Pós-Graduação em Ciência da Computação

TRANSFORMANDO MODELOS SCADE EM ESPECIFICAÇÕES SCR

Por

KAMILA NAYANA CARVALHO SERAFIM

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2016

Kamila Nayana Carvalho Serafim

Transformando modelos SCADE em especificações SCR

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Universidade Federal de Pernambuco – UFPE

Programa de Pós-Graduação

Orientador: Prof. Dr. Márcio Lopes Cornélio Centro de Informática/UFPE

Coorientador: Prof. Dr. Alexandre Cabral Mota Centro de Informática/UFPE

Recife

2016

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

S481t Serafim, Kamila Nayana Carvalho.
Transformando modelos Scade em especificações SCR / Kamila Nayana Carvalho Serafim – 2016.
59f.: fig., tab.

Orientador: Márcio Lopes Cornélio.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, Recife, 2016.
Inclui referências.

1. Métodos formais. 2. Especificação formal. 3. Certificação.. I. Cornélio, Márcio Lopes (Orientador). II. Titulo.

005.1 CDD (22. ed.) UFPE-MEI 2017-004

Kamila Nayana Carvalho Serafim

Transformando modelos SCADE em especificações SCR

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em: 08/09/2016

Banca examinadora

Prof. Dr. Juliano Manabu Iyoda Centro de Informática/UFPE

Prof. Dr. Adalberto Cajueiro de Farias
Departamento de Sistemas e Computação/UFCG

Prof. Dr. Márcio Lopes Cornélio
Centro de Informática/UFPE
Orientador

Eu dedico esta dissertação a minha mãe e ao meu noivo que sempre me opiaram em qualquer decisão que eu tomasse.

Agradecimentos

Agradeço a minha mãe, que sempre deu duro para que eu chegasse até aqui, ao meu noivo que fez tudo isso acontecer e a toda minha família por sempre me apoiarem. Meu orientador por ter me mostrado o caminho das pedras e ter me guiado em todo o tempo de mestrado.

O homem não é nada além daquilo que a educação faz dele. (Immanuel Kant)

Resumo

A construção de um software para domínios particulares tem de atender normas específicas que impõem o atendimento a fatores como rastreabilidade de requisitos e certificação. Por exemplo, a indústria aeronáutica deve atender à norma DO-178B que estabelece restrições para uso de software de aeronaves, que são considerados sistemas críticos. Para um sistema estar de acordo com essa certificação é necessário ter requisitos formais e código certificado; nesta direção, Andrade (ANDRADE, 2013) usou a notação SCR (Software Cost Reduction) para definição de requisitos e a ferramenta SCADE para modelagem de sistemas críticos, com desenvolvimento de um tradutor de SCR para artefatos xscade. A prática de desenvolvimento de sistema, porém, não está restrita à transição entre requisitos e artefatos de projeto. Modificações realizadas nestes últimos devem também ser refletidas nos requisitos. Neste trabalho desenvolvemos um tradutor de artefatos de modelagem da ferramenta SCADE para SCR. Desta forma podemos gerar especificação de requisitos a partir do código (Engenharia Reversa) e complementamos o trabalho anterior desenvolvido por Andrade (ANDRADE, 2013). Para o desenvolvimento do tradutor, utilizamos a plataforma Spoofax por meio da qual descrevemos a sintaxe do esquema XML utilizado em SCADE e também as regras de tradução tendo como alvo SCR. A validação da tradução teve como ponto de partida o resultado do uso do tradutor desenvolvido por Andrade (ANDRADE, 2013), tendo de gerar como saída a mesma entrada do tradutor desenvolvido por Andrade (ANDRADE, 2013). Além disso, desenvolvemos exemplos para demonstrar que a modificação estrutural, com preservação de semântica, em projetos SCADE, é verificável por meio do uso de testes gerados por meio da ferramenta TTM-TVEC.

Palavras-chave: Métodos Formais. Requisitos Formais. Certificação. Regras de Tradução. Testes.

Abstract

Building a software for particular domains must attend specific standards that impose attendance to factors such as traceability requirements and the certification issue. For example, the airline industry should meet the DO-178B standard that establishes restrictions on the use of aircraft software, which is considered a critical system. For a system to be in accordance with this certification, one must have formal requirements and certified code. In this direction, Andrade (ANDRADE, 2013) used SCR (Software Cost Reduction) for requirements definition and SCADE for modeling critical systems with development of an artifacts a translator from SCR. However the practice of developing is not restricted to the transition from requirements to design artifacts. Changes made on design should be reflected in the requirements. In this work we developed a translator from SCADE to SCR. In this way we can generate requirements specification from the code (reverse engineering) and complement the previous Andrade (ANDRADE, 2013) thesis. For the translator development, we use the Spoofax platform through which we describe the XML schema syntax used in SCADE and also the translation rules having SCR as the target language. The translation validation had as its starting point the result of the translator developed by Andrade (ANDRADE, 2013), where the output is the same input developed by Andrade(ANDRADE, 2013). Furthermore, examples developed to demonstrate that the structural modification that preserves semantics in SCADE, is verifiable through the use of tests generated by the TTM-TVEC tool.

Keywords: Formal Methods. Formal Requirements. Certification. Translation Rules. Tests.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

SCR	Software Cost Reduction

Sumário

1 Introdução

A fase de definição de requisitos é essencial no processo de desenvolvimento de software, pois especifica o comportamento funcional do sistema de software. Pode ser uma descrição abstrata de alto nível de um serviço, uma restrição de sistema ou até uma especificação matemática. Erros e inconsistências na definição de requisitos são responsáveis por uma parte significativa das falhas detectadas ao longo do processo de desenvolvimento de sistemas, principalmente no caso de sistemas dedicados, de tempo real e críticos (NASCIMENTO, 2015). Como exemplo podemos citar um dos mais conhecidos e desastrosos incidente que ocorreu na década de 80, mais especificamente entre junho de 85 a janeiro de 87, o Therac-25. O dispositivo computadorizado para tratamento por radiação para câncer teve diversos equipamentos ministrando doses elevadas a pacientes. Pelo menos 6 pacientes tiveram doses elevadas, alguns foram mortos e outros incapacitados (ROCHA, 2014). O FDA (Food and Drug Administration) investigou o caso e descobriu um programa mal documentado, sem especificação e nem plano de testes. Frank Houston da FDA escreveu em 85: "Uma quantidade significativa de software para sistemas críticos para a vida vêm de pequenas empresas, especialmente na indústria de equipamentos médicos; empresas que se enquadram no perfil daquelas que resistem aos princípios tanto da segurança de sistemas quanto da engenharia de software ou os desconhecem"(ROCHA, 2014). Os sistemas críticos são divididos, basicamente, em três tipos: sistemas críticos de segurança, de missão e de negócios. A principal diferença entre eles é o tipo de prejuízo que pode ser causado por falhas. No sistema de segurança uma falha pode causar riscos a vida humana ou danos ao meio ambiente. No sistema de missão, os riscos são quanto a problemas na atividade de uma meta e, no sistema de negócios, uma falha pode causar grandes prejuízos financeiros (SOMMERVILLE, 2006). Para esse tipo de sistema estar pronto para uso da população e poder ser comercializado, é necessário que ele siga padrões de segurança. Um exemplo de padrão é o certificado Software Considerations in Airborne Systems and Equipment Certification (DO-178B), que é o padrão adotado pela Federal Aviation Administration (FAA) como guia para determinar se o software de um sistema aeronáutico se comportará de forma confiável (RTCA, 1992). Garantir que um determinado sistema está de acordo com um padrão de desenvolvimento e que o mesmo está apto a receber um certificado que permite o seu uso comercial é um dos principais desafios no desenvolvimento de um sistema crítico (ANDRADE, 2013). O desenvolvimento de sistemas críticos necessita de linguagens que atendam às suas necessidades, como em questão de segurança, não ambiguidade, entendimento claro e conciso. Uma linguagem que tem essas características é a linguagem SCR (Software Cost Reduction) (HEITMEYER; BHARADWAJ, 2000), que permite a descrição formal de requisitos do sistema.

1.1 Problema

No decorrer da construção de um software mudanças trazem altos riscos para o projeto. Um deles é ter uma não conformidade entre os requisitos e os artefatos do sistema ou mesmo o código, tornando-se um desafio encontrado pelas empresas atualmente, pois o prazo para desenvolvimento de projetos geralmente é curto e a demanda para mudanças no escopo do projeto é constante. Os requisitos que são escritos na primeira fase do projeto acabam sendo ignorados e ocorrem alterações apenas no código do sistema, deixando assim a especificação desatualizada.

Outro problema é a constante mudança de requisitos que faz parte do dia a dia dos projetos de TI. Muitas vezes é difícil para o cliente repassar todas as suas exigências logo no início do projeto, o que é uma situação natural e, na medida em que o sistema evolui, os requisitos também sofrem alterações. Mudanças devem estar previstas e a equipe do projeto deve estar preparada para atendê-las. Desta forma, requisitos podem ser adicionados, alterados ou retirados do projeto, mas sempre através de um processo controlado (NASCIMENTO, 2015).

Existem muitas causas possíveis para os problemas que cercam um projeto. Entretanto, o erro mais comum que está na raiz da maioria destes problemas é a inadequada mudança de escopo, que leva muitas vezes ao insucesso do projeto por quebra de contrato no atraso do projeto para implantação de mudanças impactando também o custo (JUNIOR, 2010).

1.2 Objetivo Geral

O objetivo geral deste trabalho é realizar a tradução de artefatos da modelagem descritos na ferramenta SCADE (BERRY, 2008) tendo como alvo SCR na sua representação textual, passando assim pelo processo de engenharia reversa, em relação ao trabalho de Andrade (ANDRADE, 2013). É tido como ponto de partida modelos do SCADE, descritos por meio de xscade, que é a representação dos modelos SCADE em forma de esquema XML, e obtido requisitos em SCR.

1.2.1 Objetivos Específicos

- Representação da estrutura do xscade (XML) em uma notação para descrição da tradução com alvo em SCR;
- Descrição de regras para tradução de xscade para SCR;
- Avaliação do resultado da tradução;
- Execução de testes sobre o resultado da tradução;

- Desenvolver provas de conceitos.

1.3 Proposta de solução

Este trabalho tem como objetivo geral realizar a tradução de modelos SCADE representados no esquema XML específico da ferramenta SCADE (xscade) para SCR. Com essa primeira proposta é possível garantir que o problema de obter a não conformidade entre os requisitos e o código será facilmente resolvida, pois os requisitos serão gerados automaticamente a partir do código. Obtendo essa volta é possível atingir outros objetivos menores. A transformação de xscade para SCR se dá através da construção de um tradutor de linguagens, que será melhor detalhado no Capítulo 3. A proposta é obter conformidade com o trabalho criado por Marcelo Andrade (ANDRADE, 2013), que teve como ponto de partida SCR para obter xscade.

1.4 Organização da dissertação

O Capítulo 2 apresenta os fundamentos utilizados neste trabalho como Métodos Formais, SCR, SCADE e transformação de software. SCR é apresentado explicando de forma sucinta os elementos principais que formam uma especificação, além das ferramentas usadas para auxiliar a modelagem dos estudos de caso. Também apresentamos a ferramenta SCADE que é mostrada trazendo informações sobre o paradigma de desenvolvimento usado pela ferramenta e como a mesma armazena os modelos. Detalhes das ferramentas e linguagens de programação usadas na construção do tradutor. São introduzidos O framework Stratego/XT (H. et al., 2014) e o plugin Spoofax (H. et al., 2014) são apresentados. O capítulo também inclui informações sobre como as regras são criadas e como a gramática de SCADE (BERRY, 2008) foi modelada usando a Syntax Definition Language (SDF) (HEITMEYER; BHARADWAJ, 2000).

O Capítulo 3 resume o passo-a-passo da tradução e descreve um subconjunto das regras de tradução elaboradas no presente trabalho. As principais regras criadas para tornar a tradução entre os métodos possível são descritas neste capítulo.

O Capítulo 4 mostra a estratégia de verificação usada para testar se o tradutor mantém o comportamento definido pela especificação após a geração de requisitos SCR.

O Capítulo 5 descreve os estudos de caso usados no presente trabalho além de analisar o resultado da aplicação da estratégia de verificação.

Por fim, o Capítulo 6 finaliza o trabalho citando trabalhos relacionados e apresentando sugestões de trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo serão descritos os assuntos de embasamento da pesquisa pertinente à revisão da literatura. Serão apresentados os temas da literatura que serviram de base para este trabalho, como Métodos Formais, que é a nossa área de pesquisa, Especificação Formal, que abordará o padrão dos requisitos usados no trabalho, a linguagem de requisitos SCR, a ferramenta SCADE, que gera xscade e a ferramenta T-VEC que serve de apoio aos testes que serão executados no SCR.

2.1 Métodos Formais

Métodos formais são baseados em formalismos matemáticos para a especificação, desenvolvimento e verificação dos sistemas. Seu uso para o desenvolvimento de software e hardware é motivado pela expectativa de que possam contribuir para a confiabilidade e robustez de um projeto executando análises matemáticas apropriadas. O uso de análises matemáticas trazem algumas vantagens como a possibilidade de que na descrição de uma funcionalidade não haja inconsistências, ambiguidades e incompletudes que possam passar despercebidas. E a maior vantagem de todas, que é a facilidade de encontrar defeitos no início o projeto, no momento da especificação, torna a correção mais barata, pois o defeito pode ser corrigido na fase inicial do projeto não havendo um impacto maior posteriormente. (BUTLER, 2012)

Métodos formais são vistos com um certo grau de desconfiança. Há várias razões para isso, mas a maioria dos problemas parece ser um resultado da sua má utilização. A maioria dos sistemas formais são extremamente descritivos e abrangentes, linguagens de modelagem têm sido geralmente julgados pela sua capacidade de modelar qualquer coisa. Infelizmente, essas mesmas qualidades fazem métodos formais difícil de usar, pois essas linguagens não são triviais (RUSHBY, 1993).

