Desenvolvimento formal de um sistema de sinalização ferroviária de acordo com o normativo CENELEC usando o SCADE

UNIVERSIDADE DA BEIRA INTERIOR
Covilhã | Portugal

Henrique Costa

Desenvolvimento formal de um sistema de sinalização ferroviária de acordo com o normativo CENELEC usando o SCADE

DISSERTAÇÃO

referente aos trabalhos de investigação conducentes
à obtenção do grau de

MESTRADO

em

TECNOLOGIAS E SISTEMAS DE INFORMAÇÃO

por

Henrique Costa

RELEASE - Computation Group
Departamento de Informática
Universidade da Beira Interior
Covilhã, Portugal
www.di.ubi.pt/~release

EFACEC
Sistemas de Electrónica, S.A.
Divisão de Transportes
Maia, Portugal
www.efacec.pt

Desenvolvimento formal de um sistema de sinalização ferroviária de acordo com o normativo CENELEC usando o SCADE

Resumo

Os sistemas informáticos do sector ferroviário regem-se pelas normas EN 50126 - *Railway Applications: The specification and demonstration of Reability, Avaiability, Maintainnability and Safety (RAMS)*, EN 50128 *Railway Applications: Software*[1] *for railway control and protection systems* e EN 50129 - *Railway Applications: Safety related electronic systems for signalling*. Nomeadamente a norma EN 50128, que se refere ao desenvolvimento de software, apresenta um conjunto de recomendações e uma metodologia, com base num ciclo de desenvolvimento em V, para assegurar a correcção do software e a confiança neste. De facto, a norma EN 50128 não impõe nenhuma técnica ou ferramenta na concepção do software pretendido. Assim, a metodologia de desenvolvimento e a selecção das ferramentas fica à responsabilidade da entidade que concebe o software estando esta apenas sujeita à certificação segundo a referida norma. Dependendo do sistema em desenvolvimento, a norma EN 50128 classifica o software em 5 níveis de integridade e segurança (*Safety Integrity Level* (SIL)), onde o nível mais baixo é o 0 (SIL0) e corresponde a software onde não existem consequências numa eventual falha, e o nível mais alto é o 4 (SIL4) onde uma falha do software tem um impacto catastrófico. Nos níveis 3 e 4 da tabela SIL, a norma EN 50128 recomenda vivamente a utilização de métodos formais na fase de especificação do software. O *Safety Critical Application Development Environment (SCADE) Suite 6.0* é uma plataforma para o desenvolvimento de sistemas críticos. Integra um conjunto de ferramentas, para especificação, verificação e análise de modelos, e é certificada pela CENELEC EN 50128 até ao nível 4 da tabela SIL. Foi utilizada, neste trabalho, não só para especificar o sistema, mas também, para o verificar e provar a sua correcção.

Para além de apresentar uma metodologia, com base na ferramenta formal *SCADE*, aplicável no desenvolvimento de sistemas ferroviários, este documento evidencia, com uma aplicação prática, a aplicabilidade do método. O caso de estudo utilizado é um sub-sistema de um sistema de sinalização completo, mais precisamente, uma Passagem de Nível (PN).

[1]Os estrangeirismos *software*, *hardware*, *input*, *output* são termos bem difundidos e aceites na comunidade para a qual este documento se dirige. Por esta razão, a tipografia destas palavras e suas conjugações não serão alteradas ao longo do documento

Autor: Henrique Costa
N. Aluno: M2529

Orientação:

Orientador: Prof. Dr. Simão Melo de Sousa - Universidade da Beira Interior
Co-orientador: Eng. Luís Roboredo - EFACEC

Agradecimentos

O meu agradecimento ao Prof. Dr. Simão Melo de Sousa pela sua disponibilidade e coordenação desta dissertação de mestrado. Igualmente ao Eng. Luís Roboredo pelo apoio e confiança demonstradas e pela oportunidade de efectuar este trabalho em ambiente real, o que muito contribuiu para o sucesso do mestrado.

Agradeço aos colegas mais directos de trabalho, Raul Esteves, Claúdio Sagres e José Veiga pela ajuda, ensinamentos e companheirismo demonstrado durante toda a fase do projecto.

O meu eterno obrigado, aos pais, aos avós e à irmã. Foram eles que nos momentos mais difíceis me mostraram que uma vida sem esforço não tem nada de bom para nos oferecer. Ao meu afilhado que genuinamente me alegrou nos momentos em que mais nada me faria sorrir. Ao grande amor da minha vida, que encheu de felicidade o meu coração em cada momento deste trabalho.

Henrique Costa
15 de Junho de 2009

Conteúdo

Lista de Figuras

Acrónimos

API *Application Program Interface* 4
BDD *Binary Decision Diagram* 18
CENELEC *European Committee for Electrotechnical Standardization* 29
CPU *Central Processing Unit* 58
DOORS *Dynamic Object Oriented Requirements System* 5
FSM *Finite State Machine* 18
HTML *Hyper Text Markup Language* 41
IDE *Integrated Development Environment* 35
INE Instituto Nacional de Estatística 2
KCG *SCADE Code Generator* 37
MF Métodos Formais 7
MTC *Model Test Coverage* 45
PN Passagem de Nível i
PN's Passagens de Nível 2
PLC *Programmable Logic Controller* 59
PLC's *Programmable Logic Controllers* 3
RAMS *Reability, Avaiability, Maintainnability and Safety* i
RMG *Requirements Management Gateway* 52
SCADE *Safety Critical Application Development Environment* i
SDL *Specification and Description Language* 32
SI Sistemas Informáticos 7
SIL *Safety Integrity Level* i
SSM *Safe State Machine* 38
XML *Extensible Markup Language* 50

Capítulo 1

Introdução

Os meios de transporte, mais do que mover pessoas ou cargas, são também canais de comunicação entre povos e culturas. Nomeadamente o transporte ferroviário foi responsável por mudanças profundas nas comunidades que dele usufruíram. Ainda nos dias de hoje este meio de transporte influencia significativamente cada região por onde passa. Assim, é cada vez mais importante assegurar a qualidade do transporte ferroviário, quer por meio de prestação de serviços, eficientes e seguros quer pela eficácia dos mesmos. O crescimento deste meio de transporte obriga a aplicação de tecnologia electrónica fazendo com que este seja mais rentável, cómodo e seguro. A segurança, inevitavelmente, é exigida, e os critérios de avaliação são muito rígidos. É neste contexto que surge este documento. Este, resulta da implementação de uma metodologia, para o desenvolvimento formal de software para sistemas ferroviários, recorrendo a métodos e ferramentas. Em particular, focaremos a nossa atenção e esforços na aplicação de métodos formais, utilizando ferramentas e técnicas adequadas, na especificação de sistemas de sinalização ferroviária. Como caso de estudo vamos utilizar uma PN. As PN's são um elemento presente em todos os sistemas de sinalização ferroviária e alvo de grande atenção por parte das entidades responsáveis pelo tráfego em ferrovias. Para perceber em que contexto aparece este trabalho apresentamos, a seguir, a importância dos caminhos ferroviários, dos sistemas informáticos que os gerem e da qualidade dos referidos sistemas.

1.1 Transporte Ferroviário

Amado ou odiado, conforme a vontade de modernidade patenteada por cada pessoa, o comboio tem já uma história com cerca de duzentos anos. Em Portugal teve início há cerca de 150 anos, mais exactamente a 28 de Outubro de 1856, quando partiu da estação de Santa Apolónia o comboio '*D. Pedro V*' que, pela primeira vez, fez o percurso entre Lisboa e o Carregado, inaugurando, assim, o primeiro troço do caminho de ferro construído em Portugal [11]. O aparecimento da via férrea e a adaptação da máquina a vapor ao comboio são marcos históricos no desenvolvimento da Humanidade a nível técnico, tecnológico, cultural e intelectual. Foi exactamente por mérito do comboio que as populações, outrora distantes, se aproximaram. Estabeleceram-se ligações entre regiões, antes praticamente separadas,

permitindo atenuar ou esbater o isolamento a que muitas dessas regiões estavam votadas. O comboio pode, até, ser visto como o primeiro meio de globalização. A aceitação do comboio foi-se acentuando com o passar dos anos, não só porque os caminhos de ferro se tornaram numa das "modas" do século XIX, mas especialmente porque os benefícios dos comboios se revelaram enormes. Em Portugal essa tendência foi vincada na era do Liberalismo, que previa a modernização do país com base nas comunicações - e que tinha o comboio como vértice central - o caminho de ferro foi conquistando espaço entre os meios de transporte dos finais do século XIX e inícios do século XX. Portugal chegou mesmo a contar com mais de 3100 quilómetros de linhas. Hoje em dia, e graças à evolução que os comboios naturalmente sofreram, esses efeitos positivos multiplicaram-se e foram optimizados. Desde logo, a segurança proporcionada pelo caminho de ferro surge como um dos aspectos positivos dos transportes ferroviários. Também ao nível ambiental, o comboio apresenta argumentos de peso, pois, se pensarmos que os comboios têm vantagem de transportar "várias dezenas de camiões" ao mesmo tempo e com um impacto ao nível da poluição bem menor, não teremos dúvidas em afirmar que esta capacidade de transporte se assumirá menos prejudicial ao ambiente. Daí que seja claro que o comboio é ambientalmente mais sadio do que os outros tipos de transporte.

O Instituto Nacional de Estatística (INE), num documento de informação à comunicação social, datado de 29 de Março de 2005 [10], relativamente à evolução dos transportes ferroviários, indica que *'Em 2004 o transporte ferroviário pesado de passageiros registou um aumento homólogo de 1,4%, enquanto o transporte ferroviário de mercadorias cresceu 9,6%. No mesmo ano, o transporte ferroviário ligeiro aumentou 4,5% no número de passageiros transportados, face a 2003*. Mais ainda, segundo o mesmo documento *'...Em 2004 foram transportados nos Metropolitanos de Lisboa e Porto cerca de 190,1 milhões de passageiros, o que representou um acréscimo de 4,5% face ao ano anterior. De referir que no referido ano foram disponibilizados pelos sistemas de Metropolitano de Lisboa e Porto cerca de 4 194 milhões de Lugares-Quilómetro (+17,3% do que em 2003), tendo o volume de transporte atingido os 824,5 milhões de passageiros-quilómetro...'*. Assim se pode ver a importância do transporte ferroviário no passado da nossa cultura e crescimento enquanto país, ver também que nos dias que correm este meio é ainda o preferido, principalmente por entidades indutriais e comerciais para o transporte das suas matérias-primas e distribuição dos seus produtos, respectivamente. É importante verificar também que este meio de transporte é visto como aquele que possui maior margem de crescimento, sobretudo quando se analisam os aspectos ambientais, económicos e de segurança.

1.2 Sistemas Ferroviários

Inevitavelmente, com o crescimento das infra-estruturas ferroviárias, a complexidade da gestão das mesmas cresceu de forma proporcional. O que antes eram algumas linhas com alguns cruzamentos e Passagens de Nível (PN's) passaram agora a ser ferrovias nas áreas metropolitanas, com várias intersecções entre linhas férreas, rodovias e peões. Naturalmente, o tamanho e a complexidade são acompanhados por um risco associado. Assim, surgiu a necessidade de controlar o risco do transporte ferroviário recorrendo a meios mecânicos

primeiro, eléctricos em segundo e mais recentemente a meios electrónicos. Pretende-se, com estes, evitar o inerente erro humano no manuseamento de dispositivos de sinalização, eliminando eventuais erros que causariam perda de vidas humanas e consequente perda de confiança num meio de transporte que tem tudo para ser o mais seguro.

No sector dos transportes ferroviários o sistema de maior criticidade quanto ao risco de acidente são os de sinalização. Estes sistemas são responsáveis por evitar o choque frontal entre comboios, o choque em cadeia, o choque com veículos de outros tipos de vias, descarrilamento,... etc. Para tal as ferrovias são complementadas com dispositivos mecânicos de mudanças de direcção (agulhas), dispositivos de protecção de descarrilamento que são comandados por instruções de software, mediante os dados existentes no ambiente num determinado momento. Tais sistemas, pela importância e risco associado, possuem características que os diferenciam dos convencionais ou mais vulgares. São sistemas onde se exige uma velocidade de resposta quase imediata a determinados eventos sob pena de causar prejuízos económicos e, mais preocupante ainda, causar perdas humanas. Assim, sistemas para o sector ferroviário, nomeadamente software de sinalização, deve cumprir requisitos de segurança a nível de desenho, arquitectura, especificação e implementação, obedecendo a regras definidas por entidades próprias.

1.3 Segurança dos Sistemas

Sistemas referidos na secção 1.2 são certificados por entidades competentes. As normas EN 50126, EN 50128 e EN 50129 regulamentam esses sistemas desde a sua fase de especificação, modelação e verificação até à sua implementação em software e testes no local definitivo. Estas normas derivam da norma de segurança IEC 61508 que é mais genérica e direccionada para sistemas críticos com *Programmable Logic Controllers (PLC's)*. O software do sector ferroviário rege-se, actualmente, pela norma EN 50128 *Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems* para providenciar uma aproximação racional e consistente no desenvolvimento de sistemas críticos. Esta norma é parte de um conjunto de normas orientadas para o sector onde se incluem, também, as normas EN 50126 *Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)* e EN 50129 *Railwaay applications - Safety related electronic systems for signalling* que pretendem, em conjunto, definir um ciclo de vida para desenvolvimento do software. O que elas propõem é uma "cultura" segura através da definição de um conjunto de técnicas e medidas para assegurar qualidade e segurança de tais sistemas. Veremos, com o devido detalhe, este ciclo de desenvolvimento no capítulo 2.6.2 deste documento.

1.4 Objectivo do Trabalho Proposto

Os serviços prestados pela EFACEC, no sector dos transportes ferroviários são essencialmente, implementação e manutenção de sistemas automáticos para Passagens de Níveis (PN). Estes sistemas, são concebidos em lógica de circuitos baseada em relés de contacto[1].

[1] Dispositivos electromecânicos para comutação de contactos eléctricos.

A EFACEC adquiriu uma basta experiência em certificação de hardware para os seus sistemas ferroviários segundo este conceito. Por diversos motivos a EFACEC pretende substituir os seus actuais sistemas baseados em relés, por sistemas electrónicos. No entanto, relativamente à processos de certificação de software a EFACEC já não possui o mesmo à vontade. Assim, o objectivo desta dissertação de mestrado é implementar um método para desenvolvimento de software para sistemas de sinalização. Pretende-se introduzir uma cultura formal e racional para o desenvolvimento de software com o objectivo final da certificação do processo.

A figura central desta implementação é a plataforma de desenvolvimento *SCADE Suite 6.0*. Esta, apresenta-se como sendo a mais adequada se tivermos em conta a sua interface com outras ferramentas e a sua amplitude de aplicação. Assim, uma das actividades a desenvolver, neste trabalho, é estudar a aplicabilidade da plataforma em sistemas de sinalização ferroviária. Pretende-se, ainda, obter uma experiência suficientemente abrangente e profunda para retirar conclusões da sua utilização. No final, consolidar-se-ão um conjunto de passos de forma a obter um processo contínuo de aplicação de ferramentas e metodologias, que garantam conformidade com as principais normas do sector e a certificação de todo o processo de desenvolvimento.

1.5 Contribuições

Os principais resultados desta dissertação dividem-se em três a surgem pela ordem em que se apresentam nas secções seguintes.

1.5.1 API para sistemas ferroviários

A *Application Program Interface (API)* para sistemas de sinalização surgiu como consequência da exploração, aprendizagem e adaptação à plataforma de desenvolvimento *SCADE Suite 6.0*. O objectivo, com este desenvolvimento, era adquirir experiência com a ferramenta especificando algo que, futuramente, se pudesse reutilizar. Pelo facto do objectivo ser a exploração da ferramenta, a referida API possui apenas alguns operadores que definem a interface dos objectos que compõem um sistema de sinalização ferroviária. Estes, permitem a manipulação básica dos mesmos, no entanto, considera-se já a existência de uma arquitectura modular que deverá apenas ser devidamente complementada e refinada.

1.5.2 Ciclo de desenvolvimento

Apesar das normas do sector obrigarem vários procedimentos na tentativa de garantirem uma forma segura de desenvolvimento, estas não impõem nenhum método ou ferramenta em particular. O que elas fazem são apenas recomendações, ficando ao critério da entidade que desenvolve qual o método e as ferramentas que vai utilizar. Este trabalho contribui exactamente neste ponto, na medida em que se define um método complementado com ferramentas que se deve aplicar no processo de desenvolvimento do software. O método obtido neste trabalho, pode ser considerado com um complemento aos métodos formais no desenvolvimento de software, visto que para além deste é ainda utilizado um conjunto de

outras ferramentas que completam o ciclo de desenvolvimento, nomeadamente no que toca à gestão de requisitos e execução de testes automáticos. Aqui, pode-se, desde já, enunciar os principais passos do método obtido:

1. Gestão de requisitos, de sistema e de software, utilizando a ferramenta *Dynamic Object Oriented Requirements System (DOORS)* (3.1);
2. Especificação, verificação formal e análise do modelo de software, que cumpre os requisitos do passo anterior, recorrendo à plataforma *SCADE Suite 6.0* (3.2);
3. Concepção do software utilizando a ferramenta *ELOP II Factory* (3.3);
4. Assegurar que o comportamento do software implementado é exactamente o mesmo que o comportamento do modelo especificado e provado no passo anterior. Para tal, foi utilizada uma ferramenta desenvolvida neste trabalho designada por *Simulator 1.0* (3.4).

De facto, este processo é uma interacção de ferramentas que utilizadas de forma rigorosa e com a devida sequência acompanham o software no seu ciclo de desenvolvimento. Segundo a clausula 15 da norma EN 50128 [27] o ciclo de desenvolvimento do software deve estar especificado no plano de garantia de qualidade do software (*Quality Assurance Plan*). Assim, o resultado deste trabalho não é mais do que o referido plano de qualidade.

1.5.3 Passagem de nível

Deste trabalho resulta, também, a aplicação do método proposto (consultar capítulo 4) no desenvolvimento formal de uma PN. A modelação de um sistema PN surgiu como forma de atravessar por completo a ferramenta *SCADE Suite 6.0*. Desta forma, foi possível aplicar o processo completo do método de desenvolvimento proposto, utilizando as ferramentas na sua plenitude. O resultado final é um software de controlo electrónico para PN's desenvolvido com uma metodologia que respeita a norma EN 50128.

1.6 Estrutura

Para além deste capítulo introdutório (1) que tem como objectivo contextualizar os problemas abordados neste trabalho, o presente documento é composto por mais 4 capítulos. O capítulo 2 tem por objectivo definir os problemas abordados e dar um conhecimento geral dos conceitos teóricos que ajudam a perceber o desenvolvimento do trabalho. No capítulo número 3 são apresentadas as ferramentas utilizadas tal como as principais características que fundamentaram a sua utilização neste projecto. O capítulo 5 refere-se à aplicação das ferramentas utilizadas no desenvolvimento deste projecto. Por último o capítulo 6 apresenta as conclusões retiradas deste trabalho.

Capítulo 2

Conceitos de Base

Neste capítulo pretende-se que o leitor perceba os conceitos teóricos que vão ser referidos no decorrer deste documento, e que foram a base de desenvolvimento do trabalho apresentado. Vão ser descritos, com a devida profundidade, os conceitos teóricos que fundamentam este trabalho e as ferramentas nele utilizadas. Vamos realçar a importância do suporte matemático providenciado pelos métodos formais no que respeita à especificação e verificação formal de sistemas críticos. No seguimento falar-se-à de sistemas reactivos. Estes, são normalmente sistemas com aspectos críticos e onde a segurança é o requisito principal. As linguagens de especificação de sistemas reactivos têm uma origem semântica na designada Hipótese Síncrona. Esta hipótese assume que um sistema tem um tempo de reacção instantâneo, ou seja, o tempo entre a detecção de um evento até à emissão da resposta é nulo. Veremos neste capítulo como é que as linguagens síncronas - baseadas na Hipótese Síncrona - absorvem este conceito e como permitem explorar esta propriedade nos sistemas reactivos. Vamos, mais detalhadamente, descrever um sistema de sinalização e os seus componentes, e posteriormente evidenciar a importância da aplicação dos métodos formais em tais sistemas. No final do capítulo são apresentados alguns exemplos de aplicação de métodos formais e perceber em que ponto se encontra a utilização de métodos formais no sector dos transportes ferroviários.

2.1 Métodos Formais

O crescimento rápido, em termos de quantidade e complexidade, de sistemas informáticos não foi, reconhecidamente no seio da engenharia de software, acompanhado por qualidade no seu desenvolvimento. O facto de a engenharia de software ser uma disciplina jovem deixou de ser um argumento para tal. Enquanto que anteriormente as exigências se centravam na existência de sistemas informáticos para gerir os mais diversos sectores da sociedade moderna, nos dias que correm estas centram-se na qualidade de tais sistemas. A qualidade, na engenharia de software, mede-se pela fiabilidade, robustez e segurança dos sistemas. Na tentativa de alcançar tais objectivos surgem os métodos formais que vamos detalhar nesta secção.

Antes de qualquer definição de Métodos Formais (MF) para o desenvolvimento de

Sistemas Informáticos (SI), é necessário perceber o contexto onde eles surgem, ou melhor, o contexto onde estes são necessários.

Nos dias que correm os SI são parte importante, ou mesmo fundamental, na gestão e organização das empresas. Mais do que isso, eles são responsáveis pela coordenação de máquinas e operações demasiado duras, ou mesmo impossíveis, para serem executadas através da força humana. Se imaginarmos domínios críticos, que envolvam vidas humanas ou que tenham impacto económico ou social, como por exemplo a medicina, a bolsa de valores ou o sector dos transportes aéreos e ferroviários, percebemos facilmente que uma eventual falha destes sistemas informáticos pode ser catastrófica. É preciso, portanto, garantir a correcção de tais sistemas ou, pelo menos, termos a maior garantia possível da sua correcção.

A obtenção de tais garantias não pode passar somente pela execução de uma fase de testes. O facto de um sistema passar com sucesso por tal fase não significa que este funcione de forma correcta. Porquê? O teste só pode confirmar que num ponto específico do domínio da aplicação esta funciona correctamente, não garante a correcção em todo o seu domínio. Ainda há mais, grande parte das falhas encontradas ocorrem normalmente quando um sistema enfrenta um situação imprevista, ou seja, dados ou situações do domínio que estão fora do conjunto representativo. Assim, a validação de software usando apenas a comparação do comportamento verificado com o comportamento esperado, deixa de ser suficiente. Então, qual a solução para obter um sistema informático isento de falhas? Não há. Podem sim, melhorar-se as técnicas e/ou métodos de desenvolvimento por forma a maximizar a isenção de tais falhas. Como? A obtenção de resultados consistentes e fiáveis só se obtêm utilizando caminhos consistentes e fiáveis. Aqui, o parente mais próximo da ciência da computação volta a reclamar um lugar: a matemática. Não surpreende. A sua capacidade de abstracção, tal como o seu rigor e a sua universalidade fez com que a ciência filha voltasse a aproveitar os conhecimentos da mãe para crescer como engenharia.

Progress will only be achieved in programming if we are willing to temporarily fully ignore the interconnection between our programs (in textual form) and their implementation as executable code... ...In short: for the effective understanding of programs, we must learn to abstract from the existence of computers.

Edsger Wybe Dijkstra[1]

A utilização dos conceitos matemáticos passa, assim, a permitir uma construção formal com aplicabilidade na engenharia de software para, numa primeira fase, descrever sistemas e depois, numa segunda fase, validar a descrição dos mesmos. A ideia é que através da aplicação de técnicas matemáticas seja possível uma descrição rigorosa, precisa e completa, eliminando inconsistências e ambiguidades, que de outra forma passariam despercebidas.

[1]Matemático e cientista na área da engenharia de software. Nas suas contribuições para a ciência da computação encontram-se o algoritmo para o problema do caminho mínimo (também conhecido como algoritmo de Dijkstra), e o conceito da auto-estabilização na área de sistemas distribuídos, uma forma alternativa de garantir a fiabilidade de um sistema. O cientista também foi conhecido pelos seus ensaios sobre programação, tendo sido o primeiro a alegar que programar é tão difícil e complexo que os programadores precisam realizar qualquer abstracção possível para gerir a sua complexidade com sucesso.

Podemos ver os MF como sendo o conjunto das linguagens matemáticas, técnicas e ferramentas de especificação e verificação que permitem uma abordagem diferente, alternativa ou complementar aos tradicionais testes e simulação na garantia de fiabilidade dos SI.

Convém, no entanto, ter em conta que a engenharia de software é uma actividade humana e que os MF não garantem a correcção ou a isenção de falhas, o que estes permitem é através de um processo de reflexão com recurso a ferramentas de apoio à especificação e verificação, modelar e validar um sistema com o maior rigor possível.

'Formal methods' are the use of mathematical techniques in the design and analysis of computer hardware and software; in particular, formal methods allow properties of a computer system to be predicted from a mathematical model of the system by a process akin to calculation.

[30], página 7

Num passado não muito longínquo a utilização prática de MF era apenas uma esperança. As notações eram demasiado obscuras e as ferramentas de suporte eram inadequadas ou de difícil utilização. Existiam apenas alguns casos de estudo, não triviais, que não eram suficientes para convencer a comunidade de engenheiros de software e hardware. Eram poucas as pessoas "treinadas" para utilizar, efectivamente, tais métodos no desenvolvimento à escala industrial.

Apenas recentemente se começou a ter uma imagem mais promissora dos MF. Para a especificação de software a indústria está disponível para experimentar notações para documentar as propriedades de um sistema de forma mais rigorosa. Para verificação de hardware a indústria adopta técnicas como a verificação de modelos e demonstradores de teoremas, para complementar a tradicional simulação. Em ambas as áreas, os respectivos investigadores providenciam cada vez mais casos de estudo industriais e com isso ganham os MF.

There are two aspects to using formal methods: the 'formal' aspect, and the 'method' aspect. The former indicates a commitment to using concepts and techniques derived from mathematical logic, the latter identifies the particular way in which those concepts and techniques are to be used.

[30], página 43

Mas, o que se pretende com a utilização de MF? O problema central é garantir que determinado sistema tenha determinado comportamento através de uma abordagem formal. No coração dos MF encontra-se a especificação que não é mais que a descrição do comportamento pretendido de determinado sistema. A partir deste momento, podemos subdividir o problema central em dois:

1. Como garantir, ao nível da especificação, o comportamento desejado?

2. Como obter, de uma especificação, uma implementação que conserva o comportamento especificado? Ou, de outra forma, como garantir que uma implementação goza do mesmo comportamento que a especificação?

Na tentativa de criar respostas às questões anteriores surgiram várias metodologias e, consequentemente, ferramentas que se apoiam, sobretudo, em duas fases de ataque ao problema. A primeira abordagem é a **validação** de um modelo, enquanto que, a segunda assenta na relação entre a **especificação** e **implementação**. O desenvolvimento formal de um SI não é, ainda, um processo de uma só fase, ou seja, não existe uma ferramenta que permita resolver de forma global e satisfatória, o problema central do MF. Assim sendo, temos então as ferramentas dividas por grupos de funcionalidades.

2.1.1 Especificar e Analisar

É aqui que surge o primeiro sub-problema do problema central: *"Como garantir, ao nível da especificação, o comportamento desejado?"*. Expressar através de uma formulação matemática (rigorosa e formal) o comportamento do sistema por modelar. Descrição formal do ***"quê"*** em oposição ao ***"como"***. O processo de especificação é o acto de descrever o sistema e as suas propriedades de forma precisa. Ao fazer isto o engenheiro de software aprofunda o conhecimento do sistema em especificação e elimina falhas estruturais, inconsistências e ambiguidades. A especificação é, assim, um canal de comunicação entre os vários intervenientes no desenvolvimento de um SI, tais como os engenheiro de software responsáveis pelo desenho do sistema, os programadores que o implementam, os responsáveis pela execução das simulações e dos testes e, até com os destinatários do sistema. A especificação serve também de documentação de acompanhamento à codificação do sistema, mas numa descrição de mais alto nível. No tipo das propriedades a descrever podemos incluir as funcionalidades pretendidas no sistema, a performance, os requisitos de tempo ou a estrutura interna.

2.1.2 Especificar e Demonstrar

O facto de se efectuar formalmente uma especificação do sistema, não significa que esta seja válida. A verificação formal é o processo pelo qual se verifica se a especificação efectuada implementa as propriedades pretendidas e se cumpre os requisitos do sistema, através do uso de MF. Animar ou executar deixa de ser suficiente, é necessário demonstrar formalmente que da implementação da especificação resulta, exactamente, o comportamento desejado.

Teste *vs* Prova A palavra que, de melhor forma, evidencia a diferença entre estes os conceitos de prova e teste em software é **completude**. De facto, até em sistemas não muito complexos, testar todos os cenários possíveis é uma tarefa impossível. Assim, o engenheiro de software prepara um conjunto de testes - uma pequena parte de todo o domínio - na tentativa de mostrar que para aquele conjunto de testes o sistema cumpre determinado requisito. Ao invés, a prova tenta - nem sempre é possível - provar, com recurso a formalismos matemáticos, que determinada propriedade é verificada pelo sistema em todo o seu domínio

de aplicação. Outra desvantagem dos testes é que normalmente são testados cenários "ideais", induzidos pelo engenheiro de software. No entanto, as falhas em software ocorrem em situações não esperadas e que na fase dos teste nunca são lembradas. Quando uma falha no sistema não é detectada em período de testes, a falha continua a acompanhar o resto do processo de desenvolvimento. Quando o software é concebido, a falha é introduzida no sistema e a detecção do erro pode ocorrer em plena execução, onde as consequências podem ser extremamente danosas. A correcção desta eventual falha exige um esforço muito maior quando comparado com a detecção ainda na fase de especificação. Assim, a verificação formal, para além de ser um acto mais profundo e completo que o teste, é ainda executado numa fase primária do ciclo de desenvolvimento, o que permite encontrar falhas num estado onde ainda é fácil a sua reparação. Não se pretende, de forma alguma, dizer que o teste não tem interesse em análise e validação de software, antes pelo contrário. Mesmo em sistemas onde são utilizados métodos formais na especificação e verificação, o teste é sempre a primeira abordagem para, de forma menos profunda, perceber se a especificação em desenvolvimento se direcciona para os objectivos pretendidos. Assim, a prova formal não surge em oposição ao teste: o teste é uma boa contribuição e é necessário para validar um sistema.

Numa demonstração formal, tudo o que precisamos saber acerca de um problema particular do sistema, está codificado na especificação deste e suas propriedades, pelo que a sua conformidade pode ser estabelecida simplesmente manipulando os "termos" de acordo com as regras de inferência. Assim, a prova determina a validade do sistema pois esta não depende do conhecimento do domínio do problema nem da intuição do que "diz" o sistema, pois a prova, através da verificação formal, é um exercício puramente mecânico que pode, em principio, ser efectuada por um programa de computador com total precisão. Não se pretende aqui dizer que o conhecimento e a intuição são supérfluos na utilização de métodos formais, pelo contrário, eles são essenciais na invenção de teoremas e axiomas úteis, na sua interpretação e aplicação no mundo real e na descoberta das suas provas.

Assim, existem, fundamentalmente dois grandes tipos de demonstração:

1. **Verificação Dedutiva** - A verificação dedutiva refere-se à utilização de axiomas e regras de prova para provar a correcção de um sistema. A importância da verificação por dedução é largamente reconhecida na ciência da computação. Influenciou significativamente a área do desenvolvimento de software com a noção de invariante[2]. A vantagem da verificação dedutiva é que abrange sistemas com um espaço de estados infinito. No entanto, este método formal para verificação de sistemas consome ainda, demasiado tempo no ciclo de vida de desenvolvimento e é uma tarefa executada apenas por *experts*. Dai que, a utilização prática desta técnica é raramente utilizada. Na tentativa de reduzir o tempo de verificação utilizando a dedução foram desenvolvidas várias ferramentas de apoio, mas a própria teoria da computação estabelece limitações no que se refere ao que pode ser decidido através de um algoritmo. Assim, uma

[2]De forma geral, um invariante é uma propriedade do sistema que se mantém sempre após a execução deste

verificação por dedução completamente automática só se pode verificar em sistemas pouco complexos.

2. **Verificação de Modelos** - A Verificação de Modelos, ou *Model Checking*, é uma técnica de verificação de sistemas concorrentes de estados finitos. Tem vantagens comparativamente às técnicas mais tradicionais baseadas na simulação, teste e verificações dedutivas. Em particular, é um processo automático e relativamente rápido. Outra vantagem é que em caso de erro produz um contra-exemplo muito preciso que comprova a vulnerabilidade do sistema especificado em determinado ponto. O grande desafio na verificação de modelos é a "explosão" do espaço de estados. Este problema acontece em sistemas com muitas componentes que interagem umas com as outras ou em sistemas com estruturas de dados que assumem muitos valores diferentes. Nestes casos, o número de estados possíveis pode, simplesmente, ser enorme e a sua verificação ser, demasiadamente, morosa, obtendo-se uma resposta razoavelmente rápida em máquinas dedicadas para o efeito. No entanto, nos últimos anos têm havido desenvolvimentos muito importantes na resolução do referido problema.

Pelo facto de a verificação de modelos ser um processo automático, é normalmente preferível comparativamente à verificação dedutiva. Contudo, haverá sempre domínios críticos em que a prova por teoremas é necessária para obter uma verificação completa. Existe nesta área uma nova abordagem [12] que tenta a integração da verificação dedutiva com a verificação de modelos, de forma a que a parte com estados finitos num sistema complexo seja verificada automaticamente.

2.1.3 Especificar e Derivar

O segundo sub-problema ganha agora a sua notabilidade: *"Como obter, de uma especificação, uma implementação que conserva o comportamento especificado? Ou, de outra forma, como garantir que uma implementação goza do mesmo comportamento que a especificação?"*. O que se pretende, agora, é obter uma implementação de uma especificação aprovada e, esta, pode ser obtida de três formas:

- A própria linguagem de especificação é uma linguagem de programação;
- Refinamento - Técnica de transformação por passos que permite derivar uma implementação, de forma gradual, a partir de uma especificação;
- Extracção - É uma técnica associada à utilização de sistemas de prova assistida. Recorrem à interpretação de Curry-Howard que afirma que a demonstração de uma propriedade P é calcular uma função f que testemunha a validade de P.

Parte do trabalho desenvolvido nesta tese relaciona-se com este aspecto. Obter de uma especificação validada a implementação que obedeça a todos os seus requisitos, foi um processo que exigiu um cuidado e esforço redobrado.

2.1.4 Especificar e Transformar

Existe, muitas vezes, a necessidade de mecanismos de transformação de modelos. Isto é, definir variações, simplificações, extensões, etc... para expressar provas complexas, modulares, e obter especificações suficientemente abstractas para provar determinada propriedade do sistema alvo.

2.2 Sistemas Reactivos

Sistemas reactivos definem-se como sendo sistemas informáticos que interagem continuamente com um dado ambiente e cujos tempos de resposta cumprem restrições temporais impostas pelo referido ambiente. Este tipo de sistemas possuem uma interface com o ambiente através de dispositivos de captura de eventos externos ao sistema. O período de tempo entre a captura desses eventos, o processamento dos mesmos e a emissão da resposta ao ambiente deve cumprir o requisito temporal definido pelo ambiente de modo a que esta seja útil. Um programa de um sistema reactivo é executado e terminado numa sequência contínua e, idealmente, infinita. O intervalo entre o início do programa e o término deste é dito um ciclo de execução.

