EaSys – Uma Linguagem Orientada por Objectos para Descrição de Sistemas Digitais*

Arnaldo Oliveira, Valery Sklyarov, António Ferrari

***Resumo* – Este artigo discute a utilização de linguagens de programação orientadas por objectos no projecto de sistemas digitais. Os conceitos aqui abordados são particularmente úteis para desenvolver sistemas complexos compostos por componentes de hardware e de software. A linguagem EaSys descrita neste artigo foi concebida para permitir o uso de apenas uma linguagem ao longo de todo o fluxo de projecto. Esta linguagem é uma extensão ao C++, implementada através de uma biblioteca de classes, que adiciona à linguagem base um conjunto de abstracções e mecanismos úteis para a modelação de hardware. Para escrever, compilar e depurar um modelo de um sistema escrito em EaSys, são necessárias apenas ferramentas standard de desenvolvimento em C++.**

***Abstract* – This paper discusses the use of object-oriented programming languages in the design of digital systems. The ideas presented here are particularly useful to develop complex systems composed of hardware and software components. The EaSys language described in this paper was developed to allow the use of a single language in the entire design flow. This language is an extension to the C++, implemented through a class library, that adds to the base language a set of useful abstractions and mechanisms for hardware modelling. To write, compile and debug a system model written in EaSys, only standard C++ development tools are needed.**

I. INTRODUÇÃO

A evolução contínua dos processos de fabricação de circuitos integrados VLSI (*Very Large Scale Integration*) tem possibilitado a construção de dispositivos cada vez mais complexos, de menor dimensão e com melhor desempenho. A tecnologia actual permite já a implementação de Sistemas Integrados (*Systems-on-Chip – SoC*), os quais incluem numa única pastilha de silício um elevado número de componentes, tais como microprocessadores de uso geral, processadores digitais de sinal, memórias e circuitos específicos da aplicação. Assim, o processo de projecto de um SoC não consiste apenas no desenvolvimento do hardware, mas também do software que irá executar nos seus elementos de processamento programáveis. A execução de projectos desta complexidade com uma quantidade aceitável de recursos humanos e materiais requer uma produtividade elevada para que possa ser concluída com sucesso e em tempo útil. É precisamente este problema de produtividade que está na base das recentes mudanças ocorridas ao nível dos paradigmas usados no projecto de sistemas complexos e que se caracterizam essencialmente por:

- Utilização de componentes pré-projectados e completamente testados. A sua complexidade pode variar desde simples portas lógicas ou circuitos de aritmética a módulos complexos de processamento, como núcleos de processadores. Este procedimento é também vulgarmente designado por reutilização da propriedade intelectual (*Intellectual Property – IP*);
- Especialização de arquitecturas e plataformas genéricas predefinidas, para as quais existe um conjunto completo de ferramentas de desenvolvimento. Exemplos desta abordagem são a implementação de sistemas digitais em dispositivos lógicos programáveis de elevada capacidade como as FPGAs (*Field Programmable Gate Arrays*) e a optimização da implementação de um microcontrolador cuja arquitectura base é fixa mas em que os dispositivos de interface podem ser específicos da aplicação alvo;
- Adopção de métodos formais e uso de linguagens de especificação para construir na fase inicial do projecto um modelo abstracto e executável do sistema. Este modelo é iterativamente validado e refinado até à implementação. A linguagem usada deve também suportar eficientemente a descrição quer do hardware quer do software do sistema simplificando o projecto concorrente de todos os seus componentes.

Este artigo é dedicado à discussão de problemas e apresentação de soluções relacionadas com o último ponto. Além desta introdução, este artigo possui mais sete secções. Na secção II são discutidos alguns dos tópicos mais importantes do projecto de sistemas digitais complexos, tais como a utilização de especificações executáveis, as metodologias normalmente empregues e o uso de linguagens de programação no seu desenvolvimento. A descrição da linguagem EaSys e do respectivo núcleo de simulação é feita na secção III. Na secção IV são mostrados alguns exemplos de

* Este trabalho foi financiado pela bolsa de doutoramento SFRH/BD/3184/00 da Fundação para a Ciência Tecnologia.

componentes de hardware modelados com esta linguagem e na secção V descrito o seu processo de compilação e simulação. Na secção VI é feita uma comparação entre a linguagem EaSys e o SystemC. A extensibilidade da linguagem e a enumeração de alguns tópicos de trabalho futuro são tratadas na secção VII. Finalmente, as conclusões encontram-se na secção VIII.

II. PROJECTO DE SISTEMAS DIGITAIS COMPLEXOS

Nesta secção vão ser discutidas algumas das questões e problemas relacionados com o projecto de sistemas digitais complexos, nomeadamente, as vantagens da elaboração de especificações executáveis, as metodologias usadas no desenvolvimento de sistemas compostos por componentes de hardware e de software e a utilização de linguagens de programação orientadas por objectos ao longo de todo o fluxo de projecto deste tipo de sistemas.

A. Especificações Executáveis

A especificação inicial de um sistema, isto é, a descrição das suas características, comportamento e restrições é feita na maior parte dos casos em linguagem natural. Este método pode ser utilizado para descrever informalmente um sistema a projectar. Em sistemas simples pode também servir de base ao seu desenvolvimento. No entanto, no caso de sistemas complexos, o seu uso para efeitos de desenvolvimento levanta alguns problemas. Por não ser formal, uma descrição de um sistema em linguagem natural pode ser ambígua, incompleta, imprecisa e inconsistente, além de não poder ser verificada de uma forma sistemática. Estes problemas nem sempre são facilmente detectáveis numa fase inicial do projecto, que é quando a sua correcção é mais fácil e económica. Para os evitar, no início do fluxo de projecto, pode ser elaborada a partir da descrição inicial uma especificação executável, isto é, um modelo abstracto do sistema, implementado através de uma aplicação de software estruturada de acordo com um estilo bem estabelecido e cujo comportamento é o mesmo do sistema a projectar.

É importante notar que uma especificação executável não é apenas um conjunto de modelos parciais do sistema, construídos para validar determinados aspectos específicos, como por exemplo, os algoritmos utilizados. Uma especificação executável deve integrar num único modelo de alto-nível, toda a funcionalidade do sistema e as suas características mais importantes, tais como certas restrições temporais. A sua elaboração de acordo com um estilo bem estabelecido visa facilitar o seu desenvolvimento e utilização por diferentes pessoas.

As linguagens tradicionalmente utilizadas para escrever uma especificação executável são o C, o C++ [1] e o JAVA [2]. Esta escolha deve-se essencialmente a duas razões:

- Uma parte significativa do sistema é implementada em software escrito numa destas linguagens, o que facilita a sua integração com a especificação;
- Estas linguagens são mais poderosas que as linguagens de descrição de hardware (*Hardware Description Languages – HDLs*) tradicionais pemitindo escrever modelos mais compactos e a um nível de abstracção mais elevado. Como veremos mais à frente, o C++ e o JAVA podem também ser extendidos de forma a incluir as capacidades de descrição de hardware disponíveis nas HDLs.

A construção de uma especificação executável possui algumas vantagens muito importantes:

- O processo de desenvolvimento do programa ajuda a descobrir inconsistências e erros da descrição inicial do sistema. O teste do programa ajuda a perceber se a especificação inicial é ou não completa;
- Garante a ausência de ambiguidades na interpretação da especificação inicial do sistema. Sempre que surgir uma dúvida durante a implementação, a especificação pode ser executada para determinar qual deve ser o comportamento do sistema;
- Ajuda a validar a funcionalidade do sistema e os algoritmos utilizados antes de se iniciar a implementação;
- Facilita a criação de modelos iniciais de desempenho e a avaliar a eficiência do sistema;
- Possibilita a reutilização do conjunto de testes usados para validar a especificação executável, os quais podem ser refinados ou usados directamente para testar a implementação, permitindo reduzir consideravelmente o tempo dispendido em tarefas de verificação e teste.

Apesar destas importantes vantagens, uma especificação executável por si só permite solucionar apenas as questões relacionadas com a descrição e verificação iniciais do projecto de um sistema complexo. Para resolver o problema de produtividade referido acima, uma especificação executável deve ser integrada numa metodologia iterativa de descrição, verificação, refinamento e síntese suportada por ferramentas de computador para automação do projecto de sistemas electrónicos (*Electronic Design Automation – EDA*). A especificação executável actua como um modelo comportamental do sistema que é progressivamente transformado e validado até à implementação. Esta abordagem é uma generalização, para níveis de abstracção mais elevados, da síntese de hardware a partir de descrições comportamentais e RTL (*Register Transfer Level*) realizada actualmente e que se tem mostrado bastante eficaz no aumento da produtividade e, na maioria dos casos, na melhoria da qualidade final do sistema projectado.

B. Metodologias

Após a elaboração da especificação executável, o desenvolvimento de um sistema constituído por uma mistura de componentes de hardware e de software pode

ser em geral realizado segundo uma de duas metodologias distintas:

- Metodologia Multi-Linguagem (*Multi-Language Methodology – MLM*) – utiliza em cada etapa ou conjunto de etapas do fluxo de projecto a linguagem mais adequada, pelo que sempre que houver uma mudança de linguagem é necessário converter total ou parcialmente o modelo do sistema;
- Metodologia de Linguagem Única (*Single Language Methodology – SLM*) – utiliza a mesma linguagem em todas as etapas do projecto, dispensando qualquer conversão do modelo do sistema, mas à custa de uma maior versatilidade e sobrecarga da linguagem utilizada.

As metodologias MLM e SLM são descritas nas seguintes subsecções.

B.1. Metodologia MLM

As metodologias MLM são as tradicionalmente usadas no projecto de sistemas complexos compostos por componentes de hardware e de software. Na Figura 1 está representado o processo de desenvolvimento baseado numa metodologia deste tipo, onde são mostradas as etapas iniciais de simulação e refinamento do modelo inicial do sistema, bem como as etapas de simulação e síntese específicas do desenvolvimento dos seus componentes de hardware.

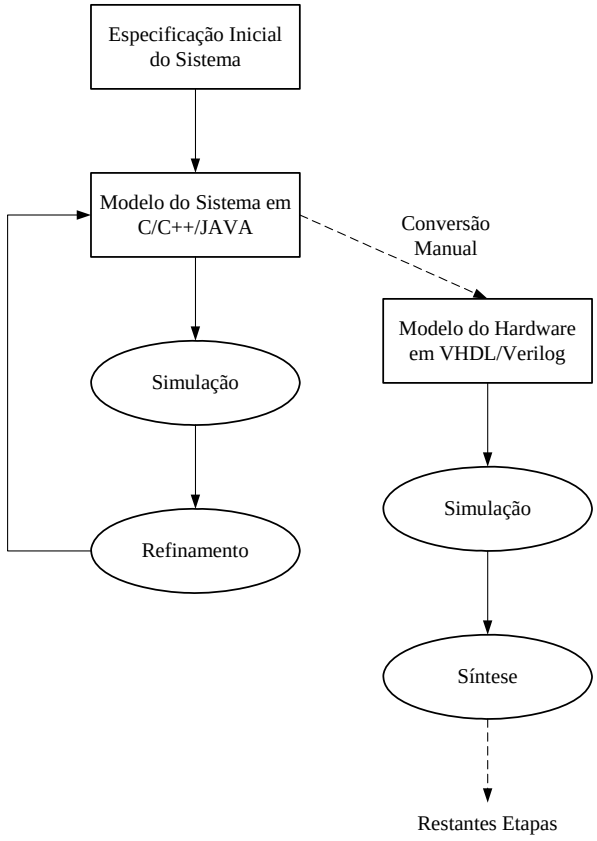

Figura 1 - Fluxo de projecto numa metodologia do tipo MLM.

