Universidade do Minho

André Ribeiro de Carvalho

Extração Automática de Modelos de Sistemas de Tempo Real

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Doutor Jorge Sousa Pinto e Doutor Simão Melo de Sousa

Setembro 2011

Agradecimentos

Em primeiro lugar quero agradecer aos meus dois orientares Doutor Jorge Sousa Pinto e Doutor Simão Melo de Sousa, a liberdade, o apoio e o auxilio prestado durante todo o período de investigação. Sem as suas experiências e conhecimentos, teria sido impossível tomar as decisões mais correctas de forma a serem atingidos os objectivos a que me propus.

Como seria de esperar não posso deixar de agradecer à minha família, em particular aos meus pais (José Carvalho e Maria Ribeiro), pois foram eles que proporcionaram todas as condições para o meu percurso académico, e consequente realização de um sonho. Infelizmente, a conjectura do nosso país não permite que toda a gente possa ter acesso ao ensino superior, por isso mesmo o meu obrigado especial aos meus pais por me proporcionarem este momento.

Quero agradecer também a todos os colaboradores do Projecto RESCUE e EVOLVE, por todas as conversas e opiniões dadas, pois ajudaram a tomar decisões importantes e consequentemente a melhorar o trabalho desenvolvido.

Por fim agradeço a todos os meus amigos e colegas por todo o apoio que me deram durante a licenciatura e mestrado, e ainda a todas as pessoas que de alguma forma contribuiriam para o meu sucesso em todo o percurso académico.

Resumo

Esta dissertação insere-se no contexto da investigação sobre verificação de programas de tempo-real. Tendo como principais referências o artigo de Alan Burns [10] e a dissertação de Joel Carvalho [16], introduz-se aqui uma ferramenta capaz de gerar automaticamente modelos Uppaal representativos de programas escritos em Ada. Através desses modelos, e com o auxílio do *model checker* do Uppaal é possível então verificar propriedades especificadas em lógica temporal, e dessa forma obter confiança quanto ao comportamento do sistema em termos temporais. Assim, ao longo desta dissertação irão ser apresentados todos os conceitos imprescindíveis na compressão do trabalho, bem como todos os pormenores do algoritmo desenvolvido. Por fim serão usados alguns casos de estudo para demonstrar as potencialidades da ferramenta desenvolvida.

Abstract

The context of this master thesis dissertation is the verification of real-time programs. Taking as main references an article by Alan Burns [10] and a master's dissertation by Joel Carvalho [16], we introduce here a tool that is able to automatically generate Uppaal models of Ada programs. With these models, it is possible to use the Uppaal model checker to verify properties specified in temporal logic, increasing the reliability on the temporal behavior of the system. In this dissertation we present all the concepts involved in this approach, as well as the details of the developed algorithm.The dissertation concludes with some case studies that illustrate the results and tool capabilities.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A engenharia informática e as ciências da computação podem ser divididas em diversas subáreas. Se olharmos em particular para a subárea dos sistemas de tempo real, é possível notar que existe um conjunto bem definido de características a ter em conta para que o desenvolvimento de aplicações seja atingido com sucesso. Entre essas características podemos destacar linguagens de programação exclusivas e diferentes maneiras de raciocinar sobre a construção dos programas, i.e. um paradigma de programação diferente.

No passado recente temos assistido a uma ascensão do número de sistemas de tempo real presentes no nosso meio. Se tivermos em conta que todos os dias novos dispositivos são inventados e que uma grande percentagem dos mesmos possui algum tipo de sistema de tempo real embutido, rapidamente percebemos que nos estamos a tornar dependentes destes sistemas, o que aumenta a preocupação acerca da segurança e fiabilidade dos mesmos. Os telemóveis, o ABS (*anti-lock braking system*), os ECU (*engine control unit*) também conhecidos como centralinas, são apenas alguns exemplos de dispositivos compostos por sistemas de tempo real e cujo uso se tornou imprescindível. De facto, este cenário é preocupante, principalmente se tivermos em conta que muitos desses sistemas são críticos, i.e. a falha pode significar a perda de grandes investimentos ou em alguns casos vidas humanas. Assim sendo, torna-se importante nesta fase começar a apostar na certificação destes sistemas tão importantes para nós nos dias que correm, e uma forma de o conseguir é aliar a investigação feita nas universidades com as empresas que desenvolvem este tipo de sistemas.

Neste contexto, não podemos deixar de referir a importância do projecto EVOLVE[1], que surge como uma tentativa de dar resposta à necessidade acima referida. No caso concreto desta tese, o projecto teve um papel extremamente importante, pois permitiu identificar não só temas que carecem de investigação, como também permitiu estabelecer objectivos e metas a atingir com os trabalhos a realizar.

Ainda neste campo, devemos igualmente destacar a parceria Universidade do Minho / Critical Software, uma vez que também esta teve influência directa no trabalho de tese. Como é sabido, a Critical Software está presente em vários projectos a nível europeu, não só propostos pela ESA[2] mas também por outras organizações. A Critical em conjunto com outras empresas de desenvolvimento de software, tem sido responsável por desenvolver diversos componentes de software para satélites. Assim sendo, uma vez que o tema desta dissertação se relaciona directamente com estes projectos, através desta parceria foi possível estabelecer contactos que permitiram definir alguns objectivos, assim como obter aconselhamento quanto às tecnologias a serem usadas.

Ao pensar-se em validação e certificação de programas salta logo à memória a utilização de métodos formais que apesar de terem uma história recente comparando com outras áreas, produziram já diversos casos de sucesso na sua aplicação. No entanto, se por um lado a verificação de sistemas representa um problema complexo, a introdução da componente tempo vem acrescentar uma nova dificuldade. Este facto pode explicar a razão pela qual é possível encontrar mais publicações lidando com o problema da verificação formal de sistemas de software tradicionais, em comparação com os sistemas de tempo real. Apesar disso, é possível encontrar publicadas algumas propostas recentes que tentam responder aos problemas de verificação ainda existentes.

Tomando como influência três publicações em concreto [23] [10] [15] (ver capítulo 2, secção 2.8), o propósito desta dissertação é descrever o desenvolvimento de uma ferramenta capaz de gerar automaticamente modelos temporais (autómatos temporizados) a partir de código Ada, permitindo posteriormente com o auxílio de um *model checker* a validação de propriedades temporais do programa. A fase inicial deste trabalho de mestrado consistiu na ambientação e aquisição dos conceitos básicos envolvidos, tendo por base a ferramenta HTL2XTA desenvolvida por Joel Carvalho, bem como a análise de casos de estudo para esta ferramenta. Por esta razão, anexa-se a esta dissertação (Apêndice 5.2) a totalidade de um artigo [15] publicado em co-autoria, relativo àquele

[1] Evolutionary Verification, Validation and Certification, financiado pelo Programa Operacional Factores de Competitividade - QREN - SI & I&DT.

[2] European Space Agency.

projecto.

1.1 Estrutura do Documento

Esta dissertação apresenta-se dividida em cinco capítulos principais. No primeiro capítulo são explicadas as motivações para o surgimento deste trabalho. De seguida apresenta-se a contextualização do trabalho, onde são explicados todos os conceitos relacionados com o trabalho desenvolvido e indispensáveis na descrição do mesmo. No capítulo seguinte (terceiro), são explicados todos os detalhes técnicos do algoritmo e da ferramenta, assim como dependências de tecnologias que tornaram possível o desenvolvimento dos mesmos. No quarto capítulo, é efectuada uma validação experimental à ferramenta, recorrendo-se para isso à utilização de diferentes casos de estudos. Finalmente, no quinto e último capítulo, apresenta-se toda a perspectiva futura em termos de trabalho a realizar na ferramenta, assim como as conclusões que este trabalho de tese permitiu retirar.

Capítulo 2

Contexto

2.1 Tempo Real

Sistemas de Tempo Real

Um sistema de tempo-real é um sistema cuja correcção não depende apenas do resultado da computação, mas também do momento em que esta é produzida, ou seja estes sistemas diferem dos sistemas tradicionais no sentido em que adicionam aos requisitos do sistema a necessidade intrínseca de se assegurar que as tarefas são executadas dentro de um intervalo de tempo bem definido. Para estes sistemas, o não cumprimento desses requisitos implica a falha global dos mesmos.

O facto de os sistemas de tempo real diferirem dos tradicionais apenas na existência de requisitos temporais, embora pareça uma pequena diferença, na verdade pode trazer mais complicações ao processo de verificação do que aquilo que se podia esperar. Enquanto que nos sistemas tradicionais é relativamente simples calcular as condições de verificação resultantes de pré-condições, pós-condições e invariantes, nos sistemas de tempo-real acrescem-se outras propriedades devido à existência de restrições temporais. Esta dificuldade pode explicar então o facto de não existir praticamente nenhuma solução capaz de certificar a correcção temporal de um programa.

No artigo [23], uma das principais influências deste trabalho, as motivações dos autores para o trabalho lá descrito, representam as dificuldades usuais encontradas no desenvolvimento destes programas, e são também essas dificuldades que impulsionam a comunidade científica a investigar este assunto:

- O comportamento temporal de um programa de tempo real de-

pende da estratégia de compilação, facto que representa uma dificuldade acrescida no desenvolvimento usando linguagens de alto nível;

- O comportamento temporal do código executável não é visível no código fonte;
- Infelizmente as linguagens de programação tradicionais, até mesmo as desenhadas para desenvolver sistemas de tempo real, não oferecem qualquer tipo de mecanismo para expressar directamente requisitos temporais;
- Por norma, os programadores recorrem ao teste e à análise de código de baixo nível, na tentativa de encontrar algumas garantias ao código desenvolvido. No entanto, é normal que este passo leve à perda de produtividade por parte dos programadores.

Linguagens de Programação em Sistemas de Tempo Real

O desenvolvimento de sistemas de tempo real é suportado actualmente por algumas linguagens de programação, como por exemplo o Ada ou o Java Real-Time. Na verdade, é notória a dificuldade acrescida para raciocinar sobre estes sistemas e por isso mesmo estas linguagens foram desenhadas por forma a facilitar o trabalho do programador no desenvolvimento dos seus programas. Apesar das diferenças que cada uma pode apresentar, existe um conjunto de características partilhadas pela maioria das linguagens de tempo real:

- Declaração de tipos *Time*;
- Leitura do tempo directamente do hardware;
- Instrução de atraso na execução segundo um tempo especificado;
- Recuperação do erro através de mecanismos de timeout.

Ada & Perfil Ravenscar

As características poderosas do Ada tornaram esta linguagem numa linguagem de referência para os sistemas de tempo-real, por isso é normal que muitos dos trabalhos de investigação encontrados nesta área sejam para esta linguagem. Em Abril de 1997, no 8^{th} *International Real-Time Ada Workshop (IRTAW)* na pequena vila de *Ravenscar* em *Yorkshire*, dois

artigos [13] e [21] levaram à discussão daquilo que iria resultar no perfil *Ravenscar* [8] [9] no final do evento. A essência deste perfil é restringir o espectro de primitivas de concorrência permitidas pelo Ada, através do uso de *pragmas*[1], e assim tentar reduzir alguns dos erros usualmente cometidos no desenvolvimento de programas de tempo real. Com estas publicações foi ainda intenção dos responsáveis estimular a investigação sobre verificação de sistemas de tempo real. O documento final do perfil *Ravenscar* contém muitos itens, e para melhor compreender o perfil é recomendada a leitura de [9]. Com o intuito de ilustrar o tipo de restrições envolvidas, os tópicos mais relevantes são listados a seguir:

- Inexistência de hierarquia de tarefas;
- Proibição da alocação dinâmica de tarefas ou objectos protegidos;
- Proibição da libertação de objectos protegidos e tarefas;
- As tarefas não terminam;
- Não existem atrasos relativos;
- Inexistência de instruções de selecção que introduzam não-determinismo na escolha;
- Inexistência de instruções que levam ao aborto do sistema;
- Inexistência de *exception handlers*;
- Inexistência de IO.

SPARK

A linguagem SPARK [14] [17] criada pela *Praxis High Integrity Systems*, é neste momento uma linguagem com bastante reputação entre os programadores de sistemas de tempo real. O facto de se tratar de um subconjunto do Ada seguindo a filosofia *Correcto por Construção*, não significa que as poucas construções disponíveis diminuam drasticamente a expressividade da mesma, e comprova isso mesmo o facto de existirem muitas aplicações industriais desenvolvidas em SPARK. No entanto, o que faz desta linguagem uma solução realmente poderosa, é o conjunto de ferramentas disponibilizado e que permite a análise estática dos programas desenvolvidos. Três ferramentas fazem parte desse conjunto, o

[1] Directiva disponível na sintaxe do Ada e usado para especificar que o programa deve ser compilado segundo o perfil indicado

SPARK Examiner, o *SPADE Proof Checker* e o *Automatic Simplifier*. Na prática a maior parte do esforço é dispendido pelo *Examiner*, e até ao final da prova é possível identificar seis passos essenciais. Durante a execução do *Examiner* no primeiro passo é realizada uma análise léxica e sintáctica em cada unidade de compilação, seguindo-se uma análise semântica que perfaz o segundo passo. Na terceira etapa, é feita uma análise ao fluxo de controlo do programa por forma a verificar a estrutura do programa. Em seguida, o fluxo de dados é analisado na tentativa de encontrar erros como variáveis declaradas mas não usadas, instruções inúteis ou até variáveis não inicializadas. No penúltimo passo é verificada a existência de possíveis erros de execução como divisões por zero ou índices de vectores fora dos limites, e para estes são adicionadas condições de verificação. No final é invocado o *Simplifier*, que tipicamente prova automaticamente a maioria das condições de verificação anteriormente geradas, sendo as restantes provadas usando o *Proof Checker*, interactivamente.

Recentemente o SPARK adoptou também o perfil *Ravenscar* [2] por forma a permitir o desenvolvimento de aplicações concorrentes. No entanto o conjunto de ferramentas permanece inalterado quanto à verificação de aspectos concorrentes dos programas. Mais ainda, a *Praxis High Integrity Systems* disponibiliza agora o SPARK com uma licença GPL, precisamente numa tentativa de aumentar o interesse da comunidade científica na utilização do SPARK.

2.2 Hierarchical Timing Language - HTL

O HTL (*Hierarchical Timing Language*) [26] foi introduzido em 2006 a partir das teses de doutoramento de dois investigadores, Arkadeb Ghosal [25] e Daniel Iercan [32]. A principal inspiração do HTL foi a linguagem de programação Giotto [28] [27], criada para o desenvolvimento de sistemas embebidos com tarefas periódicas. O HTL é reconhecido como uma linguagem de coordenação [24]: o seu principal objectivo é oferecer uma plataforma intermediária de forma a conseguir combinar os aspectos mais interessantes de outras linguagens. Assim sendo, a principal aplicação do HTL são os sistemas de tempo real críticos com tarefas periódicas. Em detalhe, o objectivo principal do HTL é especificar o comportamento temporal de sistemas que possam apresentar parte funcional descrita em C ou C++. Com esta estratégia é possível separar a descrição temporal da descrição lógico-funcional. Deste modo, o raciocínio sobre os programas fica facilitado uma vez que permite abstrair as diferentes componentes dos programas. Do ponto de vista da verificação,

este aspecto é ainda mais evidente pois permite concentrar o esforço da verificação em cada uma das componentes em separado. Embora o compilador do HTL verifique alguns aspectos do programa, esta verificação é algo limitada uma vez que se restringe essencialmente ao escalonamento. Na tentativa de responder a esta lacuna Joel Carvalho [15] propõe uma ferramenta que facilita a verificação da parte temporal dos programas. Esse trabalho resultou na sua tese de mestrado e é explicado com mais detalhe na secção 2.8 deste capítulo.

2.3 Autómatos

É um facto que existem muitos mecanismos propostos para modelar sistemas de tempo real. No entanto, o mais sucedido é a modelação usando autómatos temporizados [6] proposto em 1990 por Alur e Dell [1]. Dado que os sistemas de tempo real podem ser modelados como redes de autómatos temporizados, torna-se importante apresentar um resumo dos conceitos mais importantes destas estruturas.

Autómatos de Büchi

Antes de se focar a modelação de sistemas com autómatos temporizados, é importante relembrar primeiro as noções básicas dos autómatos em geral começando pelos autómatos de estado finito etiquetados (*labelled finite-state automata*).

Definição 2.1 (Autómato)**.** *Um autómato é descrito pelo 5-tuplo* $\mathcal{A} = (Q, E, T, Q_0, I)$ *onde:*

- Q *é o conjunto dos estados;*
- E *é o conjunto das etiquetas das transições;*
- $T \subseteq Q \times E \times Q$ *é o conjunto das transições;*
- $Q_0 \subseteq Q$ *é o conjunto dos estados iniciais. Para simplificar, consideram-se os autómatos com apenas um estado inicial,* q_0*;*
- $I : Q \to \mathcal{P}_F(Prop)$ *é uma aplicação que relaciona cada estado de* Q *com um conjunto finito de propriedades por ele satisfeitas;*

Tem particular relevância uma versão extendida destes autómatos, passando a aceitar execuções *infinitas*. Estes são chamados de Autómatos de Büchi etiquetados (*labelled Büchi automata*).

Assim sendo, um caminho num autómato de Büchi $\mathcal{A}$ é uma sequência (possivelmente infinita) σ de transições ligadas, e $|\sigma|$ é o comprimento da sequência. Um caminho é chamado de *execução parcial* se começar em q_0 e de *execução completa* se for um caminho parcial e não poder ser mais extendido. Um estado é atingível se existe um caminho que começa em q_0 e o contém.

A partir deste momento é possível estender estes autómatos introduzindo *variáveis*. Cada transição poderá estar disponível ou não conforme o teste que se faça à variável associada. Uma transição seleccionada poderá também alterar os valores das variáveis. Assim sendo uma transição é o 5-tuplo (q, g, l, a, r), onde:

- q é o estado de onde se parte;
- a guarda g é a propriedade sobre as variáveis de estado;
- l é a etiqueta da transição;
- o conjunto $a = x_1 := e_1, ...$ contém as atribuições a serem feitas;
- r é o estado destino.

Naturalmente as definições de caminhos e execuções são estendidas.

No entanto os algoritmos tradicionais de *model checking* não suportam autómatos estendidos, assim sendo é necessário transforma-los em autómatos clássicos [4] [5]. Esta transformação pode ser conseguida "desdobrando" o autómato (se este estiver em determinadas condições que o permita[2]) num autómato finito, sem variáveis, equivalente ao original. Assim, cada estado no autómato desdobrado é chamado de estado global e contém a informação do estado original correspondente, bem como o valor das variáveis na configuração concreta que este representa.

Sistemas de Autómatos

Modelando os sistemas sob a forma de autómatos, existem por vezes algumas situações em que é importante cada componente do sistema ser representada por uma autómato diferente. Assim, a sincronização destes autómatos terá igualmente de ser modelada. Para isto, pode recorrer-se ao produto síncrono de autómatos, *synchronous product of automata*.

Definição 2.2 (Produto Sincronizado de autómatos de Büchi)**.** *Consideremos a família de n autómatos* $\mathcal{A}_i = (Q_i, E_i, q_{0,i}, l_i)$ *(com* $i = 1, ..., n$*).*

[2]Estas condições terão de limitar os valores das variáveis

Consideremos também a etiqueta "_" correspondente à acção "trivial (nada a fazer)".

- *Define-se por conjunto de sincronizações sync, um conjunto tal que* $sync \subseteq \prod_{1 \leq i < n}(E_i \cup \{\_\})$.

- *Define-se por produto sincronizado da família de autómatos* $\mathcal{A}_i$*, o autómato* $\mathcal{A} = (Q, E, T, q_0, l)$ *tal que:*

- $Q = Q_i \times ... \times Q_n$;

- $E = \prod_{1 \leq i < n}(E_i \cup \{\_\})$;

- $$T = \left\{ \begin{array}{l} ((q_1, ..., q_n), (e_1, ..., e_n), (q'_1, ..., q'_n)) \mid \\ \qquad (e_1, ..., e_n) \in sync \\ \qquad \wedge \forall i, (e_i = \_ \wedge q'_i = q_i) \vee (e_i \neq \_ \wedge (q_i, e, q'_i) \in T_i \end{array} \right\}$$

- $q_0 = (q_{0,1}, ..., q_{0,n})$;

- $l(q_1, ..., q_n) = \bigcup_{1 \leq i < n} l_i(q_i)$.

Note-se que a definição $sync = \prod_{1 \leq i < n}(E_i \cup \{\_\})$, significa que trabalhamos com o produto cartesiano dos autómatos dados. Apesar deste ser fácil de obter, o tamanho é exponencial em relação ao tamanho dos autómatos originais. Assim, o sistema terá de ser analisado de forma a determinar quais os estados que são realmente válidos.

Uma vez que as transições válidas têm um número limite, possivelmente existirão estados inacessíveis. Removendo explicitamente estes estados produz-se um grafo de acessibilidade. Apesar de este grafo ser obviamente mais pequeno, é possível retirar muita informação acerca do sistema, e por isso mesmo a maioria dos *model checkers* usa esta técnica durante o processo de verificação.

Existem duas maneiras de restringir as transições de um produto de autómatos. A primeira usa mensagens para comunicar entre os autómatos, i.e. os autómatos deverão enviar ($m!$) e receber ($m?$) mensagens. A sincronização deverá conter apenas transições em que cada mensagem enviada é recebida por outro autómato. A segunda estratégia usa variáveis globais para sincronizar os autómatos. No entanto, existem normalmente problemas associados ao uso desta técnica, relacionados com o uso concorrente de variáveis.

Autómatos Temporizados e Redes de Autómatos Temporizados

Apesar dos autómatos apresentados até agora já considerarem uma ordem temporal qualitativa (por exemplo, a transição e está apenas disponível depois de ter ocorrido a transição e') via sincronização, não é possível representar uma ordem temporal quantitativa. Esta ordem irá ser conseguida através do uso de relógios. No entanto, não será necessário degenerar os modelos apresentados anteriormente para se conseguir incluir este mecanismo. Assim, com base nos autómatos anteriores, são introduzidos relógios como variáveis reais, através das quais as transições poderão realizar testes. Neste caso as transições são igualmente instantâneas pois os relógios são apenas incrementados nos estados. Por outro lado, os relógios avançam todos ao mesmo ritmo, e as únicas operações disponibilizadas sobre estes são a leitura e a reinicialização.

Neste ponto, para cada estado existem dois tipos de transições passíveis de serem executadas. O tempo poderá avançar incrementando todos os relógios mas permanecendo no mesmo estado, ou de outra forma poderá ser executada uma transição regular mudando de estado e congelando os relógios (com excepção do caso em que é suposto serem reiniciados).

Considere-se que a configuração é o par (q, μ), em que q é o estado actual e μ é o valor de todos os relógios num dado instante. Poderá também ser considerada a existência de um relógio global (que nunca é reiniciado) representativo do tempo global do sistema. Assim, a execução de um autómato temporizado, também chamada de trajectória, pode ser vista como uma aplicação ρ a partir de valores reais para configurações, i.e. $\rho(t)$ devolve a configuração do sistema no instante t.

Poderão também ser definidos invariantes de estado, os quais são condições sobre os relógios. Ao contrário das guardas, que representam "possibilidade", os invariantes representam "necessidade", de tal modo que o sistema não poderá estar num estado em que o invariante não seja satisfeito.

Finalmente, é considerada a noção de urgência, que pode ser aplicada a transições ou estados. Em relação às transições urgentes, quando alguma é elegível isto significa que não há incremento no tempo e que essa transição é tomada. Os estados urgentes seguem a mesma ideia: um estado é forçado a transitar para outro. Combinando transições urgentes e estados urgentes, cria-se o conceito de estado *commited*, que representa transições atómicas entre estados. Estes conceitos podem ser usados para evitar incrementos indefinidos em variáveis de relógio.

O sistema como um todo é representando por uma rede de autómatos

temporizados e sincronizados, onde a execução é uma sequência (possivelmente infinita) de configurações, e onde cada configuração representa o estado global de cada autómato, bem como os respectivos relógios. À semelhança de sistemas com um único autómato, existem duas possibilidades: todos os relógios de todos os autómatos são incrementados; ou uma transição de estado é executada.

2.4 Lógica Temporal

A lógica temporal é usada para descrever e raciocinar sobre sistemas, nos quais as propriedades relevantes podem ser qualificadas em termos de tempo. No entanto, o tempo não é explicitamente mencionado, mas antes representado por sequências de estados. Portanto, de forma a verificar a validade das proposições, os possíveis caminhos (sequência de estados) do sistema terão de ser considerados. Estes irão ser representados por *Estruturas de Kripke*, que na verdade são autómatos sem etiquetas.