Os sistemas reactivos distinguem-se dos sistemas de transformação, dos sistemas interactivos, e dos sistemas onde o computador determina o início de uma acção ou reacção. Entenda-se por sistemas de transformação os programas clássicos onde os inputs são disponibilizados antes da execução, e após a execução é devolvido o resultado (ex: compiladores). Os sistemas interactivos são aqueles que reagem a eventos mediante a sua disponibilidade e à sua própria velocidade (ex: sistemas operativos). Sistemas reactivos são, por natureza, sistemas deterministas, isto é, sistemas que reagem sempre da mesma forma num determinado estado e mediante determinada acção do ambiente. São maioritariamente aplicados em processos de controlo, monitorização e processamento de sinais, mas também em sistemas como protocolos de comunicação e em interfaces Homem-máquina quando os tempos de resposta exigidos são muito pequenos. Na sua generalidade estes sistemas partilham algumas características importantes:

Sistemas de tempo real - São sistemas que estão sujeitos à progressão, continua e natural, do tempo. No fundo são sistemas onde não existe a possibilidade de fazer retroceder um processo visto que este evolui paralelamente ao tempo. Assim, a única forma de tais sistemas funcionarem correctamente, é fazer certo no instante certo.

Sistemas com restrições temporais - São sistemas em que o meio que o envolve impõe restrições de tempo. Não confundir a restrição temporal com rapidez. Há sistemas em que o intervalo de resposta aceitável são milésimos de segundo enquanto que noutros sistemas pode ser de segundos. Uma restrição temporal, é um período de tempo em que o sistema deverá dar uma resposta de modo a que esta seja útil. Sistemas em que a validade do output não depende apenas do valor lógico da resposta mas também da restrição temporal associada. Esta restrição inclui a frequência dos inputs e o tempo de resposta entre estes e os respectivos outputs. Tal como já foi dito, estas restrições são impostas pelo ambiente

e devem ser cumpridas de forma imperativa, devendo ser especificadas, tidas em conta na modelação e verificadas como itens de elevada importância na correcção de tais sistema.

Sistemas síncronos - São sistemas em que as saídas do sistema são sincronizadas com as suas entradas. O comportamento de um sistema reactivo é uma sequência de reacções a eventos do ambiente, onde uma reacção implica captura dos inputs e processamento dos outputs e altera o estado interno do sistema. De forma menos formal, pode dizer-se que a execução de um sistema reactivo é processo iterativo em três fases, onde a primeira fase corresponde à aquisição dos inputs, a segunda ao processamento e a terceira à emissão dos outputs para o ambiente envolvente.

Sistemas embebidos - São sistemas com arquitecturas muito próprias e específicas em que o software pode depender do hardware e/ou o hardware depender do software.

Dependência - Sistemas reactivos são, na sua maioria, sistemas de alta criticidade e esta pode ser a sua propriedade mais importante. Uma falha na modelação de tais sistemas pode provocar catástrofes. Basta pensar num sistema de uma planta de controlo nuclear, e no sistema de navegação de um voo comercial! Neste domínio de aplicações a utilização de métodos formais é mandatária. Métodos e ferramentas com suporte formal devem ser aplicadas.

Determinismo - Esta é uma característica inerente a qualquer sistema reactivo. Indica que para uma mesma sequência de inputs o sistema reage sempre da mesma forma, ou seja, responde sempre com a mesma sequência de outputs.

Concorrência - Um sistema reactivo é composto por sub-sistemas que necessitam de cooperação e, consequentemente, comunicação de uma forma determinística. Há uma diferença comparativamente ao sistemas interactivos. A concorrência em sistemas interactivos, preocupa-se, principalmente, com a gestão de recursos que é resolvida de forma não determinística, enquanto que em sistemas reactivos trata-se de concorrência funcional.

Um sistema reactivo, na sua totalidade, é constituído por três camadas:

1. **Interface com o ambiente** através de dispositivos para captura e emissão de eventos no sistema;
2. **Núcleo Reactivo** responsável pelo processamento dos inputs obtendo os outputs;
3. **Manipulação de dados** camada de operação, controlo e manutenção.

É na segunda camada que estamos, particularmente, interessados neste projecto. As técnicas e ferramentas clássicas de especificação e verificação foram inicialmente dirigidas para sistemas de transformação e mais tarde para sistemas interactivos. Infelizmente, tais técnicas e ferramentas, não se adequam aos sistemas reactivos. Assim, surge a necessidade de desenvolver técnicas próprias apoiadas em ferramentas adequadas para o desenvolvimento de sistemas reactivos. No início dos anos 80 surgiu a abordagem síncrona. Esta

abordagem deu origem à Hipótese Síncrona, que se baseia numa semântica matemática, e gerou um conjunto de linguagens e compiladores que, nos dias de hoje, se aplicam em grande escala em sistemas reactivos críticos que vão desde os sectores da aviação, transportes ferroviários e energia nuclear, até ao desenho de circuitos eléctricos, protocolos de comunicação, ..., etc.

Normalmente, a segunda camada dos sistemas reactivos envolvem dois tipos de actividade:

1. **Transformação de dados (*data handling*)** - é a parte do sistema que processa os valores dos dispositivos que capturam eventos do ambiente, e gera outputs válidos usando, frequentemente, formulas matemáticas complexas. Os dados são, de forma regular, alvo dos mesmos processos de transformação.

2. **Operações de controlo (*control handling*)** - é a parte do sistema que altera o seu comportamento de acordo com eventos do ambiente originados por sensores discretos[3], inputs do utilizador ou eventos internos ao programa.

2.3 A Hipótese Síncrona

A Hipótese Síncrona é o principio fundamental das principais linguagens para sistemas críticos. Na prática, esta hipótese assume que o programa é capaz de reagir a eventos externos, antes que outros eventos ocorram ou, por outras palavras, o modelo de programação síncrona parte do princípio que o ambiente não evolui, não interfere, com o sistema durante os processos de reacções. Assim, a duração do processamento referente à reacção, se comparado aos tempos relacionados com o ambiente externo, é desprezível, e durante esse tempo o programa ignora qualquer alteração do ambiente.

Este modelo de programação é dito *input driven*, pelo facto de um programa reagir em instantes. Em cada instante o programa lê os inputs, calcula os outputs e envia-os para o ambiente (ver figura 2.1).

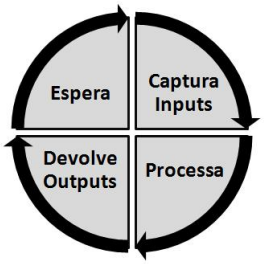


Figura 2.1: Execução cíclica do modelo síncrono

A Hipótese Síncrona separa, desta forma, o tempo físico do tempo de execução do sistema. Esta separação tem como principal consequência a simplificação de conceitos,

[3]Neste contexto, significa que são sensores que não emitem continuamente o estado do ambiente, emitem apenas mediante determinados eventos

separando a lógica de comportamento do sistema das características da sua implementação, que facilita a construção, compreensão e verificação deste.

Para perceber de que forma as linguagens síncronas respeitam esta hipótese, na secção seguinte são abordadas as linguagens Lustre e Esterel. Estas duas abordam a Hipótese Síncrona através de formalismos diferentes mas, a combinação dos dois deu origem à figura central deste trabalho, o *SCADE*.

2.4 Linguagens Síncronas

No conjunto das linguagens síncronas podemos encontrar as linguagens Esterel [6, 13, 18], Lustre [21,22], Signal [23], Argos [25], entre outras. Estas linguagens, pela sua origem na Hipótese Síncrona, providenciam conceitos que permitem aos engenheiros de software pensar nos seus programas assumindo que estes reagem instantaneamente aos eventos exteriores.

A compilação, das linguagens síncronas, é um processo não trivial pelo facto de a hipótese assumir um tempo de execução nulo. Esta propriedade pode provocar dependências cíclicas que têm de ser resolvidas, em tempo de compilação, para gerar código sequencial e *single-threaded*. O problema é conhecido por *Constructive Causality* e pode ser resolvido com métodos e técnicas que o leitor pode consultar em [14].

Como vimos na secção 2.2 na página 15, o núcleo reactivo de um sistema reactivo envolve dois tipos de actividade e, assim, as linguagens síncronas também divergiram nessas duas direcções:

1. Relação entre inputs e outputs de um sistema (*data-flow*), Lustre e Signal;
2. Relação entre eventos e reacções de um sistema (*control-flow*), Esterel e Argos

Vamos analisar, com mais detalhe, as linguagens Esterel e Lustre. Estas, são as raízes da linguagem da principal ferramenta deste trabalho, o *SCADE Suite 6.0*, e embora elas pertençam à mesma famílias das linguagens síncronas elas diferem no formalismo de especificação. O Esterel é uma linguagem direccionada para sistemas predominantemente de controlo (*control-flow*), enquanto que o Lustre é uma linguagem direccionada para sistemas de manipulação e transformação de dados (*data-flow*).

2.4.1 Esterel

A linguagem Esterel é considerada a primeira linguagem síncrona. Desenvolvida no início dos anos 80, por uma equipa de investigação e desenvolvimento liderada por Gérard Berry nos laboratórios INRIA - Sophia Antipolis e École des Mines de Paris [17], a linguagem Esterel surge para programar sistemas reactivos com predominância em operações de controlo [13]. Esterel é uma linguagem com semântica formal, textual e imperativa que permite tanto a programação de aplicações de hardware [33], compilando para VHDL[4] como aplicações de software para controladores síncronos, compilando para C.

[4] Acrónimo de VHSIC (**V**ery High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. Linguagem de desenho de sistemas digitais, desenvolvida pelo departamento de defesa americano em 1983.

Um programa Esterel é um sistema sincronizado com um relógio global. Num instante desse relógio cada processo sequencial é executado a partir do ponto de paragem do último instante. Os processos comunicam entre si e com o ambiente através de sinais e suspendem a sua execução até ao próximo instante do relógio global. Um sinal, num determinado instante, pode estar presente ou ausente, e o seu valor não persiste no instante imediato. Estes podem ser sinais puros ou sinais com valor. O valor de um sinal pode ser de qualquer tipo arbitrário. A comunicação entre tarefas concorrente é conseguida através de emissão de sinais - `emit X` - que são visíveis por todos os processos em execução através de um operador condicional que testa a presença de um sinal - `present B`. A sintaxe completa da linguagem pode ser consultada em [18].

Em seguida é apresentado um pequeno exemplo de um *module* em Esterel que utiliza manipulação sinais.

```
module Exemplo1:
        input A, B;
        output X, Y, Z;
        loop
                emit X;
                await A;
                emit Y
                present B then emit Z end
        end
end module
```

A presença de um sinal só é verificada, pelas tarefas concorrentes, no mesmo instante onde ele foi emitido, isto é, o valor de um sinal só é visível no mesmo ciclo de execução onde é emitido desaparecendo no ciclo imediatamente a seguir. Em Esterel uma instrução que testa o valor de um sinal bloqueia até ser verificada a presença ou ausência desse sinal. Por outras palavras, deve existir uma instrução que emite um sinal antes de qualquer instrução que verifica a existência ou ausência deste. Este é um problema de *Constructive Causality* que referimos em 2.4 na página 16.

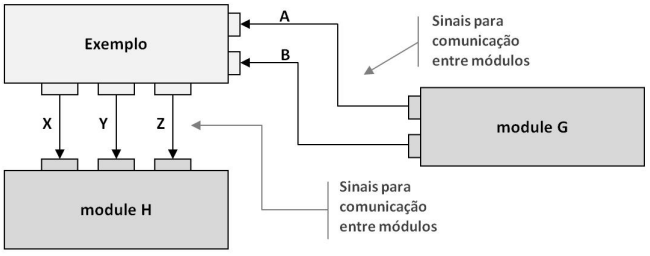


Figura 2.2: Comunicação de processos em Esterel

Os compiladores Esterel possuem já mecanismos que resolvem este problema, recorrendo a CCFG (*Concurrent Control-Flow Graph*) [7], onde o programa Esterel é traduzido

para um sistema de grafos concorrentes. Cada vértice desses grafos é definido com base em dependências de dados e operações de controlo. O código sequencial é então gerado percorrendo os grafos de vértice em vértice, onde a alternância entre os grafos é feita mediante a emissão e presença de sinais. Assim, um programa Esterel é facilmente traduzido numa *Mealy Finite State Machine (FSM)*[5].

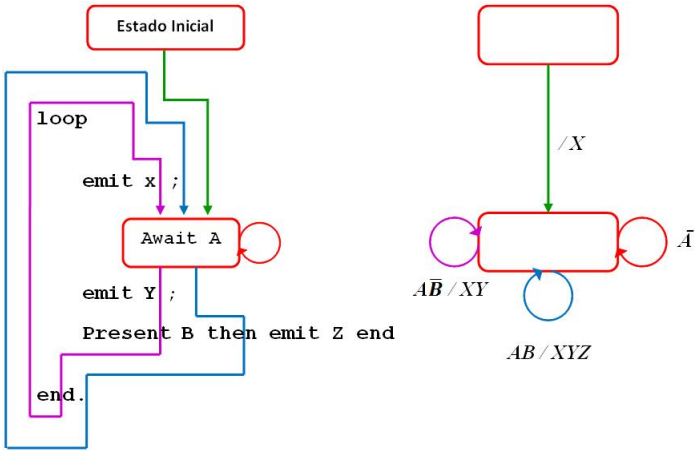


Figura 2.3: Representação com uma *Mealy FSM*

A semântica formal do Esterel permite, numa outra fase, a verificação formal de um sistema (ver secção 2.1.2). Os mais recentes motores de verificação utilizam a técnica de *Binary Decision Diagram* (BDD) [2] na redução da FSM, como por exemplo a ferramenta Xeve [8].

2.4.2 Lustre

Lustre [21, 22] é uma linguagem textual síncrona e declarativa, desenvolvida no início dos anos 80, com a intervenção de Paul Caspi, Nicolas Halbwachs nos laboratórios Verimag (Grenoble - França). Surgiu paralelamente à linguagem Esterel com o mesmo objectivo, o de facilitar a modelação de sistemas reactivos. É uma linguagem declarativa pelo facto da descrição ou especificação de um sistema ser um conjunto de equações que deverão ser sempre verificadas pelas variáveis do sistema. A modelação de sistemas é, tradicionalmente, feita com elevado nível de abstracção e com base em redes de operadores que transformam fluxos de dados. Essa transformação baseia-se em aplicação de funções booleanas, operadores lógicos e diagramas de blocos (*block-diagrams*). Este formalismo aproxima-se ao que os cientistas da computação chamam de sistemas *data-flow*. É precisamente no domínio dos sistemas reactivos predominantes em operações de transformação de dados que a linguagem Lustre actua. Um programa Lustre, e cada parte dele, assume um comportamento cíclico, onde cada ciclo é um instante de tempo em que o programa é executado na sua totalidade.

[5]Uma *Mealy FSM* é um modelo abstracto de uma máquina de estados finita em que o output depende do estado e do input

A noção central do Lustre é a de "relógio". Um relógio não é mais do que um fluxo *booleano* onde cada instante assume o valor *true* ou *false*, sendo que o valor *true* define um *tick* desse relógio. Cada programa em Lustre possui um relógio global onde todos os seus instantes estão a *true*. Assim, uma variável x é dada pelo fluxo $x_1, x_2, x_3, ..., x_n$. O valor x_1 indica o valor da variável x no instante 1, o valor x_2 é o valor de x no segundo instante,... e assim sucessivamente. A equação $(x+y)/2$ pode ser representada, utilizado o formalismo *data-flow*, da seguinte forma (Fig. 2.4):

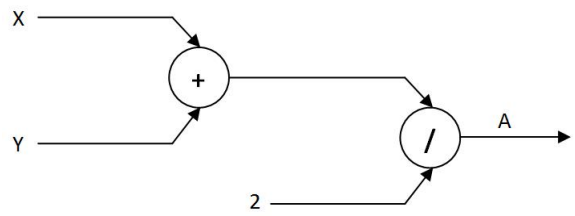


Figura 2.4: Formalismo *data-flow*

A mesma função, é deduzida em Lustre da seguinte forma:

$$\begin{aligned} x = &\quad x_0, x_1, x_2, x_3, ..., x_n \\ y = &\quad y_0, y_1, y_2, y_3, ..., y_n \\ x+y = &\quad x_0+y_0, x_1+y_1, x_2+y_2, x_3+y_3, ..., x_n+y_n \\ (x+y)/2 = &\quad (x_0+y_0)/2, (x_1+y_1)/2, (x_2+y_2)/2, (x_3+y_3)/2, ..., (x_n+y_n)/2 \end{aligned}$$

Ainda a mesma função é definida em Lustre pela seguinte função (em Lustre normalmente designa-se `node`):

```
node EXAMPLE1(x: int; y: int) returns(a: real);
let
        a = (x + y)/2;
tel
```

A linguagem Lustre é fortemente "tipada", no sentido de que cada fluxo de dados tem um tipo associado. Os tipos de dados elementares são: *boolean*, *integer*, *real*, e um tipo construtor: *tuple*. Contudo, tipos mais complexos podem ser importados da linguagem *host* e incorporados como tipos abstractos. A consistência dos tipos é verificada em tempo de compilação (*Type checker*).

Uma constante pode ser de um tipo de base da linguagem ou de um tipo importado. O fluxo que lhe corresponde é uma sequência em que o valor da constante se mantém inalterado, e o relógio que lhe está associado é o relógio global. Os operadores de base do Lustre são os que permitem manipular os seus tipos de base, tais como, operadores aritméticos: `+`, `-`, `*`, `/`, `div`, `mod`; operadores *booleanos*: `and`, `or`, `not`; operadores relacionais: `=`, `<`, `<=`, `>`, `>=` e um operador condicional: `if then else`. A linguagem permite também a importação de operadores definidos na linguagem *host* designados de *data operators*. Estes, só podem ser utilizados em operandos que partilhem o mesmo relógio. Para além dos

operadores já referidos, a linguagem posssui ainda quatro operadores designados de *"temporal" operators* e que actuam, especificamente, em fluxos:

1. `pre A` - devolve a sequência de A com um atraso de 1 instante. Se `A =1,2,3,4...`, então `pre A = nil,1,2,3,...`;

2. `A -> B` - inicializa uma sequência em que o primeiro valor é o primeiro instante do primeiro argumento e os restantes instantes os do segundo argumento. Se `A` é constante, `A = 5,5,5,5,5,...` e `B = nil,2,7,9,0,...` então `A -> B = 5,2,7,9,0...`;

3. `A when B` - permite definir uma sequência com base num relógio, ou ainda, pode ser visto como um operador de filtragem. Se `A = 5,2,8,4,2,0,5,...` e `B` uma sequência booleana `B = T,F,T,T,F,F,T, ...` então `A when B = 5,8,4,5,...`;

4. `current X` - permite interpolar uma expressão onde o seu relógio não é o global. Seja `X = A when B` do item anterior, ou seja, `X = 5,nil,8,4,nil,nil,5,....` `current X = 5,5,8,4,4,4,5,....`

Embora o Lustre seja declarativo, e um programa seja feito com base em equações, ela pode especificar propriedades. Estas generalizam as equações do sistema e consistem em expressões *booleanas* que deverão ser sempre satisfeitas. O primeiro objectivo desta especificação é dar ao compilador informação para optimização do código. De facto, existem propriedades do ambiente onde o sistema é executado, que nunca acontecem e que o compilador não precisa prever no seu código final. Por exemplo, imagine-se que num determinado sistema temos dois eventos, `X` e `Y` e, sabemos que, por restrições físicas, mecânicas ou mesmo de arquitectura do software, nunca acontecem em simultâneo. Esta propriedade do sistema pode ser expressa em Lustre, com o operador `assert`, da seguinte forma:

```
assert not (X and Y)
```

De forma similar, se o evento `X` nunca ocorre antes do evento `Y`, temos:

```
assert (true -> not(Y and pre(X)))
```

Para além da optimização de código, em momento de compilação, a especificação de propriedades assume um papel muito importante na verificação formal de um sistema, como veremos na secção 2.4.2 na página 23.

Sendo o Lustre uma linguagem declarativa - em lustre um programa é um sistema de equações -, esta sugere naturalmente, a noção de função: um sub-programa que é possível encapsular e reutilizar noutras partes do sistema. De seguida é apresentado um `node` em Lustre que define um contador genérico que tem como parâmetros de entrada o valor inicial e o valor de incremento, tal como um evento que define um *reset*:

```
node CONTADOR(ini, incr: int; reset: bool) returns(n: int);
let
  n = ini -> if reset then
               ini
               else
                 pre(n) + incr
tel
```

Tal operador pode ser instanciado de várias formas segundo o pretendido. Por exemplo, podemos utilizar o operador `CONTADOR` para obter uma sequência de número pares, ou mesmo uma sequência cíclica dos números entre 0 e 5:

```
      pares = CONTADOR (0, 2, false);
modulo5 = CONTADOR (0, 1, pre(modulo5)=4);
```

Esta propriedade modular da linguagem, assume um papel importante na arquitectura de um programa. Para além da vantagem óbvia que é aproveitar trabalho já feito, permite também o encapsulamento e atingir um nível de abstracção elevado.

Em termos de verificação do sistema global é também mais fácil fazê-lo por módulos. A utilização de módulos onde as propriedades mais elementares do sistema estão já provadas, permite a verificação de propriedades mais gerais assumindo as elementares. Em termos de verificação de modelos, *model checking*, esta é uma técnica bastante eficaz na tentativa de evitar a explosão de estados (ver secção 2.4.2, página 23).

Verificação Estática Uma verificação estática consistente é, claramente, uma questão de extrema importância quando se trata de plataformas para sistemas críticos. Para além da tradicional verificação de tipos, um compilador Lustre faz ainda as seguintes verificações:

Verificação de definições Cada output ou variável local deve ter apenas uma, e só uma, equação que a define. A seguinte definição não é permitida:

```
x = a + b;  x = c + d;
```

Verificação de consistência de relógios O problema na consistência de relógios está associado a todas as linguagens *data-flow* e é resolvida em Lustre com uma técnica conhecida no meio por *clock calculus*. No exemplo que a seguir se apresenta é fácil perceber que a equação `y = x + z` combina operandos com relógios diferentes.

```
x = CONTADOR(0,1,false)  x = 0, 1, 2, 3, 4, ...
b = true -> not pre b    b = t, f, t, f, t, ...
z = (x when b)           z = 0, ?, 2, ?, 4, ...
y = x + (x when b)       y = 0, ?, 4, ?, 8, ...
```

A verificação *clock calculus* consiste em associar um relógio a cada expressão e verificar que cada operador é aplicado com operandos com relógios apropriados:

- Um operador primitivo com mais do que um argumento é aplicado com operandos com o mesmo relógio;
- O relógio de qualquer operando do operador `current` não é o relógio básico do operador onde `current` é utilizado;
- O relógio dos operandos devem obedecer aos requisitos definidos na interface do operador.

Verificação de inicialização de expressões Este tipo de verificação detecta se no primeiro instante de uma variável, esta nunca assume o valor `nil`.

```
                b = 1,2,3,4,5,6,...
                y = 1,1,1,1,1,1,...
x = pre b       x = nil,1,2,3,4,5,...
z = x + y       z = ?,2,3,4,5,6,...
```

Neste exemplo, a solução que permitiria compilar com sucesso seria a utilização do operador `->` para inicializar a sequência de `x`, da seguinte forma:

```
                      b = 1,2,3,4,5,6,...
                      y = 1,1,1,1,1,1,...
x = 0 -> pre b        x = 0,1,2,3,4,5,...
z = x + y             z = 1,2,3,4,5,6,...
```

Verificação de definições cíclicas Esta é uma verificação que detecta um problema já referido na secção 2.4 na página 16. É um problema onde existe dependência cíclica num mesmo instante. Por exemplo as equações que se seguem dependem uma da outra. `X` depende de `Y` e `Y` depende de `X`.

```
X = if C then Y else Z;
Y = if C then Z else X;
```

A seguir a representação gráfica das equações:

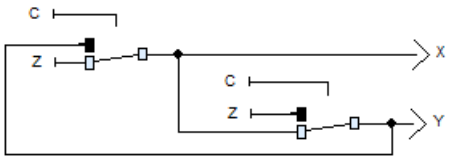


Figura 2.5: Problema de causalidade (Representação em Scade)

Lustre em máquina de estados Um programa em Lustre pode ser traduzido, ou representado, numa máquina de estados. Esta tradução baseia-se em técnicas de compilação da linguagem Esterel (ver secção 2.4.1 na página 16). A definição numa estrutura de controlo consiste em escolher um conjunto de variáveis *booleanas* que definem um estado, e cujos valores se espera que influenciem a execução em ciclos futuros. Para cada estado possível, é definido o código sequencial a ser executado durante um ciclo. Iniciando o programa num determinado estado e executando o correspondente código sequencial, é equivalente a determinar o estado seguinte. Este processo permite verificar, em tempo de compilação, que todos os estados e transições são alcançadas a partir do estado inicial. O resultado é uma FSM cujas transições são o código da correspondente reacção. Como veremos na secção 2.4.2 na página 23, esta transformação Lustre em autómato revela-se fundamental no que se refere à verificação formal.

Verificação formal Sistemas reactivos estão, normalmente, associados a sistemas críticos onde a segurança é requisito primordial. Assim, a verificação formal assume um papel importante no desenvolvimento destes sistemas. Já vimos em 2.1.2 na página 10 que existem, principalmente, dois tipos de verificação formal:

1. Verificação dedutiva, e
2. Verificação de modelos - *model checking*.

Em Lustre, pela transformação do programa numa árvore de estados, é possível efectuar verificação formal utilizando técnicas de *model checking*. Duas das principais vantagens, deste tipo de técnicas, é a automatização do processo e a apresentação de contra-exemplos em caso de falha na tentativa de provar determinada propriedade. A verificação de modelos em Lustre consiste em:

1. Construir o grafo de estados do programa (assume-se, obviamente, um número de estados finito). Aqui o problema principal é o número de estados atingido, que pode ser minimizado da seguinte forma:

   a) A propriedade declarativa da linguagem e a sua possibilidade de exprimir propriedades é um contributo na resolução desse problema. O grafo obtido no momento de compilação é uma abstracção do programa, visto que exprime apenas o controlo e ignora detalhes como variáveis não *booleanas* e variáveis *booleanas* que não influenciam o controlo do programa.

   b) Uma técnica importante na redução do grafo de estados consiste na utilização de *observers* e *property nodes*. Na tentativa de provar que determinada propriedade B é invariante no programa P, constroi-se um novo programa P' - *observer* -, em que o seu corpo é composto por P e um sistema de equações - *property node* - que definem B, cujo único output é um *booleano* que indica a validade de B. A partir do momento em que o compilador tem indicação de calcular apenas o valor de B, este terá em conta apenas a parte do programa que se relaciona com B, obtendo-se assim um grafo menor. Depois de obtido esse grafo, a verificação da propriedade B é verificar que em nenhum dos estados desse grafo o valor de B é falso.

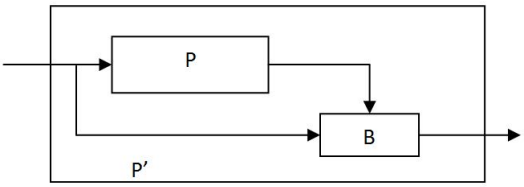


Figura 2.6: Esquema de um *observer*

c) Outra forma de reduzir o grafo, é exprimir propriedades do sistemas através de *assertions* - operador `assert` em Lustre. De facto, sabendo que a combinação de dois eventos `X` e `Y` nunca acontecem em simultâneo no sistema, esta propriedade pode ser expressa em Lustre

```
assert not (X and Y)
```

de forma a garantir que o compilador não vai gerir estados em que esta combinação `X and Y` se verifique. Esta técnica deve ser utilizada ainda quando se trata de valores numéricos. Escrever no sistema

```
A < B
```

evita que o compilador tenha que gerar os estados em que `A >= B`.

d) Outra forma de reduzir o tamanho do grafo de estados é recorrer a verificação modular. Para provar uma propriedade `B` num programa `P` que utiliza o operador `Q`, a ideia é provar `Q` e só depois provar `P` assumindo que `Q` é sempre verdadeiro - `assert Q`.

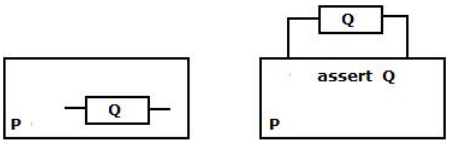


Figura 2.7: Esquema de verificação modular

2. Cada propriedade é verificada no grafo construído:

   a) Na verificação modular do sistema. Esta particularidade depende exclusivamente do programador. Este, poderá combinar propriedades provadas de sub-módulos para derivar propriedades de um módulo. Esta técnica permite reduzir a complexidade da prova graças à decomposição modular do sistema.

   b) Na verificação de que todos os estado possíveis satisfazem a propriedade que se pretende provar, ignorando a relação de transição (é utilizada apenas para construir os estados possíveis);

c) Na verificação de propriedades numa abstracção do programa. Se uma propriedade P se verifica para um programa L, então P verifica-se em todos os programas cujo seu comportamento é um subconjunto do comportamento de P;

Testes automáticos A verificação formal de um sistema reactivo nem sempre atinge o seu objectivo. Isso acontece muitas vezes quando o sistema é muito complexo, e a verificação atinge um número de estados demasiado elevado. Nessas situações a última solução é recorrer ao teste. O problema do teste comparativamente à verificação formal é conhecido (ver secção 2.1.2 na página 10). Importa automatizar o processo para que este se torne menos "viciado", mais rápido e menos dispendioso. Neste contexto existem ferramentas próprias, nomeadamente a ferramenta Lutess [9, 28], Lurrete [1, 20, 29] e GaTel [26]. Uma propriedade importante de qualquer sistema reactivo é que este é desenvolvido com base nos eventos do ambiente. Assim, o teste consiste na observação das reacções do sistema aos eventos do ambiente. Em sistemas complexos, o conjunto de todos os inputs válidos - o conjunto dos inputs que reflectem comportamentos reais do ambiente - pode ser enorme, e dai a necessidade desse conjunto ser gerado automaticamente. Percebe-se, facilmente, que o ambiente restringe os dados de teste que devem ser gerados. Essas restrições devem estar devidamente especificadas no modelo do sistema, de modo a reflectir a restrição na geração dos dados de teste. Outra particularidade, é que muitas vezes a reacção de um sistema não depende apenas dos inputs daquele instante, mas também, de execuções de instantes passados. Assim, para mais facilmente ter em conta o comportamento não determinístico do ambiente, o teste deve ser baseado numa sequência de dados gerados dinamicamente.

O método de testes utilizados pelas ferramentas de Lurette, Lutess e GaTel assentam em 3 passos:

1. **Selecção** dos dados de input os inputs para teste são fluxos de dados de comprimento igual, com valores compatíveis com os tipos definidos no sistema, e gerados automaticamente de acordo com as restrições especificadas. Ao mesmo tempo são obtidos os outputs esperados;

2. **Submissão** dos testes o programa em teste é executado mediante os inputs seleccionados e são obtidos os respectivos outputs;

3. ***Oracle*** a decisão de sucesso ou falha, mediante a comparação dos outputs obtidos durante a execução do programa com os outputs esperados.

As diferenças entre estas ferramentas baseiam-se, principalmente na forma com são obtidos os dados para input. Por exemplo, o Lutess é bastante rápido na geração dos inputs, no entanto, apenas consegue gerar inputs para sistemas *booleanos*. GaTel permite a especificação do objectivo do teste, mas os dados são gerados sempre da mesma forma, enquanto que no Lutess podemos obter os dados com base numa distribuição uniforme, em probabilidade de ocorrência, forçar a falha de determinada propriedade e com base em sequências de eventos. Lurette permite que os dados, *booleanos* e numéricos, sejam gerados com base numa especificação Lustre, Scade ou C. No entanto, obriga ao utilizador à implementação do *Oracle*.

2.5 Sistemas de Sinalização

Um sistema de sinalização ferroviária é, como o próprio nome indica, um sistema informático responsável por controlar e gerir a sinalização de trânsito ferroviário. Tratam-se de sistemas complexos que permitem estabelecer rotas para os veículos que circulam nas ferrovias, controlando electronicamente os dispositivos de linha que comandam os veículos que nela circulam.

Os sistemas electrónicos pretendem substituir os sistemas mecânicos mais antigos, argumentando as seguintes vantagens:

- As operações electrónicas não estão sujeitas ao erro inerente às operações humanas;
- A electrónica permite um auto diagnóstico que permite providenciar medidas para garantir a fiabilidade e a disponibilidade do sistema;
- A possibilidade de integrar e centralizar vários sistemas.

Um sistema ferroviário é composto por sub-sistemas mais elementares que comunicam entre si. A figura 2.8 apresenta um exemplo simples de uma linha ferroviária decomposta nos seus componentes mais elementares. Na imagem podemos encontrar 3 tipos de componentes:

1. **Sinais luminosos:** são dispositivos luminosos que apresentam dois aspectos. Um, indica a permissão de avanço do veículo. O outro, impede o seu avanço;
2. **Secções de via:** são troços de via delimitados por dois dispositivos luminosos;
3. **Agulhas:** são dispositivos que permitem mudar um veículo de direcção.

Tipicamente, um sistema de sinalização recebe o pedido de uma rota; a resposta ao pedido terá de passar por várias verificações que deverão cumprir determinadas regras:

- Uma rota não pode ser estabelecida se uma das secções de via que a compõem está ocupada;
- Uma rota está disponível se nenhuma das suas secções está já bloqueada para outra rota;
- Se a rota está disponível, então será estabelecida;
- Estabelecer uma rota implica que todos os elementos da rota sejam bloqueados (secções de via, agulhas, sinais);
- Quando uma rota é estabelecida os sinais que acompanham a rota podem assumir o sinal de aberto (cor que indica que o avanço é permitido), e só depois o pedido de rota inicial é confirmado e reconhecido no sistema;

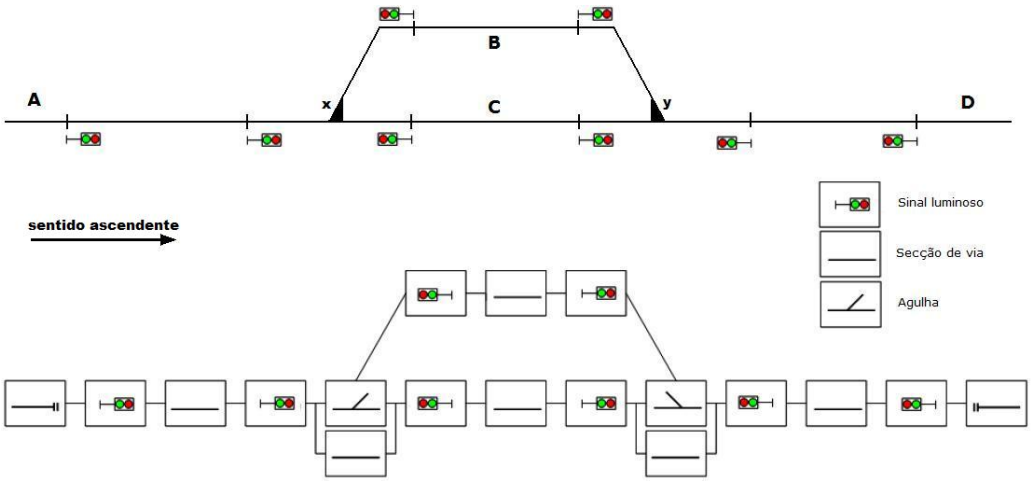


Figura 2.8: Decomposição de um sistema de sinalização

O objectivo destas verificações e acções tomadas pelo sistema, é garantir que nenhum veículo se possa mover numa rota ocupada por outro veículo. Assim, a actividade principal de um sistema de sinalização, que é um sistema embebido, é garantir a operação segura dos dispositivos físicos que constituem a linha ferroviária. Tal sistema controla um conjunto de equipamentos, de tal forma interligados entre si, que as suas funções devem ser executadas numa sequência apropriada, sequências para as quais o sistema tem regras para garantir a segurança das operações.