Numa metodologia MLM, o ponto de partida é a especificação executável do sistema, normalmente escrita em C/C++/JAVA. Tal como foi referido na secção anterior, esta descrição actua como um modelo abstracto do sistema usado para verificar sob o ponto de vista funcional conceitos e algoritmos. Esta descrição é geralmente verificada e refinada iterativamente até se obterem os resultados esperados. Após a validação deste modelo, as partes a implementar em hardware têm que ser convertidas manualmente para um formato baseado numa HDL, normalmente VHDL [3] ou Verilog [4]. A descrição resultante é posteriormente simulada e processada pelas ferramentas de síntese e implementação específicas do hardware. Como veremos mais à frente, esta conversão é necessária porque as linguagens C/C++/JAVA não preenchem os requisitos mínimos para descrever hardware de uma forma simples, eficaz e intuitiva. As metodologias MLM, apesar de muito utilizadas, possuem alguns problemas:

- Introdução de erros durante a conversão de C/C++/JAVA para HDL – Esta conversão além de ser um processo moroso é também sujeito a erros, obrigando a que o modelo do hardware também seja validado por simulação;
- Perda de consistência entre os modelos em C/C++/JAVA e HDL – Após a conversão do modelo em C/C++/JAVA para HDL, o desenvolvimento do hardware baseia-se no refinamento e síntese do respectivo modelo. O modelo do sistema em C/C++/JAVA fica rapidamente desactualizado porque as alterações realizadas na descrição em HDL não são em geral introduzidas no modelo do sistema;
- Duplicação do esforço no desenvolvimento de testes – Os testes criados para validar o comportamento do modelo em C/C++/JAVA do sistema não podem em geral ser aplicados ao modelo HDL sem que sejam previamente convertidos. Assim, o projectista além de precisar de converter o modelo do sistema, tem também de converter o respectivo conjunto de testes.

B.2. Metodologia SLM

Para tentar solucionar os problemas das metodologias MLM têm sido propostas outras metodologias para o projecto de sistema complexos compostos por componentes de hardware e de software. Todas elas consistem na adopção de uma linguagem de programação de alto-nível com as características apropriadas para que possa ser utilizada em todo o fluxo de projecto incluindo o desenvolvimento do software e a implementação do hardware do sistema. A utilização de apenas uma linguagem desde a elaboração da especificação executável até à implementação dos componentes de hardware tem a vantagem das tarefas de simulação, refinamento e síntese serem realizadas iterativamente e sempre sobre o mesmo modelo do sistema, que é obviamente mantido consistente ao longo de todo o projecto. Assim, a conversão do

modelo em C/C++/JAVA para HDL da metodologia MLM deixa de ser necessária, a simulação é feita com um modelo escrito numa única linguagem e diferentes partes do sistema podem ser descritas a diferentes níveis de abstracção. A simulação do sistema completo é realizada num ambiente homogéneo dispensando os habituais interfaces complexos para co-simulação existentes nas ferramentas de projecto baseadas em HDLs que implementam metodologias do tipo MLM. Na Figura 2 estão representadas as etapas mais importantes de uma metodologia SLM para o caso da linguagem EaSys descrita neste artigo.

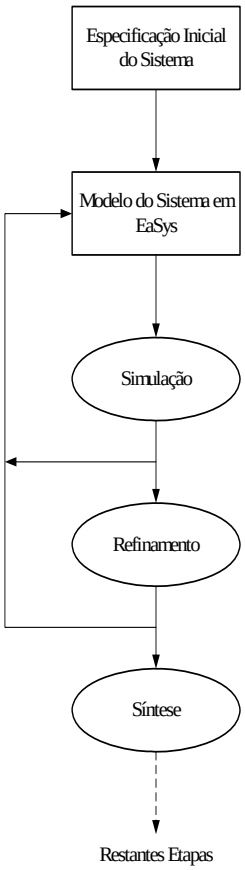

Figura 2 - Fluxo de projecto numa metodologia do tipo SLM.

Uma metodologia SLM possui relativamente a uma MLM algumas vantagens importantes:

- Refinamento progressivo – em nenhuma fase do projecto é necessário converter o modelo do sistema escrito numa linguagem para outra. O modelo é progressivamente simulado, refinado e enriquecido com detalhes desde a especificação até à implementação;
- Utilização de apenas uma linguagem – a linguagem utilizada permite não só descrever o sistema a um nível funcional abstracto mas também desenvolver o software do sistema e modelar o seu hardware ao nível comportamental, RTL ou lógico. Assim é necessário possuir apenas experiência numa única linguagem e utilizar um menor número de ferramentas de projecto;
- Reutilização dos programas de teste – os testes utilizados para validar o modelo abstracto do sistema podem ser aplicados ao modelo RTL, permitindo por um lado poupar o tempo de conversão e por outro aumentar a qualidade do processo de teste;
- Aumento da produtividade – quanto maior for o nível de abstracção de um modelo mais fácil é a sua elaboração e manutenção e a execução é mais rápida do que nos ambientes de simulação de hardware tradicionais. Além disso, tal como já foi dito, a utilização de ferramentas de síntese e implementação capazes de produzir hardware a partir de uma descrição comportamental tem-se mostrado também bastante eficaz no aumento da produtividade.

C. Linguagens

Tal como foi referido acima, a especificação executável de um sistema é normalmente feita em C/C++/JAVA porque estas linguagens além de serem as mais utilizadas no desenvolvimento de software permitem também escrever facilmente modelos a níveis de abstracção elevados. Já o hardware é tradicionalmente modelado com HDLs capazes de expressar conceitos específicos do hardware, tais como concorrência, sinais, comportamento reactivo e tipos de dados específicos deste domínio. As linguagens de programação tradicionais não suportam directamente nenhum destes conceitos, pelo que não são adequadas para descrever hardware. Por outro lado, as HDLs tradicionais foram concebidas especialmente para modelar hardware e são bastante limitativas ao nível dos paradigmas e níveis de abstracção suportados, pelo que não podem ser utilizadas no desenvolvimento de software nem na construção de modelos funcionais de alto-nível de um sistema. Por outras palavras, enquanto as linguagens de programação possuem limitações nos níveis de abstracção mais baixos normalmente usados para descrever hardware, as HDLs não são apropriadas para os níveis de abstracção mais elevados tipicamente utilizados no desenvolvimento de software. No caso de sistemas compostos tanto por componentes de hardware como de software, estas limitações levam à utilização de várias linguagens ao longo do fluxo de projecto e consequentemente a uma metodologia do tipo MLM com todos os problemas que daí resultam.

A definição de uma linguagem apropriada para ser usada numa metodologia do tipo SLM, isto é, que possa ser utilizada a vários níveis de abstracção, em todas as etapas de projecto de um sistema complexo (desde a construção da especificação ao desenvolvimento do software e projecto do hardware) e que suporte vários paradigmas de programação e modelos de computação, pode ser feita de três formas distintas:

- Concepção de uma nova linguagem capaz de lidar com todos os aspectos do projecto de um sistema digital com componentes de hardware e de software;
- Adição às HDLs de construções e das capacidades típicas das linguagens de programação;

- Extensão das linguagens de programação com os mecanismos necessários para descrever hardware.

Como será mostrado a seguir, devido à extensibilidade proporcionada pelo paradigma de orientação por objectos das linguagens C++ e JAVA, a última abordagem é mais conveniente porque a sua implementação é mais simples e a sua integração nos ambientes convencionais de especificação e desenvolvimento de software é mais directa. No entanto, antes de se discutir a extensão de uma linguagem de programação, é necesssário identificar quais os requisitos que esta deve preencher para suportar eficientemente a descrição do hardware.

C.1. Modelação de Hardware

As linguagens de programação C, C++ e JAVA foram concebidas para desenvolver software, não possuindo o suporte necessário para descrever hardware de uma forma simples, eficiente e intuitiva. Para modelar hardware com uma das linguagens referidas acima é necessário dotá-las das seguintes capacidades [5, 6]:

- Concorrência e paralelismo – o hardware é por inerência paralelo enquanto os programas de software são executados sequencialmente. A concorrência é suportada nas HDLs através da introdução da noção de processo ou tarefa. Cada tarefa executa, conceptualmente, em paralelo com os seus pares;
- Sinais e portos – no caso do software, os processos concorrentes comunicam entre si usando primitivas do tipo memória partilhada, semáforos, regiões críticas, etc. Estas primitivas assumem que cada processo pode aceder facilmente ao estado interno dos restantes, o que não acontece no hardware. Os sinais são objectos que actuam como memória partilhada para comunicação entre processos. Os portos estabelecem o interface externo de um bloco de hardware e definem o tipo de operações realizáveis sobre o sinal a que está associado;
- Comportamento reactivo – o hardware pode ser modelado como um sistema reactivo, isto é, um sistema em permanente interacção com o ambiente que o rodeia. O conceito de comportamento reactivo é fundamental para modelar hardware a todos os níveis de abtracção. Nas HDLs a implementação de comportamento reactivo baseia-se na noção de evento, isto é, uma mudança de valor de um sinal ou porto. Assim, a descrição do hardware pode ser feita com um conjunto de tarefas em execução permanente que reagem continuamente a eventos ocorridos no ambiente com o qual o sistema interage;
- Tipos de dados específicos do hardware – ao contrário do que se passa no software, em que os tipos de dados fundamentais possuem um tamanho fixo, para modelar hardware é útil dispôr de tipos de dados de tamanho parametrizável, tais como quantidades inteiras com e sem sinal de precisão arbitrária, quantidades decimais em vírgula fixa e vectores de bits para representação de valores binários e lógicos standard (alta-impedância, saídas activas/passivas e valores indefinidos para representação de conflitos em barramentos).

C.2. Extensibilidade e Orientação por Objectos

Por não suportarem os mecanismos enumerados acima, as linguagens de programação tradicionais (C/C++/JAVA) não podem ser utilizadas directamente numa metodologia do tipo SLM que cubra todas as etapas do fluxo de projecto de um sistema digital complexo. Tal como já foi dito acima, uma das abordagens para resolver este problema consiste na definição de uma linguagem de programação completamente nova, capaz de lidar não só com os aspectos de desenvolvimento do software mas também com o projecto de hardware. No entanto, esta abordagem é pouco atractiva porque requer tempos de desenvolvimento e evolução consideráveis para a linguagem e respectivas ferramentas de suporte. Além disso, os processos de adopção e aprendizagem de uma nova linguagem costumam ser bastante morosos. Felizmente, o suporte para descrever hardware pode também ser adicionado a uma das linguagens tradicionalmente usadas no desenvolvimento de software. Esta extensão pode ser feita de duas maneiras:

- Adicionando novas construções e funcionalidades que necessitem de estender a sintaxe da linguagem base e consequentemente requerem a construção de ferramentas de desenvolvimento específicas, tais como compiladores, simuladores, sintetizadores e depuradores capazes de manipular as novas construções da linguagem. Exemplos desta abordagem são o HandelC [7] e as extensões propostas para tornar as HDLs tradicionais orientadas por objectos;
- Tirando partido das capacidades de extensão disponíveis nas linguagens de programação orientadas por objectos como o C++ ou o JAVA, as quais suportam a definição de novos tipos de dados e respectivas operações. A extensão da linguagem é feita através de uma ou várias bibliotecas de classes que fornecem um conjunto de abstracções úteis para modelar os aspectos do projecto de hardware de um sistema, tais como tarefas, sinais, portos, eventos, tipos de dados e operações realizáveis sobre cada um destes objectos. Este método tem a vantagem de permitir reutilizar algumas das ferramentas disponíveis para a linguagem base, tais como compiladores, depuradores e ambientes integrados. As linguagens SystemC [8], OCAPI [9], Cynlib [10] são implementadas usando esta abordagem. No entanto, é importante referir que no caso de se pretender sintetizar hardware directamente da descrição em C/C++/JAVA continua a ser necessária a construção das respectivas ferramentas [6].