Definição 2.3 (Estruturas de Kripke)**.** *Um estrutura de Kripke é um tuplo* (S, i, R, L) *onde:*

- S *é o conjunto finito de estados;*
- $i \in S$ *é o estado inicial;*
- $R \subseteq R \times T$ *é a relação de transição total, tal que,* $\forall s \in S, \exists s' \in S, (s, s') \in R$*;*
- $L : S \rightarrow \mathcal{P}(P)$ *é a função que etiqueta cada estado com o conjunto de fórmulas atómicas válidas nesse estado.*

Existem dois modelos diferentes para representar o tempo, e ambos podem ser obtidos a partir de uma *estrutura de Kripke*. Através do modelo de *tempo linear*, o comportamento do sistema é representado por um conjunto infinito de traços a começar em i, como ilustra a figura 2.1. O modelo de *tempo ramificado* representa o comportamento como uma árvore computacional de profundidade ilimitada, cuja raíz é i, como é possível verificar na figura 2.2. Este último é mais rico em termos de informação, e existem algumas propriedades que só podem ser expressas com este modelo. De seguida apresenta-se de forma breve ambos os modelos.

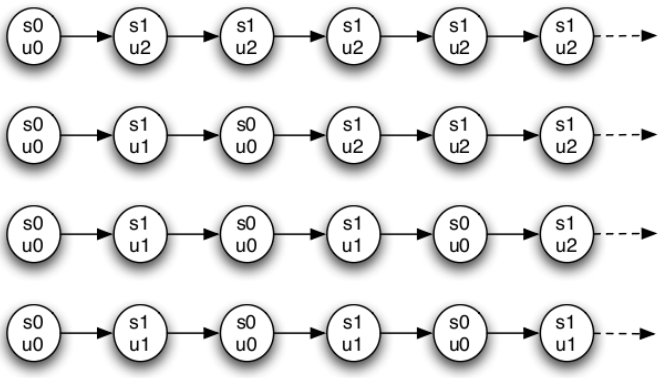

Figura 2.1: Modelo LTL (figura retirada de [19])

Linear Temporal Logic

Na *lógica temporal linear* (LTL), as proposições são interpretadas como traços infinitos em que cada um representa uma possível execução (ver figura 2.1.

Considerando um conjunto de proposições atómicas P, a sintaxe das fórmulas LTL é:

- *verdadeiro* or *falso* é uma fórmula;
- qualquer $p \in P$ é uma fórmula;
- $\neg f$, Xf, Ff e Gf são fórmulas, em que f é uma proposição LTL;
- $f \wedge g$, $f \vee g$, fUg e fRg são fórmulas, em que f, g são proposições LTL.

Com isto introduz-se novos operadores modais cuja definição informal é

Next (Xf) quando f é válido no próximo estado;

Future (Ff) quando f é válido num estado futuro;

Globally (Gf) quando f é sempre válido;

Until (fUg) quando f é válido até g o ser;

Release (fRg) quando f é válido num estado e faz com que g não o seja.

Os operadores X, F e G são por vezes denotados por $\circ$, $\Diamond$ e $\Box$.

Quando uma fórmula LTL é satisfeita num caminho π de um estrutura de Kripke M, escrevemos $M, \pi \models f$ que se lê π é um modelo de f". Uma fórmula f é satisfeita num estado s ($M, s \models f$) se for satisfeita por todos os caminhos que começam em s. Se for satisfeita em todos os estados de M, a fórmula é válida na estrutura de Kripke, denotado por $M \models f$.

Full Computational Tree Logic

O conceito base da *Full Computational Tree Logic* (CTL*) é que para cada estado poderá existir um conjunto de diferentes estados sucessores, por isso mesmo a estrutura natural desta lógica de forma a representar todas a computações possíveis, é uma árvore de estados (ver figura 2.2). Os modelos CTL* contêm mais informação do que os LTL, e de facto existem algumas propriedades que são apenas expressas usando estes modelos.

A CTL* adiciona à sintaxe LTL os quantificadores universal (A) e existencial (E). Deste modo, as fórmulas CTL* podem ser divididas em dois tipos: *fórmulas de estado*, i.e. expressões sobres estados, e *fórmulas de caminhos*, i.e. propriedades que devem ser mantidas ao longo dos caminhos.

Assim, considerando um conjunto de proposições atómicas P, a sintaxe das fórmulas de estado de CTL* é:

- *verdadeiro*, *falso* ou qualquer $p \in P$ são fórmulas de estado;
- $\neg f$, $f \wedge g$, $f \vee g$ são fórmulas de estado, assim como f, g;
- Af or Ef são fórmulas de estado, sendo f uma fórmula de caminho.

A sintaxe das fórmulas de caminho é:

- Qualquer fórmula de estado f, é uma fórmula de caminho;
- $\neg f$, $f \wedge g$, $f \vee g$, Xf, Ff, Gf, fUg or fRg são fórmulas de caminho, sendo f, g igualmente fórmulas de caminho;

Sendo f uma fórmula de caminho, Af é válido num estado se f for válido em todos os caminhos com origem nesse estado, e Ef é válido se

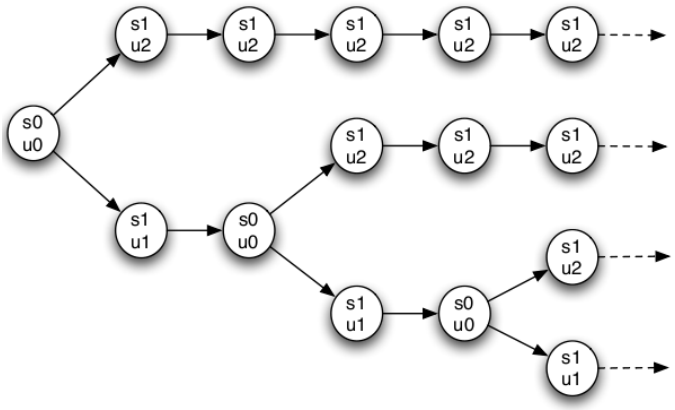

Figura 2.2: Modelo CTL (figura retirada de [19])

existir pelos menos um caminho com origem nesse estado e para o qual f é válida. Qualquer fórmula LTL f é semelhante a Af no CTL*.

A noção de satisfação de fórmulas CTL* é análoga à apresentada na secção anterior para as fórmulas LTL.

Apesar do CTL* ser a lógica temporal mais expressiva, normalmente usa-se um subconjunto desta denominado CTL. No CTL qualquer operador temporal (X,F,G,U ou R) deve ser precedido por um quantificador de caminho (A ou E). O uso deste subconjunto revela-se mais vantajoso pois apesar de ser mais limitativo, permite construir métodos de verificação mais eficientes.

LTL vs. CTL

Apesar do facto de ser possível exprimir a maioria das propriedades tanto em LTL como CTL, estas lógicas são incomparáveis, no sentido em que ambas conseguem exprimir propriedades que a outra não consegue.

Uma vez que em CTL o tempo é representado por uma árvore computacional, consegue-se especificar propriedades relacionadas com a existência de alternativa na execução. No entanto, dado que o aninhamento de quantificadores de caminho é proibido, a expressividade fica naturalmente mais limitada. Apesar disso, e como já foi referido, a principal vantagem do uso do CTL deve-se ao facto do algoritmo de *model checking* ser muito mais eficiente do que o existente para o modelo LTL.

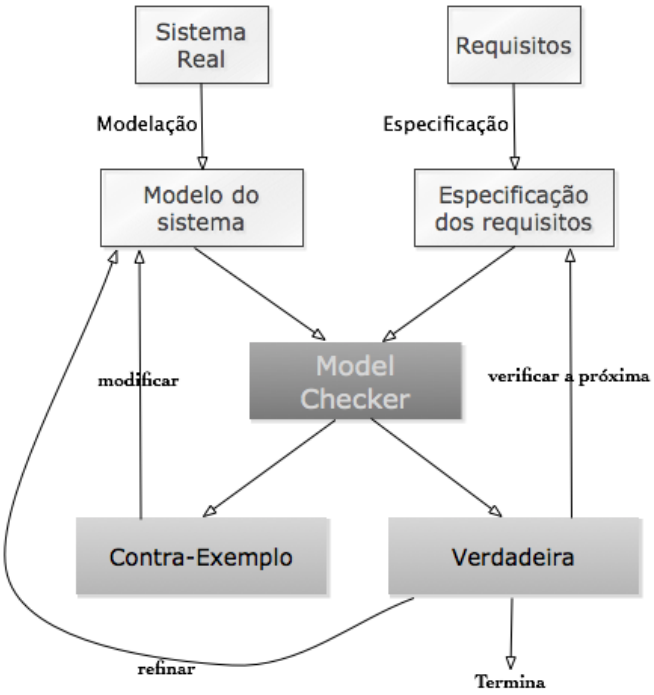

Figura 2.3: Metodologia de verificação baseada na utilização do um *model checker*

2.5 Model Checking

O *model checking* [34] [33] [31] ou em português verificação de modelos, pode definir-se como um técnica algorítmica para verificar se um dado modelo representativo de um sistema está de acordo com uma especificação. Criado inicialmente para lidar com problemas ligados à correcção de hardware, depressa se alastrou a outros campos das ciências da computação e engenharia informática, como por exemplo a verificação de software e protocolos de comunicação.

O que torna verdadeiramente possível a utilização de métodos deste género é a existência de ferramentas informáticas que implementam um algoritmo de verificação de modelos. Assim, o utilizador terá apenas de passar como *input* à ferramenta, o modelo descritivo do sistema e a respectiva especificação lógica dos requisitos, e esperar que a máquina lhe devolva o resultado (ver figura 2.3). Um facto que torna a utilização do *model checking* ainda mais interessante, é a possibilidade de obter contra-exemplos quando a verificação falha, e dessa forma proceder à alteração/correcção do modelo, se for adequado.

De uma forma geral, em termos técnicos os algoritmos de *model chec-*

king implementam basicamente uma procura exaustiva no espaço de estados do modelo de modo a testar se cada estado do modelo "se comporta da maneira correcta", i.e. se o estado satisfaz a propriedade que se pretende verificar. Existem basicamente duas abordagens na utilização do *model checking* que diferem essencialmente na forma como é descrita a especificação dos requisitos:

1. **Abordagem heterogénea (baseada em lógica)**: Nesta abordagem o comportamento do sistema é capturado por um conjunto de propriedades escritas numa lógica apropriada, tipicamente lógica temporal ou lógica modal;

2. **Abordagem Homogénea (baseada no comportamento)**: Nesta abordagem o comportamento desejado e o comportamento possível são descritos na mesma notação, por exemplo autómatos, definindo-se depois relações de equivalência que são usadas como critérios de correcção.

Como em tudo, existem prós e contras na utilização desta técnica. Assim, as principais vantagens e as principais limitações do uso do *model checking* são:

Vantagens:

- Suporta verificação parcial: Um modelo pode ser verificado segundo uma especificação parcial dos requisitos, usando para isso um subconjunto dos requisitos;

- Casos de estudo demonstram que a incorporação do *model checking* na fase de desenho do sistema, não provoca mais atraso no processo do que a utilização de simulação e/ou teste;

- Os *model checkers* podem ser potencialmente usados de forma rotineira, tal como se usa por exemplo compiladores. Isto deve-se em grande parte ao facto dos *model checkers* não requererem um grande grau de interacção com o utilizador;

- O rápido aumento de interesse por parte da indústria, faz com que cada vez mais surjam oportunidades de trabalho em que é extremamente útil possuir experiência nestas ferramentas;

- Bases matemáticas sólidas: modelação, semântica, teoria da concorrência, lógica, teoria de autómatos, estruturas de dados, algoritmos de grafos, etc. são áreas com forte presença da matemática e que constituem as principais bases do *model checking*.

Limitações:

- O *model checking* não é muito apropriado para aplicações com estruturas de dados complexas, uma vez que o tratamento de dados leva geralmente à geração de espaços de estados infinitos;
- A aplicação do *model checking* está sujeita a problemas de decidibilidade;
- Não é fácil garantir que o sistema final implementado respeite as mesmas propriedades abstraídas no modelo;
- Não existe nenhuma garantia acerca da completude das propriedades verificadas;
- Encontrar a abstracção certa requer alguma experiência;
- A utilidade de um *model-checker* é completamente dependente da correcção do algoritmo de verificação por ele implementado;
- Em geral não é possível generalizar resultados de verificação, por exemplo a verificação com sucesso da correcção de um protocolo para 1, 2, ou 3 processos não implica a correcção do mesmo sistema para n processos.

No contexto desta tese de mestrado foi efectuado algum trabalho de familiarização com ferramentas que implementam algoritmos de *model checking*, das quais se destacam sumariamente as seguintes:

- RT-Spin - Extensão da linguagem promela com noções temporais e Lógica LTL com asserções [30] [29] [39]. Adequado para sistemas concorrentes interactivos;
- Kronos e Uppaal - Autómatos temporizados (Lógica TCTL). Adequados para sistemas de tempo real;
- SMV - Autómatos temporizados (Lógica CTL). Adequado para sistemas concorrentes;
- HyTech - Autómatos Híbridos Lineares. Adequado para sistemas embebidos críticos.

Durante os últimos anos têm surgido inúmeras publicações com diversos casos de estudo onde se aplica o *model checking*, o que vem demonstrar o crescente interesse nestas ferramentas como alternativa ao teste para a validação sistemas.

2.6 Software Model Checking

O *software model checking* é a aplicação directa dos algoritmos tradicionais de *model checking* na área da verificação de software [22]. Na verdade, o *model checking* encontrou na área da verificação de software o seu principal motor de investigação e evolução, e por isso mesmo é possível encontrar diversos casos de sucesso na aplicação deste método ao software.

Neste contexto, torna-se importante em primeiro lugar apresentar a definição de um modelo de um programa. Um modelo define-se como um conjunto de estados e transições. Por sua vez, um estado possui a informação sobre o *program counter*, as variáveis, e as configurações da *stack* e da *heap*. Por outro lado as transições descrevem a forma como um programa evolui de um estado para outro.

Posto isto, graças às ferramentas que implementam algoritmos de *model checking* é possível verificar dois grandes tipos de propriedades:

- ***Safety***: Propriedades que especificam se um dado estado considerado não desejável, é atingido. Como exemplos destes estados temos os apontadores nulos, ou índices fora dos limites dos *arrays*.
- ***Liveness***: As propriedades de *liveness*, especificam a manutenção do bom comportamento do programa. Exs: Um pedido ao sistema terá invariavelmente uma resposta; Um programa deve terminar.

No entanto, apesar dos diversos casos de sucesso, continuam a existir alguns problemas dos quais o principal é da explosão de estados [20]. O espaço de estados de um programa é exponencial e influenciado por vários parâmetros como o número de variáveis ou o tamanho dos tipos de dados. Este pode ser mesmo infinito na presença de chamadas a funções, alocação dinâmica de memória ou concorrência.

Os algoritmos de *model checking* geram conjuntos de estados para serem analisados, e esses estados tem de ser armazenados de forma a garantir que cada estado é visitado pelo menos uma vez. Existem actualmente duas categorias de métodos para representar estados, e que tentam ter em conta certas abstracções de modo a reduzir o problema da explosão de estados. Essas categorias são a *Explicit-State Model Checking* e a *Symbolic Model Checking*.

- **Explicit-State Model Checking** - Os algoritmos de *model checking* inseridos nesta categoria procedem à geração de um grafo de forma recursiva, podendo esse grafo ser construído de três formas

diferentes, *breadth-first*, *depth-first* ou de forma heurística [22]. Uma vez que a propriedades vão sendo testadas a medida que o grafo é construído, não é necessário ter o grafo todo construído para se obterem resultados da verificação, o que quer dizer que se for encontrado um contra-exemplo para a propriedade testada o processo termina sem ter sido necessário gerar o grafo todo.

- **Symbolic Model Checking** - Os métodos desta categoria, ao contrário dos algoritmos de *explicit-state model checking* representam conjuntos de estados em vez de enumerar individualmente cada estado. As representações mais usuais do *symbolic model checking* são os chamados diagramas de decisão binária (BDDs - Binary Decision Diagrams) [7]. Com este método, é possível verificar modelos com mais de 10^{20} estados, enquanto que os *explicit-state model checking* escalam apenas para alguns milhares estados.

Em suma, as técnicas de *symbolic model checking*, funcionam melhor na prova da correcção do sistema, assim como lidam melhor com o problema da explosão de estados. Por outro lado, as técnicas de *explicit-state model checking* lidam melhor com a detecção de erros e com a concorrência.

De modo a reduzir o problema de explosão da estados, ambas a técnicas recorrem a abstracções para tentar reduzir o tamanho do espaço de estados. No início, essas abstracções eram introduzidas manualmente por quem modela, no entanto as ferramentas mais modernas já conseguem introduzir automaticamente muitas abstracções.

A técnica de abstracção mais predominante hoje em dia é chamada de abstracção de predicados ou em inglês *predicate abstraction*. Esta foi promovida pelo sucesso da ferramenta SLAM [3], e usa predicados lógicos para construir um domínio abstracto através da partição do espaço de estados. A principal diferença desta técnica em relação à interpretação abstracta clássica, reside no facto de esta ser parametrizada e especifica para um programa em concreto. Assim, se a execução do algoritmo de *model-checking* sobre o modelo abstraído encontrar um contra-exemplo, este muito provavelmente não existe no programa concreto. Mais ainda, esse contra-exemplo pode ser usado para identificar novos predicados e obter abstracções mais precisas.

2.7 Uppaal

O Uppaal é uma ferramenta de modelação, validação e verificação de sistemas de tempo real sobre redes de autómatos temporizados e estendidos

com tipos de dados. Criada em 1995 [35] numa parceria das Universidades de **Upp**sala e **Aal**borg, a ferramenta tem sofrido contínuas actualizações e por isso mesmo revela-se neste momento uma das alternativas mais viáveis para a verificação deste tipo de sistemas.

Autómatos Temporizados em Uppaal

No Uppaal, um sistema é uma rede de autómatos temporizados, estendidos com variáveis discretas. Assim, o estado do sistema é dado pelo conjunto das localizações[3] dos autómatos, os relógios, e os valores das variáveis.

Existem outras características que os autómatos temporizados do Uppaal estendem relativamente à noção introduzida na secção 2.3 , nomeadamente as seguintes:

Templates Autómatos em Uppaal podem receber variáveis como parâmetros para serem usadas em expressões;

Constants Variáveis com valores constantes;

Bounded variables Os valores permitidos para uma variável inteira podem ser limitados;

Binary synchronization São usados canais para proceder à sincronização dos autómatos;

Broadcast channels É possível difundir um sinal para todos os autómatos que estejam atentos a um determinado canal;

Urgent synchronization Não podem ocorrer atrasos quando é possibilitada a transição em canais urgentes;

Urgent locations Quando um autómato se encontra num estado urgente o tempo não pode ser incrementado;

Committed locations Estes são mais restritos do que as localizações urgentes, por isso além do tempo não poder ser incrementado, a transição seguinte deverá abandonar o estado *commited*;

Arrays Os vectores poderão guardar canais, relógios ou inteiros;

O Uppaal também é capaz de interpretar expressões sobre relógios e variáveis inteiras, usando para isso as seguintes etiquetas:

[3] Entende-se por localização o estado em que o autómato se encontra em determinado momento

Guard Expressões que avaliam em valores booleanos e são livres de efeitos laterais;

Synchronization Recepção $Exp?$ ou envio $Exp!$ de sincronizações livres de efeitos laterais;

Assignment Lista de expressões com atribuições;

Invariant Conjunção de condições da forma $x < e$ onde x é um relógio e e avalia para um inteiro.

O *model checker* Uppaal usa uma forma simplificada da lógica CTL. Na verdade, além de não existirem os quantificadores X, U e R, o aninhamento de fórmulas de caminho não é permitido. Assim, à semelhança do CTL, a linguagem consiste basicamente em fórmulas de estados e fórmulas de caminhos, podendo estas últimas ainda serem classificadas em quatro classes: *state formulæ*,*reachability*, *safety* e *liveness*.

State Formulæ Estas fórmulas testam o estado do sistema sem ser necessário olhar para o seu comportamento. São representadas por expressões de guarda, o que quer dizer que são livres de efeitos laterais. Permitem ainda verificar se um autómato P se encontra numa determinada localização l ($P.l$). O Uppaal disponibiliza ainda uma outra fórmula especial *deadlock* que é satisfeita quando não existem transições a partir do estado;

Reachability Properties Estas expressões testam se uma dada fórmula f pode ser satisfeita em algum estado atingível. Em CTL esta expressão pode ser escrita da forma EFf e em Uppaal `E<>f`;

Safety Properties As propriedades de *Safety* testam se uma dada fórmula pode ser satisfeita em algum estado atingível no sistema (AGf em CTL, `A[] f` em Uppaal).

Liveness Properties Testam se algo será inevitavelmente verdade (ou nunca). Expressa-se em CTL da seguinte maneira AFf e em Uppaal `A<> f`. No entanto existem algumas propriedades do género "se f é verdade então g será inevitavelmente verdade", que pela proibição de aninhamento de quantificadores imposta no Uppaal, não se consegue exprimir. Para isso foi introduzido no Uppaal o operador `f --> g`, que em CTL significa $AG(f \Rightarrow AFg)$.

De modo a ilustrar as características acima descritas, apresentamos de seguida um exemplo de aplicação do Uppaal na modelação e verificação de um sistema concreto.

Resumidamente, o sistema que se descreve a seguir é uma máquina de venda de café, através do qual um utente poderá introduzir a moeda e aguardar a preparação do mesmo. O sistema terá de seguir a seguinte especificação (ver também figura 2.4):

- O utente tenta repetidamente: inserir uma moeda, extrair o café servido e avisar o observador que obteve o que pretendia. Entre cada acção o utente espera durante um certo tempo antes de continuar.

- Depois de receber uma moeda, a máquina leva um determinado tempo para preparar o café. Se o café não for levantado pelo utente após um determinado prazo, a máquina deverá reagir e entrar em *timeout*.

- O observador deverá emitir uma queixa se o intervalo de tempo entre dois avisos do utente for superior a 8 unidades de tempo.

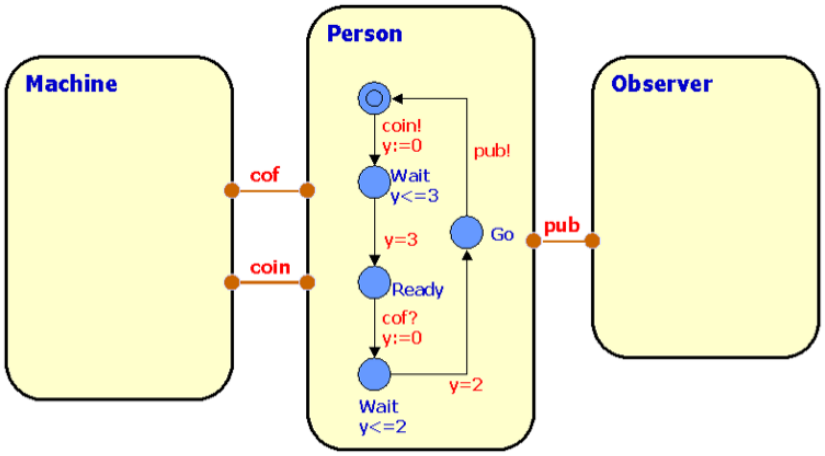

Figura 2.4: Arquitectura da máquina de venda de café (figura retirada de [38]).

Analisando bem a especificação anterior, percebe-se que esta não está completa, i.e se pegarmos na condição que diz que se o utente não levantar o café, a máquina deve entrar em *timeout*, percebe-se que o sistema irá entrar em *deadlock* quando isto acontecer, pois não foi especificado o

que a máquina deve fazer para evitar isso. A solução passaria por entregar o café, mas só depois do *timeout*, o que em termos práticos poderia significar colocar o café numa zona de entrega particular.

Na imagem 2.5 apresentam-se os autómatos Uppaal que modelam o sistema apresentado.

Através destes autómatos é possível ver a utilização de alguns dos mecanismos do Uppaal descritos nesta secção, nomeadamente sincronizações, guardas nas transições, actualização de variáveis, entre outros. Quanto à verificação de propriedades, neste exemplo podemos verificar duas propriedades essenciais: se o sistema irá entrar em deadlock (A[] deadlock); e em que situações o observador terá de fazer uma queixa (E<> Observer.Complain). Recordando a classificação das propriedades também descritas nesta secção, estas pertencem às classes *liveness Properties* e *Reachability Properties* respectivamente.