Podemos decompor um sistema de sinalização ferroviária em três tarefas distintas:

1. Na primeira, providenciam-se instruções para um determinada parte/secção da via. Esta tarefa é feita, pelo que de agora em diante se designa, por camada lógica. Esta camada pode ser totalmente automática, mas normalmente consiste num sistema automatizado que é controlado por peritos.

2. Na segunda tarefa faz-se com que as instruções definidas na primeira tarefa sejam executadas pela infraestrutura férrea (camada física). Esta é constituída por agulhas, sinais luminosos e sonoros, passagens de nível, etc... sendo, normalmente, uma tarefa completamente automática.

3. Por último, a terceira tarefa terá de garantir que a execução da segunda tarefa, não coloca em risco a segurança pretendida. Ou seja, garantir que não ocorrem colisões e/ou descarrilamentos. Tais garantias são conseguidas através de um procedimento a que se chama de **encravamento** e que se situa entre a camada lógica e a camada física do sistema;

Parte das verificações de segurança são efectuadas na camada lógica, mas a responsabilidade última reside no encravamento. A lógica do encravamento é a "personificação" de um conjunto de regras básicas e de princípios de segurança, de acordo com o movimento dos

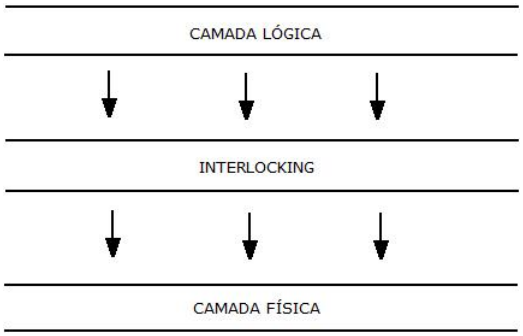


Figura 2.9: Camadas de um sistema de sinalização

veículos, num determinado sector da linha ferroviária. O encravamento restringe a colecção dos estados para uma secção de via. A camada lógica determina a ordem em que essas situações ocorrem, baseada em considerações de eficiência e conveniência de interligações entre os destinos possíveis de uma linha.

2.6 Métodos Formais em Sistemas de Sinalização

Vimos, de forma geral, nas secções anteriores, o que são os métodos formais e o que são sistemas de sinalização ferroviária. A utilização dos primeiros no âmbito dos segundos é um passo exigido pela própria construção de tais sistemas. Pela natureza e ambiente de execução desses, é requerido o mais alto nível de segurança e fiabilidade. Uma falha de tais sistemas colocará em risco vidas humanas, e elevados prejuízos financeiros, materiais, operacionais, culturais,... , etc. Assim, é preciso garantir que o desenvolvimento, de sistemas de sinalização ferroviária, obedeça a metodologias e técnicas rigorosas que permitam garantir o correcto funcionamento do sistema. Naturalmente, os métodos formais pela sua fundamentação matemática, que permite uma verificação com base em modelos abstractos ganha, por mérito próprio, um lugar destacado. A principal norma do sector dos transportes ferroviários, nomeadamente a EN 50128, define um processo de desenvolvimento rigoroso, que não obrigando a utilização de métodos formais, estes são vivamente recomendados.

2.6.1 Nível de Integridade de Segurança

O Nível de Integridade de Segurança ou SIL é uma medida de segurança que determinado sistema oferece. Há muito que a propriedade “Segurança” dos sistemas deixou ser catalogada como “seguro” ou “não seguro”. Esta medida está dividida em quatro níveis de segurança onde cada nível representa um factor de redução de risco. Quanto mais alto o nível, maior será o impacto de uma falha e, consequentemente, menor a taxa de falhas que é aceitável para o sistema em classificação.

A atribuição de um nível SIL, a um determinado item, é baseada na consequência de uma falha deste e na frequência com que ele é executado. Nos sistemas com baixa frequência de execução, a taxa de acidente é a combinação da frequência de execução com a sua

SIL	Disponibilidade	Redução de risco	Consequência qualitativa
4	> 99,99%	100000 até 10000	Ocorrência de fatalidades na comunidade
3	99,9%	10000 até 1000	Múltiplas fatalidades
2	99-99,9%	1000 até 100	Danos graves ou uma fatalidade
1	90-99%	100 até 10	Danos menores

Tabela 2.1: Tabel dos níveis SIL

probabilidade de falha, enquanto que, nos sistemas que operam continuamente com o seu meio, a taxa de acidente é a mesma que a taxa de falha. O SIL atribuído é o nível com uma taxa de redução de risco que transforma o sistema aceitável em termos de segurança.

2.6.2 O Normativo CENELEC EN 50128

A EN 50128 *Railway Applications: Software for railway control and protection systems* é uma norma europeia, da responsabilidade da *European Committee for Electrotechnical Standardization* (CENELEC), direccionada para o desenvolvimento de software no sector dos transportes ferroviários. É parte de um conjunto de normas, também elas direccionadas para o mesmo sector, nomeadamente as normas EN 50126 - *Railway Applications: The specification and demonstration of RAMS* e EN 50129 - *Railway Applications: Safety related electronic systems for signalling*. No que respeita a sistemas de sinalização, telecomunicações e processamento de dados no domínio ferroviário, o conjunto das normas EN 50126/50128/50129 veio substituir a norma europeia EN 61508-1 que é resultado de critérios genéricos direccionados para software de sistemas de segurança.

A norma EN 50126 define um processo sistemático para especificar e gerir, baseado no ciclo de vida do sistema e suas tarefas, o RAMS em termos de fiabilidade, disponibilidade, manutenibilidade e segurança. Por sua vez, a norma EN 50129 define os requisitos do hardware relacionados com a segurança. A norma EN 50128 concentra-se nos métodos que deverão ser utilizados para garantir software que cumpre os requisitos de segurança, onde o conceito base é a noção de SIL: quanto mais danosas as consequências da falha do software, mais alto o SIL será.

No âmbito deste projecto é com a norma EN 50128 que vamos trabalhar e por isso vamos analisá-la mais pormenorizadamente. A norma não pretende garantir a segurança absoluta do sistema, uma vez que não existe forma de provar a ausência de falhas em software. Ela pretende introduzir uma cultura segura através da definição de um conjunto de técnicas e medidas que se regem pelos seguintes princípios:

- Métodos de concepção descendente;
- Modularidade;
- Verificação de cada fase do ciclo de vida do desenvolvimento;
- Módulos verificados e bibliotecas de módulos;

- Documentação clara;
- Documentação auditável;
- Teste de validação.

Embora a EN 50128 não obrigue a utilização de métodos formais, ela recomenda-os, nomeadamente, na especificação dos requisitos do sistema. Na versão portuguesa da referida norma, podemos encontrar o quadro A.2, secção 8, que apresentamos na tabela 2.2, e onde a letra R significa que *"... a técnica ou medida é recomendada (Recommended) para o nível de integridade de segurança em questão."*. A sigla HR significa que *"... a técnica ou medida é Altamente Recomendada (Highly Recommended) para o nível de integridade de segurança em questão. Se esta técnica ou medida não for utilizada, o raciocínio inerente à sua não utilização deve ser detalhado no Plano de Garantia de Qualidade do Software ou noutro documento referenciado pelo Plano de Garantia de Qualidade do Software."* [27].

Técnica/Medida	**SIL 1**	**SIL 2**	**SIL 3**	**SIL 4**
1. Métodos Formais Incluindo, por exemplo, CCS, CSP, HOL, LOTOS, OBJ, VDM, Lógica temporal, Z e B	R	R	HR	HR
2. Métodos Semi-formais	R	R	HR	HR
3. Metodologia Estruturada Incluindo, por exemplo, JSD, MASCOT, SADT, SDL, SSADM e Yourdon	HR	HR	HR	HR
Requisitos: 1. A especificação de Requisitos do Software requerer sempre uma descrição do problema em linguagem natural e de todas as notações matemáticas necessárias à caracterização da aplicação. 2. O quadro reflecte requisitos adicionais que permitem definir uma especificação com clareza e precisão. Uma ou várias destas técnicas devem ser seleccionadas para satisfazer o Nível de Integridade de Segurança do Software utilizado.				

Tabela 2.2: Especificação de Requisitos segundo a norma EN 50128

A EN 50128 define um ciclo de desenvolvimento do software representado na figura 2.10 por um modelo em V. Cada fase deste ciclo de desenvolvimento está dividida em actividades elementares, bem delimitadas, e com inputs e outputs que definem a interface com as fases anterior e posterior respectivamente. A norma EN 50128 define, ainda, as técnicas e medidas apropriadas para cada uma das fases que compõem o ciclo, bem como a documentação que deve ser produzida no desenvolvimento de cada uma delas. Assim, as interfaces atrás referidas são, normalmente, efectuadas através de matrizes de rastreabilidade, onde são explícitas as ligações através de referências documentais. Cada uma das fase do ciclo é dependente da fase anterior, pelo que, em cada uma delas, deve existir uma actividade de verificação, que garanta que o resultado da fase em questão preenche os requisitos da fase que lhe antecede.

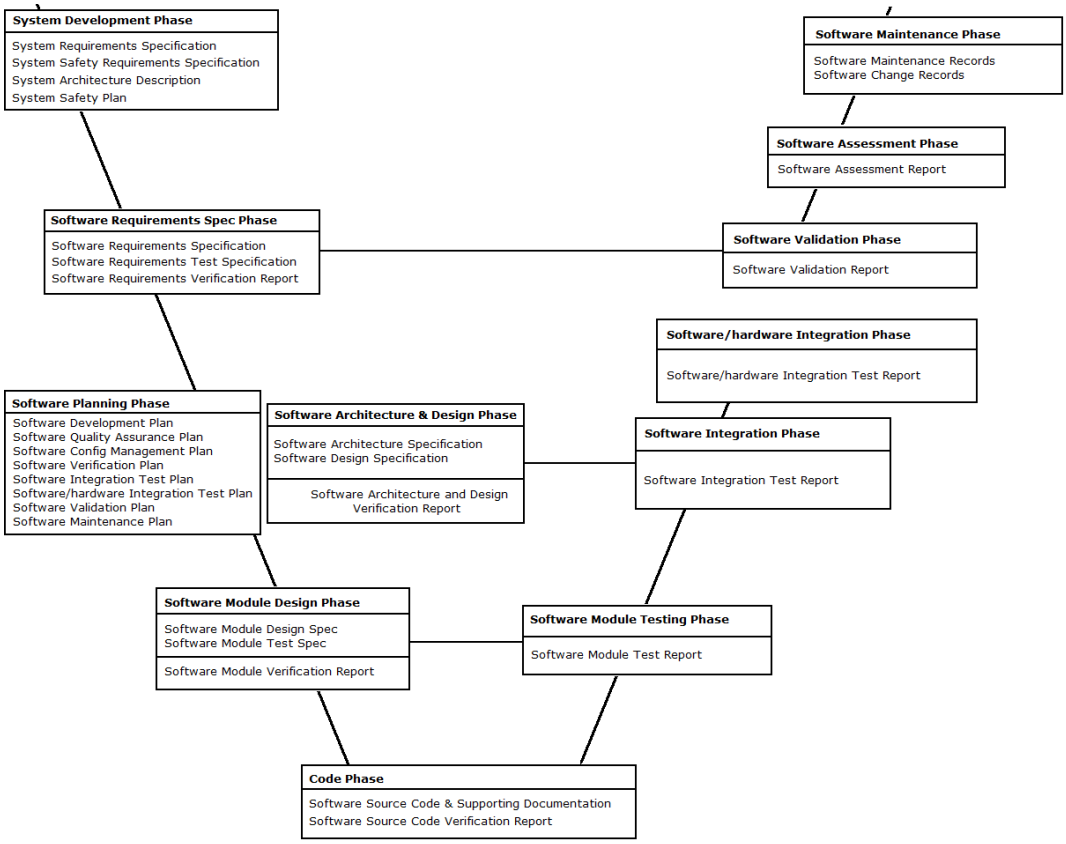


Figura 2.10: Ciclo de desenvolvimento do software definido pela norma EN 50128

2.7 Estado da Arte

A utilização de métodos formais no desenvolvimento das diversas partes que compõem um sistema de sinalização ferroviária tem vindo a crescer nos últimos tempos. Como consequência da implementação da norma EN 50128 muitas industrias do sector optaram pela utilização de métodos formais. Nomeadamente no desenvolvimento do software para tais sistemas foi possível descobrir alguns casos de aplicação.

O método formal B, como o próprio nome indica, é um método de desenvolvimento onde a figura central é a linguagem formal B. Encontram-se vários casos de aplicação industrial, no sector do transporte ferroviário, com este método formal. Sabe-se, por exemplo, que o método formal B foi utilizada, pela empresa Alstom, para especificação, desenho e verificação formal do sistema de controlo de velocidade KVB (*Contrôle de Vitesse par Balises*), aplicado pela companhia pública francesa SNCF (*Société Nationale des Chemins de fer Français*), instalando o sistema em mais de 6000 veículos desde 2003. O sistema de controlo automático de veículos ferroviários SAET METEOR, da Siemens Transportation Systems, foi também desenvolvido utilizando o método formal B [15, 16, 24].

Existem experiências académicas em especificação formal de um sistema de sinalização ferroviária utilizando a linguagem gráfica Statecharts, aplicando uma abordagem geográfica[6] [4, 5].

Embora a linguagem *Specification and Description Language* (SDL) seja maioritariamente aplicada na especificação de sistemas de telecomunicações, existem também experiências académicas na utilização desta em sistemas de sinalização ferroviária [3].

Existe uma aplicação da ferramenta *SCADE Suite 6.0* na implementação de um sistema de sinalização ferroviária durante um curso nos laboratórios Real-Time and Embedded Systems Group, do departamento de ciências da computação da Christian-Albrechts University of Kiel, alemanha. [32].

[6]Nesta abordagem o modelo é composto por entidades que correspondem aos componentes físicos da ferrovia (agulhas, sinais, etc...), e estes implementam colectivamente, as regras de sinalização. Em oposição, existe uma abordagem funcional que recorre a uma base de dados central com as regras da sinalização.

Capítulo 3

Ferramentas Usadas

Neste capítulo são apresentadas as ferramentas utilizadas na execução da solução obtida com este trabalho. Descreve-se com o devido detalhe cada uma delas, não devendo, em nenhum caso, ser utilizado como manual de utilização das respectivas.

O *DOORS* é uma ferramenta criada pela Telelogic, sendo, no entanto, parte do portefólio de software da IBM Rational, resultado de uma aquisição por parte da IBM da Telelogic. *DOORS* é uma ferramenta de suporte ao desenvolvimento e gestão de requisitos. Permite a introdução, análise e gestão de todo o processo documental que acompanha um projecto durante o seu ciclo de vida. Existe uma interface bidireccional entre o *SCADE* e o *DOORS* estando, desta forma, facilitado o processo de rastreabilidade no projecto. Foi utilizado para gerir os documentos de requisitos funcionais do sistema e requisitos do software.

O *SCADE Suite 6.0* é o principal produto da empresa Esterel Technologies, fundada em 1999 e com sede em Elancourt, França. *SCADE* é uma plataforma de desenvolvimento de aplicações seguras e permite criar, graficamente, especificações em linguagem Lustre. A base matemática que fundamenta a linguagem Lustre adicionada ao formalismo de máquinas de estados da linguagem Esterel, aliando um poder gráfico, faz com o *SCADE* se torne numa ferramenta extremamente poderosa, permitindo-a incluir num processo de desenvolvimento formal bastante intuitivo e de fácil leitura e interpretação. Foi utilizado para modelar, especificar, simular, analisar e verificar formalmente o software desenvolvido neste trabalho.

O *ELOP II Factory* é um produto da empresa HIMA, fundada em 1908 e com sede na Alemanha. É uma ferramenta desenvolvida para programar, parametrizar e configurar, graficamente, os controladores da mesma empresa, que são os utilizados nos projectos EFACEC. Foi utilizado na concepção do software especificado, simulado e verificado no *SCADE Suite 6.0*.

O *Simulator 1.0* é uma ferramenta desenvolvida neste trabalho e o seu objectivo é automatizar o processo de testes ao software. Foi desenvolvido usando o Delphi 2005. O Delphi 2005 é um produto da empresa americana Borland, fundada em 1983 e com sede em Austin no estado do Texas, Estados Unidos. O Delphi 2005 é uma plataforma de desenvolvimento integrado de aplicações para a plataforma Win32 e .Net. Integra várias linguagens de programação, tais como, Delphi, Delphi para a plataforma *.NET* e C#, sendo que a utilizada para desenvolver o *Simulator 1.0* foi o Delphi para a plataforma Win32.

3.1 DOORS

O *DOORS* é uma ferramenta de suporte ao desenvolvimento e gestão de requisitos durante o ciclo de vida de um projecto. Foi utilizada neste trabalho porque permite cumprir requisitos da norma EN 50128 relativamente ao ciclo de desenvolvimento de software nela especificado, no que se refere à relação entre cada fase do ciclo de desenvolvimento. A ferramenta permite a estruturação, o registo e a gestão de todos os requisitos do sistema tal como todos os documentos associados ao desenvolvimento. O que torna esta ferramenta imprescindível é o facto de permitir a interligação entre os documentos, e partes deles, uns com os outros, facilitando desta forma a rastreabilidade não só entre documentos, mas também entre itens desses documentos. Por exemplo, se forem introduzidos o documento que especifica os requisitos funcionais do sistema e o documento que especifica os requisitos de software, é possível indicar quais os requisitos de software que cobrem que requisitos funcionais. Da mesma forma, depois de introduzido o documento de especificação do software, é possível dizer que especificações resolvem os requisitos de software do documento anterior, e assim sucessivamente. No final, é possível estabelecer uma matriz de rastreabilidade que permite uma análise bidireccional a partir de qualquer ponto do ciclo de desenvolvimento. A flexibilidade da ferramenta é também um argumento válido. Esta ferramenta é suficientemente genérica para se aplicar a qualquer tipo de projecto. Permite ao utilizador que é administrador da ferramenta, configurá-la de modo a incorporar toda a informação necessária através da inclusão de campos específicos e relatórios personalizados. Pode ser vista como uma base onde se registam todas os dados e evoluções do projecto, permitindo, a qualquer momento e em qualquer fase do projecto uma consulta através de relatórios de base ou relatórios criados pelo utilizador. Outra particularidade é a sua capacidade de manter registos de alterações efectuadas, tais como data, hora, utilizador e, ainda, o conteúdo antes e depois da alteração.

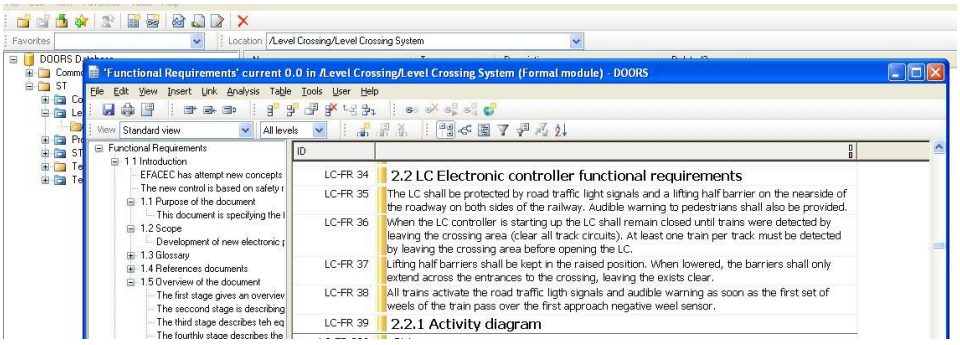


Figura 3.1: Ambiente DOORS: exemplo de um documento de requisitos funcionais

O *DOORS* possui, também, uma *gateway* para *SCADE* que permite efectuar ligações entre construções Scade e requisitos introduzidos no *DOORS*. Desta forma está facilitada a rastreabilidade entre determinado requisito no *DOORS* e o operador em Scade que satisfaz esse requisito, bastando para tal um *click* no requisito para que o *DOORS* automaticamente se posicione na construção Scade que lhe está associada. Esta facilidade poderá estar dis-

ponível, mediante a aquisição do módulo para o *SCADE* que efectua a *gateway* com o *DOORS*. Existindo essa *gateway* pode verificar-se o processo inverso, ou seja, a partir de um operador Scade obter, no *DOORS*, o requisito que o justifica. A *gateway* no *SCADE* permite que este crie, de forma automática, no *DOORS*, todo um documento relativamente à especificação e verificação efectuada no *SCADE*, evitando, assim, a introdução manual destes documentos. A ferramenta já estava a ser utilizada na EFACEC antes do início deste trabalho e, vai de encontro às necessidades deste trabalho, sendo assim utilizada para gerir toda a documentação relativa ao projecto.

DOORS na Norma EN 50128 Na norma EN 50128 o *DOORS* ocupa um lugar central. É com base na gestão de todo o processo de documentação que se baseia o ciclo de desenvolvimento em V. Cada fase do ciclo deverá ter origem em documentos da fase anterior e, por sua vez, produzir documentação para as fases seguintes. Assim, uma sequência do desenvolvimento que respeita a norma EN 50128 produz vária documentação que deverá ser gerida de forma a garantir a rastreabilidade do desenvolvimento do software.

3.2 SCADE Suite 6.0

O *SCADE Suite 6.0* é um *Integrated Development Environment* (IDE) para desenvolvimento seguro de aplicações para sistemas críticos, especialmente adequada para sistemas reactivos, como é o caso dos sistemas de sinalização ferroviária. Cobre todo o processo de desenvolvimento do software, nomeadamente a modelação, verificação de modelos, simulação, verificação formal, geração automática de código, certificado pelas normas EN 50128 e IEC 61508 para qualquer nível SIL, e geração automática de documentação em diferentes fases do ciclo de desenvolvimento.

Em [30], Rushby classifica os MF em 4 níveis de rigor:

- **Nível 0** - Não são utilizados quais queres tipos de métodos formais;
- **Nível 1** - São utilizados conceitos e notações de alguma matemática discreta;
- **Nível 2** - São utilizadas linguagens de especificação com apoio de ferramentas de suporte;
- **Nível 3** - São utilizadas linguagens de especificação com recurso a ferramentas de próprias de apoio à especificação, tal como, ferramentas de apoio à verificação e demonstração de teoremas.

Segundo esta classificação a plataforma *SCADE* está incluída no nível mais alto da escala, nível 3, visto que esta apresenta uma linguagem de especificação, tal como, ferramentas de apoio à verificação.

O *SCADE* surgiu em 1999 num projecto liderado por Daniel Pilaud nos laboratórios franceses Verilog e é um produto comercializado pela Esterel Technologies. É uma criação inspirada em duas ferramentas bem sucedidas, com origem na linguagem síncrona Lustre

(ver secção 2.4.2 na página 18), SAGA e SAO [34]. SAGA era uma ferramenta gráfica desenvolvida em 1986 pela antiga Merlin-Gerin, actual Schneider-Electric, para desenvolver e implementar os seus sistemas nucleares. Possuía já a possibilidade de gerar código de forma automática. SAO era uma ferramenta similar desenvolvida pela actual Airbus para desenvolver o software de bordo do Airbus A340/600 e A380.

Concebida para desenvolver sistemas críticos embebidos, a plataforma *SCADE* é composta por várias ferramentas, tais como: um editor gráfico, um simulador, um verificador de modelos e um gerador de código automático, que traduz a especificação do modelo em linguagem C. Cada uma dessas ferramentas é capaz de gerar documentação e relatórios, o que torna o *SCADE* numa plataforma adequada e completa em vários domínios de aplicação. Assim, o *SCADE* é a ferramenta mais bem sucedida no meio industrial, e as suas aplicações vão desde o sector da aviação - Sistema de controlo de voo do Airbus A340-600 e A380, e sistemas de pilotagem automática para a Eurocopter: EC-135, EC-145, EC-155, EC-225 -, passam pelo sector das energias nucleares - Sistema de protecção do reactor N4, o mais moderno e potente reactor em França - e vão até ao sector dos transporte ferroviários - Sistema de sinalização e controlo ferroviário da Eurostar. Uma visão geral da plataforma é dada pela imagem 3.2

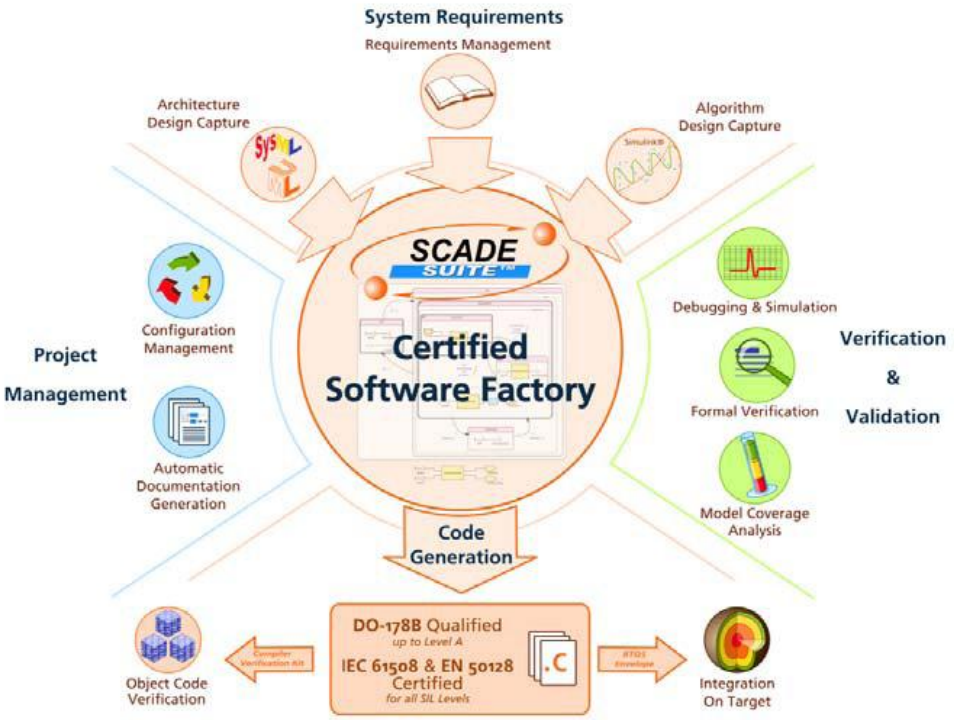


Figura 3.2: Esquema representativo da plataforma SCADE [19]

A plataforma *SCADE* baseia o seu desenvolvimento em modelos. Um modelo é uma representação - com determinado grau de abstracção - do sistema e permite uma melhor leitura e compreensão. Em *SCADE* um modelo é uma descrição precisa com uma notação formal do sistema, e permite que todas as fases de validação, verificação, concepção e manutenção tenham como base esse modelo. Assim, um modelo *SCADE* é a base de comunicação entre todos os elementos da equipa responsável pelo desenvolvimento do sistema, desde o engenheiro de software que o modela até ao programador que o concebe.

Na especificação de modelos, o *SCADE* aplica uma "regra de ouro" que é detectar erros nas fases primárias do projecto. É mais fácil e mais rápida a correcção de um erro numa fase embrionária do que um erro numa fase de concepção. Para além de minimizar o esforço na reparação de falhas, ter um modelo estável como base de desenvolvimento permite maior solidez e confiança no sistema que dele se obtém. Para sustentar esta noção de modelos a plataforma *SCADE Suite 6.0* disponibiliza um conjunto de ferramentas que atravessam de forma transversal o ciclo de vida de um sistema da seguinte forma:

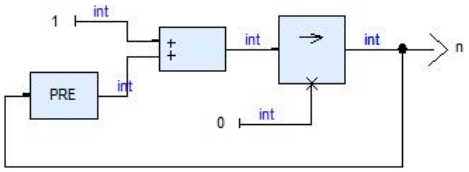
- O modelo são os requisitos do software: é a única referência no projecto com uma notação formal;
- A documentação é automática e gerada directamente a partir do modelo: está correcta, por construção, e actualizada;
- O modelo pode ser simulado utilizando o código final;
- Técnicas de prova formal podem ser aplicadas no modelo para detectar *bugs* ou provar propriedades de segurança do sistema;
- O código é gerado automaticamente e directamente a partir do modelo, através de uma ferramenta certificada KCG: o código é correcto, por construção, e actualizado.

A linguagem Scade[1] é uma linguagem de especificação gráfica, onde cada operador especificado graficamente em Scade tem a sua respectiva representação textual em Lustre. A notação textual é a semântica de referência da plataforma. Para cada modelo produzido em Scade existe uma separação das notações gráfica e textual em ficheiros distintos. Todas as ferramentas da plataforma recorrem à notação textual; a representação gráfica é apenas uma projecção da notação textual. A figura 3.3 exemplifica a referida situação.

3.2.1 Editor

O editor *SCADE* permite a especificação de um modelo de sistema de forma gráfica e textual. A modelação gráfica é feita com recurso ao conceito *drag-and-drop*. A modelação textual é feita usando a notação da linguagem Lustre 2.4.2. Uma especificação pode integrar as duas formas sem qualquer prejuízo. A combinação das duas notações pretende flexibilizar a plataforma, deixando ao critério do utilizador a utilização de uma ou de outra.

[1]Durante este documento a palavra ***SCADE*** - maiúsculas - refere-se à plataforma de desenvolvimento enquanto que a palavra **Scade** refere-se à linguagem de especificação da plataforma *SCADE*.



(a) Representação gráfica

```
node nat() returns (n:int)
let
     n = 1 -> (1 + pre n)

tel
```

(b) Representação textual

Figura 3.3: Representação gráfica *vs* textual

Normalmente recorre-se à notação textual para escrever fluxos que dependem de formulas matemáticas complexas e que são mais facilmente expressas textualmente.

O *SCADE* permite ao utilizador atravessar todas as ferramentas disponibilizadas com uma visão modular do sistema alvo. Esta propriedade permite criar e validar pequenas partes, sem ter a necessidade de modelar o sistema completo, para perceber se pequena parte do modelo está correcta. Uma arquitectura modular permite a reutilização e a consequente redução de tempos de desenvolvimento e respectivo custo.

A linguagem Scade utiliza dois formalismos de especificação: diagramas de blocos (figura 3.4) - propriedade herdada do Lustre (2.4.2 na página 18) - para sistemas predominantes em controlo continuo, e máquinas de estados seguras *Safe State Machine* (SSM) que não são mais que máquinas de estados finitas (figura 3.5) - formalismo herdado do Esterel (2.4.1 na página 16) - para sistemas orientados para um controlo discreto.

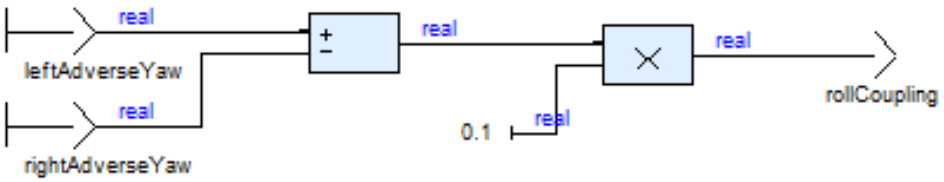


Figura 3.4: Node em SCADE

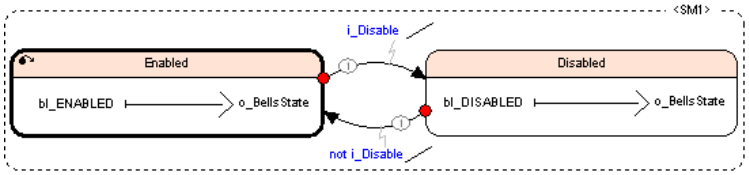


Figura 3.5: Máquina de estados em SCADE

Controlo continuo

No formalismo *data-flow* da linguagem Scade, os `nodes` são a figura central. São representados por caixas e são um encapsulamento de funções matemáticas, filtros e *delays*. As linhas de conexão indicam o fluxo de dados entre os `nodes`. Os `nodes` que não tenham

qualquer relação de dependência, são executados de forma concorrente e comunicam, entre eles, apenas através das linhas de fluxo. Os `nodes` podem processar valores inteiros, reais, *booleanos*, caracteres e estruturados. Estes desempenham um papel fundamental na arquitectura do modelo. Um `node` pode encapsular vários `nodes` e pode, também, ser encapsulado por outro `node`. Esta construção permite estruturar em hierarquia o modelo do sistema, tornando mais fácil a sua leitura e compreensão. Outra propriedade dos `nodes` é a sua modularidade: o comportamento de um `node` não varia com o contexto onde é aplicado.

Numa especificação, utilizando diagramas de blocos, todos os blocos têm um ciclo. Os ciclos podem ser diferentes entre eles. Em cada ciclo, um bloco lê os inputs e gera os outputs. Dentro de cada ciclo de execução, a linguagem Scade permite de forma simples exprimir concorrência de operações - concorrência funcional. Na figura 3.6 os operadores *RollRate::RollRateWarning* e *RollMode::RollMode* são `nodes` paralelos entre si, visto que não há qualquer dependência entre eles e, portanto, são executados em paralelo. Por sua vez o `node` *RollRate::RollRate-Warning*(**a**) é funcionalmente dependente de *RollRate::RollRateCalculate*(**b**) e a execução de **(a)** só sucede após a execução de **(b)**.

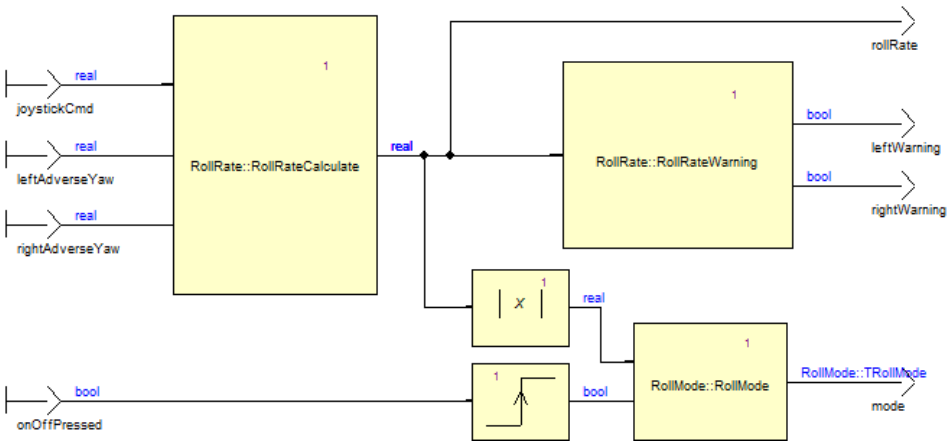


Figura 3.6: Concorrência em Scade

Controlo discreto

No formalismo *control-flow* do Scade - designado por SSM - existe a noção de hierarquia nas máquinas de estados. Assim, faz todo o sentido falar em estados simples e macro-estados. Macro estados, são estados de uma máquina que encapsulam outra máquina de estados. Esta técnica pode ser aplicada sucessivamente até atingir um nível de abstracção pretendido. É dado um exemplo pela figura 3.7.