O grau de sofisticação das bibliotecas de classes, referidas no último ponto pode ser bastante variado. Numa abordagem mais elementar pode disponibilizar apenas um conjunto de abstracções úteis para modelar hardware digital ao nível lógico ou RTL e que implementam os mecanismos normalmente disponíveis nas HDLs. Estas abstracções em conjunto com as funcionalidades da linguagem base permitem realizar manual e iterativamente o refinamento e validação do modelo do sistema. Numa abordagem mais elaborada, as bibliotecas podem também disponibilizar mecanismos de comunicação e sincronização que facilitem o refinamento progressivo e sistemático do modelo inicial do sistema. Para facilitar este processo, as bibliotecas devem também possuir informação sobre como um mecanismo de comunicação ou sincronização abstracto deve ser mapeado, para efeitos de implementação, em primitivas de baixo-nível. As bibliotecas podem também possuir implementações de elementos computacionais complexos que facilitem a reutilização da IP. Para melhorar a qualidade final do sistema, os componentes disponibilizados devem possuir informação sobre a sua implementação numa dada tecnologia alvo. Para facilitar a síntese de hardware a partir de uma descrição em C++/JAVA pode definir-se um subconjunto sintetizável da linguagem a partir do qual a geração de hardware possa ser realizada automaticamente por ferramentas construídas para o efeito.

III. DESCRIÇÃO DA LINGUAGEM

A motivação para desenvolver a linguagem EaSys resultou da necessidade de conceber uma plataforma de investigação aberta, robusta, modular, flexível e extensível para especificar, verificar e sintetisar sistemas complexos compostos por uma combinação arbitrária de componentes de hardware e de software. Esta linguagem é uma extensão à linguagem de programação orientada por objectos C++, à qual foram adicionados, através de uma biblioteca de classes, os mecanismos enumerados na secção anterior, de forma a torná-la adequada não só para desenvolver software mas também para descrever hardware. Além das classes que representam conceitos do hardware, tais como módulos, tarefas, sinais e portos, a linguagem EaSys possui um núcleo de simulação baseado em ciclos, que deve ser associado ao modelo do sistema a projectar. Este núcleo disponibiliza um interface para controlo da execução do modelo do sistema durante a sua simulação.

Numa primeira fase, a linguagem EaSys adiciona ao C++ apenas as construções e mecanismos que implementam os subconjuntos sintetizáveis das HDLs tradicionais. Há medida que a linguagem for evoluindo, vão sendo disponibilizadas novas capacidades que possibitam a construção de modelos a níveis de abstracções mais elevados. É concensual que para lidar eficientemente com a complexidade crescente dos sistemas digitais, um projecto deve ser realizado usando uma abordagem do tipo decomposição descendente (*top-down decomposition*), em que um modelo de alto-nível é sucessivamente validado e refinado até à implementação. Por outro lado, a concepção das abstracções usadas em modelos de alto-nível deve seguir uma abordagem do tipo montagem ascendente (*bottom-up assembly*) de forma a garantir que possuem implementação física.

A especificação e simulação de um sistema descrito na linguagem EaSys são feitas apenas com ferramentas standard de desenvolvimento em C++, tais como editores de texto, compiladores, depuradores e ambientes integrados. Isto é possível porque todos os mecanismos da linguagem foram implementados através de definição de classes e sem estender a sintaxe da linguagem C++. No seu estado actual, a linguagem EaSys pode ser utilizada nas etapas iniciais de uma metodologia do tipo SLM. A síntese directa de hardware a partir de um modelo escrito em EaSys ainda não é suportada porque requer a construção de ferramentas capazes de interpretar a semântica das operações realizadas sobre os objectos da linguagem.

A plataforma de desenvolvimento da linguagem EaSys foi o Microsoft Visual Studio 6.0 sobre o sistema operativo Microsoft Windows 2000. No entanto, para facilitar a portabilidade para outras plataformas (e.g. Linux) houve o cuidado de usar sempre que possível as construções e as bibliotecas standard do C++. As poucas excepções a esta regra são perfeitamente localizadas e controladas por directivas de compilação condicional dependentes da plataforma. As relações de herança das classes que constituem o núcleo da linguagem EaSys estão representadas na Figura 3. Todas as classes estão implementadas na versão actual da linguagem, à excepção da *CEaSysThread* e *CEaSysBus*. As classes parametrizáveis são implementadas usando *templates* do C++. As classes da Figura 3 podem ser divididas nos seguintes grupos funcionais:

- Serviços base
  - *CEaSysObject*
- Eventos
  - *CEaSysEventBase*
- Módulos
  - *CEaSysModule*
- Tarefas
  - *CEaSysTask*
  - *CEaSysRoutine*
  - *CEaSysThread*
- Sinais
  - *CEaSysSignalBase*
  - *CEaSysSignal*
  - *CEaSysBus*
- Portos
  - *CEaSysPortBase*
  - *CEaSysPort*
  - *CEaSysInPort*
  - *CEaSysOutPort*
  - *CEaSysInOutPort*
- Núcleo de simulação
  - *CEaSysKernel*

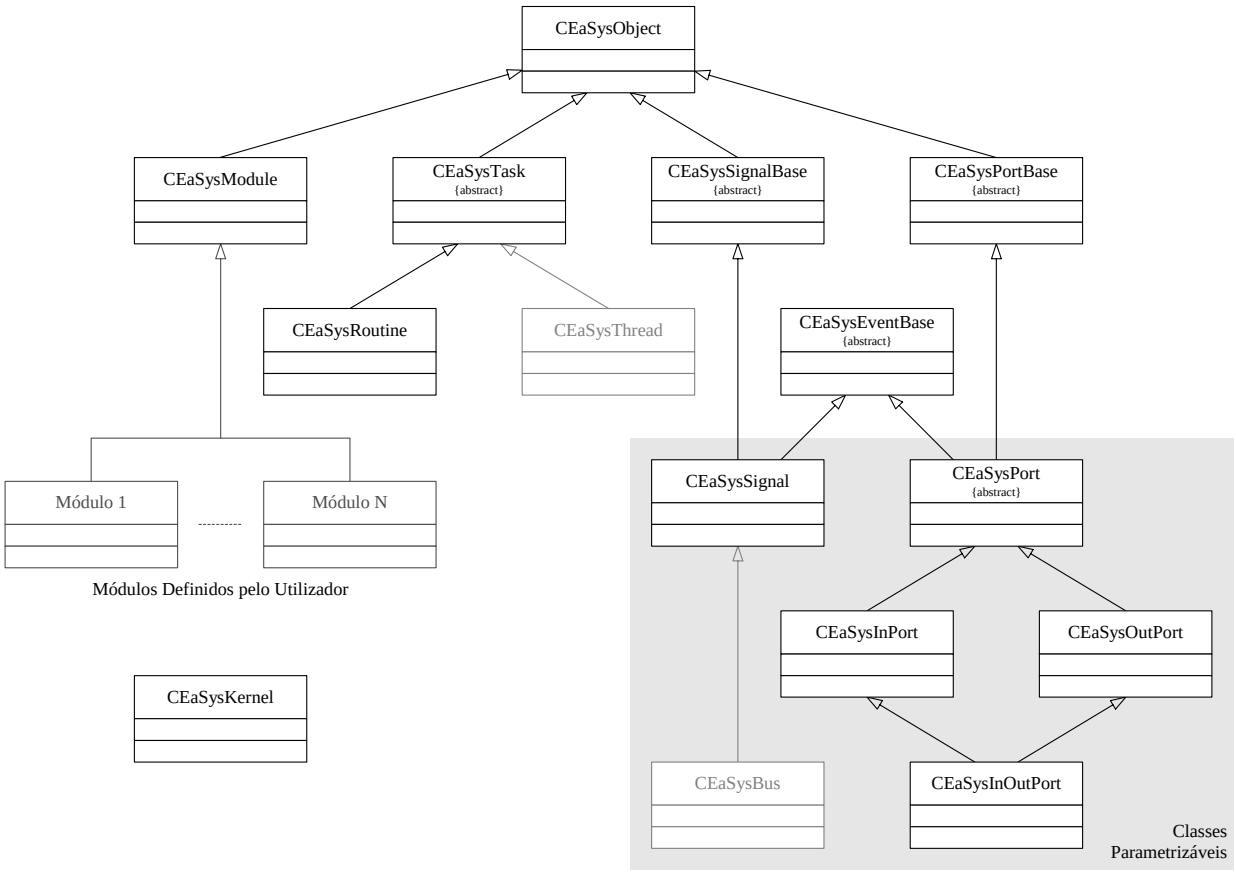

Figura 3 - Relações de herança entre as classes que implementam o núcleo da linguagem Easys.

Como veremos ao longo da descrição da linguagem, o utilizador pode usar quer os verdadeiros nomes das classes e a síntaxe normal do C++ para instanciar e manipular os respectivos objectos, quer as macros disponibilizadas para o efeito. A segunda abordagem apesar de mais restritiva é recomendável para uma utilização corrente, pois simplifica a realização das operações sobre os objectos através do encapsulamento de alguns dos detalhes associados.

As subsecções seguintes descrevem cada um dos elementos da linguagem. Esta discussão será acompanhada da apresentação progressiva de um exemplo de uma porta lógica complexa AOI representada na Figura 4. A descrição em EaSys do circuito da Figura 4 é apresentada na Figura 5. Este exemplo é interessante porque permite ilustrar as capacidades de descrição estrutural e comportamental da linguagem.

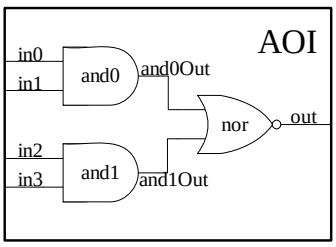

Figura 4 - Esquema de uma porta lógica complexa AOI.

```
module(AOI)
{
   iport<bool> in0;
   iport<bool> in1;
   iport<bool> in2;
   iport<bool> in3;
   oport<bool> out;

   signal<bool> and0Out, and1Out;

   method(nor)
   {
      out = !(and0Out || and1Out);
   }

   constructor(AOI)
   {
      instance(AND2, and0);
      and0->in0(in0);
      and0->in1(in1);
      and0->out(and0Out);

      instance(AND2, and1);
      and1->in0(in2);
      and1->in1(in3);
      and1->out(and1Out);

      routine(nor);
      nor->sensitive(and0Out);
      nor->sensitive(and1Out);
   }
};
```

Figura 5 - Descrição da porta lógica AOI em EaSys.

A. Serviços Base

A classe *CEaSysObject* serve de base à maioria das classes que constituem o núcleo da linguagem EaSys. Os objectos do tipo módulo, tarefa, sinal e porto devem possuir uma identificação que consiste num nome e na sua posição hierárquica dentro do sistema representada por uma referência para o objecto pai (onde é feita a instanciação). A classe *CEaSysObject* implementa os mecanismos para armazenar e manipular esta informação e disponibiliza-os às suas classes derivadas. Esta classe não pode ser instanciada directamente porque o seu construtor só está disponível por derivação.

B. Eventos

Diz-se que ocorreu um evento num objecto quando este muda de valor. Os eventos podem ocorrer em objectos do tipo sinal ou porto. Um evento pode ser positivo ou negativo dependendo do sentido da variação do valor. Os eventos são representados no núcleo da linguagem EaSys pela classe *CEaSysEventBase*. Por ser abstracta, esta classe não pode ser instanciada e define apenas um interface que os objectos, onde ocorrem eventos, devem implementar. Assim, as classes *CEaSysSignal* e *CEaSysPort* que representam, respectivamente, os sinais e os portos da linguagem devem também ser derivadas da classe *CEaSysEventBase*. Desta forma é possível realizar invocações polimórficas, isto é, independentemente se o objecto é um sinal ou um porto e do respectivo tipo de dados. Na Tabela 1 são apresentadas as construções utilizadas para testar a ocorrência de eventos em sinais e portos. A segunda e a terceira coluna da tabela ilustram, respectivamente, as construções usadas no caso do teste ser efectuado directamente sobre o objecto ou através de um ponteiro para o objecto. Estas expressões retornam verdadeiro se o respectivo evento ocorreu, ou falso em caso contrário.

