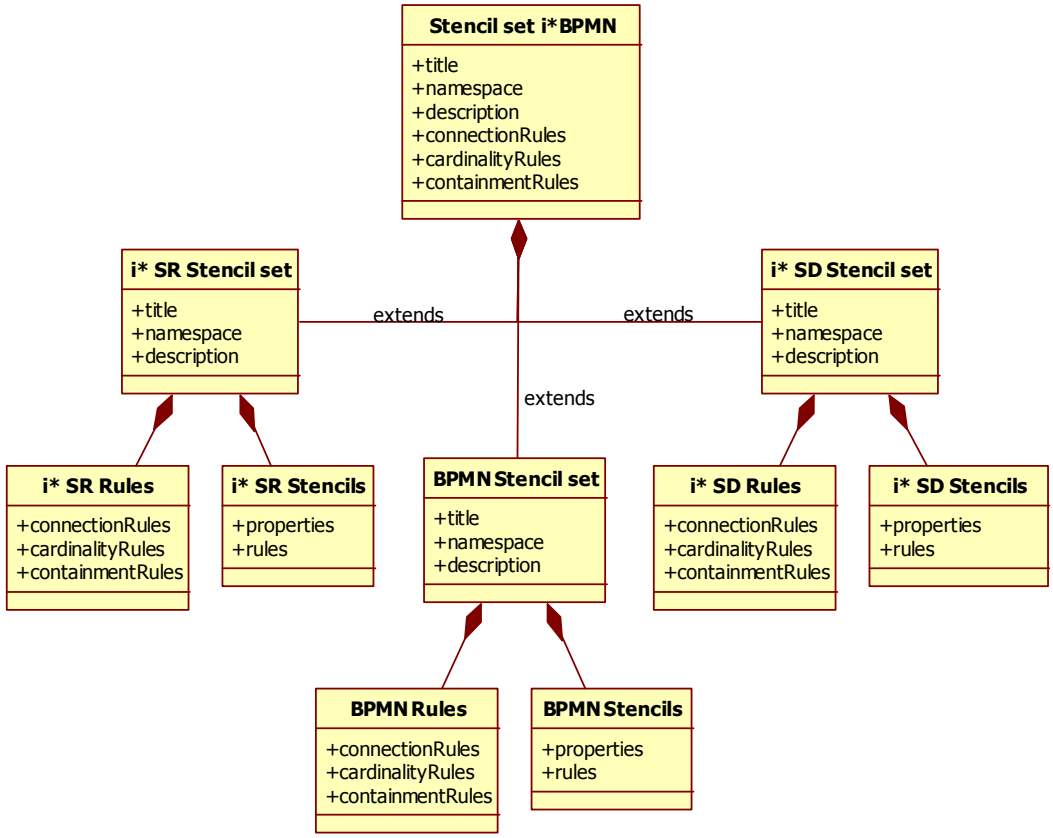


Figura 67 – Esquema de formação do *stencil set* i*BPMN a partir de extensões

O arquivo *extension.json* é formado pelo registro do conjunto de *stencils set* que são utilizados como extensão e o registro das perspectivas. O registro de um *stencil set* é formado por titulo, *namespace*, descrição, definição (*path* para o *stencil set* da extensão) e o campo de extensão, onde se define o *path* em que se encontra o *stencil set* a ser estendido. O registro de uma perspectiva é composto por titulo, *namespace*, descrição, *stencil set* que vai ser estendido e o campo onde se registra quais *stencils set* serão incluídos na perspectiva e outro campo onde se registra quais *stencils set* serão removidos da perspectiva (Figura 68).

```
{
    "extensions":
    [
        {
            "title":"iStar SD Extension",
            "namespace":"http://oryx-editor.org/stencilsets/extensions/iStar_SD_extension#",
            "description":"Include in the diagram the iStar SD elements",
            "definition":"iStar_SD_extension/iStar_SD_extension.json",
            "extends":"http://b3mn.org/stencilset/iStarBPMN#"
        }
    ],
    "perspectives":
    [
        {
            "title":"iStar SD",
            "namespace":"http://oryx-editor.org/stencilsets/perspectives/iStarSD#",
            "description":"iStar SD perspective. Features the complete standard.",
            "stencilset":"http://b3mn.org/stencilset/iStarBPMN#",
            "removeExtensions" : [ ],
            "addExtensions" : [ ]
        }
    ]
}
```

Figura 68 – Exemplo do arquivo *extension.json*

Com essas alterações também se finaliza a fase de implementação dos plugins realizada através da reengenharia de software. Uma vez que a codificação foi desenvolvida, é natural que a próxima etapa seja referente aos testes e correções. A próxima seção apresenta como foram feitos os testes a partir de um roteiro predefinido e apresenta alguns exemplos do que foi realizado.