Uma SSM tem exactamente a mesma noção de ciclo que um `node`. Numa máquina de estados simples, um ciclo consiste em processar a transição adequada do estado actual e executar essa transição, se existir. Máquinas[2] concorrentes comunicam de forma síncrona com recurso ao envio e recepção de sinais de outras máquinas.

[2]Escreve-se máquinas para simplificação de escrita, em vez de máquinas de estados

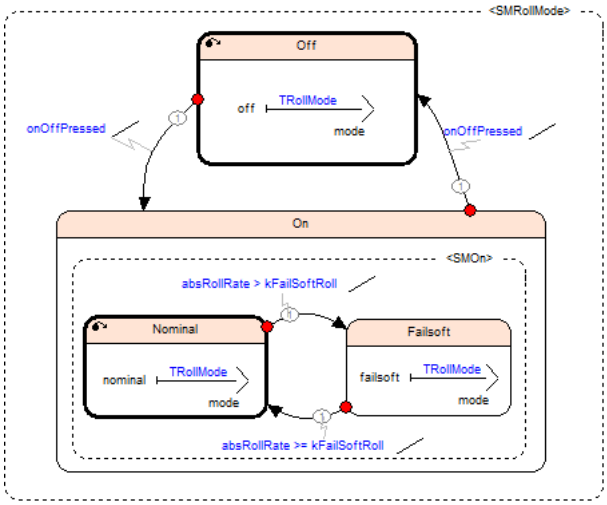


Figura 3.7: Controlo discreto em SCADE

A noção de paralelismo também existe nas SSM do Scade. Na figura 3.8 as máquinas de estados *Machine1* e *Machine2* são executadas em paralelo e comunicam entre si através do sinal *Signal1*. *Signal1* é emitido no estado *Off* da máquina *Machine1* e é testado no estado *Waiting* da máquina *Machine2* para despoletar a transição para *Detected.*

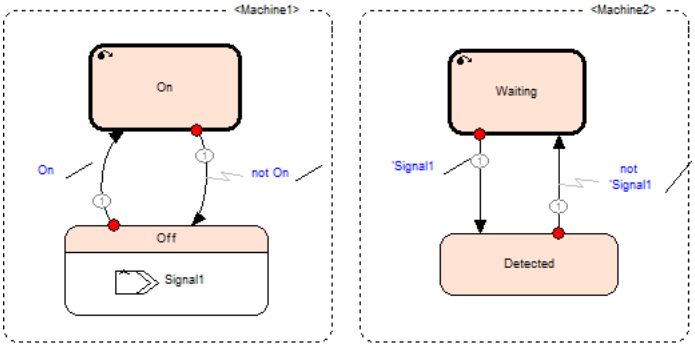


Figura 3.8: Máquinas paralelas

A combinação destes dois formalismos é mais um argumento de peso para a utilização do *SCADE*. Esta combinação é designada em *SCADE* por *Unified Modeling Style*. De facto, sistemas reactivos são, muitas vezes, sistemas grandes e complexos onde existem, paralelamente, controlo discreto e controlo continuo, sendo que, geralmente, o controlo discreto "comanda" o controlo continuo. A linguagem Scade integra-os de tal forma, que é possível, num mesmo modelo, a utilização consistente de ambos, interagindo-os e executando-os em paralelo. Mostra-se um exemplo na figura 3.9.

É importante referir que quando se fala de concorrência, trata-se de concorrência lógica e não concorrência física. A noção de concorrência em Scade não se refere à partilha de

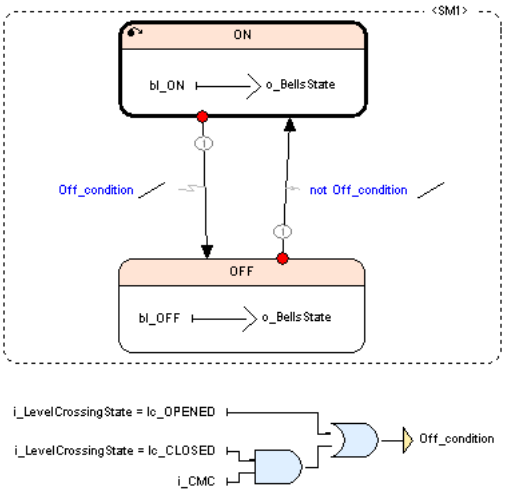


Figura 3.9: SSM e diagramas de blocos em SCADE

recursos mas sim à estrutura funcional do software. O compilador Scade é responsável por traduzir a concorrência especificada no modelo em código sequencial, analisando a dependência entre as construções desenvolvidas.

3.2.2 Verificação Estática

A actividade de verificação estática é feita no *SCADE Suite 6.0* através do *SCADE Model Checker*. Esta ferramenta assume um papel importante na medida em que valida a consistência do modelo especificado. É o responsável pela verificação sintáctica e semântica do modelo. Dentro desta actividade, ele verifica as seguintes propriedades:

- Coerência dos tipos de dados e interfaces (*Type checking*);
- Detecta a falta de definições;
- Alerta quando existem definições nunca utilizadas no modelo;
- Detecta de variáveis não inicializadas;
- Verifica semântica de relógios (2.4.2)

Após a verificação estática, o *SCADE* apresenta o resultado da verificação em formato *Hyper Text Markup Language* (HTML). As anomalias detectadas no modelo pelo *SCADE Model Checker* são *links* que o utilizador pode utilizar para aceder directamente ao ponto onde detectada a anomalia.

3.2.3 Verificação Dinâmica

Em *SCADE* a noção de verificação dinâmica refere-se à "animação" do modelo. A ferramenta que permite esta verificação designa-se de *SCADE Simulator*. É, portanto, uma

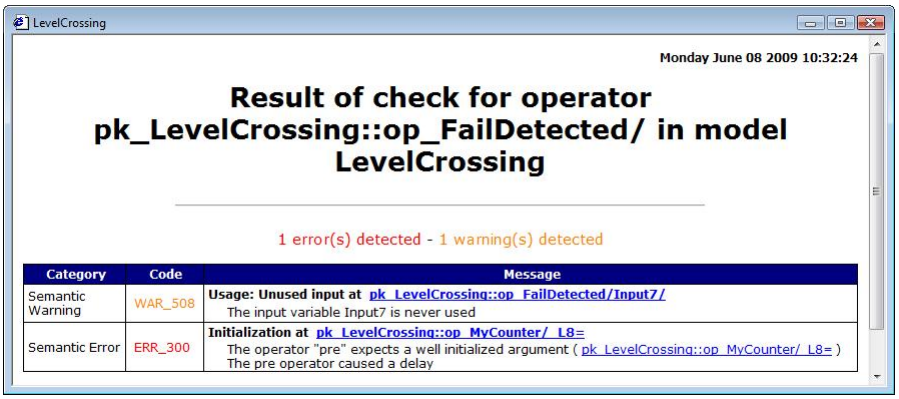


Figura 3.10: Anomalia detectada pelo *SCADE Model Checker*

actividade de verificação por simulação do modelo e corresponde ao tradicional teste. Permite de forma rápida e intuitiva perceber se, em termos gerais, o modelo corresponde ao sistema pretendido. Com o *SCADE Simulator*, o utilizador poderá:

- Configurar e executar sessões de simulação;
- Injectar eventos sobre o sistema;
- Observar reacções graficamente;
- Salvar sessões de simulação e executa-las posteriormente.

Para executar uma sessão de simulação o *SCADE* tem de criar posteriormente o executável do código criado pelo *SCADE KCG*. O executável criado para efeitos de simulação é optimizado para assegurar uma observação do código C.

Um ciclo é a unidade lógica de tempo numa simulação *SCADE* e corresponde a uma execução completa do código C gerado. Em cada ciclo de simulação são determinados os outputs consoante os inputs do modelo.

Uma sessão de simulação pode ser executada ciclo por ciclo ou então de forma contínua. O utilizador tem a possibilidade de alternar entre ambas as possibilidades sem ter de reiniciar o processo. Tem ainda a possibilidade de definir *break points* e fazer parar a execução mediante determinado acontecimento no modelo.

Em tempo de simulação é também possível observar graficamente variáveis escalares do sistema. Variáveis não escalares podem ser observadas sob a forma de uma quadro onde é possível determinar em cada ciclo de execução o seu valor ou estado.

3.2.4 Verificação Formal

A verificação formal do modelo é feita através de um processo de verificação de modelos (*model checking*) utilizando o *SCADE Design Verifier* que é uma ferramenta integrada no *SCADE* desenvolvida pela Prover Tecnologies. O *SCADE Design Verifier* permite provar que determinada "boa" propriedade permanece sempre no modelo, e que determinada

propriedade "má" nunca acontece. No entanto, a sua aplicabilidade não se resume à prova de propriedades. O *SCADE Design Verifier* pode ser usado, por exemplo, para provar a não regressão de um modelo verificando equivalência entre modelos. A granularidade com que o *SCADE Design Verifier* é aplicado depende exclusivamente do utilizador. Pode ser aplicado a um modelo completo, somente a alguns operadores, ou mesmo, a uma parte de um operador.

Para provar determinada propriedade, o *Design Verifier* executa uma procura exaustiva, no espaço de todos os cenários possíveis do modelo, para verificar que esta é sempre verdadeira. O *Design Verifier* consegue verificar propriedades expressas em formulas *booleanas* contendo condições sobre variáveis *booleanas*, variáveis de dados e operadores de tempo.

Exprimir propriedades

A actividade de verificação executada pelo *SCADE Design Verifier* é idêntica à executada em Lustre (2.4.2) com a noção de *observers*. Para provar um conjunto de propriedades em Scade, o utilizador terá de especificar um operador - *property node* - para cada uma delas, que exprima, usando o formalismo *data-flow*, a propriedade que se pretende verificar. O único output de um *property node* deve ser do tipo *booleano* que indica a validade, ou não, da propriedade. De seguida, deve criar o *observer* onde os inputs são os mesmos do sistema em teste. O seus outputs devem corresponder à totalidade dos outputs das propriedades. Na figura 3.11 podemos ver um esquema que reflecte esta técnica.

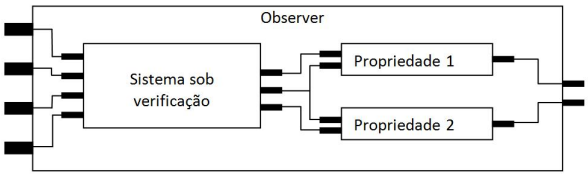


Figura 3.11: Verificação com *observer*

Especificar objectivos de prova

O acto de exprimir as propriedades que se pretendem verificar é feito com recurso ao *SCADE Editor*. Depois de expressas é necessário "informar" o *SCADE Design Verifier* de que o resultado do *Property node* no *Observer* é um objectivo de prova - *Proof objective*. O *SCADE* permite a verificação de algumas propriedades *standard*, tais como: *Division-by-zero* e *Overflow*. A indicação de qualquer um destes objectivos de prova é feita recorrendo a um menu de selecção (ver figura 3.12).

O utilizador poderá gerir os seus objectivos por hierarquias, agrupando e encapsulando-os segundo algum critério. Isto permite-lhe, para além de uma melhor organização do projecto, executar processos de verificação modulares.

(a) Objectivos Standard

(b) Objectivos especificados em *property nodes*

Figura 3.12: Criar objectivos de prova

Especificar estratégias de verificação

O *SCADE Design Verifier* permite configurar a actividade de verificação a executar. É possível indicar estratégias de verificação que influenciarão o processo de verificação. As estratégias de verificação baseiam-se em dois conceitos distintos:

- *Prove:* Este é aplicado quando não for indicada qualquer opção. Estratégias desta família provam propriedades ou encontram contra-exemplos para propriedades falsas;
- *Debug:* Deve ser aplicada este tipo de estratégia quando se suspeita que uma propriedade não é satisfeita e se pretende apenas obter um exemplo onde ela falha.

O utilizador poderá criar tantas estratégias de prova quantas quiser, baseadas nos dois conceitos acima referidos, alterando alguns parâmetros nas propriedades gerais da espécie de verificação (ver figura 3.13).

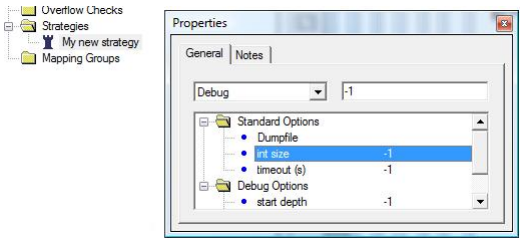


Figura 3.13: Criação e configuração de estratégias de verificação

Executar a verificação

Depois de especificados os objectivos de prova e configuradas as respectivas estratégias de verificação, pode então dar-se início ao processo de verificação. Este processo pode ser efectuado para vários objectivos em simultâneo, seleccionando um grupo de objectivos, ou objectivo a objectivo (ver figura 3.14).

Analisar resultados

Após um processo de verificação o *SCADE Design Verifier* produz resultados em vários formatos. Numa primeira instância o *SCADE* produz como que um quadro onde mostra

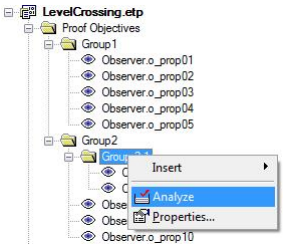


Figura 3.14: Executar verificação de forma modular

o resultado obtido para cada uma das propriedades. Essa mensagem pode ter os valores descritos na tabela 3.1 (exemplo na figura 3.15).

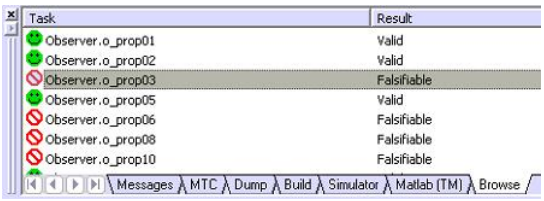


Figura 3.15: Resultados do *SCADE Design Verifier*

No caso de uma propriedade ser falsa no sistema, o *SCADE Design Verifier* apresenta um contra-exemplo para que o utilizador possa verificar, através dos valores dos inputs, uma situação em que a propriedade que tenta provar falha (ver figura 3.16).

Outra das formas de apresentação de resultados do *SCADE Design Verifier* é em HTML. Esta apresentação é bastante formal e dá indicações acerca do processo de verificação e da propriedade verificada. No caso em que a propriedade não é válida, o *SCADE Design Verifier* gera um ficheiro com o contra-exemplo, sob a forma de cenário, e permite ao utilizador executar uma sessão de simulação com o cenário gerado (ver figura 3.17). Cada vez que o *SCADE Design Verifier* é executado este relatório em formato HTML é guardado no directório do projecto - `<Project_dir>/reports/<time_stamp>` - e portanto é sempre possível analisar e comparar evoluções no estado de verificação do sistema.

3.2.5 Análise de cobertura de modelos

O *SCADE MTC* é uma ferramenta integrada no *SCADE Suite 6.0* que permite efectuar uma actividade de verificação dinâmica no modelo (ver secção 3.2.3). Por outras palavras, permite determinar dinamicamente o nível de profundidade que a simulação do modelo atingiu. A vantagem deste tipo de análise é a detecção de partes "mortas" no modelo, ou seja, é possível detectar partes do modelo que nunca são activadas e portanto desnecessárias.

A sequência de passos, a efectuar no *SCADE*, de modo a proceder a uma actividade de cobertura num modelo já especificado é a seguinte:

- **Criar cenários de teste:** deve ser criado um conjunto de testes baseados nos requisitos do sistema. Para tal recorre-se ao *SCADE Simulator*;

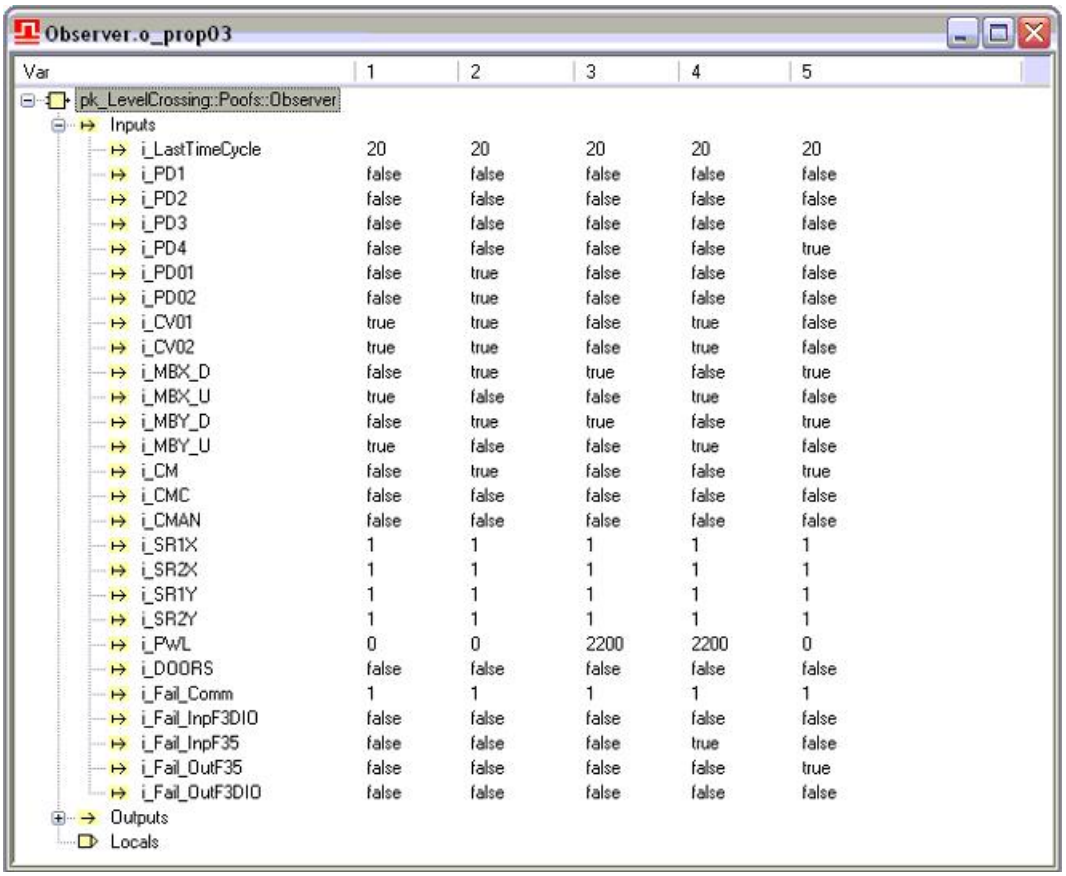


Figura 3.16: Contra-exemplo obtido pelo *SCADE Design Verifier*

- **Instrumentação do modelo:** é um processo automático que define os critérios de cobertura com base em librarias. Estas, podem ser pré-definidas do *SCADE MTC* ou criadas pelo utilizador;
- **Executar a simulação com os cenários criados:** o modelo instrumentado é executado numa sessão de simulação. Os cenários criados devem ser simulados no *SCADE MTC* para medir a sua contribuição na cobertura do modelo;
- **Análise do resultado da verificação:** o modelo pode ser analisado graficamente no *SCADE MTC* mediante a consulta de gráficos e tabelas de cobertura. Os resultados são apresentados usando uma coloração - que pode ser definida pelo utilizador - que indica o nível de cobertura dos operadores (ver figura 3.18).
- **Justificação de resultados:** a cobertura de determinados operadores pode nunca atingir os 100%. Isto acontece quando é desenvolvido um operador genérico que depois é aplicado num contexto que não exige, ou não recorre, a toda a funcionalidade providenciada pelo operador. Nestas situações o utilizador poderá apresentar, no *SCADE*

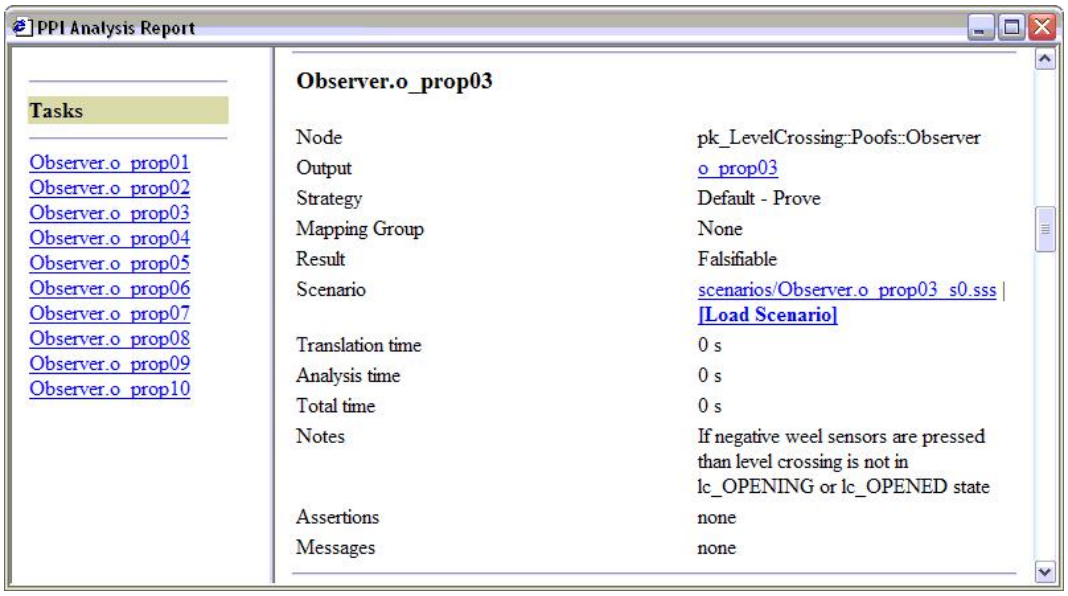


Figura 3.17: Resultados do *SCADE Design Verifier* em formato HTML

MTC, uma justificação, em linguagem natural, que indica a razão de pela qual não foi atingida uma cobertura de 100%.

- **Geração do relatório:** no final, o utilizador pode gerar de forma automática um relatório, em formato `*.html` ou `*.rtf`, que indica a cobertura total do sistema, tal como, a cobertura atingida em cada operador.

O *SCADE MTC* actua ao nível dos objectos e construtores para os quais existe um critério de cobertura, e actua nas diferentes partes do modelo da seguinte forma:

- ***Data-flow*:** verifica que todos os operadores condicionais e operadores lógicos assumem todas as combinações possíveis de inputs.

- ***Control-flow*:** verifica que todos os estados são atingidos e que todas as transições são despoletadas.

- **Sinais:** verifica que todos os sinais são emitidos e capturados.

3.2.6 Gerador de código

O *SCADE KCG* é uma ferramenta capaz de a partir de uma especificação Scade, gráfica ou textual, reproduzir o seu comportamento em linguagem C. O processo de transformação de uma especificação Scade em código C é uma actividade iterativa, efectuada em três fases:

Resultado	Significado
Valid	A propriedade verificada é sempre verdadeira matematicamente.
Falsiable	A propriedade é falsa. O *SCADE Design Verifier* detectou um estado do sistema onde a valoração dos inputs determina um output no *observer* que não corresponde ao especificado no objectivo de prova. O valoração dos outputs é apresentado como contra-exemplo.
Indeterminate	O *SCADE Design Verifier* não atingiu uma conclusão significativa.
Interrupted	Interrupção da verificação por indicação do utilizador, ou o *SCADE Design Verifier* não encontrou nenhum contra-exemplo antes do tempo limite definido na estratégia de verificação.
Stop Depth Reached	A verificação atinge a profundidade especificada na estratégia de verificação e o *SCADE Design Verifier* não consegui atingir um resultado significativo.
Raised an Error	Acontece, normalmente, quanto se procede à verificação quando o modelo contem erros sintácticos e/ou semânticos. A causa do erro é descrita no relatório da verificação.
Error: Non linear property	A verificação é impossível porque a propriedade possui expressões e/ou funções não lineares. Deve ser simplificada a propriedade em verificação com recurso a verificação modular.
Contradictory	Foram expressas propriedades, sob a forma de afirmações - *assertions* - que são contraditórias. O *SCADE Design Verifier* é incapaz de atingir qualquer resultado e para o processo de verificação.

Tabela 3.1: Resultados possíveis de uma verificação para uma determinada propriedade.

1. ***Load* do modelo Scade**. Nesta fase são activadas tarefas de verificação estática, tais como, verificação de tipos, verificação sintáctica e semântica, semântica de relógios, etc.

2. **Transformação da especificação Scade em linguagem Lustre** usando a ferramenta **S2L**. Esta fase é composta por dois passos:

   Normalização: a transformação é baseada em técnicas de reescrita que asseguram a consistência e a semântica do modelo;

   Optimização: neste passo as equações *data-flow* são optimizadas.

3. **Transformação da especificação Lustre em linguagem C** usando a ferramenta **L2C**. Esta fase é composta por três passos:

   Semântica sequencial: a linguagem Scade é uma linguagem com base na Hipótese Síncrona (er secção 2.3) e portanto permite ao engenheiro de software idealizar

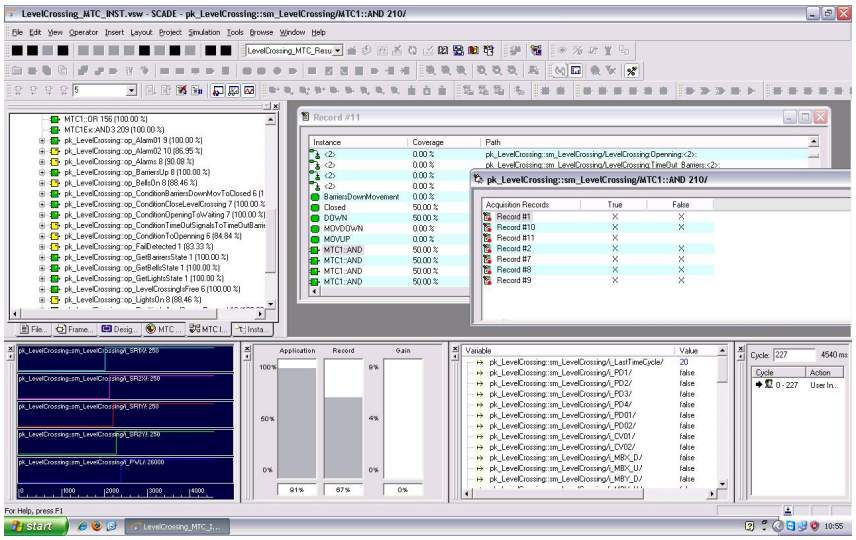


Figura 3.18: Ambiente do *SCADE MTC*

o seu programa com execuções paralelas. Neste passo é efectuada a transformação do estilo declarativo e concorrente, providenciado pelo Scade, para o estilo imperativo e sequencial da linguagem C, analisando a dependência entre as construções desenvolvidas no Scade.

Optimização: actividades de optimização de equações e variáveis de memória.

Geração de código: é o último passo e é responsável por gerar o código C correspondente ao modelo.

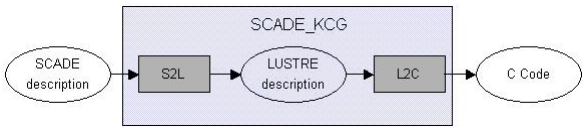


Figura 3.19: Processo de geração de código usado pelo *SCADE KCG*

Ao utilizador é permitida alguma flexibilidade na geração de código. Se assim o entender o utilizador poderá criar o código de todo o sistema, ou então, criar o código operador a operador. Na primeira situação, o utilizador terá de indicar qual o operador principal - *main*. O *SCADE KCG* permite a geração de código de dois modos diferentes, ficando ao critério do utilizador qual o modo a utilizar:

- *Call*: para cada `node` há uma função C que lhe corresponde e sempre que houver uma referência a esse `node` é utilizada a função que o define;
- *Inline*: não existe uma função C associada aos `nodes`. Quando existe uma referência a um `node` é reescrito todo o código que reflecte o seu comportamento.

O utilizador poderá combinar ambos os modos, visto que o *SCADE KCG* permite seleccionar esta opção em cada `node` do sistema. Na figura 3.20 pode comparar-se os modos.

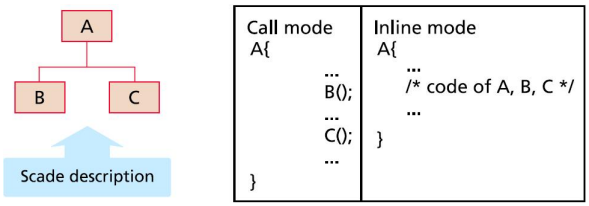


Figura 3.20: Comparação do modo Call e do modo Inline

Para providenciar uma análise de rastreabilidade automática, no acto de compilação do *SCADE KCG*, é produzido um ficheiro em formato *Extensible Markup Language* (XML) onde estão estabelecidas as correspondências entre o modelo Scade e o código C produzido. A figura 3.21 mostra um exemplo.

```
<Package scadeName="pk_LevelCrossing">
  <NoExpNode scadeName="op_T10"  targetCycleFct="op_T10_pk_LevelCrossing"  targetInitFct="op_T10_reset_pk_LevelCrossing"
    <OutCtxType targetName="outC_op_T10_pk_LevelCrossing"/>
    <StateVector targetName="op_T10_SV_pk_LevelCrossing"/>
    <Input scadeName="i_LastTimeCycle"  scadeType="int"  targetName="i_LastTimeCycle"  targetType="kcg_int"/>
    <Input scadeName="i_Reset"  scadeType="bool"  targetName="i_Reset"  targetType="kcg_bool"/>
    <Output scadeName="o_reached"  scadeType="bool"  targetName="o_reached"  targetType="kcg_bool"  inCtx="true"/>
    <Local scadeName="_L5"  scadeType="bool"  targetName="_L5"  targetType="kcg_bool"  inCtx="true"/>
    <Local scadeName="_L4"  scadeType="real"  targetName="_L4"  targetType="kcg_real"  inCtx="true"/>
    <Local scadeName="_L2"  scadeType="bool"  targetName="_L2"  targetType="kcg_bool"  inCtx="true"/>
    <Local scadeName="_L1"  scadeType="int"  targetName="_L1"  targetType="kcg_int"  inCtx="true"/>
    <NodeInstance scadeName="pk_LevelCrossing::op_MyTimer_Real"  refName="op_MyTimer_Real_pk_LevelCrossing"  instName="1"
      <OutCtxVar targetName="Context_op_MyTimer_Real"/>
    </NodeInstance>
  </NoExpNode>
```

Figura 3.21: Ficheiro XML produzido pelo *SCADE KCG*

A geração do código mantém, até certo ponto, a nomenclatura definida no modelo. Desta forma, é possível efectuar uma análise bidireccional que permite determinar a correspondência entre o modelo Scade e o código C gerado pelo *SCADE KCG*. A figura 3.22 mostra a relação entre o modelo Scade e o código C produzido.

O código gerado tem as seguintes propriedades:

- O código é portável: é código ISO-C [31];
- A estrutura do código reflecte a arquitectura do modelo Scade no que se refere aos diagramas de blocos. Relativamente à parte das máquinas de estados é assegurada rastreabilidade através dos nomes dos estados;
- O código é legível e rastreável a partir do modelo Scade e através de correspondência de nomes de estruturas e comentários introduzidos no modelo;
- Não existe gestão de memória dinâmica gestão. A gestão de memória é estática;
- A recursividade é evitada;

```
switch (SSM_SM1_dispatch_sel) {
  case SSM_SM1_Manual__pk_LevelCrossing :
    if (inC->i_Manual)
      {
        SSM_SM1_selec_dispatch_act = SSM_SM1_Manual__pk_LevelCrossing;
      }
    else
      {
        SSM_SM1_selec_dispatch_act = SSM_SM1_Automatic__pk_LevelCrossing;
      }
    outC->SSM_SM1_reset_act = (kcg_true ^inC->i_Manual);
    SSM_SM1_dispatch_act = SSM_SM1_selec_dispatch_act;
    break;
  case SSM_SM1_Automatic__pk_LevelCrossing :
    if (inC->i_Manual)
      {
        _1_SSM_SM1_selec_dispatch_act = SSM_SM1_Manual__pk_LevelCrossing;
      }
    else
      {
        _1_SSM_SM1_selec_dispatch_act = SSM_SM1_Automatic__pk_LevelCrossing;
      }
    outC->SSM_SM1_reset_act = inC->i_Manual;
    SSM_SM1_dispatch_act = _1_SSM_SM1_selec_dispatch_act;
    break;

}
switch (SSM_SM1_dispatch_act) {
  case SSM_SM1_Manual__pk_LevelCrossing :
    outC->SSM_SM1_reset_nxt = kcg_false;
    outC->SSM_SM1_dispatch_nxt = SSM_SM1_Manual__pk_LevelCrossing;
    outC->o_LevelCrossingMode = lc_MANUAL_pk_LevelCrossing;
    break;
  case SSM_SM1_Automatic__pk_LevelCrossing :
    outC->SSM_SM1_reset_nxt = kcg_false;
    outC->SSM_SM1_dispatch_nxt = SSM_SM1_Automatic__pk_LevelCrossing;
    outC->o_LevelCrossingMode = lc_AUTOMATIC_pk_LevelCrossing;
    break;

}
outC->init = kcg_false;
```

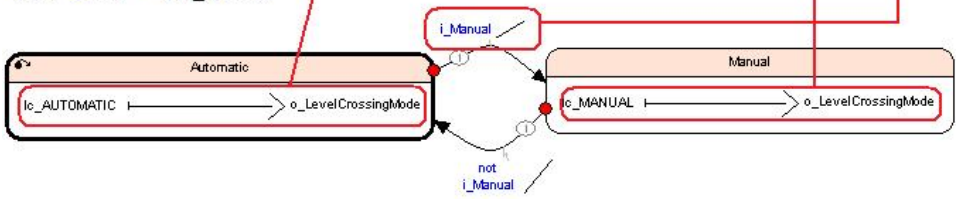


Figura 3.22: Rastreabilidade do código gerado

- Os ciclos são permitidos desde que estes tenham limites estáticos e conhecidos pelo KCG no momento de compilação;
- O tempo de execução e limitado;
- O código é decomposto em atribuições elementares a variáveis locais;

- Utilização de parênteses de forma explícita;
- Não existe acesso dinâmico à memória (não existe a aritmética de apontadores);
- Não existem conversões implícitas;
- Não existem expressões com efeitos colaterais: (`i++`, `a+=b`);
- As funções nunca são passadas como argumentos;

3.2.7 SCADE Suite 6.0 na Norma EN 50128

SCADE Suite 6.0 é uma plataforma de desenvolvimento certificada pela TÜV SÜD[3] segundo as normas EN 50128 (ver secção 2.6.2 na página 29) e IEC 61508 (Transportes e indústria em geral).

O processo de desenvolvimento da norma EN 50128 está estruturado em fases distintas. Em cada uma dessas fases devem ser desenvolvidas tarefas especificas e produzida a devida documentação. Na figura 3.23 pode ser constatada a semelhança com o ciclo de desenvolvimento definido pela norma (ver figura 2.10).