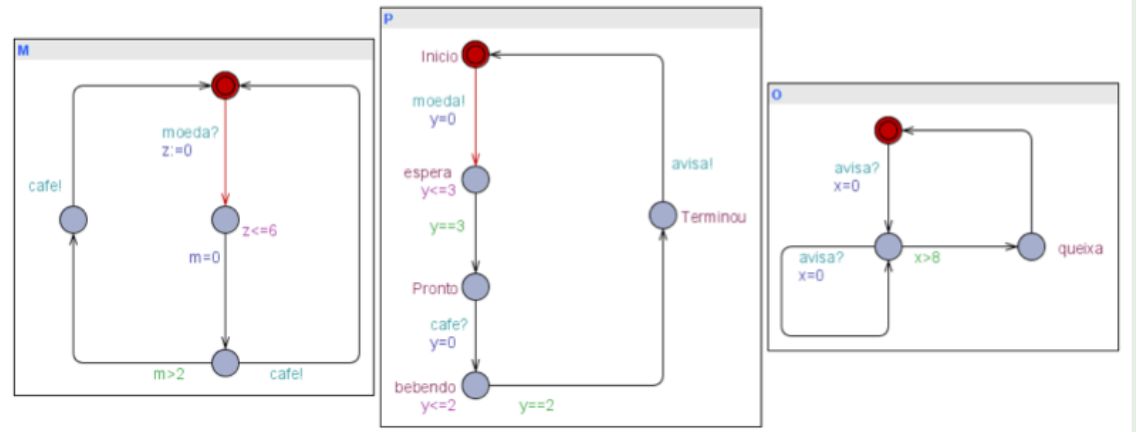

Figura 2.5: Autómatos Uppaal que modelam o sistema da máquina de venda de café (figura retirada de [38]).

Ferramentas Uppaal

A criação do Uppaal teve como principal objectivo fornecer ao utilizador uma ferramenta que permita a modelação de autómatos, assim como a simulação e a verificação de propriedades. Sabendo que a experiência que o utilizador obtém no uso da ferramenta é importante para o sucesso da mesma, eliminou-se a necessidade de usar a linha de comandos (tal continua a ser no entanto possível) para utilizar a ferramenta. Foi desenvolvida uma interface gráfica apelativa que permite tirar partido de quase todas as funcionalidades do Uppaal.

Assim, o *editor* é uma interface gráfica que permite a modelação de autómatos e redes de autómatos temporizados de forma fácil e rápida. O *simulador* ajuda a seguir a execução dos autómatos e desse modo verificar se o sistema se está a comportar da maneira prevista, permitindo a correcção de *bugs*. Finalmente, o *verificador* permite a especificação de propriedades em lógica temporal, e com o simples gesto de carregar num botão testar a validade das propriedades. Importante é o facto de no caso de alguma propriedade não se verificar, ser possível pedir ao Uppaal que produza um caminho de execução onde tal erro aconteça.

No entanto, para verificação de maior escala, poderá ser necessário ter mais controlo sobre o *model checker*. Para estas situações, recorre-se à linha de comandos invocando o comando `verifyta` (correspondente à invocação do *model checker*). A principal vantagem de proceder desta forma é o facto de se poder passar parâmetros para controlar a verificação, tais como indicar a representação do espaço de estados ou a memória usada.

2.8 Métodos de Verificação para Sistemas de Tempo Real

Como é sabido, é notório o aumento de interesse na validação formal de programas, especialmente quando o problema abarca sistemas de tempo real críticos. Nos últimos anos foram publicadas algumas propostas teóricas na tentativa de responder a este problema, sendo que algumas delas serviram de inspiração para novas técnicas e métodos. Nas secções seguintes serão apresentados vários trabalhos que representam avanços significativos na investigação, e que se relacionam directamente com o tema desta dissertação.

Finalmente, é importante referir que esta validação formal é dividida em diferentes classes, e portanto há todo um outro mundo de problemas que não são englobados no tema desta dissertação, como por exemplo a verificação de *schedulabity* [37] ou WCET (*Worst Case Execution Time*) [41]. Assim sendo, existem inúmeros trabalhos publicados que não irão ser abordados no contexto desta tese, mas que são igualmente importantes pois ajudam a implementar outros aspectos da verificação de programas de tempo real.

Abordagens baseadas em Anotações ao Código

O Comando Deadline No artigo [23] Fidge et al. propõem um comando para ser usado como anotação numa linguagem de programação genérica. Como descrito na introdução desse artigo, as motivações dos autores para esse trabalho foram as dificuldades mais comuns encontradas na validação do comportamento temporal dos programas. Os principais objectivos do comando *deadline* são:

- Exprimir limites temporais superiores, complementando a instrução *delay* existente em algumas linguagens, que permite exprimir limites inferiores de tempo;
- Ser suficientemente flexível para ser aplicado a qualquer linguagem de programação imperativa;
- Servir de directiva de compilação para análise estática, e desse modo garantir à priori a correcção temporal em tempo de execução;

Além disso, os autores desenvolveram ainda dois comandos auxiliares, o comando de *asserção* e as *constantes lógicas*. O primeiro é usado para especificar propriedades que se espera serem verdade em determinado ponto do programa, e o segundo permite a declaração de variáveis auxiliares usadas para construir expressões de tempo a serem passadas como argumento ao comando *deadline*.

Uma vez que estas construções são passíveis de serem verificadas (ou pelo menos foram criadas com esse intuito), os autores acreditam que o conjunto destes comandos permite aos programadores definir quase todo o comportamento temporal dos programas, e desse modo deixar o teste para segundo plano. No entanto, dado que isto são apenas ideias teóricas, é necessário construir uma implementação/algoritmo capaz de lidar com estes conceitos, o que se pode revelar uma tarefa difícil. Posto isto, é possível encontrar também no artigo uma proposta de algoritmo dividida em três fases:

- Primeira fase: Análise do fluxo de controlo (Fig. 2.6) de forma a encontrar todos os caminhos de execução que terminem num comando *deadline*;
- Segunda fase: Para cada um dos caminhos de execução irá ser calculada uma restrição temporal. No cálculo dessa restrição temporal entram os tempos especificados por todos os comandos presentes nesse caminho de execução.

- Terceira fase: Com o uso de técnicas convencionais de previsão de WCET (*Worst-case execution time*) aplicada ao código máquina gerado pelo compilador, tenta-se provar que o código será suficientemente rápido para executar dentro dos limites impostos/calculados.

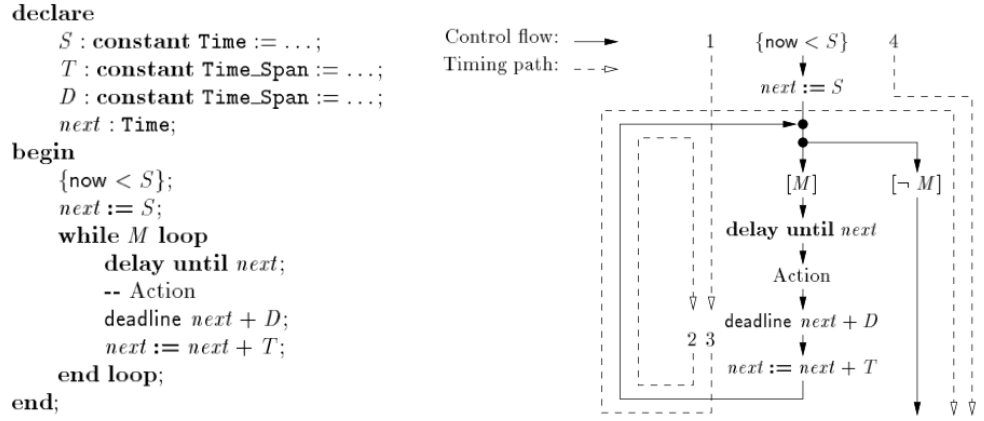

Figura 2.6: Programa periódico com o respectivo grafo de fluxo de controlo (Figura retirada de [23]).

O ambiente SPARK, por possuir características comuns com esta abordagem, é identificado pelos autores como o ponto de partida ideal para instanciar estas ideias. Escrever anotações não executáveis em forma de comentários especiais, recorrer a asserções sobre as variáveis do programa, usar constantes nas anotações ou até usar uma ferramenta (*Examiner*) para retirar informação sobre o fluxo de controlo do programa, são apenas algumas das características do SPARK em comum com esta abordagem, e que demonstram o quão próxima é a filosofia de ambos os projectos. Apesar disso, os autores avisam que poderão existir restrições capazes de hipotecar a implementação destas ideias, como a inexistência de suporte para algumas das construções de tempo real, e portanto será necessário investir algum esforço na adaptação do SPARK. Na verdade, de modo a tornar mais robusto este novo conceito, seria importante incluir nas técnicas de análise estática presentes no SPARK, a capacidade de lidar com programas multi tarefa, que obviamente trarão outras implicações no comportamento temporal dos programas.

Verificação do Perfil Ravenscar em Programas Anotados Temporalmente Alan Burns, um dos investigadores mais influentes no mundo do tempo real, desenvolveu juntamente com a sua equipa diversos trabalhos a todos os níveis desde a modelação à verificação. É notória a

preocupação destes investigadores no que diz respeito à verificação e validação de programas de tempo real, e por isso mesmo têm publicados vários artigos com propostas para tratar esta lacuna.

Um dos trabalhos mais interessantes deste grupo de investigação é o artigo [10], que se baseia principalmente no artigo descrito anteriormente (sobre o comando *deadline*) e no SPARK, e que representa um avanço significativo nesta área. Como descrito na secção anterior, o comando *deadline* prima pela simplicidade que oferece na especificação de propriedades temporais dos programas, e por isso mesmo os autores reaproveitam essa ideia. Apesar da pedra basilar deste artigo ser o artigo [23], existem algumas diferenças entre as duas abordagens, em particular temos como exemplos o facto do uso do comando *deadline* ser permitido no ficheiro de especificação dos programas Ada, ou a diferença entre as constantes lógicas do artigo anterior e as variáveis auxiliares usadas aqui, que diferem essencialmente no facto destas últimas poderem ser alteradas ao longo do programa, ao contrário das constantes lógicas.

O trabalho teórico deste artigo permitiu a inclusão de dois novos comandos nesta abordagem, o comando *rely* e o comando *guarantee*. Estes foram introduzidos pela primeira vez por Peter John Whysall [40], e têm como objectivo permitir a especificação de dependências entre tarefas e objectos protegidos eventualmente usados por estas. No entanto, de modo a enriquecer estes dois comandos, Burns e Lin estenderam estes dois comandos com a flag *with_period* para impor um limite superior à dependência, e a flag *with_freshness* para exprimir a repetição da operação.

Posto isto, e apesar desta ideia ser totalmente válida, a implementação descrita nesse artigo ainda não foi concretizada. No entanto, dado que Alan Burns contribuiu com algumas ideias no desenvolvimento do ambiente SPARK, e por isso estar bastante familiarizado com a ferramenta, é natural que o SPARK seja considerado o ponto de partida ideal para a implementação destas ideias. Assim sendo, a proposta dos autores para a implementação destes conceitos é descrita no artigo e contém essencialmente um processo com duas fases.

- A primeira fase deve verificar a consistência das anotações, i.e. se uma operação é especificada como tendo um *deadline* de 20 ms enquanto o código fonte apresenta uma instrução de *delay* de 30 ms, é óbvio que existe uma inconsistência.
- Na segunda fase, recorrendo ao *model checking* tenta-se validar a preservação do comportamento temporal, mas para isso é necessário primeiro proceder a uma tradução (manual ou automática) do código fonte para autómatos temporizados do Uppaal.

```
1.  ...
2.
3.  loop
4.     Device := smart;
5.
6.     if Device = smart then
7.
8.        Current := Clock;
9.
10.       delay until Current+Milliseconds(30);
11.
12.       Input_Data.Data.Write(Data);
13.
14.    else
15.
16. ...
```

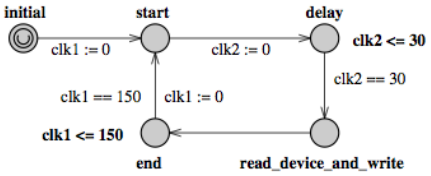

Figura 2.7: Ilustração da abordagem de Alan Burns (Excerto e imagem retirados de [10]).

Na figura 2.7 temos uma ilustração da técnica utilizada por Alan Burns. A ideia é analisar o código fonte do programa em questão, e descrever o comportamento sob a forma de um modelo Uppaal, tendo sempre em atenção as características temporais. Na figura, o autómato corresponde ao excerto de código apresentado, e se analisarmos a sua semântica constatamos que o modelo descreve o mesmo comportamento. Assim sendo, o grande objectivo desta tese é contribuir para tornar possível automatizar este processo.

Para detalhes da tradução do Ada para Uppaal, é recomendada a leitura do artigo em questão [10]. Essa leitura permite identificar facilmente a segunda fase do processo como um possível ponto de falha, pois é difícil garantir que o modelo gerado seja uma abstracção correcta do código. Tirando isso, e principalmente tendo em conta a visão futura dos autores para esta abordagem onde propõem um processo automático completo (extracção de um modelo, verificação do modelo e correcção do código), devemos considerar esta ideia um passo importante na investigação.

Finalmente, não podemos deixar de referir dois trabalhos dos mesmos investigadores. O primeiro [11] é basicamente uma versão longa deste artigo, que descreve de forma detalhada e com muitos casos de estudo todo o processo proposto. O segundo [12] é um trabalho extenso que apresenta uma análise completa de diferentes técnicas e dicas que resultam no desenvolvimento de programas correctos, i.e. desde a fase de modelação, passando pelo teste e até à verificação de programas de tempo real.

Abordagens Baseadas em Extracção Automática de modelos

Verificação Automática de Sistemas de Tempo Real Na sua tese de mestrado [16] Joel Carvalho introduz uma ferramenta capaz de extrair

automaticamente autómatos Uppaal a partir de programas HTL. Como referido na Secção 2.2, o HTL é usado para coordenar e especificar o comportamento temporal de programas escritos em diferentes linguagens que não disponham de primitivas para desenvolver programas de tempo real. Neste trabalho a ideia é estender a cadeia de verificação (figura 2.8)[4] oferecida pelo HTL, através de uma ferramenta chamada HTL2XTA desenvolvida durante a tese, e que como já foi dito permite uma extracção automática de autómatos temporizados Uppaal a partir de código HTL.

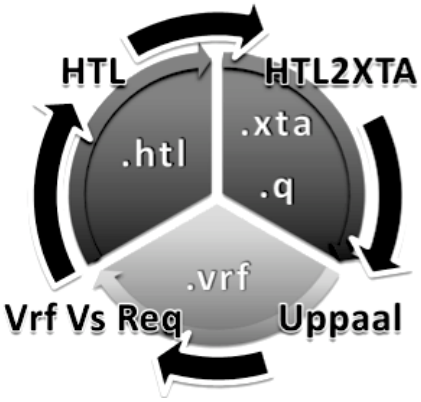

Figura 2.8: Esquema da cadeia de ferramentas que engloba o HTL2XTA

A cadeia de ferramentas do HTL permite uma análise estática na fase de compilação dos programas desenvolvidos. No entanto, dado que o HTL apenas trata questões de *schedulability*, a *toolchain* actual não é suficiente para garantir a correcção do código. Assim a abordagem de Joel Carvalho introduz a possibilidade de analisar o comportamento temporal das tarefas, complementando o processo. O autor criou ainda um algoritmo capaz de inferir propriedades automaticamente, o que no entanto não invalida a possibilidade de podermos simular o modelo e adicionar outras propriedades aquando da utilização da ferramenta Uppaal. Os detalhes deste trabalho são descritos no artigo [15].

Esta abordagem é vista como uma excelente alternativa para ajudar os programadores no desenvolvimento de programas de tempo real e ao mesmo tempo na validação do comportamento temporal dos programas. Durante o desenvolvimento facilmente se consegue gerar modelos e propriedades para serem validadas com recurso ao *model checker* do Uppaal.

[4]Figura importada de [15]

2.9 Conclusão

Neste capítulo foram apresentados os conceitos teóricos mais importantes, bem como trabalhos existentes com influência no desenvolvimento do trabalho de mestrado que resultou nesta dissertação. Atentando às abordagens existentes, é possível constatar que estas podem ser divididas em duas classes: uma baseada em anotações prévias ao código, e outra numa extracção automática, no entanto ambas levam à geração de modelos para se poder verificar propriedades temporais com o auxílio de *model checkers*.

Não é possível destacar uma das abordagens como a melhor, especialmente pelo facto de nenhuma delas estar esgotada em termos de investigação. No entanto, se quisermos definir uma métrica que classifique a melhor, é possível dizer alternativamente que a melhor é a que gera modelos que permitam verificar mais propriedades, ou então a que oferece mais garantias quanto ao motor de tradução: se a abordagem for verificada formalmente, isto significa que podemos confiar mais no seu funcionamento.

A leitura dos artigos que introduzem as abordagens discutidas anteriormente mostra que existe ainda muito trabalho a ser desenvolvido nesta área, e como constatado o SPARK possui um papel muito importante no avanço desta investigação. Assim, surge neste contexto a proposta de dissertação que agora se apresenta, e que bebe inspiração essencialmente no trabalho de Joel Carvalho, combinado com algumas técnicas propostas por Alan Burns.

Resumindo, apesar de já existirem actualmente abordagens muito interessantes, nenhuma delas consegue responder a cem por cento às necessidades actuais, especialmente no mundo que combina o Ada e os sistemas de tempo real. Assim, o esforço na investigação destes temas deve ser continuado, e sabendo que existem cada vez mais empresas a procurar respostas a este problema, esta preocupação faz ainda mais sentido. Parece ser recomendável no panorama actual efectuar parcerias/consórcios que consigam acelerar a procura de respostas conjuntas.

Capítulo 3

Método de Verificação

Após um estudo detalhado dos mecanismos de verificação de sistemas de tempo real actualmente existentes, foi possível identificar uma área onde se julgou ser interessante desenvolver um contributo para melhorar os mecanismos actuais. Nesse estudo ficou claro que a linguagem Ada, pelo papel importante que representa neste meio e pelas razões já referidas no capítulo 1 , seria o alvo ideal. Mais ainda, pelas provas dadas em diversos projectos industriais bem como pelas suas características únicas para modelação e verificação de programas de tempo real, identificou-se também o Uppaal como a ferramenta mais adequada ao esquema de verificação que se pretendia implementar. Deste modo, o objectivo a que nos propusemos foi desenvolver uma ferramenta capaz de extrair modelos Uppaal (autómatos temporizados) a partir de programas de tempo real desenvolvidos na linguagem Ada, permitindo a verificação de propriedades.

Como se pode constatar, esta decisão foi bastante inspirada nos trabalhos de Alan Burns e Joel Carvalho: muitas das estratégias usadas em todo o esquema de tradução são adaptadas desses mesmos trabalhos. Como primeiro exemplo temos o facto de nos trabalhos de Alan Burns encontrarmos vários exemplos de código Ada que foram por nós adoptados na tradução dos programas. Por outro lado, temos também as anotações usadas igualmente nesses trabalhos, e que são fundamentais para o sucesso da tradução, pois permitem a especificação de parâmetros (temporais ou outros) a serem reflectidos no modelo obtido e que tornam possível a utilização desse modelo para a validação de propriedades. Já o trabalho de Carvalho, foi importante nos aspectos relativos à tradução, i.e. tornou possível ter em conta aspectos técnicos e evitar problemas surgidos na altura, de forma a acelerar o processo.

No que diz respeito ao esquema de tradução, foi determinante perceber

através da análise de outros trabalhos, como é feita normalmente a tradução manual de código Ada para modelos Uppaal. Neste ponto, os trabalhos de Alan Burns foram mais uma vez determinantes pois permitiram perceber que por norma os modelos obtidos reflectem o fluxo de controlo do programa. Assim sendo, a estratégia usada baseia-se exactamente no conceito de grafo de fluxo de controlo (CFG), a partir do qual é possível através de manipulações baseadas nas anotações, obter o modelo Uppaal final.

Resumidamente, o algoritmo implementado processa-se em duas fases: uma primeira em que é carregada a árvore de sintaxe abstracta do código fonte e que permite construir o CFG 3.3; e uma segunda fase em que se processa o CFG gerado e conforme as anotações encontradas se procede às transformações necessárias até se chegar ao modelo Uppaal pretendido 3.4.

Por último resta dizer que a ferramenta foi totalmente desenvolvida usando a linguagem Java, escolha essa justificada pelo facto de a equipa possuir maior experiência no uso da mesma e também pelo enorme conjunto de ferramentas e bibliotecas disponíveis, como por exemplo a biblioteca de carregamento e processamento de documentos XML, formato utilizado para a representação das AST's.

Exemplo Motivador. A ferramenta cujos detalhes são explicados nas secções seguintes, é capaz de traduzir grande parte das construções sintácticas existentes na linguagem Ada. Apesar da ferramenta não verificar propriamente os programas Ada, possibilita de forma fácil, rápida e intuitiva a geração de modelos XTA passíveis de serem explorados pelo *model-checker verifyta* embutido no Uppaal. Além disso, graças a todas a funcionalidades do Uppaal é possível ainda analisar, editar e explorar passo a passo o comportamento do modelo gerado, levando a uma melhor compreensão dos problemas. Assim, para além de o modelo ser traduzido com o intuito da utilização de um *model-checker* na verificação de certas propriedades temporais, a escolha do Uppaal permite obter sem custos adicionais um conjunto de funcionalidades igualmente importantes para quem desenvolve programas de tempo-real.

De forma a motivar para a leitura dos conteúdos presentes nas secções seguintes, temos na figura 3.2 um exemplo de um modelo simples gerado pela ferramenta desenvolvida, a partir do programa Ada da figura 3.1. O programa em questão, não é nada mais nada menos do que uma tarefa cíclica que a cada ciclo faz uma operação de I/O.

A primeira fase do processo que leva à geração do modelo, é a construção do grafo de fluxo de controlo do programa. Posto isto, tendo em atenção

Cyclic.adb

```
1.  package body Cyclic is
2.    task body Cyclic_Body is
3.      Next_Period : Time;
4.      Increment : Positive;
5.    begin
6.      -- Initialization code
7.      Cycle_Time := Clock;
8.      Increment := 1;
9.      loop
10.       Ada.Text_IO.Put_Line(Increment);
11.       Cycle_Time := Cycle_Time + Milliseconds(100);
12.       delay until Cycle_Time;
13.   --@deadline Cycle_Time + Milliseconds(100);
14.       Increment := Increment + 1;
15.     end loop;
16.   end Cyclic;
17. end Cyclic;
```

Cyclic.ads

```
1.  with Ada.Real_Time;
2.  use Ada.Real_Time;
3.  package Cyclic
4.
5.  is
6.  task Cyclic_Body;
7.
8.  --@globalClock Cycle_Time;
9.  --@period 100;
10. --@guarantee Ada_Text_IO uses
    Write with_freshness 100;
11.
12. end Cyclic;
```

Figura 3.1: Exemplo Tarefa Cíclica (Ada)

os pormenores do código apresentado, rapidamente se percebe que este (CFG), com excepção das instruções irrelevantes do ponto de vista temporal, se encontra representado no modelo gerado (3.2). Embora simples, existem alguns pormenores nesta primeira fase que carecem de alguma cautela, nomeadamente relacionados com a leitura de informação a partir do documento xml, no entanto iremos detalhar este processo na secção 3.3.

Somente após a construção do CFG e com o auxílio das anotações temporais presentes em ambos os ficheiros (.adb e .ads), é que se torna possível entrar na fase da geração do modelo XTA propriamente dita. Nesta fase, procede-se à iteração do nodos do CFG onde consoante o seu tipo[1], provoca-se uma determinada transformação. A título de exemplo, atentando à figura 3.2, no nodo *Procedure_Call_4* que corresponde no código à chamada do procedimento *Ada.Text_IO.Put_Line()*, o seu tipo levou à geração de uma sincronização com um objecto protegido.

Uma vez que é necessário haver interação entre o modelo gerado e objectos protegidos, de modo a que se possa representar as chamadas de procedimentos e o respectivo impacto temporal, procede-se à geração de um autómato abstracto representativo do objecto protegido necessário (autómato à direita na figura 3.2).

Imediatamente após o término deste processo, novamente graças a algumas anotações temporais, como por exemplo a anotação que indica o período da tarefa (*@period*), foi possível criar um algoritmo que gera algumas propriedades 3.5 representativas do comportamento temporal

[1]Exemplos: Nodo Inicial; Nodo IF; Nodo LOOP; etc. A classificação dos nodos é explicada em maior detalhe na secção 3.4

esperado para o sistema.

Assim sendo, torna-se então possível a partir deste ponto, importar o modelo da ferramenta Uppaal, onde irão ser verificadas as propriedades geradas, cujo resultado nos irá indicar que melhorias/correcções terão de ser feitas ao modelo e consequentemente ao código do sistema em questão.