2.2 Especificação Formal

Uma especificação de software é um artefato que pode ter várias formas, seja um documento, um fluxo, um diagrama ou um pseudo-código. Seu objetivo é descrever um produto de software existente ou a ser construído. Assim como uma especificação descreve um produto, é preciso que o produto satisfaça a especificação. O nível de detalhes definidos em uma especificação dependerá de qual fase do ciclo de vida de desenvolvimento de software a especificação será usada. Como consequência da omissão natural de informações de implementação, uma especificação pode ter várias implementações possíveis.

Para sanar o problema de entendimento dúbio de uma especificação, é indicado o uso de especificação formal que se baseia em métodos matemáticos para construir uma especificação. A especificação formal descreve o que sistema deve fazer, e não (necessariamente) como o deve fazer. Dada uma especificação, é possível utilizar técnicas de verificação formal para demonstrar que o modelo de um sistema candidato está de acordo com a sua especificação. Isto tem a enorme vantagem de que sistemas incorretos são detectados e podem ser revistos antes de se investir na sua implementação. Uma aproximação alternativa é utilizar passos de refinamento para transformar uma especificação, gradativamente numa implementação concreta.

Para saber se uma especificação descreve o sistema corretamente, pode-se fazer uma validação. Devida a sua sintaxe precisa, é possível gerar casos de teste automaticamente a partir de uma especificação formal. A ferramenta Vector Generation System (VGS) da suíte T-VEC permite gerar casos de teste a partir de uma especificação formal descrita em uma linguagem com sintaxe baseada em SCR, que serão vistos mais a frente. A execução de uma campanha de testes não garante que um software esteja livre de problemas, porém o conjunto de casos de teste gerados a partir de uma especificação formal de acordo com critérios de cobertura de teste usado pelo guia DO-178B representa uma campanha de testes robusta.

2.3 SCR

SCR é usado para escrita de especificações de requisitos baseados em tabelas, ideal para especificar sistemas críticos. SCR pode ser aplicado a uma vasta gama de sistemas que estão no mercado, como sistemas aviônicos, redes de telefone e sistemas críticos para a segurança de centrais nucleares. Existem também quatro grandes projetos que conseguiram melhorias usando a linguagem. Um deles é o projeto da NASA que conseguiu eliminar muitos erros em seu desenvolvimento. Foi originalmente desenvolvido por pesquisadores da NRL (Naval Research Lab) para documentar os requisitos do programa operacional de voo da aeronave US Navy‘s A-7. Tem sido aplicado por um grande número de organizações na indústria (ex: Grumman, Bell Laboratories, Ontario Hydro, and Lockheed) para uma ampla variedade de sistemas de controle reais, incluindo sistemas aviônicos, redes de telefonia e segurança (HEITMEYER; BHARADWAJ, 2000).

Para escrever especificações em SCR, é necessário que haja conhecimento mais extenso possível dos requisitos do sistema. A especificação deve conter o que será construído, omitindo detalhes de como será feito. Segundo Heitmeyer (HEITMEYER; BHARADWAJ, 2000), para facilitar a escrita de requisitos em SCR o processo pode ser divido em quatro etapas para construir uma especificação. O primeiro passo é criar um sistema de especificação de requisitos (SRS - System Requirements Specification), que descreve o

comportamento externo do sistema em termos de valores das variáveis monitoradas e controladas do ambiente do sistema. O segundo passo cria um sistema de design de especificação (SDS - System Design Specification), que identifica a entrada e a saída dos dispositivos (ex: sensores e atuadores) do sistema. O terceiro passo cria uma especificação de requisitos de software (SoRS - Software Requirements Specification), que refina o SRS adicionando módulos que usam valores lidos de dispositivos de entrada para calcular variáveis de quantidades monitoradas, e usam variáveis calculadas de quantidades controladas de dispositivos de saída. O quarto passo estende de SRS adicionando comportamento para lidar com falhas de hardware. O maior objetivo é organizar as especificações de requisitos de tal modo que cada mudança altere apenas um único módulo.

Para especificar um sistema de uma maneira eficiente, o método SCR usa variáveis monitoradas que são valores do ambiente que influenciam o comportamento do sistema, variáveis controladas que são valores do ambiente que o sistema controla, termos são variáveis auxiliares que ajuda na concisão da informação e mode class, caso especial de um termo cujo valor são modos como em uma máquina de estados. Para descrever uma variável é usado um prefixo que indica o seu tipo como exibido no Código 2.1: m são variáveis monitoradas, t são termos, mc são mode classes, c são variáveis controladas. Condições e eventos são importantes construções em SCR. Uma condição é um predicado definido de um ou mais estados de uma variável (um estado de uma variável é uma variável controlada, monitorada, um termo ou uma mode class).

Nos códigos 2.1, 2.2, 2.3 e 2.4 é possível ver a declaração de variáveis monitoradas no SCR textual. A declaração segue uma ordem, primeiro vem as variáveis monitoradas, depois controladas, termos e classes de modo. Abaixo serão mostrados exemplos de todos os tipos de declaração de variáveis:

Listing 2.1 – Descrição de varáveis monitoradas em SCR.

```
monitored variables
mWaterPres: yWPres, initially 0 ;
mBlock , mReset : ySwitch , initially Off ;
```

No Código 2.2, é possível ver a declaração de variáveis controladas no SCR textual.

Listing 2.2 – Descrição de varáveis controladas em SCR.

```
controlled variable;
cSafety_Injection: ySwitch, initially On ;
```

No Código 2.3, é possível ver a declaração de termos no SCR textual.

No Código 2.4, é possível ver a declaração de classes de modo no SCR textual.

Listing 2.3 – Descrição de termos em SCR.

```
term variables
tOverridden: boolean , initially false ;
```

Listing 2.4 – Descrição de classes de modo em SCR.

```
mode classes
mcPressure: type_mcPressure, initially TooLow ;
```

Em SCR um evento ocorre quando uma variável muda de valor. Uma notação é escrita como *@T(c) WHEN d* que denota um evento condicionado. Informalmente, essa expressão denota o evento "predicado c transforma-se em true no novo status quando o predicado d se mantém no velho status". A notação *@F(c)* denota o evento *@T(NOT c)* e *@C(x)* denota o evento "variável c tem o valor alterado". A notação *DUR(c)* indica a duração do tempo que a condição c tem sido verdadeira. A Tabela 1 apresenta os eventos do trabalho de Heitmeyer (HEITMEYER; BHARADWAJ, 2000) e o SCR textual com os mesmos eventos.

Tabela 1 – Eventos em SCR

Old mode	Event	New mode
TooLow	@T(mWaterPres $\geq$ Low)	Permitted
Permitted	@T(mWaterPres $<$ Low)	TooLow
High	@T(mWaterPres $<$ Permit)	Permitted

Na Tabela 1 contém os eventos do *mode classes* mcPressure que representa um sistema de injeção de segurança que possui três sensores que monitoram valores do ambiente externo e um dispositivo de saída. Quando a pressão da água é maior ou igual a Low, o estado passa de TooLow para Permitted, quando o estado da pressão da água se encontra menor que Low, o estado passa de Permitted para TooLow, quando a pressão da água é menor que Permit o estado passa de High para Permitted.

SCR pode ser escrito em forma de tabela e manual, esse trabalho usou a forma manual de SCR. O Código 2.5 exibe um exemplo da Tabela 1, exibida no começo dessa seção, em forma textual.

2.4 SCADE

A linguagem síncrona Lustre (HALBWACHS et al., 1991) foi projetada nos anos 80, e resultou na ferramenta industrial de desenvolvimento SCADE, a qual é usada em muitas grandes empresas de desenvolvimento de software. É apoiado por quatro fortes núcleos técnicos: linguagens gráficas e textuais de alto nível e semântica formal. O comportamento

Listing 2.5 – SCR textual, Descrição da Tabela 1 em forma textual

```
var mcPressure :=
        case mcPressure
        [ ] TooLow
        ev
                @T( mWaterPres >= Low ) -> Permitted
        ve
        [ ] Permitted
        ev
                @T( mWaterPres >= Permit ) -> High
                @T( mWaterP res < Low ) -> TooLow
        ve
        [ ] High
        ev
                @T( mWaterPres < Permit ) -> Permitted
        ve
```

de um programa síncrono é uma sequência de reações, onde cada reação consiste de leitura de entradas e mudança do estado interno. Um programa modelado em SCADE é constituído por um conjunto de nós operadores, que definem operações. Os nós podem ser conectados uns com os outros através de canais de entrada e saída. Um nó pode também possuir variáveis locais para facilitar a modelagem. (BERRY, 2008)

SCADE evoluiu a partir da ferramenta SAGA que foi originalmente desenvolvido em 1986 pela Schneider Electric para os sistemas de segurança das centrais nucleares, como uma versão gráfica da língua síncrona Lustre. Em seguida, foi desenvolvida para substituir gradualmente a ferramenta interna Airbus SAO para software no ar. Agora, é usado por um grande número de sistemas aviônicos, ferroviário, indústria e empresas automotivas, controle motor, controle de travagem, controle de segurança, controle de energia, tratamento de alarmes, etc. SCADE incorpora o compilador KCG a partir de projetos de alto nível para C e adere a certificação DO-178B.

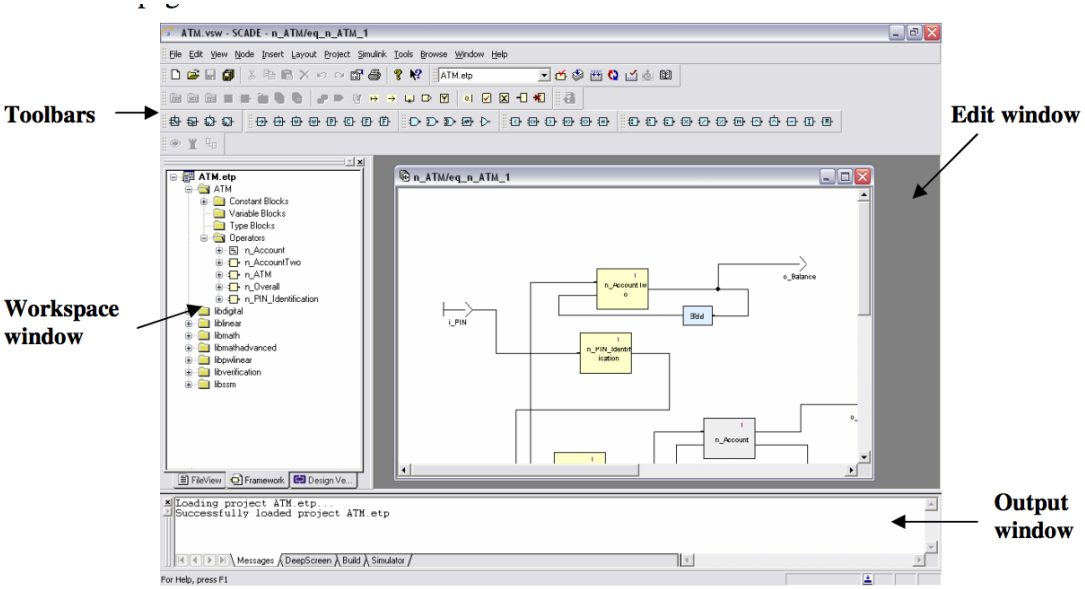
SCADE usa uma IDE que desenvolve modelagens baseadas em blocos e diagramas. A ideia geral é de gerar a construção correta da implementação embarcada a partir do alto nível de especificações formais executáveis, aumentando a qualidade do software, enquanto diminui os custos de concepção e de validação. Uma vez que a especificação é executável, ela pode ser completamente simulada e verificada antes da homologação. como a implementação é gerada automaticamente, não há erros introduzidos na fase de implementação.

SCADE suporta o desenvolvimento de controladores com um comportamento determinístico (isto é, para qualquer entrada e estado há uma saída única determinada pelo design) e pode, então, ser usado para executar uma simulação do modelo. Um

modelo SCADE é construído a partir de diagramas de blocos hierárquicos. Os elementos hierárquicos são chamados nós. Um nó tem entradas e saídas, e sua funcionalidade pode ser definida por um diagrama de blocos que contém outros submódulos. A interface real do sistema é definida em um nó superior. Os próprios nós definem a funcionalidade (ou comportamento) do (sub)sistema correspondente (ELMQVIST JERKER HAMMARBERG, 2005).

Os nós podem ser gráficos ou textuais que dão ao designer duas formas diferentes de modelagem. Ao modelar o comportamento com um nó gráfico, o usuário pode usar os operadores predefinidos (matemáticos, lógicos) encontrados na *Toolbars* (ELMQVIST JERKER HAMMARBERG, 2005).

Figura 1 – Interface gráfica de SCADE (ELMQVIST JERKER HAMMARBERG, 2005)



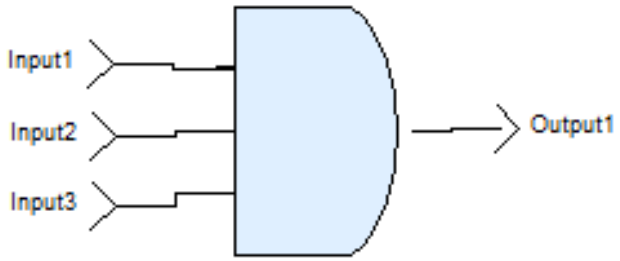
O SCADE apresenta o modelo em uma interface gráfica de usuário na Figura 1. A estrutura hierárquica do sistema pode ser visualizada na janela *Workspace* sob a guia *Framework* e cada modelo pode ser aberto na janela *Edit*, tanto para modelagem gráfica quanto para modelagem textual. Todos os operadores incorporados podem ser visualizados nas Barras de Ferramentas e informações para o designer (como erros, progresso de compilação, etc.) são apresentadas na *Output Window*.