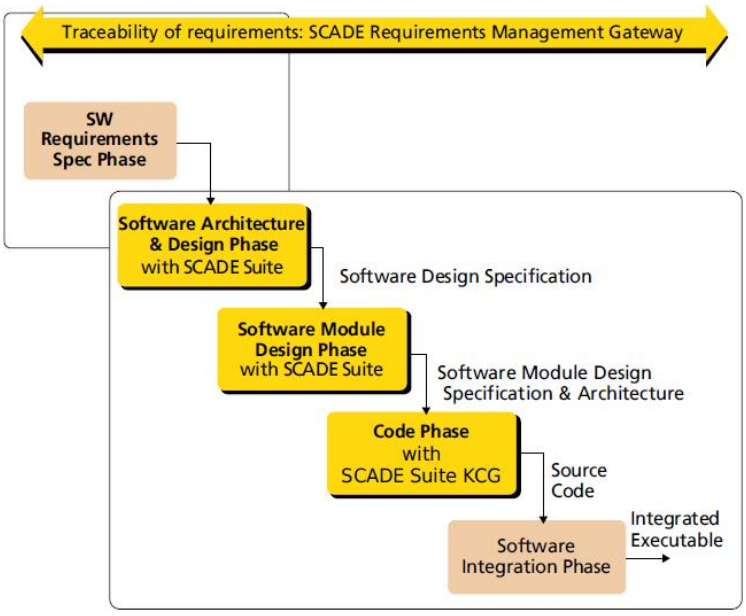


Figura 3.23: Processo de desenvolvimento de software em SCADE

Gestão de Requisitos do Software A clausula 8 da norma EN 50128 define o objectivo da fase de Especificação dos Requisitos de Software:

[3]Entidade alemã em que os principais sectores de actividade são o controlo de qualidade e a certificação

- Descrever um documento que define um conjunto completo de requisitos para o software, que cumpra todos os requisitos do sistema, no âmbito requerido pelo nível de Integridade de Segurança do Software. Constitui um documento abrangente para todos os técnicos de software e evita a necessidade de procurar os requisitos em qualquer outro documento;

- Descrever a Especificação dos Testes dos Requisitos do Software.

Nesta fase actua o *SCADE Requirements Management Gateway (RMG)* que permite gerir toda a documentação relativa ao software e obter uma matriz de rastreabilidade entre ela. A figura 3.24 mostra o módulo *SCADE RMG* na gestão de requisitos dos sistema. Ainda

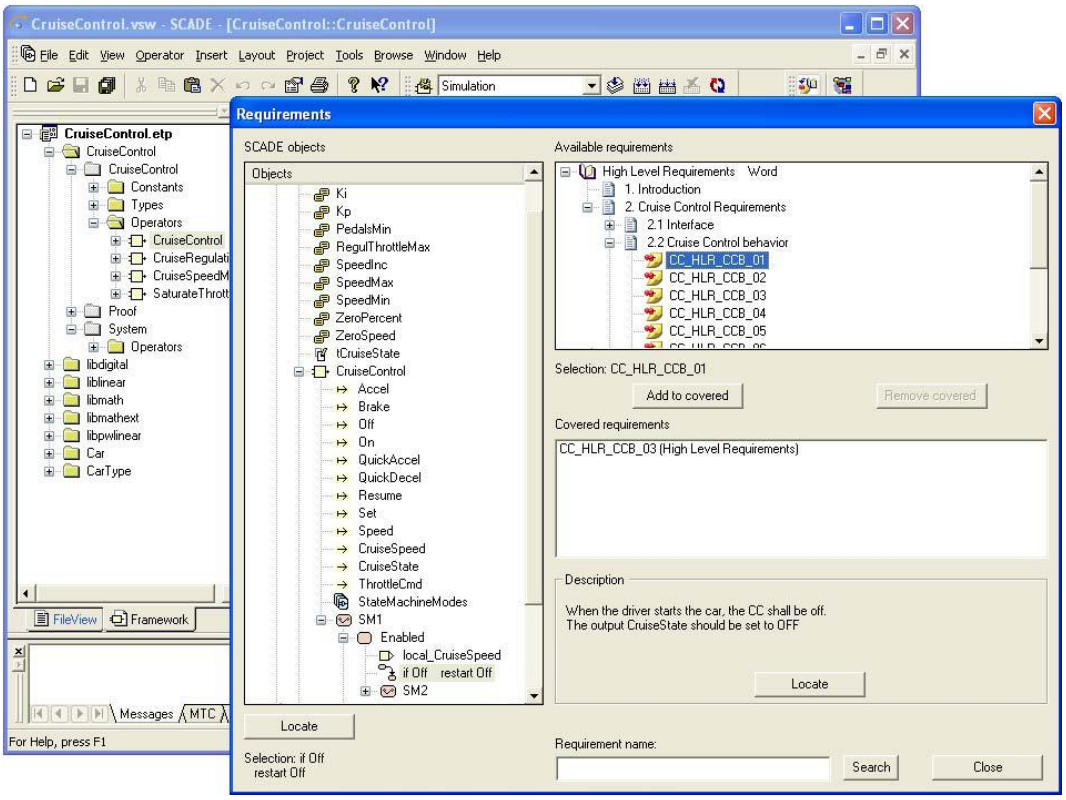


Figura 3.24: Rastreabilidade usando o SCADE RMG

nesta fase, o *SCADE* possui um módulo de interface com outras ferramentas de gestão de requisitos, nomeadamente o *DOORS* (ver secção 3.1). O *SCADE* suporta a exportação de um projecto para um módulo formal do *DOORS*, onde a hierarquia importada neste, é similar à hierarquia definida no *SCADE*. A hierarquia inicia-se em quatro elementos distintos e é constituída por constantes, variáveis, tipos e operadores.

Com um "click" do rato é possível navegar entre a arquitectura do software, o desenho do software e desenho dos módulos de software, para se perceber o impacto de uma mu-

dança na implementação dos módulos do software, etc. A análise de rastreabilidade pode ser executada para detectar requisitos não implementados, falta de requisitos, etc.

Arquitectura do Software A clausula 9 da norma EN 50128 define os objectivos da fase de Desenho e Arquitectura do Software da seguinte forma:

- Desenvolver uma arquitectura de software que cumpra os requisitos da Especificação dos Requisitos do Software na medida requerida pelo nível de Integridade e Segurança do software;
- Examinar os requisitos impostos ao software pela arquitectura do sistema;
- Identificar e avaliar a importância das interacções hardware/software para a segurança;
- Escolher um método de concepção se não tiver já sido definido.

Nesta fase o editor *SCADE* permite a modelação abstracta do software. Um modelo, nesta fase, pode ser parcial e indicar apenas a estrutura e interfaces de topo do sistema. Este modelo *SCADE* deve identificar as funções de mais alto nível, formalizar as interfaces dessas funções, tais como: nome e tipos de dados, descrever o fluxo de dados entre essas funções, verificar consistência de dados e preparar a *framework* para o processo de desenho. Uma boa arquitectura do software deve assegurar:

- Estabilidade e Manutenção: a equipa responsável pelo projecto precisa de uma plataforma de desenvolvimento estável no início do desenvolvimento, bem como após operações de actualização;
- Compreensão e Análise: a compreensão do modelo advém de uma especificação clara e sem ambiguidades. A semântica Scade e a notação gráfica, simples e intuitiva, permitem uma leitura clara do modelo. O formalismo da linguagem permite a análise dos modelos;
- Eficiência: uma arquitectura modular permite a reutilização e a consequente redução de tempos de desenvolvimento e respectivo custo.

Nesta fase, o *SCADE Semantic Checker* assume um papel importante na medida em que valida a consistência da arquitectura modelada. Esta verificação semântica pode, e deve, ser feita de forma modular. O documento *SCADE*, gerado de forma automática, relativo ao desenho e arquitectura do modelo, pode ser anexado aos documentos de Especificação de Desenho de Software e Especificação de Arquitectura de Software.

Desenho do Software A clausula 10 da norma EN 50128 define o objectivo da fase de implementação e desenho do software:

- Conceber e desenvolver software para o nível de segurança de software definido na Especificação dos Requisitos do Software e na Especificação da Arquitectura do Software;

- Conseguir software que seja analisável, testável, verificável e manutenível. O teste dos módulos também está incluído nesta fase. Como a verificação e os testes serão um elemento crítico para a validação, deverá ser dedicada particular atenção às necessidades de verificação e testes durante a concepção e o desenvolvimento, de forma a assegurar que o sistema resultante e respectivo software serão facilmente testáveis desde o início;

- Seleccionar um conjunto de ferramentas adequadas, incluindo linguagens e compiladores, que apoiem a verificação, validação, avaliação e manutenção, para o nível de Integridade de Segurança de Software requerido, ao longo de todo o ciclo de vida do software;

- Efectuar a integração do software.

A partir do momento em que existe a Especificação de Arquitectura do Software inicia-se a Especificação do Desenho do Software. A segunda não é mais do que um refinamento da primeira. O objectivo desta actividade de refinamento é produzir um modelo completo e consistente do software. Nesta fase existem algumas "regras de ouro" que devem ser seguidas para consolidar o modelo:

- Nunca confiar nos inputs externos: devem ser explicitamente providenciadas medidas de verificação e consolidação dos dados introduzidos. Aqui o Scade disponibiliza "bibliotecas" com operadores de filtragem, de confirmação e limitação de valores.

- De forma similar, os outputs devem ser validados antes de serem passados ao actuador.

Codificação do Software O passo seguinte à verificação formal do modelo é a codificação do software. Aqui, o *SCADE* joga a seu trunfo. Com base na ferramenta certificada *SCADE KCG* é capaz, de forma automática, de gerar o código C que implementa o desenho do software especificado em Scade. Não se trata apenas da geração dos cabeçalhos das funções, mas sim, do código completo. O comportamento do código obtido pelo *SCADE KCG* é, garantidamente, igual ao comportamento do modelo Scade. A ferramenta é certificada pela norma EN 50128 até SIL4.

Integração do Software Normalmente, um sistema completo integra vários softwares. A integração de software é uma actividade que trata, essencialmente, da combinação de todos os softwares num todo. A interface com sensores ou qualquer outro tipo de dispositivos de captura de dados do ambiente é feita através de controladores de software especificos, vulgarmente designados por *drivers*. Se a aquisição de dados é feita de forma sequencial então o *driver* deve passar os seus inputs directamente para o programa Scade. Se a aquisição é feita mediante determinados eventos então é necessário garantir que em cada ciclo do programa *SCADE* os inputs são estáveis enquanto este executa um ciclo. Esses *drivers* não são, normalmente, desenvolvidos em Scade, mas sim em C ou *assembly*. A plataforma

SCADE permite a importação de novos tipos e funções. Esses tipos e funções podem, perfeitamente, ser os *driver* de que falamos. O *SCADE Simulator* permite a ligação dessas funções ao código gerado pelo *SCADE KCG*.

Verificação do Software Após o desenho do software deve proceder-se à actividade de validação e verificação. A norma EN 50128 define **verificação** como a "actividade que visa determinar, por análise e testes, que o resultado de cada fase do ciclo de vida preenche os requisitos da fase anterior;" e **validação** como a "actividade que visa demonstrar, por análise e testes, que o produto satisfaz, de todos os pontos de vista, os requisitos definidos;".

Por outras palavras, a verificação trata em verificar a consistência do desenho do software enquanto que a validação trata da consistência dos requisitos relativamente ao sistema pretendido. Claramente, o que nos interessa neste trabalho é a actividade de verificação. Esta actividade pode, e deve, ser executada em 3 fases distintas:

1. Verificação do desenho e arquitectura do software
2. Verificação da especificação dos módulos
3. Verificação da codificação

Verificação do desenho e arquitectura do software Trata-se de verificar que o desenho e arquitectura proposta implementa correctamente os requisitos do sistema. Nesta fase a arquitectura está definida num modelo demasiado abstracto e a actividade de verificação resume-se a uma revisão das notações textuais do modelo e verificação de consistência nas interfaces da arquitectura através do *SCADE Model Checker* (verificação sintáctica e semântica do modelo).

Verificação da especificação dos módulos Nesta fase já é possível executar uma verificação mais aprofundada com base no *SCADE Model Checker*. Esta ferramenta da plataforma *SCADE* efectua uma análise sintáctica e semântica do modelo que inclui:

- Detecção a falta de definições;
- Aviso de definições nunca utilizadas no modelo;
- Detecção de variáveis não inicializadas;
- Coerência dos tipos de dados e interfaces;
- Coerência de relógios.

Um modelo em Scade descreve um módulo completo e detalhado, e desenha o software correspondente. Visto que a notação Scade é definida formalmente então é possível a verificação formal. Segundo a norma EN 50128 [27] **Verificação** é "a actividade que visa determinar, por análise e testes, que o resultado de cada fase do ciclo de vida preenche os requisitos da fase anterior". Aproximando esta definição ao *SCADE*, a actividade de verificação é determinar a conformidade entre o modelo especificado, a Especificação do

Desenho do Software e a Especificação da Arquitectura do Software. Esta determinação é efectuada em três fases distintas:

- **Revisão:** Técnica essencial, pela qual se faz verificação através de gestão documental. Nesta parte, o *SCADE* permite a geração de um documento que contem toda a informação relativa ao modelo especificado em Scade, tais como, tipos, dados, operadores, etc. Neste aspecto a notação Scade possui vantagens relativamente à notação textual:

  - A descrição não está sujeita a interpretações. Isto, porque a notação Scade é formal;

  - A descrição é completa. A não completude é detectada automaticamente pelo *SCADE Model Checker*;

  - A notação gráfica é simples e intuitiva.

- **Simulação:** À medida que se vão criando os diversos módulos do software é possível proceder a actividades de simulação utilizando o *SCADE Simulator*. A simulação é uma técnica usual e essencial pela qual se pode verificar que a especificação corresponde, de certo modo, aos requisitos do software;

- **Verificação formal:** Verificação baseada em técnicas formais. A simulação e/ou teste, não garantem a correcção do software (ver secção 2.1.2. A verificação formal recorre a propriedades matemáticas para provar, com rigor e completude, as propriedades de segurança do sistema. No *SCADE*, a ferramenta que, de forma integrada, permite esta verificação é o *SCADE Design Verifier*.

A actividade de verificação formal do sistema é uma tarefa complexa que requer, a cima de tudo, rigor. Aplicar várias técnicas e métodos não pode ser considerado exagero, principalmente quando se tratam de aplicações onde a segurança é o principal requisito do sistema.

3.3 ELOP II Factory

O *ELOP II Factory* é uma ferramenta desenvolvida especificamente para programar, parametrizar e configurar controladores de lógica programável do fabricante alemão HIMA. Os controladores HIMA são utilizados há vários anos e em vários projectos pela EFACEC para implementar os seus sistemas na área da sinalização ferroviária. São autómatos certificados pela TÜV SÜD, com nível de segurança SIL 4 para várias normas incluindo a EN 50128. Por este motivo a EFACEC não abdica destes equipamentos nem do *know-how* adquirido ao longo dos últimos anos.

HIMatrix F3 DIO O HIMatrix F3 DIO é um dispositivo compacto revestido em metal e possui 16 entradas digitais 8 saídas digitais. É utilizado como extensão de um controlador e não possui nenhum programa em execução. As comunicações com o controlador são feitas através de *ethernet*, controlada pelo sistema operativo da *Central Processing Unit* (CPU).

HIMatrix F35 O controlador HIMatrix F35 é compacto, revestido a metal e com nível de segurança SIL 4 certificado pela TÜV SÜD. Possui 24 entradas digitais, oito saídas digitais, dois contadores e oito saídas analógicas. Internamente, possui dois processadores sendo que um é o CPU, onde será executada a aplicação pretendida com acesso a todos a todas as entradas e saídas do controlador. O outro é o processador das comunicações (COM), responsável pela comunicação com outros controladores e sem acesso às entradas e saídas do controlador. A comunicação entre os dois processadores é feita através da emissão de sinais. Na CPU só é permitida a execução de software criado com a ferramenta *ELOP II Factory*, enquanto que, para o segundo já é possível descarregar um programa em C, desenvolvido pelo utilizador. Portanto, o processador COM é considerado um processador com comportamento não seguro e não é permitida a interacção deste com a CPU. Esta filosofia faz com que não seja possível executar um programa, no processador principal do controlador, que não tenha sido desenvolvido pelo *ELOP II Factory*.

(a) HIMatrix F3 DIO (b) HIMatrix F35

Figura 3.25: Dispositivos de lógica programável

Consequência O facto de não ser possível executar, nos controladores acima referidos, código que não tenha sido gerado pelo *ELOP II Factory* significa, logo à partida, que uma das principais e mais importantes ferramentas do *SCADE* não pode ser utilizada. O código certificado gerado pelo *SCADE KCG*, não é utilizável nos controladores HIMA. Assim, a codificação do software passa a ser um processo não certificado e manual utilizando o *ELOP II Factory*.

Como parte integrante de certificação SIL4 dos controladores acima referidos, o fabricante só permite que estes dispositivos apenas executem programas desenvolvidos pela ferramenta *ELOP II Factory*. O *ELOP II Factory* é uma plataforma de programação, parametrização e configuração de controladores HIMA. Providencia, ao utilizador, um ambiente de programação gráfica utilizando o conceito *drag&drop*.

Esta ferramenta utiliza todas a funções e variáveis da norma IEC 61131-3, norma responsável pela definição de linguagens de programação para dispositivos do tipo *Programmable Logic Controller* (PLC). Pela norma atrás referida, existem quatro tipos de linguagens de programação para PLC, duas delas textuais e duas gráficas:

1. Ladder diagram (LD), gráfica;
2. Function block diagram (FBD), gráfica;

3. Structured text (ST), textual;

4. Instruction list (IL), textual.

No entanto, o *ELOP II Factory* apenas permite a programação gráfica utilizando linguagem FBD (ver figura 3.26), e não permite qualquer integração de programas LD, ST ou IL.

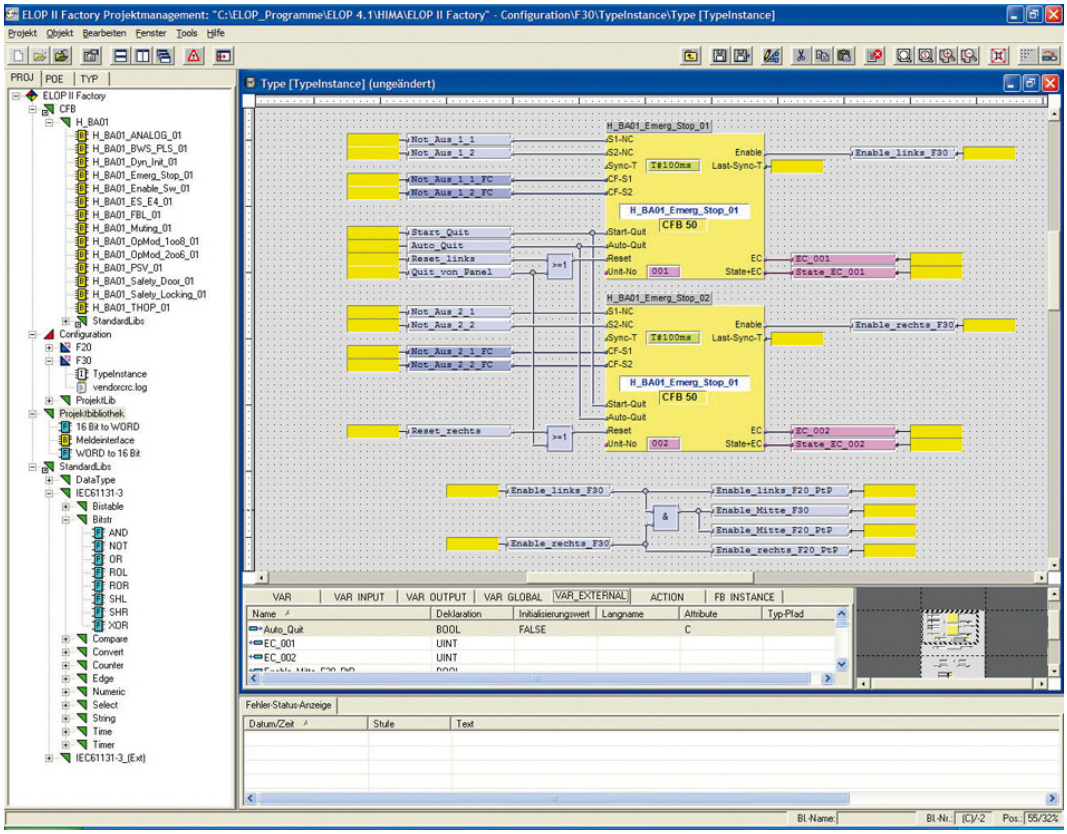


Figura 3.26: Ambiente de programação do *ELOP II Factory*

Consequência A primeira ideia, com a impossibilidade da utilização do *SCADE KCG*, foi criar um compilador com a capacidade de, a partir de um modelo Lustre - todos os modelos Scade possuem um modelo Lustre - gerar o mesmo modelo numa das linguagens textuais da norma IEC 61131-3. O objectivo era traduzir um modelo Scade num modelo *ELOP II Factory* para automatizar o processo de codificação. Dessa forma não se resolvia a questão relativa à certificação do processo, mas seria um passo importante nesse caminho. Evitava-se, também, a codificação manual e as falhas inerentes a qualquer processo manual. No entanto, a limitação do *ELOP II Factory* na importação de modelos, noutras linguagens da norma IEC 61131-3 que não a FBD, não permitiu resolver o problema da codificação do software.

O *ELOP II Factory* possui, também, uma ferramenta integrada para gerar documentação relativamente ao software programado e à configuração do controlador que vai executar o software. A documentação do programa é particularmente interessante quando se pretende manter uma relação documental entre a especificação e a concepção do software.

ELOP II Factory na norma EN 50128 No ciclo de desenvolvimento especificado na norma EN 50128, o *ELOP II Factory* é responsável pela codificação do software. A codificação, por ser um processo manual, é uma actividade de risco. No fim desta actividade é necessário garantir que o software concebido corresponde à especificação efectuada na fase anterior. A documentação que a ferramenta permite gerar de forma automática, permite estabelecer uma relação documental com a especificação *SCADE*. Esta relação, entre o documento de especificação gerado automaticamente pelo *SCADE* e o documento de codificação gerado automaticamente pelo *ELOP II Factory*, constitui um elo de ligação que permite obedecer à norma EN 50128 relativamente à rastreabilidade no ciclo de vida por ela definido.

3.4 Simulator 1.0

Num processo de codificação manual é preciso garantir que o que resulta desse processo tem o mesmo comportamento que a especificação. O *Simulator 1.0* é uma solução interna que constitui, não só o elo de ligação entre a especificação e a concepção, mas também uma ferramenta de análise e apoio à verificação de que a concepção se comporta como a especificação. Por outras palavras, o que se pretende com o *Simulator 1.0*, é fortalecer a relação de confiança que temos na existência de uma equivalência entre a concepção e a especificação com o auxilio de métodos que são recomendados pelas normas e pelos processos de certificação. Em traços gerais, o funcionamento do *Simulator 1.0* baseia-se na captura de um cenário de teste proveniente do *SCADE*, envio dos inputs para o controlador electrónico que os processa e devolve os correspondentes outputs e comparação do resultado obtido da execução com o resultado esperado pela simulação *SCADE*. Veremos com mais detalhe o funcionamento desta ferramenta na secção 4.4, página 66.

Simulator 1.0 na norma EN 50128 No ciclo de desenvolvimento proposto, a codificação do software é manual. Esta concepção tem como base uma especificação formal, analisada e provada, mas uma intervenção humana é considerada um factor de risco. Assim, é necessária uma actividade de verificação para garantir que a intervenção não introduziu falhas no sistema. O *Simulator 1.0* para além de providenciar tal verificação, com base em testes automáticos, apresenta, também uma funcionalidade que permite gerar, automaticamente, o relatório dos testes efectuados. No ciclo de desenvolvimento definido pela norma EN 50128 (ver figura 2.10) o *Simulator 1.0* actua no eixo ascendente.

Capítulo 4

Ciclo de Desenvolvimento

O desenvolvimento de sistemas ferroviários rege-se pela norma EN 50128 (2.6.2). O objectivo desta norma é fazer com que a equipa responsável pelo desenvolvimento do software tenha uma abordagem racional, metódica e rigorosa, que evite ao máximo as falhas inerentes a qualquer processo humano. Esta, apenas recomenda uma metodologia com base em fases de vida do software, não impondo qualquer ferramenta ou técnica. Assim, o que se pretende mostrar neste capítulo é a definição de uma metodologia de desenvolvimento de software para o sector ferroviário, de acordo com as recomendações definidas pela norma EN 50128, com base nas ferramentas apresentadas no capítulo anterior (3). Por outras palavras, propõe-se um ciclo de desenvolvimento, mas sobretudo uma *toolchain* que acompanha este mesmo ciclo.

O ciclo de desenvolvimento de software proposto engloba fundamentalmente quatro fases:

1. Especificação de requisitos do sistema e do software no *DOORS*;
2. Modelação formal do software em *SCADE Suite 6.0*:
   - Especificação formal;
   - Simulação;
   - Verificação formal;
   - Análise de cobertura do modelo;
3. Concepção do software usando o *ELOP II Factory*;
4. Testes à concepção com o *Simulator 1.0*.

4.1 Especificação de Requisitos

A especificação dos requisitos engloba os requisitos do sistema e requisitos do software. Recolher requisitos é uma tarefa de análise e levantamento que pode ser levada a cabo mediante entrevistas, análise de documentação, verificação de sistemas já existentes, etc. A origem de todo o processo de desenvolvimento tem como base este levantamento, pelo que

não é aceitável que um requisito tenha mais do que uma interpretação. Deve ser removida qualquer ambiguidade. Os requisitos, devem descrever, de forma clara, detalhada e precisa, todas as funcionalidades previstas e as propriedades de segurança que estas devem satisfazer. O resultado deste levantamento é um documento que deve corresponder a um módulo formal no *DOORS*. No documento introduzido no *DOORS*, cada funcionalidade ou requisito deve dar origem a um, e apenas um, identificador. Cada identificador no *DOORS* deve identificar um, e apenas um, requisito ou funcionalidade do sistema.

4.2 Modelação Formal do software

A modelação do software será feita no *SCADE Suite 6.0* e deve responder à especificação dos requisitos do software. Mediante a utilização do *SCADE RMG* ou da interface com o *DOORS* do *SCADE*, nesta fase deve-se dar início a uma metodologia que permitirá a rastreabilidade do processo. Este passo do ciclo encontra-se sub-dividido em quatro fases que se aplicam pela ordem em que são apresentadas a seguir.

4.2.1 Especificação formal

A actividade de especificação formal é feita recorrendo ao *SCADE Editor*. O utilizador pode combinar as técnicas de especificação - *data-flow* ou *control-flow* consoante o módulo em desenvolvimento. Nesta fase do ciclo o utilizador terá a responsabilidade de desenvolver a arquitectura do sistema. É uma tarefa importante tendo em conta que o desenvolvimento é efectuado numa técnica de *Top-Down*. Ou seja, o utilizador parte de uma abstracção do sistema e, de forma iterativa, vai refinando-o até obter o nível de especificidade pretendida. Deve existir, desde o início, uma abordagem modular do sistema. Esta abordagem, para além da vantagem imediata que é a reutilização, privilegia também, o encapsulamento. Esta característica permite a melhor leitura e compreensão do sistema ao mesmo tempo que permite uma melhor estrutura do projecto. Antes de avançar para o passo seguinte, o utilizador aplica o *SCADE Model Checker* para efectuar uma verificação semântica do modelo. Também ela, pode ser executada por módulos.

A especificação formal do sistema deve ser desenvolvida recorrendo apenas a operadores de lógica *booleana*, operadores matemáticos simples, aos operadores mais simples relativos a circuitos digitais e máquinas de estados simples - sem encapsulamento de máquinas. Esta limitação na utilização do *SCADE Editor* tem em conta o passo da codificação do software. Como vimos no parágrafo 3.3 na página 60, a codificação é um processo manual que será efectuado no *ELOP II Factory*. Esta ferramenta, não permite a mesma flexibilidade do *SCADE* e é necessário simplificar ao máximo a tradução da especificação, com o objectivo de evitar potenciais falhas na codificação.

4.2.2 Simulação

Um processo de simulação é executado recorrendo ao *SCADE Simulator*. Obedecendo a um desenvolvimento modular do sistema, o utilizador pode também lançar processos de simulação de forma modular. Após a definição de um módulo é possível simula-lo e verifi-

car, de imediato, que a especificação corresponde ao esperado. A simulação não garante a correcção da especificação mas permite, em primeira mão, perceber se esta vai de encontro aos requisitos do sistema. A correcção do módulo é verificada no passo seguinte.

4.2.3 Verificação formal

Tal como a simulação, a verificação formal pode, também, ser executada de forma modular. Para efectuar a verificação formal de determinado módulo da especificação, o utilizador deve definir as propriedades que pretende verificar no sistema - *Property nodes*. Depois de definidas as propriedades, deve então, definir o *observer* que define o contexto das propriedades a verificar. O engenheiro do software deve, também na verificação formal, definir uma estrutura de actividades de verificação, de tal modo que seja possível executar tarefas de verificação formal por grupos ou tipos de propriedades. A verificação formal é tarefa executada pelo *SCADE Design Verifier*.

4.2.4 Análise de cobertura do modelo

Ainda na fase de modelação formal do software, o utilizador deve proceder à análise de cobertura do modelo. Para tal actividade, deve ser utilizada o *SCADE MTC*. Com base nesta análise, o utilizador poderá detectar partes do modelo que nunca são activadas e mesmo determinar partes do modelo para as quais não existe qualquer requisito. Pretende-se nesta fase, garantir que todos os requisitos do sistema foram activados, durante um processo de simulação, e que este se comporta de acordo com os requisitos.

4.3 Concepção do Software

Nesta fase do ciclo de desenvolvimento não podemos usufruir da ferramenta *SCADE KCG*. A não utilização do *SCADE KCG* está relacionado com os controladores de lógica programável utilizados na EFACEC para implementação dos seus sistemas. São controladores certificados em SIL4 e não permitem embeber software que não seja programado com recurso à ferramenta *ELOP II Factory*- consultar 3.3 na página 58. Assim, no ciclo de desenvolvimento proposto, a actividade de codificação do software é uma tarefa manual usando o *ELOP II Factory*. Para que o programador dê início a esta fase, deverá ter em seu poder, um documento da fase anterior - gerado automaticamente pelo *SCADE*- que lhe providencie a arquitectura, estruturas de dados, módulos e funções a implementar. Desta forma, o programador deve apenas efectuar uma tradução da especificação *SCADE*- verificada formalmente - para um programa *ELOP II Factory*. A tradução será um processo já simplificado pelas restrições na utilização de certos operadores *SCADE*- consultar secção 4.2.1 na página 64. Após a codificação do software o programador deve criar um documento - automático no *ELOP II Factory*- que permita estabelecer a relação entre todas as construções do *ELOP II Factory* e as construções do *SCADE*.

4.4 Testes à Concepção

O processo manual, do qual a codificação é alvo, introduz um factor de risco que precisa ser controlado e minimizado. É preciso garantir que o que resulta desse processo tem o mesmo comportamento que a especificação. Este é o passo que falta para completar o processo de desenvolvimento definido neste documento, e é necessário apresentar uma solução para colmatar esta necessidade. Esta secção trata de verificação e não de validação do software, ou seja, o que se pretende é garantir que a concepção respeita a especificação e não garantir que a concepção respeita os requisitos do sistema.

Testes usando o Simulator 1.0 Na tentativa de detectar erros introduzidos na fase de codificação foi desenvolvido o *Simulator 1.0*. O *Simulator 1.0* é, portanto, uma solução interna que constitui, não só o elo de ligação entre a especificação e a concepção, mas também uma ferramenta de análise e apoio à verificação de que a concepção se comporta como a especificação. Por outras palavras, a responsabilidade do *Simulator 1.0*, no ciclo de desenvolvimento, é reforçar a confiança que temos na equivalência entre a concepção e a especificação. Se tal relação existir, então pelo menos para os cenários testados, a concepção é igual à especificação. O funcionamento do *Simulator 1.0* baseia-se na captura de um cenário de teste proveniente do *SCADE*, envio dos inputs para o controlador electrónico que os processa e devolve os correspondentes outputs e comparação do resultado obtido da execução com o resultado esperado pela simulação *SCADE*.

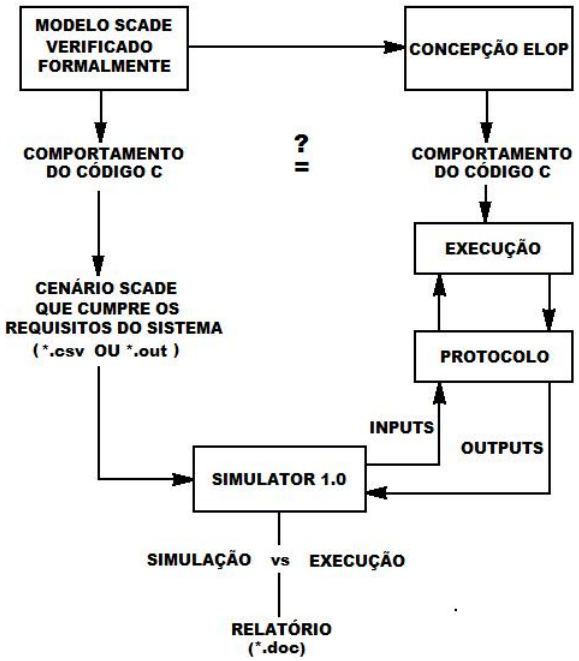


Figura 4.1: Esquema de utilização do *Simulator 1.0*

Para reforçar, através do *Simulator 1.0*, que determinada propriedade permanece na

concepção, o utilizador deve exportar - em formato `*.out` ou `*.csv` - uma actividade de simulação, do *SCADE*, que reflicta a referida propriedade. A simulação efectuada é interpretada pelo *Simulator 1.0* que, posteriormente, envia os inputs, dos vários ciclos que compõem o cenário, ao controlador que os executará. Após a execução de cada ciclo, o controlador retorna uma mensagem com os outputs correspondentes. O *Simulator 1.0* será, então, capaz de comparar o resultado da execução com o resultado da simulação. Sendo a simulação resultado de uma especificação formal, verificada e provada formalmente, se a comparação der um resultado de comportamento idêntico, então, naquele cenário, a concepção corresponde aos requisitos do sistema. Para terminar o ciclo o *Simulator 1.0* está capacitado com um gerador automático do relatório do teste efectuado.

