Joel Silva Carvalho

Verificação Automatizada de Sistemas de Tempo Real Críticos

Universidade da Beira Interior
Departamento de Informática
Junho 2009

Joel Silva Carvalho

Verificação Automatizada de Sistemas de Tempo Real Críticos

Dissertação referente aos trabalhos de investigação conducentes
à obtenção do grau de Mestre em Engenharia Informática,
Orientado pelo Professor Doutor Simão Melo de Sousa

Universidade da Beira Interior
Departamento de Informática
Junho 2009

Agradecimentos

Agradeço ao meu orientador (Doutor Simão Melo de Sousa) a liberdade, apoio e orientação nas escolhas que me levaram à apresentação desta dissertação. Os objectivos foram definidos com o seu consentimento e felizmente cumpridos. Apesar de a dissertação focar uma linguagem académica, com visibilidade menor do que algumas soluções industriais, devo admitir ter desde já atingido um momento de realização pessoal.

Agradeço ainda aos meus pais (Prazeres Barata da Silva Gouveia de Carvalho e José António Chaves de Carvalho) por terem possibilitado a realização deste sonho. Sem eles não teria tido o financiamento e apoio necessário para o percurso académico realizado. Num país onde se apoia e avalia de forma estranha a investigação e o ensino, só alguns conseguem este privilégio.

Agradeço também aos meus colegas do Release, do Socia e do NMCG pelo companheirismo e bom ambiente proporcionado nos diversos momentos do dia-a-dia. Bem como aos professores que tive ao longo de todo meu percurso académico e aos elementos do projecto RESCUE que contribuíram para o enriquecimento dos meus conhecimentos.

Por fim agradeço a todos os meus familiares e amigos pelo apoio.

Conteúdo

Lista de Figuras

Acrónimos

Release RELiablE And SEcure Computation Group

RESCUE REliable and Safe Code execUtion for Embedded systems

HTL Hierarchical Timing Language

TDL Timing Definition Language

CTL Computation Tree Logic

CTL* Full Computation Tree Logic

TCTL Timed Computation Tree Logic

LTL Linear Temporal Logic

LET Logical Execution Time

SCC Schedule Carrying Code

xGiotto eXtended Giotto

IRTAW International Real-Time Ada Workshop

PCC Proof Carrying Code

EDF Earliest Deadline First

EAL Evaluation Assurance Level

CC Common Criteria

E-Code Embedded Code

HE-Code Hierarchical Embedded Code

E-Machine Embedded Machine

HE-Machine Hierarchical Embedded Machine

S-Code Scheduling Code

WCET Worst-Case Execution Time

WCTT Worst-Case Transmission Time

3TS Tree Tank System

ECU Engine Control Unit

ABS Anti-lock Braking System

DBC Design By Contract

SCC Schedule Carrying Code

RAT Rede de Autómatos Temporizados

LPTA Linearly Priced Timed Automata

BNF Backus-Naur Form

AST Abstract Syntax Tree

Capítulo 1

Introdução

A miniaturização das componentes electrónicas permite que os sistemas computacionais tenham uma proliferação acelerada. Estes sistemas estão integrados nos mais diversos meios, por exemplo, no cartão do cidadão, nos telemóveis, nos automóveis, nos aviões, entre outros.

Novas exigências surgem com a evolução destes sistemas computacionais. De facto, a capacidade de processamento, por si só, já não é suficiente para o preenchimento de todos os requisitos industriais. Nos sistemas críticos a segurança e a fiabilidade são os aspectos fundamentais. Apesar de ser importante, não basta reunir condições técnicas para executar um dado conjunto de tarefas num sistema, é preciso que o sistema (como um todo) execute correctamente essas tarefas (cap. 2).

1.1 Objectivo

Esta dissertação foca-se no estudo da fiabilidade de um subconjunto de sistemas computacionais, mais precisamente os Sistemas de Tempo Real Críticos (cap. 2). Comparativamente com os sistemas tradicionais, os sistemas de tempo real acrescentam, à questão da fiabilidade, a necessidade intrínseca de algo ser executado num intervalo de tempo bem definido. Para este tipo de sistemas, não se conseguir finalizar uma tarefa, no tempo que é devido, corresponde sumariamente a uma falha do sistema.

Enquadrada no RELiablE And SEcure Computation Group (Release) e no projecto REliable and Safe Code execUtion for Embedded systems (RESCUE), esta dissertação teve como ponto de partida um estudo sobre os sistemas de tempo real e sobre os diversos mecanismos de verificação aplicados neles (cap. 2). Após terem sido identificadas as linguagens de interesse, foi definido como objectivo principal da dissertação a implementação de um tradutor automatizado da linguagem Hierarchical Timing Language (HTL) (cap. 4) para Uppaal (cap. 3). Este tradutor enquadra-se numa *Tool-Chain* que estende a verificação de programas HTL recorrendo a um mecanismo de verificação formal.

Sendo o HTL uma linguagem de coordenação, capaz de verificar o escalonamento de programas, constatou-se que a mesma podia ser complementada com outro mecanismo de verificação, nomeadamente o *Model Checking* (doravante verificação de modelos). O tradutor apresentado nesta dissertação (cap. 5) constrói, de forma automatizada, modelos com base em programas HTL e especifica propriedades sobre os modelos. Após serem verificadas pelo *Model Checker Uppaal* (cap. 3), as propriedades permitem estabelecer uma relação entre o que o programador produziu e aquilo que se pretende do sistema. Esta relação não é completa, uma vez que só é possível verificar propriedades temporais sobre as tarefas, mas sempre complementa a verificação de escalonamento feita pelo HTL. De notar ainda que, a especificação automatizada de propriedades pode ser enriquecida manualmente permitindo a verificação de requisitos temporais não inferidos pelo tradutor.

1.2 Motivação e Contribuição

Após ter estudado computação ubíqua, na realização do projecto de final de curso, a consciência para a dependência de todo o tipo de sistemas computacionais aumentou. Neste sentido foi impossível ficar indiferente aos desafios inerentes ao desenvolvimento de aplicações para sistemas embebidos. Para além da vontade de estudar sistemas embebidos e sistemas de tempo real, existiu sempre a esperança e vontade de poder contribuir com algo na verificação deste tipo de sistemas.

A liberdade na escolha das linguagens permitiu abordar algo menos convencional. No conjunto de mecanismos de verificação e linguagens estudadas era evidente que, dado o contexto, seria mais arriscado mas mais interessante focar o estudo numa linguagem académica como o HTL. Assim, para além do estudo dos diversos mecanismos de verificação aplicados nos sistemas de tempo real (cap. 2), do estudo mais aprofundado do Uppaal (cap. 3) e do HTL (cap. 4) destaca-se a implementação de um mecanismo de tradução automatizado (cap. 5) que contribui na verificação dos sistemas de tempo real desenvolvidos em HTL. O funcionamento do tradutor foi confirmado com a execução (simulação) e verificação dos casos de estudo disponíveis para o HTL no Uppaal, sendo que nesta dissertação apenas é apresentado detalhadamente o caso de estudo mais recorrente (cap. 6).

1.3 Organização do relatório

Este documento estrutura-se da seguinte forma.

Cap. 1 - Introdução Enquadramento do trabalho desenvolvido no âmbito da dissertação.

Cap. 2 - Verificação de Sistemas de Tempo Real Numa primeira fase deste capítulo são apresentados conceitos fundamentais aos sistemas de tempo real. Numa segunda fase são apresentados os mecanismos de verificação estudados.

Cap. 3 - UPPAAL Introdução de alguns conceitos para uma melhor compreensão das potencialidades da verificação de modelos, baseada em autómatos temporizados e lógica temporal, quer isolada como enquadrada com outros mecanismos de verificação.

Cap. 4 - Hierachichal Timing Language Apresentação da sintaxe da linguagem HTL, síntese das características principais da linguagem e descrição do funcionamento do compilador.

Cap. 5 - HTL2XTA, O Tradutor Descrição do algoritmo de tradução e apresentação das capacidades e fraquezas do tradutor.

Cap. 6 - Validação Experimental Apresentação de um caso de estudo em particular, bem como síntese dos resultados obtidos na verificação e execução de outros casos.

Cap. 7 - Conclusão Apresentação de mais algumas considerações sobre a dissertação.

Capítulo 2

Verificação de Sistemas de Tempo Real

Ao longo deste capítulo são apresentados conceitos fundamentais aos sistemas de tempo real e à sua verificação. Numa primeira parte (referência principal [36]), os conceitos são apresentados de forma sucinta para uma familiarização com o vocabulário e para reflexão sobre alguns cuidados necessários no desenvolvimento de sistemas de tempo real. Numa segunda parte, é apresentado o estudo feito sobre os mecanismos de verificação utilizados neste tipo de sistemas.

2.1 Sistemas de Tempo Real

2.1.1 Definições e Características

Um sistema embebido é um sistema computacional dedicado à realização de tarefas específicas. Este tipo de sistemas distingue-se dos sistemas de propósito geral (desktop, laptop, netbook, etc.) pelas suas limitações e enfoque nas tarefas. Essas limitações são as mais diversas, desde a necessidade de serem de dimensões reduzidas, passando por limitações energéticas e acabando na capacidade reduzida de processamento e armazenamento. Bons exemplos de sistemas embebidos são, os telemóveis, os PDA's, os *Smart Card*, as máquina fotográfica, os micro ondas, entre outros.

Os sistemas de tempo real são geralmente sistemas embebidos porque controlam um sistema genérico [32]. No entanto, um sistema de tempo real é um sistema

em que a correcção das tarefas depende do resultado lógico da computação e do instante em que são executadas. Alguns exemplos de sistemas de tempo real são, os Anti-lock Braking System (ABS) dos carros, as Engine Control Unit (ECU) (mais tradicionalmente conhecidas por centralinas), os *Pacemakers*, entre outros.

Ao contrário do que o nome pode sugerir, este tipo de sistemas não implica que o tempo de resposta seja imediato. Implica sim, que o tempo resposta seja adequado independentemente da unidade temporal (milissegundos, segundos, etc.). O tempo de resposta não depende do tempo médio de execução das tarefas, mas do Worst-Case Execution Time (WCET). Em média os requisitos temporais podem ser cumpridos, mas se no pior caso não forem o sistema fica comprometido.

> ...e então há um homem que se afogou ao atravessar uma corrente com uma profundidade média de seis polegadas... Stankovic em [42]

Segundo Stankovic [41], o importante nos sistemas de tempo real é a previsibilidade e não a rapidez. Os requisitos temporais necessitam de ser conhecidos à partida e deve ser possível identificar quais as tarefas que não cumprem com os requisitos temporais. Como vai ser possível perceber, este aspecto introduz muitas limitações na verificação dos sistemas de tempo real. Quanto mais determinista for o sistema mais simples é verificar propriedades sobre ele.

Outra característica inerente aos sistemas de tempo real reside na concorrência das tarefas. Estes sistemas executam múltiplas tarefas de forma concorrente para conseguir interagir com o ambiente envolvente.

2.1.2 Classificação dos sistemas de tempo real

Uma vez que os sistemas de tempo real se diferenciam pelas suas características é necessário classifica-los. De seguida apresentam-se três classificações seleccionadas de um conjunto maior. Dado que a dissertação aborda questões de periodicidade de tarefas, e estando a verificação mais relacionada com sistemas críticos, estas classificações são imprescindíveis.

2.1.2.1 Quanto à criticidade do sistema

Na classificação quanto à criticidade do sistema, diz-se que um sistema de tempo real crítico (*Hard Real-Time Systems*) implica o cumprimento de todos os requisitos temporais. O não cumprimento de um requisito pode implicar a falha do sistema, situação que não é desejada. Neste tipo de sistemas inclui-se, por exemplo, o sistema de navegação de uma aeronave.

Por sua vez um sistemas de tempo real não crítico (*Soft Real-Time Systems*) não implica o cumprimento de todos os requisitos temporais. O seu cumprimento é desejado mas, quando não verificado, não origina uma falha do sistema. Neste tipo inserem-se, por exemplo, as consolas de jogos.

2.1.2.2 Quanto à criticidade das tarefas

Na classificação quanto à criticidade das tarefas diz-se que uma tarefa crítica é aquela em que o não cumprimento do requisito temporal associado implica uma falha no sistema.

Uma tarefa firme é aquela em que o resultado obtido deixa de ser útil. Neste tipo de tarefas as consequências do incumprimento do requisito temporal tornam-se irrelevantes.

Por fim, a tarefa não crítica, é aquela em que o resultado obtido continua a ter um valor decrescente, mas no qual o incumprimento do requisito temporal não implica uma falha.

2.1.2.3 Quanto à natureza temporal

Neste tipo de classificação, diz-se que as tarefas periódicas são aquelas que são executadas com um período fixo (de x em x unidades de tempo). Este tipo de tarefas é bastante utilizado na leitura de dados de sensores ou na gestão de actuadores.

As tarefas não periódicas são aquelas cujo conhecimento prévio sobre o instante da próxima execução é nulo.

As tarefas aperiódicas, são aquelas que não são críticas e simultaneamente não são periódicas.

Por fim as tarefas esporádicas, são aquelas que são críticas e não são periódicas.

2.1.3 Tolerância a Falhas

Nos sistemas críticos, a possibilidade de existirem perdas de vidas humanas ou perdas materiais relevantes, do ponto de vista financeiro, deve ser a mais reduzida possível. Para além da utilização de métodos formais durante o desenvolvimento e da fase de testes no fim do desenvolvimento, é muitas vezes necessário acrescentar a utilização de mecanismos de tolerância a falhas.

Este tipo de mecanismos permite que o sistema continue em funcionamento sobre a presença de falhas [40]. No entanto os algoritmos de detecção e correcção de falhas introduzem um custo adicional de *overhead*. Assim eles devem ser previstos e introduzidos desde cedo no processo de desenvolvimento, para que os requisitos temporais contemplem a detecção e correcção de falhas em tempo de execução.

Estes mecanismos consistem muitas vezes na introdução de redundância no sistema, quer do ponto de vista espacial (replicação de componentes) como temporal (repetição da execução de tarefas) [40]. No entanto isto pode ser prejudicial à previsibilidade do sistema, pelo que a redundância tem que ser feita garantindo a preservação da previsibilidade e fiabilidade do sistema.

Um dos resultados clássicos da engenharia de software refere que é cerca de vinte vezes mais complicado corrigir uma falha depois do sistema estar desenvolvido. Deste modo, o interesse em evitar falhas durante o desenvolvimento continua a ser predominante.

> O uso de um processo sólido de desenvolvimento é um bom passo na direcção certa, mas outros podem ser dados, como por exemplo, a utilização de ferramentas automáticas para maior parte do processo, ou

a utilização de métodos formais... Luís Miguel Pinho em [36]

2.1.4 Linguagens de Programação

Existem várias linguagens de programação para sistemas de tempo real e todas elas possuem as suas características, no entanto baseiam-se nalguns requisitos.

No que respeita à segurança, as linguagens fortemente tipadas são preferíveis porque permitem forçar o programador a evitar alguns erros de utilização de variáveis. A utilização de recursividade e apontadores deve ser cuidada, preferencialmente até evitada. Estas duas técnicas reduzem significativamente a previsibilidade do sistema.

Relativamente à flexibilidade, a linguagem deve possuir mecanismos capazes de evitar o uso de construções específicas ao hardware ou linguagem *assembly*, no entanto nem sempre assim acontece. Os motivos são diversos, mas basta pensar que por vezes os requisitos temporais são de tal forma complicados de atingir, que é necessário optimizar o código para utilizar características especificas do *hardware* ou do sistema operativo.

Quanto maior for o uso destas técnicas menos portável vai ser o código. Mas como os sistemas de tempo real baseiam-se na avaliação e cumprimento de requisitos temporais, a eficiência é fundamental. Aumentando a eficiência com as técnicas relatadas reduz-se a portabilidade, e vice-versa. Por outro lado, as linguagens devem ser suficientemente simples para permitir implementações fiáveis dos compiladores e de outras ferramentas. Assim, a linguagem ou um subconjunto da mesma deve ser suficiente para permitir algum tipo de verificação formal.

Facilmente se define aqui um ciclo vicioso, não existe nenhuma linguagem perfeita mas sim linguagens que são mais adequadas para um determinado tipo de sistemas e outras para outros. As linguagens de programação, para sistemas de tempo real, devem suportar concorrência, o que implica de alguma forma a capacidade de construção de módulos, a criação e gestão de tarefas, a comunicação entre tarefas, a exclusão no acesso a recursos, a utilização de interrupções e dispositivos

de entrada e saída, o tratamento de excepções e a previsibilidade. Quando estas características não são suportadas pela linguagem de programação então devem ser suportadas pelo sistema operativo.

2.1.5 Sistemas Operativos

Apesar de algumas linguagens permitirem criar aplicações para sistemas de tempo real sem a necessidade de utilização de um sistema operativo, estes tornam a programação mais abstracta. Tal como um sistema operativo tradicional, um sistema operativo de tempo real precisa de módulos para gestão de tarefas, gestão de memória, comunicação e sincronização entre tarefas e gestão de entradas/saídas.

Focando apenas o aspecto relevante para esta dissertação, identifica-se como função principal do módulo de gestão de tarefas o escalonador. O módulo de gestão de tarefas é responsável pelo escalonamento bem como pelo despacho. O despacho altera o contexto da tarefa que está a ser executada pelo da próxima tarefa. Enquanto o escalonador é responsável por seleccionar qual a próxima tarefa a executar através da implementação de algoritmos de escalonamento.

2.1.6 Escalonamento de tempo real

Nos sistemas de tempo real a especificação dos requisitos inclui informação temporal na forma de prazos de execução [9]. Estes prazos de execução, são como já foi referido, de importância extrema. Para um correcto funcionamento do sistema é necessário garantir que as tarefas são executadas respeitando os seus requisitos temporais.

O processo de escalonamento consiste em distribuir eficientemente o acesso aos recursos e providenciar tempo de processamento às tarefas para cumprirem os seus requisitos temporais. O estudo do escalonamento nos sistemas de tempo real difere dos sistemas tradicionais e como tal não deve ser confundido. Nos sistemas tradicionais o escalonamento pretende apenas minimizar o tempo necessário para execução de todas as tarefas. Nos sistemas de tempo real o escalonamento pretende cumprir os requisitos temporais de cada tarefa.

Os algoritmos de escalonamento de tempo real podem dividir-se em dois conjuntos diferentes, os algoritmos estáticos e os algoritmos dinâmicos. Estes diferem no facto do sistema ter ou não conhecimento prévio sobre as características das tarefas.

Deste modo, os algoritmos estáticos planeiam o escalonamento antes mesmo da execução das tarefas. Basta para isso que o algoritmo tenha conhecimento da totalidade das tarefas e dos seus requisitos. Estes algoritmos podem-se dividir de duas formas:

- Algoritmos baseados em tabelas - Nestes algoritmos o escalonamento é planeado antes da execução e concretizado na construção de tabelas que determinam quando uma tarefa deve iniciar a sua execução. Estes algoritmos são utilizados quando as tarefas são periódicas ou podem ser transformadas em tal.

- Algoritmos baseados em prioridades - Nestes algoritmos o escalonamento é baseado em prioridades atribuídas antes da execução de modo a que a cada instante da execução seja definida qual a tarefa que é processada.

Por sua vez, os algoritmos dinâmicos apenas necessitam ter conhecimento das tarefas activas e dos seus requisitos uma vez que o escalonamento é feito em tempo de execução. Estes algoritmos podem-se dividir de duas formas:

- Algoritmos baseados em planeamento - Nestes algoritmos o escalonamento é adaptado aquando da chegada de uma nova tarefa de modo a que o novo escalonamento consiga cumprir simultaneamente os requisitos da nova tarefa e das tarefas anteriormente escalonadas.