A geração de código fonte no SCADE tem como ponto de partida a notação gráfica da ferramenta. Nesta notação estão presentes os diagramas de bloco que representam, por exemplo, operadores ou componentes de entrada e saída. Verificação funcional é realizada de duas maneiras: técnicas de simulação convencionais melhoradas por uma animação gráfica da análise de cobertura de projeto e modelo e verificação formal das propriedades de segurança por verificação de modelo ou interpretação abstrata. Verificação funcional é necessária somente em nível de diagrama de bloco, uma vez que o código C embarcado é

gerado pelo compilador certificado KCG sendo considerado correto e qualificável (COSTA, 2009).

Em SCADE primeiro é feita a modelagem gráfica do sistema usando a própria IDE da ferramenta SCADE. Há operadores para representar soma, subtração, divisão, multiplicação, há if, else e laços. A Figura 2 um Operador And com 3 entradas e 1 saída escrito em forma de modelagem gráfica no SCADE 2.

Figura 2 – Modelagem gráfica em SCADE



O Código 2.6 representa o xscade de uma modelagem em SCADE. O xscade é o esquema XML gerado para representar a modelagem gráfica no SCADE. O exemplo abaixo é uma pequena representação de tipo, no caso o *<TypeRef* name=/> representa o nome do tipo.

Listing 2.6 – xscade de um tipo

```
<type>
 <NamedType>
  <type>
   <TypeRef name="bool"/>
  </type>
  </NamedType>
</type>
```

2.5 T-VEC

T-VEC (TEAM, 2012) é uma suíte de ferramentas que foca no desenvolvimento de especificações e na geração automática de casos de teste. T-VEC (TEAM, 2012) inclui o Tabular Modeller (TTM), onde são modeladas as especificações baseadas na linguagem SCR. O TTM ajuda no desenvolvimento de modelos de requisitos de software. Estes modelos geram testes a partir do uso integrado da ferramenta VGS também da T-VEC e suportam a análise automática de defeitos. Em resumo, a ferramenta TTM modela requisitos baseados em SCR e traduz estes requisitos para a ferramenta VGS, que por sua vez gera casos de teste.

Um modelo de requisitos em TTM é um refinamento dos requisitos de um sistema e descreve os relacionamentos entre as suas entradas e as saídas. A verificação e validação automática de requisitos com a suíte T-VEC começa com a modelagem dos requisitos na ferramenta TTM. Em seguida é feita a tradução dos modelos para a ferramenta VGS para tornar possível a verificação do modelo e a geração de casos de teste.

Modelar exemplos de SCR em TTM é uma atividade quase automática, com exceção das instruções Assumption, Assertion e DUR. O foco do uso da suíte T-VEC neste trabalho é a geração de vetores de teste para os estudos de caso, portanto não há a necessidade de modelar Assertions no TTM. Uma Assumption pode interferir na geração de casos de teste e por isso deve ser modelada apropriadamente no TTM, que por sua vez não suporta este tipo de instrução diretamente. Já uma Assertion corresponde a uma propriedade do sistema que pode ser analisada, independentemente da geração de casos de teste.

Assim, como solução cada Assumption foi modelada como uma variável auxiliar em TTM. Para cada instrução Assumption, foi criada uma variável booleana em TTM definida por uma tabela de evento. A primeira linha da tabela define que, se uma entrada do sistema mudou de valor e a expressão do Assumption tiver o valor verdadeiro, a variável auxiliar será verdade. Já a segunda linha define que, se uma entrada do sistema muda de valor e a expressão do Assumption tiver o valor falso, então a variável auxiliar definida pela tabela também será falsa, indicando que a Assumption foi violada. A Figura 2.5 ilustra a tabela auxiliar criada para uma Assumption (ANDRADE, 2013) dos estudos de caso do presente trabalho.

#	Assignment	Event
1	TRUE	((@C(mBrake) OR @C(mEngRunning) OR @C(mIgnOn) OR @C(mLever) OR @C(mSpeed) OR @C(time)) AND ((mSpeed' - mSpeed) <= kMaxAccel))
2	FALSE	((@C(mBrake) OR @C(mEngRunning) OR @C(mIgnOn) OR @C(mLever) OR @C(mSpeed) OR @C(time)) AND NOT ((mSpeed' - mSpeed) <= kMaxAccel))

Figura 3 – Assumption modelado in TTM.

O operador DUR foi modelado em TTM como uma combinação de tabelas de eventos. Para cada expressão diferente com o operador DUR em uma especificação, uma variável auxiliar do tipo termo foi criada para armazenar o valor do ciclo (time) referente ao momento em que a expressão DUR torna-se verdadeira. O termo é definido por uma tabela de evento.

Em TTM são usados arquivos de mapeamento para criar drivers de teste com VGS. Os arquivos de mapeamento são específicos para cada tabela de SCR e cada arquivo basicamente contém detalhes de configuração, além de código C esquemático que

automatiza. Para mapear é necessário ter arquivos de mapeamento que descrevem para cada vetor de teste como a ferramenta deve gerar código C.

O Código 2.7 ilustra um laço *(<vector->input bindings...>)* que irá iterar sobre as entradas de um vetor de teste e, para cada entrada, irá gerar o código C referente ao termo *<binding->object->set descriptor>*. Este termo corresponde ao código C concreto de uma instrução de atribuição definida em outra seção do arquivo de mapeamento. O Código 2.8 exemplifica a seção de código de mapeamento referente à instrução de atribuição para a variável *mWaterPres*. O termo *<inC struct instance>* será traduzido para o nome da estrutura *(struct)* de C que armazenará as entradas de um ciclo do sistema. Por sua vez, o termo *<binding->value>* será traduzido para o valor que a variável *mWaterPres* deverá receber definido no vetor de teste. Por fim, o termo *<project->name>* é traduzido no nome do sistema.

Listing 2.7 – Código que descreve como um mapeamento deve ser escrito em código C

```
<vector -> input_bindings...>
{
<binding -> object -> set_descriptor>
}
```

Listing 2.8 – Código descreve como uma instrução de como uma instrução de atribuição da variável mWaterPres será traduzida para código C.

```
mBlock
{
SET_DESCRIPTOR ' < inC_struct_instance >.mWaterPres =
< binding > value > _ < project > name > ; ' ;
}
```

Em todo SCR gerado nesse trabalho foram executados testes com o T-VEC. Ao gerar SCR a partir do xscade, o SCR foi modelado no T-VEC em forma de tabela, os testes foram automaticamente gerados usando o TTM. O próximo passo foi gerar código C na ferramenta SCADE a partir do xscade. Ao obter o código C foi criado um projeto em um compilador C, nesse projeto foi usado a IDE CodeBlocks (TEAM, 2012), mas poderia ter sido usado qualquer compilador C. Os testes gerados pelo TTM foram incluídos nesse projeto, executados e obtido um resultado.

2.6 StrategoXT

Stratego/XT é uma linguagem usada através de Spoofax para transformação de programa. Stratego fornece regras de reescrita para o framework Stratego/XT que é composto por duas partes: Stratego, uma linguagem para implementar transformações de

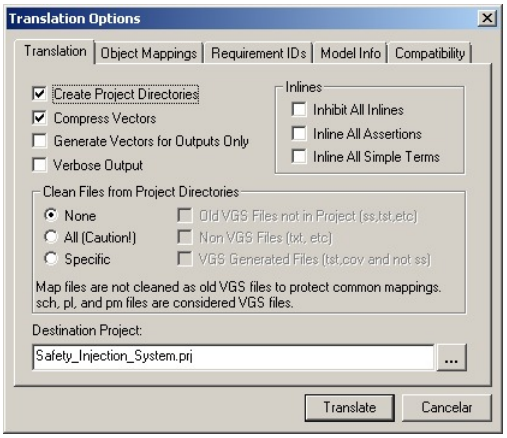


Figura 4 – Tradutor do procedimento de TTM para VGS

software e XT, uma coleção de ferramentas de transformação. A linguagem Stratego é uma linguagem poderosa para a implementação das transformações de um sistema. A ferramenta XT ajuda com a implementação da infra-estrutura necessária para as transformações, tais como um analisador e um formatador de saída.

Spoofax deriva automaticamente implementações eficientes para vários recursos da IDE de projetos de linguagem declarativa. Ele também pode ser usado através um plugin do Eclipse (BRAND P. KLINT, 2008). Spoofax suporta desenvolvimento de linguagem incremental e interativa, fazendo com que seja simples de incluir uma mudança, oferece um ambiente completo integrando para definição de sintaxe, name biding (ligações de nomes a objetos), análise de tipo (verificação de compatibilidade de tipos), transformação de programa e geração de código. Para cada um desses aspectos, são usadas metalinguagens declarativas que abstraem sobre a implementação de processadores de linguagem (BRAVENBOER; VISSER, 2005).

Além disso o Spoofax Language Workbench suporta a definição de todos os aspectos das linguagens textuais usando meta-linguagens declarativas de alto nível, incluindo SDF3 para definição de formalismo de sintaxe, NaBL para linguagem de binding de nomes, Stratego para transformação de linguagem, DynSem dynamic para linguagem de especificação semântica dinâmica e ESV editor para serviços e configuração de linguagem.

Stratego é o núcleo de Stratego/XT. É uma linguagem de transformação de software baseada no paradigma de reescrita, com uso de táticas ou estratégias. Transformações básicas são definidas usando termos condicionais. Sistemas de reescrita formalizam as modificações de termos ou árvores. Uma regra de reescrita descreve como um fragmento de programa correspondente a um certo padrão é transformado em outro fragmento programa (H. et al., 2014).

Uma regra de reescrita tem a forma *L: l -> r*, onde L é o rótulo da regra e o termo. Os padrões l e r são o lado esquerdo e o lado direito, respectivamente. Um padrão de termo

é uma variável, um construtor nulo C ou a aplicação C *(p1, ..., pn)* de um construtor n-ário C para padrões de p_i. Em outras palavras, um padrão de termo é um termo com variáveis. Uma regra de reescrita condicional é uma regra da forma *L: l -> r* onde s, com s uma computação que deve ser bem sucedida para que a regra se aplique. Uma regra de exemplo é a regra 2.9 de dobra constante (VISSER, 2004).

Listing 2.9 – Regra para adição de duas constantes

```
EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add>(i, j) => k
```

O código 2.10 que reduz a adição de duas constantes a uma constante chamando a função de biblioteca adicionar para add inteiros. Outro exemplo é a regra 2.10 que divide uma lista de bindings em ligações separadas (VISSER, 2004).

Listing 2.10 – Regra para adição de duas constantes

```
LetSplit : Let([d1, d2 | d*], e*) -> Let([d1], Let([d2 | d*], e*))
```

Segue como exemplo o Código 2.11 que traduz um tipo em xscade para SCR. Stratego lê as regras de forma recursiva, portanto primeiro criamos uma regra para obter a lista com os tipos, depois lemos a lista de tipos pegando cada elemento da lista como nome, namedType, que é o tipo usado, como booleano, inteiro, etc. e o pragmaId, que é o elemento que o xscade usa para guardar as coordenadas dos blocos gráficos mapeados na tela. O pragmaId não impacta no atual trabalho, pois para traduzir xscade para SCR o pragma não tem impacto.

Listing 2.11 – Regra de tradução de Tipos escrita em Stratego.

```
to-type:
[head | tail] ->
$[[resolvedHead]
[resolvedTail]
]
with resolvedHead := <resolve-type> head
with resolvedTail := <to-type> tail

resolve-type:
Type(name, namedType, pragmaId) -> $[[name]: [typeString]]
with typeString := <write-type-named-type> namedType
```

2.6.1 SDF

O *Syntax Definition Formalism (SDF)* é uma meta-sintaxe usada para definir gramáticas livres de contexto. SDF descreve gramáticas para linguagens de domínio

específico, linguagens de aplicação, linguagem específica de domínio formato de dados e outros tipos de linguagens formais. Seu objetivo principal é a descrição da sintaxe de uma linguagem, mas também fornece um conjunto de ferramentas para a geração de analisador sintático para a linguagem definida (BRAND P. KLINT, 2008). Uma outra forma de definir uma gramática livre de contexto é o EBNF.

Houve uma atualização de SDF, hoje estamos em SDF3. as maiores mudanças na atualização está na definição de sintaxe declarativa altamente modular que combinam sintaxe lexical livre e contexto em um formalismo e pode definir sintaxe concreta e abstrata juntos em modelos. Uma especificação SDF3 consiste de uma série de declarações em módulo. Cada módulo pode definir seções que contenham importações, sintaxe, sintaxe lexical livre de contexto, desambiguações e opções de modelo.

No trabalho de (ANDRADE, 2013), foi usado SDF. Nestre trabalho, porém, usamos SDF3, pois SDF já se encontra fora de uso pelo Spoofax, que só permite construções a partir de SDF3. O tradutor SCR -> xscade não foi atualizado para SDF3. Até houve a tentativa, mas não obtivemos sucesso, pois seria necessário a mudança das regras de tradução, obtendo um esforço maior do que o esperado. A migração de SDF para SDF3 ficará para um trabalho futuro. Além de que, mesmo usando SDF o tradutor SCR -> xscade funciona perfeitamente.