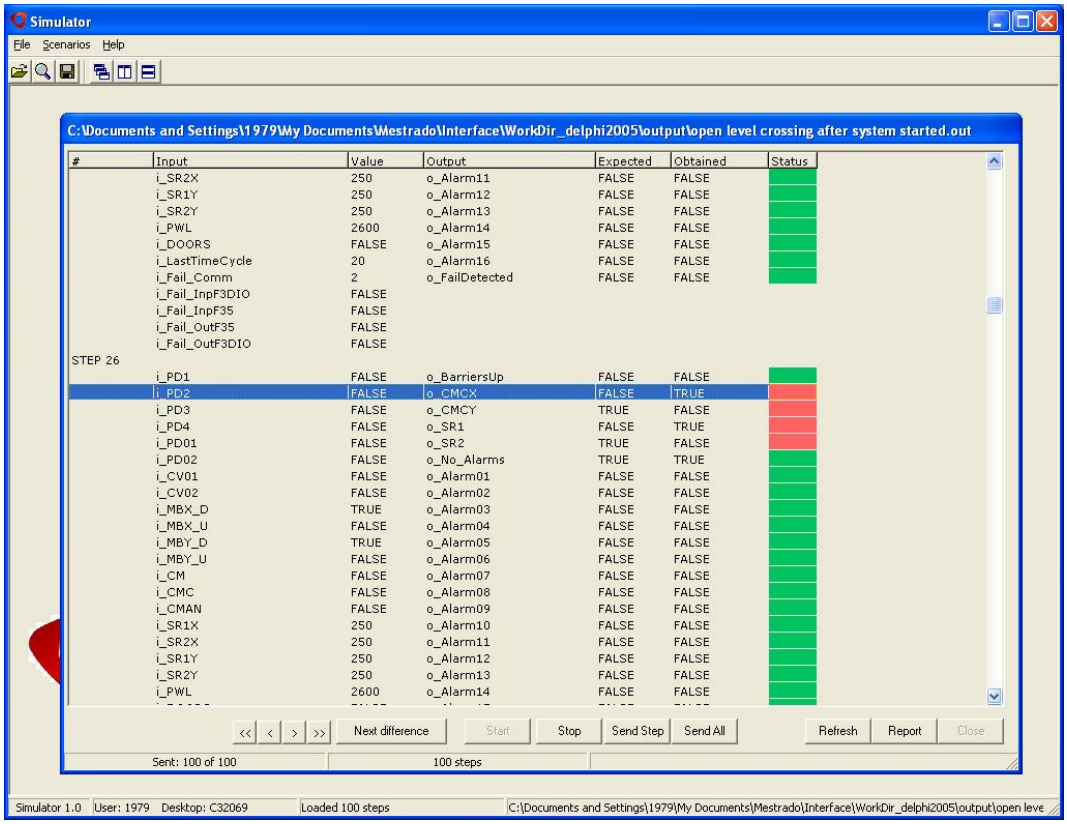


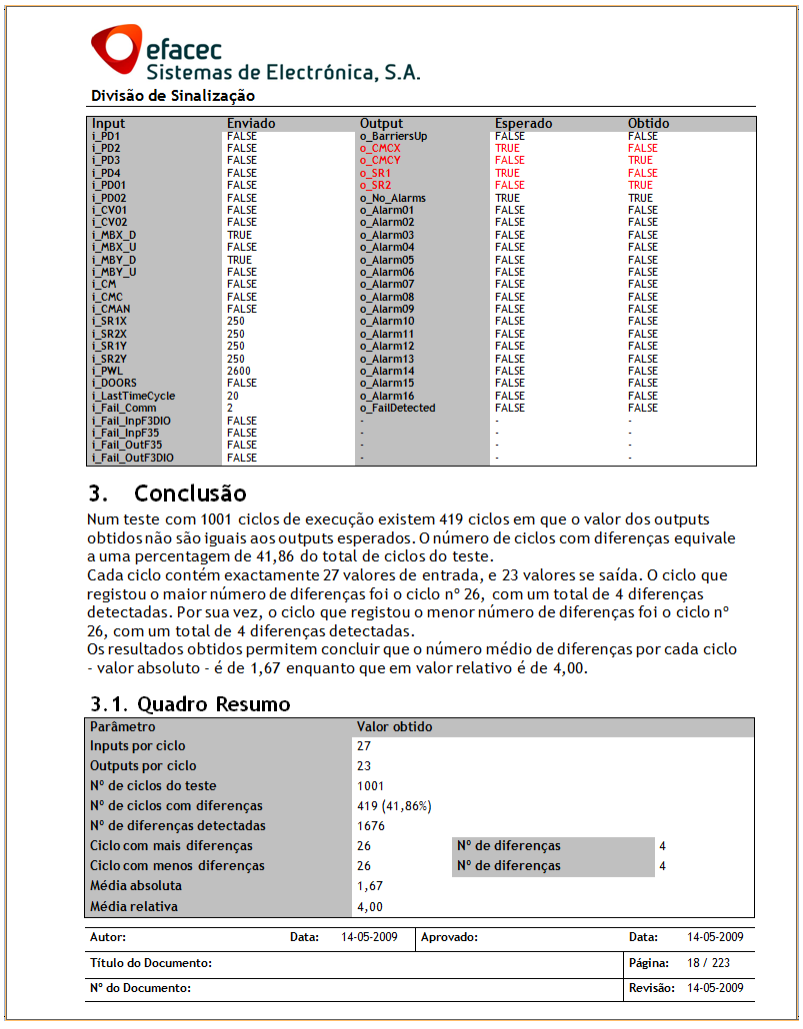
Figura 4.2: Análise gráfica do *Simulator 1.0*

O *Simulator 1.0*, depois de importar uma simulação *SCADE*- nos formatos `*.out` ou `*.csv` - permite enviar, através de mensagens sincronizadas, os inputs da simulação para serem executados no controlador de lógica programável. Após cada ciclo de execução é efectuado o processo inverso, ou seja, o controlador envia uma mensagem de resposta ao aplicativo com o resultado da execução. Após receber essa mensagem, o *Simulator 1.0* compara a resposta obtida com resultado esperado do mesmo ciclo. O processo de troca de mensagens repete-se sucessivamente até ser totalizado o número total de ciclos do cenário

simulado em *SCADE*.

Com o *Simulator 1.0* é possível, depois, efectuar uma análise detalhada de cada ciclo (ver figura 4.2), e detectar diferenças entre o esperado e o obtido. Assim, é possível verificar a correspondência de comportamento entre o especificado e simulado no *SCADE* e a concepção efectuada no *ELOP II Factory*.

É possível, no final, gerar de forma completamente automática um relatório que indica os ciclos onde foram detectadas diferenças e apresenta alguns dados estatísticos da análise efectuada (ver figura 4.3).



efacec
Sistemas de Electrónica, S.A.
Divisão de Sinalização

Input	Enviado	Output	Esperado	Obtido
i_PD1	FALSE	o_BarriersUp	FALSE	FALSE
i_PD2	FALSE	o_CMCX	TRUE	FALSE
i_PD3	FALSE	o_CMCY	FALSE	TRUE
i_PD4	FALSE	o_SR1	TRUE	FALSE
i_PD01	FALSE	o_SR2	FALSE	TRUE
i_PD02	FALSE	o_No_Alarms	TRUE	TRUE
i_CV01	FALSE	o_Alarm01	FALSE	FALSE
i_CV02	FALSE	o_Alarm02	FALSE	FALSE
i_MBX_D	TRUE	o_Alarm03	FALSE	FALSE
i_MBX_U	FALSE	o_Alarm04	FALSE	FALSE
i_MBY_D	TRUE	o_Alarm05	FALSE	FALSE
i_MBY_U	FALSE	o_Alarm06	FALSE	FALSE
i_CM	FALSE	o_Alarm07	FALSE	FALSE
i_CMC	FALSE	o_Alarm08	FALSE	FALSE
i_CMAN	FALSE	o_Alarm09	FALSE	FALSE
i_SR1X	250	o_Alarm10	FALSE	FALSE
i_SR2X	250	o_Alarm11	FALSE	FALSE
i_SR1Y	250	o_Alarm12	FALSE	FALSE
i_SR2Y	250	o_Alarm13	FALSE	FALSE
i_PWL	2600	o_Alarm14	FALSE	FALSE
i_DOORS	FALSE	o_Alarm15	FALSE	FALSE
i_LastTimeCycle	20	o_Alarm16	FALSE	FALSE
i_Fail_Comm	2	o_FailDetected	FALSE	FALSE
i_Fail_InpF3DIO	FALSE	-	-	-
i_Fail_InpF35	FALSE	-	-	-
i_Fail_OutF35	FALSE	-	-	-
i_Fail_OutF3DIO	FALSE	-	-	-

3. Conclusão

Num teste com 1001 ciclos de execução existem 419 ciclos em que o valor dos outputs obtidos não são iguais aos outputs esperados. O número de ciclos com diferenças equivale a uma percentagem de 41,86 do total de ciclos do teste.
Cada ciclo contém exactamente 27 valores de entrada, e 23 valores se saída. O ciclo que registou o maior número de diferenças foi o ciclo nº 26, com um total de 4 diferenças detectadas. Por sua vez, o ciclo que registou o menor número de diferenças foi o ciclo nº 26, com um total de 4 diferenças detectadas.
Os resultados obtidos permitem concluir que o número médio de diferenças por cada ciclo - valor absoluto - é de 1,67 enquanto que em valor relativo é de 4,00.

3.1. Quadro Resumo

Parâmetro	Valor obtido		
Inputs por ciclo	27		
Outputs por ciclo	23		
Nº de ciclos do teste	1001		
Nº de ciclos com diferenças	419 (41,86%)		
Nº de diferenças detectadas	1676		
Ciclo com mais diferenças	26	Nº de diferenças	4
Ciclo com menos diferenças	26	Nº de diferenças	4
Média absoluta	1,67		
Média relativa	4,00		

Autor:	Data: 14-05-2009	Aprovado:	Data: 14-05-2009
Título do Documento:			Página: 18 / 223
Nº do Documento:			Revisão: 14-05-2009

Figura 4.3: Relatório gerado pelo *Simulator 1.0*

4.5 Conclusão

A metodologia e ferramentas apresentadas para o desenvolvimento formal de software para sistemas de sinalização, estão em conformidade com a norma EN 50128.

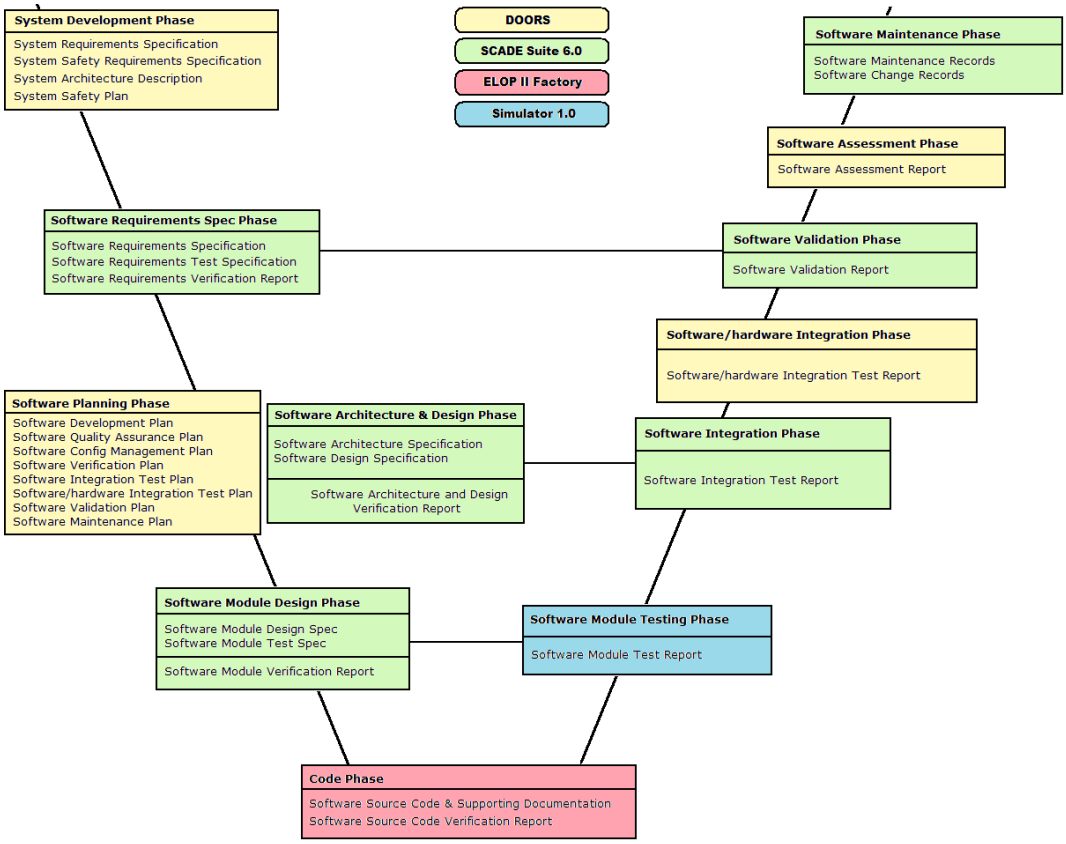


Figura 4.4: As ferramentas no ciclo de desenvolvimento definido na norma EN 50128

A figura 4.4 representa a aplicação das ferramentas descritas no capítulo 3 no ciclo de desenvolvimento definido pela norma EN 50128. Pode verificar-se, que o conjunto das ferramentas utilizadas neste trabalho cobrem, por completo, o ciclo de desenvolvimento definido na norma EN50128. Para perceber o enquadramento do ciclo de desenvolvimento proposto é dada a figura 4.5.

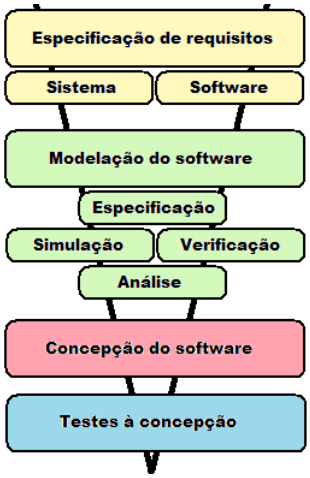


Figura 4.5: As ferramentas no ciclo de desenvolvimento definido neste trabalho

Capítulo 5

Aplicação aos Sistemas Ferroviários

O objectivo principal deste trabalho é o desenvolvimento formal de um sistema de sinalização ferroviária de acordo com o normativo CENELEC usando o *SCADE Suite 6.0*. Neste capítulo são apresentados dois casos de aplicação da *toolchain* descrita no capítulo 4, nomeadamente numa API para sistemas de sinalização e numa Passagem de Nível (PN).

5.1 API para Sistema de Sinalização

Pretende-se, nesta secção, aplicar o ciclo de desenvolvimento, proposto no capítulo anterior, no desenvolvimento de uma API para um sistema de sinalização.

Um sistema de sinalização tem como objectivo implementar a gestão da circulação dos comboios de uma forma segura. Para isso, controla a disponibilidade dos vários elementos que compõem a linha. Uma linha ferroviária simples é composta, pelo menos, com os seguintes elementos:

1. **Sinais luminosos:** são dispositivos luminosos que apresentam dois, ou mais, aspectos. Assumindo um sinal básico com apenas dois aspectos, um indica a permissão de avanço do veículo e o outro, impede o seu avanço;

2. **Secções de via:** são troços de via delimitados por dois dispositivos luminosos;

3. **Agulhas:** são dispositivos que permitem mudar um veículo de direcção.

A movimentação de veículos na linha é feita estabelecendo-se rotas. Uma rota é o trajecto entre uma estação e a que se lhe segue. Para mover um veículo de uma estação de origem para uma estação de destino é necessário estabelecer um itinerário. Um itinerário é o trajecto de um veículo de uma estação de origem até uma estação de destino. Um itinerário é constituído por rotas.

Para que a deslocação seja feita em total segurança, o sistema deve controlar todos os elementos em consonância com a deslocação e localização de outros veículos. O sistema, não deve permitir a ocupação de uma secção de via por mais do que um comboio, evitando assim choques frontais e choques em cadeia. O movimento não convencional na linha ferroviária é designado um movimento em modo degradado. Estes, são movimentos especiais

na linha que apenas são efectuados em situação de emergência ou situações de manutenção na via.

A decomposição de sistema de sinalização em sub-sistemas mais elementares permite abordar a modelação numa perspectiva geográfica. Esta abordagem, evita que o sistema a sinalização tenha como base uma tabela de controlo e que todas as decisões sejam tomadas segundo essa tabela. O modelo geográfico toma as decisões sobre a linha através de troca de mensagens entre os objectos. Estas, são basicamente mensagens de pergunta e resposta que cada elemento da linha troca com os seus elementos vizinhos. A linha utilizada como caso de estudo é dada pela figura 5.1.

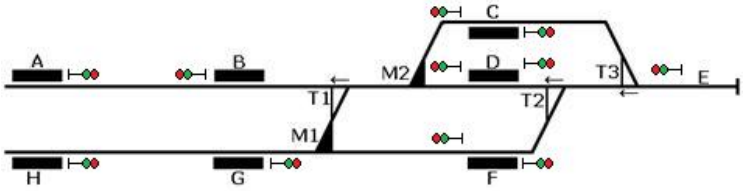


Figura 5.1: Linha ferroviária utilizada como caso de estudo

As figuras 5.2 e 5.3 representam o esquema da linha recorrendo a uma modelação em termos de grafos. A representação da linha neste formato permite formalizar as propriedades da linha e mais facilmente estudar essas propriedades.

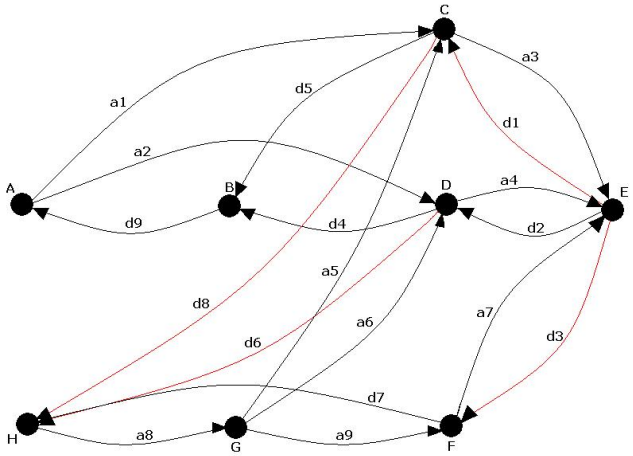


Figura 5.2: Representação em grafo de todos as rotas possíveis na linha em estudo

Nas secções seguintes vamos abordar apenas alguns elementos que compõem uma linha ferroviária simples. Vamos exemplificar a aplicação do método na especificação de uma agulha 5.1.1, de um sinal 5.1.2 e de uma secção de via 5.1.3. As propriedades verificadas em cada uma das secções referem-se a propriedades dos elementos e não as propriedades do sistema de sinalização. O objectivo é desenvolver uma API de objectos especificados

X = Matriz de adjacencia

	A	B	C	D	E	F	G	H
A	0	0	1	1	0	0	0	0
B	1	0	0	0	0	0	0	0
C	0	1	0	0	1	0	0	1
D	0	1	0	0	1	0	0	1
E	0	0	1	1	0	1	0	0
F	0	0	0	0	1	0	0	1
G	0	0	1	1	0	1	0	0
H	0	0	0	0	0	0	1	0

modo degradado

Nº de caminhos de comprimento 2

	A	B	C	D	E	F	G	H
A	0	2	0	0	2	0	0	2
B	0	0	1	1	0	0	0	0
C	1	0	1	1	0	1	1	0
D	1	0	1	1	0	1	1	0
E	0	2	0	0	3	0	0	3
F	0	0	1	1	0	1	1	0
G	0	2	0	0	3	0	0	3
H	0	0	1	1	0	1	0	0

Nº de caminho de comprimento 3

	A	B	C	D	E	F	G	H
A	2	0	2	2	0	2	2	0
B	0	2	0	0	2	0	0	2
C	0	2	2	2	3	1	0	3
D	0	2	2	2	3	1	0	3
E	2	0	3	3	0	3	3	0
F	0	2	1	1	3	1	0	3
G	2	0	3	3	0	3	3	0
H	0	2	0	0	3	0	0	3

Nº de caminhos de comprimento 4

	A	B	C	D	E	F	G	H
A	0	4	4	4	6	2	0	6
B	2	0	2	2	0	2	2	0
C	2	4	3	3	5	3	3	5
D	2	4	3	3	5	3	3	5
E	0	6	5	5	9	3	0	9
F	2	2	3	3	3	3	3	3
G	0	6	5	5	9	3	0	9
H	2	0	3	3	0	3	3	0

Figura 5.3: Matrizes de adjacência

e verificados formalmente para facilitar a especificação de sistemas de sinalização mais complexos.

5.1.1 Agulha

No transporte ferroviário existem, por necessidades de logística e operacionais, cruzamentos e convergência de linhas férreas. As operações de mudança de linha são providenciadas por dispositivos electromecânicos capazes de orientar um veículo. Esses dispositivos designam-se por agulhas. Uma agulha ferroviária assume apenas dois estados possíveis: direita ou esquerda. A agulha na posição direita provoca o movimento do comboio para a esquerda, a agulha na posição esquerda movimenta o comboio para a posição mais à direita (ver figura 5.4).

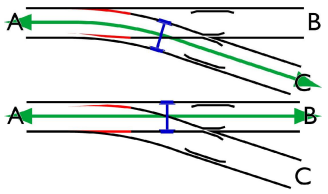


Figura 5.4: Esquema de movimentação das agulhas ferroviárias

Uma agulha quando está numa das duas posições possíveis - direita ou esquerda -, possui, ainda, uma outra propriedade que indica se esta se encontra bloqueada ou desbloqueada. Uma agulha encontra-se bloqueada se o sistema de sinalização estabeleceu uma rota da qual esta agulha faz parte. Ao estabelecer uma rota o sistema, controla e bloqueia todos os seu componentes de forma a que não seja possível alterar o estados dos componentes até a

rota estar concluída. A figura 5.5 apresenta a especificação *SCADE* que representa, numa máquinas de estados, o objecto agulha.

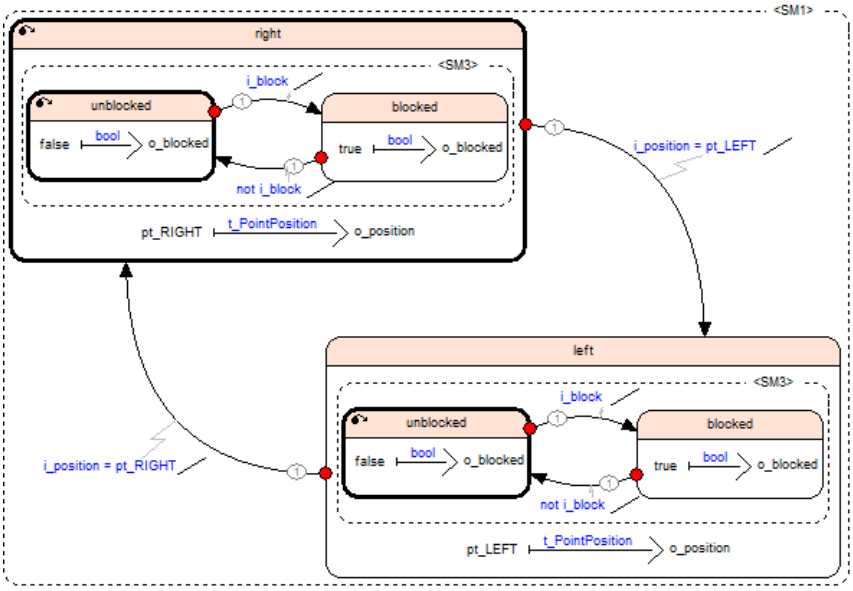


Figura 5.5: Máquina de estados de uma agulha ferroviária.

A figura 5.6 representa um *property node* que verifica que no modelo especificado a agulha nunca se encontra em dois estados distintos num mesmo instante.

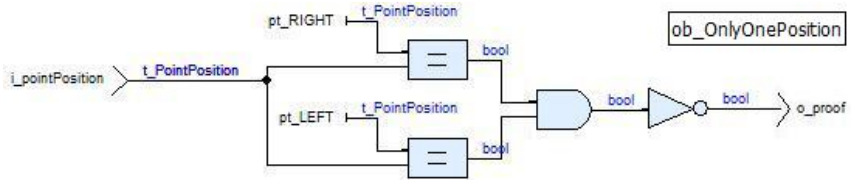


Figura 5.6: *Property node* que exprime uma propriedade de segurança do objecto agulha

Na imagem 5.7 é apresentado o *observer* que verifica formalmente, a propriedade definida em 5.6.

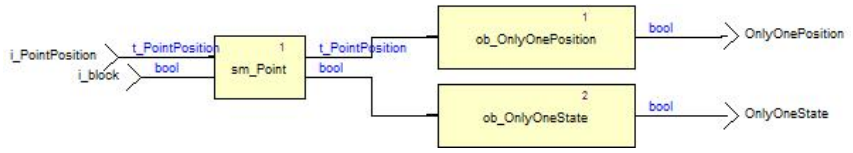


Figura 5.7: *Observer* do objecto agulha

Na figura 5.8 é apresentada a verificação formal aplicada ao sistema onde consta a verificação à propriedade da agulha definida em 5.6.

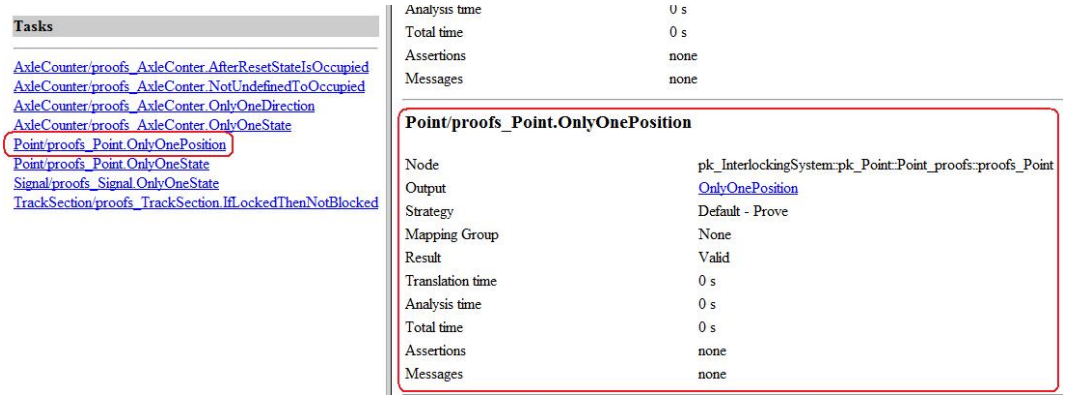


Figura 5.8: Resultado da prova formal usando o *SCADE Design Verifier*

5.1.2 Sinal

Os sinais luminosos, nas linhas ferroviárias, são os dispositivos que indicam o estado de ocupação da secção de via que lhe sucede. Estes são a única interface dinâmica que o operador do veículo possui para determinar o avanço ou não da sua composição. Mediante um estado de restrição - pode ser uma cor vermelha, um traço na horizontal, etc - o operador tem indicações expressas para não avançar o movimento. O movimento só pode ser efectuado mediante um sinal de permissão de avanço - sinal com cor verde, traço vertical, etc. O sinal modelado neste trabalho, reflecte um sinal com três aspectos possíveis: Verde para permitir o avanço, vermelho para impedir e vermelho intermitente para indicar um condição de avanço mas com restrições de velocidade. A modelação *SCADE* que reflecte a alternância entre os estados deste sinal é dado pela figura 5.9.

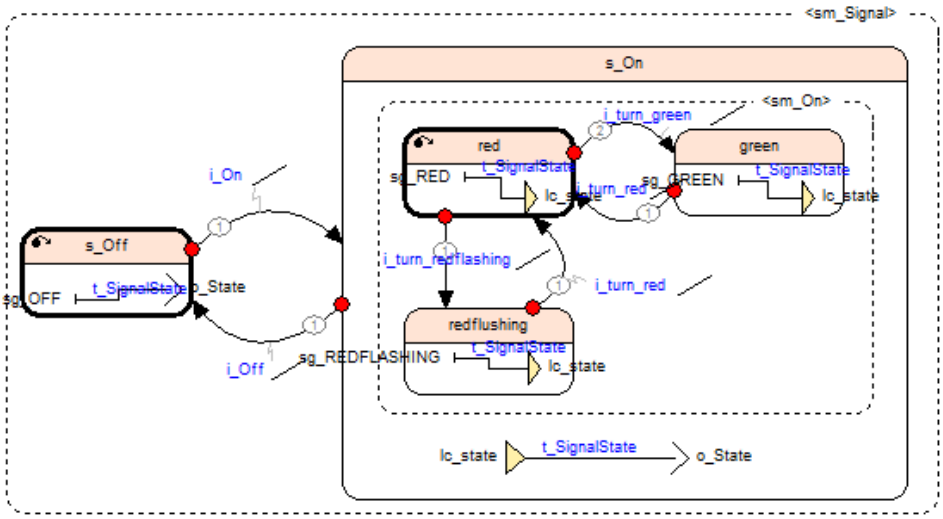


Figura 5.9: Máquina de estados de um sinal luminoso.

A figura 5.10 define o *observer* que verifica a propriedade definida pelo operador.

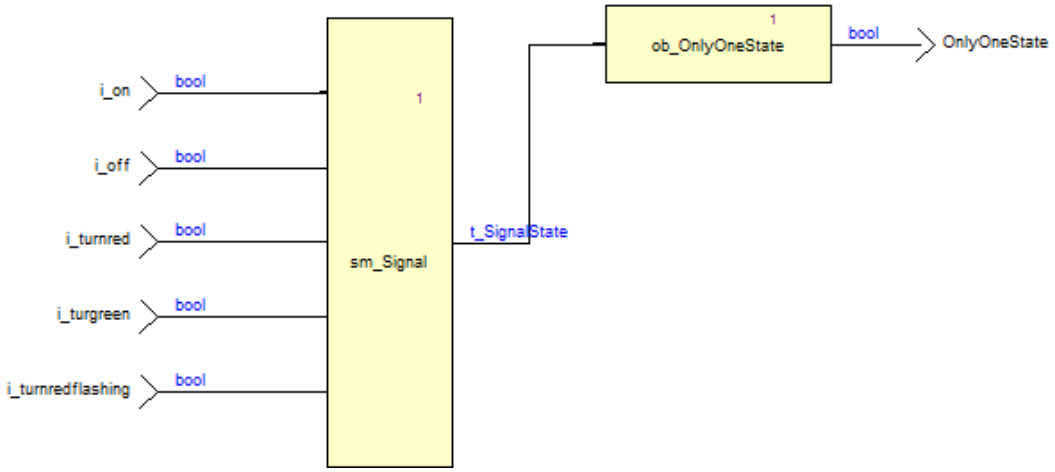


Figura 5.10: *Observer* para verificar as propriedades do sinal

Na figura 5.11 é apresentada a verificação formal aplicada ao sistema onde consta a verificação da propriedade do sinal definido pelo *observer* 5.10.

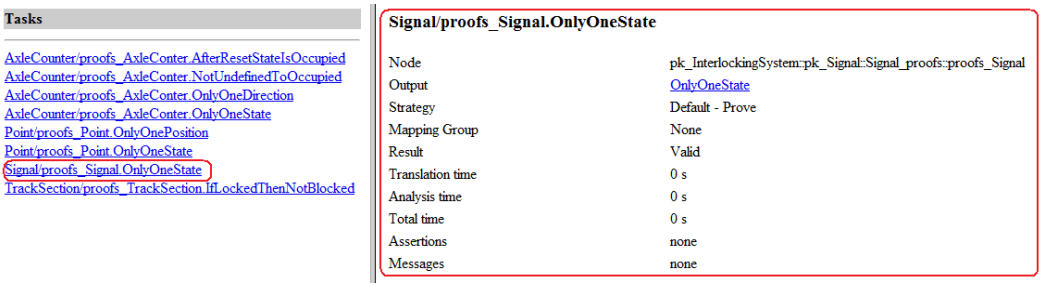


Figura 5.11: Resultado da prova formal usando o *SCADE Design Verifier*

5.1.3 Secção de Via

Uma secção de via é um troço de via delimitado, no nosso caso, por sinais luminosos. O objectivo das secções de via é permitir alocar pedaços da linha para garantir, através do sistema de sinalização, que estes nunca são ocupados por mais do que um veículo e assim evitar choques frontais e/ou choques em cadeia. As secções de via, permitem também, sem linhas mais longas e complexas, a localização de veículos mediante a ocupação destas. Em primeira instância uma secção de via pode estar bloqueada ou não. Uma secção de via pode estar bloqueada quando ocorrem, por exemplo, operações de manutenção nessa secção. Uma secção de via bloqueada não pode, em caso algum, ser utilizada para estabelecer uma

rota. No caso de não estar bloqueada, uma secção de via pode assumir um dos seguintes estados:

- **Bloqueada** - a secção não pode ser requisitada para estabelecer rotas;
- **Disponível** - a secção pode ser utilizada para estabelecer rotas;

  - **Alocada** - a secção está atribuída para uma rota;

    Ocupada - a secção está ocupada por um veículo;

    Livre - a secção está livre de veículos;

    Indefinido - não é possível determinar a ocupação da secção de via.

  - **Livre** - a secção não faz parte de nenhuma rota;

  - **Indefinida** - não é possível determinar o estado da secção.

A figura 5.12 reflecte o comportamento de uma secção de via.

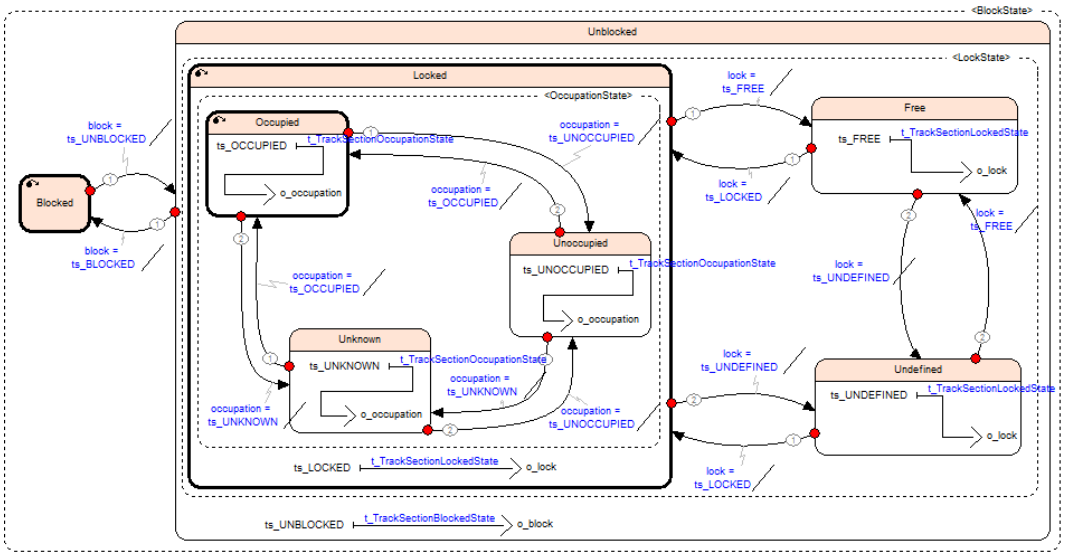


Figura 5.12: Máquina de estados de uma secção de via.

Uma propriedade importante de segurança que é preciso garantir no sistema é a de que se uma secção está num estado de alocada, então não pode, em simultâneo, estar no estado bloqueada. A propriedade alvo de verificação, é expressa em *SCADE* segundo a figura 5.13.