- Algoritmos baseados no melhor esforço - Nestes algoritmos não é feito um novo escalonamento aquando da chegada de uma nova tarefa. Inclusivamente a tarefa é executada sem o sistema saber se vai conseguir cumprir os seus requisitos temporais. Este tipo de algoritmos é bastante utilizado nos sistemas de tempo real em que a prioridade das tarefas é determinada durante a execução, tendo em consideração as características das tarefas activas. A maior desvantagem deste tipo de algoritmos reside na sua falta de previsibilidade.

2.1.6.1 Earliest Deadline First

O Earliest Deadline First (EDF) é um algoritmo de escalonamento dinâmico por prioridades. Este algoritmo baseia-se na atribuição da maior prioridade à tarefa cujo prazo temporal está mais próximo do fim. Ou seja é dada prioridade à tarefa com menor tempo para conclusão dentro dos limites temporais definidos.

Este algoritmo é considerado óptimo uma vez que existindo um conjunto de tarefas possíveis de serem escalonadas (respeitando os limites temporais) o algoritmo consegue proceder ao seu escalonamento efectivo. No entanto, ele não tem capacidades de previsibilidade quando um escalonamento efectivo não é possível. Ou seja, não sendo cumpridos todos os requisitos temporais, o algoritmo não é capaz de identificar qual a tarefa ou quais as tarefas que não vão ser cumpridas de acordo com os requisitos.

2.2 Mecanismos de Verificação

Nesta dissertação entende-se por mecanismos de verificação aqueles que estão directamente associados aos métodos formais. Isto é, mecanismos baseados em algoritmos matemáticos, capazes de construir ou utilizar provas que determinem a veracidade de certas propriedades do sistema.

> The term formal methods refers to the use of mathematical modeling, calculation and prediction in the specification, design, analysis and assurance of computer systems and software. The reason it is called formal methods rather than mathematical modeling of software is to highlight the character of the mathematics involved. John Rushby em [38].

Há cerca de 14 anos foi estimado que qualquer pessoa era diariamente confrontada com cerca de 25 dispositivos computacionais [4], hoje estes valores são muito superiores e difíceis de quantificar. A nossa dependência por estes sistemas é cada vez maior. Um bom exemplo disso consiste na utilização de armas de impulso electromagnético para imobilizar veículos motorizados de nova geração. Já lá vão os tempos em que a polícia só conseguia imobilizar um veículo motorizado bloqueando

o caminho, furando os pneus, etc. Este exemplo torna-se ainda mais interessante considerando que parte da electrónica está dissimulada em mecanismos de controlo. Um carro da década de 90, exposto à utilização de uma arma destas, não fica imobilizado mas fica com todo o sistema eléctrico danificado. Um carro mais recente fica com a ECU danificada o que implica a imobilização do veículo.

Parte destes sistemas são construídos sem qualquer utilização de métodos formais. Prova disso reside na falta de fiabilidade dos telemóveis, quantas vezes não é necessário desligar e voltar a ligar estes dispositivos para que voltem a funcionar correctamente? A utilização de mecanismo de verificação contínua quase sempre associada a sistemas críticos. Como já foi referido, um sistema crítico pode ser entendido como aquele que, em caso de falha, pode por em risco vidas humanas ou investimentos onerosos. Fala-se geralmente deste tipo de sistemas em áreas como a aeronáutica, o aeroespacial, os caminhos-de-ferro, nas centrais nucleares, etc.

As mentalidades e exigências de mercado mudam e cada vez mais as empresas procuram oferecer produtos de qualidade certificada. Neste sentido a norma ISO-15408, designada por Common Criteria (CC), introduziu um modelo de avaliação da segurança e fiabilidade de sistemas. Este modelo divide-se numa métrica de 7 níveis de confiança designado por Evaluation Assurance Level (EAL). Para os níveis mais elevados (EAL 5 a 7) os métodos formais são obrigatórios.

2.2.1 Verificação Estática

Nesta dissertação vão ser abordados dois tipos de mecanismos de verificação. O primeiro consiste nos mecanismos de verificação estática e o segundo nos mecanismos de verificação de modelos.

O primeiro faz uso de processos de verificação em tempo de compilação que permitem libertar recursos na execução. Este facto torna este tipo de verificação bastante interessante para os sistemas de tempo real. Tudo o que puder ser feito em tempo de compilação, não vai ser feito em tempo de execução para que os requisitos temporais sejam mais facilmente atingíveis.

A verificação estática está correlacionada com a linguagem de programação. O próprio compilador, ou ferramentas construídas separadamente, analisam o código dos programas e fornecem um resultado formalmente provado. Neste tipo de verificação também se podem incluir técnicas como o Design By Contract (DBC) no qual se definem contractos sobre aquilo que o programa recebe e produz. Este tipo de verificação tem ainda a possibilidade de associar um selo de qualidade (certificado) ao código [21]. No entanto é preciso ter em atenção que nem sempre as técnicas de DBC são implementadas estaticamente. É possível que as mesmas sejam implementadas dinamicamente isto é, em tempo de execução, ou dinamicamente e estaticamente.

Nesta dissertação é feita a distinção entre as ferramentas industriais e as ferramentas académicas. Enquanto as ferramentas industriais, de verificação estática, providenciam algumas garantias, nomeadamente no suporte das mesmas, as ferramentas académicas servem essencialmente para investigação. Este último tipo de ferramentas também peca por algumas vezes não ter a devida documentação e muito raramente suporte.

2.2.1.1 Perfil Ravenscar

O estudo dos mecanismos de verificação começou com o estudo de um perfil de segurança. Em 1997 na 8th International Real-Time Ada Workshop (IRTAW) ficou definido o perfil Ravenscar [11][12][34]. Este perfil destina-se aos sistemas de tempo real críticos e consiste numa colecção de primitivas concorrentes e de restrições à linguagem Ada, no entanto não se limita a esta. O Ravenscar permite um desenvolvimento eficiente de aplicações capazes de serem verificadas quer na sua componente funcional como temporal.

Este perfil promove o suporte da concorrência ao nível da linguagem, permitindo verificação estática quer pelo compilador como por outras ferramentas. O perfil introduz um conjunto de restrições com objectivo de aumentar a eficiência e previsibilidade dos programas. Para isso proíbe a utilização de funcionalidades com um overhead elevado. Reduz a utilização de mecanismos da linguagem Ada que levam a situações não deterministas. Remove a utilização de funcionalidades de

fraco suporte na verificação formal bem como funcionalidades que inibem análises temporais efectivas.

Em termos concretos o perfil Ravenscar permite que a declaração de tarefas e objectos protegidos seja apenas feita ao nível da biblioteca. Não permite a alocação dinâmica quer de tarefas como de objectos protegidos. Só permite objectos protegidos com uma ou nenhuma entrada. Todas as tarefas são assumidas como não terminais. Não permite o uso de instruções *abort* ou ATC (Asynchronous Transfer of Control). Na construção das tarefas periódicas não pode ser utilizada a instrução *delay*, só sendo permitido o *delay until*. Não é permitida a utilização de estruturas de controle *Select*. Não é permitido o uso de prioridades dinâmicas. O pacote calendário não pode ser utilizado, sendo apenas permitido a utilização do pacote Real-Time, entre outras restrições.

2.2.1.2 SPARK (industrial)

O SPARK [3][14][13] é uma linguagem de programação desenvolvida pela Praxis com base no Ada. Desde sempre que esta linguagem segue uma abordagem correcta por construção (*Correctness by Construction*) recorrendo a anotações. Estas anotações permitem uma extensa análise estática incluindo análises de fluxo de controlo, de dados e informação.

Em 2003 o SPARK adoptou o perfil Ravenscar para permitir o desenvolvimento de aplicações concorrentes e fiáveis. No entanto num modelo de desenvolvimento tradicional o Spark não substitui nem dispensa o processo de modelação [10].

As ferramentas envolvidas na verificação do SPARK são o SPARK Examiner, o SPADE Proof Checker e o Automatic Simplifier. Na prática o processo de verificação é iniciado com o Examiner. Quando o Examiner enceta a sua execução é feita uma análise lexical e sintáctica de cada unidade de compilação. No segundo passo é feita uma análise semântica de cada unidade de compilação. No terceiro passo é feita uma análise de fluxo de controlo que verifica se o código está bem estruturado. No quarto passo é feita a análise de fluxo de dados e informação para controlar a estabilidade dos ciclos, a existência de variáveis declaradas mas não utilizadas, a

existência de instruções sem efeito, a utilização de variáveis não inicializadas e a consistência entre o fluxo de informação esperado e o actual.

No quinto passo é feita a detecção dos erros que podem ocorrer em tempo de execução (divisão por zero, índice fora do tamanho, etc.) e são geradas obrigações de prova sobre esses possíveis erros. No sexto passo pretende-se provar que as obrigações de prova, anteriormente geradas, nunca vão ocorrer. Neste passo algumas das obrigações de prova são descartadas com uma ferramenta automática, o Simplifier, no entanto as restantes deverão ser provadas manualmente recorrendo ao SPADE Proof Checker.

No sétimo e último passo é feita uma análise do pior caso de execução(WCET) bem como das necessidades de memória do programa. Isto é feito combinando o gráfo de control de fluxo para cada subprograma e o WCET de cada objecto. O resultado imediato desta combinação permite a obtenção da execução que origina o WCET do programa.

De seguida são apresentadas algumas das anotações em forma de exemplo.

- *global* Torna visível uma variável global com o modo especificado.

```
1  procedure Control;
2    # global in Sensor.State;
3    # out Valve.State;
```

- *derives* Especifica o fluxo de informação entre os parâmetros e variáveis globais de um procedimento.

```
1  procedure Flt_Integrate(Fault : in Boolean;
2  Trip : in out Boolean;
3  Counter : in out Integer)
4    # derives Trip from *, Fault, Counter &
5    # Counter from *, Fault;
```

- *pre* Especifica um requisito essencial para o correcto funcionamento do programa.

- *post* Especifica o resultado garantido após uma correcta execução.

- *assert* Utilizado para especificar condições que devem ser sempre verdadeiras.

```
1  procedure Div(M, N: in Integer; Q, R: out Integer)
2    # derives Q, R from M, N;
3    # pre (M >= 0) and (N > 0);
4    # post (M = Q * N + R) and (R < N) and (R >= 0);
5  is
6  begin
7   Q := 0; R := M;
8   loop
9       # assert (M = Q * N + R) and (R >= 0);
```

2.2.1.3 Giotto (académico)

O Giotto [23][26][24][28] é uma linguagem de programação para sistemas embebidos com tarefas periódicas. A principal característica do Giotto reside na abstracção que é feita ao nível da arquitectura. A componente lógica (funcional e temporal) divide-se da componente física na qual o código vai ser executado. Esta particularidade torna os programas completamente independentes da plataforma, necessitando apenas da existência de uma implementação do ambiente de execução para o sistema de destino.

Num programa Giotto as tarefas representam a funcionalidade de base da linguagem. Elas são executadas em intervalos de tempos regulares (tarefas periódicas). Estas tarefas são associadas a funções e possuem um número arbitrário de portos de entrada e saída. A função é que dita a funcionalidade da tarefa, no entanto ela é implementada fora do Giotto recorrendo (teoricamente) a uma qualquer linguagem. Apesar da abstracção feita, para a implementação de um programa Giotto numa dada máquina o compilador necessita conhecer o WCET de todas as funções.

Num programa Giotto as tarefas agrupam-se em modos que são executados ciclicamente. De notar que, um programa só pode estar num modo de cada vez mas os modos podem conter instruções para troca de modo.

Os portos utilizados nas tarefas representam variáveis tipadas num espaço partilhado. Esse espaço pode ser de memória partilhada ou de outro tipo. Cada porto é persistente no sentido que mantém o seu valor ao longo do tempo até ser actualizado. Por sua vez os drivers são os elementos que permitem a comunicação entre os portos das tarefas.

Outra característica fundamental no Giotto são as anotações. Enquanto o código puro do Giotto é independente da plataforma, o mesmo pode ser refinado com directivas de compilação na forma de anotações dependentes da plataforma. Essas directivas podem mapear uma determinada tarefa para uma unidade de processamento, escalonar uma tarefa num intervalo de tempo ou ainda escalonar um evento de comunicação entre tarefas num intervalo de tempo. Todavia estas anotações não influenciam as funcionalidades do programa, apenas introduzem indicações ao compilador para implementação numa dada plataforma.

Existem três níveis de anotações no Giotto, o primeiro é designado por Giotto-H (H de Hardware). Neste é especificado o conjunto de unidades de processamento disponíveis, as redes e informações sobre os WCET de cada tarefa bem como outras informações sobre os tempos de comunicação. O segundo nível é designado por Giotto-HM (M de Map) e neste é acrescentada informação sobre o mapeamento entre as tarefas e as unidades de processamento. O terceiro e último nível é designado por Giotto-HMS (S de Scheduling) e especifica informações sobre o escalonamento de cada unidade de processamento.

Estas anotações servem de directivas que indicam como os programas Giotto se devem comportar numa dada plataforma. Apesar de a linguagem possuir boas propriedades não existe nenhuma garantia de escalonamento dos programas. No entanto a dada altura foi apresentado um método que permite verificação de programas Giotto. Esse método consiste num esquema de tradução manual da linguagem para um modelo de redes de autómatos temporizados [37] (mais precisamente para a ferramenta de verificação de modelos UPPAAL). Este esquema de tradução divide-se em duas partes. Numa primeira é considerado apenas o código puro do Giotto (sem as anotações) e numa segunda parte são consideradas as anotações. Esta verificação permite na primeira parte determinar se um programa Giotto cumpre ou não com determinados requisitos temporais e na segunda se é ou não escalonável.

Infelizmente poucos resultados são apresentados e o esquema de tradução nunca chegou a ser automatizado. Apesar de tudo, a dissertação inspira-se nesta ideia de construção de um modelo a partir de uma dada linguagem.

2.2.1.4 Schedule Carrying Code (académico)

Em termos práticos, do processo de compilação do Giotto obtêm-se o designado Embedded Code (E-Code). Este é executado numa máquina virtual, a Embedded Machine (E-Machine). O E-Code é considerado *time-safe* se for encontrado um escalonamento possível para a plataforma na qual se quer implementar a aplicação.

Deste facto surge o Schedule Carrying Code (SCC) [27][15][25] que introduz um novo conceito à linguagem Giotto. Mais precisamente o Scheduling Code (S-Code), i.e. uma linguagem máquina executável que permite especificar o escalonamento (despacho). Ao contrário do que acontece no Giotto original, neste modelo o E-Code é independente da plataforma e este novo S-Code é dependente da plataforma.

O S-Code só é gerado se for encontrado um escalonamento que considere a execução do programa como *time-safe*. Este código pode ser visto como uma prova de escalonamento para uma determinada plataforma à semelhança do Proof Carrying Code (PCC). O S-Code consiste então num conjunto de instruções que determinam qual a tarefa a ser executada até que um determinado evento ocorra. Esse evento pode estar relacionado com o relógio, com uma tarefa ou até mesmo com um sensor. Outra particularidade interessante é que o S-Code pode ser gerado consoante qualquer estratégia de escalonamento, seja em tempo de compilação, execução ou parcialmente em compilação e parcialmente em execução.

2.2.1.5 xGiotto (académico)

Na continuação do processo de melhoramento do Giotto surgiu o eXtended Giotto (xGiotto) [19][39], como uma extensão da linguagem anterior. Esta nova linguagem liberta-se da dependência das tarefas periódicas e passa a ser auto-contida integrando o código funcional das tarefas.

O xGiotto realiza adicionalmente várias análises de integridade. A primeira visa rejeitar programas que contenham possíveis *race conditions*, i.e. quando duas tarefas escrevem no mesmo porto e são terminadas pelo mesmo evento. Isto permite que os programas continuem deterministas. Numa segunda análise é feita uma

previsão sobre a capacidade de memória necessária. Na terceira análise é verificado o escalonamento do programa para uma determinada plataforma. Esta última análise requer que os WCET das tarefas esteja especificado segundo a plataforma de destino.

No entanto, tal como é referido em [20], esta linguagem rapidamente torna o processo de determinação do escalonamento intratável. Esta situação torna a linguagem muito menos atractiva apesar de não se limitar a tarefas periódicas.

2.2.1.6 TDL e HTL

O Giotto ainda deu origem a outras linguagens como o Timing Definition Language (TDL) [35][43] e o HTL [20][29][17][18]. O TDL foi desenvolvido no projecto MoDECS e disponibiliza uma sintaxe mais conveniente bem como um conjunto de ferramentas adicionais. Este representa o primeiro passo para tentar tornar estas linguagens, derivadas do Giotto, menos académicas e mais industriais (`http://www.preetec.com`).

O TDL ainda acrescenta um novo conceito relativamente ao Giotto. Trata-se da arquitectura baseada em componentes (módulos). Isto proporciona uma maior flexibilidade na construção de programas uma vez que estes módulos são independentes uns dos outros.

Por sua vez, o HTL é o resultado mais recente do Giotto. Esta nova linguagem é reconhecida como uma linguagem de coordenação. Ao contrário do TDL e do xGiotto, o HTL não é auto-contido uma vez que tal, como o Giotto, não inclui na sua especificação a componente funcional das tarefas.

Esta linguagem possui no entanto características bastante interessante como a noção de precedência de tarefas, refinamentos hierárquicos, módulos, etc.. Como vantagem sobre o xGiotto, o HTL consegue simplificar o processo de prova de *time-safety*, muito devido à sua estrutura hierárquica. Num programa HTL basta garantir que o modelo é *time-safe* ao nível mais alto da hierarquia. A linguagem é descrita com maiores detalhes no capítulo 4.

2.2.2 Verificação de Modelos

Num processo de desenvolvimento com uso de métodos formais é comum serem utilizados mais do que um tipo de mecanismos de verificação. Mecanismos distintos tendem a complementar-se mas as ferramentas que empregam mecanismos idênticos também podem verificar propriedades distintas se usarem lógicas/algoritmos distintos. Sucintamente a verificação de modelos (Model Checking) [4][7] consiste numa formulação matemática representativa do sistema (o modelo), num formalismo de especificação de propriedades e num conjunto de algoritmos capazes de explorar sistematicamente todos os estados do modelo. Os algoritmos são a parte nuclear da verificação, são estes os responsáveis por indicar se uma dada propriedade é verificada ou não. Se a propriedade não for verificada, tradicionalmente, a ferramenta produz um contra-exemplo, i.e. um exemplo de execução no qual é possível verificar a falha dessa propriedade.

Em todas as linguagens anteriormente referidas muito do que é feito em termos de verificação temporal consiste numa análise de escalonamento (*Scheduling Analysis*). Mas pouco ou nada é feito em termos de análise temporal (*Timing Analysis*). Poucas são as situações em que é possível estabelecer comparações sobre os períodos ou relações temporais entre a execução de tarefas. Na verificação estática, vista anteriormente, o importante é determinar se o programa é ou não escalonável. Nada é referido sobre a possibilidade das tarefas não serem executadas pela sequência correcta. Desde que elas sejam escalonáveis tudo está bem do ponto de vista da análise de escalonamento.

Apesar de não ser assim tão recorrente, é possível encontrar estudos sobre a utilização da verificação de modelos juntamente com as ferramentas anteriormente referidas [37][10][34]. Um dos grandes problemas da utilização da verificação de modelos reside no facto de ser necessário traduzir o modelo pelo código ou o código pelo modelo. Este processo é penoso, repetitivo e pode adulterar a verificação se não for garantido que o modelo corresponde precisamente ao sistema.

> Any verification using model-based techniques is only as good as the model of the system. Em [4]

Um modo de tornar a utilização deste mecanismo de verificação mais apelativo consiste em automatizar o processo de tradução do modelo para a linguagem na qual o sistema é implementado, ou em automatizar a tradução da linguagem para o modelo. Uma vez que o modelo é tendencialmente mais abstracto do que a linguagem, este processo de tradução dificilmente se automatiza na totalidade, no entanto o contrário já pode ser feito de forma mais eficiente. Da linguagem facilmente se consegue abstrair a informação desejada para a construção de um modelo e consequentemente torna-se possível proceder à verificação do mesmo.