A sintaxe lexical geralmente descreve a estrutura de baixo nível de programas (muitas vezes referida como símbolos lexicais). No entanto, em SDF3 o conceito token não é realmente relevante, uma vez que apenas as classes de caracteres são terminais. As seções de sintaxe lexicais em SDF3 são simplesmente uma notação conveniente para o baixo nível de sintaxe de uma linguagem.

As especificações escritas em SDF3 são organizadas em módulos. Estes módulos são utilizados para dividir especificações mais complexas em partes mais simples e com foco em um contexto específico do problema. Assim no Código 2.12 é exibida a estrutura básica de SDF3.

Listing 2.12 – Estrutura básica de uma especificação SDF.

```
module <ModuleName>
<ImportSection>*
<ExportOrHiddenSection>*
```

O módulo é definido no *<ModuleName>* é o ponto onde deve ser definido de um módulo. Os outros atributos da definição tratam da importação ou uso de outros módulos e da visibilidade de definições que estão contidas neste módulo. Um *<ExportOrHidden-Section>* é uma seção de exportação ou uma seção oculta. O módulo começa com a palavra-chave *Export* e faz com que todas as entidades na seção sejam visíveis para outros

Listing 2.13 – Exemplo de definição de lexical syntax.

```
module Common

lexical syntax

ID              = [a-zA-Z] [a-zA-Z0-9]*
MVAR                = "m" [A-Z] [a-zA-Z0-9]*
CVAR                = "c" [A-Z] [a-zA-Z0-9\_]*
MCVAR               = "mc" [A-Z] [a-zA-Z0-9]*
TVAR                = "t" [A-Z] [a-zA-Z0-9]*
OPNAME              = MVAR | CVAR | MCVAR | TVAR
OPKIND              = "node" | "function" | "assume"
INT             = "-"? [0-9]+
STRING          = "\"" StringChar* "\""
Booleano            = "true" | "false"
TYPE            = INT | Booleano
Value               = INT | STRING
```

módulos. *Hidden* significa que quando outro módulo importa algum módulo específico, nenhuma das seções hidden estarão presentes na composição.

O módulo Lexical Syntax 2.13 é o local onde serão descritos os elementos básicos do programa. Nesta seção geralmente são definidas as associações entre alguns símbolos variáveis da gramática, como por exemplo os identificadores de funções e variáveis da linguagem. Cada módulo deve conter no mínimo um nome identificador que pode ser usado posteriormente.

O módulo *Context-free Syntax* é o local onde deve ser descrita a estrutura de alto nível das sentenças da linguagem. Este módulo é composta basicamente de um conjunto de regras de produção (BRAND P. KLINT, 2008). No Código 2.14 é mostrado um exemplo de uma definição do módulo *context-free syntax* de um tipo e uma constante em xscade. Toda a estrutura do esquema foi retirada do resultado tradutor de (ANDRADE, 2013) e os valores como name é usado um *ID*, como mostrado no Código 2.14, um *ID* é uma combinação de letras maiúsculas, minúsculas e números, o *ConstValue* é o valor da constante representado por *Value*, onde pode ser usado números inteiros, números decimais ou um nome. Também há a chamada de um módulo *[namedType]*, onde serão pegos os tipos, como booleano, inteiro ou um novo tipo definido.

Listing 2.14 – Tipo e constante em SDF.

```
Type.Type = [
<Type name="[ID]">
<definition>
[NamedType]
</definition>
<pragmas>
<ed:Type oid="!ed/type/[ID]" />
</pragmas>
</Type>
]

Constant.Constant =
<Constant name="[ID]">
<type>
[NamedType]
</type>
<value>
<ConstValue value="[Value]"/>
```

3 Traduzindo xscade para SCR

Um software é um objeto estruturado com sintaxe e semântica, e é essa estrutura que permite transformar um programa. A semântica é o comportamento de um programador nos dando os meios para comparar programas e raciocinar sobre a validade de transformações. Uma linguagem de programação é uma coleção de programas. Esta é uma definição bastante ampla que é projetada para cobrir linguagens adequadas de programação (com uma interpretação operacional), mas também formatos de dados e linguagens específicas de domínio.

Uma transformação de linguagem é uma operação que gera uma linguagem de programação a partir de outra linguagem. A transformação é feita com a ajuda de uma ferramenta que seja capaz de criar um tradutor. No presente trabalho foi usado o Plugin Spoofax no Eclipse. Essa ferramenta inclui um analisador que transforma sintaxe concreta em uma representação abstrata e detecta erros de sintaxe, um formatador que transforma uma representação abstrata em sintaxe concreta, realce de sintaxe e autocomplitação de código (VOELTER; VISSER, 2011).

É muito mais prático utilizar um sistema de transformação que se aplica as especificações da transformação requeridas do que fazer transformação manualmente. Transformações de programas podem ser especificadas como procedimentos automatizados que modificam as estruturas de dados do compilador quando acontece o processo de descrever uma linguagem em outra linguagem.

A generalização da equivalência de semântica é a noção de refinamento de programa: um programa é o refinamento de outro se ele inicia com todos os estados iniciais do qual o programa original inicia e, para cada estado é garantido terminar em um estado final possível para o programa original. Em outras palavras, um refinamento de um programa é mais definido e mais determinístico que o programa original. Se dois programas são refinamentos de outro, então eles são equivalentes.

Neste trabalho [1] a tradução entre linguagens será de SCADE para SCR, a estrutura de SCADE é dada no Código 3.1:

A seguir no código 3.2 é exibida a estrutura do SCR, onde será traduzida a estrutura de SCADE até chegar na estrutura de SCR:

Na Tabela 2 será mostrada a equivalência entre a estrutura de XSCADE e SCR.

[1] O tradutor está disponível em https://github.com/kamcarvalho/tradutorXscadeSCR

Listing 3.1 – Estrutura SCADE.

```
<file>
   <declarations>
           <Package name=>
           <declarations>
                   <Type name=>
               </type>
               <Constant name=>
               </Constant>
               <Operator kind="" name="">
               <inputs>
               </inputs>
               <outputs>
               </outputs>
               <locals>
               </locals>
               <Equation>
                   <left>
                   </left>
                   <right>
                   </right>
                   <Assertion  kind="guarantee>
                   </Assertion>
                   <Assertion  kind="assume>
                   </Assertion>
               </Equation>
               </Operator>
         </package>
    </declarations>
   </file>
```

3.1 Regras de Tradução

Neste capítulo apresentamos as regras de transformação de xscade para SCR. Abaixo é exibida uma regra de transformação genérica, onde o lado esquerdo da regra p1 representa um termo na linguagem xscade e o lado direito p2 da regra representa um termo em SCR.

R: p1 -> p2

Escrever regras é um trabalho mais complexo do que definir a gramática de um sistema, pois a descrição de notações SDF é mais uma reprodução de padrões do xscade em uma gramática. Para escrever regras é necessário ler listas seja de tipos, constantes, variáveis, transformar e imprimir valores que correspondem à linguagem de saída. Para um

Listing 3.2 – Estrutura SCR.

```
spec
type definitions
constant definitions
monitored variable
controlled variable
term
mode classes
assumptions
assertions
function definitions
```

completo entendimento das regras descritas nesta seção, é importante entender algumas notações complementares. A notação representada pela expressão abaixo significa que head é a cabeça de uma lista e o tail é o resto da lista composta por elementos. O caractere + representa o operador de concatenação de listas.

[head + tail]

Para cada operador em xscade é criada uma tabela em SCR. Essas tabelas possuem o mesmo comportamento e as mesmas entradas e saídas que os operadores que a representam. Em xscade foi criado um operador principal que contém toda a especificação do sistema, que coordena o processamento entre os diversos operadores criados. Esse operador principal foi criado alterando o xscade original, pois em uma especificação SCR, só é possível obter o resultado final de um ciclo após o processamento de todas as tabelas de uma especificação em SCR em conjunto.

3.2 Regras Principais

Serão ilustradas a seguir algumas regras de tradução com o intuito de mostrar a estratégia de tradução criada neste trabalho. Como já mostrado um pouco acima neste capítulo na estrutura de xscade no Código **??**, são recebidos como entrada alguns fragmentos de código xscade e produz-se uma especificação de SCR completa, além de fazer chamadas a outras regras de tradução. O código SCR criado declara um pacote de nome Spec mostrado na Regra 1, que em xscade são delimitadas pela tag *<declarations>. . .</ declarations>*. Esse pacote conterá todas as declarações do xscade e o pragma que é um termo criado pelo xscade para identificação de cada elemento usado na estrutura.

Na Regra 2 trata de declarações de tipos *([Types] to-types)*, declarações de constantes *([Constants] to - Constants)* e, definido dentro dos operadores, variáveis *(([(monitored, controlled, termo, modeClasses) (Assumptions) (Assertions) (Functions)] to-function)*. O

Tabela 2 – Equivalência entre XSCADE e SCR

XSCADE	**SCR**	**Resumo**
declarations	spec	é o pacote onde estará todo o código do programa, tanto em XSCADE como em SCR
type	type definitions	os tipos encontrados no programa, tanto em XSCADE como em SCR
constant	constant definitions	as constantes encontradas no programa, tanto em XSCADE como em SCR
inputs e outputs	monitored variable, controlled variables term e mode classes	em XSCADE todas as variáveis do programa encontram-se concentradas no input e output. Em SCR as variáveis são dividas por suas características.
Assumptions	Assertion kind="assume"	em xscade assumptions e assertions são representados por Asserions, mas o que diferencia é o tipo da assertion, se ela é do tipo assume é uma assumption em SCR, se é do tipo guarantee é um assertion em SCR.
Assertion	Assertion kind="guarantee"	Assertion é uma assertion em xscade com o tipo guarantee
Funcation definitions	left e right	as funções em xscade são definidas nas equações, cada equação tem o lado direito e o lado esquerdo, a tradução de cada equação transforma-se nas funções de SCR

operador principal em SCADE é criado para satisfazer a necessidade de SCR, para que seja possível obter o resultado final do processamento de um ciclo após o processamento de todas as tabelas de uma especificação SCR em conjunto. É um componente principal que conectará todos os outros operadores. Traduzido para SCR, esse operador contém todas as variáveis declaradas divididas em variáveis monitoradas, variáveis controladas, termos e classes de modo.

Rule 1. *Package(spec, declarations, pragmas) to-package -> [spec]*

Rule 2. *Declarations(types*, constants*, operators*) to-declarations -> [type definitions] to-type [constant definitions] to-constant [Operator] to-operator*

3.2.1 Tipos

Para cada definição de tipo em xscade, um tipo é criado em SCR. Assim, na Regra 3 um tipo inteiro em SCADE é transformado em um tipo com limites entre *Sint1* e *Sint2* em SCR desconsiderando os limites. Para corrigir esta limitação, um conjunto de Assumes e Guarantees é criado no decorrer da tradução para limitar as entradas e saídas que são de algum tipo numérico e que possuem restrições em SCR.

Rule 3. *[type + typelist] -> [resolvedType] resolve-enum-value [resolvedTypeList to-type]*

A regra 4 é a leitura da lista de enum. É onde são lidos os valores de cada tipo do xscade.

Rule 4. *Enum(enumValues*) -> [enum in [valores];] with valores := <to-enum-value> enumValues**

Outra limitação de SCADE comparado com SCR, é o fato de que SCADE não suporta enumeradores diferentes com valores com nomes iguais. Por exemplo, a representação abaixo ilustra dois tipos enumeráveis declarados em SCR onde ambos possuem B e C como opções de valores. Em SCADE isso não é possível.

1 type_A : enum i n A, B , C ; 2 type_B : enum i n B , C , D;

A tradução feita por Andrade (ANDRADE, 2013) não suportava especificações SCR que possuam esses elementos. Era necessário ajustar a especificação antes de aplicá-la à tradução. Era manipulada uma lista de enumeradores criando uma tag XML para cada elemento da lista. Na Regra 5 do presente trabalho são colocados todos esses enumeradores em apenas uma variável.

Rule 5. *EnumValue(valor, pragma) resolve-enum-value -> [valor]*

3.2.2 Constants

Semelhante à tradução de tipos, uma constante é criada em SCR para cada constante definida em xscade. A regra se dá de modo generalizado, onde sempre trabalhará com listas. A tradução de uma lista de constantes é resolvida pela Regra 6. A declaração do tipo da constante é traduzida pelo termo *[Type] Type* e a expressão que define o valor da constante pelo termo *[Value] Value*, exibidos na Regra 7. No tradutor SCR > xscade, o parâmetro 0 indicava que a tradução da expressão não levaria em consideração o apóstrofo de uma variável tratando todas as variáveis da expressão a ser traduzida como variáveis do estado corrente.