Tipo do Evento	Objecto	Ponteiro
Qualquer	`obj.event`	`pObj->event`
Positivo	`obj.posevent`	`pObj->posevent`
Negativo	`obj.negevent`	`pObj->negevent`

Tabela 1 - Construções usadas para testar a ocorrência de eventos em objectos do tipo sinal ou porto.

C. Módulos

Os módulos são os elementos básicos de decomposição estrutural da linguagem EaSys. Estes permitem dividir um sistema em blocos mais simples, logo mais fáceis de modelar, desenvolver, testar e integrar. No caso de projectos complexos permitem também fazer a sua distribuição entre os vários projectistas da equipa de desenvolvimento. Um módulo permite esconder a representação de dados e os algoritmos internos dos restantes módulos, isto é, promovem a separação entre o interface e a implementação dos componentes de um sistema. Isto possibilita o estabelecimento de interfaces públicos e bem definidos, o que facilita a interacção entre os diversos módulos do sistema, bem como a manutenção e introdução de modificações. Por exemplo, um projectista pode decidir alterar completamente a implementação interna do módulo. Se o seu interface externo e o comportamento permanecerem inalterados, os utilizadores do módulo não se apercebem das alterações internas, permitindo assim a realização de optimizações locais.

Um módulo é declarado com a palavra-chave *module*, tal como é ilustrado no exemplo seguinte:

```
module(AOI)
{
   ...
};
```

O identificador após a palavra-chave "*module*" é o nome do modulo, que neste caso é "AOI". Como veremos mais à frente, um módulo é na realidade uma classe em C++ derivada da classe "*CEaSysModule*" definida no núcleo da linguagem. A palavra-chave "*module*" é uma macro cujo parâmetro é o nome da classe que representa o módulo. Assim, em vez de se utilizar a macro "*module*", um módulo pode ser declarado usando a sintaxe usual do C++ para derivação de classes, ou seja:

```
class AOI : public CEaSysModule
{
   ...
};
```

A declaração de um módulo usando a macro "*module*" tem a vantagem de ser simultaneamente mais legível e mais concisa. Um módulo pode conter outros tipos de objectos, tais como portos, sinais internos, variáveis locais, tarefas e outros módulos. Além disso, todos os módulos devem possuir um construtor onde são declaradas as tarefas e/ou definida a sua estrutura interna. Em conjunto, estes objectos implementam a sua funcionalidade. O construtor de um módulo pode ser declarado da seguinte maneira:

```
module(AOI)
{
   ...
   constructor(AOI)
   {
      ...
   }
   ...
};
```

O identificador entre parêntesis a seguir à palavra-chave "*constructor*" é o nome da classe do módulo ao qual pertence o construtor. Este é invocado sempre que for criada uma instância desse módulo. No corpo do construtor são declaradas e/ou inicializadas as estruturas de dados (módulos, tarefas, sinais, portos e variáveis) utilizadas internamente. O número e tipo de parâmetros do construtor é arbitrário. No entanto, existe uma restrição: o construtor deve obrigatoriamente possuir dois parâmetros para receber durante a instanciação: o nome e uma

referência para o objecto onde o módulo é criado. A macro "*constructor*" existe precisamente para esconder estes detalhes, simplificando a declaração do construtor.

No corpo de um construtor podem ser instanciados outros módulos usando a palavra chave "*instance*" implementada também por uma macro, cujos parâmetros são o nome (mais precisamente o tipo) do módulo a intanciar e o nome do objecto:

```
module(AOI)
{
   ...
   constructor(AOI)
   {
      instance(AND2, and1);
      ...
   }
   ...
};
```

A macro "*constructor*"pode ser usada apenas no caso dos parâmetros do construtor serem os obrigatórios acima referidos. Sempre que se utilizar outros parâmetros é necessário escrever o protótipo completo do construtor e passar ao construtor da classe base os parâmetros obrigatórios. Neste caso, a instanciação de um módulo deve também ser feita sem recorrer à macro "*instance*". O exemplo seguinte ilustra esta situação:

```
module(AOI)
{
   ...
   AOI(type1 param1, ..., typeM paramM,
       const string& name,
       CEaSysModule* pParent) :
      CEaSysModule(name, pParent)
   {
      AND* and1= new AND(arg1, ..., argN,
                         "and1", this);
      ...
   }
   ...
};
```

O valor dos parâmetros obrigatórios referidos acima (nome e referência para o objecto pai) apesar de não afectarem o comportamento do sistema, são importantes, pois é deles que depende a construção da hierarquia do sistema e a visualização intuitiva dos resultados da simulação.

No núcleo da linguagem, a funcionalidade básica de um módulo é implementada pela classe *"CEaSysModule"*. É importante referir que esta classe destina-se apenas a servir de base às classes definidas pelo utilizador e que representam os módulos de um dado projecto ou biblioteca. Por este motivo a classe *"CEaSysModule"* não pode ser instanciada directamente.

D. Tarefas

A execução de uma aplicação de software num processador de uso geral consiste na execução sequencial das instruções que constituem o respectivo programa. As linguagens de programação tradicionais são bastante adequadas para desenvolver este tipo de aplicações, podendo também ser utilizadas facilmente para modelar o comportamento sequencial de um sistema. No entanto, os sistemas electrónicos são intrinsecamente paralelos, isto é, são compostos por vários elementos que operam em simultâneo. A modelação destas actividades paralelas com uma linguagem sequencial pode revelar-se uma tarefa bastante complexa, pouco intuitiva e sujeita a erros.

As tarefas são o elemento básico de processamento da linguagem EaSys, onde são realizadas as operações que emulam o funcionamento em paralelo dos vários elementos do sistema modelado.

As tarefas possuem listas de sensibilidade onde são especificados os eventos responsáveis pela sua activação. Uma tarefa é activada, ou seja, a sua execução é iniciada ou retomada, sempre que o valor de um sinal ou porto especificado na sua lista de sensibilidade mudar. Por defeito, isto é, na ausência de uma lista de sensibilidade, uma tarefa é executada em todos os ciclos de simulação.

As tarefas não podem ser hierárquicas, isto é, uma tarefa não pode invocar outra directamente. No entanto, uma tarefa pode provocar a activação de outra(s) através da atribuição de um valor a um sinal pertencente à(s) respectiva(s) lista(s) de sensibilidade. Além disso, as tarefas podem invocar métodos e funções ordinárias.

Existem vários tipos de tarefas. A versão actual da linguagem apenas suporta tarefas do tipo rotina. Além destas, estão também previstas tarefas do tipo *thread* que ao contrário das primeiras possuem um contexto de execução independente.

D.1. Rotinas

Este é o único tipo de tarefa suportado actualmente e que corresponde em termos funcionais aos processos da linguagem VHDL. Uma rotina é implementada por um método cuja execução é iniciada sempre que ocorrer um evento ao qual a tarefa seja sensível. Como uma rotina corre no mesmo contexto de execução da entidade invocadora (o núcleo de simulação da linguagem) é importante que, sempre que for activada, execute completamente e retorne o controlo ao núcleo de simulação. Por outras palavras, uma rotina não pode suspender a sua execução nem possuir ciclos infinitos e executa conceptualmente em tempo zero.

D.2. Contextos de execução independentes – *Threads*

Este tipo de tarefa ainda não é suportado na versão actual da linguagem. Ao contrário de uma tarefa do tipo rotina, uma do tipo *thread* possui o seu próprio contexto de execução. Isto permite que uma tarefa deste tipo possa suspender a sua execução num dado ponto. Mais tarde, quando for novamente activada, a execução é retomada a partir do ponto onde foi suspensa. Uma das aplicações das tarefas do tipo *thread* é a descrição de sistemas sequenciais sem definir explicitamente os estados e portanto a um nível de abstracção mais elevado do que o proporcionado pelo modelo de máquina de estados finitos.

Em qualquer dos tipos de tarefa (rotina ou *thread*), o método que a implementa pertence ao módulo onde esta foi declarada. Quando é instanciado um objecto do tipo tarefa é feito o seu registo no núcleo de simulação para que a sua activação seja feita sempre que ocorrer um evento ao qual esta seja sensível. Por esta razão uma tarefa não é apenas uma rotina mas um objecto registado no núcleo de simulação e que internamente possui uma lista de sensibilidade e uma referência para o método que implementa as suas operações.

O exemplo seguinte ilustra a declaração e a definição de uma tarefa do tipo rotina chamada "nor".

```
module(AOI)
{
   ...
   method(nor)
   {
      out = !(and0Out || and1Out);
   }
   ...
   constructor(AOI)
   {
      ...
      routine(nor);
      nor->sensitive(and0Out, any);
      nor->sensitive(and1Out, any);
      ...
   }
   ...
};
```

A declaração de uma tarefa do tipo rotina é feita através da palavra-chave "*routine*" seguida do respectivo nome entre parêntesis. Esta palavra-chave é na realidade uma macro cujo parâmetro é o nome da tarefa. A sua implementação é feita no método com o mesmo nome e precedido pela palavra-chave "*method*".

A especificação da lista de sensibilidade da tarefa é também ilustrada no exemplo anterior. Neste caso, a tarefa é sensível a qualquer evento ocorrido nos sinais *"and0Out"* e *"and1Out"*. Se em vez disso se pretender que a tarefa seja sensível apenas a eventos positivos ocorridos em *"and1Out"* e a eventos negativos de *"and2Out"*, a lista de sensibilidade deveria ser definida da seguinte maneira:

```
nor->sensitive(and0Out, pos);
nor->sensitive(and1Out, neg);
```

No caso de uma tarefa ser sensível a qualquer evento de um sinal, como acontece no exemplo apresentado, o especificador "any" pode ser omitido:

```
nor->sensitive(and0Out);
nor->sensitive(and1Out);
```

A declaração de uma tarefa e a especificação da sua lista de sensibilidade devem ser feitas no corpo do construtor do respectivo módulo.

A implementação das tarefas no núcleo da linguagem EaSys consiste nas seguintes três classes:

- *CEaSysTask* – implementa a funcionalidade base de uma tarefa. Independentemente do tipo, uma tarefa deve possuir uma lista de sensibilidade onde são especificados os eventos responsáveis pela sua activação. É através desta classe que o núcleo da linguagem verifica se uma tarefa deve ou não ser activada num dado ciclo de simulação. A forma como é feita a activação é considerada específica de cada tipo de tarefa, pelo que não está definida nesta classe, devendo ser implementada nas classes derivadas desta;
- *CEaSysRoutine* – esta classe implementa os aspectos específicos de uma tarefa do tipo rotina. É através desta classe que o núcleo de simulação invoca o método que implementa a tarefa, ficando a aguardar até este concluir a sua execução;
- *CEaSysThread* – esta classe implementa os aspectos específicos das tarefas do tipo *thread*. Neste caso o controlo de execução da tarefa é mais complexo porque envolve a verificação da suspensão de execução da tarefa e a sua reactivação. Esta classe ainda não está implementada.

E. Sinais

Os sinais da linguagem EaSys são objectos especiais usados para armazenar informação e interligar portos. É também nos sinais que ocorrem os eventos responsáveis pela activação das tarefas.