Nos sistemas de tempo real, a verificação de modelos baseia-se bastante nas lógicas temporais. De notar que estas lógicas dividem-se em diversas famílias com expressividades distintas e como tal nem todas as ferramentas de verificação de modelos utilizam a mesma lógica temporal.

Genericamente as lógicas temporais permitem verificar as seguintes classes de propriedades:

- Segurança (*Safety*) - Em certas condições, um determinado acontecimento não vai ocorrer.
- Acessibilidade (*Reachability*) - Uma situação particular pode ser atingida.
- Equidade (*Fairness*) - Em certas condições, um determinado evento vai ocorrer (ou não vai ocorrer) uma infinidade de vezes.
- Vivacidade (*Liveness*) - Em certas condições, um determinado evento vai acabar por acontecer.
- Ausência de *DeadLock* - O sistema não pode chegar a uma situação a partir da qual nenhum progresso é possível.

A compreensão deste tipo de classes de propriedades é determinante para uma correcta metodologia de modelação e especificação das propriedades do sistema. Se o tipo de propriedades que se querem verificar não corresponde a estas classes, então a verificação de modelos temporizados não é adequada. De notar ainda que neste tipo de verificação de modelos, as propriedades de acessibilidade e de segurança são em geral mais importantes porque são mais fáceis de verificar. Deste modo

é conveniente definir boas estratégias de modelação (incluindo transformações no modelo, se necessário) para que as propriedades visadas possam ser verificadas.

No estudo feito, às ferramentas de verificação de modelos [22][7][4][31], sumariamente destacam-se as seguintes:

- RT-Spin - Extensão da linguagem promela com noções temporais e Lógica LTL com asserções. 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.

2.3 Conclusão

Neste capítulo apresentaram-se os mecanismos de verificação estudados no âmbito da dissertação. Apesar de não terem sido estudadas todas as ferramentas, o estudo foi suficiente para ficar com uma percepção geral do tipo de verificação desejável nos sistemas de tempo real. Um dos aspectos que tem dominado a investigação na matéria concentra-se muito na questão do escalonamento das tarefas. No entanto muito ainda pode e deve ser feito relativamente ao cumprimento total dos requisitos temporais e funcionais.

Os mecanismos actuais não se focam o suficiente sobre questões como: 'A tarefa X nunca pode ocorrer simultaneamente que a tarefa Y', ou 'Se a tarefa X ocorrer, passado T unidades de tempo a tarefa Y tem de ocorrer', ou 'A tarefa X alguma vez vai ser atingida'. Este aspecto faz com que um sistema até possa ser escalonado mas funcionalmente o resultado não é o desejado. Se apenas forem verificadas propriedades deste tipo também se pode perder a verificação sobre o escalonamento, pelo que o ideal é combinar ambas.

De modo a tirar benefício das duas situações, esta dissertação apresenta uma solução onde ambas são conjugadas. Para isso foi necessário escolher uma linguagem que permitisse alguma verificação estática. Infelizmente cada uma destas linguagens possui as suas próprias restrições. O SPARK e as linguagens que respeitam o perfil Ravenscar são mais completos em termos de verificação, mas inibem a utilização de diversas técnicas de programação. Desconhece-se a existência de uma prova que indique que estas linguagens não permitam fazer tudo o que se faz com as suas versões não limitadas. Todavia reconhece-se que as restrições dificultam significativamente o processo de desenvolvimento. Para além disto, o facto do HTL funcionar como linguagem de coordenação, possuir hierarquias, permitir precedência de tarefas e garantir o escalonamento despertou interesse suficiente para descartar linguagens baseadas no perfil Ravenscar. Tanto que num futuro, mais ou menos distante, é aceitável considerar que o HTL permita a especificação funcional em ADA ou numa linguagem DBC.

Estando a linguagem nuclear identificada, restava escolher a ferramenta de verificação de modelos. Sendo o objectivo da solução apresentada no capítulo 5 complementar a verificação dos requisitos temporais, a escolha preferencialmente recaia numa ferramenta baseada na Lógica TCTL. Das ferramentas identificadas verificou-se que aquelas que empregam este tipo de lógica adaptam-se melhor aos sistemas de tempo real. Entre o Kronos e o Uppaal a escolha facilmente tendeu para o Uppaal, quer pela quantidade de informação existente sobre a ferramenta como pelas qualidades reportadas na literatura.

Nos dois próximos capítulos são apresentados alguns conceitos relacionados com a ferramenta Uppaal, bem como a linguagem HTL. Nos capítulos seguintes a estes, é apresentado o tradutor de HTL para Uppaal e os resultados obtidos na aplicação do tradutor ao caso de estudo Tree Tank System (3TS).

Capítulo 3

UPPAAL

Este capítulo baseia-se nos livros [4][31][7] e nos documentos [6][5][2][1][30]. Introduz-se neste capítulo alguns conceitos sobre a ferramenta de verificação de modelos Uppaal. No âmbito da dissertação foi produzido um documento descritivo da componente prática da ferramenta que pode ser consultado no URL `http://floyd.di.ubi.pt/release/jcarvalho/TutorialPT_Uppaal.pdf`. Esse documento complementa, de alguma forma, a informação exposta na dissertação.

Apesar da verificação de modelos ser vista, nesta dissertação, mais na vertente dos autómatos temporizados e da lógica temporal, não se pode ignorar a existência de outras representações. Nomeadamente no que respeita aos modelos operacionais é possível identificar diversos sistemas de transição como, os autómatos de estados finitos, os autómatos de Büchi, as estruturas de Kripke, etc. No que respeita às lógicas subjacentes tem-se, as lógicas modais, temporais, dinâmicas, etc..

O processo de verificação assume a existência de, pelo menos, um sistema de transição que especifica o sistema, uma lógica particular e uma fórmulas desta lógica que estabelecem o que se pretende verificar. Este processo efectua constatações baseadas em percursos adequados do sistema de transição (verificação de modelos directa) ou de uma representação compacta (verificação de modelos simbólica).

Sumariamente, a verificação de modelos baseada em autómatos temporizados (como no caso do Uppaal) recorre-se de uma versão refinada dos autómatos de

Büchi. Um autómato de Büchi consiste num autómato de estados finitos estendido de forma a aceitar entradas infinitas. Este sustenta-se na noção de aceitação, i.e. execução potencialmente infinita que passa infinitamente pelo estado final. Os autómatos temporizados são então autómatos de Büchi com a noção de relógios locais. Com este formalismo é possível discursar sobre a sequência das operações durante as execuções possíveis (concorrência e distribuição), mas também sobre os prazos esperados para cada uma delas (tempo real). No caso do Uppaal este discurso é produzido por um subconjunto da linguagem Timed Computation Tree Logic (TCTL).

3.1 Sistemas de Autómatos

Em forma de revisão, assuma-se a existência de um conjunto $Prop = \{P_1, \ldots\}$ de proposições elementares.

3.1.1 Autómato

Um autómato é o $5-tuplo$ $\mathcal{A} = (Q, E, T, Q_0, l)$ onde, Q é um conjunto de estados, E um conjunto de etiquetas (label) de transições, $T \subseteq Q \times E \times Q$ o conjunto das transições e $Q_0 \subseteq Q$ o conjunto de estados iniciais. Considera-se aqui (sem perda de generalidade) somente o caso em que há um só estado inicial. q_0 é o estado inicial do autómato; E $l : Q \to \mathcal{P}_F(Prop)$ é uma aplicação que associa a cada estado q de Q o conjunto finito de propriedades que este verifica.

Apesar de ser pouco relevante para esta dissertação não se deve esquecer que, caso se pretenda um autómato com *output*, considera-se uma sexta componente O que é uma aplicação que atribui a cada estado o *output* que este produz (fala-se neste caso de autómatos de Moore).

3.1.2 Caminho

Considerando a existência de um autómato $\mathcal{A}$, chama-se *caminho* de $\mathcal{A}$ a uma sequência (possivelmente infinita) σ de transições (q, e, r) encadeadas (i.e. para toda a transição (q, e, r) da sequência, a transição seguinte é da forma (r, e', s)).

Ao *comprimento* de um caminho σ nota-se $|\sigma|$ (número de transições do caminho). $|\sigma| \in \mathbb{N} \cup \{\omega\}$ (o caminho pode ser infinito). Para um caminho σ, um índice $i < |\sigma|$, $\sigma(i)$ refere-se ao estado q da i-ésima transição (q, e, r) de σ.

3.1.3 Execução

Chama-se *execução parcial* de $\mathcal{A}$ ao caminho começado pelo estado inicial q_0. Chama-se *execução completa* ao caminho parcial *máximo* (aquele que não pode ser mais estendido). Chama-se *execução com bloqueio* quando o caminho considerado é infinito ou quando o caminho acaba num estado a partir do qual não há transições possíveis. A *árvore de execução* é a representação arborescente de todas as execuções possíveis a partir dum determinado estado (em geral q_0). Um estado é designado de *acessível* (ou atingível) se aparece na árvore de execução com raiz q_0. Ou seja, se existir pelo menos uma execução que passe por ele.

3.1.4 Extensões de Autómatos

Para além destas características é possível realizar algumas extensões aos autómatos de base. Por exemplo, pode-se juntar variáveis de estado na modelação por autómatos. Nesta situação as transições são estendidas com a capacidade de testar as variáveis (guardas) e, caso o teste permita a selecção da transição, de alterar o valor das variáveis. O autómato com essa funcionalidade passa a considerar um conjunto de variáveis (tipificadas) e uma transição passa a ser um 5-tuplo (q, g, l, a, r) onde: q é o estado de partida; r é o estado de chegada; g é uma guarda sobre uma variável do autómato; l é uma etiqueta; e o conjunto $a = \{x_1 := e_1 \ldots\}$ das actualizações de variáveis (atribuições) que são realizadas.

Uma transição (q, g, l, a, r) pode ser seleccionada quando o estado q é o estado de controlo, a guarda g é verificada e a entrada do autómato começa pela etiqueta l da transição. Como consequencia da selecção desta transição a actualização a é accionada e o estado de controlo passa do estado q para o estado r.

Apesar destes autómatos estendidos serem úteis a verificação de modelos não se

faz ao nível da extensão. Para que se aplique algum algoritmo de verificação torna-se necessário passar da extensão para uma representação clássica. Este processo de tradução resulta no *autómato desdobrado*, ou no *sistema de transição associado*. É possível, sob certas condições, do autómato A (com variáveis) obter o autómato finito equivalente sem variáveis. Essas condições são: O conjunto dos valores possíveis para as variáveis ser finito; Se o conjunto dos valores possíveis para as variáveis for infinito então deve-se descobrir uma partição finita pertinente (pares/impares, positivo/negativo...); etc.

3.1.5 Redes de Autómatos / Produto Sincronizado

Considerando uma família de n autómatos $A_i = (Q_i, E_i, q_{0,i}, l_i)$ (com $i = 1, \ldots, n$). Considerando a etiqueta "_" que corresponde à acção "nada por fazer".

Define-se por *conjunto de sincronizações sync*, um conjunto tal que $sync \subseteq \Pi_{1 \leq i < n}(E_i \cup \{\_\})$. Define-se por produto sincronizado da família de autómatos A_i (Notação: $A_1 || A_2 || \cdots || A_n$), o autómato $A = (Q, E, T, q_0, l)$ tal que: $Q = Q_i \times \cdots \times Q_n$; $E = \Pi_{1 \leq i < n}(E_i \cup \{\_\})$;

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

$q_0 = (q_{0,1}, \cdots, q_{0,n})$; $l((q_1, \cdots, q_n)) = \bigcup_{1 \leq i < n} l_i(q_i)$. Quando $sync = \Pi_{1 \leq i < n}(E_i \cup \{\_\})$, obtém-se o produto cartesiano dos autómatos considerados (não há restrições sobre o funcionamento e a cooperação). As sincronizações são essencialmente relações, entre autómatos, que permitem limitar/definir o seu comportamento conjunto.

3.1.6 Grafo de Acessibilidade

Chama-se *estado inacessível* ao estado pelo qual nunca poderá passar um caminho oriundo do estado inicial. Sabe-se que o produto (sincronizado) de autómatos gera autómatos com estados que não são acessíveis. À representação cujo estados não acessíveis são removidos designa-se por grafo de acessibilidade. Além da comodidade em lidar com autómatos menores, a noção de acessibilidade permite expressar algumas propriedades interessantes sobre acontecimentos desejados ou

não desejados. Como por exemplo um evento de alarme ser accionado.

3.1.7 Sincronizações

As sincronizações podem ser feitas de dois modos distintos, por mensagens ou por variáveis partilhadas. No primeiro caso uma sincronização corresponde virtualmente ao envio e recepção de mensagens aquando de uma transição. A sincronização só contempla transições onde qualquer emissão de mensagem (etiqueta semelhante a !*mensagem*) é acompanhada pela recepção da própria (etiqueta semelhante a ?*mensagem*). Embora seja possível evitar o uso das variáveis, pode ser prático beneficiar do seu uso explícito. No caso das sincronizações feitas por variáveis estas podem ser utilizadas sobre a forma de *flags* ou acompanhadas por sincronizações de mensagens. No primeiro caso elas apenas funcionam como ponto de encontro, no segundo caso o ponto de encontro é feito pela sincronização de mensagem e a variável partilhada serve de portadora da mensagem (com um conteúdo específico).

3.2 Autómatos Temporizados

Os autómatos clássicos permitem raciocinar sobre a ordem no tempo dos eventos que nele ocorrem. Mas não permitem raciocinar sobre a duração dos eventos ou sobre os intervalos de tempo que ocorrem entre eventos. Para tal, Rajeev Alur e David L. Dill definiram em 1994 uma extensão dos autómatos que cobre os aspectos quantitativos do tempo. Esta extensão ficou conhecida por autómato temporizado.

Um autómato temporizado é composto por um autómato finito clássico e variáveis de relógio que permitem quantificar a progressão do tempo. As transições deste tipo de autómatos são consideradas instantâneas, sendo que o tempo evolui nos estados. Para se poder discursar sobre a duração de um evento é necessário considerar um estado que modela o início do evento e outro o fim. Existindo uma sequência de vários estados, pode não ser necessário que cada um deles apenas represente o fim ou o inicio de um evento. Se for considerado que não existe passagem de tempo entre eventos, define-se um determinado estado como sendo o fim do evento anterior e simultaneamente o início do evento actual. Se existir a

necessidade de efectuar alguma transição entre o fim de um evento e o inicio de outro, então este tipo de simplificação do modelo não é possível.

3.2.1 Relógios e Transições

Os relógios são variáveis reais de $\mathbb{R}^+$ com valor inicial 0. Nos autómatos temporizados todos eles evoluem à mesma velocidade. Já nos autómatos híbridos o mesmo não acontece, i.e. neste tipo de autómatos é permitido que os relógios evoluam a ritmos diferentes. As transições dos autómatos temporizados são constituídas por uma guarda, uma etiqueta ou acção e acções de reinicialização de variáveis de relógio.

Devido às reinicializações possíveis, os relógios medem prazos e não o tempo. O sistema modelado por este tipo de autómatos funciona considerando a existência de um relógio global que permite a sincronização com todos os relógios utilizados.

3.2.2 Configuração e Execução

Seja μ uma aplicação que associa a cada relógio o seu valor (designada *valoração*) e q o estado de controlo activo. Designa-se por configuração (q, μ) ao par do estado de controlo activo e do valor de cada relógio nesse instante. Ao efectuar-se uma alteração de configuração tem-se um passo da execução do autómato (uma transição).

Para que o sistema mude de configuração é preciso que decorra um prazo $d \in \mathbb{R}^+$, designada *transição de prazo* na qual todos os relógios são actualizados de acordo com o prazo d. Ou que o autómato seja activado, i.e seja executada uma transição do autómato, designada *transição discreta* ou *transição de acção*, na qual os relógios por reinicializar são reinicializados e o valor dos restantes é mantido.

A execução de um autómato temporizado também pode ser designada por *trajectória*. Esta, pode ser vista como uma aplicação ρ dos reais positivos para as configurações. $\rho(0)$ representa a configuração no instante global 0 e $\rho(t)$ representa a configuração no instante global t. Exemplo: $(init, 0) \rightarrow (init, 8.8) \stackrel{?req}{\rightarrow} (process, 0) \rightarrow$

$(process, 9.4) \xrightarrow{?req} (process, 0) \rightarrow (process, 3.4) \xrightarrow{?req} (alarm, 0)$, $\rho(9.9) = (process, 1.1)$.

Pode-se dizer que as execuções baseiam-se numa perspectiva em que o tempo é global e externo ao sistema enquanto as valorações μ numa perspectiva local e interna.

3.2.3 Redes e Sincronizações

Uma Rede de Autómatos Temporizados (RAT) representa o modelo completo do sistema. Como tal, trata-se de uma composição adequada de diversos autómatos temporizados (com as devidas sincronizações). À execução da RAT corresponde uma sequência, geralmente infinita, de configurações. Numa RAT a configuração (q, μ) representa o registo do estado de controlo de cada um dos autómatos (vector de estados q) bem como dos valores de cada um dos relógios (μ).

As execuções da RAT decorrem como as execuções de um autómato único, com a consideração de que ao ser activada uma acção aplicável (uma transição ou várias transições sincronizadas) apenas o controlo dos estados ligados a essa acção são alterados. Assim todos os autómatos são executados em paralelo e à mesma velocidade.

3.2.4 Invariantes

Os estados podem conter condições (sobre os relógios) designadas por invariantes. O invariante acrescenta a noção de obrigação de progresso, ao contrário das guardas nas transições que são uma possibilidade de progresso. Um invariante I num estado q significa que o controlo só pode estar em q enquanto I for verificado. Na existência de invariantes, uma configuração é válida se os invariantes dos estados de controlo são todos verificados. Assim o tempo só pode decorrer numa determinada configuração enquanto esta permanecer válida. Quando esta deixa de o ser, uma transição elegível deve ser accionada para evitar o designado *livelock* (deadlock temporizado), situação em que nenhuma configuração válida é atingível.

3.2.5 Estados e Sincronizações Urgentes

Nalgumas situações pode ser desejável que certas acções sejam tomadas sem que o tempo decorra (i.e. de forma urgente). Numa sincronização urgente não existe passagem de tempo, isto implica que a transição respectiva não contemple guardas sobre os relógios. Num estado urgente não existe passagem de tempo enquanto esse estado for de controlo. No Uppaal ainda existe a noção de estado *committed*, i.e. estado que é urgente e para além disso obriga imediatamente a execução de uma transição que parte dele. Este tipo de estados permite modelar sequências de acções atómicas.

3.2.6 Autómato de teste

Dada uma propriedade ϕ por verificar sobre uma RAT D, por vezes é conveniente transformar a verificação de ϕ num problema de acessibilidade. Isto pode ser feito recorrendo ao autómato de teste, i.e. ao autómato T_ϕ cujo objectivo é atingir um estado q, interagindo com D, caso a propriedade ϕ seja violada. Afirmar que D respeita ϕ significa verificar que q não é acessível [8]. Mas também pode ser feito acrescentando ao modelo, relógios, variáveis, guardas, etc., para que a propriedade ϕ possa ser simplificada [33].

3.3 Lógica Temporal

Nas duas secções anteriores foram vistos fundamentos da teoria dos autómatos clássicos e dos autómatos temporizados. Seja, fundamentos sobre a modelação do sistema ao qual se pretende aplicar a verificação de modelos. Falta agora abordar os fundamentos relacionados com a especificação das propriedades a verificar. Tal como referido anteriormente, essa especificação é feita por intermédio de uma lógica, no caso do Uppaal da TCTL. A TCTL, por sua vez, é uma extensão da Computation Tree Logic (CTL) e esta um subconjunto da Full Computation Tree Logic (CTL*).