Rule 6. *[constant + constantList] to-constant -> [resolvedconstant] resolve-constant [resolvedConstantList to-constant]*

Rule 7. *Constant(name, type, value, pragmaId) -> [name]= [value]: [type];*

3.2.3 Eventos

Em SCR há três tipos de eventos. A notação *"@T(c)"* define um evento que será considerado verdadeiro se *c* for verdade no estado corrente do sistema e for falso no estado anterior. Em xscade *"@T(C) WHEN d"* equivale a um *"NAry and"* seguido de um *"UnaryOp not"*. Em SCR, quando uma variável não é seguida de um apóstrofo em uma expressão de evento, isso indica que o valor da variável será o do ciclo anterior do sistema, e se a variável é seguida de um apóstrofo, significa que ela está levando em consideração o estado corrente do sistema. Em xscade quando a variável é um*idExpression* de um input ou output a variável recebe apóstrofo. O *idExpression* são tags usadas para referenciar variáveis. Veja por exemplo o Código 3.3:

Listing 3.3 – Representação IdExpresion

```
<IdExpression>
   <path>
                <ConstVarRef name="c_the_command_in_control_output}"/>
      </path>
    </IdExpression>
```

A Regra 8 mostra um evento *@T* e a Regra 9 uma tradução quando existe apóstrofo. O evento *@F* pode ser transcrito como o evento *"@T(NOT c)"*, equivale a um *"NAry and"* seguindo de um *"UnaryOp not"* e outro *"UnaryOp not"*. O evento *"@C(x)"*, o qual será verdade caso a variável x mude de valor de um estado para o outro, em xscade equivale a um *"NAry and"*, seguindo de um event *@T* e outro operador. Uma observação a ser feita é que nos estudos de casos mostrados mais a frente, o primeiro estudo de caso Safety Injection originalmente está modelado usando um evento *"@T(NOT c)"*, que por ser igual a *"@F(c)"* não é possível encontrar diferenças no xscade e obter a tradução com o *"@T(NOT c)"*. Sendo assim, por ser equivalente será sempre usado o *"@F(c)"*.

Rule 8. *Event(firstOperand, secondOperand) resolve-operand -> @T [resolvedSecondOperandNoApostrophe]*

Rule 9. *IdExpression(expressionName) resolve-operand -> [expressionNameResolved] where if <eq> ("true", <within-inputs-outputs-locals> (expressionName, inputs, outputs, locals)) then expressionNameResolved := [expressionName]' else expressionNameResolved := [expressionName] end*

3.2.4 Variáveis

As variáveis monitoradas, controladas, os termos e as classes de modo são encontradas dentro do operador principal que é o primeiro operador de um sistema definido

em xscade. As variáveis se encontram dentro das tags *<inputs>...</inputs>* e *<outputs>...<outputs>* e concatenadas em uma lista de declarações de variáveis chamada Variable. A tradução é realizada lendo-se o último operador do xscade de um sistema. Depois lê-se as variáveis e através da sua nomenclatura elas são dividas em *"monitored", "controlled", "terms"e "mode classes". Todas as variáveis monitored começam com "m", controlled "c", terms com "t"e mode classes com "mc"*. No estudo de caso Cruise Control houve uma exceção, porque foi necessário criar uma variável auxiliar do tipo term com o nome DUR, que armazenava o valor do ciclo (time) referente ao momento em que a expressão DUR torna-se verdadeira. As regras 10 a 13 exibem como foi realizada a tradução de variáveis monitoradas e os outros tipos de variáveis usa o mesmo padrão de tradução.

A Regra 10 lista todos os termos que há dentro de um operador principal.

Rule 10. *MainOperator(opkind, opname, inputs, outputs, locals, data, pragmas) resolve-monitored -> [resolved-controlled-inputs] [resolved-controlled-outputs] [resolved-controlled-locals]*

A Regra 11 pega a lista com as variáveis monitoradas.

Rule 11. *Inputs(variables) resolve-monitored-inputs -> [resolvedVariables]*

A regra 12 pega todas as variáveis que estão dentro da lista de variáveis monitoradas.

Rule 12. *[monitored + monitoredList] resolve-monitored-variables -> [resolved-Monitored][resolvedMonitoredList]*

A regra 13 lista o conteúdo das variáveis monitoradas.

Rule 13. *MonitoredVariable(variableName, type, lasts, pragma) resolve-monitored-variable -> [variableName]: [typeName], initially [initValue];*

3.2.5 Operador principal

Após cada nó em SCADE se tornar uma tabela em SCR, a tradução cria mais uma tabela para representar toda a especificação. Esta tabela faz chamadas aos outras tabelas que representam os nós de SCADE. Os nós podem ser conectados uns com os outros através de canais de entrada e saída. Um nó pode também possuir variáveis locais para facilitar a modelagem. Um nó faz chamadas aos outros nós que representam as tabelas de SCR. As variáveis de entrada do nó são traduzidas para monitoradas, as variáveis de saída para controladas e os termos e as variáveis locais são traduzidas para classes de modo e termos que são variáveis auxiliares. A Regra 14 constrói um operador principal que contém toda a lógica descrita na especificação. É resolvida uma lista de operadores e cada elemento chama seu conjunto de regras que está em uma classe específica para cada um. São encontrados mais detalhes nas seções 3.2.4 e 3.2.6.

Rule 14. *[Operator + OperatorList] to-operator -> monitored variables [resolved-*

Monitored] controlled variables [resolvedControlled] term variables [resolvedTerm] mode classes [resolvedModeClass] assumptions [resolvedAssumptions] assertions [resolvedAssertions] function definitions [resolvedFunctions]

3.2.6 Assumptions e Assertions

Para adicionar restrições a uma especificação, SCR implementa as seguintes notações: Assumption e Assertion. Uma Assumption descreve as restrições impostas nas variáveis monitoradas e controladas pelo ambiente em que o sistema modelado se encontra (HEITMEYER; BHARADWAJ, 2000). Uma Assertion, por sua vez, é usada para verificar propriedades, como por exemplo propriedades de segurança e confiabilidade, que uma especificação deve satisfazer (HEITMEYER; BHARADWAJ, 2000).

Para cada Assume de xscade as regras de tradução criam um Assumption em SCR e para cada Guarantee de xscade é criado uma Assertion de SCR. É importante notar que um Assume de xscade suporta, em suas equações, saídas apenas com referência a valores do ciclo anterior do sistema. Como os Assumptions de SCR suportam saídas com valores do estado anterior ou do estado corrente. As regras 15 e 16 traduzem um Assumption. O Assumption tem um nome e é formado por Operands que podem ser adição, subtração, evento, valor, etc. O operand será resolvido por meio da regra resolved Operand.

Rule 15. *AssumptionOperator(assumption) resolve-assumption-or-assertion -> [resolvedAssumption]*

Rule 16. *Assumption(assumptionName, mainOperand, pragmas) resolve-assumption -> [assumptionName]: [resolvedOperand]*

4 Avaliação do Tradutor

A estratégia de verificação desse trabalho se dá no trabalho da volta de xscade para SCR. A partir disso, há construções de validações no desenvolvimento de aplicações em xscade e SCR, e também refatorações de circuitos. A estratégia de validação usada por Marcelo Andrade (ANDRADE, 2013) foi construída a partir do uso de T-VEC, VGS e código C gerado pelo SCADE em dois estudos de caso, o Safety Injection e o Cuise Control. Os dois estudos de caso estão presentes na literatura, especificados em SCR textual, havendo a possibilidade de serem utilizados diretamente no tradutor SCR -> SCADE. No presente trabalho ambos estudos serão usados. Foi adicionado mais um estudo de caso, o Priority Command incluímos um exemplo usando redundância modular tripla.

Em todas as estratégias que o tradutor xscade -> SCR se fez presente fizemos uso de T-VEC para construir os testes e do VGS para transformar a modelagem do T-VEC em casos de testes. Para executá-los no código C gerado pelo SCADE foi usada a ferrameta CodeBlocks (TEAM, 2012), mas poderia ter sido usado qualquer compilador C.

4.1 Verificando a conformidade da tradução xscade - SCR

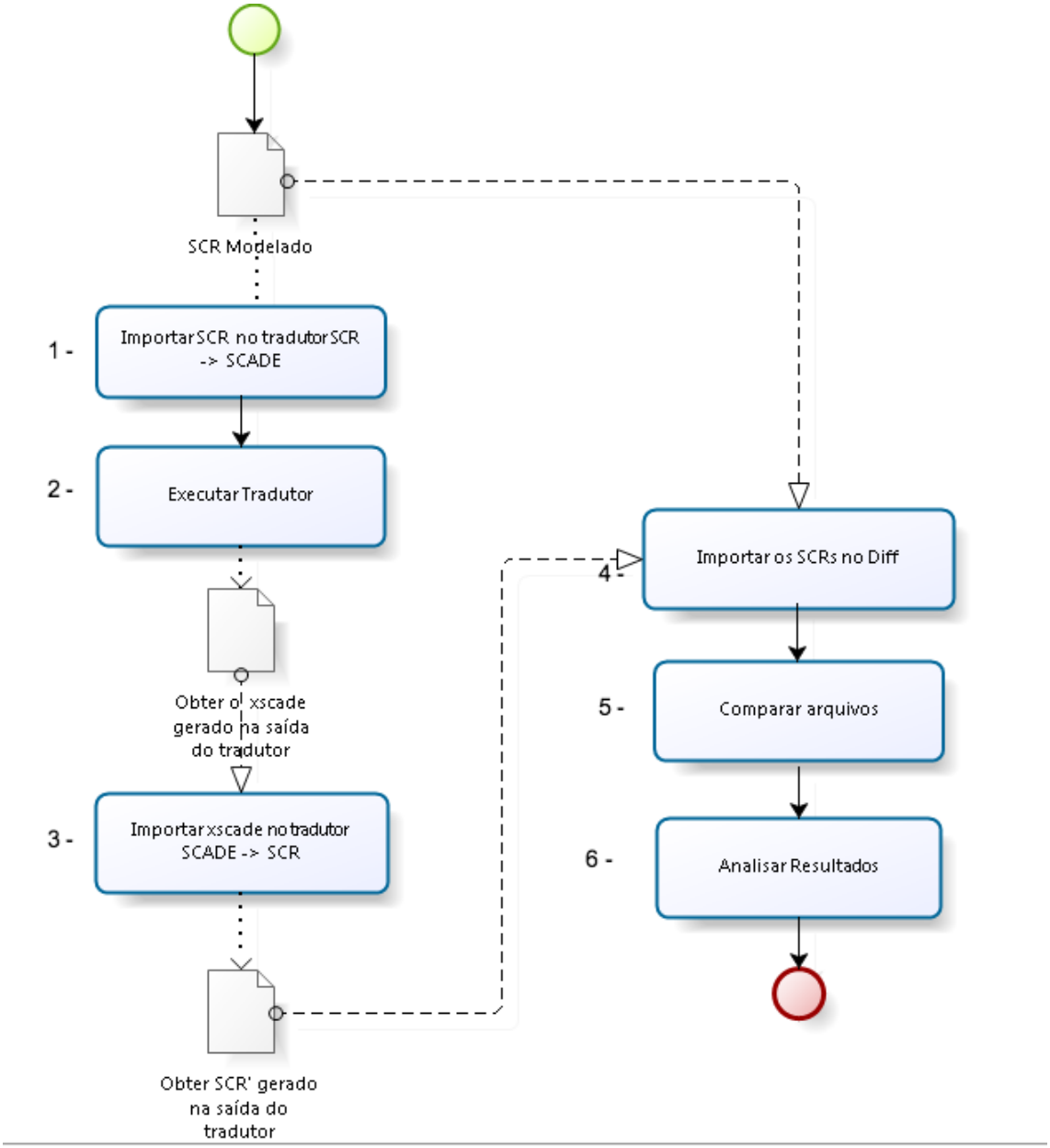
Tendo como ponto de partida o resultado da tradução para xscade a partir de um ponto SCR, utilizamos o tradutor utilizamos o tradutor a fim de obter uma especificação em SCR. Esta deverá ser igual a fonte do SCR inicial. Essa estratégia também verifica a conformidade entre o tradutor SCR -> xscade e xscade -> SCR. Se torna mais fácil de entender através da análise do diagrama de atividades presente na Figura 5 que explica as atividades individualmente.

1 - Importar SCR textual no tradutor SCR -> SCADE: O SCR encontrado na literatura que está no formato de SCR textual é importado no tradutor SCR > xscade, desenvolvido por Marcelo de Andrade (ANDRADE, 2013).

2 - Executar Tradutor SCR -> SCADE: Assim que for finalizada a criação de um arquivo SCR e modelada a especificação SCR, é possível executar o tradutor para gerar código xscade. O resultado da tradução é um arquivo com extensão xscade cujo conteúdo é similar ao Código 4.1;

3 - Importar aquivo xscade no tradutor xscade -> SCR: A saída gerada do tradutor SCR -> SCADE é importada no tradutor xscade -> SCR. Assim que for finalizada a modelagem do código xscade, é possível executar o tradutor para gerar especificações SCR. O resultado da tradução é um arquivo SCR, cuja estrutura já foi exibida anteriormente nas seções 2.3 e 3.

Figura 5 – Fluxo de validação da conformidade da tradução xscade - SCR



4 - Importar Arquivos SCRs no Diff: Com o arquivo SCR correspondente ao código SCADE em mãos, é possível importar o arquivo na ferramenta Diff que faz comparação entre arquivos textuais e importar o SCR modelado no tradutor SCR -> SCADE para que seja possível realizar a comparação entre o SCR da literatura usado por (ANDRADE, 2013) e o SCR de saída da ferramenta do presente trabalho.

A partir desse ponto os SCRs estão prontos para verificação.

5 - Comparar Arquivos: É possível comparar o arquivo textual SCR usado por (ANDRADE, 2013) vindo da literatura e do SCADE -> SCR, a ferramenta Diff faz automaticamente a comparação e exibe as linhas dos textos onde há divergências.

6 - Analisar Resultados: Após comparação dos SCRs textuais, a ferramenta Diff

Listing 4.1 – Ilustração de um código SCADE gerado pelo tradutor SCR > SCADE

```
<Package name="Safety_Injection_System">
  <declarations>
    <Type name="ySwitch">
      <definition>
        <Enum>
          <values>
            <Value name="Off">
              <pragmas>
                <ed:Value oid="!ed/enumValue/ySwitch_Off"/>
              </pragmas>
            </Value>
           <Value name="On">
            <pragmas>
                  <ed:Value oid="!ed/enumValue/ySwitch_On"/>
            </pragmas>
            </Value>
          </values>
        </Enum>
      </definition>
```

não exibe diferenças entres os arquivos evidenciando o tradutor está correto.