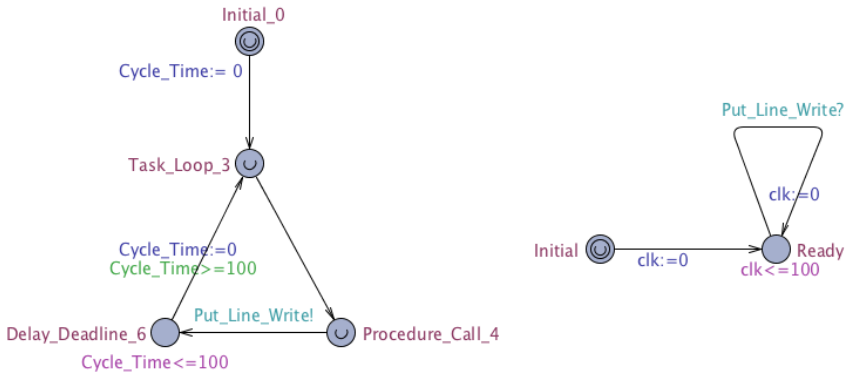

Figura 3.2: Tarefa Cíclica (Modelo Uppaal)

3.1 Processamento sintáctico de código Ada

Na planificação do desenvolvimento, foi necessário em primeiro lugar investigar a melhor forma de se conseguir fazer o *parsing* do código Ada. Na verdade este passo revelou-se uma tarefa mais demorada do que antecipámos, pois o Ada é uma linguagem bastante complexa e com muitas construções, e também porque não existem muitas ferramentas que auxiliem o processo de *parsing*.

Após alguma pesquisa, foi encontrada uma plataforma completamente dedicada a este problema, chamada *ASIS*[2]. Esta *framework* totalmente escrita em Ada tornou-se um standard (ISO/IEC 8652) no mundo Ada e grande parte das ferramentas Ada actualmente existentes assentam sobretudo nesta plataforma. A interface *ASIS* inclui informação semântica e sintáctica de cada unidade de compilação (ver arquitectura na figura 3.3). Através de *queries*, é possível aceder a essa informação sob a forma de uma árvore de sintaxe abstracta (AST), e que contém numa estrutura hierárquica abstracções para todos os componentes Ada. No entanto, a utilização desta *framework* obrigaria à utilização do próprio Ada como

[2]http://www.sigada.org/wg/asiswg/

linguagem de desenvolvimento. Dada a falta de experiência desta equipa no desenvolvimento de programas em Ada, depressa se concluiu que o uso desta alternativa levaria a um processo de desenvolvimento bastante demorado. Ainda assim, graças ao estudo feito ao *ASIS* foi possível descobrir outras ferramentas alternativas. Após o estudo das mesmas, a escolha recaiu sobre a ferramenta *Avatox* (*Ada Via ASIS To XML*), que se descreve em mais pormenor em seguida.

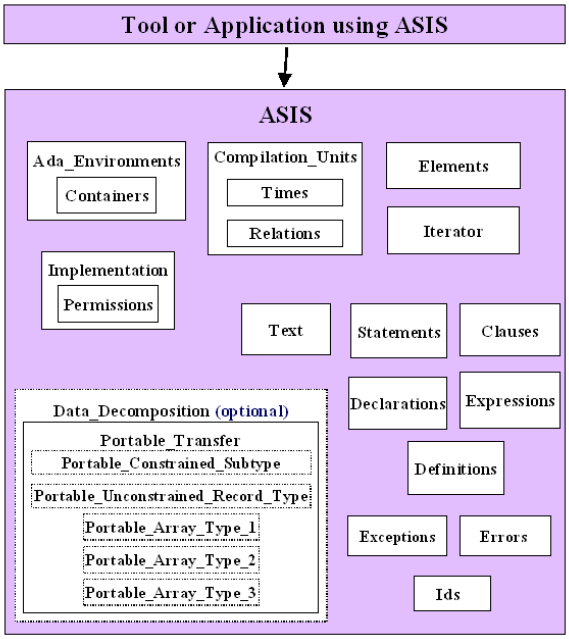

Figura 3.3: Arquitectura da plataforma Asis

O *Avatox*[3] [18] é uma aplicação capaz de fazer travessias sobre unidades de compilação Ada e que retorna a representação *ASIS* dessas unidades, sob a forma de um documento XML. Adicionalmente, o documento XML resultante poderá ter associado um *XSL stylesheet*[4] de modo a permitir personalizar o documento XML da maneira mais adequada ao problema. Dado que o documento XML gerado pelo *Avatox* representa em compreensão todo o conteúdo e organização do código Ada, a tecnologia actual disponibilizada para tratamento de documentos XML permite simplificar o problema inicial de *parsing* de código Ada.

É importante referir o factor determinante para a escolha do Avatox, pois esta não é a única ferramenta existente capaz de fazer a tradução de código Ada para documentos XML. Existe também por exemplo a ferramenta *asis2xml*. No entanto, ao contrário do *Avatox*, os documentos

[3] http://www.mckae.com/avatox.html
[4] http://www.w3.org/

XML gerados pelo *asis2xml* não incluem os comentários existentes no código, o que invalidou o seu uso uma vez que os comentários têm um papel fulcral na estratégia de tradução (as anotações ao código são incluídas como comentários). Além disso, apesar de os documentos XML gerados pelo *Avatox* serem mais complexos, o que pode ser visto como um reflexo de uma maior abrangência em relação ao *asis2xml*, estes são facilmente simplificados usando um estilo (*XSL stylesheet*) apropriado.

```
package body Example is
      task body Producer is
         ...
      begin
            ...
            loop
                  ...
            end loop;
      end Producer;
end Example;
```

```
<codeRepresentation>
  <A_PACKAGE_BODY unitName="Example">
  ...
  <A_TASK_BODY_DECLARATION>
        </A_TASK_BODY_DECLARATION>
        <A_DEFINING_NAME>
        <A_DEFINING_IDENTIFIER name="Producer"/>
        ...
        <A_STATEMENT>
              <A_LOOP_STATEMENT/>
        ...
        </A_STATEMENT>
        ...
  </A_PACKAGE_BODY>
</codeRepresentation>
```

Figura 3.4: Excerto de código Ada e respectivo documento XML.

Na figura 3.4 é possível visualizar um excerto de código Ada e respectivo documento XML gerado pelo *Avatox*. As *tags* XML visíveis na figura, são obtidas através da representação que a plataforma *ASIS* gera para o programa em questão, ou seja são um mapeamento directo das estruturas de dados internas do *ASIS* (Fig. 3.3).

3.2 Anotações Temporais

Tal como foi dito anteriormente, grande parte do sucesso do método de verificação proposto deve-se à utilização de anotações no código que auxiliam a tradução, e à construção de modelos que abstraem o comportamento temporal dos programas em questão. É certo que grande parte dessas anotações são influências directas dos trabalhos descritos no capítulo anterior, no entanto durante o desenvolvimento foi necessário repensar algumas dessas anotações, bem como desenvolver outras de modo a tornar possível a conclusão do trabalho. Deste modo, é importante descrever aqui quais as anotações usadas e qual a influência que cada uma tem na obtenção do modelo resultante do esquema de tradução.

Dado que estas anotações são escritas sob a forma de comentários Ada, iniciados pelos caracteres "- -", e sabendo que o SPARK usa o "#" para

identificar as suas anotações (irrelevantes do ponto de vista da compilação, mas relevantes do ponto de vista das ferramentas de análise), adoptou-se para a ferramenta desenvolvida neste trabalho identificar o início de uma anotação com os caracteres "- -@".

- -@deadline arg;

O comando *deadline*, é uma das influências directas da abordagem proposta por Alan Burns. Este permite completar de certa forma a sintaxe do Ada, pois fornece ao programador uma forma de exprimir limites superiores de tempo, como complemento da primitiva *delay until* que permite exprimir limites inferiores de tempo. Assim, é possível de uma forma simples e mais completa especificar o comportamento temporal do programa. No que diz respeito à influência destas anotações no modelo Uppaal, pode-se dizer que são determinantes pois permitem adicionar condições a um relógio sob a forma de um invariante de estado.

Como se pode verificar o comando *deadline* recebe um argumento, que poderá ter uma das formas ilustradas a seguir:

- ***- -@deadline 100;*** - Neste caso estamos a especificar directamente qual o valor que pretendemos impor como limite superior de tempo;
- ***- -@deadline Var;*** - Neste caso estamos a especificar através de uma variável atribuída durante o programa qual o valor que pretendemos impor como limite superior de tempo;
- ***- -@deadline Exp;*** - Neste caso estamos a especificar através de uma expressão qual o valor que pretendemos impor como limite superior de tempo. A expressão poderá conter um conjunto de operações aritméticas e os argumentos dessas operações poderão ser valores inteiros ou variáveis atribuídas ao longo do programa, bem como a mistura dos dois casos;

A figura 3.5 pretende ilustrar a aplicação directa do comando *deadline*. No exemplo usado, podemos reparar que, sempre que pretendemos definir um limite temporal superior para uma determinada instrução, colocamos a anotação imediatamente a seguir. Por outro lado é possível também usar a anotação em conjunto com a instrução *delay until* do Ada, sendo que nesses casos se o parâmetro especificado tiver o mesmo valor, pretendemos dizer que nesse ponto do programa o tempo decorrido terá que ser exactamente esse valor especificado.

```
1.  ...
2.
3.  Start := Clock;
4.
5.  --@deadline Start+Milliseconds(15);
6.  delay until Start+Milliseconds(5);
7.
8.  Start := Clock;
9.
10. loop
11.
12. PO.Data.Read();
13. --@deadline Start+Milliseconds(10);
14.
15. PO.Data.Write();
16. --@deadline Start+Milliseconds(20);
17.
18. ...
```

Figura 3.5: Exemplo de utilização da anotação *deadline*

- -@period arg;

A anotação *period* à semelhança do comando *deadline* é também proposta nos trabalhos de Alan Burns. No caso desta anotação, a sua principal função é especificar o período da tarefa principal do programa, e dado que não tem nenhuma influência no corpo da tarefa, esta deverá ser utilizada no ficheiro de especificação da tarefa (.ads). Através da especificação do período torna-se possível a geração automática de propriedades relacionadas com o cumprimento do mesmo. Este comando recebe um argumento que deverá ser um valor inteiro indicativo do período da tarefa.

- -@guarantee arg1 uses arg2 with_freshness arg3;

A introdução desta anotação, prende-se com o objectivo de se poder exprimir uma relação entre um programa e um objecto protegido que interaja com o sistema. Mais uma vez, temos neste caso uma anotação inspirada nos trabalhos de Alan Burns, mas que no entanto tem uma semântica ligeiramente diferente no contexto desta dissertação. Atentando ao exemplo seguinte, na abordagem de Alan Burns a anotação pretende dizer que o programa em questão comunica com o objecto protegido chamado buffer, através do programa Write e com um limite superior de tempo de 100 ms. A principal diferença entre ambas, reside então na especificação do programa pertencente ao objecto protegido, que se pretende utilizar.

- - -@guarantee buffer uses Write with_freshness 100;

Enquanto que nesta abordagem o nome utilizado remete directamente para a função ou procedimento existente no objecto protegido em questão, no contexto desta dissertação estamos limitados apenas a duas opções, READ e WRITE, conforme se trate de uma operação de escrita ou leitura. Na verdade, a limitação a estas duas operações faz perder expressividade em relação à proposta de Alan Burns, no entanto nada impede de no futuro podermos expandir a sintaxe e possibilitar a utilização de procedimentos/funções existentes no código do objecto protegido. Este facto justifica a utilização da anotação com a mesma forma sintáctica proposta por Alan Burns. Adiantando já pormenores sobre a geração do modelo Uppaal, pode-se dizer que para as tarefas dependentes de objectos protegidos, através dos quais irão ser realizadas escritas e/ou leituras, a ferramenta gera um autómato que abstrai estas operações em vez de construir um autómato baseado no CFG. Isto explica-se pelo facto da implementação da leitura ou escrita não ser relevante para a verificação do comportamento temporal, e portanto poder ser abstraída por um autómato simples. Assim sendo, através desta anotação a ferramenta consegue obter as informações correspondentes ao nome do objecto protegido (arg1), a indicação de leitura ou escrita (arg2) e também a taxa de tempo com que este é acedido (arg3), e desse gerar o autómato abstracto de leitura ou escrita necessário ao sistema. Uma vez que temos a indicação de uma taxa de tempo, é possível ainda com esta anotação gerar propriedades para garantir esses aspectos aquando da verificação do modelo. Por fim resta dizer que tal como a anterior, esta anotação não tem influência directa no CFG da tarefa, e portanto deverá ser igualmente utilizada no ficheiro de especificação (.ads).

Quanto aos argumentos deverão respeitar as seguintes regras:

- ***arg1*** - Este argumento deverá ser apenas o nome do objecto protegido, de outra forma não será reconhecida a anotação;

- ***arg2*** - Este argumento deverá ter um de dois valores possíveis *WRITE* ou *READ* indicativo da operação que se está a usar no objecto protegido. A utilização de um outro valor que não um destes dois torna a anotação inválida;

- ***arg3*** - Este argumento deverá ser apenas um valor inteiro indicativo do período com que o objecto protegido é acedido. Qualquer outro valor que não seja um inteiro levará ao não reconhecimento da anotação;

- -@globalClock arg;

Esta anotação pretende dar resposta a uma necessidade do algoritmo a que nos propusemos desenvolver. A dada altura do desenvolvimento deparou-se com a dificuldade em conseguir determinar qual o relógio principal da tarefa, assim decidiu-se criar esta anotação de forma a tornar possível indicar-se previamente qual o nome da variável correspondente ao relógio global da tarefa. O argumento desta anotação deverá ser então o nome do relógio global, pois de outra forma a anotação não será reconhecida e não será concluído o processo de tradução. Uma vez mais, tal como os dois casos anteriores, dado que não há influência directa desta anotação no CFG, esta deverá ser também utilizada no ficheiro de especificação (.ads).

3.3 Construção do Grafo de Fluxo de Controlo

Nesta secção são apresentados os detalhes da primeira parte do algoritmo implementado, ou seja focando a geração de grafos de fluxo de controlo. O processo tem como input um documento XML, definido como um conjunto de termos organizados numa estrutura arbórea representativa da sintaxe do programa. O algoritmo pode ser visto sumariamente como uma travessia *depth-first* sobre a árvore de sintaxe abstracta presente no documento XML.

Antes de entrar em pormenores é necessário referir que foi imprescindível analisar vários documentos XML por forma a encontrar um padrão comum a todos os programas. Após este estudo reparou-se que todos seguem uma mesma estrutura: uma primeira parte em que são declaradas todas as variáveis e/ou tipos não primitivos, os quais são identificados pela tag *declarations*; e uma segunda parte onde estão presentes todas as instruções do programa representadas pela tag *statements*.

O algoritmo começa por reunir em colecções Java todas as declarações e todas as instruções do documento, usando para isso métodos já existentes na biblioteca XML do Java. De seguida, a função principal de construção do CFG processa dois passos essenciais, um primeiro onde trata todas as declarações, e um segundo onde trata as instruções do programa. Para cada uma das declarações ou instruções, são chamados métodos específicos conforme o tipo de cada uma, usando para isso as respectivas *tags*. Por exemplo para uma declaração de uma variável poderemos ter uma *tag* do género *a_variable_declaration*, invocando-se neste caso um método capaz de tratar este tipo de declaração. Já no caso das instruções poderemos ter por exemplo a *tag a_loop_statement*, sendo neste caso

invocado um método capaz de processar ciclos.

Naturalmente, existem alguns tipos de instruções que possuem sequências de outras instruções dentro do seu corpo, como por exemplo os ciclos ou as instruções condicionais. Nestes casos a estratégia usada é a mesma, i.e. reúnem-se todas as instruções interiores e volta-se a invocar o método para processar cada uma delas. Este processo repete-se recursivamente até que sejam encontradas *tags* que indiquem casos de paragem, que são por norma os identificadores (*an_identifier*) de variáveis, nomes de funções, nomes de procedimentos, e outros casos similares.

Para representar o grafo, foi criada uma estrutura capaz de conter toda a informação necessária não só para a construção do CFG mas também na fase seguinte de geração do modelo Uppaal. Assim sendo, um CFG é representado em memória por uma estrutura Java chamada *TreeMap*, que pode ser vista simplesmente como uma tabela de hash em que a um dado nodo (chave) é associada uma colecção de ligações deste com os seus sucessores. Deste modo, fica representada a estrutura básica de um grafo através da sua lista de adjacências, que com o decorrer do desenvolvimento foi enriquecida com informação necessária tanto a nível dos nodos como a nível das ligações. O aspecto final destes é explicado mais à frente.

Posto isto, a geração do CFG assenta na identificação de três casos base: instruções simples, como por exemplo as atribuições ou os comentários; instruções condicionais; e instruções de ciclo. Para cada uma delas o algoritmo executa uma transformação diferente, as quais são agora apresentadas em mais detalhe.

Instruções Simples

Este é o caso mais imediato encontrado na geração do CFG do programa. Quando se encontra na árvore de sintaxe abstracta uma instrução simples, por exemplo uma atribuição de uma variável, basta criar um novo nodo e adicionar ligações deste com o seu antecessor. Se tivermos uma sucessão de instruções simples, o resultado irá ser obviamente um grafo sequencial, tal como ilustrado na figura 3.6. É importante referir que nesta fase os comentários são também tratados como instruções simples.

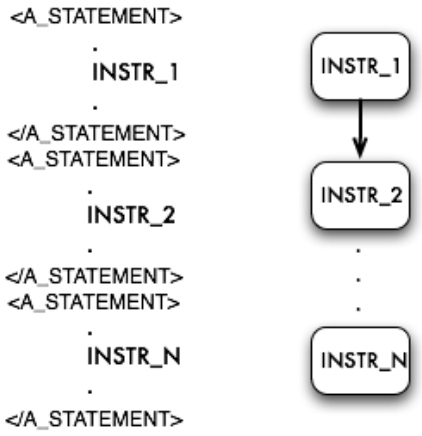

Figura 3.6: Instrução Simples

Instruções Condicionais (If's)

Em termos de fluxo de controlo, uma instrução condicional especifica a possibilidade da execução de um programa tomar um de dois caminhos possíveis, dependendo da avaliação de uma condição booleana. Assim sendo, o grafo de fluxo de controlo terá de exprimir este mesmo facto, i.e. aquando do processamento do documento XML, cada vez que se encontra uma *tag* representativa de uma expressão condicional, são gerados dois novos nodos e adicionadas ligações entre o nodo anterior e estes dois. Esta situação é ilustrada na figura 3.7.

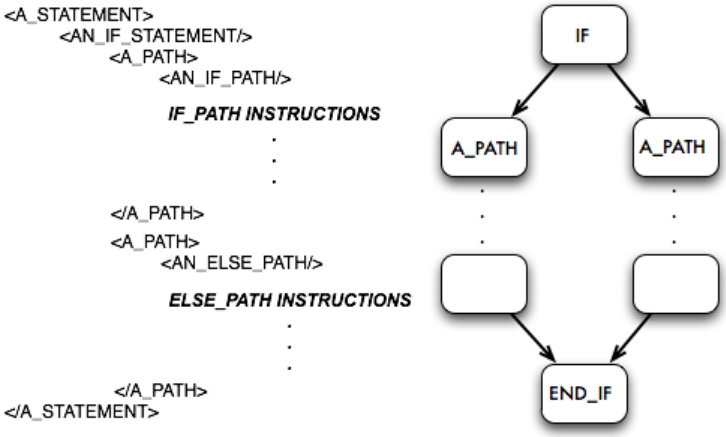

Figura 3.7: Instrução Condicional

Instruções de Ciclo

Na linguagem Ada é possível encontrar declarados dois tipos de ciclos: condicionados e não condicionados. Os ciclos não condicionados são vistos tipicamente como ciclos infinitos, e tendo em conta que os programas de interesse no contexto do presente trabalho, são desenvolvidos segundo o perfil Ravenscar [9], a declaração deste tipo de ciclo surge sempre imediatamente após uma instrução do tipo *"task body statement"*. Abstraindo os nodos que possam surgir com as instruções encapsuladas no ciclo, nestes casos basta adicionar uma ligação a partir do nodo final do ciclo até ao nodo inicial.

Por outro lado no caso dos ciclos condicionados, além da ligação descrita para o caso anterior, é necessário adicionar-se um nodo para a avaliação da expressão booleana que condiciona o ciclo, e uma ligação deste para o nodo final, representando dessa forma os casos em que a execução sai do ciclo. Na figura 3.8 temos a ilustração de ambos os casos, à esquerda para os ciclos não condicionados e à direita para os ciclos condicionados.

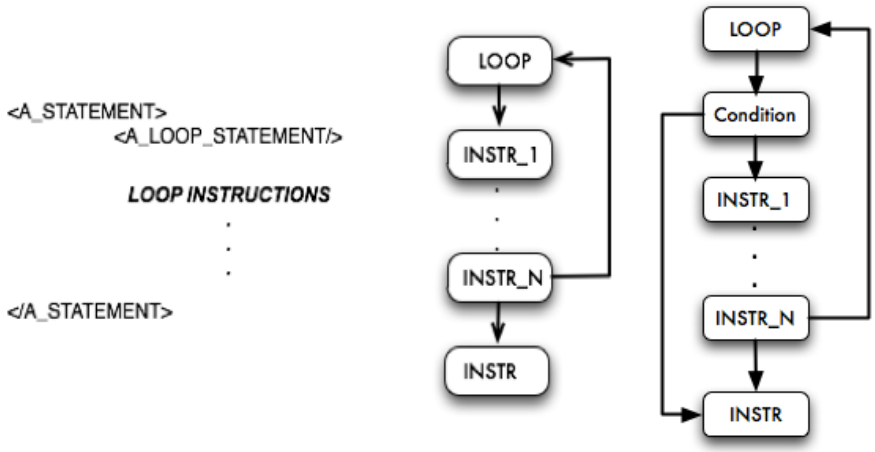

Figura 3.8: Instrução de Ciclo

Após o desenvolvimento do algoritmo básico para a geração de CFG's, foi necessário estudar em pormenor a forma de se obter a partir destes um modelo XTA. Durante esta análise, verificou-se que alguma da informação temporal contida no código estava a ser perdida durante o processo de geração do GFG. Assim sendo, uma vez que essa informação é essencial tendo em conta o objectivo da ferramenta em gerar um modelo XTA, chegou-se à conclusão que a estrutura de dados teria de ser enriquecida de modo a incluir essa informação. Desse modo, procedeu-se ao melhoramento do algoritmo, e usando alguma informação já contida no

grafo (como por exemplo nodos contendo transformações sobre variáveis temporais) bem como outra informação proveniente do XML e que foi descartada na primeira versão do algoritmo, enriqueceu-se a estrutura do objecto CFG com informação relativa a relógios, especificações de tarefas, variáveis classificadas como influentes no modelo final, etc. Deste modo a estrutura final do CFG gerado contém os seguintes elementos:

- **globalClock** - Relógio global da tarefa;
- **clocks** - Lista com todos os relógios utilizados na tarefa;
- **declarations** - Estrutura com toda a informação correspondente à parte das declarações (variáveis, tipos, etc) do programa;
- **graph** - Estrutura que guarda a informação do grafo propriamente dita (nodos e ligações);
- **initialNode** - Nodo inicial;
- **variables** - Lista das variáveis com influência na geração do modelo XTA;
- **taskspec** - Estrutura que guarda informação proveniente da parte da especificação da tarefa, como por exemplo os objectos protegidos usados, o período, etc.

Como consequência deste enriquecimento da estrutura de dados do CFG, e tendo mais uma vez em conta a fase seguinte do processo de geração do modelo XTA, foi necessário rever também as estruturas de dados dos nodos e das ligações do grafo. Sendo assim, à estrutura dos nodos foi acrescentada informação sobre a linha de código que este representa, i.e. criou-se uma classe abstracta representativa de uma instrução que pode ser instanciada em subclasses específicas para cada tipo de instrução (ciclos, atribuições, condicionais, etc), e dessa forma auxiliar o processo.

- **id** - Identificador do nodo;
- **name** - Nome do nodo;
- **father** - Nodo antecessor;
- **instr** - Instrução associada ao nodo.

Já na estrutura das ligações foram acrescentadas etiquetas que permitem indicar que operações estão associadas à transição de estado (alteração de variáveis, sincronizações, guardas, etc)

- **origem** - Nodo de onde parte a ligação;
- **destino** - Nodo para onde transita a ligação;
- **select** - Etiqueta que indica a operação de select;
- **update** - Etiqueta que indica a operação de update;
- **guard** - Etiqueta que indica a guarda da transição;
- **sync** - Etiqueta que indica a operação de sincronização;

Sendo a geração do grafo de fluxo de controlo o meio de se obter o modelo Uppaal pretendido, julgou-se interessante incluir na versão final da ferramenta a exportação do CFG para ficheiro em formato dot. Esta decisão não só ajudou imenso no desenvolvimento do algoritmo, permitindo visualizar graficamente (Fig. 3.9) o grafo (usando a ferramenta graphviz para gerar a figura), como também possibilita o acesso ao grafo de fluxo de controlo de modo a que outro tipo de análises baseadas em CFG's possam ser feitas.