Muito genericamente, a lógica temporal é um formalismo de especificação adequado a sistemas que lidam com propriedades nas quais existe uma relação

temporal, mais precisamente noções de ordenação no tempo. Este tipo de lógica baseia-se em noções matemáticas próximas das construções linguísticas: "sempre", "enquanto", "nunca", "antes", "depois", etc.

Todavia, a lógica temporal não especifica propriedades sobre o tempo de forma explícita, mas sim tendo em consideração sequências de acções (vistas como estados), i.e. os possíveis caminhos. As propriedades onde o tempo é absoluto (unidades de tempo) não são, por defeito, apropriadas à especificação em lógica temporal. O que acontece na prática é que existem métodos capazes de contornar esta situação, recorrendo por exemplo à utilização das variáveis de relógio.

Tal como já tinha sido sugerido, as lógicas temporais diferenciam-se pela forma como representam o tempo. No modelo de tempo linear, Linear Temporal Logic (LTL), o comportamento do sistema é representado por um conjunto de execuções infinitas. Cada uma das execuções é independente, sem a existência de ramificações entre elas. No modelo de tempo ramificado, CTL, o comportamento do sistema é representado por uma árvore de computação de profundidade. Devido a estas especificidades cada modelo possui capacidades de expressividade diferentes.

3.3.1 Full Computation Tree Logic

3.3.1.1 Sintaxe (formato BNF)

$$
\begin{array}{lll}
\phi, \psi ::= & P_1 \mid P_2 \mid \cdots & \text{(proposições atómicas)} \\
& \neg\phi \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (\phi \implies \psi) \mid \cdots & \text{(conectivas lógicas clássicas)} \\
& X\,\phi \mid F\,\phi \mid G\,\phi \mid (\phi\ U\ \psi) \mid & \text{(conectivas temporais de estado)} \\
& E\,\phi \mid A\,\phi & \text{(conectivas temporais de caminho)}
\end{array}
$$

3.3.1.2 LTL vs CTL

Considerando o conjunto das execuções de um autómato. Este pode-se representar como um conjunto de sequências (LTL) ou como uma árvore (CTL). Por sua vez a CTL* reúne o poder expressivo da LTL e da CTL.

O tempo do ponto de vista linear apenas permite a utilização de conectivas de estado, tais como: $X\ \phi$: Na execução considerada, o estado imediatamente a seguir (*neXt*) verifica ϕ; $F\ \phi$: Na execução considerada, um estado qualquer a seguir (*Finally*) verifica ϕ; $G\ \phi$: Na execução considerada, todos os estados seguintes (*Globally*) verificam ϕ; $\phi\ U\ \psi$: Na execução considerada, até (*Until*) se chegar a um estado em que ψ se verifica, verifica-se ϕ.

O tempo visto como uma árvore permite também a utilização de conectivas de caminho, tais como: $A\ \phi$: qualquer execução (*All*) que parte do estado corrente satisfaz a fórmula ϕ; $E\ \phi$: existe (*Exist*) uma execução que parte do estado corrente e que satisfaz a fórmula ϕ.

Nesta abordagem é possível realizar combinações entre estes dois tipos de conectivas, no entanto com algumas limitações. Nomeadamente, as conectivas X, F e U devem estar sempre no âmbito (*scope*) imediato de uma conectiva de caminho: EX, AX, EF, AF, $E\_U\_$, $A\_U\_$. A validade da fórmula a verificar depende exclusivamente do estado actual (e da exploração de todos os caminhos que saem dele).

3.3.1.3 Estruturas de Kripke

Uma estrutura de Kripke é um tuplo (S, i, R, L) onde S é um conjunto finito de estados. $i \in S$ é o estado inicial. $R \subseteq S \times S$ é uma relação de transição. No caso presente procura-se relações totais (que verificam, $\forall s \in S, \exists s' \in S, (s, s') \in R$) para garantir a ausência de deadlock. $L : S \rightarrow \mathcal{P}(P)$ é uma função que etiqueta cada estado com o conjunto de fórmulas atómicas válidas nesse estado.

Estas estruturas são na realidade um modelo da CTL* e definem formalmente o que se entende por ser verdade em CTL*. São estas estruturas que permitem estabelecer a relação entre a lógica e os autómatos, uma vez que elas são um mero caso particular de autómato onde as etiquetas são ignoradas. As considerações de semântica permitem estabelecer as regras para afirmar quando um autómato respeita ou não uma fórmula (propriedade) CTL*. A verificação de modelos é então o processo algorítmico que permite estabelecer esta afirmação (ou não).

Sendo $\mathcal{A}$ uma estrutura de Kripke (autómato), σ uma execução, $i \in \mathbb{N} \cup \{\omega\}$ um momento na execução e $\sigma(i)$, o estado correspondente (o i-ésimo elemento da execução). Notação: $\mathcal{A}, \sigma, i \models \phi$, ou $\sigma, i \models \phi$. No tempo i da execução σ do autómato $\mathcal{A}$, ϕ é válida.

$$
\begin{array}{lll}
\sigma, i \models P & sse & P \in l(\sigma(i)) \\
\sigma, i \models \neg\phi & sse & \sigma, i \not\models \phi \\
\sigma, i \models \phi \wedge \psi & sse & (\sigma, i \models \phi) \wedge (\sigma, i \models \psi) \\
 & & \\
\sigma, i \models X\,\phi & sse & i < |\sigma| \ \wedge\ \sigma, i+1 \models \phi \\
\sigma, i \models F\,\phi & sse & \exists j \geq i,\ j < |\sigma| \ \wedge\ \sigma, j \models \phi \\
\sigma, i \models G\,\phi & sse & \forall j \geq i,\ j < |\sigma| \implies \sigma, j \models \phi \\
 & & \\
\sigma, i \models \phi\ U\ \psi & sse & \exists j \geq i,\ j < |\sigma| \ \wedge\ \sigma, j \models \psi \\
 & & \wedge\ \forall k, i \leq k < j \implies \sigma, k \models \phi \\
 & & \\
\sigma, i \models E\,\phi & sse & \exists \sigma', \sigma(0) \ldots \sigma(i) = \sigma(0)' \ldots \sigma(i)' \wedge \sigma', i \models \phi \\
\sigma, i \models A\,\phi & sse & \forall \sigma', \sigma(0) \ldots \sigma(i) = \sigma(0)' \ldots \sigma(i)' \wedge \sigma', i \models \phi
\end{array}
$$

$\mathcal{A} \models \phi$ sse para toda a execução σ de $\mathcal{A}$ verifica-se $\sigma, 0 \models \phi$.

3.4 Timed Computation Tree Logic

3.4.1 Sintaxe (formato BNF)

$$
\begin{array}{lll}
\phi, \psi ::= & P_1 \mid P_2 \mid \cdots & \text{(proposições atómicas)} \\
 & \neg\phi \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (\phi \implies \psi) \mid \cdots & \text{(conectivas lógicas classicas)} \\
 & EF_{\sharp k}\,\phi \mid EG_{\sharp k}\,\phi \mid E(\phi\ U_{\sharp k}\ \psi) \mid & \text{(conectivas temporais)} \\
 & AF_{\sharp k}\,\phi \mid AG_{\sharp k}\,\phi \mid A(\phi\ U_{\sharp k}\ \psi) \mid &
\end{array}
$$

onde $\sharp \in \{<, \leq, =, >, \geq\}$ e $k \in \mathbb{Q}$.

A TCTL estende a CTL introduzindo as tais variáveis de relógio também presentes nos autómatos temporizados. Contudo, como o tempo é contínuo a conectiva X desaparece. Na TCTL qualquer fórmula passa a assumir equivalências como $AG = AG_{\geq 0}$, etc. Informalmente, uma configuração verifica, por exemplo, $E\phi U_{\leq 3}\psi$ se existe uma trajectória iniciada nesta configuração e um tempo $t \leq 3$ tal que $\rho(t) \models \psi$ e $\forall t' < t,\ \rho(t') \models \phi$.

3.4.2 Exemplo

A riqueza temporal da TCTL permite, na modelação de um sistema com um alarme, especificar a seguinte propriedade: "Se um problema ocorre, o alarme é accionado imediatamente e por um tempo mínimo de 5 unidade de tempo" ($AG(pb \implies AG_{\leq 5} alarme)$). Ou especificar que, "existe uma execução onde uma acção especifica provoca uma reacção em precisamente 11 unidades de tempo" ($EG(acc \implies AF_{=11} rea)$).

3.5 A ferramenta

O Uppaal (`http://www.uppaal.org`) foi desenvolvido pelas universidades de **Upp**sala e de **Aal**borg, e consiste numa aplicação com interface gráfica capaz de modelação (especificação por RAT), simulação e verificação (especificação por um subconjunto da TCTL) de sistemas de tempo real. A verificação fica à responsabilidade do verificador de modelos automático implementado separadamente. Sempre que um utilizador pretende fazer verificação pela interface gráfica, o Uppaal chama o verificador de modelos apresentando textualmente e graficamente o resultado da verificação das propriedades especificadas na ferramenta.

Uma vez que o verificador de modelos está implementado separadamente isso trás uma mais-valia na produção automatizada dos modelos e das especificações das propriedades. É possível verificar um modelo tendo uma especificação textual do mesmo, podendo ser no formato *ta*, *xta* ou *xml*, e tendo a especificação das propriedades no formato *q*. Esta abordagem foi utilizada no âmbito da dissertação tal como é descrito no capítulo 5.

3.5.1 Modelação

Esta ferramenta para além de utilizar as RAT também as estende. É possível, de forma muito semelhante à linguagem C, utilizar variáveis que são globais ou locais a cada autómato, variáveis locais a uma transição, constantes, variáveis inteiras limitadas, variáveis booleanas, matrizes, estruturas de dados, funções e variáveis

de relógios. Estruturalmente um autómato corresponde a uma instanciação de um *template/process*. Estes *templates* podem ser parametrizados de forma a generalizar um subsistema que se replica com parâmetros distintos.

No Uppaal, as sincronizações podem ser feitas por variáveis ou por mensagens através de canais de comunicação binários ou múltiplos. No primeiro tipo de canal uma emissão implica obrigatoriamente uma recepção, no segundo tipo já não. Caso seja pretendido, estes canais também podem ser definidos como urgentes. As sincronizações, tal como está definido na teoria dos autómatos, são executadas nas transições. Assim, a ferramenta permite, na definição das transições, a utilização de canais de sincronização, actualizações de variáveis e relógios, utilização de expressões com resultado booleano para as guardas e a definição de variáveis locais com possibilidade de inicialização não determinista. Relativamente aos estados que compõem cada autómato, é obrigatória a existência de um estado inicial por *template*. Cada estado pode assumir um nome, para identificação do mesmo na verificação do sistema, uma invariante sobre uma variável de relógio e ser definido como urgente ou *commited*.

No que respeita à especificação textual das RAT existem, como referido, duas formas de o fazer. A mais simples, mas menos interessante para esta dissertação, é a especificação gráfica. Utilizando a interface do Uppaal facilmente se produzem os autómatos com os respectivos parâmetros, geram-se estados e transições com simples cliques do rato, bastando depois instanciar cada um deles recorrendo à declaração do sistema. A parte mais interessante consiste na especificação textual dos autómatos sem perda de informação. Dos três formatos de ficheiros disponíveis, apenas um é referido na interface gráfica. Esse formato é o *xml*, trata-se do mais recente, e nele inclui-se a definição da RAT juntamente com a descrição gráfica das mesma.

No entanto o Uppaal consegue lidar com o formato *ta* mais antigo e o *xta*. Na realidade o formato *ta* foi abandonado e estendido pelo *xta*. O formato *xta* consiste numa sintaxe adaptada e mais próxima da linguagem C na qual é possível especificar tudo o que foi descrito anteriormente. No apêndice C consta um exemplo automaticamente gerado pelo tradutor apresentado no capítulo 5. De

notar que este formato pode ser acompanhado de outro ficheiro *ugi*, no qual consta a descrição gráfica da RAT. Como a preocupação da dissertação não se dirigia tanto à representação gráfica, esse ficheiro não é gerado pelo tradutor. O formato *xta* acabou por ser o escolhido por ser bastante legível e separar a descrição gráfica da RAT.

3.5.2 Verificação

Na componente de verificação do Uppaal destaca-se o motor do verificador de modelos implementado em *C* no executável *verifyta*. Numa utilização tradicional do Uppaal, sem recorrer directamente a este executável, sempre que é definida alguma propriedade na interface gráfica para além do ficheiro *xml* é gerado um ficheiro com formato *q*. Pegando nestes dois, ou no respectivo ficheiro *xta* e *q*, é possível utilizar o *verifyta* para obter os resultados da verificação (na forma de uma simples mensagem: *Property is satisfied*). No caso de uma propriedade não ser satisfeita é possível receber um trace da execução que leva a essa situação, no entanto o output por defeito não apresenta essa informação.

Tal como já foi referido, o verificador de modelos recorre-se da TCTL para a especificação das propriedades. Mas introduz algumas limitações no uso desta lógica e faz as necessárias adaptações sintáxicas. Nomeadamente não é permitido o aninhamento de fórmulas de caminho, as conectivas G e F passam a [] e <> respectivamente e é introduzido o operador $\longrightarrow$, que significa que verificando a proposição do lado esquerdo do operador então a do lado direito também tem que ser verificada nalgum estado de todos os caminhos seguintes.

Sendo *Ta* um autómato, *e* um estado desse autómato, *t* um relógio local e *v* uma variável booleana local então é possível especificar propriedades na qual as seguintes expressões podem ser utilizadas: *Ta.e*, significa que o estado *e* é estado de controlo; $Ta.t > 0$, significa que o relógio local *t* é maior que zero; $Ta.t == 0$ && $Ta.v == true$, significa que o relógio local *t* é zero e a variável *v* é *true*; $Ta.e\ imply\ Ta.t > 0$, significa que sendo *e* o estado de controlo então o relógio *t* tem que ser maior que zero.

A especificação de uma propriedade deve respeitar certas regras. Por exemplo, as expressões devem ser bem tipificadas, não conter efeitos laterais e reduzirem-se a um valor booleano. Nelas as proposições só se podem referir a (vectores de) variáveis inteiras, constantes, relógios e estados. O Uppaal suporta de raíz a verificação de propriedades de *reachability*, *safety*, *(bounded) liveness*, *deadlock freeness*. Quando não directamente suportadas, as propriedades podem ser expressas e verificadas com base nas técnicas do autómato de teste e enriquecimento do autómato (com relógios e varáveis).

3.5.3 Ferramentas Derivadas

O Uppaal tem sido alvo de um investimento contínuo, pelas universidades envolvidas, permitindo que hoje existam já variadíssimas ferramentas derivadas dele. Cada uma dessas ferramentas usa parte dos fundamentos na base do Uppaal e adapta-se às necessidades de uma área específica. Por exemplo, o Uppaal TRON possui uma interface de testes para sistemas de tempo real baseados em controladores comuns. O Uppaal CORA utiliza outro tipo de autómatos temporizados designados por Linearly Priced Timed Automata (LPTA) de modo a permitir que o modelo seja anotado com a noção de custos. Esta ferramenta é adequada para estudos sobre execuções óptimas do modelo. O Times é outro derivado especializado no escalonamento, passando outras ferramentas.

3.6 Conclusão

Neste capítulo foram introduzidos alguns conceitos necessários para uma correcta utilização do Uppaal. Nem tudo ficou descrito da forma mais detalhada (consultar `http://floyd.di.ubi.pt/release/jcarvalho/TutorialPT_Uppaal.pdf` para mais informações), mas os fundamentos foram introduzidos e neste momento o interesse desta ferramenta deve ser mais perceptível.

O tipo de verificação do Uppaal complementa muito bem aquilo que a linguagem HTL verifica. Enquanto no HTL é feita uma análise de escalonamento e é garantido que o sistema ao ser executado cumpre com esse requisito, o Uppaal permite fazer uma análise temporal sobre o comportamento das tarefas (eventos). Se nos

requisitos do sistema está especificado que a situação *A* não pode ser verificada simultaneamente que a situação *B*, e sabendo quais as tarefas que implementam essas situações, então o Uppaal pode verificar se isso é ou não verdade.

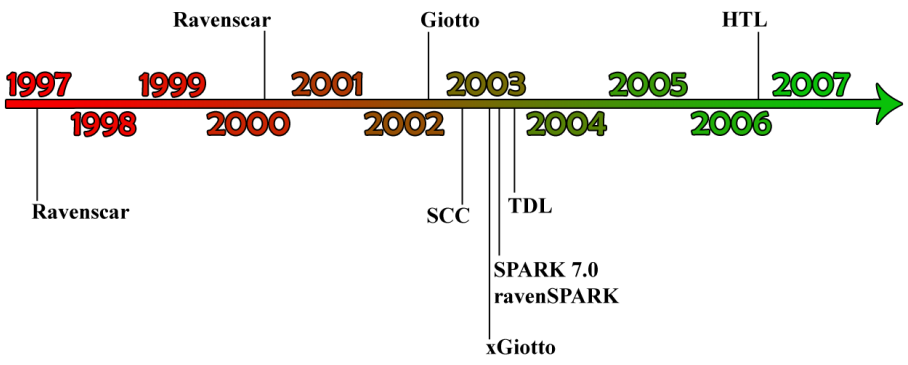
Capítulo 4

Hierarchical Timing Language

Neste capítulo descreve-se o HTL, uma das linguagens derivadas do Giotto. Esta linguagem de coordenação hierárquica destina-se a sistemas de tempo real críticos, com tarefas periódicas, e permite verificação da *time-safety* na vertente do escalonamento. O capítulo é essencialmente baseado nas teses de doutoramento do Daniel Iercan [29] e do Arkadeb Ghosal [17] (apresentadas em 2008) e no seguintes documentos [20][18].

As linguagens de Coordenação [16] tem por principal objectivo a combinação e/ou manipulação de linguagens existentes. Estas usufruem das propriedades desejadas de uma ou diversas linguagens servindo de intermediário. No caso do HTL a ideia é que esta sirva de especificação do comportamento temporal das funções definidas em C/C++. Assim a descrição temporal é separada da descrição funcional das tarefas que compõem o sistema.

Figura 4.1: Timeline de alguns mecanismos de verificação

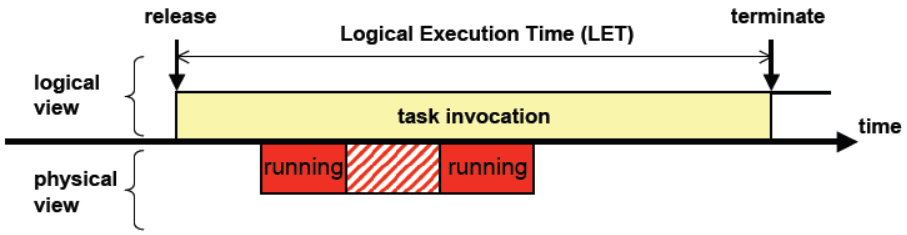


Em 2006 (ver figura 4.1) foi publicado o primeiro documento sobre o HTL, no entanto esta linguagem foi progressivamente desenvolvida ao longo dos dois anos seguintes, culminando na publicação das duas teses referidas. Das muitas vantagens, destaca-se desde já, o facto do HTL permitir um maior reaproveitamento do código C/C++ disponível e permitir verificação na componente temporal independentemente da componente lógica. É certo que a verificação produzida pelo HTL é algo limitada, porque restringe-se essencialmente ao escalonamento, mas esta dissertação apresenta uma forma de contornar essa situação.