4.3.Testes

A codificação desenvolvida neste trabalho é um conjunto de especificações que são interpretadas pelo núcleo da ferramenta *Oryx* e geram um resultado gráfico em seu ambiente de modelagem. Para testar a codificação desenvolvida, foi escolhido o método de testes assistido, baseado em um roteiro onde são pré-registrados os resultados esperados por cada funcionalidade a ser testada. Uma vez que os resultados esperados são atingidos através do uso da funcionalidade, a função é considerada satisfatória. Caso contrário, é identificado o defeito que deve ser consertado e novamente testado ou registrada a existência do problema, caso seja avaliado e justificado que o código não deve ser alterado neste caso.

Os testes foram projetados a partir de roteiros onde se definem especificações que devem ser integralmente cumpridas ao utilizar a função na ferramenta. Os elementos alvo dos testes foram cada objeto e cada regra aplicada ao objeto. Aqui descreveremos apenas o roteiro de testes aplicados e alguns exemplos de testes, os resultados mais detalhados se encontram em [Sousa&Leite12].

Como um *plugin* é formado por stencils e regras, basicamente serão estes elementos que serão testados. A especificação dos elementos é repassada para a

codificação e o resultado esperado é o elemento gráfico e regras de acordo com o esperado.

Para cada *stencil* foram testados:

- No caso de ser um nó, o elemento gráfico esperado, localização do texto no objeto, efeito de ampliação do objeto, regras específicas.
- No caso de ser um relacionamento, o elemento gráfico esperado, localização do texto no objeto, efeito de ampliação do objeto, regras específicas e regras de conexão.

Nem todos os testes são aplicáveis a todos os stencils e regras, sendo neste caso, marcado como não aplicável. A sintaxe geral do programa é automaticamente verificada pelo *Oryx*, que somente abre o *stencil set* para modelagem se todas as regras de sintaxe estiverem corretas.

Os testes foram divididos em 2 tipos, chamados de Teste gráfico e Teste de regras, sendo que este último ainda possui variações, já que as regras foram testadas separadamente para cada situação esperada por um objeto em relação aos outros elementos que fazem parte do domínio da regra

O objetivo do teste gráfico (Tabela 17) é verificar se o elemento está de acordo com as especificações esperadas. A verificação de aceite do teste é visual e comparativa. A especificação padrão dos elementos é composta dos seguintes campos: Elemento – nome do elemento na ferramenta; Formato – formato gráfico do elemento; Cor – cor do corpo do elemento; Texto – Configuração do texto em relação ao objeto; O campo Resultado, mostra a figura que é resultante do código que foi implementado. Esta figura deverá ser comparada com a especificação para verificar se o objeto obedece a todas as especificações, o que caracteriza um teste com sucesso. Uma vez que o elemento passou no teste, o campo Teste é preenchido com “OK”.

Tabela 17 – Teste gráfico

Teste gráfico	
Especificação	**Resultado**
Elemento:	
Formato:	
Cor:	
Texto:	
Teste:	

O teste de regras do tipo *Containment Rules* (Tabela 18) possuem os seguintes parâmetros: elemento – nome do elemento o qual a regra está atribuída; conteúdo – os elementos que podem encontrar-se modelados no corpo do elemento que deve obedecer a regra.

Tabela 18 – Teste de regras do tipo *Containment Rules*

Teste de regras (*Containment Rules*)	
Especificação	**Resultado**
Elemento:	
Conteúdo:	
Teste:	

O Teste de regras do tipo *Cardinality Rules* (Tabela 19) possuem os seguintes parâmetros: elemento – nome do elemento o qual a regra está atribuída; elemento de entrada – nome do relacionamento que chega ao elemento; Card. Entrada – Cardinalidade do elemento de entrada; Elemento de saída – nome do relacionamento que sai do elemento; Card. Saída – Cardinalidade do elemento de saída.