Os sinais são, do ponto de vista da sintaxe, análogos às variáveis ordinárias do C++. Um sinal possui sempre um tipo de dados associado que deve ser estabelecido no momento da sua instanciação. Este, tanto pode ser um dos tipos de dados predefinidos da linguagem C++, como um tipo de dados abstracto definido pelo utilizador. No segundo caso, existem alguns requisitos que devem ser cumpridos. Em particular, os operadores relacionais (==, !=, <, >) devem estar definidos, pois são usados pelo núcleo da linguagem na detecção de eventos.

A principal diferença entre um objecto ou variável ordinária do C++ e um sinal reside na semântica das operações de leitura e escrita. Enquanto no primeiro caso uma operação de leitura devolve sempre o último valor atribuído, num sinal as alterações produzidas por uma operação de atribuição ou escrita só se tornam vísiveis (i.e. devolvidas pelas operações de leitura) quando todas as tarefas activadas num dado ciclo de simulação tiverem executado. Isto permite simular a execução paralela das tarefas e detectar a ocorrência de eventos. Por este motivo, ao contrário de uma variável ou objecto ordinário, um sinal pode ser utilizado na lista de sensibilidade de uma tarefa.

No caso de um sinal ser utilizado para interligar os portos de diferentes módulos, a informação é transferida entre portos como se estes estivessem ligados directamente.

A instanciação de um sinal consiste em três elementos: a palavra-chave "signal", seguida do tipo de dados associado ao sinal especificado entre parêntesis <...> (i.e. o parâmetro da classe) e, finalmente, o nome do objecto. Na seguinte declaração são instanciados dois sinais do tipo predefinido "bool":

```
module(AOI)
{
   ...
   signal<bool> and0Out, and1Out;
   ...
};
```

Um sinal do tipo abstracto "EState" seria declarado da seguinte forma:

```
signal<EState> currentState;
```

A palavra-chave "signal" é na realidade uma macro que cria um pseudónimo para a classe *CEaSysSignal* do núcleo da linguagem.

Para simplificar a realização de operações sobre sinais, foi redefinida, para os objectos deste tipo, a maioria dos operadores da linguagem C++. A listagem dos operadores redefinidos encontra-se na Tabela 2. Assim, o mesmo operador pode ser aplicado quer a objectos do tipo sinal, quer a objectos do tipo de dados associado ao sinal. Além disso, pode-se na mesma expressão misturar objectos de ambos os tipos. É importante referir que os operadores da Tabela 2 só estão efectivamente disponíveis se estiverem definidos para o tipo de dados associado ao sinal. Isto acontece com os tipos predefinidos do C++, mas não com os tipos de dados abstractos definidos pelo utilizador, onde é necessário definir os operadores que se pretende utilizar. Uma discussão sobre a implementação dos operadores para os tipos de dados definidos pelo utilizador está fora do âmbito deste artigo. Em [11] é feita uma descrição detalhada sobre este assunto.

A implementação dos objectos do tipo sinal no núcleo da linguagem EaSys está distribuída por três classes:

- *CEaSysSignalBase* – esta classe serve de base à parametrizável *CEaSysSignal* possibilitando a utilização dos mecanismos de polimorfismo disponíveis no C++. Esta classe é necessária, uma vez que não existe qualquer relação entre as diferentes especializações de uma classe parametrizável. Além de não ser instanciável directamente, a sua existência é completamente transparente ao utilizador;
- *CEaSysSignal* – esta classe é parametrizável, implementando, entre outras coisas, todos os operadores listados na Tabela 2. As operações definidas nesta classe são independentes do tipo de dados associado ao sinal que actua como parâmetro da classe;
- *CEaSysBus* – esta classe ainda não está implementada. Como o próprio nome indica, destina-se a fornecer uma abstracção para objectos do tipo barramento. Ao contrário dos sinais convencionais implementados pela classe anterior, um barramento pode ser controlado simultaneamente por várias saídas, pelo que o tipo de dados que lhe está associado deve representar os valores lógicos standard ('0', '1', 'L', 'H', 'Z', 'X') e necessita de mecanismos para resolver eventuais conflitos.

	Símbolo	***Nome***	***Tipo***
Leitura	(type)	Conversão	-
	==	Igualdade	Binário
	!=	Desigualdade	Binário
	<	Menor que	Binário
	>	Maior que	Binário
	<=	Menor ou igual que	Binário
	>=	Maior ou igual que	Binário
	!	NOT lógico	Unário
	&&	AND lógico	Binário
	||	OR lógico	Binário
	~	NOT bit a bit	Unário
	&	AND bit a bit	Binário
	|	OR bit a bit	Binário
	^	EXOR bit a bit	Binário
	<<	Deslocamento para a esquerda	Binário
	>>	Deslocamento para a direita	Binário
	+	Adição unária	Unário
	-	Subtracção unária	Unário
	+	Adição	Binário
	-	Subtracção	Binário
	*	Multiplicação	Binário
	/	Divisão	Binário
	%	Resto de divisão inteira	Binário
Escrita	++	Pré/Pós incremento	Unário
	--	Pré/Pós decremento	Unário
	=	Atribuição	Binário
	&=	AND bit a bit/Atribuição	Binário
	|=	OR bit a bit/Atribuição	Binário
	^=	EXOR bit a bit/Atribuição	Binário
	<<=	Deslocamento para a esquerda/Atribuição	Binário
	>>=	Deslocamento para a direita/Atribuição	Binário
	+=	Adição/Atribuição	Binário
	-=	Subtracção/Atribuição	Binário
	*=	Multiplicação/Atribuição	Binário
	/=	Divisão/Atribuição	Binário
	%=	Resto de divisão inteira/ Atribuição	Binário
	[]	Indíce de vector	-
	,	Vírgula	-

Tabela 2 - Listagem dos operadores definidos na classe parametrizável *CEaSysSignal* que define os objectos do tipo sinal da linguagem EaSys.

F. Portos

Os portos definem o interface de um módulo com o exterior, através do qual é transferida infomação. Um porto pode ser usado na lista de sensibilidade de uma tarefa, pelo que também pode disparar acções no interior do módulo. Um porto pode funcionar num dos seguintes três modos:

- Entrada - transfere dados do exterior para o módulo;
- Saída - transfere dados do módulo para o exterior;
- Entrada/Saída - pode transferir dados em ambos os sentidos dependendo da operação do módulo em cada instante.

Um porto pode estar ligado a outro porto ou a um sinal. Quando dois portos estão interligados através de um sinal, a informação é transferida entre os dois como se estes estivessem ligados directamente. Quando é realizada uma operação de escrita num porto o novo valor é atribuído ao sinal a que está directa ou indirectamente ligado. Numa operação de leitura é devolvido o valor do referido sinal.

Tal como acontece com os sinais, os valores atribuídos aos portos só se tornam visíveis através de uma operação de leitura após a conclusão do ciclo de simulação.

Na Tabela 3 estão indicadas as conexões permitidas entre os portos dos vários modos e sinais. A mesma informação está também representada graficamente na Figura 6. Na Figura 7 é ilustrado um exemplo complexo de uma hierarquia de módulos e interligação entre os seus sinais e portos. A verificação das ligações entre portos e sinais é feita a dois níveis: sintático e semântico. A verificação sintática é feita segundo as regras de compatibilidade apresentadas na Tabela 3 e no momento da compilação do modelo. A verificação semântica consiste na análise do número de portos ligados a um sinal ou porto e é realizada durante a inicialização do modelo e dinamicamente durante a sua execução. Todas as conexões da Figura 7 estão correctas, à excepção das marcadas com o símbolo "**×**" porque os portos de entrada e bidireccionais não podem ser deixados em aberto e os portos de saída e bidireccionais não podem ser controlados por mais do que uma saída ou sinal.

		Tipo do porto		
		I	*O*	*I/O*
Objecto do nível seguinte	*I (porto)*	✓	×	×
	O (porto)	✓	✓	×
	I/O (porto)	✓	✓	✓
	S (sinal)	✓	✓	✓

Tabela 3 - Ligações permitidas entre os portos de um módulo e os objectos (sinais ou portos) do nível hierárquico seguinte.

À semelhança dos sinais, a instanciação de um porto também consiste em três elementos: uma palavra reservada que depende do modo de funcionamento do porto, um tipo de dados e, finalmente, o nome do objecto. O exemplo seguinte ilustra a declaração dos portos para o exemplo da porta lógica complexa AOI:

```
module(AOI)
{
   iport<bool> in0;
   iport<bool> in1;
   iport<bool> in2;
   iport<bool> in3;
   oport<bool> out;
   ...
};
```

As palavras-chave usadas para instanciar objectos do tipo porto depedem do seu modo de operação e são: *iport* (porto de entrada), *oport* (porto de saída) e *ioport* (porto de entrada/saída – bidireccional). Os portos de um módulo não são declarados como parâmetros formais do seu construtor, porque isso representaria, em termos de escrita, uma sobrecarga considerável.

A ligação dos portos de um módulo deve ser feita logo após a sua instanciação e antes de iniciada a simulação. Assim, o sítio mais correcto para o fazer é o construtor do módulo, uma vez que é onde se define a sua estrutura. O

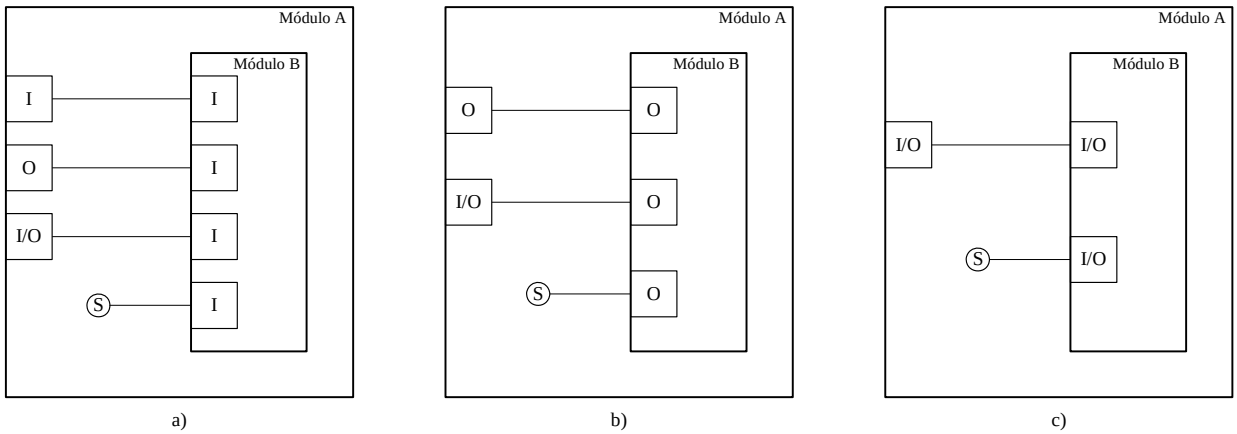

Figura 6 - Representação gráfica das ligações permitidas entre os portos de um módulo e os objectos (sinais ou portos) do nível hierárquico seguinte no caso de um: a) porto de entrada; b) porto de saída; c) porto bidireccional.