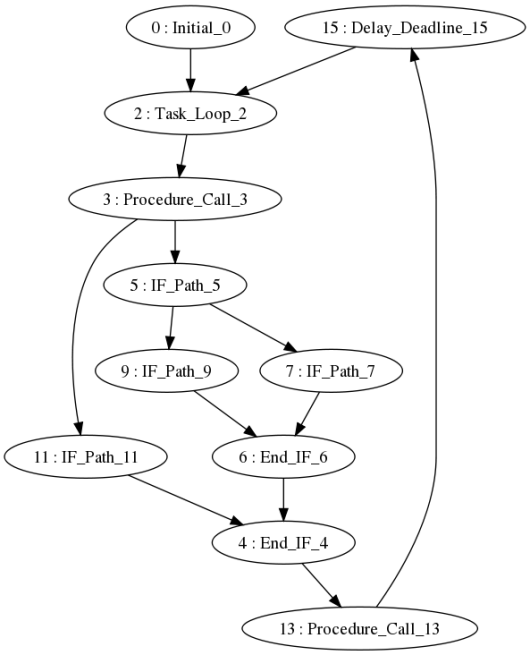

Figura 3.9: Exemplo de um CFG gerado pela ferramenta e obtido através do graphviz

3.4 Construção do modelo XTA

Chegados a esta secção, procedemos à apresentação da segunda parte do algoritmo de geração de modelos XTA. Tal como foi dito anteriormente todo este processo descrito em seguida tem como base a construção prévia de um grafo de fluxo de controlo, o que revela o grau de importância do investimento que se teve de fazer na primeira fase do algoritmo por forma a simplificar esta fase do processo. Assim sendo, de uma forma resumida este processo passa por percorrer todos os nodos e ligações do grafo procedendo-se a sucessivas transformações até chegar a uma estrutura próxima da estrutura de um autómato Uppaal. É certo que apesar de grande parte do esforço estar na geração do CFG, estas transformações não são tão simples como se possa pensar. Para auxiliar este processo foi necessário olhar para diversos exemplos, e aqui uma vez mais foi imprescindível olhar para os trabalhos de Alan Burns, de forma a possibilitar a visualização de toda a transformação necessária e ao mesmo tempo construir um padrão capaz de ser automatizado.

Primeiramente, começou-se por desenvolver uma estrutura de dados capaz de representar toda a informação presente em qualquer sistema de autómatos temporizados em UPPAAL, e que é exportada por este para ficheiros XTA.

- **UppaalState** - Estrutura para representar um estado do autómato Uppaal e constituída por:
  - Nome
  - Invariante
- **UppaalTransition** - Estrutura para representar uma transição do autómato Uppaal e constituída por :
  - UppaalState origem
  - UppaalState destino
  - Guarda da transição
  - Select
  - Actualização de variáveis
  - Sincronizações
- **UppaalAutomaton** - Estrutura para representar um autómato Uppaal e constituída por:
  - Conjunto de argumentos/parâmetros

  - Conjunto de variáveis
  - Conjunto de estados
  - Conjunto de transições
  - Lista de estados urgentes
  - Estado inicial
  - Conjunto de declarações
  - Conjunto de relógios

- **UppaalModel** - Estrutura para representar um modelo Uppaal e constituída por:
  - Conjunto de autómatos
  - Conjunto de declarações
  - Conjunto de propriedades

Observe-se que esta estrutura é extremamente importante uma vez que aspectos tais como tratamento de nomes (identificadores de estados, transições, etc), tratamento de objectos protegidos relacionados com a tarefa, assim como a inferência de algumas propriedades imediatas são todos feitos com base nesta estrutura de dados final.

Antes de focarmos os detalhes da função de transformação do CFG acima referida, devemos começar por dizer que tal só é executada com sucesso graças a uma classificação prévia dos nodos do CFG, e que é feita pela função de geração do grafo. Podemos constatar aqui uma vez mais o grau de importância da primeira fase do algoritmo. Assim sendo, graças a esta classificação foi possível criar uma função de iteração sobre o conjunto de nodos do CFG, executando uma determinada transformação conforme o tipo do nodo em questão, os quais são enumerados de seguida:

1. Nodo Inicial (*Initial*)
2. Atribuição (*Assignment*)
3. Instrução If (*If_Path*)
4. Instrução Else (*Else_Path*)
5. Fim de uma instrução condicional (*End_If*)
6. Início do ciclo de uma tarefa (*Task_Loop*)
7. Início de um ciclo For (*For_Loop*)

8. Fim de um ciclo (*End_Loop*)
9. Chamada de procedimento (*Procedure_Call*)
10. Instrução de delay (*Delay_Until*)
11. Comentário (*Comment*)

Posto isto, procedemos agora à explicação detalhada da função que executa as transformações necessárias conforme o tipo de cada nodo iterado.

Initial No nodo inicial do grafo de fluxo de controlo, uma vez que simboliza o arranque da execução do programa, não são necessárias grandes transformações, sendo este apenas marcado como *urgent*.

Assignment Antes de se explicarem as transformações feitas sobre nodos do tipo atribuição, é necessário referir que é executada em primeiro lugar uma função que elimina todos os nodos irrelevantes para o modelo, i.e nodos que apresentem atribuições a variáveis que nem sejam temporais ou então que os ciclos não tenham influência sobre estas, são simplesmente apagadas do CFG. Posto isto, cada vez que se encontra um nodo deste tipo, se a variável for do tipo *Time* verifica-se que atribuição é feita, elimina-se o nodo, e adiciona-se uma ligação entre o nodo anterior e o posterior onde está definida uma operação de actualização (*update*) com a atribuição em causa. Por outro lado, se a variável tiver influência sobre o progresso de algum ciclo do programa (tipicamente se esta for interior ao ciclo) deixa-se simplesmente a indicação para esta ser posteriormente tratada no nodo do tipo *For_Loop*.

If_Path & Else_Path Quando se encontram nodos representativos de instruções condicionais, é necessário verificar em primeiro lugar se estes pertencem ao corpo de algum ciclo *for*. Isto deve-se ao facto de ser necessário testar se a variável envolvida na condição da instrução condicional, está a ser afectada pela condição do ciclo. Nos casos em que isso acontece, a instrução condicional terá de ser reflectida da mesma forma no modelo Uppaal, ou seja terá de existir a representação da variável bem como todas as operações feitas sobre esta. Pensando em termos de verificação do modelo, esta estratégia faz mais sentido uma vez que para o *model checker* termos um nodo a representar uma instrução condicional mas que no entanto não tem nenhuma variável a condicioná-lo, é irrelevante pois este irá explorar ambos os casos de execução. Por outro

lado se tivermos guardas nas duas transições do *If_Path* para os nodos seguintes, estamos a dizer explicitamente ao *model checker* que queremos que as transições sejam executadas para determinados valores da variável condicionante. Assim sendo em termos de transformações ao CFG, no caso em que é importante ter a variável representada além de termos de incluir a variável no conjunto das variáveis do sistema, teremos de criar guardas nas transições nos dois caminhos (transições) de execução que o *if* pretende simular. No outro caso procede-se simplesmente à etiquetagem do nodo como *urgent*. Por fim quanto aos nodos classificados como *Else_Path*, estes são transformados como se de um *If_Path* se tratasse e seguem portanto a linha de ideias explicadas anteriormente.

End_If Um nodo classificado como *End_If*, à semelhança de outros nodos como por exemplo o nodo inicial é também ele um nodo simples, e neste caso pretende apenas representar o fim do corpo de uma instrução condicional. Portanto, uma vez mais a transformação necessária é simplesmente a inclusão da etiqueta *urgent*.

Task_Loop O nodo *Task_Loop* é também ele um nodo simples. Existe apenas um nodo classificado desta forma em todo o CFG, sendo que este simboliza o arranque de uma tarefa. Mais uma vez a transformação efectuada é somente a inclusão da etiqueta *urgent*.

For_Loop & End_Loop Para o caso dos nodos classificados como *For_Loop*, ou seja representativos de ciclos *for*, a estratégia usada não é tão simples. Nesta situação é necessário efectuar transformações nas ligações tanto do nodo inicial como do nodo final do ciclo, o que torna o processo mais complexo e mais susceptível a falha sendo então necessário um maior cuidado na elaboração da lógica de transformação. Assim sendo, em primeiro lugar é necessário colocar uma guarda na ligação que faz a transição entre o nodo inicial do ciclo e o nodo final para que esta seja somente utilizada quando a condição do ciclo não for verificada. Em adição, é necessário ainda reiniciar a variável incrementada pelo ciclo para que não afecte a próxima iteração da tarefa. À semelhança do caso anterior, na ligação que estabelece a transição entre o nodo inicial e o nodo representativo da primeira instrução do ciclo é colocada uma guarda para que seja igualmente testada a condição de acesso ao corpo do ciclo. Em relação à ligação do nodo final do ciclo que faz a transição entre este e o nodo inicial (repetição da iteração do ciclo) é necessário colocar uma operação de actualização para que a variável de incrementação do ciclo seja alterada. As restantes ligações do nodo permanecem

inalteradas.

Na verdade, não seria necessário manter o nodo classificado como *End_Loop* uma vez que este serve apenas para marcar o final do ciclo, e portanto seria facilmente substituído pelo nodo da instrução imediatamente a seguir. No entanto durante o desenvolvimento reparou-se que o facto de este existir no autómato final ajuda na leitura e interpretação do modelo, por isso decidiu-se que seria mantido no modelo final mas com uma pequena modificação, a inclusão da etiqueta *urgent*.

Procedure_Call Os nodos representativos de invocações de procedimentos são também alvo de transformação. Em primeiro lugar é necessário testar se o procedimento em causa está associado a um objecto protegido, recorrendo-se para isso à especificação da tarefa. Relembramos aqui que algumas das anotações propostas são utilizadas na parte da especificação dos programas Ada para indicar períodos, dependências com objectos protegidos, etc. Deste modo, caso a chamada do procedimento esteja associada a um objecto protegido e caso seja uma operação de escrita no mesmo, actualiza-se a transição colocando o envio (**sync!**) de um sinal de sincronização para o autómato que modela o objecto protegido. Se por outro lado corresponder a uma operação de leitura, a transição terá de aguardar (**sync?**) que lhe seja enviado um sinal de sincronização do autómato correspondente ao objecto protegido.

Delay_Until Tal como explicado anteriormente, uma instrução de *delay_until* indica que o programa deverá esperar uma dada quantidade de tempo antes de prosseguir com a execução. Ora, quando se encontra um nodo representativo desta instrução, o algoritmo irá naturalmente proceder às transformações necessárias para se representar correctamente este conceito no modelo XTA. Para se proceder às transformações é necessário em primeiro lugar ter em conta que a instrução poderá assumir várias formas conforme os parâmetros que lhe são passados, i.e poderemos ter um valor constante como parâmetro, uma variável, ou até uma expressão aritmética envolvendo constantes e/ou variáveis. Portanto, o primeiro passo será determinar o valor temporal passado como parâmetro à instrução, se este for um valor constante a obtenção é imediata, se for uma variável ou uma expressão é necessário procurar os respectivos valores e proceder ao cálculo. Em seguida, uma vez que este tipo de instrução impõe um limite inferior de tempo resta reflectir isso no modelo XTA, e para isso são colocadas guardas em todas as transições que saem deste tipo de nodo para os seus sucessores, garantindo dessa forma que a execução nunca prossiga sem que o tempo de espera imposto seja atingido.

Comment Um nodo classificado com o tipo *comment*, terá transformações relacionadas com a semântica do comando *deadline*. A utilização desta anotação em dado ponto de um programa Ada, indica que o tempo de execução do programa até esse ponto não deve exceder o valor especificado pela anotação, isto é impõe um limite temporal superior. Como se sabe o nosso modelo final pretende reflectir vários aspectos do programa, e em especial os aspectos temporais. Por isso mesmo foi necessário primeiro investigar a melhor forma de se poder utilizar este conceito no modelo XTA, chegando-se à conclusão que é possível representar a imposição temporal colocando um invariante sobre a variável temporal, no nodo correspondente. Assim, cada vez que o *model checker* passa por esse nodo e a condição não seja verificada, estamos perante um caso em que o nosso programa não se irá comportar devidamente em relação aos aspectos temporais especificados.

Considerações Explicados todos os passos tomados de modo a transformar o CFG num modelo Uppaal, é necessário referir alguns aspectos importantes. Em primeiro lugar, devemos salientar que apesar destes detalhes explicados acima levarem à geração de um modelo perfeitamente interpretável pelo Uppaal, existem algumas optimizações que se podem fazer de modo a tornar o modelo mais simples e consequentemente reduzir a possibilidade de ocorrer explosão de estados durante a execução do *model checker*.

Em alguns casos como por exemplo os nodos classificados de *End_Loop* e *End_If*, tal como foi referido anteriormente, mesmo sendo possível retirar estes nodos do modelo decidiu-se que seriam mantidos em prol da melhor compreensão do mesmo. No entanto, quanto aos nodos do tipo *Delay_Until* e *Comment* chegou-se à conclusão que em determinados casos seria possível efectuar uma simplificação importante. Existem determinadas situações em que estes dois tipos de nodos podem surgir seguidos, nesses casos se o nodo do tipo *Comment* corresponder a um comando *deadline*, podemos testar se as variáveis e valores parametrizados tanto no *deadline* como no *delay until* são iguais, procedendo-se nesse caso à aglutinação dos dois nodos. Quando esta situação ocorre, elimina-se um dos nodos e copia-se toda a informação de um para o outro tanto a nível do nodo (invariantes) como a nível das transições (guardas, sincronizações etc). Aos nodos aglutinados atribuiu-se o tipo *Delay_Deadline*.

Apesar do algoritmo descrito acima conter as operações mais importantes desta fase, existem ainda alguns passos igualmente importantes a serem dados por forma a obtermos o modelo Uppaal final. Assim sendo, após todo o processo de transformação sobre os CFG's das tarefas pre-

sentes no sistema, é necessário executar um conjunto de operações de modo a colocar a informação (declarações e propriedades) de cada autómato nas estruturas de dados do modelo (**UppaalModel**). Quanto às declarações é necessário também verificar se existem dependências dos autómatos com objectos protegidos, e nesse caso para cada objecto protegido dependente gerar um autómato *default* que o modele. Tal como foi explicado anteriormente, a informação sobre a dependência de objectos protegidos está anotada no ficheiro de especificação do programa Ada (.ads), portanto a estratégia usada foi criar uma função que conforme a anotação encontrada consiga gerar um autómato que modele as operações de escrita e/ou leitura de objectos protegidos. Em relação às propriedades, desenvolveu-se um algoritmo que gera algumas propriedades imediatas, e o qual se explica na secção seguinte 3.5.

A ferramenta de tradução De forma a tornar mais apelativa a utilização da ferramenta decidiu-se criar uma interface gráfica (ver figura 3.10), que permite utilizar todas as funcionalidades da ferramenta de uma forma mais simples. Nesta interface é possível de forma rápida e fácil seleccionar a partir do sistema de ficheiros o programa Ada que se pretende traduzir, e com a simplicidade de carregar num botão gerarmos os ficheiros *.dot* e/ou *.xta* correspondentes ao programa Ada seleccionado.

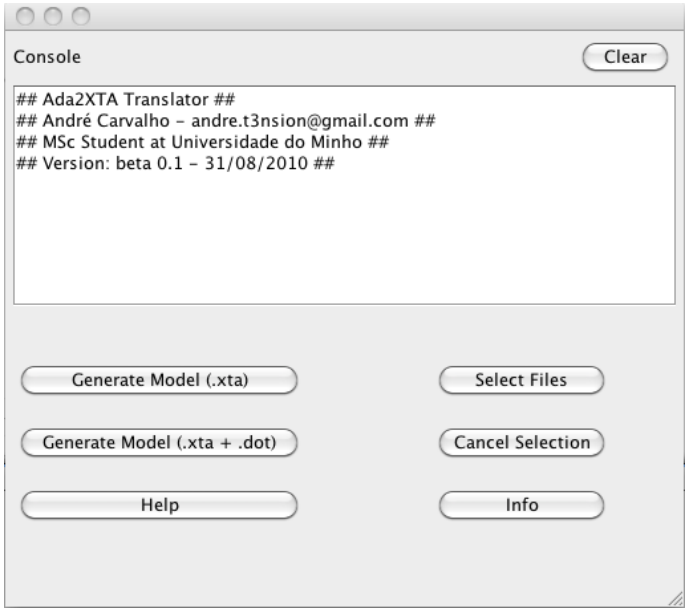

Figura 3.10: Interface gráfica da ferramenta desenvolvida

Apesar de tudo, a equipa está perfeitamente consciente que existe bastante trabalho a ser feito na busca de uma alternativa sólida que resulte

numa ferramenta robusta. O projecto futuro do trabalho a realizar é explicado em mais pormenor na conclusão 5.

3.5 Inferência de propriedades

Com o intuito de complementar e melhorar o processo desenvolvido e detalhado nas secções anteriores, a equipa achou que seria importante chegar-se a um mecanismo automático de inferência de propriedades, capazes de representar as características temporais esperadas para o modelo gerado.

Deste modo, e mais uma vez tendo em conta as anotações presentes no código, foi possível desenvolver um algoritmo capaz de gerar algumas propriedades simples, mas importantes para a verificação de características básicas do sistema.

Assim, em primeiro lugar, pelo facto de não ser necessária a leitura de qualquer variável do sistema, a propriedade mais imediata de se obter, é a que testa se o sistema entra em *deadlock*, e que em lógica temporal do Uppaal exprime-se como (A[] not deadlock).

Em seguida, foi necessário estudar o impacto de algumas variáveis temporais de modo a se perceber que tipo de propriedades seriam possíveis extrair através destas. Mais uma vez percebeu-se que de facto existem algumas propriedades que são de fácil obtenção.
Sendo assim, graças às variáveis temporais representadas no modelo, é possível gerar uma propriedade que testa se o período da tarefa é cumprido. Para isso recorre-se às anotações presentes na especificação dos programas Ada (.ads), onde deverá ter indicado o valor do período, e procede-se à geração da propriedade que testa se o relógio global do sistema nunca ultrapassa o valor especificado para o período da tarefa, exprimindo-se em Uppaal da seguinte forma:

- A[] Tarefa.RelogioGlobal <= período

Finalmente, tentou-se também perceber qual a influência dos objectos protegidos no comportamento temporal do sistema. Recorrendo novamente às anotações usadas na especificação das tarefas, nomeadamente através da anotação @*guarantee*, foi possível obter os intervalos de tempo especificados para as leituras e escritas nos objectos protegidos, gerando assim propriedades que verificam a manutenção dos tempos a serem cumpridos. Estes tipo de propriedade exprime-se em notação Uppaal da seguinte forma:

- A[] ObjectoProtegido.Relogio <= Intervalo

Conclusão. Os trabalhos desenvolvidos neste campo da inferência de propriedades, levou-nos a perceber que apesar de se ter conseguido desenvolver um algoritmo capaz de gerar algumas propriedades importantes para a verificação de aspectos temporais de programas de tempo-real, existem muitas outras que poderão aumentar a potencialidade da ferramenta. No entanto, será necessária uma investigação mais aprofundada deste tema de modo a se perceber que tipo de alterações a ferramenta estará sujeita. Sendo este um assunto com um grau de complexidade acrescido, será necessário um período de tempo mais alargado para se proceder a esse estudo. Além disso, de forma a reforçar a confiança no método de verificação proposto e principalmente na ferramenta, será necessário proceder à validação rigorosa (em última análise formal) dos algoritmos adjacentes. No entanto, sendo este um processo igualmente complexo, ambos os desenvolvimentos explicados serão considerados como de trabalho futuro (secção 5.1).

Capítulo 4

Validação Experimental

Neste capítulo iremos apresentar alguns casos experimentais para demonstrar todo o ciclo de utilização da ferramenta, assim como os pontos onde o seu uso poderá ser importante na detecção de eventuais problemas temporais. Assim sendo, cada um dos exemplos irá ter um foco diferente, tentando demonstrar as diferentes vertentes na utilização da ferramenta desenvolvida.

4.1 Produtor - Consumidor

Este primeiro caso de estudo que se apresenta é especial pelo facto de ter acompanhado todo o desenvolvimento da ferramenta, i.e sendo este o caso de estudo apresentado no artigo que serviu de base a esta dissertação [10], decidiu-se tomar como meta inicial a geração de um modelo semelhante ao modelo apresentado no artigo. Além disso, o conjunto de construções usadas no código do programa em questão é suficientemente alargado, de forma que conseguindo-se traduzir esse programa seria possível traduzir uma quantidade considerável de diferentes programas.

O código apresentado em 4.1 representa um sistema do tipo produtor-consumidor onde existe uma tarefa cíclica que a cada iteração faz uma leitura a partir de um sensor entre dois possíveis, escrevendo o valor lido num objecto protegido. Os dois sensores diferem na forma como o valor pode ser obtido, enquanto que num deles (*simple sensor*) para se obter um valor credível é necessário efectuar 10 leituras consecutivas, no outro (*smart sensor*) bastará apenas uma. Além disso, existem ainda alguns requisitos temporais a ser cumpridos.

- Deverá existir um novo valor no objecto protegido a cada 300 milisegundos.

- O sensor *smart*, após a sua activação, deverá permitir uma nova leitura a cada 30 milisegundos.

- O sensor *simple* requer que cada leitura seja feita no intervalo de tempo entre os 2 e os 4 milisegundos após a sua activação. Além disso cada conjunto de 10 leituras consecutivas deverá ser feito somente a cada 10 milisegundos.

Para simplificar a implementação do sistema, assume-se como não determinística a escolha entre os dois sensores.

Example.adb

```
1.  package body Example is
2.    task body Producer is
3.      type Device_Type is (smart,simple);
4.      Device : device_Type;
5.      Next : Time;
6.      Current : Time;
7.      Data : Integer;
8.
9.      begin
10.       Next := Clock;
11.       Data := 0;
12.       loop
13.       Device := smart;
14.       if Device = smart then
15.         Current := Clock;
16.         delay until Current+Milliseconds(30);
17.         Input_Data.Data.Write(Data);
18.         null;
19.       else
20.         for Count in 1..10 loop
21.         Current := Clock;
22.         delay until Current+Milliseconds(2);
23.         --@deadline Current+Milliseconds(4);
24.         if Count =10 then
25.           Input_Data.Data.Write(Data);
26.         else
27.           Current := Clock;
28.           delay until Current+Milliseconds(15);
29.           --@deadline Current+Milliseconds(15);
30.           null;
31.         end if;
32.       end loop;
33.     end if;
34.     Next := Next + Milliseconds(150);
35.     delay until Next;
36.     --@deadline Next+Milliseconds(150);
37.     null;
38.   end loop;
39.  end Producer;
40. end Example;
```

Example.ads

```
1.  with Ada.Real_Time;
2.  use Ada.Real_Time;
3.  with Input_Data;
4.  use Input_Data;
5.
6.  package Example
7.
8.  is
9.  task Producer;
10.       --@globalClock Next;
11. --@period 150;
12. --@guarantee Input_Data uses Write
        with_freshness 300;
13.
14. end Example;
```

Figura 4.1: Produtor Consumidor (Ada)

O modelo Uppaal correspondente à implementação descrita acima e gerado pela ferramenta desenvolvida encontra-se apresentado na figura 4.2. A comparação do código e do modelo apresentados, permite-nos concretizar os conceitos explicados no capítulo anterior relativos ao algoritmo de tradução. Podemos constatar que toda a semântica do programa

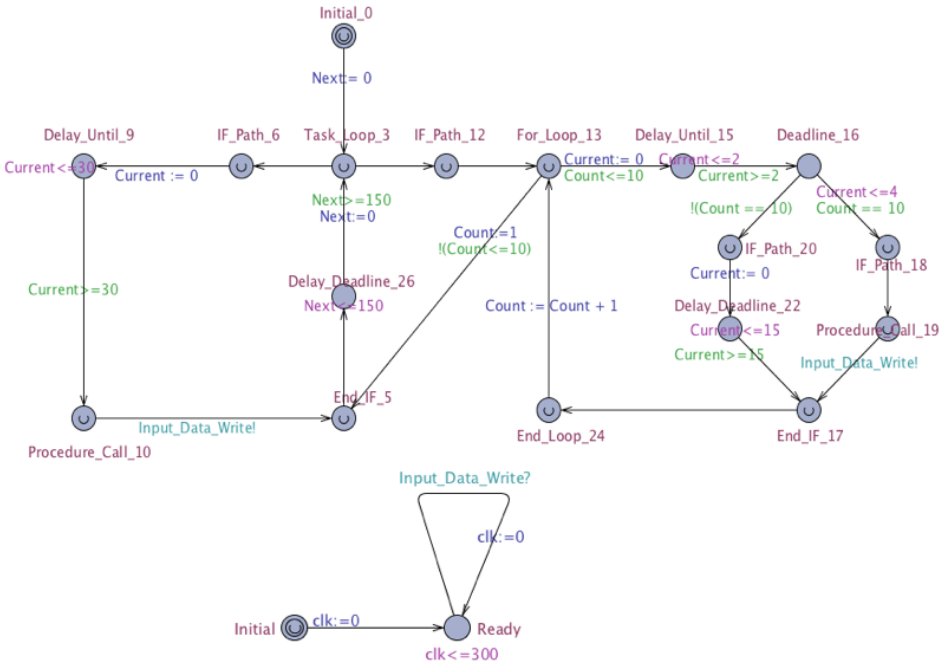

Figura 4.2: Example Producer versão incorrecta.

se encontra representada (através do seu grafo de fluxo de controlo), perdendo-se apenas variáveis e condições irrelevantes temporalmente. Podemos ainda verificar a existência de optimizações como no caso do nodo *Delay_Deadline_20* em que se aglutinou um *delay_until* e um *deadline*.