A figura 5.14 mostra o *observer* responsável por contextualizar a propriedade definida em 5.13. Por fim, a actividade de verificação formal que verifica a veracidade da propriedade, é dada pela figura 5.15

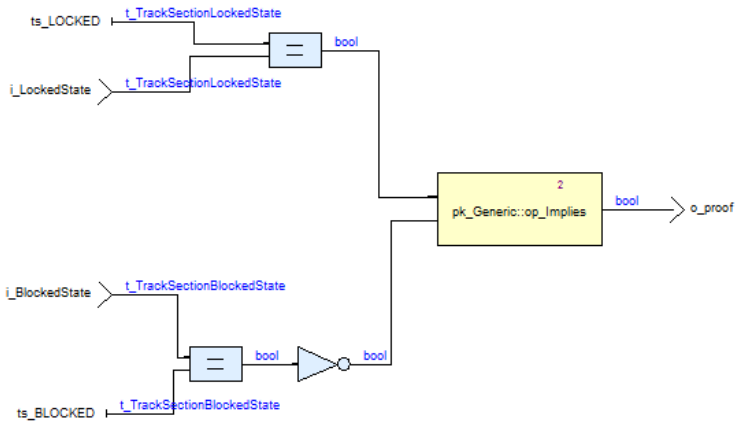


Figura 5.13: *Property node* que exprime a propriedade

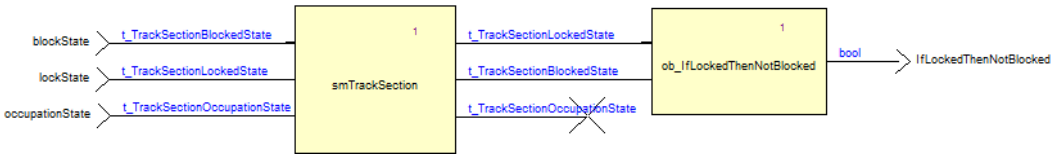


Figura 5.14: *Observer* que verifica a referida propriedade

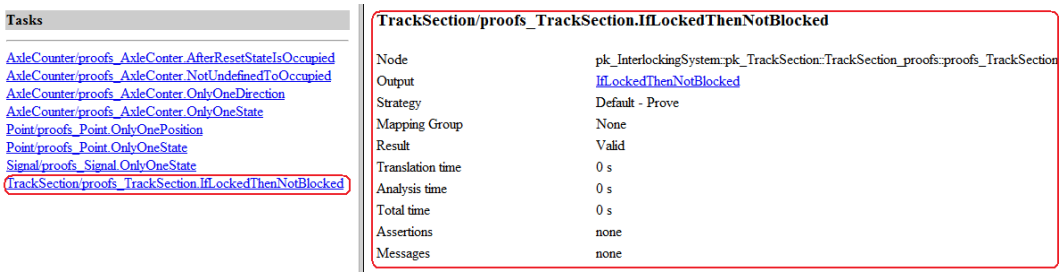


Figura 5.15: Resultado da actividade de verificação formal

5.1.4 Conclusão

A construção da API na sua versão actual ainda merece algum esforço em termos de cobertura dos elementos constituintes dum sistema de sinalização. No entanto, tanto a metodologia utilizada como os elementos já modelados (e as propriedades sobre estes demonstrados) revelou-se suficientemente modular para esperar uma utilização e extensão com facilidade em projectos futuros.

5.2 Passagens de Nível

A especificação de um sistema de PN surgiu como forma de atravessar, transversalmente, a ferramenta *SCADE Suite 6.0*. Uma PN é um cruzamento, ao mesmo nível, de uma ou mais vias de trânsito ferroviário com uma via, principal ou secundária, de trânsito rodoviário. Em linguagem natural, uma especificação geral para um sistema de PN automática é a seguinte:

Devem ser detectados, automaticamente, veículos ferroviários em circulação na linha no sentido da PN. Quando detectado um veículo nessa condição, a PN deve impedir que o transito rodoviário atravesse a linha enquanto este não passar na PN.

5.2.1 Requisitos do sistema

Esta é a primeira fase do ciclo proposto. É necessário proceder à análise e levantamento do sistema pretendido e criar a devida documentação no *DOORS*. Pela tradição da EFACEC no sector, não foi difícil obter ajuda no que toca à junção de informação. Mais difícil foi agrupar toda essa informação de forma clara e suficientemente detalhada com a qual fosse possível avançar para a sua introdução do *DOORS* na forma de um documento de requisitos do sistema. Existiam já alguns documentos descritivos mas não muito detalhados. Eram basicamente documentos de apresentação do produto e, portanto, apenas descreviam os aspectos gerais do funcionamento. Para obter mais detalhes, foi necessário consultar modelos de especificação de circuitos eléctricos. Esta análise permitiu atingir o grau de detalhe pretendido.

A especificação em linguagem natural apresentada em 5.2 apenas permite retirar ideias muito gerais acerca do funcionamento do sistema. Contudo, esta serve como base para um processo de aprofundamento e detalhe no funcionamento do sistema. Esta especificação dá origem a outras questões tais como o "Como?" e "Quando?". De seguida é apresentada um especificação funcional mais detalhada:

O trânsito rodoviário é protegido do trânsito ferroviário com recurso a barreiras, sinais luminosos e sinais sonoros controlados electronicamente e colocados em ambos os lados do tráfego ferroviário. A detecção de comboios é conseguida mediante a utilização de pedais electrónicos direccionais (pedais de anúncio) colocados no ponto de origem do anúncio. Na passagem de um comboio por um pedal de anúncio no sentido da PN, esta, após determinado período de tempo, fecha-se aos utentes. Numa primeira fase entram em funcionamento os sinais rodoviários, (luminosos e sonoros) e, numa segunda fase, o abaixamento das meias-barreiras. Esta situação permanece até que a composição passe pelo pedal electrónico não orientado (pedal de circuito de via) e saia completamente do circuito de via. Após o veículo ter libertado o circuito de via, a PN deve abrir as meias-barreiras, desligar os sinais rodoviários e permitir que o tráfego rodoviário atravesse a linha ferroviária.

Esta especificação já permite obter mais detalhe do funcionamento do sistema. Para que o leitor perceba melhor o funcionamento a cima descrito, é apresentado um esquema 5.16 que representa, fisicamente, uma PN.

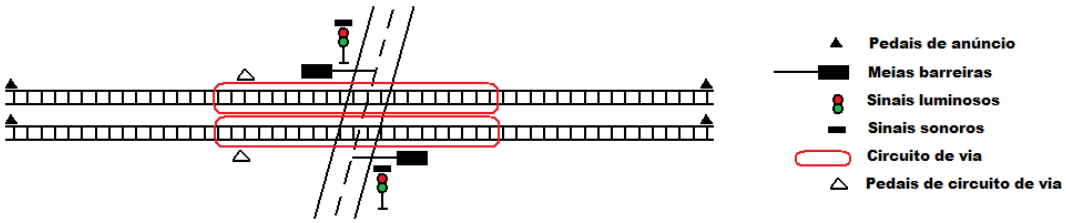


Figura 5.16: Esquema geral de uma PN de via dupla.

Apesar de apresentar já algum detalhe de funcionamento, esta descrição permite apenas a idealização de um sistema demasiado abstracto. Não estão definidas, por exemplo, as restrições temporais entre cada uma das fases da PN e podem ser colocadas as seguintes perguntas acerca do funcionamento:

1. Após detectada uma composição no sentido da PN, qual o período de tempo até dar início ao anúncio ao trânsito rodoviário?

2. Numa primeira fase entram em funcionamento os sinais rodoviários e, numa segunda fase, o abaixamento das barreiras. Qual o período de tempo entre elas?

3. Após a passagem do veículo deve ser aguardado algum período de tempo até dar início ao levantamento das barreiras?

O que se pretende mostrar com isto, é que o processo para obter uma especificação funcional suficientemente detalhada de um sistema, é um processo *Top-Down*. É necessário partir de um nível de abstracção elevado e obter, em passos sucessivos, detalhe do sistema de forma a chegar a um nível tão baixo em que este seja já a especificação pormenorizada do sistema.

Os requisitos do sistema obtidos, após o levantamento e análise efectuada nesta fase, deram origem a um módulo formal no *DOORS* que está representado na figura 5.17. Para integrar no documento de especificação dos requisitos do sistema foi elaborado um diagrama de actividades. A figura 5.18 apresenta o diagrama de actividades do sistema PN.

Para evitar a transcrição do documento obtido, é apresentada uma descrição, em linguagem natural, do funcionamento da PN numa situação normal de detecção de um veículo ferroviário. É considerada uma situação normal de detecção quando um veículo é detectado através de um pedal de anúncio. Existem situações em que a detecção é feita apenas quando o comboio entra no circuito de via, por exemplo, quando ocorre uma falha no pedal. Assim, o funcionamento da PN, no modo mais comum, é dado pela seguinte descrição:

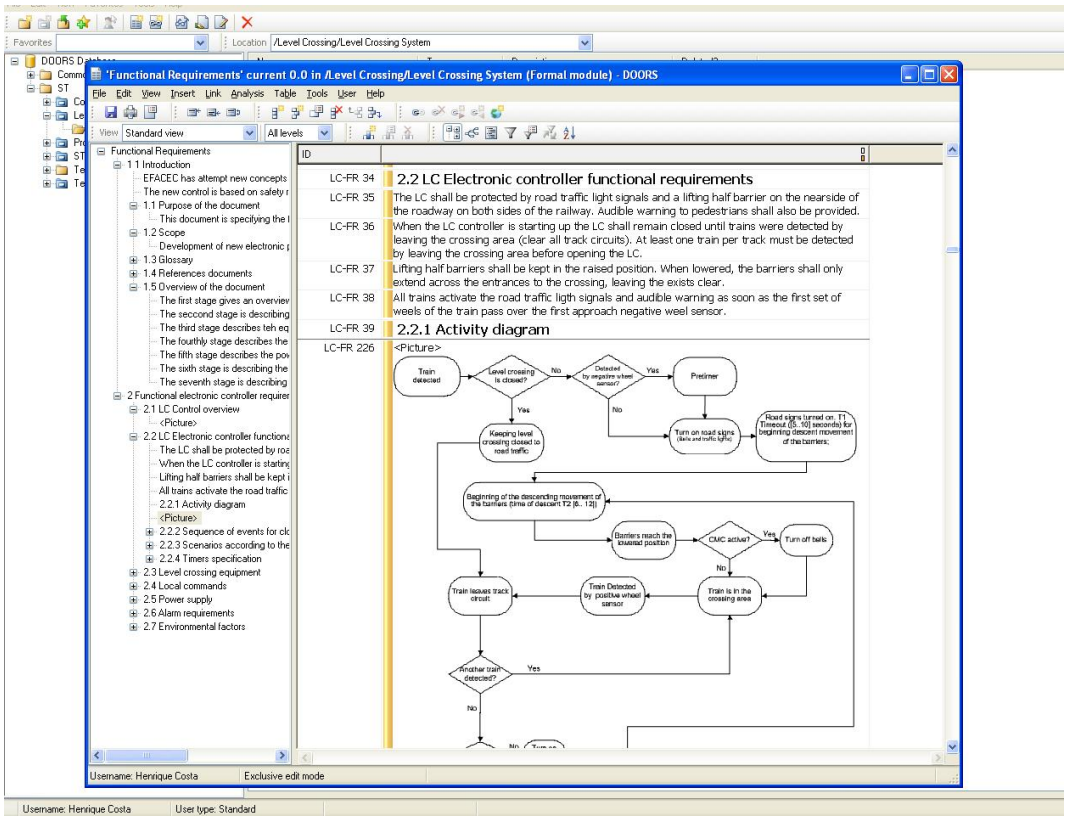


Figura 5.17: Documento de requisitos introduzido no DOORS

1. Após detectado um veiculo ferroviário, por um pedal de anuncio, deve ser iniciada uma contagem de tempo - normalmente 0 segundos - para dar inicio ao sinais luminosos;

2. Os sinais luminosos iniciam o anúncio de aproximação de veiculo ferroviário. O sinal é composto por duas lâmpadas que devem alternar entre elas quando este está em anúncio. Deve ser iniciada uma contagem de tempo - normalmente 7 segundos - para dar início aos sinais sonoros e ao movimento descendente das barreiras;

3. As barreiras descem através de meios mecânicos e demoram entre 6 a 12 segundos. O movimento descendente deve ser acompanhado pelos sinais sonoros e luminosos;

4. O período de tempo entre o inicio de actividade dos sinais luminosos até à chegada do veiculo ferroviário à PN - à velocidade máxima permitida - deve ser suficiente de forma a libertar a zona da PN (normalmente 25 segundos);

5. A detecção de que um veículo passou pela zona da PN é feito através do pedal positivo e do circuito de via;

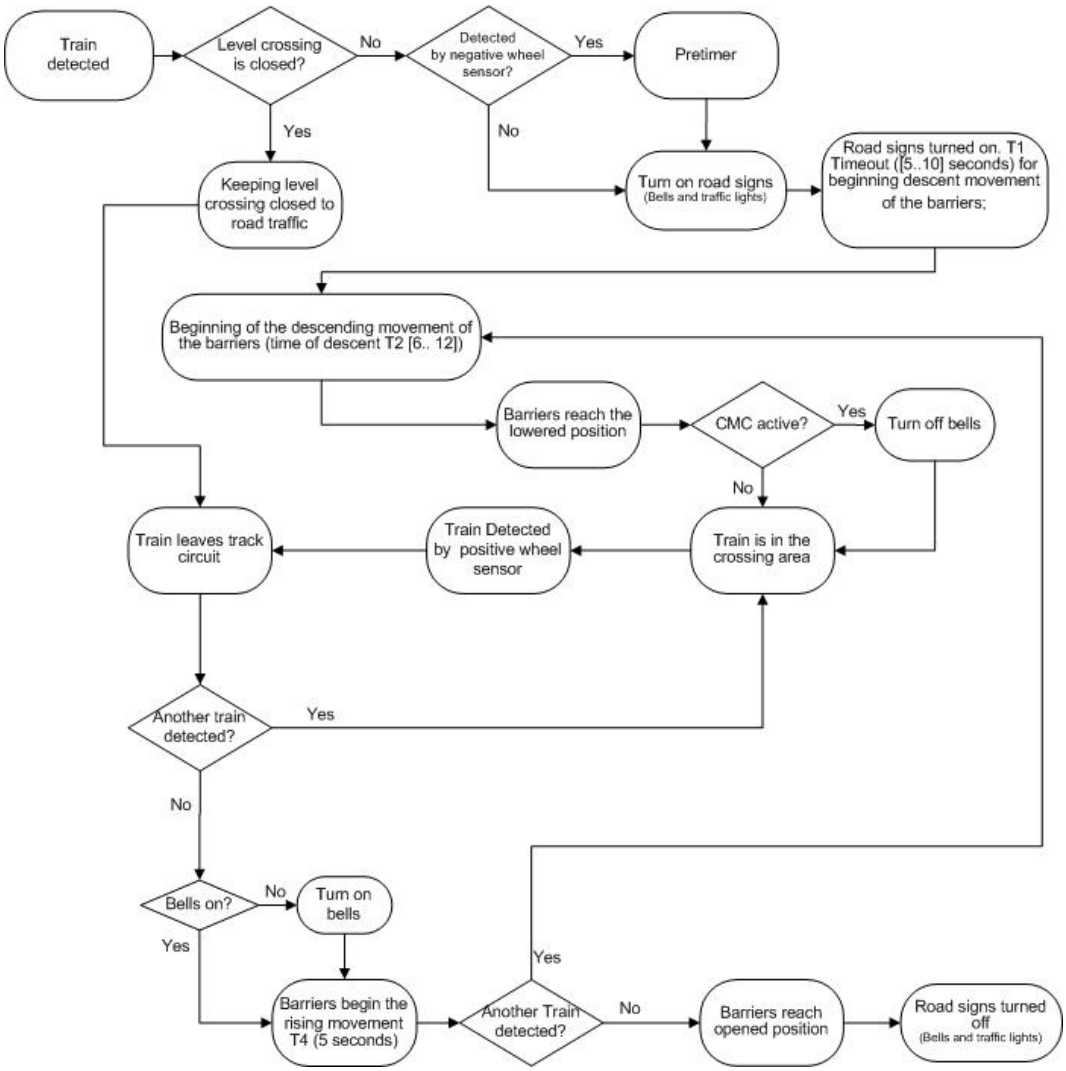


Figura 5.18: Diagrama de actividades numa PN

6. Se não foi detectado outro veículo e foi atingido o tempo mínimo de fecho da PN, então as barreiras iniciam o seu movimento ascendente - processo mecânico que demora cerca de 5 segundos;

7. Se detectado um comboio durante o movimento ascendente das barreiras, então, estas devem descer imediatamente;

8. Quando as barreiras atingem a posição vertical, os sinais sonoros e luminosos são desactivados.

Requisitos Para evidenciar a aplicação do método desenvolvido num sistema da PN são apenas referidos, neste documento, uma pequeníssima parte de todos os requisitos obtidos. Para nos acompanhar em todo este processo de demonstração do método foram seleccionados os seguintes requisitos do sistema:

1. O sistema deve arrancar em estado seguro, ou seja, deve arrancar com a PN em estado fechado;

2. A PN não pode estar em estado fechada e ter as barreiras na posição vertical (permitir que o trânsito rodoviário avance);

3. Se as barreiras não estiverem na posição vertical (abertas), então os sinais luminosos estão em funcionamento;

4. Se um dos pedais negativos for pressionado (detectado um veiculo), então a PN não pode estar aberta nem em processo de abertura;

5.2.2 Modelação SCADE

A modelação da PN em *SCADE* corresponde ao segundo passo do ciclo de desenvolvimento proposto neste documento. Neste ponto do ciclo temos já referências documentais que devem ser seguidas à risca no processo de modelação.

Especificação

O conjunto dos requisitos que foram seleccionados, referem propriedades relativas aos elementos PN, barreiras, e sinais luminosos. Na estrutura do projecto foram definidos os tipos enumerados apresentados na figura 5.19.

Type	Definition	Comments
t_BarriersState	<enumeration>	Enumeration to provide states for barriers
br_UP		Barriers are in up position
br_DOWN		Barriers are in down position
br_MOVUP		Barriers are moving up
br_MOVDOWN		Barriers are moving down
t_BellsState	<enumeration>	Enumeration to provide flashing bells states
bl_ON		Bells are in ringing state
bl_OFF		Bels are off
t_LevelCrossingState	<enumeration>	Enumeration to provide level crossing states
lc_OPENED		Level crossing are opened to traffic. Last state was lc_OPENNING
lc_CLOSED		Level crossing are closed to traffic. Train was detected and last state was lc_CLOSING
lc_OPENING		Level crossing are closed to traffic. Train was passed in the crossing and last state was lc_CLOSED
lc_CLOSING		Level crossing are closed to traffic. Train was detected and is in approach. The last state can be lc_OPENED or lc_OPENNING
lc_TIMEOUT_SIGNALS		Level crossing are opened to traffic. Can be viewed like a subset of lc_CLOSING state. The last state was lc_OPENED
lc_TIMEOUT_BARRIERS		Level crossing are opened to traffic. Can be viewed like a subset of lc_CLOSING state. The last state was lc_TIMEOUT_SIGNALS
t_LigthsState	<enumeration>	Enumeration to provide ligths states
lg_OFF		Ligths are off
lg_ON		Lights are flashing

Figura 5.19: Tipos criados no projecto da passagem de nível

A máquina de estados *SCADE* que modela o comportamento geral da PN, é dada pela figura 5.20. A estrutura modular do projecto permite que as condições de transição entre os estados da FSM apresentada estejam definidas em operadores próprios. A figura 5.21 apresenta a especificação de uma dessas condições.

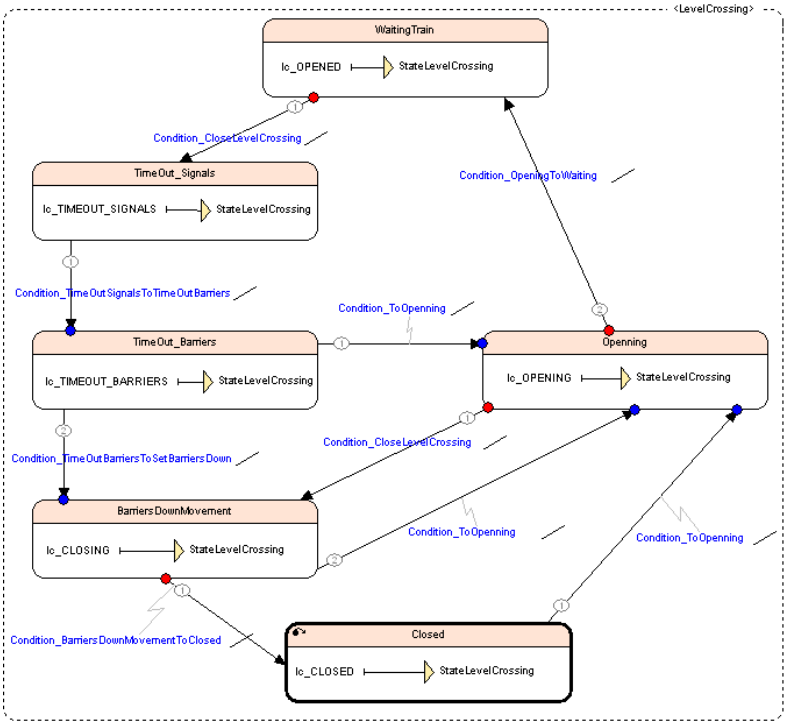


Figura 5.20: Máquina de estados que modela uma PN

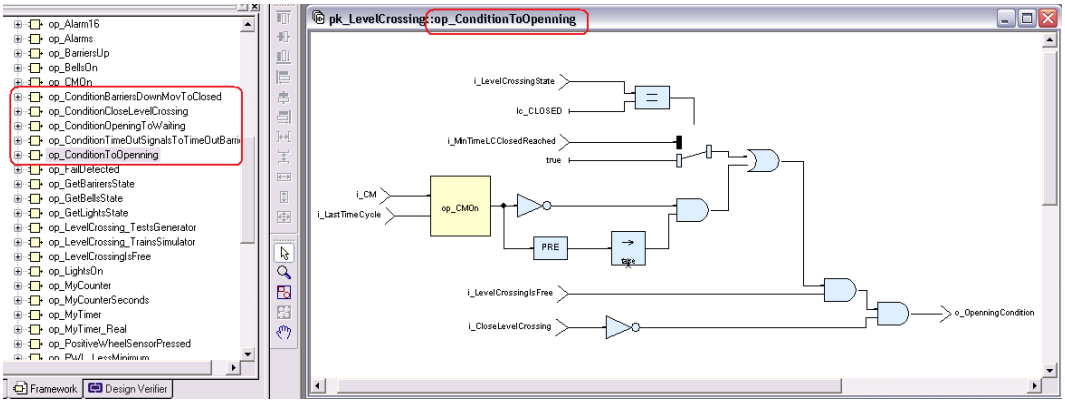


Figura 5.21: Exemplo de uma condição de transição

O objecto PN que define a interface com a máquina de estados da figura 5.20 é dada pela figura 5.22. Este objecto por ser modular pode ser aplicado em projectos futuros, tais como um sistema de sinalização.

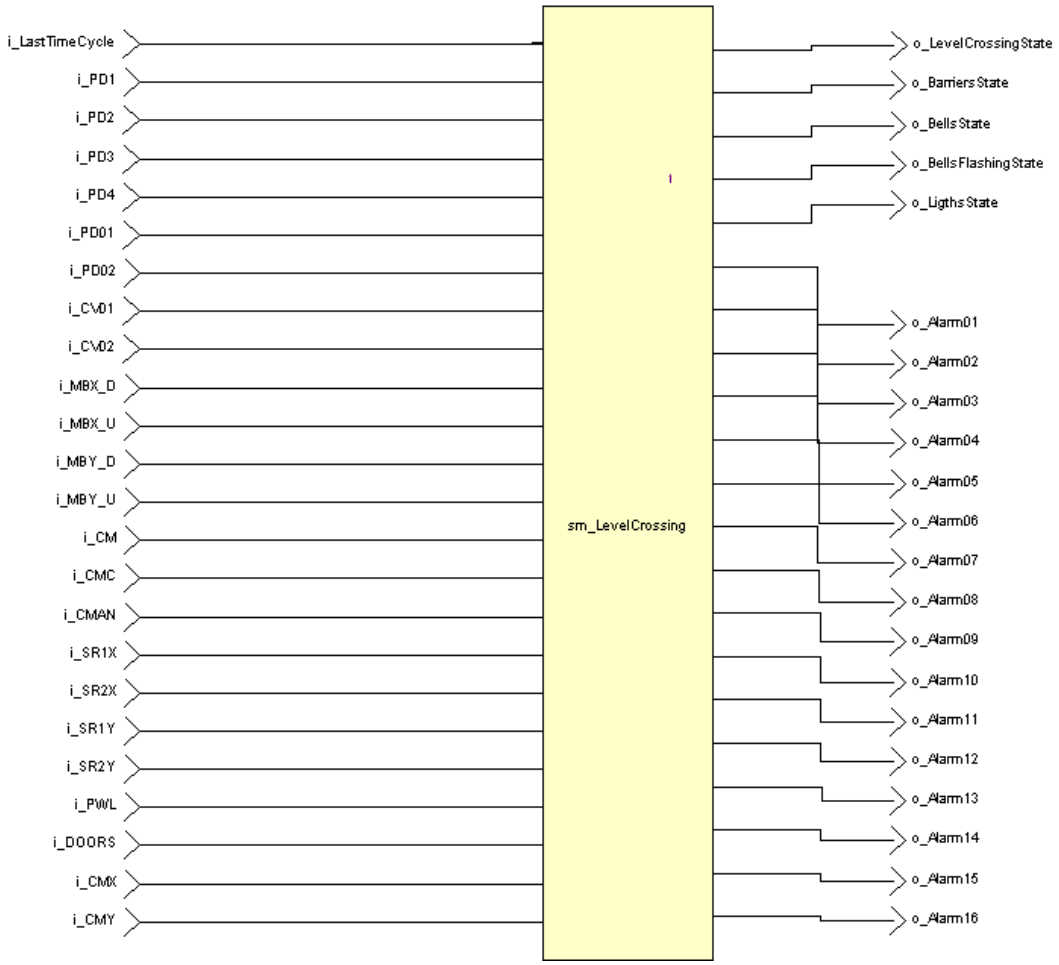


Figura 5.22: Objecto PN

A especificação *SCADE* que modela o comportamento das barreiras, sinais luminosos e sinais sonoros é dado pelas figuras 5.23, 5.24 e 5.25, respectivamente.

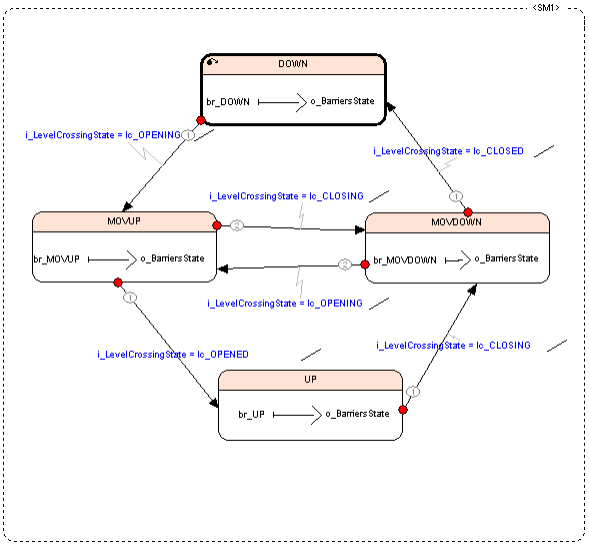


Figura 5.23: Máquina de estados das barreiras

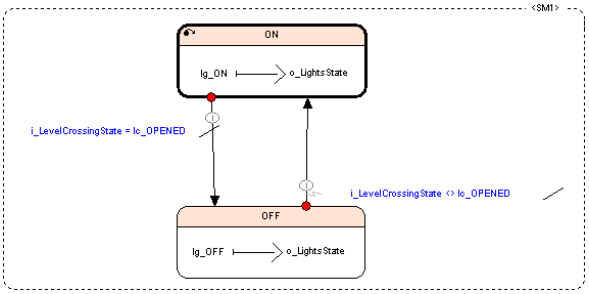


Figura 5.24: Máquina de estados dos sinais

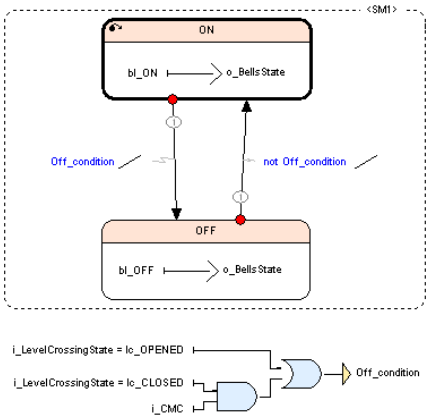


Figura 5.25: Máquina de estados dos sinais sonoros

A especificação da PN em *SCADE* resultou num total de 53 operadores, para gerir diversas situações relativas ao funcionamento da PN. Para uma actividade de simulação menos interactiva com o utilizador foi desenvolvido também em *SCADE* um *package* designado *Feeder* com a finalidade de alimentar o objecto da PN em actividades de verificação dinâmica. Este *package* possui um operador que possibilita parametrizar, no máximo 3 veículo ferroviários em cada um das linhas. Nessa parametrização são pedidos os dados relativos à velocidade, distancia ao pedal de anuncio, distancia entre inicio de circuito de via e pedal de circuito de via, entre outros. Com este módulo podemos alimentar PN diferentes em termos de configurações físicas. A figura 5.26 apresenta a utilização do operador principal do pacote *Feeder*, que é utilizado para alimentar cada uma das linhas do objecto PN.

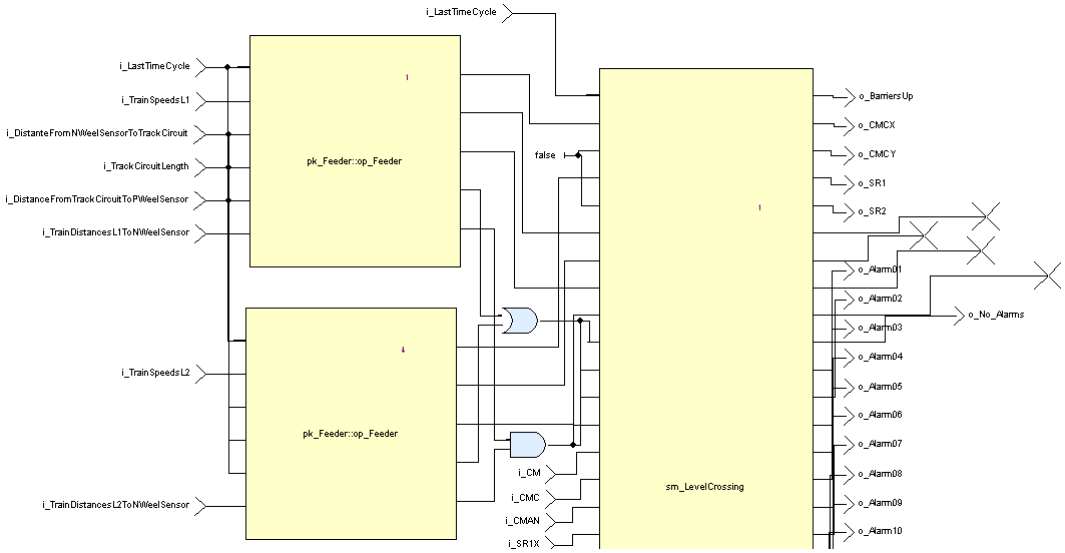


Figura 5.26: Aplicação do pacote Feeder

Verificação Dinâmica

A verificação dinâmica do modelo foi usada constantemente em cada passo da especificação. A flexibilidade do *SCADE* permite recorrer à simulação de pequenos operadores e verificar dinamicamente o seu comportamento de forma isolada do restante modelo. Assim, este processo de verificação dinâmica foi utilizado em todas as fases do ciclo de modelação, não existindo especificamente uma fase de verificação dinâmica isolada. A seguir é apresentada um figura que mostra uma actividade de simulação sobro todo o modelo da PN. Na impossibilidade de representar numa única figura todo o modelo, optou-se por mostra apenas a máquina de estados da PN em simulação (ver figura 5.27).

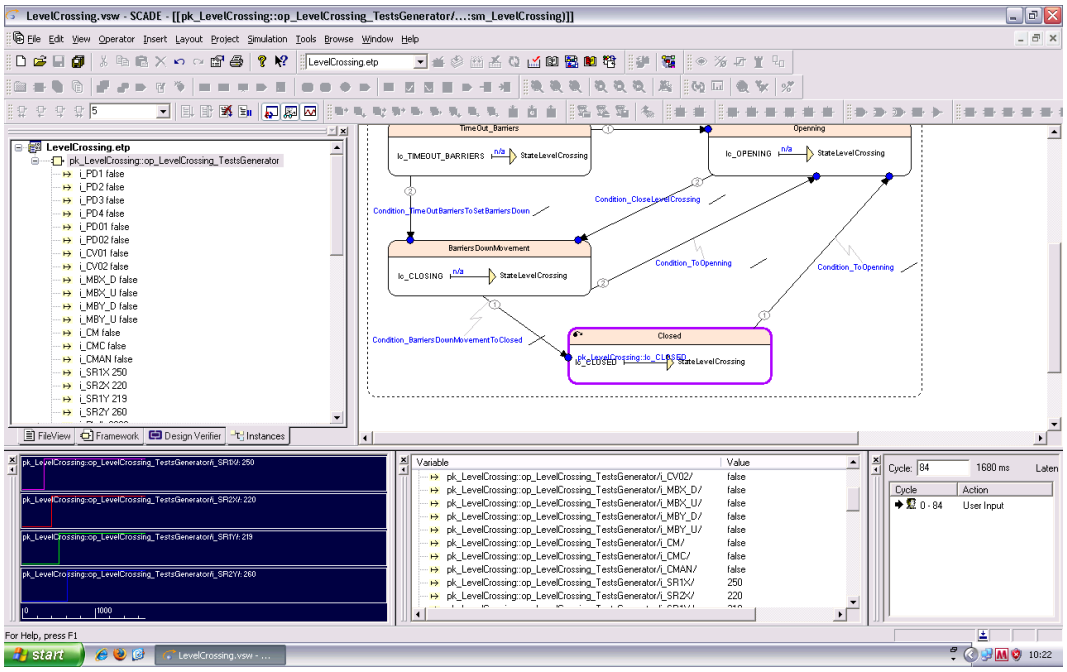


Figura 5.27: Actividade de simulação da PN

O *SCADE Simulator* foi, também, utilizado para criar cenários de teste. Estes, serão utilizados na análise de cobertura do modelo - ver secção 5.2.2. Os cenários gerados correspondem a cada um dos requisitos do sistema e é verificada a sua conformidade com o pretendido.

Verificação Formal

A verificação formal das propriedades de segurança do sistema PN foi feita recorrendo ao *SCADE Design Verifier*. Não vamos mostrar o processo de prova efectuado para todos os requisitos do sistema, vamos apenas abordar as especificadas em 5.2.1.

Requisito 1 O primeiro requisito do sistema que vamos verificar formalmente é: "O sistema deve arrancar em estado seguro, ou seja, deve arrancar com a PN em estado fechado". Para provar esta propriedade foi necessário proceder à especificação do *property node* que a exprime. A figura 5.28 apresenta o diagrama em Scade que define essa propriedade.