4.1 Logical Execution Time

Apesar de ainda não ter sido referido, o mecanismo nuclear da verificação do Giotto e das linguagens derivadas é o Logical Execution Time (LET). Trata-se de uma abstracção que divide o comportamento temporal de uma tarefa da sua execução física numa dada plataforma. Esquematicamente (ver figura 4.2) a execução de uma tarefa é divida em duas componentes. Na componente lógica define-se o LET, janela temporal entre um evento inicial *release* e um evento final *terminate* representativo do comportamento temporal desejado. Na componente física define-se a execução da tarefa na plataforma com as devidas interrupções.

Figura 4.2: Logical Execution Time [17]



O LET é na realidade o código funcional com o seu espaço de memória local e sem a existência de sincronizações internas. A especificação lógica é definida por um conjunto de variáveis de programa que são lidas pela tarefa (variáveis de entrada), um conjunto de variáveis de programa actualizadas pela tarefa (variáveis de saída) e os respectivos requisitos temporais. As variáveis de programa são consideradas globais, i.e. podem ser lidas e actualizadas por todas as tarefas, no entanto o que reside na memória local não. Os eventos (*release* e *terminate*) são activados por tiques

do relógio ou por interrupções. De facto, quando o evento inicial do LET é activado, os respectivos valores das variáveis de programa são copiados da memória global para a memória local. Quando o evento final é activado os respectivos valores das variáveis de programa são actualizados com o resultado da computação da tarefa. O processo de cópia dos valores das variáveis é feito de forma síncrona, i.e. tempo lógico zero, e em termos práticos o LET de uma tarefa é determinado pela leitura da última variável de entrada e pela escrita da primeira variável de saída.

Como se pode inferir pelo esquema apresentado, a tarefa não inicia forçosamente a sua execução física no *release* do LET e não termina forçosamente a sua execução física no *terminate*. A execução física é mais precisamente determinada pela estratégia de escalonamento definida no HTL. O escalonador é que define quando as tarefas do sistema devem ser iniciadas bem como quando e quantas vezes devem ser preemptadas e resumidas. Para que um programa HTL seja considerado *time-safe*, numa determinada plataforma, cada uma das tarefas deve ser iniciada e finalizada dentro da janela temporal do LET. Uma tarefa é considerada *time-safe*, numa plataforma onde apenas é executada essa tarefa, se o seu WCET for inferior ao intervalo temporal do LET.

A cópia (actualização) das variáveis de saída só é feita no evento *terminate*, independentemente da execução da tarefa ter sido previamente finalizada ou não. Tal como a cópia das variáveis de entrada é sempre feita no evento *release*, independentemente da execução da tarefa ser imediatamente iniciada. Isto permite que o LET tenha sempre o mesmo comportamento lógico e temporal independentemente da plataforma, desde que claro o programa seja *time-safe* para as plataformas em questão.

4.2 Sintaxe

Para facilitar a descrição e compreensão de todas as características do HTL é conveniente apresentar a sintaxe da linguagem.

4.2.1 program

Começando pelo elemento mais geral, um programa HTL é composto por um ou mais programas. Cada programa é definido pela palavra reservada *program* seguida de um identificador (nome) e um corpo. O corpo de um programa é limitado por chavetas e consiste num conjunto de comunicadores e um conjunto de módulos.

Exemplo 1 Declaração de um conjunto de programas

program P { . . . }
program P2 { . . . }
program P3 { . . . }

No exemplo 1, é declarado um programa *P*, seguido de um programa *P*2 e outro *P*3, de momento ignora-se o corpo dos programas. Uma vez que os programas são a estrutura pela qual o refinamento é implementado, isso significa que dos três programas um deles é obrigatoriamente o programa principal. Enquanto o programa principal abstrai o comportamento temporal do sistema, os restantes refinem o comportamento temporal para situações concretas. A questão dos refinamentos é tratada dentro do corpo de cada programa, sendo que olhando apenas para o cabeçalho da descrição dos programas torna-se impossível identificar a hierarquia do sistema.

4.2.2 communicator

Um comunicador é uma variável tipada que pode ser acedida em determinados instantes da execução do sistema. A declaração dos comunicadores (quando necessário) é feita no início do corpo dos programas. A sua declaração é feita em bloco e iniciada pela palavra reservada *communicator*. Cada comunicador é descrito por um tipo, um identificador (nome), um período (que determina os momentos em que a leitura e escrita pode ser feita) e um *driver* de inicialização. Um *driver* de inicialização é uma função cuja especificação é feita fora do HTL, por exemplo com a linguagem C/C++, e destina-se a ser executada na declaração de qualquer variável.

No exemplo 2 são declarados três comunicadores *c*1, *c*2 e *c*3. O comunicador *c*1 é do tipo inteiro, tem por período 100 unidades de tempo e é inicializado com o

Exemplo 2 Declaração de um conjunto de comunicadores

```
communicator
  int c1 period 100 init zero;
  double c2 period 15 init zero;
  bool c3 period 43 init false;
```

driver zero. O comunicador *c*2 é do tipo real, tem por período 15 unidades de tempo e é inicializado com o *driver zero*. O comunicador *c*3 é do tipo booleano, tem por período 43 unidades de tempo e é inicializado pelo *driver f alse*. Os comunicadores podem ser lidos e escritos por qualquer tarefa do sistema cumprindo o seu período e tendo em consideração algumas limitações ao nível da invocação de tarefas (ver secção 4.4.1.3).

4.2.3 module

Os módulos são declarados após os comunicadores ou no início de qualquer programa, caso não se verifique a existência de comunicadores nesse programa. Apesar de não ser obrigatório, é recorrente que os comunicadores sejam todos definidos no programa principal. Cada módulo é declarado com a palavra reservada *module* seguida de um identificador (nome), um conjunto de *hosts* (caso estejam definidos), o identificador (nome) do modo inicial e um corpo limitado pelo uso de chavetas.

O corpo de um módulo consiste na declaração de um conjunto de portos, um conjunto de tarefas e um conjunto de modos. Cada módulo possui obrigatoriamente um modo inicial independentemente de ser único ou não. O conjunto de portos, definido no corpo do módulo, representa os portos que podem ser utilizados pelas invocações das tarefas em cada modo desse módulo. O conjunto de tarefas representa as tarefas que podem ser invocadas nos modos desse módulo. Por sua vez o conjunto de modos determina o comportamento de um conjunto de tarefas em cada instante da execução. De notar ainda que, a cada instante, os módulos de cada programa são executados simultaneamente, enquanto dentro dos módulos apenas um modo é executada de cada vez.

Exemplo 3 Declaração de um conjunto de módulos

module M **start** m1 { . . . }
module M2 [HOST1 10.0.5.123:10002, HOST2 10.0.5.122:10004] **start** m2 { . . . }
module M3 [HOST 10.0.5.122:10004] **start** m3 { . . . }

De momento ignoram-se os detalhes sintáxicos dos elementos do corpo do módulo. No exemplo 3 são declarados os módulos *M*, *M*2 e *M*3. A declaração do primeiro módulo *M*, inicializado com o modo *m*1, é exemplificativa de um módulo cuja plataforma de execução é o próprio sistema. O módulo *M*2 é inicializado pelo modo *m*2 e a sua execução é dividida pelas máquinas *HOST*1 e *HOST*2, cujos respectivos ip's e portos são identificados de forma clara. Por fim o módulo *M*3 é inicializado pelo modo *m*3 e executado por uma máquina *HOST* devidamente identificada.

4.2.4 port

Um porto é uma variável tipada (à semelhança de um comunicador) declarada no início de cada módulo. A declaração de um conjunto de portos inicia-se com a palavra reservada *port*. Cada porto consiste num tipo, num identificador e um driver de inicialização.

Exemplo 4 Declaração de um conjunto de portos

port
int local1 := zero;
double local2 := zero;

No exemplo 4 são declarados dois portos, *local*1 e *local*2. O primeiro é do tipo inteiro enquanto o segundo é do tipo real, mas ambos são inicializados pelo *driver zero*. Este *driver* terá então que ter a capacidade de lidar com a atribuição de valores inteiros e reais, consoante o tipo da variável em questão.

4.2.5 task

A declaração das tarefas é feita logo a seguir à declaração dos portos ou logo no início de cada módulo, caso não se verifique a existência de portos nesse módulo. A declaração de cada tarefa é iniciada com a palavra reservada *task* seguida de um identificador, um conjunto de portos de entrada, um conjunto de portos de estado, um conjunto de portos de saída, um possível identificador da função, e a possível especificação do WCET da tarefa. Os portos de entrada, estado e saída consistem num nome e num tipo, mas um porto de estado requer também a descrição de um driver de inicialização.

Os portos de entrada e saída não são aplicações directas dos portos vistos anteriormente mas sim uma declaração temporária, substituída na invocação das tarefas com a utilização de um porto concreto ou de um comunicador. Enquanto os portos de entrada e saída representam as variáveis de entrada e saída da função associada à tarefa, o porto de estado representa uma variável local da função que não tem interacção directa com o resto do programa. Se não for associada uma função à tarefa significa que a tarefa é abstracta e obrigatoriamente é refinada por outra(s) tarefa(s). Uma tarefa associada a uma função designa-se por tarefa concreta. A cada tarefa pode e deve ser associado o WCET para a plataforma de destino, no entanto é possível omitir e/ou alterar estes valores sem influenciar o comportamento temporal e lógico do programa, desde que claro o programa se mantenha *tme-safe*.

Exemplo 5 Declaração de um conjunto de tarefas

task t **input**() **state**() **output**(int o1, int o2) **wcet** 100;
task t2 **input**(int i1) **state**() **output**() **function** f2 **wcet** 80;
task t3 **input**(int i2) **state**(int s:= zero) **output**(int h1) **function** f3;

No exemplo 5 são declaradas as tarefas t, $t2$ e $t3$. A tarefa abstracta t não possui portos de entrada e de estado, mas possui dois portos de saída $o1$ e $o2$ do tipo inteiro, para além de ter um WCET de 100 unidades de tempo. Na declaração da tarefa concreta $t2$ é possível constatar a existência de apenas um porto de entrada $i1$ do tipo inteiro, bem como a associação da função $f2$ e um WCET de 80 unidades de tempo. Por fim a tarefa concreta $t3$ possui um porto de entrada $i2$ do tipo inteiro, um porto de estado s do tipo inteiro inicializado com o driver *zero*, um porto de

saída $h1$ e está associada à função $f3$. De notar que na declaração destas três tarefas existe uma situação atípica de um programa HTL. O WCET quando existente, deve estar definido em todas as tarefas e não apenas nalgumas.

4.2.6 mode

Nos módulos, a declaração de um conjunto de modos é a componente central e fundamental para determinar o comportamento do módulo. É através da declaração dos modos que se determina quais as tarefas a serem executadas em que momentos. A declaração de cada modo inicia-se com a palavra reservada *mode* seguida de um identificador (nome), um período, um possível identificador de programa que refina o modo e um corpo limitado por chavetas. O corpo do modo é constituído por um conjunto de invocações de tarefas e um conjunto de instruções de mudança de modo (*switch*). O período do modo determina o intervalo de tempo necessário para que as instruções se repitam.

Exemplo 6 Declaração de um conjunto de modos

```
mode m period 230 { ... }
mode m2 period 100 program P1 { ... }
```

No exemplo 6 são declarados dois modos, m e $m2$. O modo $m2$ possui um período de 100 unidades de tempo e é refinado pelo programa $P1$ enquanto o modo m possui um período de 230 unidades de tempo.

4.2.7 invoke

As invocações das tarefas ditam quando uma tarefa previamente declarada deve ou não ser executada. É através destas invocações que o LET de cada tarefa é definido, considerando o instante em que o último porto de entrada é lido e o instante em que o primeiro porto de saída é escrito. Cada invocação inicia-se com a palavra reservada *invoke* e é sucedida pelo identificador da tarefa que se pretende executar, pelo mapeamento dos portos concretos e/ou instâncias dos comunicadores (relativas ao período do modo) para os respectivos portos de entrada e saída bem como pelo identificador da tarefa pai (quando existente). Para que um comunicador possa

ser utilizado na invocação de uma tarefa é necessário que o período do modo seja múltiplo do período do comunicador.

Exemplo 7 Declaração de um conjunto de invocações

invoke t **input**() **output**((h1,1),(c1,1));
invoke t2 **input**(p) **output**();
invoke t3 **input**(c1,2) **output**(p) **parent** t0;

No exemplo 7 são declaradas três invocações de tarefas, *t*, *t*2 e *t*3. A invocação da tarefa *t* apenas define dois comunicadores como portos de saída. Esses comunicadores são actualizados na primeira unidade de tempo do seu período, i.e. se o período de *h*1 for 200 então será no instante 200. A invocação da tarefa *t*2 apenas define um porto concreto como porto de entrada da função. Por fim a invocação da tarefa *t*3 define como porto de entrada o comunicador *c*1 na instância 2 e como porto de saída o porto concreto *p*, esta tarefa é ainda o refinamento da tarefa *t*0.

4.2.8 switch

A declaração de um *switch* é feita após a declaração das invocações de tarefas. Cada declaração inicia-se com a palavra reservada *switch* seguida de uma condição e do identificador de um modo. No exemplo 8 é declarado um *switch* para o modo *m*1 quando *cond*(*c*) for verdade. A condição refere-se sempre a uma função descrita fora do HTL pelo que ela representa uma autentica caixa negra para quem apenas olha para a especificação HTL.

Exemplo 8 Declaração de um switch
switch (cond(c)) m1;

4.3 Características Principais

4.3.1 Refinamento

O refinamento é uma das características mais fortes do HTL pois permite simplificar a análise de escalonamento feita estaticamente (pelo compilador). Esse refinamento

pode ser designado por refinamento de modos, pois um modo pode ser totalmente substituído por um programa. No entanto, o comportamento do refinamento interage ao nível mais específico com a descrição das tarefas. Tal como foi referido, no HTL existe a noção de tarefas abstractas e tarefas concretas. Enquanto a primeira não possui qualquer tipo de funcionalidade a segunda representa uma função produzida fora do HTL. No processo de refinamento apenas as tarefas abstractas podem ser refinadas, elas servem essencialmente como invólucros de um comportamento temporal desejado.

O refinamento está definido de modo a restringir a análise de escalonamento ao programa principal. Assim, um programa correctamente hierarquizado é verificado mais facilmente do que na ausência das hierarquias. A responsabilidade de tentar programar partindo de descrições generalistas, passando posteriormente para descrições mais especificas fica totalmente à responsabilidade de quem programa. Neste contexto, uma tarefa abstracta pode dar origem a várias tarefas concretas e a sensibilidade para detectar estas situações requer algum treino. Tendo uma abstracção suficiente e correcta do sistema, é possível determinar se o programa é escalonável antes mesmo de realizar as descrições concretas.

4.3.2 Distribuição da Computação

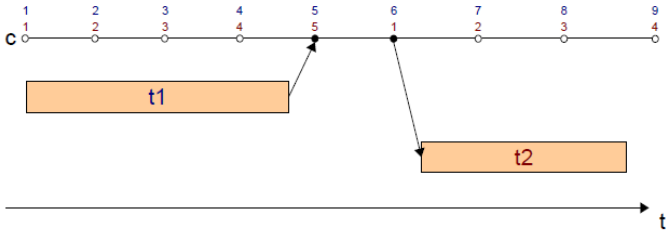
Muitos dos sistemas de tempo real são distribuídos, i.e. o esforço de computação é distribuído por várias componentes embebidas que interagem por canais de comunicação. No HTL a distribuição é especificada pelo mapeamento dos módulos pelas respectivas máquinas. O processo de distribuição é implementado através da replicação, dos comunicadores partilhados, pelas diversas máquinas. Sempre que uma tarefa actualiza um comunicador, automaticamente esse valor é transmitido por todas as máquinas. O comportamento temporal do programa HTL é independente do número de máquinas em que é executado, no entanto os mapeamentos são devidamente tidos em consideração na geração do código bem como da análise de escalonamento. O mesmo programa pode ser $time - safe$ com um determinado mapeamento e não com outro. No entanto se for $time - safe$ para ambos os mapeamentos, o comportamento temporal mantêm-se.

4.3.3 Comunicação Inter-Tarefas

O HTL suporta dois modelos de comunicação distintos implementados pela utilização de comunicadores e portos. No caso dos comunicadores está-se perante comunicação inter-tarefas. Este tipo de comunicação é mais abrangente e permite que as tarefas comuniquem entre si sem restrições de maior. A restrição mais relevante pretende-se com o facto de o HTL lidar com sistemas periódicos. Logo, o período do modo, onde um comunicador é utilizado, tem de ser múltiplo do período do comunicador.

Existem outras restrições (ver secção 4.4.1.2) sendo que uma delas impede que dentro do mesmo modo duas tarefas escrevam para a mesma instância do mesmo comunicador, no entanto esta restrição serve para manter alguma coerência lógica do programa. Isto elimina parte dos erros que podem existir na comunicação das tarefas, mas não impede que no mesmo momento duas tarefas escrevam na mesma instância do mesmo comunicador, originando um incorrecto funcionamento do sistema.

Figura 4.3: Comunicação por comunicadores [29]



Na figura 4.3 é exemplificada a utilização do comunicador *c* pela tarefa *t*1 e *t*2. O período de *c* é de 1 unidade de tempo, o período da tarefa *t*1 é de 10 unidades de tempo e o período da tarefa *t*2 é de 5 unidades de tempo. Uma vez que o período de *t*1 é de 10 unidades de tempo significa que existem 10 instâncias do comunicador *c* que podem ser acedidas por *t*1. Uma vez que o período de *t*2 é de 5 unidade de tempo só existem 5 instâncias acessíveis para essa tarefa. Não confundir o LET das tarefas com o período das tarefas. O período das tarefas corresponde ao período do modo onde elas são invocadas, apesar de na descrição do tradutor este termo não ser utilizado com o mesmo significado. No exemplo a tarefa *t*1 escreve na instância

5 do comunicador e a tarefa $t2$ na instância 1 do comunicador.

Este exemplo demonstra que tarefas com períodos diferentes não acedem o mesmo número de vezes ao mesmo comunicador, a tarefa com maior período vai fazer menos acessos. Neste exemplo a tarefa $t2$ vai ler ciclicamente valores iguais uma vez que enquanto a tarefa $t1$ só é executada uma vez a tarefa $t2$ vai ser executada duas vezes. Não existe nenhum mecanismo automático de análise de fluxo, ligado ao HTL, que permita detectar comportamentos indesejados na manipulação dos comunicadores. Até porque este comportamento está de acordo com o pretendido. A componente funcional de cada tarefa (descrita por uma função) representa, neste momento, uma caixa negra para o programa HTL.

4.3.4 Comunicação Directa de Tarefas

A comunicação directa de tarefas é feita por intermédio dos portos sendo que estes não possuem instâncias como os comunicadores. Na figura 4.4 é representada a comunicação directa na forma de uma simples transmissão de dados entre duas tarefas. A tarefa $t1$ comunica directamente o valor do porto para a tarefa $t2$. Este tipo de comunicação introduz a noção de precedências no HTL limitando consideravelmente os efeitos laterais deste tipo de noção.

Figura 4.4: Comunicação por Ports [29]

Este tipo de comunicação só é possível entre duas tarefas com o mesmo período, ao contrário dos comunicadores. Isto altera um pouco o determinismo absoluto do LET, porque na presença de um porto o LET da tarefa pode variar a cada invocação. Sempre que um porto é utilizado, a execução da tarefa não pode ser iniciada enquanto o porto for lido e deve ser imediatamente terminada assim que o porto for escrito. Assim, o LET pode ser encurtado consoante o momento em que o porto é lido e/ou escrito e isso pode variar a cada invocação das tarefas.

4.3.5 Composição Sequencial

O HTL suporta composição sequencial de tarefas, i.e. as tarefas são executadas em momentos distintos, através dos modos e dos seus *switchs*. A composição sequencial de tarefas, no HTL, consiste na substituição da execução de um conjunto de tarefas s por outro conjunto s'. Cada um dos conjuntos pode ter um único elemento ou vários. Quando as tarefas do conjunto s estão a ser executadas, as tarefas do conjunto s' não estão, e vice-versa. A mudança de contexto (modo) é feita pelo *switch* quando a respectiva condição é avaliada como verdadeira.