Devemos ainda destacar as propriedades automaticamente geradas:

- A[] $Instanceof\_Example\_Producer.Next <= 200$
- A[] $Instanceof\_Input\_Data.clk <= 300$
- A[] *not deadlock*

Estas propriedades são inferidas a partir das anotações colocadas no ficheiro *.ads* associado ao código Ada correspondente ao modelo que se apresenta neste caso de estudo. Apesar de serem apenas três propriedades, reflectem requisitos temporais essenciais. No entanto, tal como foi dito no capítulo anterior, o *Uppaal* oferece-nos a possibilidade de especificar outras propriedades que achemos igualmente pertinentes de verificar.

Em termos de verificação, recorrendo ao *model checker* do Uppaal verifica-se que a propriedade de *deadlock* não é mantida, pois o modelo permite

que se alcançe o estado *End_IF_5* com o valor do relógio *Next* maior do que 150 milisegundos, ou seja uma vez que o estado seguinte tem um invariante que diz que o valor do relógio *Next* nunca pode ser maior do que 150 milisegundos, a execução fica impedida de avançar colocando o modelo em *deadlock*.

De modo a que este problema seja solucionado, é necessário aumentar o período da tarefa para 200 milisegundos. No entanto, a alteração que soluciona o primeiro problema levanta um outro relacionado com o caminho de execução oposto (correspondente ao sensor *smart*), sendo agora necessário rever as anotações temporais presentes nesse caminho. Através da ferramenta de simulação do Uppaal, assim como a opção do *model checker* de *diagnostic trace*, que possibilita a apresentação do caso de erro encontrado, torna-se possível encontrar as restrições temporais adicionais a ser aplicadas ao caminho de execução em questão. Assim sendo, deveremos adicionar ao modelo um nodo de *delay_deadline* (restrição temporal superior e inferior) imediatamente antes do nodo *Delay_Until_9* com o valor especificado igual a 70 milisegundos, e um nodo de *deadline* imediatamente após o nodo *Procedure_Call_10* com um valor de 130 milisegundos, de modo a que os 200 milisegundos que o caminho deve demorar a executar sejam cumpridos.

Após todas as alterações acima descritas e necessárias à correcção temporal do sistema, o modelo resultante é apresentado na figura 4.3

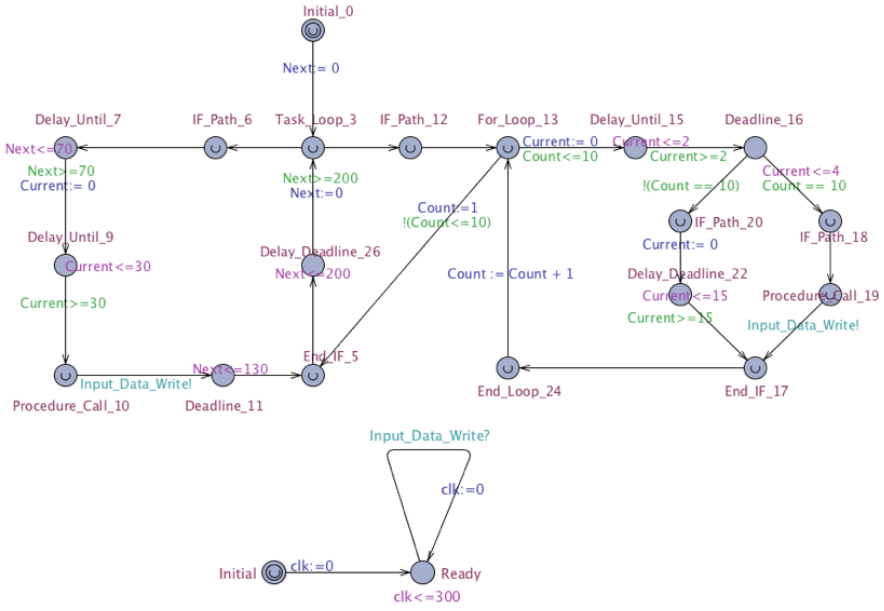

Figura 4.3: Example Producer versão corrigida.

4.2 Auto Pilot

O exemplo descrito em seguida pretende descrever de forma simples um sistema de controlo de altitude e inclinação de uma aeronave. Neste sistema, conforme os valores lidos num painel de instrumentos procede-se à actualização da potência (*throttle*) aplicada e do grau de abertura dos *flaps* (ver figura 4.4). Como era de esperar, existem igualmente algumas restrições temporais para que o sistema tenha um comportamento correcto:

1. A actualização dos controlos deve ocorrer a cada 20 milisegundos;
2. A leitura do painel de instrumentos deve ser feita nos primeiros 5 milisegundos;
3. A actualização da potência e dos *flaps* deve ser feita nos 3 millisegundos seguintes à leitura dos valores do painel de instrumentos.

Autopilot.adb

```
1.  package body Autopilot is
2.
3.  task body Control_Task is
4.
5.  Start : Time;
6.  Reading : Integer;
7.
8.
9.  begin
10. Start := Clock;
11.
12. --Initialization
13.
14. --@deadline Start+Milliseconds(15);
15. delay until Start+Milliseconds(5);
16.
17. Start := Clock;
18.
19. loop
20.
21. Instruments.Data.Read(Reading);
22. --@deadline Start+Milliseconds(5);
23.
24. Flap.Data.Write(Reading+1);
25. Throttle.Data.Write(Reading+2);
26. --@deadline Start+Milliseconds(8);
27.
28. Start := Start+Milliseconds(20);
29. delay until Start;
30. null;
31. end loop;
32. end Control_Task;
33. end Autopilot;
```

Autopilot.ads

```
1.  with Ada.Real_Time;
2.  use Ada.Real_Time;
3.  with Instruments;
4.  use Instruments;
5.  with Flap;
6.  use Flap;
7.  with Throttle;
8.  use Throttle;
9.
10. package Autopilot
11.
12. is
13. task Control_Task;
14. --@globalClock Start;
15. --@period 20;
16. --@deadline 8;
17. --@guarantee Instruments uses Read with_freshness 20;
18. --@guarantee Flap uses Write with_freshness 20;
19. --@guarantee Throttle uses Write with_freshness 20;
20.
21. end Autopilot;
```

Figura 4.4: Auto Pilot (Ada)

O modelo Uppaal que descreve o sistema acima proposto, encontra-se apresentado na figura 4.5. Como é possível analisar pela imagem, mais

uma vez se observa que as variáveis irrelevantes do ponto de vista temporal foram descartadas. Por outro lado, podemos constatar que todas as instruções temporais (*delay_until* e *deadline*) e respectivos valores especificados, se encontram representados no autómato gerado, sob diferentes formas (estados, transições, guardas e invariantes). O facto de estas instruções temporais estarem representadas no modelo, permite-nos assim ter alguma confiança no modelo temporal gerado a partir do código previamente anotado.

Devemos ainda realçar os três autómatos auxiliares, também eles presentes na figura 4.5. Cada um desses autómatos representa uma operação (de escrita ou leitura) diferente, correspondente a cada um dos três procedimentos presentes no código Ada, *Instruments.Data.Read()*, *Flap.Data.Write()* e *Throttle.Data.Write()*. Uma vez que as operações modeladas pelos três autómatos têm influência temporal no sistema, torna-se crucial que estes sejam incluídos no modelo, pois permitem a especificação de certas propriedades importantes.

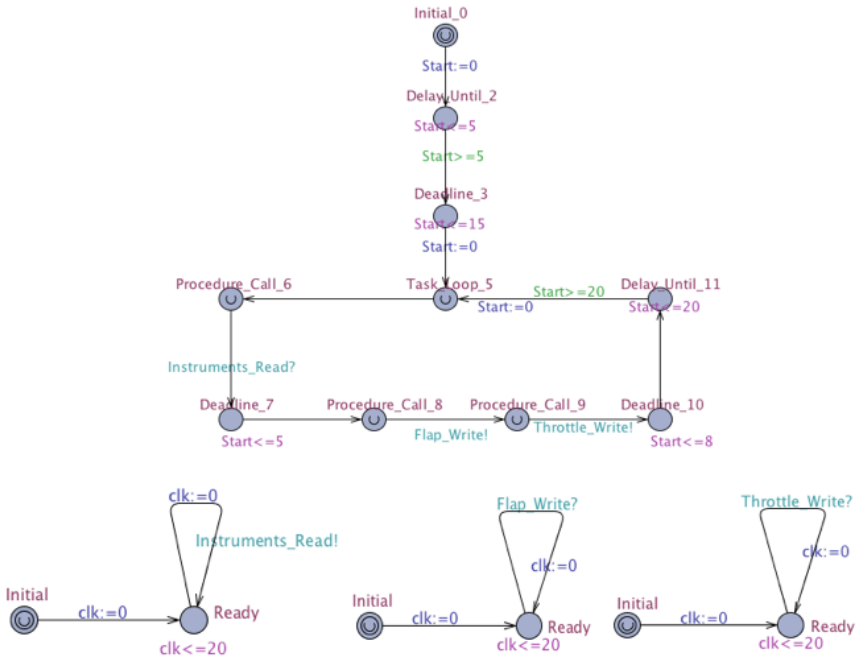

Figura 4.5: Auto Pilot (Modelo Uppaal)

Antes de entrarmos no contexto da verificação, apresentamos em primeiro lugar as propriedades geradas automaticamente pela ferramenta:

- $A[]\ Instanceof\_Autopilot\_Control\_Task.Start <= 20$

- $A[]\ Instanceof\_Instruments.clk <= 20$
- $A[]\ Instanceof\_Flap.clk <= 20$
- $A[]\ Instanceof\_Throttle.clk <= 20$
- $A[]\ not\ deadlock$

Recorrendo ao *model checker* do Uppaal, obtemos sucesso na verificação de todas as propriedades geradas automaticamente, apresentadas na figura 4.6.

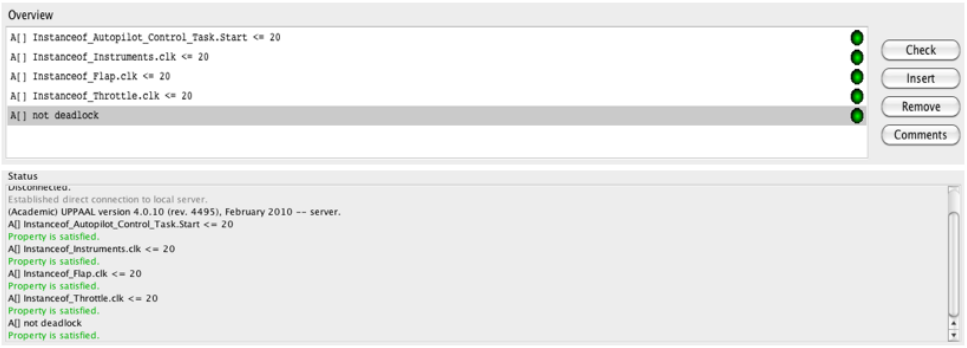

Figura 4.6: Verificação das propriedades geradas automaticamente

Contudo, apenas com as propriedades geradas não é possível garantir todos os requisitos temporais especificados para o problema, nomeadamente os requisitos 2 e 3. Assim sendo, de forma a verificar que estes requisitos também são mantidos, devemos adicionar novas propriedades ao modelo para que sejam também elas verificadas pelo *model checker* do Uppaal. As propriedades em lógica temporal, que descrevem os requisitos referidos são:

- $A[]\ (Instanceof\_Autopilot\_Control\_Task.Procedure\_Call\_6)\ imply\ (Instanceof\_Autopilot\_Control\_Task.Start <= 5)$
- $A[]\ (Instanceof\_Autopilot\_Control\_Task.Procedure\_Call\_8)\ imply\ (Instanceof\_Autopilot\_Control\_Task.Start <= 5 + 3)$
- $A[]\ (Instanceof\_Autopilot\_Control\_Task.Procedure\_Call\_9)\ imply\ (Instanceof\_Autopilot\_Control\_Task.Start <= 5 + 3)$

Mais uma vez, após a utilização do *model checker* a verificação é bem sucedida para todas as propriedades, garantindo assim a manutenção de

todos os requisitos temporais inerentes ao sistema. Apesar disso, é necessário ter atenção a alguns aspectos que poderão trazer problemas. Neste campo é extremamente importante analisar em pormenor o código dos objectos protegidos presentes no sistema, pois apesar das propriedades existentes garantirem a correcta sincronização temporal entre estes, isto não quer dizer que o código mantenha o comportamento correcto. Uma das soluções passa por traduzir também o código de todos os objectos protegidos presentes no sistema, à semelhança do que acontece com o sistema principal, no entanto isso acarreta outras complicações e por isso mesmo identificamos esta solução como um item para trabalho futuro. A outra solução será verificar se existem instruções temporais no código que permitam manter as propriedades especificadas para o sistema. Tomando como exemplo o painel de instrumentos referido, se alterarmos o invariante do estado *Ready* para um valor inferior a 20 ms, rapidamente verificamos que o modelo entra no estado de *deadlock*. Analisando o código do objecto protegido, reparamos que a situação descrita poderá ocorrer, pois não existe nenhuma instrução temporal que garanta que o objecto protegido execute nos 20 ms desejados. A solução para este problema será adicionar uma instrução de *delay until* no código do objecto protegido.

Apesar de ainda ser uma limitação a impossibilidade de se gerarem autómatos para o código dos objectos protegidos, a identificação deste problema demonstra mais uma vez a importância da ferramenta desenvolvida, pois permitiu corrigir uma situação errónea temporalmente.

4.3 Mine Pump

O sistema usado neste caso de estudo é bastante conhecido no mundo do tempo-real. A arquitectura deste encontra-se detalhada na figura 4.7. O objectivo geral do sistema é controlar uma bomba com base no nível de água de uma mina. Para isso existem dois sensores que indicam dois níveis críticos de água, um baixo e um alto. Quando o nível está muito alto, a bomba é comandada para bombear a água para fora da mina, quando o nível é baixo a bomba é desligada. No entanto, existem ainda três sensores adicionais que monitorizam o fluxo de ar e os níveis de dióxido de carbono e de metano. Pelo facto de existir risco de explosão quando o nível de metano é alto, a bomba deve ser imediatamente desligada. Quanto aos níveis de gases, deverá soar um alarme quando é detectado que estes se encontram elevados.

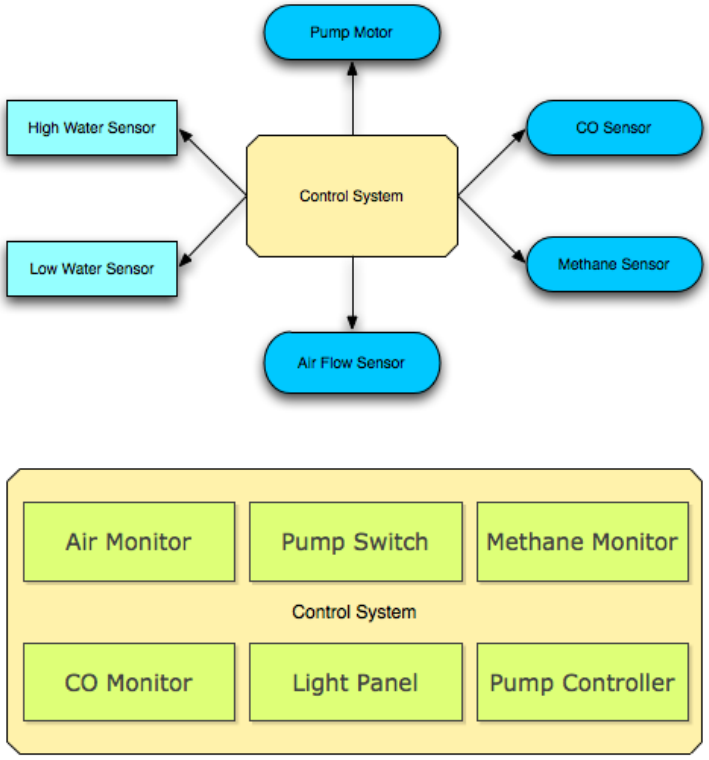

Figura 4.7: Arquitectura do sistema Mine Pump

Dada a sua dimensão, este sistema é composto por vários módulos Ada cada um com o respectivo código que implementa os diversos subsistemas. Recorrendo à ferramenta é possível gerar modelos para cada um desses subsistemas e verificar as respectivas propriedades. Nas figuras 4.8 e 4.9, temos dois exemplos gerados pela ferramenta, sobre os quais

é possível raciocinar de forma independente. Esta estratégia permite-nos abstrair determinadas características irrelevantes para o dado subsistema, e assim conseguir uma maior compreensão do código. Aliás, esta técnica é frequentemente usada em publicações ligadas à verificação, e onde encontramos a título de exemplo o artigo de Kristina Lundqvist [36].

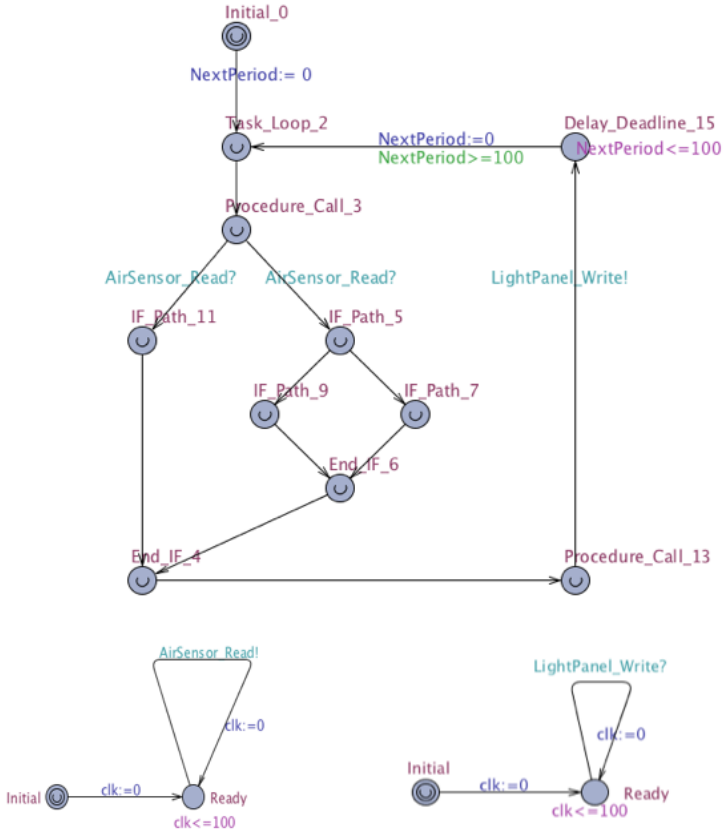

Figura 4.8: Air Monitor

Considerando a figura 4.7 onde se encontra ilustrada a arquitectura do sistema *Mine Pump*, devemos destacar o módulo *Pump Switch* dado o seu grau de importância no sistema. Uma vez que é neste módulo que se comanda a activação e desactivação da bomba de água, assim como a interacção entre os restantes subsistemas como o *Methane Monitor* ou o *Air Monitor*, este é considerado o módulo principal. Assim para além da possibilidade de analisarmos cada módulo do sistema em separado, a geração de um modelo para o módulo *Pump Controller* 4.10, permite-nos após algum trabalho de modelação, obter um modelo Uppaal representativo de todo o sistema.

Esta tarefa passará por gerar com a ferramenta um modelo para cada

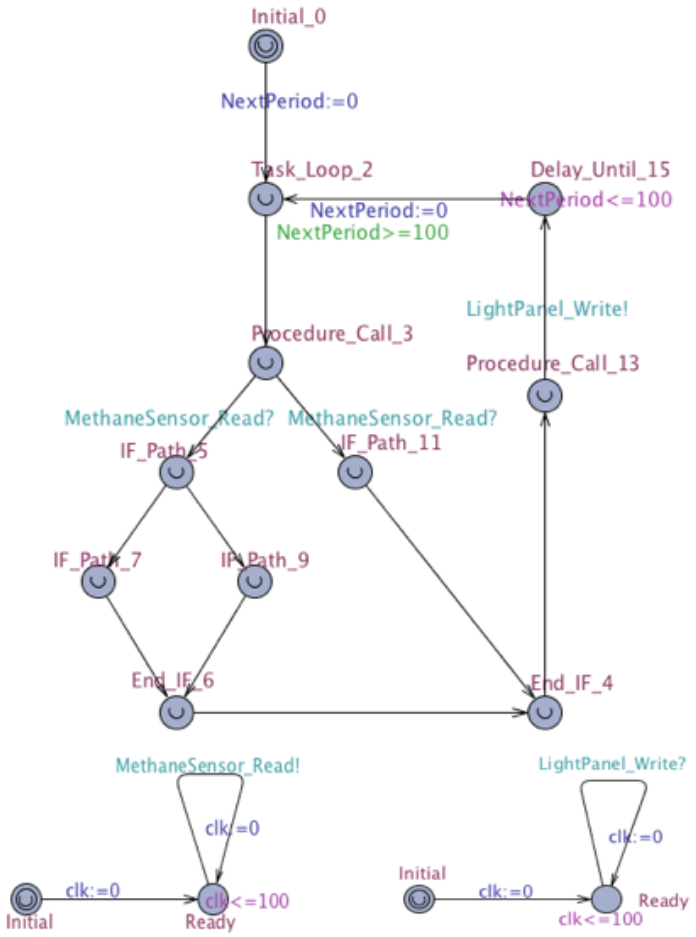

Figura 4.9: Methane Monitor

um dos subsistemas, os quais poderão ser reunidos num único após algum esforço de modelação. Esse esforço passa por pequenos acertos aos estados e transições, mas o grosso do trabalho centra-se no acerto das sincronizações entre os diversos autómatos.

Apesar de não ser esse o objectivo principal deste exemplo, esta metodologia traz naturalmente vantagens quanto à verificação de sistemas deste género. Assumir que cada subsistema já foi analisado e verificado individualmente, permite direccionar as nossas preocupações para o comportamento genérico do sistema, ou seja as propriedades especificadas para este modelo, não necessitam de contemplar aspectos particulares de cada subsistema, mas antes aspectos respeitantes ao comportamento do conjunto dos autómatos a cooperar em simultâneo. Desta forma, é possível então obter um maior grau de confiança no sistema, pois são analisados os aspectos individuais assim como os aspectos gerais do sistema. No

entanto, é indispensável ter atenção a possibilidade de ocorrer explosão de estados aquando da execução do *model checker*. Portanto, devemos ter em conta na modelação possíveis optimizações de forma a que este problema seja mais tarde evitado.