4.2 Verificando a conformidade tendo como ponto de partida xscade gerado pelo SCADE

Essa estratégia foca em gerar especificações em SCR a partir do xscade gerado por diagramas do SCADE. O estudo de caso Priority Command detalhado no Capítulo 5 foi modelado graficamente na ferramenta SCADE, gerado o xscade na mesma ferramenta e a partir desse xscade gerado o SCR no tradutor xscade -> SCR. O código de entrada é insumo para geração dos requisitos. É possível garantir que ao final da tradução toda a especificação será obtida de acordo com a certificação DO-178B. Segue fluxo na figura6:

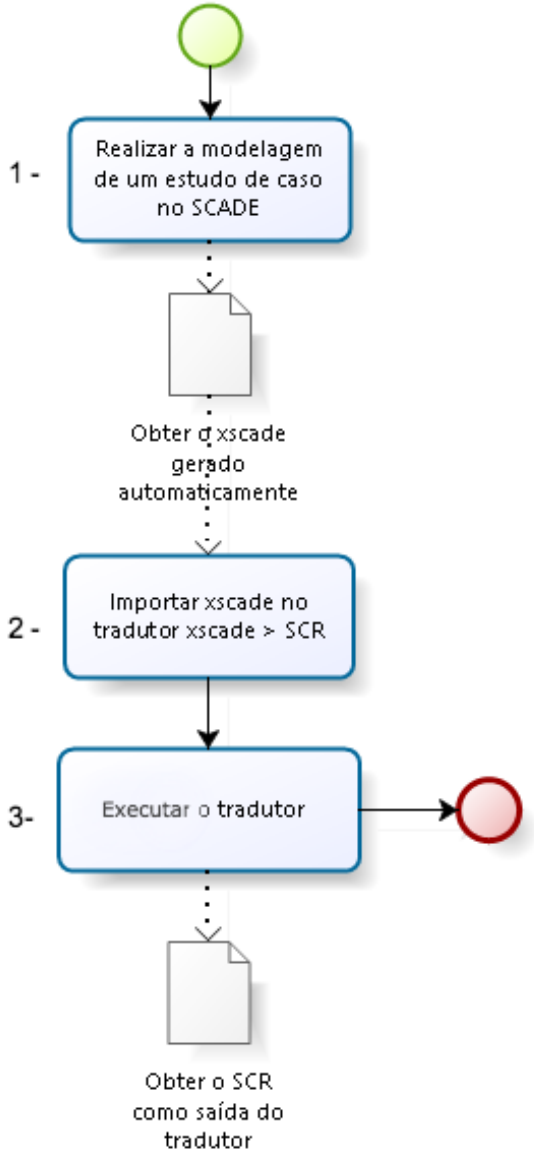
1 -Realizar modelagem do estudo de caso no SCADE: O estudo de caso é modelado graficamente no SCADE, e a partir disso o SCADE gera xscade automaticamente.

2 -Importar xscade no tradutor SCADE: O xscade é importado na tradutor xscade > SCR.

3 -Executar o tradutor: Após o xscade ser importado, o tradutor é executado para gerar SCR.

Obter o SCR atualizado: O SCR gerado corresponde à especificação formal do

Figura 6 – Fluxo de validação da conformidade tendo como ponto de partida xscade gerado pelo SCADE



estudo de caso.

Ao final dessa estratégia fizemos uso de T-VEC para construir os testes e do VGS para transformar a modelagem do T-VEC em casos de testes, garantindo a corretude da especificação com todos os testes aprovados. Para executá-los no código C gerado pelo SCADE foi usada a ferrameta CodeBlocks, mas poderia ter sido usado qualquer compilador C.

4.3 Aplicando Redundância Modular Tripla

Neste trabalho houve ainda a oportunidade de trabalhar com redundância modular tripla usando modelagens no SCADE e de execução de testes gerados automaticamente no VGS.

Usamos a reestruturação preservando a semântica para melhorar alguns sistemas de controle. Um dos exemplos foi a bomba de insulina de Sommerville. Esse estudo de caso se resume a um sensor de insulina que tem duas entradas e uma saída causando assim insegurança no resultado apresentado pelo sensor, pois se o sensor quebrar não é possível saber que o valor emitido pelo sensor está correto. Para sanar o problema de haver apenas um sensor foram adicionados mais dois sensores e assim é possível saber com mais exatidão qual valor está correto a partir do momento que dois ou os três sensores obtém o mesmo resultado. Serão criados testes no T-VEC e executados no código do primeiro sensor. Posteriormente Os mesmos testes executados no primeiro sensor serão executados após modificação da estrutura sem modificar a semântica do código quando mais sensores são adicionados.

Segue o fluxo usado na refatoração de redundância modular tripla na Figura 7:

1 - Modelar circuito no SCADE: O circuito da Figura 8, foi modelado em SCADE e após a modelagem obtido o xscade que é gerado automaticamente.

2 - Gerar código C no SCADE: Após o circuito ser modelado e o xscade ser gerado, é gerado código C automaticamente pelo SCADE. Um pré-requisito para a geração é ter executado a modelagem e ela não conter nenhum erro.

3 - Modelar SCR referente ao circuito no SCR: Após o código C estar pronto, foi modelado o circuito no T-VEC. A modelagem no T-VEC é criada usando a sintaxe SCR.

4 - Gerar testes com a ferramenta VGS: Após ter feito a modelagem do SCR no T-VEC, é possível gerar drivers de testes na ferramenta VGS. Para que o VGS possa gerar os drivers, é necessário fornecer arquivos de mapeamento que possuem a configuração necessária para indicar ao VGS como deve ser feito a geração de código C. Na Seção 2.5 há mais detalhes sobre o processo de geração dos drivers de testes.

5 - Criar projeto em um compilador C: Para que seja possível executar os drivers de testes gerados pelo VGS no código C gerado pelo SCADE, é necessário criar um projeto em uma ferramenta que compile código C. O arquivo de mapeamento para geração dos drivers de testes quando bem mapeados conseguem gerar código muito similar com código C.

6 - Executar testes no código C: Ao ter criado um projeto com todas as classes C geradas automaticamente e incluído uma nova classe no projeto com os drivers de testes, é possível executar o projeto e a saída será um arquivo com resultado dos testes, que tem

que ser similar ao resultado gerado pelo GVS.

Modelagem do circuito refatorado no SCADE:

O circuito foi refatorado e remodelado no SCADE para serem executados os mesmos drivers de testes do primeiro circuito e verificar se o resultado será o mesmo.

7 - Modelar circuito refatorado no SCADE: Após refatoração modelar o xscade gerado pelo SCADE no tradutor xscade > SCR

8 - Gerar código C no SCADE: Após a remodelagem no SCADE, a geração do xscade automaticamente, é possível gerar o código C certificado em um processo automático pela própria ferramenta.

9 - Criar projeto em um compilador C: Após ter o código C gerado, criou-se um novo projeto em um compilador C com o código C do circuito refatorado e os mesmo drivers de testes do circuito antes da refatoração.

10 - Executar testes criados pelo VGS antes da refatoração: O projeto foi compilado e obtido um novo resultado dos testes.

11 - Comparar resultado: Em ambos os circuitos o arquivo de saída gerado estava igual ao arquivo de saída de testes gerado pelo VGS.

12 - Analisar resultado: A partir do último passo, tendo verificado a conformidade ao resultado final de ambos os circuitos, houve uma mudança estrutural, mas o comportamento é o mesmo. Pode-se afirmar que é possível fazer uma refatoração na modelagem de um sistema e obter o mesmo resultado ao final.

4.4 Executando Testes

Para todo SCR gerado neste trabalho, em cada estudo de caso foram executados testes através da ferramenta TTM. Todos os testes passaram a mostrar que era possível traduzir xscade para SCR. Para executar os testes é necessário seguir o seguinte fluxo:

• Primeiro é necessário modelar a especificação usando T-VEC TTM. A modelagem no TTM é feita manualmente. Com TTM, é possível construir especificações usando uma sintaxe baseada em SCR. Idealmente, a modelagem com TTM deveria ser idêntica à modelagem usando a gramática definida para o tradutor, porém a modelagem com TTM possui algumas diferenças de sintaxe. A Figura 10 ilustra uma tabela modelada no TTM.

• O segundo passo é converter a especificação de TTM para VGS. Este é um procedimento automático fornecido pela própria TTM. São executadas análises estáticas simples sobre a especificação, garantindo a conformidade da especificação com a gramática de TTM. A Figura 4 mostra a janela Translation Options de TTM, pela qual é possível acionar o procedimento de conversão. Foi desmarcada a opção Generate Vectors for Outputs

Only para garantir que seriam gerados vetores de testes para todas as tabelas de variáveis auxiliares e tabelas de variáveis de saída. As outras opções de tradução não foram alteradas mantendo-se o valor padrão;

• O terceiro é compilar projeto VGS Com a especificação corretamente importada como um projeto na ferramenta VGS. O próximo passo é gerar os vetores de teste para cada tabela. Este é um procedimento automático no momento em que o projeto é compilado pela ferramenta.

• O quarto é a configuração de arquivos de mapeamento. Para ser possível aplicar os vetores de teste no código C gerado pelo SCADE, é necessário gerar drivers de teste. Para que o VGS possa gerar os drivers, é necessário fornecer arquivos de mapeamento que possuem a configuração necessária para indicar ao VGS como deve ser feito a geração de código C. É necessário gerar código C para cada vetor de teste gerado pelo VGS. Os arquivos de mapeamento são detalhados na Seção 2.5.

• Por último são gerados os drivers de teste que, após fornecerem os arquivos de mapeamento, é possível acionar o VGS para gerar os drivers de teste.

Figura 7 – Fluxo de refatoracão de redundância modelar tripla

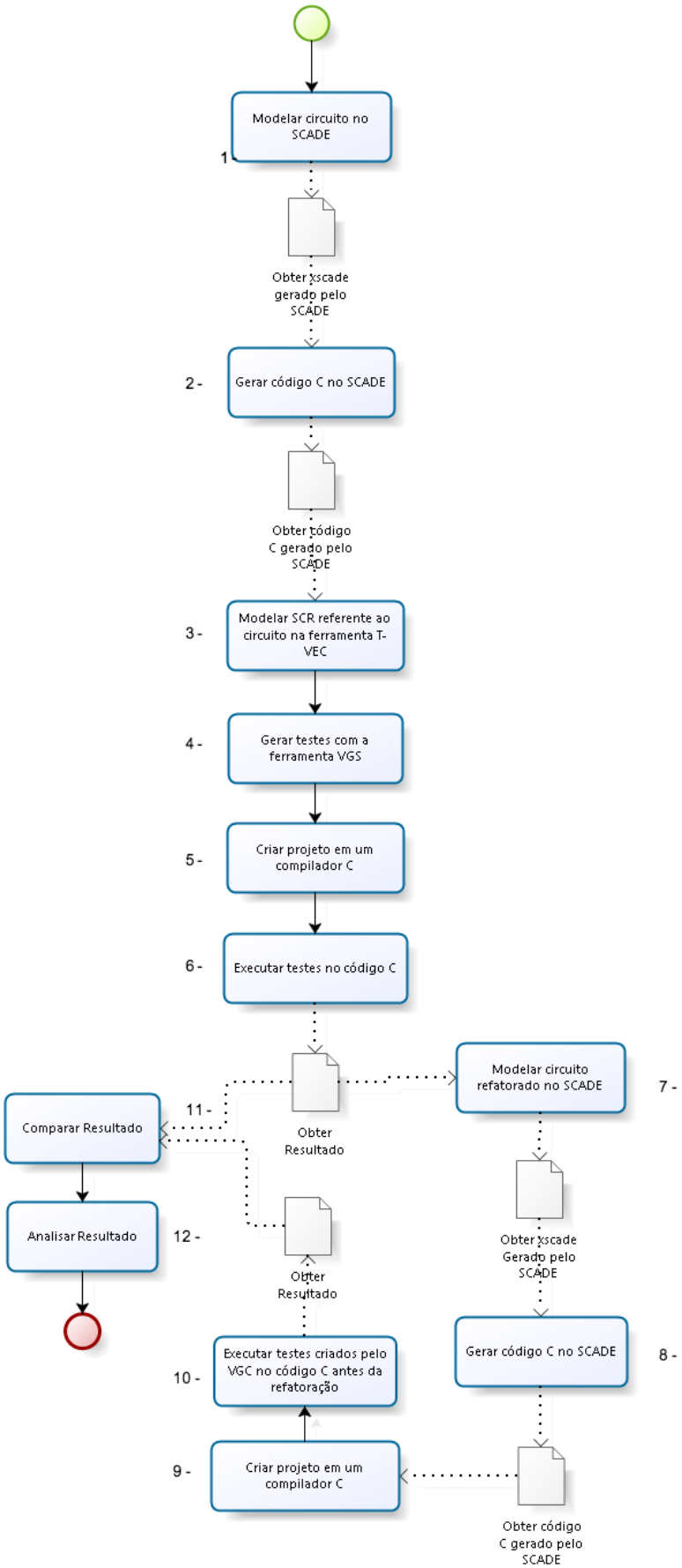


Figura 8 – Circuito modelado no SCADE

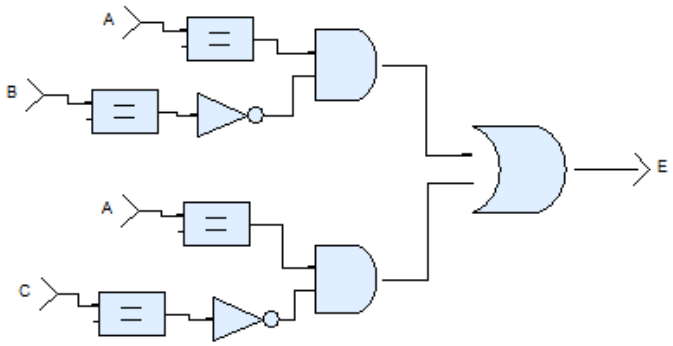


Figura 9 – Circuito Refatorado

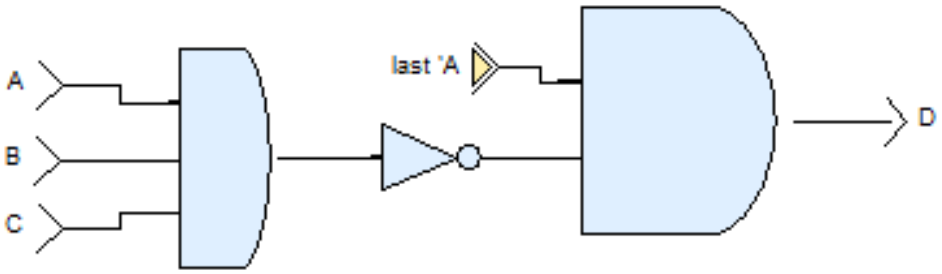


Figura 10 – Tabela modelada na ferramenta TTM

#	Assignment	Condition	Mode
1	Off	tOverridden	TooLow
2	On	NOT tOverridden	TooLow
3	On	FALSE	High Permitted
4	Off	TRUE	High Permitted

5 Prova de Conceito

Neste capítulo, apresentamos alguns exemplos presentes na literatura que serão usados no tradutor.