4.3.6 Composição Paralela

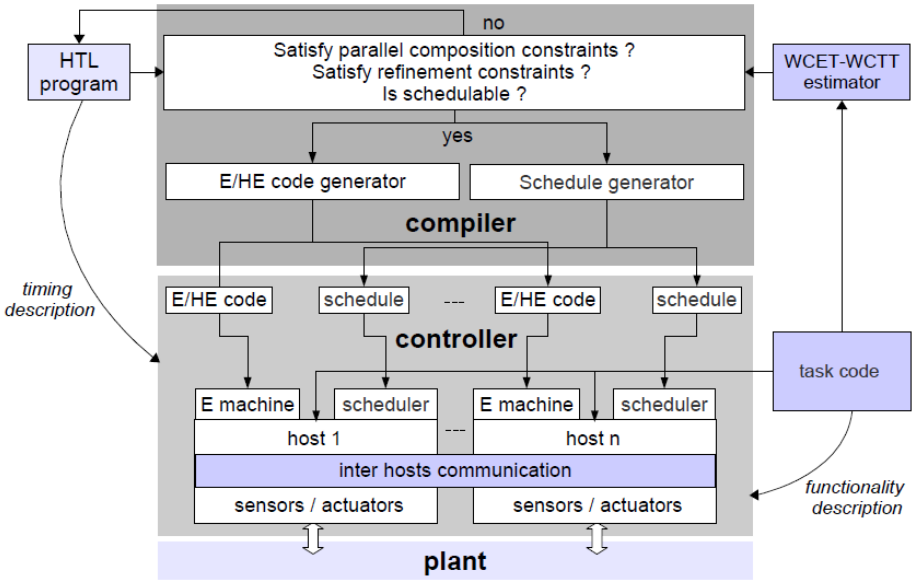
O HTL suporta também a composição paralela de tarefas, i.e. as tarefas podem ser executadas em simultâneo. A composição paralela é concretizada, no HTL, pelos módulos. Cada módulo é executado em paralelo (simultâneo) independentemente do período dos modos de cada um deles. Não existe qualquer tipo de restrição na composição paralela de tarefas, no entanto o HTL verifica se as tarefas compostas paralelamente são escalonáveis numa determinada plataforma. Para que isso seja feito o programa HTL deve estar anotado com os WCET de cada tarefa para a plataforma de destino.

4.4 Compilador HTL

Na figura 4.5 é representada a estrutura do compilador HTL bem como o ambiente de execução. A compilação de um programa HTL envolve uma análise da descrição do programa (verificação das restrições sobre a composição sequencial e paralela) bem como a geração do respectivo código. A análise feita pelo compilador garante que a descrição do programa está correctamente formada ($Well - Formed$), que a mesma tem um comportamento temporal interno coerente ($Well - Timed$) e que é escalonável para uma determinada plataforma. Na questão do escalonamento os WCET e os Worst-Case Transmission Time (WCTT) das tarefas são obtidos por uma ferramenta externa e o algoritmo de escalonamento utilizado, na versão actual do compilador, é o EDF. É possível determinar o WCET por uma análise estática pura ou por combinação de diversas análises estáticas [44], no entanto o compilador não

inclui nenhuma destas funcionalidades.

Figura 4.5: Esquema do Compilador HTL [29]



Se e só se a análise for devidamente validada é que é gerado o código para as respectivas máquinas. A geração do código é feita compilando toda a descrição do programa para cada uma das máquinas, sendo que cada uma delas possui uma replicação de todos os comunicadores e de todos os portos. Apesar da replicação de código, só são executadas as tarefas mapeadas para essa máquina. Sempre que uma tarefa actualizar um comunicador, o valor desse comunicador é distribuído pelas diversas máquinas. Aquando da geração do código, o mesmo é dividido em duas componentes. Uma componente alimenta a máquina virtual Hierarchical Embedded Machine (HE-Machine) de cada *host* através do Hierarchical Embedded Code (HE-Code), ou a E-Machine através do E-Code no caso de ser utilizada a versão *flatten* do compilador HTL. A outra componente alimenta o escalonador de tarefas das respectivas máquinas virtuais.

4.4.1 Well-Formedness

Para que um programa HTL seja considerado bem formado o mesmo deve respeitar um conjunto de restrições sobre a manipulação dos programas, dos comunicadores, das invocações de tarefas e dos refinamentos.

4.4.1.1 Restrições nos Programas

Só pode existir um programa principal; Esse programa principal tem de ser super-programa para todos os restantes programas; Para cada programa (excepto para o principal), só existe um único super-programa directo; Para cada módulo (excepto os módulos principais), só existe um único super-módulo directo; Cada programa só pode refinar um modo de um módulo; O modo inicial de um módulo deve estar definido no conjunto de modos desse módulo; O modo de destino de um *switch* deve estar contido no conjunto dos modos do seu módulo.

4.4.1.2 Restrições nos Comunicadores

Se um comunicador é declarado num programa P então não pode ser novamente declarado num dos subprogramas de P; Se um comunicador é utilizado por uma invocação de tarefa ou por um *switch* num programa P então ele deve estar declarado nalgum dos super-programas de P; Se um comunicador for actualizado por algum sub-módulo do módulo *mdl* então ele não pode ser actualizado por nenhum outro sub-módulo de *mdl*.

4.4.1.3 Restrições nas Invocações de Tarefas

Para cada invocação, o instante em que as leituras são feitas deve ser inferior ao das actualizações; As precedências entre as tarefas de um modo são acíclicas; Se uma invocação de uma tarefa num modo m acede a um porto então esse porto deve estar declarado no módulo onde o modo m é declarado; Num modo duas invocações não podem escrever para o mesmo porto nem escrever para a mesma instância do mesmo comunicador; Cada invocação de tarefa, não pode escrever mais de uma vez para o mesmo porto ou para a mesma instância do mesmo comunicador; Uma tarefa só pode ser invocada num modo m do módulo *mdl* se a tarefa for declarada no módulo *mdl*; Se a invocação da tarefa t acede a um comunicador, então o tipo do comunicador deve corresponder com o tipo do porto temporário definido na declaração da tarefa t; Se uma invocação de tarefa no modo m acede a um comunicador c, então o período do modo deve ser múltiplo do período do comunicador c.

4.4.1.4 Restrições nos Refinamentos

O período do modo *m* deve ser idêntico ao período de todos os modos do programa que refina *m*; Todas as invocações feitas no modo de um módulo, que não seja principal, devem ter uma tarefa pai; A invocação de uma tarefa no modo *m* do módulo *mdl* deve ter uma tarefa pai única relativamente, a todas as invocações de tarefas do modo *m*, e a todas as invocações de tarefas feitas em módulos paralelos ao módulo *mdl*; O tempo de leitura de uma invocação de tarefa não pode deve ser superior ao tempo de leitura da invocação da sua tarefa pai, tal como o tempo de actualização de uma invocação de tarefa não pode ser inferior ao tempo de actualização da invocação da sua tarefa pai (LET do Pai é menor ou igual ao LET do filho); Cada relação do conjunto de precedências de um modo deve ser preservada no modo da invocação das tarefas pais.

4.4.2 Well-Timedness

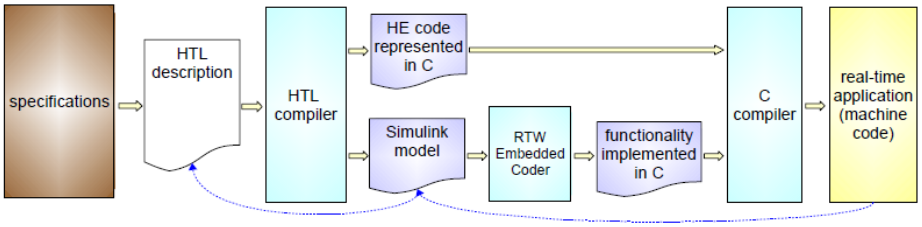
Um programa HTL possui um comportamento temporal coerente se o WCET/WCTT da invocação de todas as tarefas concretas for menor ou igual ao WCET/WCTT da invocação das respectivas tarefas pai. Ao contrário da *Well-Formedness* a *Well-Timedness* é dependente da plataforma. Se um programa for *Well-Timed* reúne as condições para que possa ser escalonável, no caso contrário não. Inclusivamente o escalonamento do programa principal é válido para os seus refinamentos se o programa for *Well-Formed* e *Well-Timed*.

4.4.3 HTL-Simulink Tool Chain

No âmbito da tese do Daniel Iercan foi desenvolvida uma *Tool Chain* envolvendo o Simulink como ferramenta simulação e geração de código *C* automático. O compilador de HTL gera o HE-Code em *C*, código esse que tem posteriormente de ser compilado juntamente com o código *C* das funções. Só então é gerado código executável para as plataformas. Nesta fase de desenvolvimento ainda não é o caso, no entanto parte da filosofia por trás do HTL pretende que o código funcional possa ser desenvolvido com qualquer linguagem sequencial. O que foi feito para simplificar o processo de geração do código funcional foi estender o compilador para que juntamente com o HE-Code fosse gerado um modelo simulink. Essa extensão

é no entanto limitada uma vez que apenas permite a construção do modelo para programas sem precedências de tarefas e com apenas um *switch* por modo.

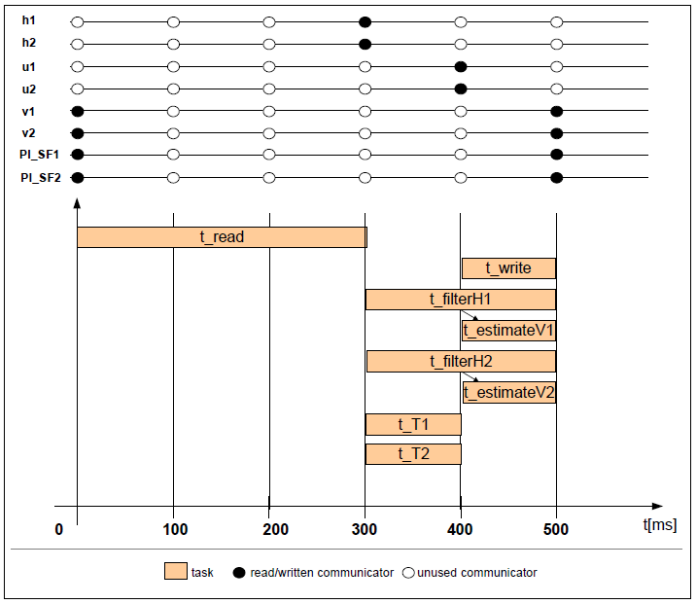
Figura 4.6: Esquema HTL-Simulink Tool-Chain [29]



4.5 Conclusão

Ao estudar o HTL encontraram-se várias análises "manuais" do comportamento temporal de programas como exemplificado na figura 4.7. Estas análises são preciosas porque representam graficamente os LETs das tarefas de um programa HTL. Através de uma simples visualização destas análises conseguem-se retirar várias informações sobre quais as tarefas que são ou não são executadas simultaneamente, sobre o fluxo de dados, etc.

Figura 4.7: Modelo Temporal de um Programa HTL [29]



No entanto, estas análises não estão de nenhuma forma automatizadas ou formalizadas. Na implementação actual do HTL não existe nenhuma forma de relacionar aquilo que o programador quer com aquilo que ele fez. O programador tem a garantia que, se foi gerado código executável, então o seu programa é escalonável e respeita aquele conjunto de boas propriedades anteriormente definidas. Mas não tem mais garantias. Se ele se enganar na instância de um comunicador que actualiza uma variável fundamental para o bom funcionamento do seu programa e o programa por mero azar continuar a ser escalonável, como faz ele para detectar algum conflito? Por simulação através do Simulink? Por tentativas?

Não será desejável e possível garantir, com métodos mais formais, que uma tarefa realmente não acontece simultaneamente que outra? Que um conjunto de tarefas não é executado simultaneamente que outro? Que uma tarefa é sempre executada quando outra for executada? Ou até mesmo que uma dada tarefa é alguma vez executada? Desejável é, o reforço da análise temporal nunca é de desprezar nos sistemas de tempo real críticos. Possível também é, basta para tal usar a verificação de modelos. No capítulo seguinte é apresentada a ferramenta capaz de traduzir um programa HTL num modelo Uppaal para verificação de propriedades complementares, como as anteriormente referidas.

Capítulo 5

HTL2XTA, O Tradutor

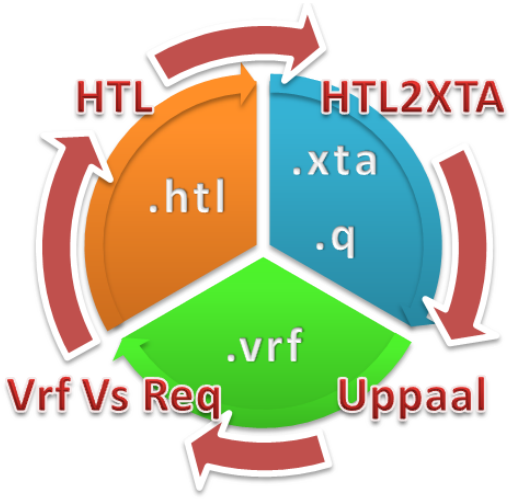
No estudo feito aos mecanismos de verificação ficou claro que o HTL era uma linguagem com potencial na verificação de sistemas de tempo real. Apesar de a verificação feita pelo HTL não ser tão completa como a do SPARK, não existe qualquer restrição no uso da memória dinâmica, no uso da recursividade, etc. No estudo feito, também ficou claro que qualquer mecanismo de verificação não era por si só suficiente. Concluiu-se que a verificação de modelos, mais precisamente a ferramenta Uppaal, era um bom candidato para a extensão da verificação do HTL, tal como o é para muitas outras linguagens.

Neste capítulo descreve-se o HTL2XTA como ferramenta automatizada de geração de modelos e especificações de propriedades. O tradutor (HTL2XTA) enquadra-se numa *Tool-Chain* (figura 5.1) delineada com o objectivo de estender a verificação de programas HTL. Esta *Tool-Chain* pode ser utilizada paralelamente à *Simulink Tool-Chain*. Enquanto a segunda permite a geração do código C/C++ a partir de uma especificação HTL, a primeira permite verificar se a especificação cumpre com os requisitos temporais previamente definidos.

O HTL2XTA recebe uma especificação HTL (ficheiro .htl) e devolve o respectivo modelo (ficheiro .xta) juntamente com as propriedades automaticamente inferidas (ficheiro .q). Tendo estes ficheiros, é possível utilizar a interface gráfica do Uppaal, ou directamente o motor do verificador de modelos (verifyta), para apurar se as propriedades são ou não satisfeitas. Para facilitar este processo, foram criados

scripts para executar o verificador de modelos gerando um relatório (ficheiro .vrf) por cada modelo verificado. Com este relatório e com a respectiva especificação de propriedades é possível saber se parte dos requisitos temporais são ou não cumpridos.

Figura 5.1: Esquema HTL2XTA Tool-Chain



Para completar a verificação dos requisitos temporais basta acrescentar à especificação de propriedades, automaticamente gerada, novas fórmulas que verifiquem os requisitos que não foram automaticamente contemplados. Propriedades mais interessantes como, 'A tarefa X nunca pode ocorrer simultaneamente que a tarefa Y', ou 'Se a tarefa X ocorrer, passado T unidades de tempo a tarefa Y tem de ocorrer' não são automaticamente geradas. É preciso estudar os requisitos temporais e encontrar correspondências com as tarefas que implementam essas situações de modo a poder especificar propriedades sobre esses requisitos.

5.1 Automatização da Verificação

No estudo realizado constatou-se que alguns artigos apresentavam esquemas de tradução para modelos de autómatos temporizados. Infelizmente em nenhum dos artigos estudados (relacionados com sistemas de tempo real) foi encontrado uma versão automatizada. Existindo um esquema de tradução o processo de automatização é uma mera formalidade, mas essa formalidade torna a tradução muito mais exequível. Num processo de desenvolvimento quanto mais as ferramentas forem

automatizados maior vai ser a recepção das mesmas por parte das equipas de desenvolvimento. Sem esquecer que, uma tradução manual também permite que sejam introduzidos erros com maior facilidade, enquanto numa tradução automatizada não. Para ambas as situações o complicado é sempre provar formalmente que o esquema de tradução está correcto.

No caso do tradutor HTL2XTA optou-se por automatizar o processo de tradução reconstruindo parcialmente o compilador do HTL. Não se reaproveitou totalmente o trabalho feito pela equipa que desenvolveu o HTL, uma vez que o compilador de HTL está implementado em java e pretendia-se uma implementação que beneficiase mais de uma linguagem de programação funcional como o Ocaml. Na realidade o tradutor desenvolvido é praticamente um novo compilador HTL, mas que no lugar de produzir *bytecode* produz código Uppaal.

Foi necessário reconstruir o lexer, o parser e a respectiva Abstract Syntax Tree (AST) (ver apêndice A) da linguagem em Ocaml. Não se fez nenhuma implementação da verificação de tipos do HTL para permitir que o tradutor possa ser utilizado mesmo sem o programa HTL estar concluído. Desde que não existam erros léxicos e sintáxicos é possível obter um modelo do programa. Seja, se um programa não for *Well-Timed* ou *Well-Formed* é possível obter o respectivo modelo.

5.1.1 Tradução do Modelo

Para tornar o tradutor eficiente e viável foi imperativo generalizar ao máximo os mecanismos de tradução, pelo que é difícil produzir modelos com os autómatos minimizados. A difícil existência de autómatos minimizados também está relacionada com o facto de alguns autómatos terem de possuir um ou outro estado suplementar para representar um determinado evento. Só desta forma se permite que posteriormente sejam especificadas propriedades que incluam esse evento.

Sendo a verificação o principal objectivo do tradutor, decidiu-se evitar a construção de um esquema de tradução que produzisse modelos muito complexos. Isto porque, quanto mais complexos forem os modelos mais difícil se torna verificar as propriedades pretendidas. Adoptou-se um esquema que produz um modelo

suficientemente abstracto. Tendo algo abstracto facilmente se consegue acrescentar informação, tal como até foi feito (por exemplo, a primeira versão do esquema não contemplava refinamento). Fazer o contrário já se torna complexo devido às dependências sobre a informação que se pretende remover do modelo.

Utilizando este mesmo princípio, o esquema foi construído com uma abstracção maior do que a actual, mantendo sempre como objectivo a verificação de propriedades temporais. Uma vez que o HTL já faz uma análise de escalonamento, a abstracção utilizada ignorou por completo a execução física das tarefas. Ao contrário de [37], onde é apresentado um esquema de tradução de Giotto para Uppaal, decidiu-se que a abstracção a ter em consideração não devia, nem podia, considerar a execução física das tarefas. Como não interessa verificar o escalonamento das tarefas, o LET destas linguagens é mais que suficiente para verificar as restantes propriedades temporais.

Através do LET é possível saber em que intervalo de tempo a tarefa vai ser executada, a partir deste facto pode-se construir propriedades temporais sobre as tarefas. Tendo isto em consideração, fez-se corresponder a cada invocação de tarefa um autómato temporizado cujo LET é definido através do cálculo dos portos concretos e dos comunicadores utilizados na sua declaração. O limite inferior do LET corresponde ao momento em que é lida a última variável e o limite superior corresponde ao momento em que é escrita a primeira variável.