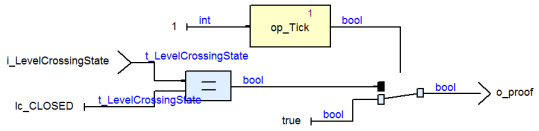


Figura 5.28: *Property node* do primeiro requisito

Requisito 2 O segundo requisito do sistema que vamos verificar formalmente é: "A PN não pode estar em estado fechada e ter as barreiras na posição vertical (permitir que o trânsito rodoviário avance)". O *property node* que a exprime em Scade é dado pela figura 5.29.

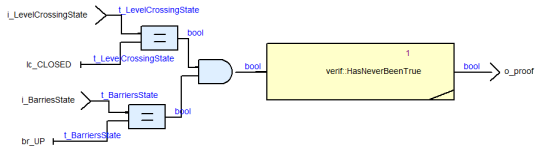


Figura 5.29: *Property node* do segundo requisito

Requisito 3 O terceiro requisito do sistema que vamos verificar formalmente é: "Se as barreiras não estiverem na posição vertical (abertas), então os sinais luminosos estão em funcionamento". A propriedade que, em Scade, exprime esta propriedade é dada pela figura 5.30.

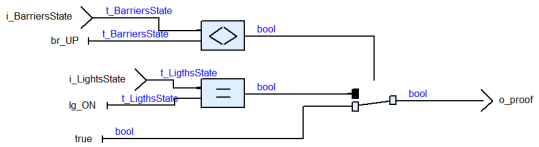


Figura 5.30: *Property node* do terceiro requisito

Requisito 4 O quarto requisito do sistema que vamos verificar formalmente é: "Se um dos pedais negativos for pressionado (detectado um veiculo), então a PN não pode estar aberta nem em processo de abertura". A figura 5.31 apresenta o diagrama em Scade que define essa propriedade.

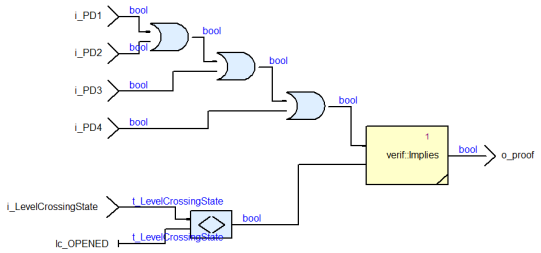


Figura 5.31: *Property node* do quarto requisito

Depois de definidas, ou expressas, em Scade todas as propriedades que pretendemos verificar no sistema, procede-se à especificação do *observer*. O *observer* Scade definido para contextualizar as propriedades do sistema PN é utiliza dois diagramas de especificação. O primeiro define as restrições aplicadas aos inputs do sistema e as variáveis do sistema que vão ser utilizadas na verificação das propriedades (ver figura 5.32). No segundo diagrama temos a contextualização das propriedades no sistema (ver figura 5.33).

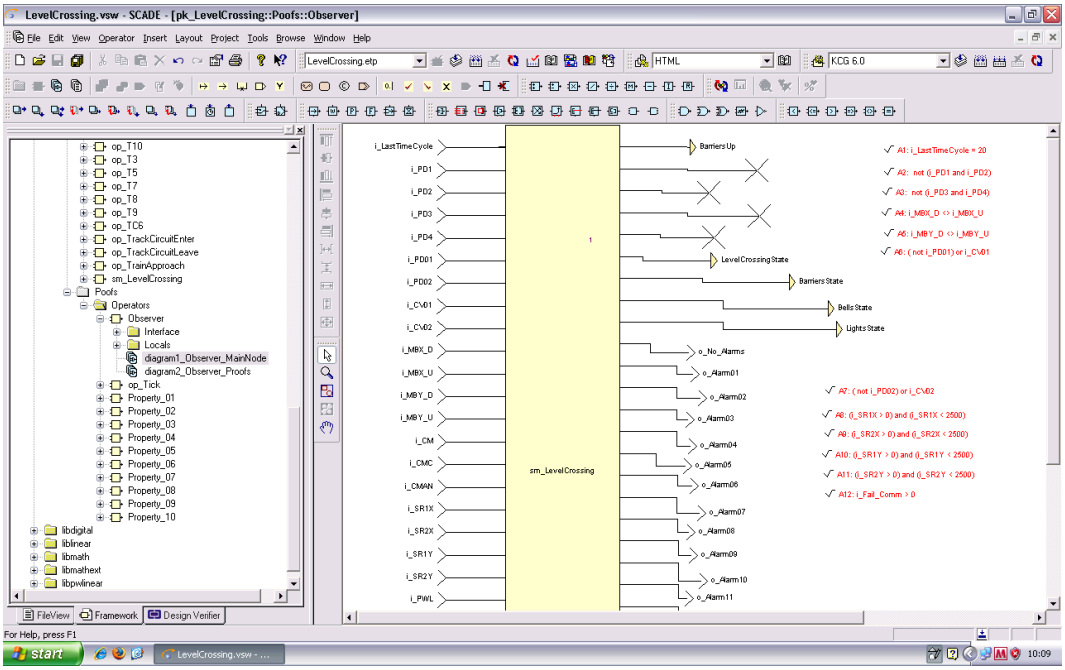


Figura 5.32: Diagrama 1 do *observer*

Definidas as propriedades e o operador que observa a validade dessas propriedades, procedeu-se à verificação formal. O resultado da actividade de verificação é dado pela figura 5.34.

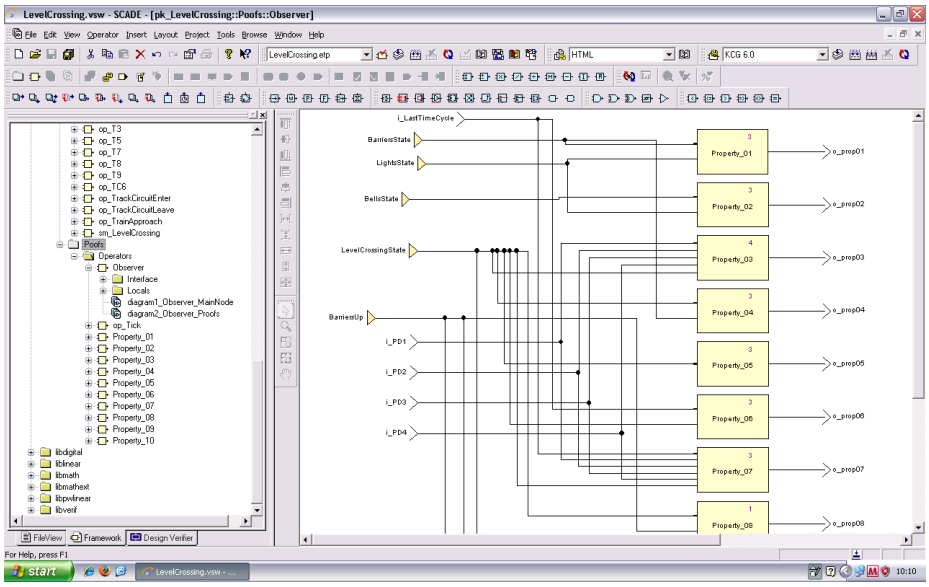


Figura 5.33: Diagrama 2 do *observer*

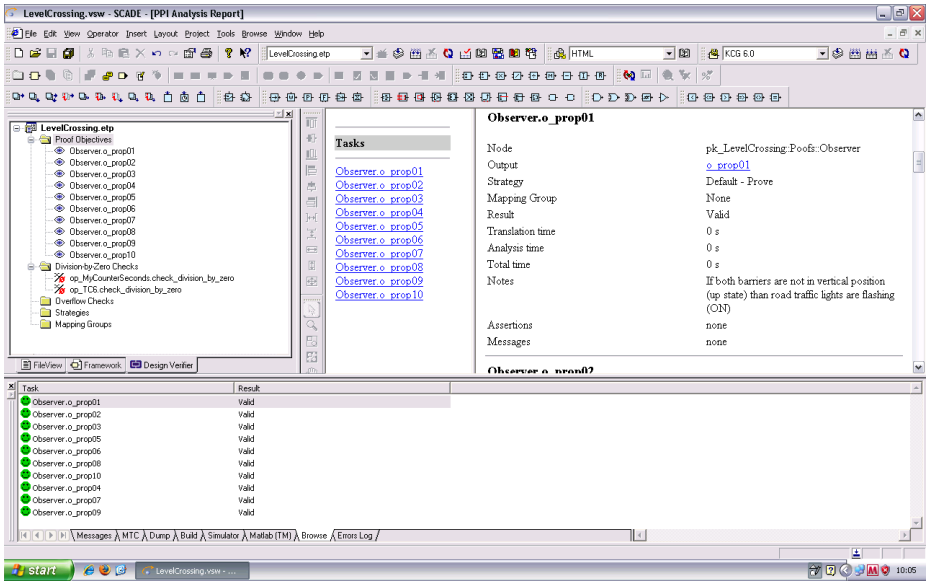


Figura 5.34: Resultado do *SCADE Design Verifier*

Análise do Modelo

Para analisar a cobertura ao modelo foram utilizados os cenários de teste gerados na fase de verificação dinâmica. - ver secção 5.2.2. A análise de cobertura ao modelo atingiu uma taxa de 91% (ver figura 5.35). Esta taxa significa que 9% do modelo não foi alvo de qualquer activação na execução dos requisitos do sistema. Esta taxa de 9% é explicável

pela arquitectura modular do projecto. Existem operadores genéricos que aplicados em certas situações, possuem estados internos que nunca são atingidos. Por exemplo, a figura 5.36 apresenta um contador básico que permite definir o relógio da contagem, ou seja, os instantes em que o contador incrementa o seu valor.

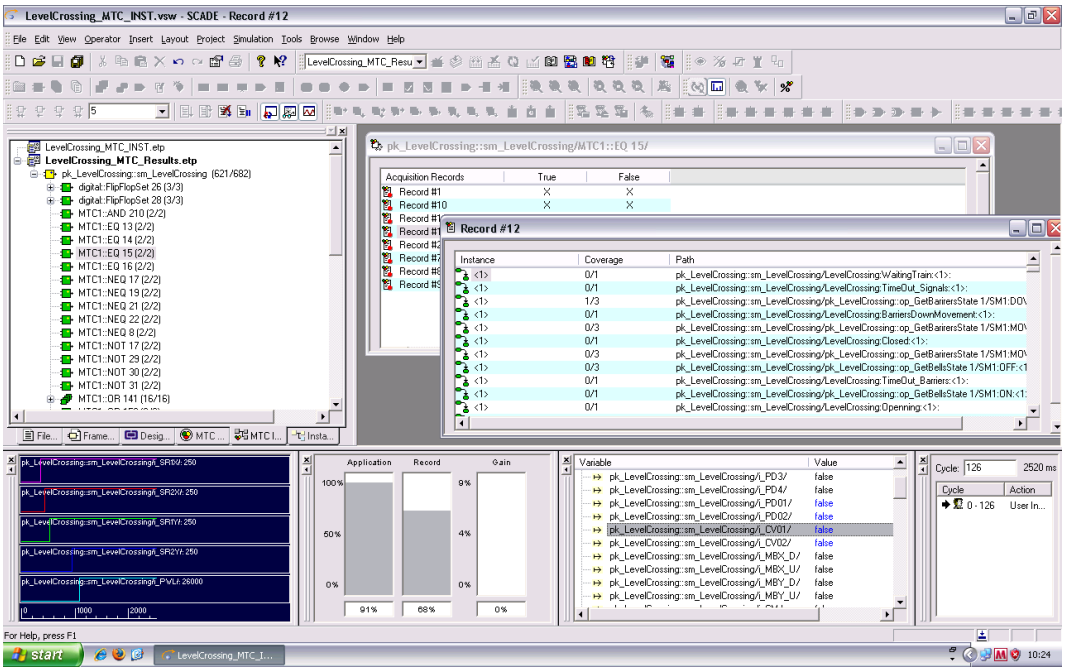


Figura 5.35: Actividade de análise de cobertura ao modelo

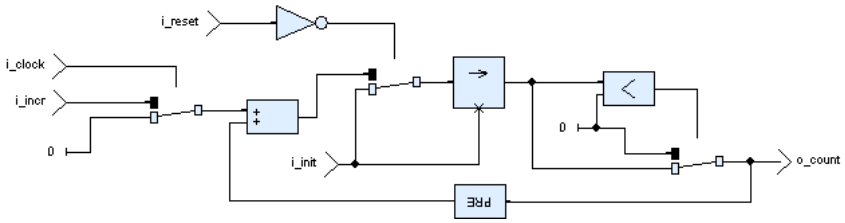


Figura 5.36: Contador definido em SCADE

No entanto a aplicação deste operador em diversos contextos recorre apenas a um relógio de valor constante `TRUE` e por isso a condição do operador `IF` nunca é `FALSE`. Tal como, por vezes, o valor do parâmetro `Reset` nunca é `TRUE`.

5.2.3 Concepção

A concepção do modelo na ferramenta *ELOP II Factory* foi executada com base no documento gerado automaticamente pelo *SCADE Suite 6.0*. A tradução foi directa pelo facto de termos restringido a utilização de certos operadores em *SCADE*. Inicialmente tinha siso modela uma máquina de estados para a PN usando todos os recursos providenciados pelo *SCADE*. No entanto, para simplificar ao máximo a tradução do modelo e facilitar a sua implementação no *ELOP II Factory*, foi necessário proceder à reconfiguração do modelo. As figuras que se seguem evidenciam a similaridade entre as construções *SCADE* e as construções *ELOP II Factory*. A figura 5.37 representa o operador `TC6`.

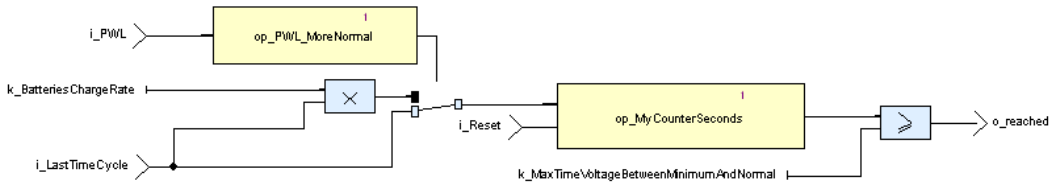


Figura 5.37: Operador `TC6` em Scade

O mesmo operador no *ELOP II Factory* é dado pela figura 5.38.

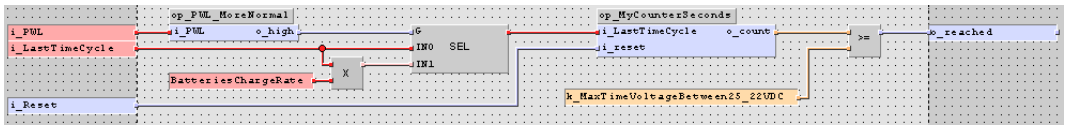


Figura 5.38: Operador `TC6` no *ELOP II Factory*

A figura 5.39 apresenta a máquina de estados do objecto PN, programada no *ELOP II Factory*. Pode comparar-se com a figura 5.20 que especifica a mesma máquina no ambiente *SCADE*.

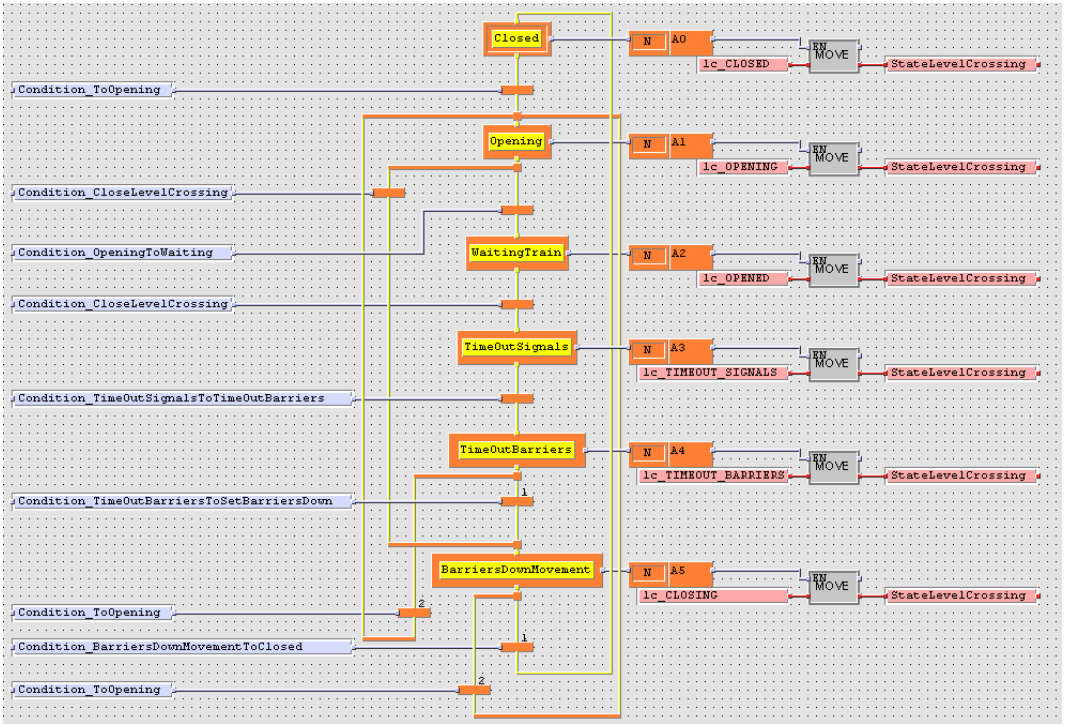


Figura 5.39: Máquina de estados da PN em *ELOP II Factory*

5.2.4 Testes e Simulação

Para reforçar a confiança no processo de tradução de *SCADE* para *ELOP II Factory* recorremos, nesta fase, ao *Simulator 1.0*. Os cenários de teste gerados na fase de verificação dinâmica e que representam cada um dos requisitos do sistema devem ser utilizados, também, nesta fase.

A figura 5.40 mostra a análise gráfica, do resultado de um cenário de teste que simula o arranque do sistema (PN fechada) e a abertura da PN mediante a passagem de dois veículos. A simulação contém 100 ciclos de execução e foram todos executados no controlador. O resultado da comparação originou quatro ciclos em que execução obtida difere da simulação *SCADE*. As diferenças obtidas acontecem sempre nos mesmos outputs mas não em todos os ciclos. A última página do relatório da simulação, obtido de forma automática, é representado pela figura 5.41.

Nos inputs existem ordens que indicam uma alteração do estado do ambiente. O problema é que as linguagens síncronas, como o SCADE, por assumir uma resposta instantânea aos eventos no modelo os outputs devolvidos na simulação são processados o mesmo ciclo. As diferenças acontecem nos outputs que controlam os sinais sonoros e luminosos e explica-se pela contagem de tempo diferente no *SCADE* e no *ELOP II Factory*. O *ELOP II Factory* possui operadores próprios para contagem de tempo enquanto que no *SCADE* não existem tais operadores. O que foi feito para colmatar esta necessidade foi criar um operador em

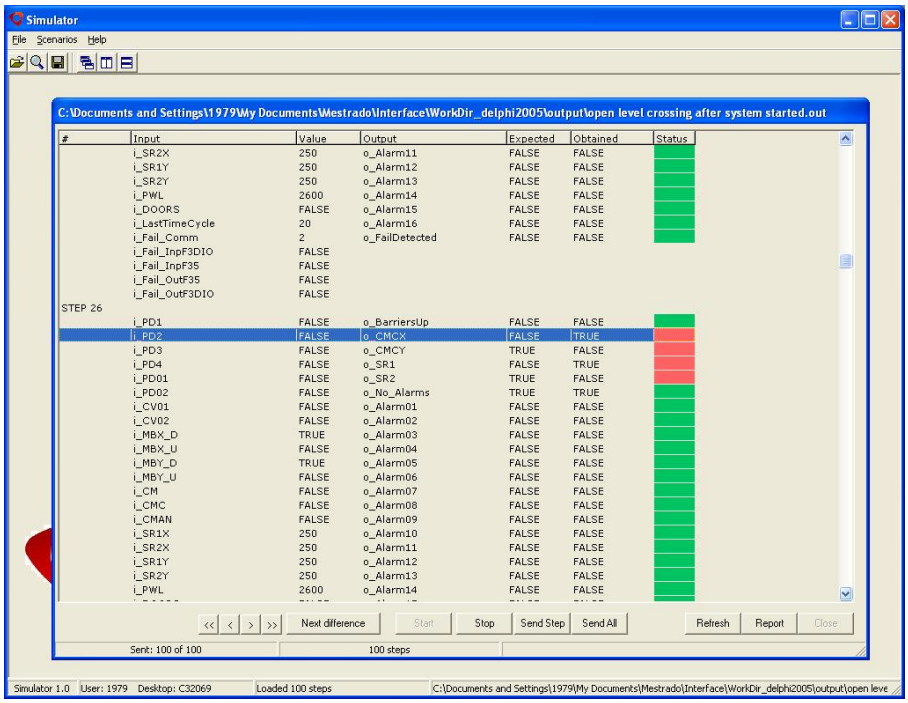


Figura 5.40: Análise gráfica do *Simulator 1.0*

SCADE para fazer contagem de tempo com base no tempo de execução do último ciclo efectuado. Em cada ciclo de execução o controlador, que executa o software, possui uma variável que indica o tempo de execução do ciclo anterior, e assim, a tarefa ficou facilitada. A figura 5.42 apresenta esse operador em *SCADE*.

Foi necessário, para manter a correspondência entre o modelo *SCADE* e o modelo *ELOP II Factory*, criar o mesmo operador no *ELOP II Factory*. Por teste e simulação os operadores funcionam da mesma forma mas no entanto temos os referidos problemas. Neste momento ainda não foi detectado o real problema do desfasamento na contagem do tempo nos outputs referidos. Este processo esta ainda a ser alvo de análise e testes.

O resultado do teste usando o *Simulator 1.0* permitiu, desde logo, detectar um desfasamento na contagem de tempo efectuada no modelo *SCADE* e no modelo *ELOP II Factory*. É um facto relevante e que dá importância ao *Simulator 1.0*. O desfasamento encontrado resume-se a alguns milissegundos no tempo de execução, o que seria impossível de detectar sem recurso a uma ferramenta automática de testes. Relativamente aos restantes outputs, da mesma simulação a correspondência entre o modelo *SCADE* e o modelo *ELOP II Factory* foi de 100%.

efacec
Sistemas de Electrónica, S.A.
Divisão de Sinalização

3. Conclusão

Num teste com 100 ciclos de execução existem 4 ciclos em que o valor dos outputs obtidos não são iguais aos outputs esperados. O número de ciclos com diferenças equivale a uma percentagem de 4,00 do total de ciclos do teste.
Cada ciclo contém exactamente 27 valores de entrada, e 23 valores se saída. O ciclo que registou o maior número de diferenças foi o ciclo nº 26, com um total de 4 diferenças detectadas. Por sua vez, o ciclo que registou o menor número de diferenças foi o ciclo nº 26, com um total de 4 diferenças detectadas.
Os resultados obtidos permitem concluir que o número médio de diferenças por cada ciclo - valor absoluto - é de 0,16 enquanto que em valor relativo é de 4,00.

3.1. Quadro Resumo

Parâmetro	Valor obtido
Inputs por ciclo	27
Outputs por ciclo	23
Nº de ciclos do teste	100
Nº de ciclos com diferenças	4
Nº de diferenças detectadas	16
Ciclo com mais diferenças	26
Nº de diferenças	4
Ciclo com menos diferenças	26
Nº de diferenças	4
Média absoluta	0,1600
Média relativa	4,0000

Autor:	Data: 14-05-2009	Aprovado:	Data: 14-05-2009
Título do Documento:			Pagina: 6 / 6
Nº do Documento:			Revisao: 14-05-2009

Figura 5.41: Relatório gerado pelo *Simulator 1.0*

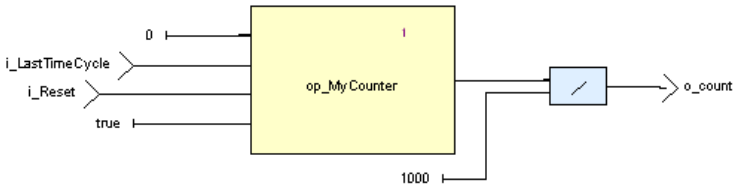


Figura 5.42: Operador para contagem de tempo

Capítulo 6

Conclusão

O trabalho apresentado neste documento define um conjunto de passos, com base numa *toolchain* composta pelas ferramentas *DOORS*, *SCADE*, *ELOP II Factory* e a ferramenta interna *Simulator 1.0*, para desenvolver sistemas ferroviários de acordo a norma EN 50128.

6.1 Análise Critica

O processo apresentado não usufrui da potencialidade total do *SCADE*. Nomeadamente, não usufrui do *SCADE KCG*. O principal motivo prende-se com o facto de os controladores electrónicos utilizados não permitem embeber o código gerado automaticamente pelo *SCADE KCG*. Uma solução ideal, seria automatizar o processo de codificação com base num compilador. O compilador seria capaz de traduzir uma especificação em linguagem Lustre numa das linguagens definidas pela norma IEC 61131-3. Esta norma é responsável pela definição de linguagens de programação para controladores electrónicos de lógica programável, e prevê quatro tipos de linguagem de programação:

1. Ladder diagram (LD), gráfica;
2. Function block diagram (FBD), gráfica;
3. Structured text (ST), textual;
4. Instruction list (IL), textual.

Esta seria uma solução óptima caso o *ELOP II Factory* permitisse a importação do resultado do compilador idealizado. Assim sendo, foi implementado um processo de codificação manual. Este processo, para além de ser um processo mais demorado, exige ainda uma verificação da correspondência entre o modelo *SCADE* e o modelo introduzido no *ELOP II Factory*. A solução apresentada baseia-se na ferramenta *Simulator 1.0* para testes automáticos ao software concebido. Claramente, esta solução não é tão completa e apenas garante a correspondência entre uma simulação - obviamente um conjunto limitado de todo o domínio possível de cenários - e a execução deste.

Outra desvantagem desta solução, de testes automáticos usando o *Simulator 1.0*, é a forma como os cenários são gerados. Estes, são resultado de uma premeditação do utilizador e não de uma selecção aleatória. Neste âmbito, foram analisadas algumas ferramentas de geração automática de testes com base em especificações Lustre, nomeadamente as ferramentas Lutess [9, 28], Lurrete [1, 20, 29] e Gatel [26]. No entanto, estas são ferramentas académicas, o seu acesso é difícil e os casos de aplicação industrial são praticamente inexistentes. No entanto, e apesar das limitações, considera-se que a contribuição da solução, recorrendo ao *Simulator 1.0*, é extremamente positiva no ciclo de desenvolvimento apresentado neste trabalho.

A não utilização do *SCADE KCG* e a consequente restrição na utilização de certos operadores *SCADE* também é uma questão que deve ser alvo de reparo. Essa restrição implica modelo menos legíveis, menos flexíveis e de implementação mais difícil. A restrição também só funciona até certa medida. Na especificação de sistemas reactivos utilizando o *SCADE* é inevitável a utilização de operadores de lógica temporal. No entanto, no *ELOP II Factory* não existem operadores de base que permitam responder a esta necessidade. A implementação destes operadores no *ELOP II Factory* não garante um funcionamento idêntico aos dos *SCADE*. Por outras palavras, a única forma de garantir a que os operadores partilham o mesmo comportamento é com base em testes e simulação. Os operadores de lógica temporal são apenas um pequeno exemplo, de entre todos os operadores *SCADE* que devem ser evitados no ciclo de desenvolvimento apresentado neste documento.

6.2 Trabalho Futuro

Na tentativa de obter todo o potencial do *SCADE*, evitar o processo de codificação manual e a utilização do *Simulator 1.0* para análise do software, poder-se-à optar por uma das seguintes soluções:

1. A primeira solução propõe a alteração de controladores electrónicos de forma a ser possível integrar nestes o software gerado pelo *SCADE KCG*.

2. A ideia de um compilador Lustre -> LD || FBD || ST || IL mantém-se. Poderá haver interesse comercial do fabricante HIMA em possuir uma interface que lhe permita obter uma implementação, a partir de um modelo que é resultado de uma ferramenta certificada e com sucesso industrial no domínio dos transportes ferroviários e aviação. O processo de compilação em linguagem máquina continuaria a ser uma actividade do *ELOP II Factory*. A existência desse compilador permitiria automatizar todo o processo de desenvolvimento descrito neste documento. O risco inerente ao processo manual seria evitado e deixaria de ser necessária a utilização do *Simulator 1.0*.

3. Um outra solução passa por usar ferramentas de geração e execução de testes automáticos. As ferramentas Lutess, Lurrete e Gatel são um exemplo real do funcionamento de tais ferramentas e podem ser usadas como referência.

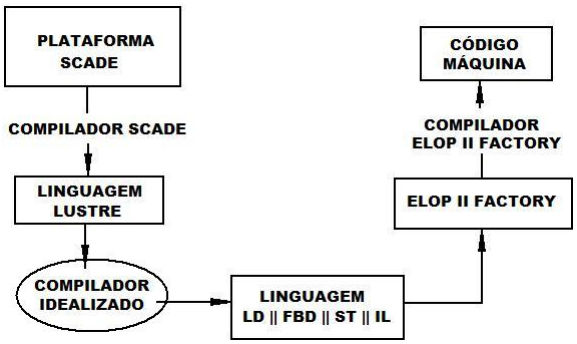


Figura 6.1: Esquema do processo de transformação de um modelo Scade em código máquina

6.3 Conclusão

O ciclo de desenvolvimento proposto tem como figura central o *SCADE*. Esta, providencia ferramentas que permitem acompanhar, de forma sustentada, o desenvolvimento de software para o sector ferroviário. Mais do que acompanhar o software no seu ciclo de vida, a plataforma *SCADE* permite fazê-lo com base em formalismos matemáticos, que por sua vez permitem uma abordagem formal ao software. Esta abordagem é altamente recomendada pela norma EN 50128.

A gestão documental, relativa ao projecto, através do *DOORS* permite uma rastreabilidade em qualquer fase do ciclo de desenvolvimento, que é também um requisito essencial na norma EN 50128.

O *ELOP II Factory*, é uma ferramenta de programação, parametrização e configuração, dos controladores do fabricante HIMA, certificada em SIL4 na norma EN 50128.

O *Simulator 1.0* é uma ferramenta interna não certificada. No entanto, a norma EN 50128 prevê apenas a execução de testes ao software, ignorando a forma como eles são feitos. Portanto, o *Simulator 1.0* é uma ferramenta de apoio que permite maior rapidez e comodismo na execução dos testes e, consequentemente, permite uma execução de testes mais exaustiva o que, certamente, é uma vantagem.

Desta forma, considera-se que o software resultante da aplicação do ciclo apresentado neste documento, cumpre o normativo CENELEC EN 50128.

Bibliografia

[1] The lurette v2 user guide. Technical Report TR-2004-5, Verimag Research Report.

[2] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture notes for 49285 Advanced Algorithms E97, October 1997.

[3] Stefano Bacherini, Simone Bianchi, Leonardo Capecchi, Carlo Becheri, Alessandro Felleca, Alessandro Fantechi, and Emilio Spinicci. Modelling a railway signalling system using sdl. Via Pietro Fanfani, 21, Firenze, Italy, I-50127 and Via di S. Marta, 3, Firenze, Italy, I-50139, 2001.

[4] Michele Banci. Geographical vs. functional modelling by statecharts of interlocking systems.

[5] Michele Banci, Alessandro Fantechi, and Stefania Gnesi. Some experienceson formal specification of railway interlocking systems using statecharts.

[6] Gérard Berry. The foundations of esterel. Technical report, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France.

[7] Gérard Berry. The contrutive semantics of pure esterel. Draft 3, Centre de Mathématiques Appliqués, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France, July 1999.

[8] Amar Bouali and Projet Meije. Xeve: an esterel verification environment (version v1.3), 1997.

[9] L. Bousquet, F. Ouabdesselam, J. l. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *In 21st International Conference on Software Engineering*, pages 267–276. ACM Press, 1999.

[10] Instituto Nacional de Estatística. Transportes ferroviários, Março 2005. `http://www.ine.pt/xportal/xmain?xpid=INE&xpgid=ine_destaques&DESTAQUESdest_boui=73175&DESTAQUESmodo=2`.

[11] António Alves e Dario Silva. Carris de ferro em portugal, Maio 2009. http://www.ocomboio.net/PDF/montpellier/portugais/antonioalves.pdf.

[12] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2001.

[13] Stephen A. Edwards. An esterel compiler for large control-dominated systems, 2002. To appear in IEEE transactions on computer-aided design of circuits and systems.

[14] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Trans. Des. Autom. Electron. Syst.*, 8(2):141–187, 2003.

[15] CLEARSY System Engineering. B language, reference manual - version 1.8.6.

[16] CLEARSY System Engineering. Industrial use of the b method, Janeiro 2009. http://www.clearsy.com/pdf/ClearSy-Industrial_Use_of_%20B.pdf.

[17] Esterel history, Maio 2009. http://www.softwaresafety.net/Esterel.org/History.htm.

[18] Esterel Technologies SA, France. *The Esterel v7 Reference Manual Version v7.51 for Esterel Studio 6.0*, December 2007.

[19] Esterel Technologies SA, France. *SCADE Suite Documentation*, September 2007.

[20] F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *Fifth International Workshop on Automated and Algorithmic Debugging AADEBUG'2003,Ghent (Belgium)*, 2003. http://www.irisa.fr/vertecs/Publis/Ps/aadebug03.ps.gz.

[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, September 1991.

[22] Nicolas Halbwachs and Pascal Raymond. A tutorial of lustre. Lustre V4, January 2002.

[23] Bernard HOUSSAIS. *The Synchronous Programming Language SIGNAL - A Tutorial*. IRISA. ESPRESSO Project, France, April 2002.

[24] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal methods in safety-critical railway systems, Abril 2009. http://www.fersil.fr/presse/Formal_methods_in_safety_critical_railway_systems.pdf.

[25] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

[26] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions: Gatel. Technical report.

[27] *Norma Portuguesa: Aplicações ferroviárias - Sistemas de sinalização, telecomunicações e processamento de dados - Software para sistemas de protecção e comando ferroviário*, Outubro 2002. Versão Portuguesa da EN 50128:2001.

[28] Virginia Papailiopoulou. Automatic test generation for lustre/scade programs, 2008.

[29] Pascal Raymond and Daniel Weber. Lurette: User manual.

[30] John Rushby. Formal methods and the certification of critical systems. Technical report, Computer Science Laboratory SRI International, Menlo Park CA 94025 USA, December 1993.

[31] Esterel Technologies SA. *Methodology Handbook - Efficient Development of Safe Railway - Applications Software with EN 50128 - Objectives Using SCADE Suite*. Esterel Technologies SA, Parc Euclide - 8, rue Blaise Pascal, 78990 Elancourt, FRANCE, second edition, September 2008.

[32] Model - railway laboratory course, Abril 2009. `https://rtsys.informatik.uni-kiel.de/trac/railway/wiki/WikiStart`.

[33] Satnam Singh. Design and verification of peripheral control circuits in esterel. Technical report, Microsoft Research Cambridge, 7 JJ Thomson Avenue, Cambridge, CB3 0FB, U.K., 2008.

[34] VERIMAG. A short story of lustre, November 2008. `http://www-verimag.imag.fr/~synchron/index.php?page=lustre-story/history`.