5.1 Safety Injection

O primeiro estudo é uma versão simplificada de um sistema de controle para injeção de segurança em uma usina nuclear adaptada de (HEITMEYER, 2010). Esse exemplo foi também utilizado por Marcelo Andrade (ANDRADE, 2013). O sistema que será especificado possui três sensores que monitoram valores do ambiente externo e um dispositivo de saída. O sistema também possui uma máquina de estados, um termo e algumas constantes. Basicamente o objetivo do sistema é monitorar a pressão da água no reator e, caso a pressão esteja em um nível considerado muito baixo, o sistema injeta uma substância refrigeradora.

O SCR deste estudo de caso foi modelado no tradutor SCR -> SCADE. Logo após o xscade foi modelado no tradutor xscade -> SCR. Por fim a partir do xscade foi gerado código C na ferramenta SCADE. Paralelamente o SCR foi modelado na ferramenta TTM e gerados testes automaticamente com a ferramenta VGS. Os testes foram executados no código C através de um compilador de C.

5.2 Priority Command - Automóvel

O segundo estudo é uma versão simplificada de um sistema real de controle de cruzeiro para automóvel adaptado de Heitmeyer (HEITMEYER; BHARADWAJ, 2000). O sistema monitora diversas variáveis do ambiente e usa esta informação para controlar a aceleração do automóvel. Basicamente, se a ignição estiver ativa, o motor em funcionamento e os freios não acionados, o automóvel entra no modo de controle de cruzeiro movendo a alavanca (mLever) para a posição const. Enquanto estiver no modo de controle de cruzeiro, a velocidade do automóvel determina se o controle de aceleração deve desacelerar, acelerar ou manter a velocidade atual. O motorista anula o controle de cruzeiro se pressionar os freios, reassume o controle se mover a alavanca para a posição resume ou sai do modo se mover a alavanca para off (ANDRADE, 2013).

Neste exemplo foi usado o mesmo processo de verificação do processo anterior.

5.3 Priority Command - Avião

O terceiro estudo de caso é uma versão simplificada de um sistema aviônico usado pela empresa de aviação. É composto por um sistema que decide se o piloto ou co-piloto terá a prioridade para controlar as varas laterais do avião com base em sua posição atual e um botão de prioridade. O sistema especificado tem três operadores com variáveis monitoradas e controladas.

Os estados apresentados de acordo com a situação (se o botão está pressionado ou não) são descritos abaixo:

Requisito 1 A função lógica Prioridade deve atribuir o valor 0 (zero) para comando do controle de saída quando:

- O botão prioridade da esquerda não está pressionado e;
- O botão prioridade da direta não está pressionado e;
- O comando da esquerda está em posição neutra e;
- O comando direito está em posição neutra.

Requisito 2 A função lógica Prioridade deve atribuir o valor 1 (um) a comando no controle de saída quando:

- O botão prioridade à esquerda não está pressionado e;
- O botão prioridade à direita não está pressionado e;
- O comando de prioridade à esquerda não está na posição neutra e;
- Comando direito está em posição neutra.

Requisito 3 A função lógica Prioridade deve atribuir valor 2 (dois) para o comando do controle de saída quando:

- O botão de prioridade à esquerda não está pressionado e;
- O botão prioridade à direita não está pressionado e;
- O comando da esquerda está em posição neutra e;
- O comando direito não está em posição neutra.

Requisito 4 A função lógica Prioridade deve atribuir valor 3 (três) para o comando do controle de saída quando:

- O botão de prioridade à esquerda não está pressionado e;
- O botão de prioridade à direita não está pressionado e;
- O comando esquerdo não está na posição neutra e
- O comando direito não está em posição neutra.

Requisito 5 A função lógica Prioridade deve atribuir valor 4 (quatro) para o comando do controle de saída quando:

- O botão de prioridade à esquerda é pressionado e;
- O botão direito de prioridade não está pressionado.

Requisito 6 A função lógica Prioridade deve atribuir valor 4 (quatro) para o comando do controle de saída quando:

- O botão de prioridade à esquerda é pressionado e;
- O botão direito de prioridade é pressionado e;
- O pedido de prioridade esquerdo ocorre antes do pedido de prioridade direita.

Requisito 7 A função lógica Prioridade deve atribuir o valor 5 (cinco) para o comando do controle de saída quando:

- O botão de prioridade à esquerda não está pressionado e;
- O botão de prioridade à direita é pressionado.

Requisito 8 A função lógica Prioridade deve atribuir o valor 5 (cinco) para o comando do controle de saída quando:

- O botão de prioridade à esquerda é pressionado e;
- O botão direito de prioridade é pressionado e;
- O pedido direito de prioridade ocorre antes do pedido de prioridade esquerdo.

Esse estudo de caso foi modelado na ferramenta SCADE, paralelamente o SCR do exemplo foi modelado no tradutor SCR - > xscade. Ao final foram comparados o xscade gerado pelo SCADE com o xscade gerado.

5.4 Aplicando Formal Normal Disjuntiva

Na lógica booleana, uma forma normal disjuntiva (FND) é uma normalização de uma fórmula lógica a qual é uma disjunção de cláusulas conjuntivas. Como uma forma normal, a FND é útil em provas automáticas de teoremas. Uma fórmula lógica é considerada uma FND se, e somente se, for uma disjunção de uma ou mais conjunções de um ou mais literais. Como na forma normal conjuntiva (FNC) exibida na Figura 11, os únicos operadores proposicionais na FND são "e", "ou"e "não". O operador "não"pode ser usado apenas como parte de um literal, o qual significa que pode apenas preceder uma variável proposicional.

O objetivo aqui não é lidar com uma fórmula da lógica proposicional que não é tão complicada, mas demonstrar que, tendo um projeto inicial, pode haver uma modificação

estrutural que mantém a semântica e os testes gerados asseguram isso.

Esse estudo de caso foi modelado na ferramenta SCADE, gerado o xscade, o xscade foi modelado no tradutor xscade -> SCR. Houve a geração de testes com o TTM-TVEC, logo após foi gerado o código C na ferramenta SCADE, os testes gerados pelo TTM-TVEC foram executados no código C. Em seguida foi feita a modificação do circuito para à FNC.

Figura 11 – Forma Normal Disjuntiva usada na refatoração

$$
\begin{array}{ll}
 & \mathcal{A} \wedge \neg(\mathcal{B} \wedge \mathcal{C} \wedge \mathcal{A}) \\
\equiv & \text{“Predicate law A.18”} \\
 & \mathcal{A} \wedge (\neg\mathcal{B} \vee \neg\mathcal{C} \vee \neg\mathcal{A}) \\
\equiv & \text{“Predicate law A.7”} \\
 & (\mathcal{A} \wedge \neg\mathcal{B}) \vee (\mathcal{A} \wedge \neg\mathcal{C}) \vee (\mathcal{A} \wedge \neg\mathcal{A}) \\
\equiv & \text{“Predicate law A.15”} \\
 & (\mathcal{A} \wedge \neg\mathcal{B}) \vee (\mathcal{A} \wedge \neg\mathcal{C}) \vee \mathsf{false} \\
\equiv & \text{“Predicate law A.12”} \\
 & (\mathcal{A} \wedge \neg\mathcal{B}) \vee (\mathcal{A} \wedge \neg\mathcal{C}) \ .
\end{array}
$$

5.5 Bomba de Insulina

A bomba de insulina de Somerville (SOMMERVILLE, 2006) é um sensor que calcula a dose de insulina a ser liberada por uma medição do nível atual de açúcar no sangue, comparando-o a um nível previamente medido e calculando a dose requerida. O sistema deve medir o nível de açúcar no sangue e liberar a insulina, se requerida, a cada 10 minutos. A quantidade de insulina a ser liberada deve ser calculada de acordo com a leitura medida por um sensor. No presente trabalho a bomba de insulina é usada de duas formas:

1 - Existe apenas uma bomba, com duas entradas e uma saída, onde o resultado obtido pela única saída é considerado correto. É possível visualizar a modelagem desse modo na Figura 12.

2 - Como exemplo de redundância modular tripla foram adicionadas mais duas bombas, onde é possível obter mais de um resultado na saída de um sensor e comparar quais resultados estão em conformidade. Ou seja, se é obtido o resultado do cálculo da quantidade de insulina a partir de apenas um sensor, se esta saída está errada, não será possível saber, então adiciona-se outro sensor. Mas se um sensor falhar, novamente os resultados vão diferir e ainda assim não é possível saber qual sensor está funcionando corretamente na Figura 12. Para sanar este problema é adicionado mais um sensor, sendo assim, há 3 sensores, onde cada sensor calcula a quantidade de insulina a ser injetada

em um paciente e ao final os resultados são comparados. A saída que for obtida como resultado em mais de um sensor será considerada como resultado correto ver Figura 13. Esta medida diminui a probabilidade de um sensor falhar e o erro não ser percebido.

Figura 12 – Sistema modelado apenas com uma bomba de insulina

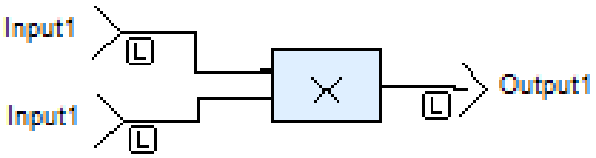
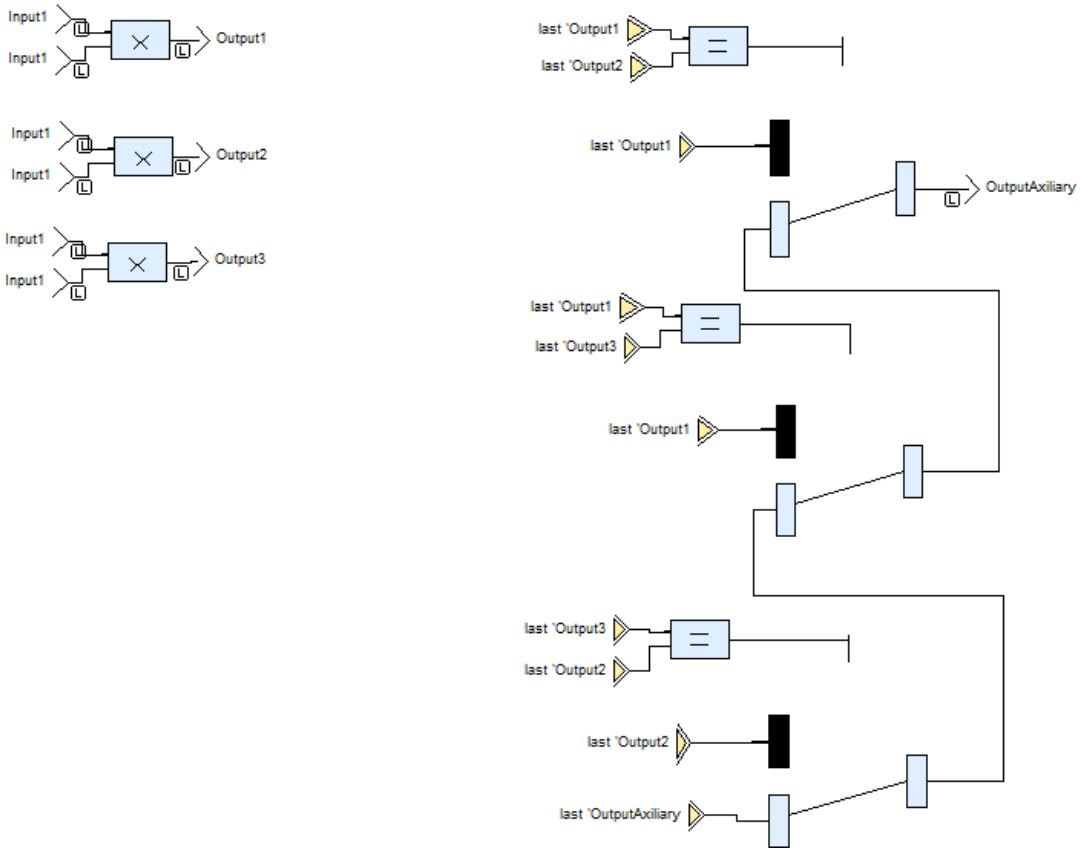


Figura 13 – Sistema modelado com 3 bombas de insular



6 Conclusão

Esse trabalho começou com a descoberta do objetivo do trabalho de (ANDRADE, 2013), que o tradutor resolve o problema encontrado na indústria de desenvolvimento de software aviônico. Como relatado na dissertação dele, o tradutor facilita o trabalho de desenvolvedores, automatizando a geração de código com os requisitos e facilitando que o código seja compatível com a certificação DO-168B. Pois como o código é gerado através de SCR, a linguagem de definição dos requisitos, faz com que o código sempre esteja de acordo com a especificação.