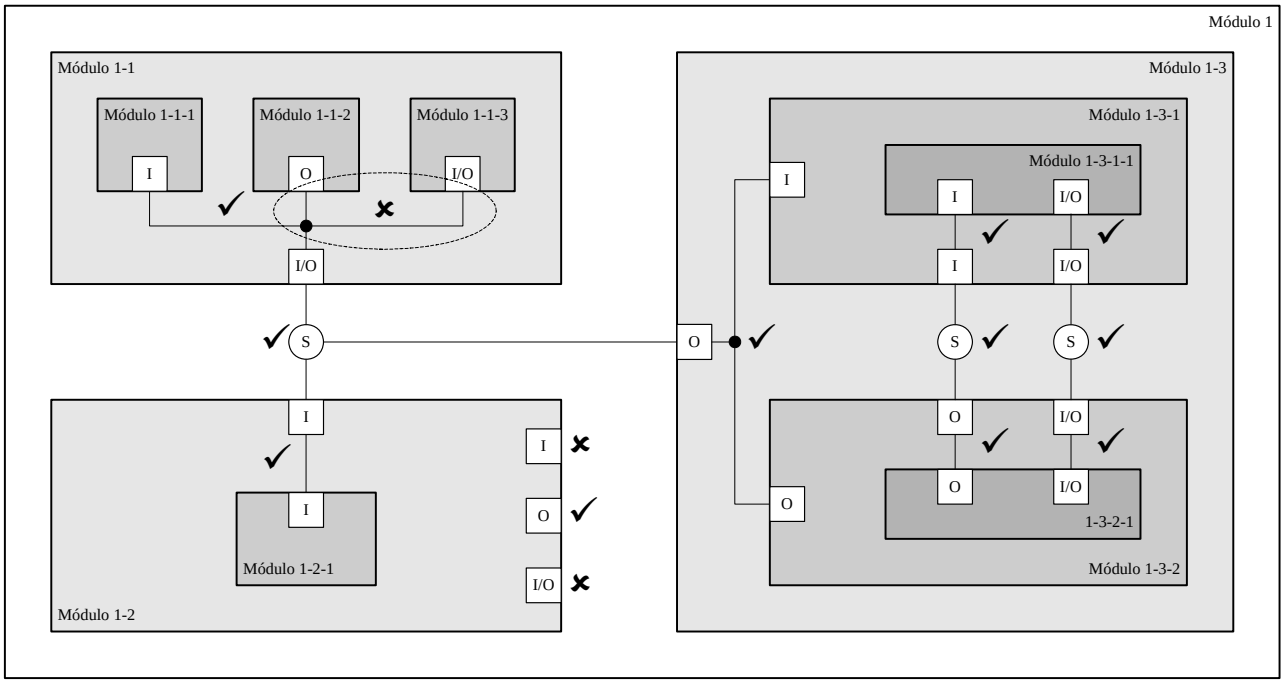

Figura 7 - Exemplo de ligações entre sinais e portos de diferentes modos.

exemplo seguinte ilustra a ligação de duas portas lógicas AND de duas entradas, quer aos portos de entrada do módulo AOI, quer a dois sinais internos.

```
module(AOI)
{
   ...
   constructor(AOI)
   {
      instance(AND2, and0);
      and0->in0(in0);
      and0->in1(in1);
      and0->out(and0Out);

      instance(AND2, and1);
      and1->in0(in2);
      and1->in1(in3);
      and1->out(and1Out);
      ...
   }
};
```

Podem também ser instanciados vectores de portos usando a sintaxe usual de declaração de vectores do C++. O exemplo seguinte ilustra esta facilidade, substituindo as quatro entradas do módulo AOI por um vector de quatro portos:

```
iport<bool> in[4];
```

Um vector de portos bidireccionais pode ser declarado da seguinte maneira:

```
ioport<bool> data[16];
```

Cada um dos portos do vector deve ser individualmente interligado e se necessário adicionado separadamente à lista de sensibilidade das tarefas que deles dependam.

A implementação dos portos na linguagem EaSys consiste nas seguintes cinco classes:

- *CEaSysPortBase* – as razões da existência desta classe são análogas às da classe *CEaSysSignalBase*. Esta classe também é transparente para o utilizador;
- *CEaSysPort* – à semelhança da classe *CEaSysSignal*, a classe *CEaSysPort* também é parametrizável, permitindo instanciar e efectuar operações sobre portos independentemente do tipo de dados subjacente. Nesta classe são implementadas as funcionalidades base de um porto que não dependam do seu modo de funcionamento (entrada, saída, entrada/saída);
- *CEaSysInPort* – esta classe é derivada da *CEaSysPort* e implementa os aspectos específicos de um porto de entrada, o qual deve obrigatoriamente estar ligado a outro porto ou sinal e sobre o qual só podem ser realizadas operações de leitura;
- *CEaSysOutPort* – esta classe também é derivada da *CEaSysPort* e implementa a funcionalidade de um porto de saída. Ao contrário do que acontece com as linguagens de descrição de hardware tradicionais, sobre um porto de saída além de poderem ser efectuadas operações de escrita, também podem ser realizadas operações de leitura. Esta alteração além de não introduzir efeitos indesejados no comportamento do circuito, simplifica a interligação de módulos, dispensando a declaração de sinais auxiliares sempre que a saída de um módulo esteja ligada tanto a entradas como a saídas de objectos nele instanciados. Na próxima secção será ilustrada uma aplicação prática desta facilidade. Um porto de saída pode ser ligado a um sinal ou a outro porto de saída ou bidireccional, mas também pode ser deixado em aberto;
- *CEaSysInOutPort* – esta classe representa um porto bidirecional (entrada/saída) e é derivada das classes

CEaSysInPort e *CEaSysOutPort*. Sobre um objecto desta classe é possível efectuar operações quer de leitura quer de escrita. Um porto bidireccional tem de estar ligado a um porto do mesmo tipo ou a um sinal.

G. Núcleo de Simulação

O núcleo de simulação da linguagem EaSys é baseado em ciclos e reune um conjunto de estruturas de dados e funções responsáveis pelo controlo da simulação. A sua implementação é feita na classe *CEaSysKernel*, que disponibiliza os seguintes serviços:

- Verificação semântica do modelo do sistema;
- Detecção de eventos;
- Escalonamento das tarefas;
- Construção da hierarquia do sistema;
- Visualização dos resultados da simulação.

A classe *CEaSysKernel* não é instanciável e todos os seus métodos e atributos são estáticos para que sejam únicos dentro de cada modelo de simulação. A maior parte do métodos disponibilizados por esta classe destinam-se a ser usados pelas classes do núcleo da linguagem. Por exemplo, todos os objectos do tipo módulo, tarefa, sinal e porto são registados no núcleo de simulação durante a sua instanciação. Na versão actual da linguagem, a interacção entre a aplicação de teste de um modelo e o núcleo de simulação faz-se através dos seguintes métodos:

- `EaSys::Initialize()` – Efectua a inicialização do núcleo, devendo ser invocada pelo programa de teste logo após a instanciação do modelo a simular. Internamente, inicializa as estruturas de dados do núcleo e verifica o modelo do ponto de vista semântico;
- `EaSys::Cycle()` – Executa um dado número de ciclos de simulação. Internamente, detecta a ocorrência de eventos nos sinais, verifica quais as tarefas que devem executar e no final de cada ciclo de simulação actualiza os valores dos sinais e dos portos.

Mais à frente será descrito o processo de compilação e mostrado um exemplo de uma aplicação de teste do modelo da porta lógica complexa AOI apresentada nesta secção.

IV. EXEMPLOS

Para complementar a descrição da linguagem efectuada na secção anterior e ilustrar as potencialidades e as construções mais comuns da linguagem EaSys vão ser agora apresentadas mais algumas descrições de componentes digitais comuns. Os exemplos escolhidos para este efeito foram os seguintes:

- Porta lógica NAND de duas entradas;
- Flip-flop tipo D com entrada de inicialização assíncrona;
- Latch SR com portas lógicas NAND;
- Somador de números inteiros;
- Máquina de estados finitos.

É importante referir que a combinação das contruções e mecanismos do C++ (e.g. parametrização das classes) com as funcionalidades proporcionadas pela linguagem desenvolvida, tais como a especificação de concorrência e as abstracções de entidades existentes no hardware possibilita a construção de bibliotecas de componentes genéricos e flexíveis. Como veremos no final deste artigo, este é um dos possíveis pontos de trabalho futuro.

A. Porta lógica NAND de duas entradas

Uma porta lógica NAND de duas entradas é provavelmente um dos componentes digitais mais simples que pode ser descrito com esta linguagem. O módulo que descreve um componente deste tipo possui apenas duas entradas (in0, in1) e uma saída (out) (ver Figura 8).

O processamento é efectuado numa tarefa do tipo rotina, que é sensível a qualquer evento que ocorra numa das entradas. No corpo do método que implementa a tarefa é especificada a função booleana da porta lógica.

```
module(NAND2)
{
   iport<bool> in0;
   iport<bool> in1;
   oport<bool> out;

   method(update)
   {
      out = !(in0 && in1);
   }

   constructor(NAND2)
   {
      routine(update);
      update->sensitive(in0);
      update->sensitive(in1);
   }
};
```

Figura 8 - Descrição em EaSys de uma porta lógica NAND de 2 entradas.

B. Flip-flop tipo D com entrada de inicialização assíncrona

Um flip-flop tipo D com entrada de inicialização possui 4 portos (Figura 9):

- A entrada de dados (d);
- A entrada de sincronização (clk);
- A entrada de inicialização (clr);
- A saída de dados (q).

A actualização da saída é feita no corpo do método "*update*" que implementa uma tarefa do tipo rotina e é sensível ao flancos ascendentes (i.e. eventos positivos) da entrada de sincronização e a qualquer evento da entrada de inicialização.

```
module(FFDC)
{
   iport<bool> d;
   iport<bool> clk;
   iport<bool> clr;
   oport<bool> q;

   method(update)
   {
      if (clr)
      {
         q = false;
      }
      else if (clk.posevent)
      {
         q = d;
      }
   }

   constructor(FFDC)
   {
      routine(update);
      update->sensitive(clk, pos);
      update->sensitive(clr);
   }
};
```

Figura 9 - Descrição em EaSys de um flip-flop tipo D com entrada de inicialização assíncrona.

C. Latch SR com portas lógicas NAND

Nos dois exemplos apresentados acima, a operação dos módulos era descrita comportamentalmente através de tarefas e listas de sensibilidade. A latch SR é um exemplo de um módulo descrito de forma estrutural, isto é, através da instanciação e interligação de outros componentes. No caso concreto da latch SR os componentes utilizados são duas portas lógicas NAND de duas entradas cuja descrição já foi apresentada na Figura 8. A sua instanciação e interligação é efectuada no corpo do construtor (ver Figura 10). Neste caso não é necessário declarar nenhuma tarefa pois todo o processamento é realizado no interior dos componentes instanciados.

```
module(LatchSR)
{
   iport<bool> n_s;
   iport<bool> n_r;
   oport<bool> q;
   oport<bool> n_q;

   constructor(LatchSR)
   {
      instance(NAND2, gate0);
      gate0->i0(n_s);
      gate0->i1(n_q);
      gate0->o(q);

      instance(NAND2, gate1);
      gate1->i0(n_r);
      gate1->i1(q);
      gate1->o(n_q);
   }
};
```

Figura 10 - Descrição em EaSys de uma Latch SR.

D. Somador de números inteiros

O módulo somador da Figura 11 é semelhante à porta lógica apresentada na Figura 8, tendo sido escolhido apenas para ilustrar a realização de operações aritméticas.

```
module(Adder)
{
   iport<int> in0;
   iport<int> in1;
   oport<int> out;

   method(update)
   {
      out = in0 + in1;
   }

   constructor(Adder)
   {
      routine(update);
      update->sensitive(in0);
      update->sensitive(in1);
   }
};
```

Figura 11 - Descrição em EaSys de uma somador de quantidades inteiras.

E. Máquina de estados finitos

O módulo escolhido para concluir a apresentação desta sequência de exemplos de descrições foi uma máquina de estados finitos (*Finite State Machine – FSM*). Uma FSM é um modelo usado para descrever o comportamento de circuitos sequenciais e pode ser representado de forma gráfica ou textual. Apesar das suas limitações, uma das representações mais usuais é o diagrama de transição de estados (*State Transition Diagram – STD*). Na Figura 12 encontra-se o STD da FSM utilizada neste exemplo. Esta FSM segue o modelo de Moore, isto é, as saídas dependem apenas do estado actual, e possui: duas entradas (in0; in1), duas saídas (out0; out1) e quatro estados (S0; S1; S2; S3).