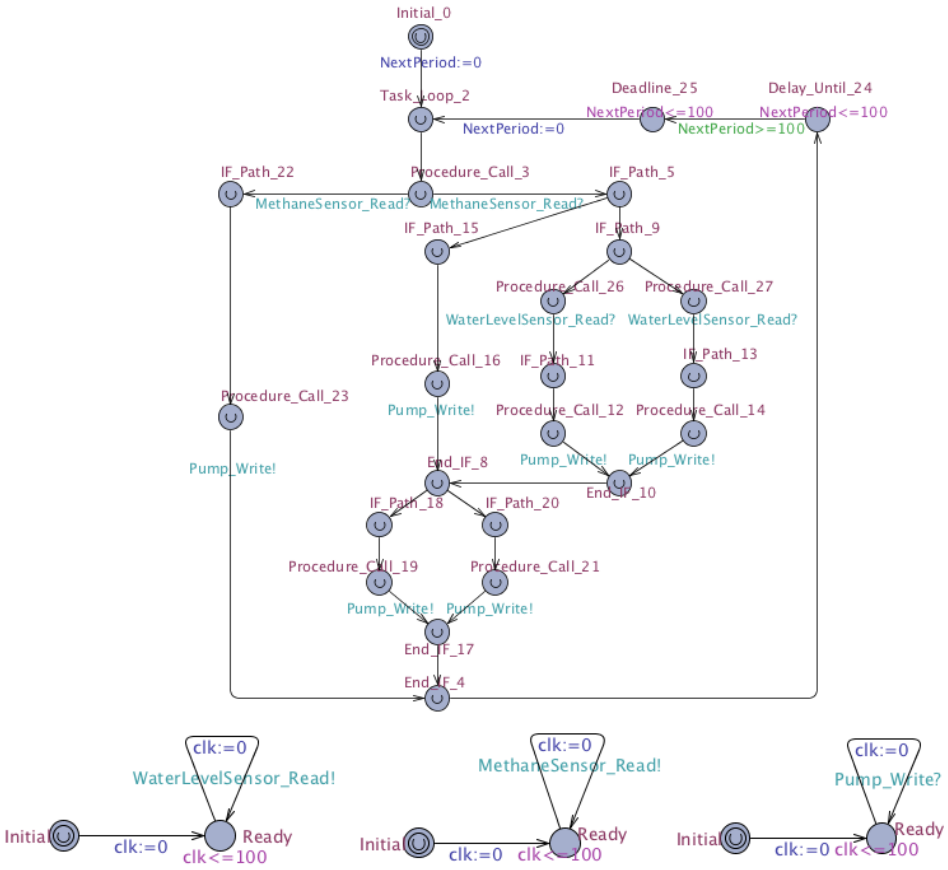

Figura 4.10: Pump Controller

Concluindo, neste exemplo fica evidente uma nova característica da ferramenta desenvolvida. Apesar de se estar a referir este aspecto apenas nesta secção, esta funcionalidade esteve sempre em mente durante o desenvolvimento da ferramenta, pelo que mesmo não sendo esse o objectivo principal, sempre se acreditou que esta poderia representar também um contributo para a modelação de sistemas de tempo-real. Assim, a principal conclusão que se deve tirar deste caso de estudo, prende-se então com o facto de a ferramenta representar uma ajuda significativa na modelação deste tipo de sistemas.

4.4 Conclusão

Neste capítulo foram apresentados alguns casos de estudo que permitiram ter uma perspectiva geral sobre a forma de funcionamento e utilização da ferramenta desenvolvida no contexto desta dissertação. Como se pode constatar cada um dos exemplos tem um objectivo diferente, permitindo assim ter uma noção mais genérica daquilo que a ferramenta permite fazer. No primeiro exemplo, incidiu-se mais na verificação, usando-se para isso o caso de estudo mais utilizado nos trabalhos de Alan Burns publicados. No segundo, apesar da primeira instância do modelo gerado estar correcta perante as propriedades também elas geradas pela ferramenta, a utilização do Uppaal permitiu-nos especificar outras propriedades igualmente importantes, de forma a que todos os requisitos temporais fossem contemplados. Além disso, graças a essa experiência de uso da ferramenta Uppaal e consequente análise exaustiva do sistema modelado, permitiu-nos identificar um ponto de falha no sistema representado, relacionado com a interacção entre os objectos protegidos e o sistema propriamente dito. Deste último facto, advém ainda a identificação de um item considerado para trabalho futuro, que está relacionado com a geração de autómatos representativos de objectos protegidos, baseados também eles nos CFG's dos programas. Por último mas não menos importante, pretendeu-se ilustrar a potencialidade da ferramenta para a geração dos diferentes módulos que os sistemas analisados possam apresentar, e dessa forma (modularmente) podermos analisar passo a passo o comportamento do sistema complexo, ou se quisermos partes do sistema em separado. Na verdade, sempre se acreditou que a ferramenta poderia ser mais do que uma ponte entre o código e um modelo verificável (ou seja, com preocupações direccionadas apenas para a verificação), e é isso que se pretende demonstrar neste último exemplo, ou seja a utilização da ferramenta como um gerador automático de modelos, permitindo a modelação de sistemas de uma forma mais fácil. Podemos pensar na ferramenta como algo que constrói a estrutura de uma casa a partir do qual se pode ir enriquecendo com acabamentos à medida de cada um, ou neste contexto à medida do sistema especificado.

Capítulo 5

Trabalho Futuro & Conclusão

5.1 Trabalho Futuro

Com base em todos os detalhes explicados nos dois capítulos anteriores, é possível identificar alguns pontos imprescindíveis para que se possa dar continuidade a todo o esforço despendido neste trabalho de tese.

Apesar de todas as funcionalidades disponibilizadas pela versão actual da ferramenta, a equipa tem a perfeita noção de que existe ainda muito trabalho pela frente, sendo que muitos desses futuros desenvolvimentos estão identificados e planeados.

Em primeiro lugar, convém referir a dificuldade que o Ada representa pelo facto de apresentar uma sintaxe bastante extensa, sendo portanto necessário neste momento proceder a uma revisão da mesma de forma a incluir todas as construções da linguagem no *parser* da ferramenta. Assim sendo o ponto de trabalho imediato passará por esse mesmo esforço.

Após este esforço, será necessário investigar uma forma de responder ao problema levantado no exemplo *Auto Pilot* (secção 4.2). Recapitulando esse exemplo, existem situações em que podem existir determinadas condições temporais difíceis de detectar na ausência da semântica completa do objectos protegidos. Portanto é necessário encontrar uma estratégia que permita à ferramenta gerar também os autómatos representativos dos objectos protegidos dos quais o sistema em questão possa depender. Dessa forma conseguimos garantir a correcção dessas situações, que neste momento é conseguida graças à experiência de quem está a explorar o modelo.

Esta última alteração poderá ter no entanto implicações na complexidade dos modelos gerados. Assim sendo, devemos desenvolver esta funcionalidade com cautela tendo sempre em consideração a possibilidade de

ocorrência de uma explosão de estados. Portanto, esta mudança na ferramenta poderá também implicar um esforço adicional na tentativa de optimizar o modelo de forma a evitar este problema.

Em relação à verificação propriamente dita, é reconhecido que existe ainda algum trabalho a fazer na questão da inferência das propriedades. Será necessário estudar mecanismos que permitam inferir propriedades relativas a aspectos que possam facilmente passar despercebidos.

Além destes aspectos técnicos e não técnicos referidos, a equipa tem uma visão futura da ferramenta que passará por uma suite completa de modelação/verificação, isto é, pretendemos tornar esta ferramenta capaz de traduzir código Ada para modelos Uppaal, proceder à verificação das propriedades necessárias, e finalmente fazer o caminho inverso sugerindo alterações ao código de forma a que as propriedades não satisfeitas possam ser cumpridas. No entanto, até este ponto existe ainda um longo caminho, durante o qual será necessário antes de tudo pensar-se na validação matemática das transformações implementadas pela ferramenta de forma a demonstrar a correcção dos algoritmos desenvolvidos, e aumentar a confiança do utilizador na ferramenta.

5.2 Conclusão

Chegados ao fim da dissertação, torna-se indispensável tecer algumas considerações e apresentar as principais conclusões do projecto. Em primeiro lugar, não podemos deixar de referir o desafio que este trabalho representou, pelo facto de esta ser a primeira incursão pelo "mundo"Ada. Assim sendo, um dos objectivos que podemos dizer que foi atingido com sucesso foi a exploração destas novas matérias. Para isso, foi necessário um grande esforço no estudo e descoberta das características inerentes ao Ada. Esse estudo permitiu antes de tudo obter a confiança de que realmente seria possível desenvolver uma ferramenta capaz de fazer uma tradução automática de código Ada para modelos Uppaal, no entanto não existiam garantias que tal tarefa seria possível.

Neste contexto, foi determinante a existência de trabalhos como os de Alan Burns e a tese do Joel Carvalho, mas não podemos deixar de frisar que grande parte da ajuda partiu do meio em que este projecto se inseriu (RESCUE e EVOLVE) pois permitiu criar uma rede de conhecimentos e experiência determinante nas alturas em que se teve de tomar decisões.

Assim, a principal conclusão a tirar deste trabalho de tese é que realmente é possível desenvolver uma ferramenta capaz de gerar modelos Uppaal com base em código Ada, e a prova disso mesmo é a ferramenta

descrita nos capítulos anteriores desta dissertação. A possibilidade de gerar modelos, permite naturalmente que se explorem e verifiquem os mesmos com o auxílio da ferramenta Uppaal, sendo então essa funcionalidade o grande contributo desta dissertação. Não podemos esquecer que o principal objectivo esteve sempre relacionado com a verificação de programas de tempo real, nomeadamente quanto às suas características temporais. Assim, apesar da necessidade da ferramenta ser melhorada, podemos dizer que o objectivo foi cumprido e que este foi o primeiro passo na existência de uma alternativa de modelação/verificação de programas de tempo real. Além disso, inerente ainda à ferramenta, podemos concluir que um contributo secundário da mesma será aproveitar a possibilidade de se gerarem modelos para facilitar a modelação deste tipo de sistemas.

Apesar do sucesso no cumprimento do objectivo, durante o desenvolvimento foram surgindo naturalmente problemas que requereram um esforço por parte da equipa de modo a serem ultrapassados, por isso mesmo devo salientar que foi uma mais valia aprender a contornar esses problemas com o auxílio de todas as pessoas envolvidas no projecto, sendo portanto este trabalho um contributo para o meu crescimento pessoal. Destaco aqui este esforço conjunto pois é algo que não se encontra reflectido ao longo desta dissertação.

Posto isto, resta referir a minha visão pessoal da ferramenta desenvolvida. Na minha opinião, esta será extremamente útil principalmente em duas situações: uma primeira onde os utilizadores poderão anotar os seus programas e de forma simples obter um modelo explorável e verificável com o Uppaal; e uma segunda onde os utilizadores poderão simplesmente pegar nos seus programas Ada e gerar modelos (abstracções) de modo que possam de uma forma mais fácil explorar e estudar o problema que pretendem resolver.

Finalmente, além de todo o trabalho futuro descrito na secção anterior, há ainda a ambição de publicar um artigo numa conferência adequada ao tema, onde se irá apresentar todo este trabalho desenvolvido durante o ano de tese.

Bibliografia

[1] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proc. of Int. Colloquium on Algorithms, Languages and Programming*, pages 322–335, 1990.

[2] P. N. Amey and B. J. Dobbing. Static analysis of ravenscar programs. *Ada Lett.*, XXIII(4):58–64, 2003.

[3] R. Arm, J. Mantovani, and L. Platania. Bounded model checking of software using smt solvers instead of sat solvers. In *In: SPIN. Volume 3925 of LNCS*, pages 146–162. Springer, 2006.

[4] G. Behrmann. A performance study of distributed timed automata reachability analysis. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[5] G. Behrmann. Distributed reachability analysis in timed automata. *Int. J. Softw. Tools Technol. Transf.*, 7(1):19–30, 2005.

[6] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. 2009.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[8] A. Burns. The ravenscar profile. *Ada Lett.*, XIX(4):49–52, 1999.

[9] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2):1–74, 2004.

[10] A. Burns and T.-M. Lin. Adding temporal annotations and associated verification to ravenscar profile. In J.-P. Rosen and A. Strohmeier, editors, *Reliable Software Technologies—Ada-Europe 2003*, volume 2655 of *Lecture Notes in Computer Science*, pages 80–91. Springer-Verlag, 2003. Ravenscar Profile, Model Checking, UPAAL, SPARK.

[11] A. Burns and T. M. Lin. An engineering process for the verification of real-time systems. *Form. Asp. Comput.*, 19(1):111–136, 2007.

[12] A. Burns and A. Wellings. Delivering real-time behaviour. In C. George, Z. Liu, and J. Woodcock, editors, *Domain Modeling and Duration Calculus*, volume LNCS 4710 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2007.

[13] A. Burns and A. J. Wellings. Restricted tasking models. In *IRTAW '97: Proceedings of the eighth international workshop on Real-Time Ada*, pages 27–32, New York, NY, USA, 1997. ACM.

[14] B. Carré and J. Garnsworthy. Spark—an annotated ada subset for safety-critical programming. In *TRI-Ada '90: Proceedings of the conference on TRI-ADA '90*, pages 392–402, New York, NY, USA, 1990. ACM.

[15] A. Carvalho, J. Carvalho, J. Pinto, and S. a. Melo. Model-checking temporal properties of real-time htl programs. In *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[16] J. Carvalho. Verificação automatizada de sistemas de tempo-real críticos. Master's thesis, Universidade da Beira Interior, 2009.

[17] R. Chapman. Spark - a state-of-the-practice approach to the common criteria implementation requirements, 2001.

[18] M. A. Criley. Avatox (ada, via asis, to xml), Aug. 2007.

[19] M. A. Cunha. A perspective on model checking, 2007.

[20] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state explosion problem in model checking. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '02, pages 620–631, London, UK, UK, 2002. Springer-Verlag.

[21] B. Dobbing and M. Richard-Foy. T-smart—task-safe, minimal ada realtime toolset. In *IRTAW '97: Proceedings of the eighth international workshop on Real-Time Ada*, pages 45–50, New York, NY, USA, 1997. ACM.

[22] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[23] C. Fidge, I. Hayes, and G. Watson. The deadline command. 1998.

[24] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[25] A. Ghosal. *A hierarchical coordination language for reliable real-time tasks*. PhD thesis, University of California, Berkeley, CA, USA, January 2008.

[26] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, New York, NY, USA, 2006. ACM.

[27] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the IEEE*, pages 166–184. Springer-Verlag, 2000.

[28] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with giotto, 2001.

[29] G. J. Holzmann. Basic spin manual. Technical report, 1994.

[30] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[31] M. Huth and M. Ryan. Logic in computer science: Modelling and reasoning about systems, 1999.

[32] D. Iercan. *Contributions to the Development of Real-Time Programming Techniques and Technologies*. PhD thesis, University of California, Berkeley, CA, USA, September 2008.

[33] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[34] J. Katoen. Concepts, algorithms and tools for model checking, 1999.

[35] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[36] K. Lundqvist and L. Asplund. A formal model of the ada ravenscar tasking profile; delay until. *Ada Lett.*, XIX(3):15–21, 1999.

[37] F. Mueller. Real-time schedulability analysis for ada.

[38] S. Sousa and J. Carvalho. Especificação e verificação de sistema de tempo real: Introdução ao uppaal, 2009.

[39] S. Tripakis and C. Courcoubetis. Extending promela and spin for real-time (extended abstract).

[40] P. J. Whysall. *Object oriented specification and refinement*. PhD thesis, 1991.

[41] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

Apêndice: Artigo Joel Carvalho et al.

Model-Checking Temporal Properties of Real-Time HTL Programs

André Carvalho[2], Joel Carvalho[1], Jorge Sousa Pinto[2], and Simão Melo de Sousa[1]

[1] Departamento de Informática, Universidade da Beira Interior, Portugal, and LIACC, Universidade do Porto, Portugal
[2] Departamento de Informática / CCTC Universidade do Minho, Braga, Portugal

Abstract. This paper describes a tool-supported method for the formal verification of timed properties of HTL programs, supported by the automated translation tool HTL2XTA, which extracts from a HTL program (i) an Uppaal model and (ii) a set of properties that state the compliance of the model with certain automatically inferred temporal constraints. These can be manually extended with other temporal properties provided by the user. The paper introduces the details of the proposed mechanisms as well as the results of our experimental validation.

1 Introduction

New requirements arise from the continuous evolution of computer systems. Processing power alone is not sufficient to satisfy all the industrial requirements. For instance in the context of critical systems, the safety and reliability aspects are fundamental [14]: it is not sufficient to merely provide the technical means for a set of tasks to be executed; it is also required that the system (as a whole) can correctly execute all of the tasks in due time. The focus of this paper is precisely on the reliability of safety-critical systems. Such systems are usually real-time systems [11] that add to traditional reliability requirements the intrinsic need to ensure that tasks are executed within a well-established time scope. For such systems, missing these timing requirements corresponds to a system failure.

Our study considers the Hierarchical Timing Language (HTL) [5, 6, 10] as a basis for real-time system development, and addresses the issue of the (automated) formal verification of timing requirements. Since HTL is a coordination language [4] for which schedulability analysis is decidable, our focus here is on the verification of complementary timing properties. The verification framework we propose relies on model checking based on timed automata and timed temporal logic. The contribution of this paper is a detailed description of the methodology and its underlying tool-supported verification mechanism.

Our tool takes as input a HTL program and extracts from it an Uppaal model and a set of proof obligations that correspond to certain expected timed temporal properties. The resulting model can be used to run a timed simulation of the program execution, and the properties can be checked using the proof facilities provided by the Uppaal tool. With the help of these mechanisms, the development team can audit the program against the expected temporal behaviour.

Motivation and Related Work. The HTL language is derived from Giotto [9]. Giotto-based languages share the important feature that they allow one to statically determine the schedulability of programs. Although academic, these languages have a number of interesting properties that cannot be found in languages currently used in industry, including efficient reuse of code; theoretical ease of adaptation of a program to several platforms; hierarquical construction of programs; and the use of functional features of languages without limitations.

HTL introduces several improvements with respect to Giotto, but the HTL platform still lacks verification mechanisms to complement schedulability analysis, in order to allow the language to compete with other tools more widely used in industry. Bearing in mind this aspect, we propose to complement the verification of temporal HTL with model checking [3]. While the static analysis performed by the HTL compiler enforces the schedulability (seen as a safety property) of the set of tasks in a program, a model checker allows the system designer to perform a temporal analysis of the tasks' behaviour from the specified timing requirements – an aspect that is ignored by the HTL tools.

The verification methodology proposed in this paper is inspired by [13], but uses a different abstraction based on the *logical execution time* of each task. Unlike [13], a key point of our tool chain is that the verification is fully automatic. [12] proposes the use of Uppaal with a related goal: the verification of a Ravenscar-compliant scheduler for Ada applications.

HTL. The Hierarchical Timing Language [5–7, 10] is a coordination language [4] for real-time critical systems with periodic tasks, which allows for the static verification of the schedulability of the implemented tasks. The aim of coordination languages is the combination and manipulation of programs written in heterogeneous programming languages. A system may be implemented by providing a set of tasks written in possibly different programming languages, together with a HTL layer, and additionally specifying how the tasks interact. This favours a clear separation, in the system design, between the functional layer and the concurrent and temporal aspects. The HTL toolchain provides code generators that translate the HTL layer into executable code of the target execution platform.

A fundamental aspect of HTL is the *Logical Execution Time* (LET), that provides an abstraction for the physical execution of tasks. The LET of a task considers a time scope in which the task can be executed regardless of how the operating system assigns resources to this task. The LET of a periodic task implementing a *read data; process; write data* cycle begins in the instant when the last variable is read and ends when the first variable is written.

For illustration purposes, we give in Listing 1.1 an excerpt of a HTL program (based on the 3TS_Simulink case study, see Section 5). A HTL program is composed by a number of main commands which allow programmers to describe the desired behaviour of almost any program. These commands are *communicator*, *module*, *task*, *port*, *mode*, *invoke* and *switch*. Briefly, a communicator is a typed variable which can be accessed any time during the execution; modules have to be declared after communicators and their bodies are composed by ports, tasks and modes. At least one (initial) mode must be declared. The task command,

```
1  module IO start readWrite{
2    task t_read
3         input() state()
4         output(c_double p_h1, c_double p_h2,c_bool p_V1, c_bool p_V2)
5         function f_read;
6    (...)
7
8    mode readWrite period 500{
9         invoke t_read
10             input()
11             output((h1,3), (h2,3), (v1,1), (v2,1));
12        (...)
13   }
14 }
```

Listing 1.1. 3TS Simulink code snippet

as the name indicates, is used to declare tasks, taking as arguments possible input/output ports and a Worst Case Execution Time (WCET) estimation. Similarly to a communicator, a port is a typed variable accessed during program execution, but in this case declared inside a module. The set of modes declared inside a module defines the module's behaviour. Through the modes declaration it is possible to know which tasks will be executed, and at which moment. The invocations are responsible for dictating when the tasks should be executed, and define the LET of each task. Finally, the switch command, which takes as input a condition and a mode identifier, is used to change the current execution mode.

HTL favours a layered approach to the development of programs. Tasks can be organized in refinements that allow programmers to provide details gradually, and also allow for a more finely grained task structure. A concrete task refines an abstract task if it has the same frequency as the abstract task and it is able to provide a time behaviour that is at least as good as the behaviour of the abstract task. The notion of refinement correctness is then expressed in terms of *time safety*. The refined task must be time-indistinguishable from the abstract task; a concrete HTL program is schedulable if it contains only time-safe refinements of the tasks of a schedulable abstract HTL program.

Uppaal. The Uppaal tool is a modelling application developed at the universities of **Upp**sala and **Aal**borg, based on networks of timed automata [2]. The tool offers simulation and verification functionality based on model checking of formulas of a subset of the TCTL logic [1]. Uppaal is particularly suitable for modeling and analysing the timed behaviour of a set of tasks; properties like *two given tasks t_1, t_2 do not reach the states A and B simultaneously* are typical of the kind of analyses that can be performed with Uppaal.

Since the model checking engine is independent from the GUI, both visual and textual representations of timed automata can be used for the verification tasks. This is particularly interesting when Uppaal is used in cooperation with other tools. Timing requirements (target properties to be checked) can be specified using the editing facilities of the GUI, or separately in a file. This last approach is used by the toolchain introduced in this paper.

2 The HTL2XTA Toolchain

The purpose of the verification methodology proposed in this paper is to extend the verification capabilities provided by the HTL platform. Given a HTL program and the schedulability analysis provided by the regular HTL toolchain [7], the methodology consists in the following two steps:

1. From a HTL program, the HTL2XTA translator produces two files: one (.xta) contains a model of the program (timed automata); the other (.q) a set of automatically inferred properties (timed temporal logic formulas). The translation algorithm has a recursive structure and requires only two depth-first traversals of the AST: the first one produces the model and the second one infers the properties.
2. Both these files are fed to the Uppaal model checker; the GUI or the model checker engine (*verifyta*) can be used to check if the properties are satisfied.

We remark that the automatically generated properties correspond to relatively simple timing requirements; formulas for more complex requirements, such as "task X must not execute at the same time as task Y", or "if task X executes, then after T time units task Y must also execute" are not automatically generated, but can of course be manually incorporated in the .q file after the first step above. Writing the appropriate TCTL formulas must of course take into consideration the requirements and the generated model. We now turn to an exploration of the involved translation mechanisms, which will be detailed in the next two sections.

Model Translation. With the classic state space explosion limitation of model checking [3] in mind, and given the central role of the models in the verification process, it was decided to avoid translation schemes that would result in the construction of very complex models. Therefore, and given that the HTL platform already performs a scheduling analysis, the translation abstracts away from the physical execution of tasks, unlike, say, the approach described in [13]. As such, we consider that the notion of LET is sufficient to allow the remaining interesting timing properties to be checked. A network of timed automata is then obtained from a HTL program as follows:

- Each task is modeled as a single automaton with its own LET, calculated from the concrete ports and the communicators given in the task's declaration. The lower bound of the LET corresponds to the instant in which the last variable reading is performed, and the upper bound to the instant in which the first variable writing is performed.
- For each module in the HTL program a timed automaton is created. Note that each mode in a module represents the execution of a set of tasks, and that, at any moment, each module can only be in one operation mode, thus there is no need to have more than one automaton for each module. Whenever the (module) automaton performs an execution cycle, it will synchronize

```
1  /* Deadlock Free -> true */
2  A[] not deadlock
3
4  /* P1 mode readWrite period 500 @ Line 19 -> true */
5  A[] sP_3TS_IO.readWrite imply ((not sP_3TS_IO.t>500) && (not sP_3TS_IO.t
       <0))
6
7  /* P2 mode readWrite period 500 @ Line 19 -> true */
8  sP_3TS_IO.readWrite --> (sP_3TS_IO.Ready && (sP_3TS_IO.t==0  ||
       sP_3TS_IO.t==500))
9
10 /* P1 Let of t_write = [400;500] @ Line 21 -> true */
11 A[] (IO_readWrite_t_write.Let imply (not IO_readWrite_t_write.tt<400 &&
       not IO_readWrite_t_write.tt>500))
```

Listing 1.2. Example of annotated properties

with the automata representing the tasks invoked in the specified mode. The level of abstraction adopted completely ignores the type of communicator as well as the initialization driver.