Uma vez que cada modo dentro de um módulo representa a execução de um conjunto de tarefas e que cada módulo só pode estar num modo de cada vez, faz-se corresponder a cada módulo um único autómato temporizado. A cada ciclo de execução do autómato, faz-se a sincronização com os autómatos representativos das tarefas invocadas num determinado modo. Na abstracção adoptada é ignorado por completo o tipo dos comunicadores bem como o driver de inicialização. Desconsideram-se os valores dos comunicadores bem como a implementação funcional das funções. Para além do HTL considerar ambas as situações como caixas negras e aqui se pretender um comportamento análogo, não se considerou pertinente tentar recolher esse tipo de informação no código das funções devido à complexidade que isso iria trazer ao modelo.

Sumariamente todo o esquema de tradução rege-se por esta abstracção, i.e. autómatos temporizados que representam o LET das tarefas e autómatos temporizados que representam cada módulo. O tradutor foi testado com diversos programas HTL e verificou-se que os modelos de programas de complexidade elevada não permitem verificação, apesar da simplicidade da abstracção. Todos os casos de complexidade intermédia foram alvo de verificação bem sucedida, mas constatou-se que os refinamentos aumentam substancialmente a complexidade do modelo. Para aliviar esta situação, o tradutor permite a construção de modelos com os níveis de refinamento desejados.

5.1.2 Inferência de Propriedades

A produção automatizada de propriedades tem essencialmente dois objectivos, servir de suporte na validação do tradutor e reduzir a quantidade de propriedades a especificar manualmente. As propriedades inferidas estão todas relacionadas com características bem definidas do HTL, como os períodos dos modos, o LET de cada tarefa, as invocações de tarefas feitas em cada modo e o refinamento dos programas.

A cada uma das características referidas corresponde, quase sempre, mais do que uma propriedade. Assim é comum encontrar duas ou mais propriedades, que se complementam, para validar o LET de uma tarefa, o período de um modo, entre outros. À semelhança de uma tabela de restreabilidade, cada propriedade é devidamente comentada com, uma descrição textual da característica a verificar, uma referência da posição da respectiva descrição da característica no ficheiro HTL e o resultado booleano pretendido nessa propriedade.

Esse resultado booleano foi introduzido, no comentário de cada propriedade, uma vez que é possível recorrer à não validação de uma propriedade para concluir que um determinado requisito temporal é cumprido. No entanto foi feito um esforço, que resultou na não existência de propriedades com uma conotação negativa. Torna-se mais intuitivo para o utilizador verificar que todas as propriedades foram satisfeitas, do que umas não e outras sim e constatar se realmente era esse o valor esperado.

As propriedades inferidas podem e devem ser complementadas manualmente com informação extraída dos requisitos temporais estabelecidos. Para tal é preciso ter em consideração a identificação de todos os estados e dos respectivos autómatos temporizados. Considerando um Programa *P*, um módulo *M*, um modo *m* e uma invocação de tarefa *t*, o autómato do módulo M é identificado por *sP_M*, o estado que representa a invocação da tarefa é identificado por *sP_M.m_t*, o autómato representativo do LET da tarefa (futuramente designado por autómatos de tarefa) é identificado por *M_m_t* e o estado do próprio LET é identificado por *M_m_t.Let*. Associado a cada autómato de um módulo existe ainda um estado representativo da execução de cada modo identificado por *sP_M.m*, bem como outros estados que não possuem uma relação directa com o HTL (a descrição dos mesmos é feita na secção seguinte).

A especificação de propriedades permite a utilização de diversos relógios presentes no modelo. Cada autómato é constituído de, pelo menos, um relógio local reinicializado numa transição que sai do estado inicial. No caso dos autómatos de módulo, o relógio local é designado por *t* e é identificado de forma análoga a um estado desse autómato. No entanto a utilização de um relógio implica que o mesmo seja comparado com uma expressão inteira, por exemplo: *sP_M.t* >= 0. No caso dos autómatos das tarefas, o relógio local (representativo do período do modo) onde essa tarefa é invocada é designado por *tt*. Neste tipo de autómatos existe ainda outro relógio local designado por *t* reinicializado no instante 0 do LET dessa tarefa. Existe um relógio global designado por *global* que, apesar de não ser utilizado em nenhuma propriedade inferida do modelo, pode ser utilizado nas propriedades especificadas manualmente.

5.2 Algoritmo de Tradução do Modelo

É possível consultar no apêndice C o resultado da aplicação prática do algoritmo de tradução, descrito nesta secção, a um caso de estudo descrito no capítulo 6. Este algoritmo baseia-se na AST apresentada no apêndice A.

5.2.1 Considerações Iniciais

Alguns aspectos da linguagem HTL são puramente ignorados pelo algoritmo do tradutor. Ou porque não acrescentam informação relevante, ou porque não são suficientemente abstractos para o modelo. Uma vez que a AST foi pensada para suportar qualquer eventualidade, a mesma possui informação que não é analisada ou traduzida pelo algoritmo.

Considera-se nesta secção a definição da função T, que aceita um programa HTL (de facto a AST do HTL) e que devolve a RAT correspondente. Esta função é definida naturalmente por recursividade sobre a estrutura da AST da linguagem HTL. Assim passa-se a definir T para cada caso particular da AST em questão. Considera-se ainda a função A, que aceita um programa HTL e que devolve informação necessária para construção da RAT.

5.2.1.1 Mudança de Modo

Seja (n, s, p) a declaração de uma instrução switch onde, n é o nome do modo para o qual se pretende fazer a mudança de execução, s o nome da função (no código funcional) que valida se a mudança é ou não efectuada e p a posição da declaração da instrução no ficheiro HTL. Seja $Prog$ o conjunto de todos os programas, $Prog_i$ o programa i e $Prog_1$ o programa principal, então $\forall switch \in Prog, T_{switch}(n, s, p) = \emptyset$.

Isto é, qualquer mudança de modo (switch) não produz um efeito directo na aplicação do algoritmo de tradução. Recorre-se ao não determinismo do Uppaal para permitir que os modos sejam alternados em cada execução. Com isto perde-se a informação de qual modo permite a mudança para outro e parte-se do princípio que, dentro de um módulo, qualquer modo permite a passagem para todos os outros modos desse módulo.

Esta abordagem reduz a complexidade do modelo sem o invalidar. Hipoteticamente todos os modos dentro de um módulo são atingíveis, independentemente da forma como essa troca é feita. Como a função que valida se a mudança de modo é feita representa uma caixa negra para o HTL, o algoritmo nunca consegue inferir

informação sobre essa situação.

5.2.1.2 Tipos e Drivers de Inicialização

Seja dt uma declaração, ct o tipo de uma declaração e ci o driver de inicialização de uma declaração, então $\forall dt \in Prog, T_{dt}(ct, ci) = \emptyset$. Quer o tipo como o valor inicial (*driver* de inicialização) de uma declaração não tem qualquer impacto na aplicação do algoritmo de tradução. Esta informação é considerada pouco pertinente, já que não contribui de forma significativa para uma análise temporal. Inclusivamente os valores das variáveis são desconhecidos do HTL. Apesar de ser possível iniciar uma variável com um *driver* de inicialização, o valor que a variável assume é desconhecido e apenas aqui é retratado pelo nome do driver de inicialização.

5.2.1.3 Declaração de Tarefas

Seja (n, ip, s, op, f, w, p) a declaração de uma tarefa, onde n é o nome da tarefa, ip a lista dos *input port*, s a lista dos estados internos, op a lista dos *output port*, f o nome da função que implementa a tarefa, w o WCET da tarefa e p a posição no ficheiro HTL da declaração da tarefa, então $\forall task \in Prog, T_{task}(n, ip, s, op, f, w, p) = \emptyset$. A declaração das tarefas, à semelhança das situações anteriores, não tem qualquer impacto na aplicação do algoritmo de tradução. Nem do ponto de vista de uma análise de tipos a declaração das tarefas é considerada. Relativamente à implementação das tarefas no modelo, a declaração das invocações combinada com o período dos comunicadores proporcionam toda a informação necessária.

5.2.1.4 Declaração de Comunicadores

Seja (n, dt, pd, p) a declaração de um comunicador, onde n é o nome do comunicador, dt o tipo do comunicador e o driver de inicialização (ct, ci), pd o período do comunicador e p a posição no ficheiro HTL da declaração do comunicador, então $\forall communicator \in Prog, T_{communicator}(n, dt, pd, p) = \emptyset$. Mais uma vez, a aplicação do algoritmo de tradução ignora a declaração dos comunicadores. A declaração do comunicador *com* não tem uma representação directa na abstracção adoptada, no entanto a utilização do mesmo é analisada nas invocações de tarefas para determinar o LET, ou seja, $\forall communicator \in Prog$ em que $n = com$ então $A_{communicator}(n, dt, pd, p) =$

pd.

5.2.1.5 Declaração de Portos

Seja (n, dt, p) a declaração de um porto, onde n é o nome do porto, dt o tipo do porto e o driver de inicialização (ct, ci) e p a posição no ficheiro HTL da declaração do porto, então $\forall port \in Prog, T_{port}(n, dt, p) = \emptyset$. A aplicação do algoritmo de tradução ignora também a declaração dos portos e não é feita qualquer análise na invocação das tarefas onde um porto é utilizado. Na invocação das tarefas os comunicadores são um par com o nome e a instância do mesmo, enquanto um porto é apenas um nome.

5.2.2 Transposição do LET

Na base do algoritmo de tradução está uma implementação do LET. Esta implementação é retratada pela construção de quatro autómatos temporizados predefinidos com parâmetros de entrada. Estes parâmetros permitem que seja produzida uma instanciação de cada invocação de tarefa coerente com o seu LET.

Estes autómatos temporizados são identificados no modelo por *taskTA*, *taskTA_S*, *taskTA_R*, *taskTA_SR*. Existem quatro implementações devido à utilização de portos concretos nas invocações das tarefas. As instanciações do autómato temporizado, *taskTA* representam invocações de tarefas onde apenas são utilizados comunicadores, *taskTA_S* (S de *send*) invocações onde é utilizado um porto concreto na saída, *taskTA_R* (R de *receive*) invocações onde é utilizado um porto concreto na entrada e *taskTA_SR* invocações onde é utilizado um porto concreto na entrada e outro na saída.

No HTL ainda é possível declarar invocações de tarefas com múltiplos portos concretos. Devido à complexidade dessa situação e à sua fraca utilização nos exemplos disponíveis optou-se por deixar este tipo de declaração fora do modelo. Parte-se do princípio que nunca são utilizados mais do que um porto concreto, quer na entrada como na saída dos portos de uma invocação de tarefa.

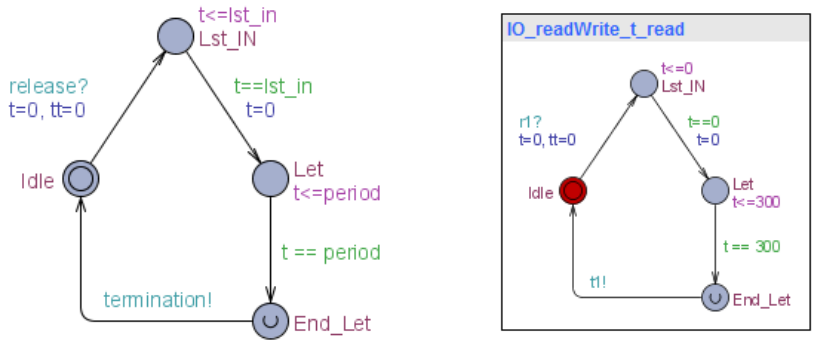
5.2.2.1 Invocações de tarefas

Considera-se (n, ip, op, s, pos) uma invocação de uma tarefa, onde n é o nome da tarefa a invocar, ip o mapeamento dos portos (variáveis) de entrada, op o mapeamento dos portos (variáveis) de saída, s o nome da tarefa pai e pos a posição no ficheiro HTL da declaração da invocação.

5.2.2.2 taskTA

Seja $Port$ o conjunto de todos os portos concretos e (r, t, p, li) um autómato temporizado $taskTA$ onde, r é uma sincronização urgente de $release$, t uma sincronização urgente de $termination$, p o período do LET da tarefa e li o instante em que a última variável de entrada é lida então $\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \notin op, T_{invoke}(n, ip, op, s, pos) = taskTA(r, t, p, li)$.

Figura 5.2: Autómato *taskTA* à esquerda e instanciação à direita



A cada invocação de tarefa, onde não exista nenhum porto concreto, quer nas variáveis de entrada como de saída, corresponde uma instanciação de um autómato $taskTA$ (figura 5.2). Os canais de sincronização urgentes r e t são determinados na declaração do sistema. O nome do canal r de cada instanciação de tarefa é único e produzido de forma enumerada $r1, r2, r3, ...$ O nome do canal t de cada instanciação de tarefa é único, para cada conjunto de autómatos de um modo, e produzido de forma enumerada $t1, t2, t3, ...$ O nome destes canais é agrupado para depois serem utilizados na instanciação do respectivo autómato do módulo.

O instante em que a última variável de entrada *li* é lida determina-se pelo valor máximo da instância de cada comunicador de entrada multiplicado pelo período, no caso de não existir nenhuma variável de entrada então o instante é o zero. O período *p* do LET é a diferença entre o instante da escrita do primeiro porto de saída (se não existir nenhum então é o valor do período do respectivo modo) e *li*.

Este autómato temporizado é constituído (figura 5.2) por dois relógios *t* e *tt*, por um estado inicial *Idle*, um estado *Lst_IN*, um estado *Let* e um estado urgente *End_Let*. A execução de cada instanciação do autómato é activada pela sincronização de *release* existente na transição do estado *Idle* para o estado *Lst_IN*, nesta transição os dois relógios são reinicializados. Do estado *Lst_IN* para o estado *Let* existe uma transição que é activada assim que o relógio *t* é igual a *li*. Nesta transição, o relógio *t* é reinicializado para permitir que a transição, do estado *Let* para o estado *End_Let*, só é activada quando o período *p* é igual ao relógio *t*. Por fim, a transição do estado urgente *End_Let* para o estado inicial *Idle* emite uma sincroniza para o autómato do módulo respectivo.

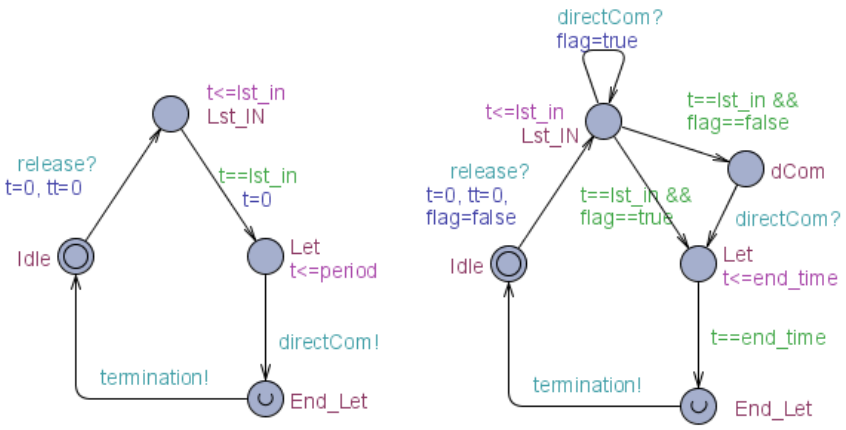
5.2.2.3 taskTA_S

Seja (r, t, dc, p, li) um autómato temporizado *taskTA_S* onde, *r* é uma sincronização urgente de *release*, *t* uma sincronização urgente de *termination*, *dc* uma sincronização urgente de comunicação directa (*directCom*), *p* o período do LET da tarefa e *li* o instante em que a última variável de entrada é lida então $\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \in op, T_{invoke}(n, ip, op, s, pos) = taskTA_S(r, t, dc, p, li)$.

A cada invocação de tarefa, onde exista um porto concreto nas variáveis de saída e nenhum nas de entrada, corresponde uma instanciação de um autómato *taskTA_S*. Este autómato é muito semelhante ao *taskTA*, introduzindo apenas a questão da sincronização para comunicação directa.

A constituição interna (figura 5.3) dos relógios e dos estados mantêm-se trocando a transição do estado *Let* para o estado *End_Let*. Esta transição passa a emitir uma sincronização directa para o respectivo receptor sem qualquer restrição. Isto é, essa transição pode ser activada tenha ou não sido completado o LET "fixo". Uma vez que se desconhece o instante em que uma comunicação directa ocorre introduz-se

Figura 5.3: Autómato *taskTA_S* à esquerda e *taskTA_R* à direita



aqui a possibilidade do fim do LET variar de execução para execução. O que não é permitido, por intermédio de uma invariante no estado *Let*, é que este seja superior ao período determinado pelos comunicadores. Se tiver chegado o instante de ser feita a primeira escrita de variável então o LET termina normalmente, sendo de seguida efectuada a comunicação directa.

5.2.2.4 taskTA_R

Seja (r, t, dc, p, li) um autómato temporizado *taskTA_R* onde, r é uma sincronização urgente de *release*, t uma sincronização urgente de *termination*, dc uma sincronização urgente de comunicação directa (*directCom*), p o período do LET da tarefa e li o instante em que a última variável de entrada é lida então $\forall cp \in Port, \forall invoke \in Prog, cp \in ip, cp \notin op, T_{invoke}(n, ip, op, s, pos) = taskTA\_R(r, t, dc, p, li)$.

A cada invocação de tarefa, onde exista um porto concreto nas variáveis de entrada e nenhum nas de saída, corresponde uma instanciação de um autómato *taskTA_R*. Este autómato é semelhante ao *taskTA*, necessitando de uma sincronização para comunicação directa.

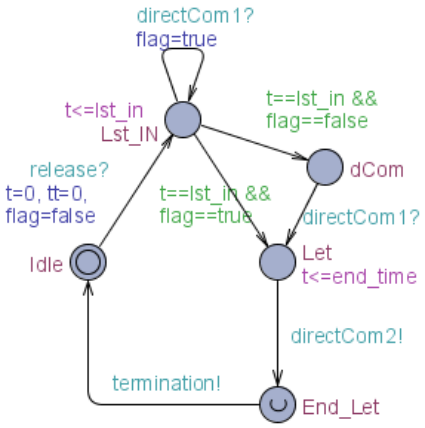
A constituição interna (figura 5.3) altera-se um pouco com a introdução de um estado *dCom* e uma variável $end\_time = li - p$ e outra variável $flag$. Na sincronização do estado *Idle* para *Lst_IN* é acrescentada a actualização da variável $flag = false$ e

passam a existir três transições possíveis partindo do estado *Lst_IN*. Uma transição parte deste estado e vai para ele próprio, servindo para completar a comunicação directa antes do inicio do LET. Ou seja, é a transição que é activada quando a comunicação directa não altera o LET da tarefa. A segunda transição vai do estado *Lst_IN* para o estado *Let* e é activada na sequência da situação anterior no instante $t == lst\_in$. A terceira transição vai do estado *Lst_IN* para o estado *dCom* e é activada quando o instante inicial do LET "fixo" foi atingido mas a comunicação directa ainda não foi activada. A passagem por este estado permite que o estado *Let* só seja atingido assim que a comunicação directa é realizada. Mais uma vez, as alterações deste autómato temporizado permitem um comportamento ligeiramente diferente, permitindo atrasar o instante inicial do LET da tarefa.

5.2.2.5 taskTA_SR

Seja $(r, t, dc1, dc2, p, li)$ um autómato temporizado *taskTA_SR* onde, r é uma sincronização urgente de *release*, t uma sincronização urgente de *termination*, $dc1$ uma sincronização urgente de comunicação directa (*directCom*1), $dc2$ outra sincronização urgente de comunicação directa (*directCom*2), p o período do LET da tarefa e li o instante em que a última variável de entrada é lida então $\forall cp \in Port, \forall invoke \in Prog, cp \in ip, cp \in op, T_{invoke}(n, ip, op, s, pos) = taskTA\_SR(r, t, dc1, dc2, p, li)$.