Logo após começaram o entendimento do propósito do trabalho de (ANDRADE, 2013) começamos a entender e pesquisar sobre o desenvolvimento do atual trabalho. Descobrimos então que poderíamos através da engenharia reversa criar um tradutor de xscade -> SCR e obter um ciclo completo de desenvolvimento de SCR e xscade. Também provaria que a volta de xscade para SCR é possível, assim como poderíamos apenas ter um xscade e obter as especificações.

Assim, começamos entender como seria a representação de cada elemento de xscade em SCR, que é um processo complexo, onde demanda grande esforço do xscade por ter um formato XML e ser muito extenso. O próximo passo foi o estudo das ferramentas usadas no trabalho, a modelagem usando o plugin Spoofax para escrever código em Stratego/XT. A validação executada foi auxiliada pela ferramenta T-VEC que faz a modelagem de tabelas em SCR, integra com o TTM, que automaticamente gera drivers de testes que podem ser executados no código C gerado pela ferramenta SCADE a partir do xscade.

O desenvolvimento do trabalho exigiu a aproximação com estas tecnologias, desconhecidas anteriormente pela autora. Uma grande dificuldade foi a falta de documentação do esquema XML para SCADE (xscade). A fonte para a escrita da sintaxe em SDF foi primordialmente o resultado da tradução de SCR para xscade desenvolvido por Andrade (ANDRADE, 2013). Um outro fator limitador foi a não renovação da licença da ferramenta SCADE na fase final do desenvolvimento do trabalho, o que limitou a realização de validações.

6.1 Trabalhos relacionados

Dentro da área de requisitos com notação formal visando sistemas de controle e tendo como alvo código certificado, de acordo com a DO-178B, o trabalho relacionado mais próximo é o de Andrade (ANDRADE, 2013). Em particular, nosso trabalho visou a tradução de artefatos de projeto (xscade), para requisitos (SCR). Dessa forma, o tradutor

desenvolvido neste trabalho, ao complementar o tradutor desenvolvido por Andrade (ANDRADE, 2013), fecha o ciclo no desenvolvimento, levado ao que é conhecido como "road trip engineering".

Um trabalho encontrado na literatura faz a ligação entre Simulink e Scade. Nele é apresentado uma plataforma distribuída usando uma abordagem de ponta a ponta em camadas para a concepção e implementação de software. A abordagem compreende uma camada de modelagem e simulação de alto nível (Simulink), uma camada de programação e validação de nível médio (SCADE / Luster) e uma camada de execução de baixo nível (TTA). Primeiro foi feito um tradutor de Simulink para Lustre. Segundo, um conjunto de extensões de tempo real e de distribuição de código para o Lustre. Em terceiro lugar, técnicas de implementação para a decomposição de um programa Lustre em tarefas e mensagens, programação de tarefas e mensagens nos processadores e no barramento, distribuição do código Lustre na plataforma de execução e geração do código. (CASPI ADRIAN CURIC, 2003)

Este trabalho descreve como ferramentas de análise formal podem ser inseridas em um processo de desenvolvimento baseado em modelo para diminuir custos e aumentar a qualidade de software de aviônica crítico. A adoção pela indústria aeroespacial de ferramentas de desenvolvimento baseadas em modelos como o Simulink e o SCADE Suite TM está removendo barreiras ao uso de métodos formais para a verificação de software de aviônica crítico. Os sistemas aeroespaciais militares incluirá sistemas de controle avançado, cujo tamanho e complexidade desafiarão as atuais abordagens de verificação e validação. Por os métodos formais usarem a matemática para provar que os modelos de design de software atendem às suas necessidades, a confiança na segurança e na correção do software podem aumentar consideravelmente. A verificação formal de software se tornam mais práticos com as ferramentas de análise formal importantes desses modelos para garantir que os defeitos de projeto sejam identificados e corrigidos no início do ciclo de vida. (WHALEN DARREN COFER, 2008)

A necessidade de ferramentas eficientes para verificar a confiabilidade cresce à medida que os sistemas críticos de segurança aumentam em tamanho e complexidade. Neste outro trabalho relacionado apresentamos uma ferramenta que ajuda os engenheiros a projetar sistemas seguros e confiáveis. Os sistemas são confiáveis se eles continuarem operando com segurança quando os componentes falharem. A ferramenta é o núcleo do *Scade Design Verifier* integrado no Scade, um produto desenvolvido pela *Esterel Technologies*. Esta ferramenta estende automaticamente os modelos Lustre injetando falhas, usando bibliotecas de falhas típicas. A ferramenta pode calcular combinações mínimas de falhas que rompem a segurança dos sistemas, o que é semelhante à Análise de Árvore de Falhas. O documento inclui verificações bem-sucedidas de exemplos da indústria aeronáutica. Também permite executar o Modo de Falha e a Análise de efeitos,

que consiste em verificar se os sistemas permanecem seguros quando os componentes selecionados falham. (ABDULLA JOHANN DENEUX, 2003)

6.2 Trabalhos futuros

Dentre os trabalhos futuros, citamos primeiramente a integração dos tradutores desenvolvidos por Andrade (ANDRADE, 2013) e neste trabalho. Em uma mesma ferramenta, o ciclo entre requisitos e projetos (e o inverso) ficarão integrados.

Gustavo Carvalho (CARVALHO, 2013) criou um tradutor de linguagem natural para SCR, seria possível também fazer uma integração com esse tradutor e obter automaticamente SCR a partir de linguagem natural.

Um outro trabalho é o desenvolvimento de sistemas representativos da indústria, assim teríamos exemplos mais reais e seria possível nos aproximar cada vez mais dos problemas reais do mercado incluindo melhorias no tradutor.

Usar a ferramenta Simulink (MATHWORKS, 1994) desenvolvido pela MathWorks, é um ambiente de programação gráfica para modelagem, simulação e análise de sistemas dinâmicos multidomínio. Sua interface principal é uma ferramenta gráfica de diagrama de blocos e um conjunto personalizável de bibliotecas de blocos. Por ser um ferramenta atual e muito usada para sistemas críticos, poderia haver a construção do ciclo Simulink -> xscade. Isso faria com que os desenvolvedores tivessem duas opções de especificação e a facilidade na geração do código.

Como feito nesse trabalho, poderia também haver a construção de um tradutor que fizesse a engenharia reversa de xscade -> Simulink, fazendo com que fosse possível obter a especificação através do código.

Referências

ABDULLA JOHANN DENEUX, G. S. H. A. O. A. P. A. *Designing Safe, Reliable Systems Using Scade*. 2003. Institute for Informatics, University of Potsdam. Disponível em: <http://link.springer.com/chapter/10.1007%2F11925040_8#page-1>. Acesso em: 20 nov 2016. 55

ANDRADE, M. C. M. de. *Gerando modelos SCADE a partir de especificações descritas em SCR*. Dissertação (Dissertação de Mestrado) — Universidade Federal de Pernambuco, 2013. 7, 8, 14, 15, 16, 24, 28, 29, 35, 39, 40, 48, 53, 54, 55

BERRY, G. Scade: Synchronous design and validation of embedded control software. *Esterel Technologies gerard.berry@esterel-technologies.com*, p. 15, 2008. 15, 16, 21

BRAND P. KLINT, V. J. M. *The Syntax Definition Formalism SDF*. 2008. CodeBlocks. Disponível em: <http://homepages.cwi.nl/~daybuild/daily-books/learning-about/sdf/sdf.pdf>. Acesso em: 10 out 2016. 26, 28, 29

BRAVENBOER, B.; VISSER, E. *Program Transformation with Scoped Dynamic Rewrite Rules*. [S.l.]: Department of Information and Computing Sciences Url = https://pdfs.semanticscholar.org/20ad/7f3de49340dd6280e591b4ad639ef6ab1a40.pdf, Urlaccessdate = 10 out 2016, Utrecht University, 2005. 26

BUTLER, R. W. *Introduction to Formal Methods*. 2012. National Aeronautics and Space Administration. Disponível em: <https://shemesh.larc.nasa.gov/PVSClass2012/pvsclass2012/lectures-printer/Intro_FM.pdf>. Acesso em: 29 set 2016. 17

CARVALHO, G. H. P. de. *Modelagem, Verificação e Teste Composicional de Sistemas com Aplicações na Indústria Aeronáutica*. UFPE - Universidade Federal de Pernambuco, 2013. Disponível em: <http://www.escavador.com/sobre/6605085/gustavo-henrique-porto-de-carvalho>. Acesso em: 9 nov 2016. 55

CASPI ADRIAN CURIC, A. M. C. S. S. T. P. N. P. *From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications*. 2003. ACM New York, NY, USA. Disponível em: <https://www.mathworks.com/products/simulink/?requestedDomain=www.mathworks.com>. Acesso em: 20 nov 2016. 54

COSTA, H. *Desenvolvimento formal de um sistema de sinalização ferroviária de acordo com o normativo CENELEC usando o SCADE*. 2009. Universidade Da Beira Interior. Disponível em: <https://ubibliorum.ubi.pt/bitstream/10400.6/3746/1/thesis.pdf>. Acesso em: 06 out 2016. 23

ELMQVIST JERKER HAMMARBERG, S. N.-T. D. S. J. *Embedded Systems Simulation and Verification*. UFPE - Universidade Federal de Pernambuco, 2005. Disponível em: <http://www.ida.liu.se/~TDDC05/material/ScadeTutorial.pdf>. Acesso em: 10 nov 2016. 9, 22

H., G. et al. Language design with the spoofax language workbench. *Esterel Technologies gerard.berry@esterel-technologies.com*, p. 15, 2014. 16, 26

HALBWACHS, N. et al. The synchronous data flow programming language lustre. *IEEE*, p. 7, 1991. 20

HEITMEYER. Requirements models for requirements models for system safety and security. *Naval Research Laboratory*, p. 21, 2010. 48

HEITMEYER; BHARADWAJ. Applying the scr requirements method to the light control case study. *Future Computing Systems*, p. 29, 2000. 14, 16, 18, 20, 38, 48

JUNIOR, F. F. S. *Falhas no Gerenciamento de Projetos*. 2010. TecHoje. Disponível em: <http://www.techoje.com.br/site/techoje/categoria/detalhe_artigo/972>. Acesso em: 27 set 2016. 15

MATHWORKS, I. T. *T-VEC Tabular Modeler*. 1994. The MathWorks, Inc. Disponível em: <https://www.mathworks.com/products/simulink/?requestedDomain=www.mathworks.com>. Acesso em: 20 nov 2016. 55

NASCIMENTO, R. J. O. do. *Como usar os Métodos Formais no desenvolvimento de Software*. 2015. DevMedia. Disponível em: <http://www.devmedia.com.br/como-usar-os-metodos-formais-no-desenvolvimento-de-software/31339>. Acesso em: 27 set 2016. 14, 15

ROCHA, F. G. *A importância dos testes para a qualidade do software*. 2014. DevMedia. Disponível em: <http://www.devmedia.com.br/a-importancia-dos-testes-para-a-qualidade-do-software/28439>. Acesso em: 27 set 2016. 14

RTCA, R. T. C. for A. *Software Considerations in Airborne Systems and Equipment Certification - DO-178B*. 1992. Radio Technical Commission for Aeronautics RTCA. Disponível em: <hhttp://afuzion.com/>. Acesso em: 27 set 2016. 14

RUSHBY, J. Formal methods and the certification of critical systems. *SRI International*, p. 319, 1993. 17

SOMMERVILLE, I. *Software Engineering*. 6. ed. Boston, MA, USA: CACM Digital Library, 2006. Acesso em: 27 set 2013. 14, 51

TEAM, C. *T-VEC Tabular Modeler*. 2012. CodeBlocks. Disponível em: <http://www.codeblocks.org/>. Acesso em: 10 out 2016. 25, 39

TEAM, T.-V. *T-VEC Tabular Modeler*. 2012. T-VEC. Disponível em: <https://www.t-vec.com/solutions/ttm.php>. Acesso em: 10 out 2016. 23

VISSER, E. *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems*. Institute of Information and Computing Sciences - Utrecht University, 2004. Disponível em: <http://dspace.library.uu.nl/bitstream/handle/1874/18038/visser_04_program_transformation.pdf?sequence=2>. Acesso em: 14 nov 2016. 27

VOELTER, M.; VISSER, E. *Product Line Engineering using Domain-Specific Languages*. IEEE Software Computer Society, 2011. Disponível em: <http://www.voelter.de/data/pub/VoelterVisser-PLEusingDSLs.pdf>. Acesso em: 22 out 2016. 31

WHALEN DARREN COFER, S. M.-B. H. K. W. S. M. *Integration of Formal Analysis into a Model-Based Software Development Process*. 2008. Springer Berlin Heidelberg. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-540-79707-4_7#page-1>. Acesso em: 20 nov 2016. 54