A forma de modelar uma FSM com a linguagem EaSys é em tudo idêntica à utilizada nas HDLs (ver Figura 13). Além da declaração dos portos, a descrição de uma FSM é constituída normalmente pelos seguintes elementos:

- Enumeração dos estados e definição do respectivo tipo. Este procedimento torna possível a declaração de sinais de um tipo específico que podem tomar apenas um conjunto restrito de valores;
- Dois sinais:
  - "*currentState*" – para armazenar o estado actual;
  - "*nextState*" – para armazenar o estado seguinte;
- Duas tarefas do tipo rotina:
  - "*sequential*" – sensível aos eventos positivos da entrada de sincronização e a qualquer evento da entrada de inicialização. Esta tarefa é responsável pelas transições de estado da FSM, implementando a memória do circuito sequencial (registo de estado);

- "*combinatorial*" – sensível a qualquer evento que ocorra nas entradas e às transições de estado. Esta tarefa implementa a componente combinatória do circuito sequencial descrito pela FSM, determinando o estado seguinte em função do estado actual e das entradas e as saídas em função do estado actual. O cálculo das saídas e do estado seguinte usa a construção "*switch-case*" do C++.

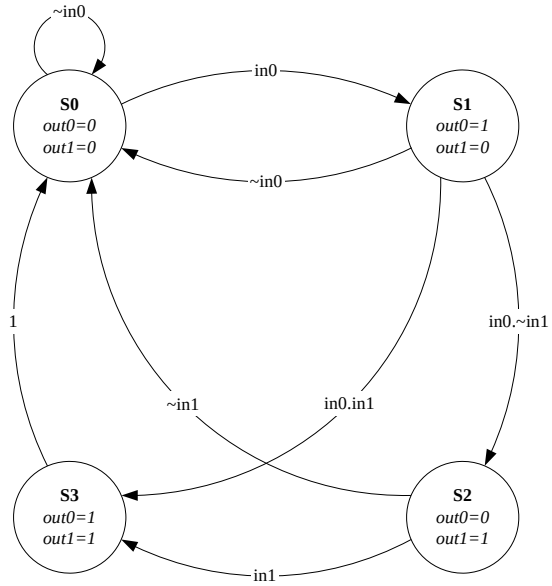

Figura 12 - Exemplo de uma máquina de estados finitos representada por um diagrama de transição de estados.

```
module(FSM)
{
   iport<bool> clk;
   iport<bool> reset;
   iport<bool> in0;
   iport<bool> in1;
   oport<bool> out0;
   oport<bool> out1;

   enum EState {S0, S1, S2, S3};

   signal<EState> currentState, nextState;

   method(sequential)
   {
      if (reset)
      {
         currentState = S0;
      }
      else if (clk.posevent)
      {
         currentState = nextState;
      }
   }

   method(combinatorial)
   {
      switch (currentState)
      {
         case S0 :
         {
            out0 = false;
            out1 = false;

            if (in0)
            {
               nextState = S1;
            }
            else
            {
               nextState = S0;
            }
            break;
         }
         case S1 :
         {
            out0 = true;
            out1 = false;

            if (!in0)
            {
               nextState = S0;
            }
            else
            {
               if (!in1)
               {
                  nextState = S2;
               }
               else
               {
                  nextState = S3;
               }
            }
            break;
         }
         case S2 :
         {
            out0 = false;
            out1 = true;

            if (in1)
            {
               nextState = S3;
            }
            else
            {
               nextState = S0;
            }
            break;
         }
         case S3 :
         {
            out0 = true;
            out1 = true;

            nextState = S0;
            break;
         }
      }
   }

   constructor(FSM)
   {
      routine(sequential);
      sequential->sensitive(clk, pos);
      sequential->sensitive(reset);

      routine(combinatorial);
      combinatorial->sensitive(currentState);
      combinatorial->sensitive(in0);
      combinatorial->sensitive(in1);
   }
};
```

Figura 13 - Descrição em EaSys de uma máquina de estados finitos.

As transições de estado podem ser escritas de uma forma mais compacta usando o operador "?:" do C++. No caso da Figura 13 poderiam ser expressas da seguinte maneira:

```
nextState = (in0 ? S1 : S0);
nextState = (in0 ? (in1 ? S3 : S2) : S0);
nextState = (in1 ? S3 : S0);
```

V. COMPILAÇÃO E SIMULAÇÃO DE UM MODELO

Os processos de compilação e simulação do modelo de um sistema escrito em EaSys são em tudo idênticos à construção e execução de uma aplicação em C++. Assim, para simular um modelo basta instanciá-lo numa aplicação de teste, compilar e executar o programa resultante. Todo o processo é efectuado com ferramentas standard de desenvolvimento em C++. O projecto deve incluir todos os ficheiros que possuem o código fonte do modelo do sistema a simular, a biblioteca que implementa a linguagem EaSys e, eventualmente, outras bibliotecas de componentes e classes utilizadas. A simulação consiste na aplicação de vectores de teste que permitam cobrir, se possível, todas as situações de funcionamento do sistema. Durante a simulação é analisada a resposta aos estímulos aplicados para assim validar o modelo elaborado para o sistema a projectar. Existem várias formas de aplicar estímulos e analisar a resposta do modelo:

- O utilizador pode aplicar manual e iterativamente os vectores de teste e observar as saídas produzidas;
- A aplicação dos vectores de teste pode ser automática, sendo as respostas produzidas armazenadas para posterior análise;
- O processo de análise dos resultados pode também ser simplificado através da detecção automática de erros e emissão de mensagens de notificação no caso de um funcionamento incorrecto ou inconsistente do modelo do sistema.

Na Figura 14 é mostrada a listagem completa do ficheiro "AOI.h" que possui o código fonte da porta lógica complexa AOI apresentada acima. Neste caso, a descrição do componente é feita num único ficheiro. Em geral, a descrição dos módulos de um sistema ou biblioteca pode ser feita de duas formas:

- Completamente em ficheiros de interface (*.h);
- Distribuída entre ficheiros de interface (*.h) e de implementação (*.cpp).

Devido à sua simplicidade, no exemplo da Figura 14 é usado o primeiro método. O segundo método possui a vantagem de permitir separar o interface da implementação e assim esconder os detalhes internos do módulo. Além das directivas para controlo de inclusão e da descrição do módulo AOI, o ficheiro "AOI.h" possui duas directivas para o preprocessador do C++ para inclusão dos ficheiros de definições da linguagem EaSys e do módulo "AND2" usado neste componente.

A listagem da Figura 15 mostra um exemplo de uma aplicação de teste em modo consola usada para simular o modelo da Figura 14. Após a instanciação e interligação do módulo "AOI" que se pretende simular é feita a inicialização do núcleo da linguagem (função `EaSys::Initialize()`). Seguidamente, a aplicação entra num ciclo infinito onde são aplicados os vectores de teste, invocado o núcleo de simulação (função `EaSys::Cycle()`) e visualizados os resultados.

```
#ifndef __AOI_H_INCLUDED__
#define __AOI_H_INCLUDED__

#include "EaSys.h"
#include "AND2.h"

module(AOI)
{
   iport<bool> in0;
   iport<bool> in1;
   iport<bool> in2;
   iport<bool> in3;
   oport<bool> out;

   signal<bool> and0Out, and1Out;

   method(nor)
   {
      out = !(and0Out || and1Out);
   }

   constructor(AOI)
   {
      instance(AND2, and0);
      and0->in0(in0);
      and0->in1(in1);
      and0->out(and0Out);

      instance(AND2, and1);
      and1->in0(in2);
      and1->in1(in3);
      and1->out(and1Out);

      routine(nor);
      nor->sensitive(and0Out);
      nor->sensitive(and1Out);
   }
};

#endif // __AOI_H_INCLUDED__
```

Figura 14 - Listagem completa do ficheiro "AOI.h" que possui a descrição da porta lógica complexa AOI.

```
#include <iostream.h>
#include <conio.h>
#include "EaSys.h"
#include "AOI.h"

int main(int argc, char* argv[])
{
   signal<bool> in0, in1, in2, in3, out;

   AOI gate("AOI");
   gate.in0(in0);
   gate.in1(in1);
   gate.in2(in2);
   gate.in3(in3);
   gate.out(out);

   EaSys::Initialize();

   while(1)
   {
      char ch = getch();

      switch (ch)
```

```
        {
           case '0' :
           {
              in0 = !in0;
              break;
           }
           case '1' :
           {
              in1 = !in1;
              break;
           }
           case '2' :
           {
              in2 = !in2;
              break;
           }
           case '3' :
           {
              in3 = !in3;
              break;
           }
           case 'q' :
           {
              exit(0);
           }
        }

        EaSys::Cycle(2);

        cout << "in3 = " << in3 << " , ";
        cout << "in2 = " << in2 << " , ";
        cout << "in1 = " << in1 << " , ";
        cout << "in0 = " << in0 << " | ";
        cout << "out = " << out << endl;
     }

     return 0;
  }
```

Figura 15 - Código fonte da aplicação de teste do módulo AOI.

Após compilação e execução da aplicação podem ser aplicados estímulos de forma interactiva para avaliar a resposta do componente a vários vectores de teste. A Figura 16 ilustra os resultados obtidos para o módulo AOI ao qual foram aplicados dezasseis vectores de teste. Da análise da resposta obtida pode-se concluir que o funcionamento do componente é o correcto pois corresponde à tabela de verdade (segundo codificação de *Gray*) da respectiva função lógica.

```
C:\Tmp>AOI
in3 = 0 , in2 = 0 , in1 = 0 , in0 = 0 | out = 1
in3 = 0 , in2 = 0 , in1 = 0 , in0 = 1 | out = 1
in3 = 0 , in2 = 0 , in1 = 1 , in0 = 1 | out = 0
in3 = 0 , in2 = 0 , in1 = 1 , in0 = 0 | out = 1
in3 = 0 , in2 = 1 , in1 = 1 , in0 = 0 | out = 1
in3 = 0 , in2 = 1 , in1 = 1 , in0 = 1 | out = 0
in3 = 0 , in2 = 1 , in1 = 0 , in0 = 1 | out = 1
in3 = 0 , in2 = 1 , in1 = 0 , in0 = 0 | out = 1
in3 = 1 , in2 = 1 , in1 = 0 , in0 = 0 | out = 0
in3 = 1 , in2 = 1 , in1 = 0 , in0 = 1 | out = 0
in3 = 1 , in2 = 1 , in1 = 1 , in0 = 1 | out = 0
in3 = 1 , in2 = 1 , in1 = 1 , in0 = 0 | out = 0
in3 = 1 , in2 = 0 , in1 = 1 , in0 = 0 | out = 1
in3 = 1 , in2 = 0 , in1 = 1 , in0 = 1 | out = 0
in3 = 1 , in2 = 0 , in1 = 0 , in0 = 1 | out = 1
in3 = 1 , in2 = 0 , in1 = 0 , in0 = 0 | out = 1
```

Figura 16 - Resultados da simulação do módulo AOI com a aplicação de teste da Figura 15.

VI. COMPARAÇÃO COM O SYSTEMC

Dado que a linguagem EaSys não é a única opção disponível para utilizar o C++ como linguagem de descrição de sistemas digitais, faz todo o sentido dedicar uma secção à comparação com outra abordagem semelhante existente para o mesmo efeito. Assim, para esta comparação foi escolhida a linguagem SystemC, devido, por um lado, à sua popularidade e por outro às semelhanças com a linguagem EaSys. Em termos sintáticos e semânticos, a linguagem EaSys possui muitas semelhanças com a linguagem SystemC. Mais concretamente, os elementos básicos da linguagem são os mesmos, isto é, módulos, tarefas, sinais e portos estão disponíveis em ambos os casos, mudando apenas a forma como são declarados. Ambas as linguagens necessitam apenas de um compilador de C++ standard para construção de um modelo executável, o que relativamente a outros métodos e linguagens de especificação constitui uma vantagem para ambas. O código fonte de ambas as linguagens é aberto. No entanto, a implementação da linguagem EaSys está melhor documentada e é mais modular, o que para efeitos de extensibilidade constitui uma vantagem importante.