Since HTL is a hierarchical coordination language, a very relevant aspect is the number of refinements in the program (directly related to program hierarchies), which can naturally increase the complexity of the model substantially. By default, the translation process reflects faithfully the refinement present in the HTL programs. However in some cases this could make the model exploration impracticable, and for this reason the translator allows for the construction of models to take the desired level of refinement as input.

Inference of Properties. Listing 1.2 shows examples of automatically produced timing properties. The automatically inferred properties are all related with some HTL feature, like the modes' periods, the LET of each task, the tasks invoked in each mode, and the program refinement. To allow for traceability, each property is annotated with a textual description of the feature to check, a reference to the position of the respective feature in the HTL file, and the expected verification result. The inferred properties can and should be manually complemented with information extracted from the established temporal requirements. The automata corresponding to a given module and tasks, as well as the states corresponding to task invocations and LETs, are identified by clearly defined labels, which facilitates writing properties manually.

3 Model Translation

Some aspects of HTL are purely ignored by the translation process, either because they do not bring any relevant information, or because the abstraction level of the model is not sufficient to cope with it. The translation process is syntax-oriented and based on the abstract syntax tree (AST) of the HTL language, which was built using a HTL grammar. It supports all of the HTL language, however there is information that is not analysed or translated by the tool.

Let us consider the definition of the function T that takes as input a HTL program and returns a network of timed automata (NTA). Naturally, this function is defined recursively over the structure of the AST. An auxiliary function A is used for task invocation analysis, that takes as argument a HTL program and returns relevant information to build the NTA.

Translation of Mode_Switch. Consider the abstract representation of a switch instruction as the tuple (n, s, p), where n is the name of the mode for which the change of execution is pretended, s the name of the function (in the functional code) that evaluates whether the change should take place, and p the declaration position in the HTL file. Let *Prog* denote the set of all programs, then we have $\forall switch \in Prog, T_{switch}(n, s, p) = \emptyset$. Note that the non-determinism of Uppaal will be important to guarantee that the modes are alternated during the execution. The translation itself is not affected by any mode switches.

Translation of Types and Initialization Drivers. Let ct be a type and ci the declaration of the initialization driver. We have $\forall dt \in Prog, T_{dt}(ct, ci) = \emptyset$. Neither the type nor the initial value (initialization driver) of a declaration have any impact on the application of the translation process. This information does not contribute to the temporal analysis.

Translation of Task Declarations. Consider the abstract representation of a task as the tuple (n, ip, s, op, f, w, p), where n is the name of the task, ip the list of *input ports*, s the list of internal states, op the list of *output ports*, f the name of the function which implements the task, w the task's WCET, and finally p the task declaration position in the HTL file. We have $\forall task \in Prog, T_{task}(n, ip, s, op, f, w, p) = \emptyset$. Analogously to the previous situations, task declarations do not have any impact on the translation.

Translation of Communicator Declarations. Consider the abstract representation of a communicator as the tuple (n, dt, pd, p), where n is the communicator's name, dt the communicator's type with ct, ci as initialization driver, pd the communicator's period and p the communicator's declaration position in the HTL file, then $\forall communicator \in Prog, T_{communicator}(n, dt, pd, p) = \emptyset$.

Once more the translator ignores the declaration. In order to evaluate the LET (see below) the following clause is defined for the auxiliary function A: $\forall communicator \in Prog,\ A_{communicator}(com, dt, pd, p) = pd$, even if the communicator *com* does not have a direct representation in the model given the abstraction level adopted.

Translation of Ports Declaration. Let the abstract representation of a port be the tuple (n, dt, p), where n is the ports's name, dt the port's type with ct, ci the initialization driver and p the port's declaration position in the HTL file, then $\forall port \in Prog, T_{port}(n, dt, p) = \emptyset$. The port's declaration is ignored in the translation, and moreover no task invocation analysis is performed. In task invocations ports are just names.

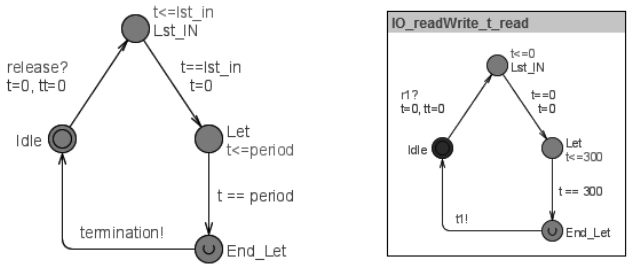

Fig. 1. $taskTA$ automata on the left and instantiation on the right

LET Transposition

This translation is based on an implementation of the concept of LET, based on the timed automata *taskTA*, *taskTA_S*, *taskTA_R* and *taskTA_SR*. These four automata result from the use of concrete ports in the task invocations: $taskTA$ represents the task invocations where only communicators are used, $taskTA\_S$ (S for *send*) those where a single concrete port is used as output, $taskTA\_R$ (R for *receive*) those where a single concrete port is used as input, and finally $taskTA\_SR$ where two concrete ports are used, as input and as output.

In the following, a task invocation will be seen in abstract terms as the tuple (n, ip, op, s, pos) where n is the invoked task's name, ip is the input port's (variables) mapping, op the output port's (variable) mapping, s the name of the task's parent, and finally pos is the task's declaration position in the HTL file.

TaskTA. Let $Port$ be the set of all concrete ports, cp be one concrete port, and $taskTA(r, t, p, li)$ be a timed automaton where r is a *release* urgent synchronization, t is a *termination* urgent synchronization, p the task's LET period and li the exact moment where the last variable is read. Then we have

$$\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \notin op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA(r, t, p, li)$$

Each task invocation in which no concrete ports are used either in the input or in the output variables, gives rise to an automaton *taskTA* (see Figure 1). The urgent synchronization channels r and t are calculated in the system declaration. For each task instantiation the channel r has an unique name, produced by an enumeration $r_1, r_2, r_3, \ldots$ Similarly, the channel t has an unique name for each set of mode automata, produced by an enumeration $t_1, t_2, t_3, \ldots$

The instant at which the last input variable li is read is calculated as a product of the maximum value of each instance of input communicator and the period (in the case of non-existence of input variable, this instant is considered to be zero). The LET's period p is the subtraction between the instant where the

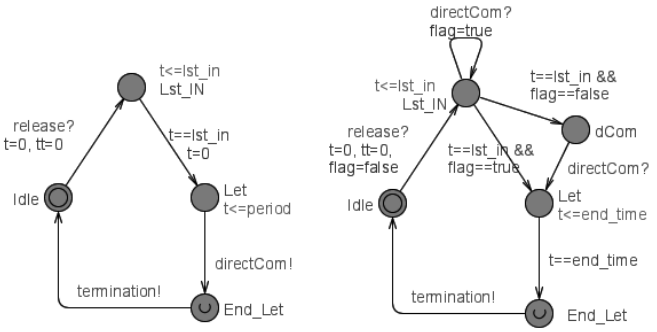

Fig. 2. $taskTA\_S$ automaton on the left and $taskTA\_R$ on the right

first output port is written (in case of non-existence, the value is the respective mode's period) and li.

TaskTA_S. Let $taskTA\_S(r, t, dc, p, li)$ be a timed automaton where r is the *release* urgent synchronization, t a *termination* urgent synchronization, dc the urgent synchronization of a direct communication (*directCom*), p the task's LET period and li the instant where the last input variable is read, then

$$\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \in op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA\_S(r, t, dc, p, li)$$

For each task invocation, the existence of a concrete port in the set of output variables and non-existence in the set of input variables originates the instantiation of a $taskTA\_S$ automaton (Figure 2, left). This automaton is very similar to $taskTA$ – the difference is just the inclusion of direct communication.

TaskTA_R. Let $taskTA\_R(r, t, dc, p, li)$ be a timed automaton with r, t, dc, p, and li the same as in the previous case, then

$$\forall cp \in Port, \forall invoke \in Prog, cp \in ip, cp \notin op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA\_R(r, t, dc, p, li)$$

For each task invocation, the existence of a concrete port in the set of input variables and non-existence in the set of output variables originates the instantiation of $taskTA\_R$ automaton (Figure 2, right). This automaton is slightly more complex than $taskTA\_S$ since it considers two alternative paths for the initialization of the task's LET. The first one encodes the direct communication done before the reading of the last communicator (with no impact on the LET's start) and the later encodes awaiting of the port reading after all communicators have been read (the LET's start becomes dynamic). In this last case, the start of the LET depends on a direct communication with another task in the same mode.

Modules and Modes. Consider a module abstracted as a tuple (n, h, mi, bm, pos), where n is the module's name, h a list of hosts, mi the initial mode, bmu the module's body and pos the module's declaration position in the HTL file.

Let $moduleTA(ref, rl, tl)$ be a timed automaton with ref the refinement's urgent synchronization channel (if it exists), rl the set of all release urgent synchronization channels coming from the invocations of module tasks, and finally tl the set of all termination urgent synchronization channels coming from the invocations of module tasks, then

$$\forall module \in Prog, T_{module}(n, h, mi, bm, pos) = moduleTA(ref, rl, tl)$$

For each module a timed automaton is dynamically created. Unlike tasks automata, where the instantiation of the different default automata is done by just matching the input parameters, here a single automaton is attributed to each module, instantiated by passing as parameters the synchronization channels used by the module's task invocations.

Consider now a mode abstracted as the tuple $(n, p, refP, bmo, pos)$, where n is the mode's name, p is the period, $refP$ is the refinement program for that mode (if it exists), bmo the mode's body, and pos the mode's declaration position in the HTL file. Let $subModule(e, t)$ be a subset of the timed automaton's $moduleTA$ declaration where e is the set of states (with invariants) and t the set of transitions (with guards, updates and synchronizations), then we have

$$\begin{aligned}\forall mode \in module, \exists subModule(e, t) \in moduleTA,\\ T_{mode}(n, p, refP, bmo, pos) = subModule(e, t)\end{aligned}$$

4 Inference of Properties

This section presents the definition of a function P which accepts a HTL program and returns the specification of properties to verify. Naturally, this function is again defined recursively over the AST structure of the HTL language.

Absence of Block. Let $Prog$ be the set of all programs and df be the absence of blocking property description, then we have $P_{Prog} = df$. The application of this method to any program always produces the same absence of blocking property ($A\,\Box\ not\ deadlock$).

Modes Period. Let the tuple $(n, p, refP, bmo, pos)$ be the abstraction of a mode, where n is the mode's name, p the period, $refP$ the refinement program for that mode (if it exists), bmo the mode's body and pos the mode's declaration position in the HTL file. In the following vm denotes the property specifications of a mode's period. We have

$$\forall mode \in Prog, P_{mode}(n, p, refP, bmo, pos) = vm(p1, p2)$$

We also have, with *moduleTA* a module automaton and *NTA* a set of timed automata,

$$\begin{aligned}
&\forall mode \in Prog, \exists moduleTA \in NTA,\\
&\quad p1 = A\,\square\, moduleTA.n \Rightarrow ((\neg\ moduleTA.t > p) \wedge\ (\neg\ moduleTA.t < 0)),\\
&\quad p2 = moduleTA.n \Rightarrow (moduleTA.Ready\\
&\qquad \wedge\ (moduleTA.t == 0\ \vee\ moduleTA.t == p)
\end{aligned}$$

The first property $p1$ states that whenever the control state is the mode state, the module's (automaton) local clock is lower than the mode's period, and not negative. The second property $p2$ on the other hand states that whenever the mode's state is reached, the state *Ready* is also reached, which implies that the local clock is either zero or exactly equal to the period's value. The combination of both properties allows the restriction of a mode's period to the interval $[0, p]$, and guarantees that the period's maximum value is reached.

Task Invocations. Let (n, ip, op, s, pos) be a task invocation, where n is the task's name, ip the input port's (variables) mapping, op the output port's (variable) mapping, s the name of the parent task and finally pos the task's declaration position in the HTL file. In the following vi denotes the specification of properties in a mode's task invocation. We have

$$\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vi(p1, p2)$$

Let $taskTA_i$ be the automaton of task i, *taskTA* the set of task automata, $taskState_i$ the task i invocation's state, *modeState* the mode's state where the invocation is done, *moduleTA* a module automaton and *NTA* a set of timed automata, then

$$\begin{aligned}
&\forall i, \exists moduleTA \in NTA, \exists taskTA_i \in TaskTA,\\
&\quad p1 = A\,\square\,(moduleTA.taskStaste_i \Rightarrow (\neg\ taskTA_i.Idle))\\
&\qquad \wedge\ (moduleTA.Ready \Rightarrow taskTA_i.Idle),\\
&\quad p2 = A\,\square\,(taskTA_i.Let\ \wedge\ taskTA.tt! = 0) \Rightarrow moduleTA.modeState
\end{aligned}$$

The property $p1$ states that for all executions, every time an invocation's state is equal to a control state, that task's automaton cannot be in the *Idle* state. Moreover, when the respective *moduleTA*'s control state is equal to *Ready*, the task's automaton must be in the *Idle* state. The second property specifies that whenever a task's automaton *Let* state is the control state and the local clock *tt* is different from zero, the execution of the module's automaton must be in the state representing the mode in which the tasks are invoked.

Tasks LET. Considering a task invocation *vlet* in a correct mode, its properties are specified as

$$\begin{aligned}
&\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vlet(p1, p2, p3), \forall i, \exists moduleTA \in NTA,\\
&\quad p1 = A\,\square\,(taskTA_i.Let \Rightarrow (\neg\ taskTA_i.tt < 0\ \wedge\ \neg\ taskTA_i.tt > p)),\\
&\quad p2 = A \diamond moduleTA.modeState \Rightarrow (taskTA_i.Lst_IN\ \wedge\ taskTA_i.tt == 0),\\
&\quad p3 = A \diamond moduleTA.modeState \Rightarrow (taskTA_i.Let\ \wedge\ taskTA_i.tt == p)
\end{aligned}$$

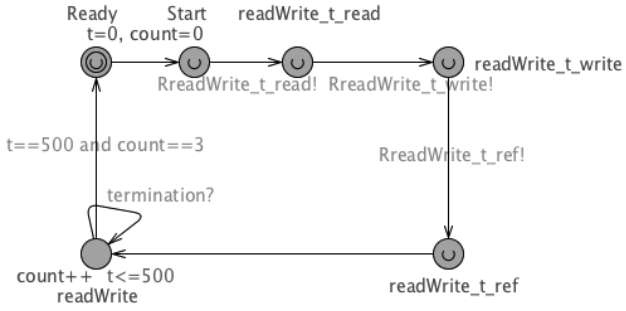

Fig. 3. `P_3TS_IO` automaton (automatic instantiation of `taskTA`)

The LET's validation is done via three distinct properties. Property $p1$ specifies that whenever a task's *Let* is reached, the automaton's local clock tt must lie between 0 and the period. Property $p2$ specifies that every time the mode's state is reached, the Lst_IN state is also reached, necessarily with the local clock tt set to zero. Finally, property $p3$ specifies that every time the mode's state is reached, the *Let* state is inevitably reached, with the clock tt set to the maximum value of the task's period.

5 Case Studies

We consider here the main case study used for illustration purposes by the HTL team: the *three-tank system*. A HTL program implements the controller of a physical system that includes three interconnected tanks with two pumps (for tanks 1 and 3), three taps (one for each tank) and two interconnection taps. The controller supervises the taps in order to maintain the liquid at a specific level.

Description of the Problem. The controller is implemented as a program that contains three modules; two of them (T1 and T2) specify the timing for the controllers of tanks T1 and T2, and the third module specifies the timing for the communications (IO) controller. Each controller module contains one mode which invokes one task and which is refined by a program into a P or PI controller. We assume that in addition to height measuring sensors there exist also sensors that detect perturbation in a tank (this determines the switch between P and PI). The IO module contains one mode named `readWrite` and invokes three tasks: `t_read` reads sensor values and updates communicators `h1`, `h2`, `v1` and `v2`; `t_write` reads communicators `u1` and `u2` and sends commands to the pumps; `t_ref` reads target values and updates communicators `h1_ref` and `h2_ref`.

Generated Model (excerpt). The HTL program is translated into a network consisting of nine timed automata, of which four are default automata that are instantiated by each task invocation depending on the modules and ports declared;

File	Levels	HTL	Model	Verifications	States
3TS-simulink.htl	0	75	263	62/62	7'566
	1	75	199	30/30	666
3TS.htl	0	90	271	72/72	18'731
	1	90	207	40/40	1'123
3TS-FE2.htl	0	134	336	106/106	280'997
	1	134	208	42/42	1'580
3TS-PhD.htl	0	111	329	98/98	172'531
	1	111	201	34/34	1'096
steer-by-wire.htl	0	873	1043	617/0	N/A
	1	873	690	394/0	N/A
flatten_3TS.htl	0	60	203	31/31	411

Table 1. Results

the remaining five represent the modules and respective execution modes. Taking as example the IO module, in which three tasks (`t_read`, `t_write` and `t_ref`) and no ports are used, it is translated as three timed automata, with the default `taskTA` automaton instantiated for each task. An example is shown below, extracted from the (.xta) file produced by the tool. Task invocations are represented by signals RreadWrite_t_read, RreadWrite_t_write and RreadWrite_t_ref.

Properties. In abstract terms the automatically inferred properties can be seen as divided in four classes: *Absence of Block*, *Modes Period*, *Task Invocations* and *Tasks' LET*. We give below an excerpt from the (.q) file generated for the 3TS_Simulink program, that shows two classes of properties : *Absence of Block* for the first property shown, and *Modes Period* for the second. The properties are annotated with a descriptive string and the expected verification result.

```
//Deadlock Free -> true
A[] not deadlock

//P1 mode readWrite period 500 @ Line 19 -> true
A[] sP_3TS_IO.readWrite imply ((not sP_3TS_IO.t>500) && (not sP_3TS_IO.t
    <0))
```

In some situations, small modifications in the code can have serious effects in the program and affect the verification of properties. In these cases the solution is to analyse and manually specify properties appropriate to each scenario.

Verification. In this case study the HTL2XTA translator for all levels of refinement (switch -L 0) has generated 62 properties automatically, which were all successfully checked (using *verifyta* version 4.0.10). 291794 states were explored and the maximum number of states consumed by a single property was 7566. Some properties were trivially verified. These numbers contribute to an increased confidence degree on the 3TS_Simulink's HTL specification. In spite of the large number of properties and states, this goal is achieved in reasonable time.

Other Case Studies. Using the current version of the translator it was possible to successfully generate models and properties for several HTL programs from [6, 10], and the HTL website. Table 1 summarizes relevant information

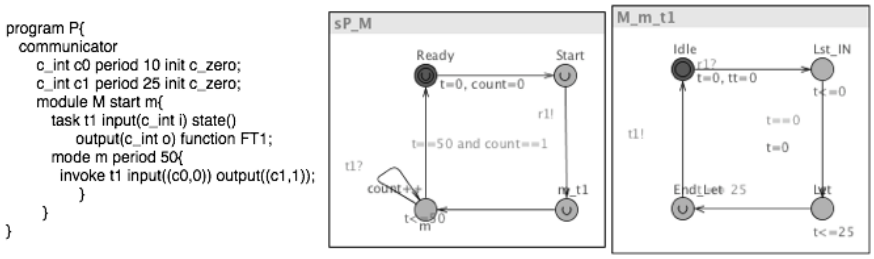

Fig. 4. A misbehaved HTL program and the corresponding Uppaal automata

about the results, specifically the number of applied levels (0=all, 1=main program), the number of lines in the HTL file, the number of lines in the model's specification file, the number of specified properties versus the number of properties successfully verified, and the number of states explored. The values in the table concern the Uppaal verification, given several different models translated from HTL to XTA.

The proposed toolchain was able to cope with all except one program, the more complex *steer-by-wire* example, for which the verification process does not terminate in reasonable time. Clearly, this is due to the use of all the advanced features of HTL (including a large and complex coordination layer).

6 Towards Correctness

The correctness of the proposed approach has not yet been established; we give here some preliminary remarks. The desired correctness property can be formulated as follows, where p is a HTL program, and MC corresponds to execution of the Model Checker.

If $MC(T(p)) = Error$ then there exists an execution that derives to a timed error execution, following the operational semantics of HTL [5].

Although we have not proven such a correctness result, we give here an example to give the reader an intuition of why the approach should in principle be correct.

The example is shown in Figure 4. The period of the last task's (t_1) output is 25 and the first input is 0; the mode's period is 50, so it is trivial to conclude that this system is schedulable. As such, this program is validated by the HTL toolchain. However, this is not satisfactory, since with these values the LET of this task is specified as [0;25]. Due to the period of the communicator c_0 this task must not execute between instants 0 and 9, and the standard HTL toolchain contains no mechanism to specify or prove situations like this.

It is obvious that in such a small example this problem could be easily detected and corrected by simply changing the instant when the c_0 communicator

is used from 0 to 1. But in more complex systems it is hard to obtain any insight about this kind of temporal behaviours.

Considering again the above example, it is straightforward to see that the HTL2XTA translator preserves the bad temporal requirement in the timed automata model. Checking the property $A[]M\_m\_t_1.Let\ imply\ (not(M\_m\_t_1.tt < 10))$, which can be manually inserted in Uppaal and specifies that task t_1 must never occur in an instant inferior to 10, will produce a counter-example.

7 Conclusion and Future Work

The HTL language was created in an academic context, and its transfer to the industrial context remains a challenge. This work is a contribution towards that goal. The tool is available online[3] and runs only on the Linux platform. The HTL2XTA translator was developed in Ocaml, following the traditional compiler design process (but we rely on the HTL compiler for type checking).

We envision two natural improvements of our current methodology. First, the translation methodology has not yet been formally verified (i.e. it has not been proved that the translation preserves the timed semantics of HTL programs). The proof of the theorem sketched in Section 6 is a heavyweight task that must be carefully carried out.

Secondly, the current version of the translator is unable to deal with large-scale HTL programs, and moreover there are still some features of HTL syntax that are not covered by the current version. The translation of the currently covered HTL features can be improved in order to lower the size of the resulting NTA. As future work we plan to analyse these possibilities, and also to extend HTL with *annotations* to introduce supplementary behaviour rules. For instance this may provide insight about the behaviour of programs in the presence of *switch cases*. The impact of such annotations in the model and their influence on the design of the translator will of course have to be carefully considered.

Moreover, in the short term, the toolchain could be improved with a script that provides an automatic analysis of the logfile generated by Uppaal. Such a script could establish conveniently which timing requirements have been checked and which have not, and create a final report based on this information.

Finally, we are interested in transferring our work to the context of the SPARK/Ada language, widely used in the development of safety-critical systems. The *Giotto in Ada* [8] initiative should make this process quite straightforward. We also believe that our translation mechanisms may in principle be applied to other (more exploratory) concurrency models, but this remains an open issue.

Acknowledgment. This work was partially supported by the projects Rescue (PTDC/EIA/65862/2006) and FAVAS (PTDC/EIA-CCO/105034/2008), and by LIACC-UP through the Programa de Financiamento Plurianual, all funded by Fundação para a Ciência e Tecnologia (FCT).

[3] http://sourceforge.net/projects/htl2xta/

References

1. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
2. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools, 2004.
3. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
4. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
5. Arkadeb Ghosal. *A Hierarchical Coordination Language for Reliable Real-Time Tasks*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2008.
6. Arkadeb Ghosal, Thomas A. Henzinger, Daniel Iercan, Christoph Kirsch, and Alberto L. Sangiovanni-Vincentelli. Hierarchical timing language. Technical Report UCB/EECS-2006-79, EECS Department, University of California, Berkeley, May 2006.
7. Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, New York, NY, USA, 2006. ACM.
8. Helge Hagenauer, Norbert Martinek, and Werner Pohlmann. Ada meets giotto. In Albert Llamosí and Alfred Strohmeier, editors, *Ada-Europe*, volume 3063 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2004.
9. Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 166–184, London, UK, 2001. Springer-Verlag.
10. Daniel Iercan. *Contribuitions to the Development of Real-Time Programming Techniques and Technologies*. PhD thesis, EECS Department, University of California, Berkeley, Set 2008.
11. Shem-Tov Levi and Ashok K. Agrawala. *Real-time system design*. McGraw-Hill, Inc., New York, NY, USA, 1990.
12. Kristina Lundqvist and Lars Asplund. A ravenscar-compliant run-time kernel for safety-critical systems*. *Real-Time Syst.*, 24(1):29–54, 2003.
13. Rajiv Kumar Poddar and Purandar Bhaduri. Verification of giotto based embedded control systems. *Nordic J. of Computing*, 13(4):266–293, 2006.
14. John Rushby. Formal methods and their role in the certification of critical systems. Technical report, Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop, 1995.