Tabela 19 – Teste de regras do tipo *Cardinality Rules*

Teste de regras (*Cardinality Rules*)	
Especificação	**Resultado**
Elemento:	
Elemento de entrada	
Card. Entrada:	
Elemento de saída:	
Card. Saída:	
Teste:	

O teste de regras do tipo *Connection Rules* (Tabela 20) possui os seguintes parâmetros: Elemento – nome do relacionamento que será avaliado; Elem. Inicial – nome do elemento que deve estar na ponta inicial do relacionamento; Elem. Final – nome do elemento que deve estar na ponta final do relacionamento.

Tabela 20 - Teste de regras do tipo *Connection Rules*

Teste de regras (Connection Rules)	
Especificação	**Resultado**
Elemento:	
Elem. Inicial:	
Elem. Final:	
Teste:	

Testes de "negação" das regras não serão aplicados já que o *Oryx* automaticamente impede a execução de regras que não estão explicitas. Por exemplo, se definir somente uma regra que relaciona o objeto A ao objeto B nesta direção, a partir do relacionamento X, não é necessário testar a conexão para a direção inversa porque ela não acontecerá, já que não está definida.

A Tabela 21 e Tabela 22 apresentam dois exemplos de teste gráfico, neste caso, para elementos do i*. A tabela é dividida em duas partes, na esquerda, encontram-se as especificações do elemento, do lado direito, o elemento modelado. O teste consiste em uma avaliação visual do elemento que, uma vez modelado, apenas deve corresponder às especificações. Caso corresponda, o teste é concluído de forma positiva.

Tabela 21 - Teste gráfico para o elemento Agent (Diagrama SD)

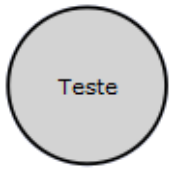
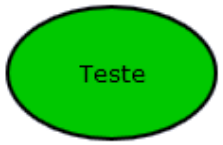
Teste gráfico	
Especificação	**Resultado**
Elemento: Agent	
Formato: circular	
Cor: cinza	
Texto: centralizado	
Teste: OK	

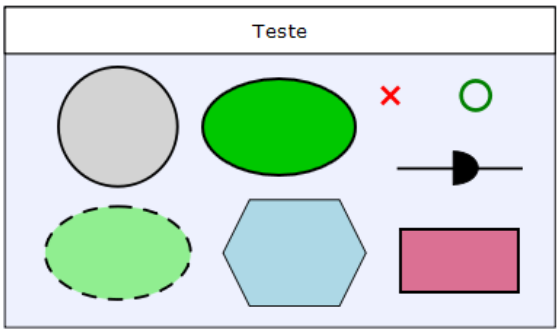
Tabela 22 – Teste gráfico para o elemento meta (Diagrama SD)

Teste gráfico	
Especificação	**Resultado**
Elemento: Meta	
Formato: oval	
Cor: verde	
Texto: centralizado	
Teste: OK	

A Tabela 23 apresenta o teste para a regra *Containment Rule*. A configuração desta regra permite que elementos sejam modelados dentro de outros elementos, o qual se torna seu limite de mobilidade. Por exemplo, podemos citar o próprio diagrama principal que recebe os elementos de modelagem. O diagrama deve estar configurado nesta regra para todos os elementos de linguagem. No teste apresentado, foi testado o elemento *AgentLane*, que é uma *Lane* que representa um papel/agente. De acordo com a especificação da tabela, os elementos que devem ser modelados dentro da *lane* estão descritos no campo "Conteúdo". Os elementos são: *Agent,* Meta, Meta-flexível*, Task, Resource, Critical Mark, Open Mark, Dependency*. O teste consiste em inserir estes elementos na *lane.* Uma vez que a ferramenta permite o desenho dos elementos esperados, o teste foi realizado com sucesso.

Observando o resultado do teste, verifica-se que todos os elementos esperados foram modelados corretamente no interior da *lane*, validando o resultado.