A linguagem EaSys tem a desvantagem de se encontrar numa fase mais primitiva do que a SystemC. No entanto, esta também pode ser uma vantagem se considerarmos que muitos dos aspectos da sintaxe e da sua implementação podem ainda, se necessário, ser facilmente alterados. Uma vantagem importante do SystemC é a sua biblioteca de sistema que permite a elaboração de modelos a níveis de abstracção mais elevados.

Na linguagem EaSys existe uma separação clara entre o núcleo de simulação, as classes que implementam as várias abstracções da linguagem e os tipos de dados. Além disso, houve a preocupação de eliminar alguma redundância existente nos tipos de dados do SystemC, reduzindo o número de tipos de dados disponíveis e aumentando a sua flexibilidade através de uma parametrização simples e intuitiva das respectivas classes. Isto permite simplificar bastante a manutenção do código de implementação da linguagem.

Uma limitação do SystemC reside no tipo de dados usados para representar as variáveis lógicas e que só podem assumir quatro valores distintos: ‘Z’ (alta-impedância), ‘0’ (‘zero’ activo), ‘1’ (‘um’ activo), ‘X’ (indefinido). Como veremos na próxima secção, o tipo de dados correspondente previsto para a linguagem EaSys, além dos valores anteriores suporta também os valores ‘L’(‘zero’ passivo) e ‘H (‘um’ passivo) habitualmente disponíveis nas HDLs e úteis para a modelação de saídas passivas ligadas a barramentos.

Na linguagem EaSys foi dada uma atenção especial à definição dos operadores aplicáveis a objectos do tipo sinal e porto. O objectivo é fazer com que do ponto de vista da sintaxe a realização de uma operação de leitura/escrita sobre uma variável de qualquer tipo seja exactamente igual à realização da mesma operação sobre

um sinal ou porto com o mesmo tipo de dados. Desta forma é possível simplificar a escrita das operações porque não é necessário efectuar invocações explícitas de métodos para esse efeito. No SystemC, as operações efectuadas sobre portos devem recorrer aos métodos `read()/write()` do respectivo objecto. A Tabela 4 ilustra estas diferenças.

Operação	EaSys	SystemC

Leitura	`if (clk)`
	`{`
	`  ...`
	`}`
		`if (clk.read())`
		`{`
		`  ...`
		`}`

Escrita	`q = d;`	`q.write(d);`

Tabela 4 - Diferenças de sintaxe entre a linguagem EaSys e a SystemC nas operações de leitura e escrita em objectos do tipo porto.

O SystemC possui um problema grave na modelação de barramentos. A descrição de um barramento requer, além da declaração de um objecto que representa o meio partilhado, que os portos de saída dos módulos a ele ligados sejam de um tipo especial e diferente do utilizado nas ligações ponto a ponto. O segundo aspecto é problemático na construção de bibliotecas.

Finalmente, ao contrário do que acontece com o SystemC, a linguagem EaSys não impõe nenhuma restrição quanto ao ponto de entrada da aplicação que implementa o modelo de um sistema. Na linguagem SystemC, o ponto de entrada da aplicação deve ser uma função cujo nome é "sc_*main*", enquanto na linguagem EaSys basta que a aplicação seja compilada e que a biblioteca da linguagem seja associada durante a construção do executável. A instanciação do modelo do sistema, a inicialização do núcleo da linguagem e o controlo da simulação podem ser realizados em qualquer função, desde que executados pela ordem correcta. Apesar de ser um pormenor, este aspecto pode ser muito importante na integração de um modelo em diferentes tipos de ambientes de simulação.

VII. EXTENSIBILIDADE E TRABALHO FUTURO

Apesar da linguagem EaSys permitir já a descrição e simulação de uma gama considerável de sistemas digitais, existe ainda muito trabalho a realizar. Um dos aspectos que facilita a extensão da linguagem é a arquitectura aberta e bem documentada do seu núcleo de implementação. As possibilidades de trabalho futuro dividem-se essencialmente nos seguintes tópicos:

- Expansão das funcionalidades do núcleo da linguagem
  - implementação da classe *CEaSysThread* responsável pelos mecanismos de activação e suspensão específicos das tarefas do tipo *thread*;
  - implementação da classe *CEaSysBus* que suporta a instanciação de objectos que representam barramentos, isto é, sinais que podem ser controlados por múltiplas saídas simultaneamente. Um barramento de N bits é um sinal especial capaz de armazenar valores lógicos standard e que possui mecanismos para resolver conflitos;
- Definição de novos tipos de dados úteis para descrever hardware digital. Estes tipos deverão ser parametrizáveis para que as suas características (e.g. tamanho) possam ser definidas durante a criação dos respectivos objectos. Além disso, deve também ser implementado o subconjunto dos operadores pretendidos para cada tipo de dados. Por exemplo, no caso de uma classe que representa uma quantidade inteira, devem ser definidos os operadores aritméticos, enquanto para um vector de dígitos binários devem ser definidos os operadores lógicos. Além disso, pode também ser útil a definição de operadores de conversão entre alguns dos tipos de dados da linguagem EaSys e os existentes no C++. Os tipos de dados de interesse identificados até ao momento são:
  - bit<N> - vector parametrizável de N dígitos binários (podem possuir apenas os valores '0' ou '1');
  - logic<N> - vector parametrizável de N dígitos que representam valores lógicos standard, podendo cada um tomar um dos valores 'Z' (alta-impedância), '0' ('zero' activo), '1' ('um' activo), 'L'('zero' passivo), 'H ('um' passivo), 'X' (indefinido);
  - integer<B, N> - quantidade inteira de N bits construída com o tipo base B (char, short, long, int);
  - fixed<…> - quantidade decimal em vírgula fixa. Os parâmetros deste tipo de dados ainda não estão definidos;
- Construção de bibliotecas – tal como já foi dito atrás, a combinação das capacidades da linguagem desenvolvida com as pré-existentes no C++ permite a construção de bibliotecas de componentes genéricos e parametrizáveis. Desta forma é possível simplificar a descrição de sistemas complexos e em certos casos até sua implementação através da associação de determinados componentes da biblioteca a macro-células disponíveis na tecnologia alvo. As bibliotecas a implementar podem ser essencialmente de dois tipos:
  - As que representam componentes digitais genéricos, tais como portas lógicas, registos, contadores, comparadores, (des)codificadores, (des)multiplexadores, circuitos de aritmética, memórias, ou blocos mais específicos ou complexos como filtros digitais, circuitos de interface, processadores, etc;
  - As que disponibilizam construções e mecanismos úteis para modelar sistemas complexos a um nível comportamental abstracto. Um modelo que utilize estas abstracções deve ser validado e

progressivamente refinado até poder ser sintetizado e implementado. Exemplos de abstracções úteis para este fim são os mecanismos de comunicação e sincronização entre módulos e tarefas, a especificação de restrições temporais, etc;

- Construção de ferramentas de síntese de hardware – dentro de determinadas restrições, uma descrição de hardware em EaSys pode ser convertida numa descrição VHDL ou Verilog ao nível RTL para posterior processamento pelas ferramentas tradicionais de síntese de hardware. Contudo, uma abordagem mais interessante e que importa considerar é a síntese directa de hardware a partir da descrição de um sistema na linguagem EaSys. Esta é provavelmente uma das tarefas mais complexas deste trabalho, mas também a mais interessante do ponto de vista de investigação;
- Criação de um ambiente de desenvolvimento integrado – para que os resultados deste trabalho sejam utilizados de uma forma fácil e intuitiva, é importante integrar as várias ferramentas construídas num ambiente de desenvolvimento. Este ambiente deve, entre outras coisas, permitir:
  - especificar e modelar o sistema a projectar;
  - automatizar a criação de programas de teste para o modelo resultante;
  - efectuar a compilação do modelo e respectivo programa de teste;
  - correr a especificação executável do sistema para efeitos de simulação;
  - visualizar intuitivamente os resultados obtidos;
  - refinar de forma sistemática o modelo até que este possa ser sintetizado;
  - sintetizar o hardware que implementa o sistema;
  - invocar as ferramentas de implementação específicas da tecnologia utilizada;
- Análise do desempenho e da qualidade dos resultados produzidos (*benchmarking*) – para qualquer linguagem e ferramenta de projecto é fundamental comparar os tempos de execução e a qualidade final dos resultados obtidos com outras técnicas e ferramentas disponíveis para o mesmo fim;
- Adaptação do núcleo da linguagem EaSys a outras linguagens base - a biblioteca de classes que implementa a linguagem EaSys pode ser reescrita noutras linguagens orientadas por objectos, sendo a linguagem JAVA uma das possibilidades mais interessantes devido à sua crescente utilização quer no desenvolvimento do software, quer como linguagem de especificação.

VIII. CONCLUSÕES

A utilização de linguagens de programação tradicionais nas fases iniciais do projecto de sistemas complexos para efeitos de especificação e validação é uma abordagem cada vez mais utilizada. Por outro lado, a uniformização das linguagens usadas ao longo de todas as etapas de projecto possui vantagens importantes ao nível da produtividade e consequentemente no tempo de desenvolvimento. A linguagem EaSys apresentada neste artigo adiciona à linguagem de programação orientada por objectos C++ um conjunto de funcionalidades e mecanismos que a tornam adequada também para descrever hardware. A abordagem utilizada para estender a linguagem C++ consiste numa biblioteca de classes permitindo a utilização de ferramentas standard de desenvolvimento em C++ para descrever, compilar e simular um modelo de um sistema. Esta linguagem proporciona uma plataforma uniforme para a modelação, simulação, verificação e síntese de sistemas complexos compostos tanto por componentes de hardware como de software. A arquitectura do núcleo de implementação da linguagem é extremamente compacto, aberto, modular, flexível, extensível e portável podendo ser utilizado em qualquer plataforma que possua um compilador de C++ standard. A disponibilização do código fonte e das bibliotecas pré-compiladas para as plataformas Microsoft Windows 2000 e Linux está prevista para o início do quarto trimestre de 2001.

REFERÊNCIAS

[1] D. Verkest, J. Kunkel and F. Schirrmeister, “System Design using C++”, *Proceedings of DATE-2000*, pp. 74-83, Paris, France, March 2000.

[2] R. Helaihel and K. Olukotun, "JAVA as a Specification Language for Hardware/Software Systems", *Proceedings of ICCAD-97*, pp. 690-697, San Jose, CA, November 1997.

[3] P. Ashenden, “The Designer’s Guide to VHDL”, Morgan Kaufmann, 1996.

[4] D. Thomas and P. Moorby, “The Verilog Hardware Description Language”, Kluwer Academic Publishers, 1996.

[5] R. Gupta and S. Liao, “Using a Programming Language for Digital System Design”, *IEEE Design and Test of Computers*, pp. 72-80, April-June 1997.

[6] A. Ghosh, J. Kunkel, and S. Liao, “Hardware Synthesis from C/C++”, *Proceedings of DATE-99*, pp. 387-389, Munich, Germany, March 1999.

[7] HandelC, http://www.celoxica.com/.

[8] Open SystemC Initiative, http://www.SystemC.org/ .

[9] OCAPI, http://www.imec.be/ocapi/.

[10] Cynlib, http://www.CynApps.com/.

[11] B. Stroustrup, “The C++ Programming Language”, 2nd Edition, Addison-Wesley Publishing Company, 1995.