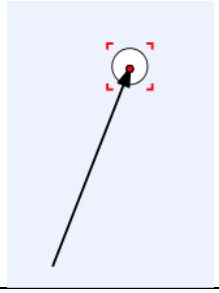
Tabela 23 – Teste da regra *Containment Rule* para o elemento AgentLane (Diagrama SD)

Teste de regras (*Containment Rules*)	
Especificação	**Resultado**
Elemento: AgentLane	
Conteúdo: Agent, Meta, Meta-flexível, Task, Resource, Critical Mark, Open Mark, Dependency.	
Teste: OK	

A Tabela 24 apresenta o teste para a regra *Cardinality Rule*. Esta regra define quantos relacionamentos de um dado tipo um elemento pode ter como entrada/saída. No exemplo, o relacionamento de fluxo de sequência foi testado para os eventos do

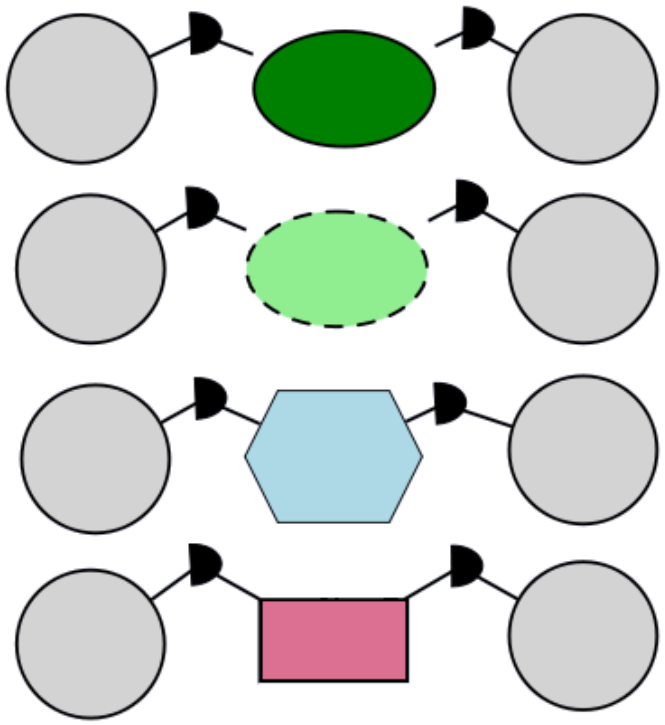
tipo inicial. A cardinalidade de um evento inicial é zero, uma vez que só o processo inicia-se com ele. Portanto, para este teste, tenta-se ligar um relacionamento em direção ao evento inicial, e espera-se que a ferramenta recuse a ligação, conforme mostra o resultado do teste. A ferramenta apresenta sinais em vermelho demonstrando que a ligação não é permitida. Isso demonstra que a restrição da regra está funcionando conforme o esperado.

Tabela 24 - Teste da regra *Cardinality Rule* para os elementos do tipo Evento inicial

Teste de regras (*Cardinality Rules*)	
Especificação	**Resultado**
Elemento: Regra genérica (Eventos iniciais) Elemento de entrada: SequenceFlow Card. Entrada: 0 Elemento de saída: N/A Card. Saída: N/A	

A Tabela 25 apresenta o teste para a regra *Connection Rule.* Esta regra define quais objetos podem se relacionar. No exemplo, o teste é para o relacionamento de dependência em um diagrama SD. A especificação apresenta o elemento inicial e o elemento final, ou seja, também define a direção do relacionamento. O resultado deve apresentar os elementos relacionados, seguindo a direção correta no relacionamento.

Tabela 25 - Teste da regra *Connection Rule* para os elementos que utilizam o relacionamento *Dependency* (Diagrama SD)

Teste de regras (Connection Rules)	
Especificação	**Resultado**
Elemento: Dependecy	
Elem. Inicial: Agent	
Elem. Final: Meta	
Elem. Inicial: Agent	
Elem. Final: Meta-flexível	
Elem. Inicial: Agent	
Elem. Final: Task	
Elem. Inicial: Agent	
Elem. Final: Resource	
Elem. Inicial: Meta	
Elem. Final: Agent	
Elem. Inicial: Meta-flexível	
Elem. Final: Agent	
Elem. Inicial: Task	
Elem. Final: Agent	
Elem. Inicial: Resource	
Elem. Final: Agent	

Esta seção apresentou o roteiro de testes que foram aplicados na ferramenta e exemplificou cada tipo de teste aplicado para as diferentes regras que os elementos podem receber. Os testes completos, incluindo a listagem de erros presentes na ferramenta (e não nos *plugins* desenvolvidos), as dificuldades encontradas e o manual da ferramenta podem ser vistos em [Sousa&Leite12]. Os resultados dos testes foram satisfatórios resultando na possibilidade de uso integral das linguagens como extensão da ferramenta *Oryx*. A próxima seção apresenta o estudo de caso aplicado à proposta deste trabalho.