Figura 5.4: Autómato *taskTA_SR*



A cada invocação de tarefa, onde exista um porto concreto nas variáveis de entrada e outro nas de saída, corresponde uma instanciação de um autómato *taskTA_SR*. Este autómato combina os anteriormente vistos, necessitando de duas

sincronizações para comunicação directa. A primeira sincronização é de recepção enquanto a segunda é de emissão.

A constituição interna deste autómato temporizado (figura 5.4) combina as alterações do autómato *taskTA_S* com as do *taskTA_R* para permitir a modelação da invocação de tarefas com uma comunicação directa na entrada e outra na saída. Mais uma vez, o LET possui um valor "fixo" determinado como no caso do *taskTA*, no entanto este permite variações a cada execução quer antecipando o instante final quer atrasando o instante inicial.

5.2.2.6 Comunicação Directa

A cada instanciação de uma invocação de tarefa cujo resultado é um autómato de tarefa com comunicação directa, é utilizado um ou dois canais de sincronização cujo nome é o nome do próprio porto antecedido do nome do módulo com um _. Este tipo de canal não é utilizado na instanciação do autómato do módulo uma vez que não afecta directamente o comportamento do mesmo. As comunicações directas afectam sim o LET de cada uma das tarefas invocadas tal como ficou descrito.

5.2.3 Módulos e Modos

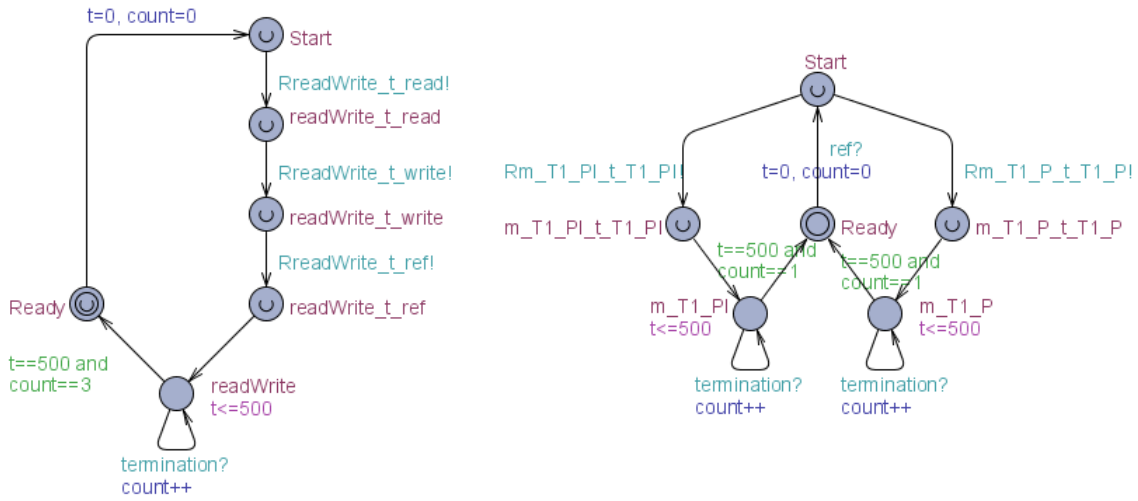
Considerando (n, h, mi, bm, pos) um módulo, onde n é o nome do módulo, h é a lista de *hosts*, mi o modo inicial, bmu o corpo do modulo e pos a posição no ficheiro HTL da declaração do módulo. Seja (ref, rl, tl) um autómato temporizado *moduleTA*, onde ref é um canal de sincronização urgente de refinamento (se existir), rl o conjunto dos canais de sincronização urgentes de *release* de todas as invocações de tarefas de um módulo e tl o conjunto dos canais de sincronização urgentes de *termination* de todas as invocações de tarefas de um módulo, então $\forall module \in Prog, T_{module}(n, h, mi, bm, pos) = moduleTA(ref, rl, tl)$.

Por cada módulo produz-se um autómato temporizado dinamicamente. Ao contrário dos autómatos das tarefas, em que apenas se realiza a instanciação dos diferentes autómatos "fixos" para fazer corresponder os parâmetros de entrada, a cada módulo corresponde um autómato único. Este autómato é posteriormente

instanciado colocando nos parâmetros os canais de sincronização utilizados pelas tarefas invocadas nesse módulo.

Seja $(n, p, refP, bmo, pos)$ um modo, onde n é nome do modo, p o período, $refP$ o programa que refina esse modo (caso exista), bmo o corpo do modo e pos a posição no ficheiro HTL da declaração do modo. Seja (e, t) um subconjunto $subModule$ da declaração do autómato temporizado $moduleTA$, onde e é um conjunto de estados (com invariantes) e t um conjunto de transições (com guardas, actualizações e sincronizações) então $\forall mode \in module, \exists subModule \in moduleTA, T_{mode}(n, p, refP, bmo, pos) = subModule(e, t)$.

Figura 5.5: Autómatos de módulo, com um modo à esquerda e dois modos à direita



Internamente (figura 5.5), apesar de um autómato de módulo ter dois estados típicos *Ready* e *Start*, os restantes estados dependem integralmente dos modos declarados nesse módulo. Um autómato de módulo também possui uma transição típica do estado *Ready* para o estado *Start* onde é reinicializado o relógio local t e o contador *count*. Nesta abstracção foi utilizado um contador único para cada autómato de módulo. Este contabiliza o número de tarefas terminadas (i.e. cuja sincronização de *termination* foi efectuada). Através da combinação deste mecanismo com o relógio local permite-se que cada modo, dentro de um módulo, tenha um período distinto e um número de invocações de tarefas distinto.

A cada invocação de tarefa de um modo faz-se corresponder um estado urgente no *subModule* respectivo, um estado representativo do modo, um conjunto de transições de inicialização dos autómatos de tarefas e transições de controlo de execução do modo. A primeira transição do conjunto é feita do estado *Start* para o estado da primeira invocação de tarefa. A segunda transição é feita do estado da primeira invocação de tarefa para o estado da segunda e assim sucessivamente até se atingir a ultima invocação de tarefa. A última transição, desta sequência, vai então do estado da última invocação de tarefa para o estado do modo. Cada uma destas transições apenas emite uma sincronização para o respectivo autómato de tarefa, exceptuando a última que consiste numa transição simples. Como todos os estados das invocações de tarefas são urgentes isso permite colocar a execução do autómato no estado do respectivo modo sem que decorra tempo.

Para controlar a execução do modo são utilizadas duas transições. Uma delas vai do estado do modo para ele próprio e serve para incrementar o contador a cada sincronização de *termination* das tarefas desse modo. A outra transição vai do estado do modo para o estado $Ready$, esta só é activada quando todas as tarefas desse modo tiverem terminado a sua execução e o período do modo tiver chegado ao fim. Deste modo, cada *subModule* permite criar ramos de execução independentes para o respectivo autómato temporizado *moduleTA*.

5.2.3.1 Refinamentos

Nesta abstracção também se permite que os autómatos temporizados dos módulos possam permanecer no estado $Ready$ sem sequer iniciarem a execução de um dos seus modos. Isto não é válido para todos os autómatos dos módulos, nomeadamente para os do programa principal onde o estado $Ready$ é urgente. Mas para os restantes esta situação verifica-se devido ao refinamento. Assim os autómatos dos módulos do programa principal distinguem-se dos restantes por não serem um refinamento de um modo. Todos os outros são obrigatoriamente o refinamento de algum modo.

Este refinamento traduz-se num canal de sincronização ref partilhado entre diferentes autómatos de módulo. O autómato temporizado *moduleTA* cujo programa refina um dado modo, passa a ter na transição do estado $Ready$ para o estado

Start um receptor de sincronização. O emissor dessa sincronização é colocado no *moduleTA* do modo refinado, mais precisamente na transição do estado da última invocação de tarefa para o estado do modo.

Nesta versão do algoritmo de tradução parte-se do principio que o programa principal é o primeiro programa declarado no ficheiro HTL. Esta situação pode ser revista de futuro para completar uma análise em profundidade dos programas a fim de determinar qual deles é o programa principal. Esta opção facilita também a aplicação do algoritmo por níveis. Isto é, quando no tradutor é fornecida a opção de tradução por níveis (0=todos os programas, 1=programa principal, 2= programa principal + 1 programa, etc.), o algoritmo olha para a declaração sequencial dos programas sem se preocupar realmente com questões de ordem de profundidade. A única verificação que é feita em termos de profundidade das hierarquias consiste em determinar se o programa que refina um modo consta na lista dos programas que vão ser traduzidos. Se consta, a sincronização de refinamento é criada nos respectivos autómatos, se não é ignorada permitindo um correcto comportamento do modelo.

5.3 Algoritmo de Tradução das Propriedades

No apêndice D consta o resultado da aplicação prática do algoritmo de tradução, descrito nesta secção, a um caso de estudo descrito no capítulo 6. Tal como na secção anterior este algoritmo também se baseia na AST apresentada no apêndice A, mas no lugar de produzir a descrição de uma RAT produz a descrição de propriedades a verificar.

Considera-se nesta secção a definição da função P, que aceita um programa *HTL* (de facto a AST do HTL) e que devolve a especificação das propriedades a verificar. Esta função é definida naturalmente por recursividade sobre a estrutura da AST da linguagem HTL. Assim passa-se a definir P para cada caso particular da AST em questão.

5.3.1 Ausência de Bloqueio

Seja $Prog$ o conjunto de todos os programas e df a descrição da propriedade de ausência de bloqueio, então $P_{Prog} = df$. A aplicação do algoritmo a qualquer programa produz sempre a propriedade de ausência de bloqueio ($A[]$ *not deadlock*).

5.3.2 Período dos modos

Seja $(n, p, refP, bmo, pos)$ um modo, onde n é nome do modo, p o período, $refP$ o programa que refina esse modo (caso exista), bmo o corpo do modo e pos a posição no ficheiro HTL da declaração do modo. Seja $(p1, p2)$ a especificação das propriedades vm do período de um modo, então $\forall mode \in Prog, P_{mode}(n, p, refP, bmo, pos) = vm(p1, p2)$.

Seja $moduleTA$ um autómato de módulo e Rat um conjunto de autómatos temporizados, então $\forall mode \in Prog, \exists moduleTA \in Rat$,
$p1 = A[]\ moduleTA.n\ imply\ ((not\ moduleTA.t > p)\ \&\&\ (not\ moduleTA.t < 0))$,
$p2 = moduleTA.n\ \rightarrow\ (moduleTA.Ready\ \&\&\ (moduleTA.t == 0\ ||\ moduleTA.t == p)$.

A primeira propriedade $p1$ indica que sempre que o estado de controlo for o estado do modo, isso implica que o relógio local desse autómato de módulo não seja superior ao período do modo ou inferior a zero. A segunda propriedade $p2$ indica que sempre que é atingido o estado do modo, o estado $Ready$ também é atingido e quando isso acontecer o relógio local ou é zero ou é precisamente o valor do período. A combinação destas duas propriedades permite limitar o período do modo ao intervalo $[0, p]$ e ter a garantia que o valor máximo do período é atingido.

5.3.3 Invocações de tarefas

Seja (n, ip, op, s, pos) uma invocação de tarefa, onde n é o nome da tarefa a invocar, ip o mapeamento dos portos (variáveis) de entrada, op o mapeamento dos portos (variáveis) de saída, s o nome da tarefa pai e pos a posição no ficheiro HTL da declaração da invocação. Seja $(p1, p2)$ a especificação das propriedades vi da invocação de tarefa num modo, então $\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vi(p1, p2)$.

Seja $taskTA_i$ o autómato da tarefa i, $taskTA$ o conjunto dos autómatos de tarefa, $taskState_i$ o estado da invocação da tarefa i, $modeState$ o estado do modo em que a invocação é feita, $moduleTA$ um autómato de módulo e Rat um conjunto de autómatos temporizados, então $\forall i, \exists moduleTA \in Rat, \exists taskTA_i \in TaskTA$,
$p1 = A[]$ ($moduleTA.taskStaste_i$ *imply* (*not* $taskTA_i.Idle$)) &&
($moduleTA.Ready$ *imply* $taskTA_i.Idle$),
$p2 = A[]$ ($taskTA_i.Let$ && $taskTA.tt != 0$) *imply* $moduleTA.modeState$.

A propriedade $p1$ indica que para todas as execuções, sempre que o estado de uma invocação é o estado de controlo, isso implica que o respectivo autómato de tarefa não esteja no estado *Idle* e no respectivo *moduleTA* quando o estado de controlo é o estado *Ready* isso implica que o autómato de tarefa esteja no estado *Idle*. A segunda propriedade indica que sempre que o estado *Let* de um autómato de tarefa é o estado de controlo e o relógio local *tt* é diferente de zero isso implica que a execução do autómato do módulo respectivo esteja no estado representativo do modo onde as tarefas são invocadas.

5.3.4 LET das tarefas

Seja $(p1, p2, p3)$ a especificação das propriedades *vlet* da invocação de tarefa num modo, então $\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vlet(p1, p2, p3)$,
$\forall i, \exists moduleTA \in Rat$,
$p1 = A[]$ ($taskTA_i.Let$ *imply* (*not* $taskTA_i.tt < 0$ && *not* $taskTA_i.tt > p$))
$p2 = A <>$ $moduleTA.modeState$ *imply* ($taskTA_i.Lst\_IN$ && $taskTA_i.tt == 0$)
$p3 = A <>$ $moduleTA.modeState$ *imply* ($taskTA_i.Let$ && $taskTA_i.tt == p$).

A validação do LET é feita com três propriedades distintas. A propriedade $p1$ indica que sempre que o *Let* de uma tarefa é atingido, isso implica que o relógio local *tt* desse autómato de tarefa não seja nem menor que zero nem maior que o período do LET. A propriedade $p2$ indica que sempre que o estado do modo é atingido, inevitavelmente o estado *Lst_IN* é atingido com o relógio local *tt* a zero. A propriedade $p3$ indica que sempre que o estado do modo é atingido, inevitavelmente o estado *Let* é atingido com o relógio *tt* no valor máximo do período da tarefa.

5.3.5 Refinamentos

Para cada refinamento de um modo são geradas duas propriedades que verificam, que sempre que o estado do modo refinado é atingido e o relógio local t é menor que o valor do período, isso implica o autómato do módulo de refinamento não se encontre no estado *Ready*. E sempre que o estado do modo é atingido com o relógio t precisamente com o valor do período isso implica que a dada altura o autómato do módulo de refinamento não vai estar no estado *Ready*.

Tal como na construção do modelo, sempre que é detectado um refinamento num modo é verificado se o programa que o refina se encontra na lista dos programas a traduzir. Isto porque existe a possibilidade de traduzir um programa HTL utilizando apenas o n primeiros programas.

5.4 Conclusão

Neste capítulo foram descritos vários aspectos do funcionamento do tradutor HTL2XTA. Como foi visto, este tradutor enquadra-se numa *Tool-Chain* de HTL que pretende estender a verificação feita. No capítulo seguinte é apresentado um caso de estudo com a devida utilização do tradutor e verificação das propriedades geradas.

Capítulo 6

Validação Experimental

6.1 Caso de Estudo - 3TS

Neste capítulo descrevem-se os resultados da aplicação da *Tool-Chain* apresentada a um caso de estudo típico do HTL, mais precisamente o sistema dos três reservatórios (3TS). Este caso de estudo é apresentado nas duas teses de doutoramento [17] [20], bem como na página oficial do HTL `http://htl.cs.uni-salzburg.at/`.

Figura 6.1: Planta Real do 3TS [29]

Na figura 6.1 é apresentada a planta real do 3TS. Considerando os reservatórios $T1$, $T3$, $T2$ (da esquerda da figura para a direita). Cada com uma válvula de saída

de água, outra válvula entre $T1$ e $T3$, outra entre $T2$ e $T3$, uma bomba $P1$ que injecta água para $T1$ e outra bomba $P2$ para $T2$. O objectivo do sistema de tempo real é manter o nível de água nos reservatórios $T1$ e $T2$ controlando o funcionamento das bombas $P1$ e $P2$.

$T3$ bem como todas as válvulas apenas existem para criar perturbação nos reservatórios $T1$ e $T2$. As válvulas são de controlo manual, pelo que o sistema de tempo real não tem qualquer interacção com as mesmas. O programa HTL regula assim a intensidade de injecção de água, tendo por base dois modos de funcionamento para cada bomba, perante a perturbação existente em cada reservatório. Se não existir perturbação o controlador P (*Proportional*) é utilizado, se existir perturbação o controlador PI (*Proportional Integral*) é utilizado. A modelação deste sistema implica assim quatro cenários possíveis, (1) a bomba $P1$ e $P2$ geridas pelo controlador P, (2) a bomba $P1$ e $P2$ geridas pelo controlador PI, (3) a bomba $P1$ gerida por P e $P2$ por PI e (4) a bomba $P2$ gerida por P e $P1$ por PI.

6.1.1 3ts-simulink

O caso de estudo 3TS possui diversas implementações HTL. Apesar de todas as que foram identificadas terem sido testadas com a aplicação da *Tool-Chain*, apenas se vai apresentar mais detalhadamente uma versão para o simulador da planta e não para a planta real. No apêndice B consta a especificação HTL desta implementação.

Descrevendo sumariamente esta implementação, destaca-se no programa principal *P_3TS* a existência de três módulos. O módulo *IO* no qual são invocadas as tarefas concretas de manipulação dos sensores e actuadores. O módulo $T1$ onde é feita a descrição abstracta da tarefa que manipula a bomba $P1$. Nesse mesmo módulo, o modo *m_T* é refinado pelo programa *P_T*1, concretizando a descrição. E o módulo $T2$, análogo ao módulo $T1$, mas desta feita para a descrição abstracta da manipulação da bomba $P2$.

No programa *P_T*1 é declarado um módulo *T*1*_P_PI* cujos modos concretizam a descrição da manipulação da bomba $P1$ pelos controladores P e PI. Esses controladores são representados pelas tarefas *t_T*1*_P* e *t_T*1*_PI* invocadas em modos

distintos. No programa *P_T2* é declarado um módulo *T2_P_PI*, semelhante ao anterior, cujos modos concretizam a descrição da manipulação da bomba *P2* pelos respectivos controladores *P* e *PI*. Neste programa, esses controladores são representados pelas tarefas *t_T2_P* e *t_T2_PI* invocadas nos modos *m_T2_P* e *m_T2_PI* respectivamente.

As tarefas do programa *P_T1* são refinamentos da tarefa *t_T1* e as tarefas do programa *P_T2* são refinamentos da tarefa *t_T2*. A modelação deste sistema indica que durante a execução das tarefas abstractas só é executada uma das suas tarefas concretas. Verificando-se os quatro cenários anteriormente referidos, (1) a bomba *P1* e *P2* geridas pelo controlador *P* (respectivamente *t_T1_P* e *t_T2_P*), (2) a bomba *P1* e *P2* geridas pelo controlador *PI* (respectivamente *t_T1_PI* e *t_T2_PI*), (3) a bomba *P1* gerida por *P* e *P2* por *PI* (respectivamente *t_T1_P* e *t_T2_PI*) e (4) a bomba *P2* gerida por *P* e *P1* por *PI* (respectivamente *t_T2_P* e *t_T1_PI*).

6.1.2 Modelo Uppaal

A aplicação do tradutor HTL2XTA ao programa anteriormente descrito produz o modelo que se apresenta de seguida e cuja especificação é facultada no apêndice C. Como já foi referido, por cada módulo de um programa é produzido um autómato temporizado, bem como para cada invocação de tarefa. Estes autómatos estão devidamente sincronizados para permitir a modelação do comportamento desejado.

Na figura 6.2 consta, o autómato do módulo *IO*, designado por *sP_3TS_IO*. Uma vez que o módulo em questão só possui um modo, neste autómato apenas existe um caminho na execução do mesmo. Esse caminho consiste na sincronização dos autómatos de tarefas respectivos, *IO_readWrite_t_read* para a invocação da tarefa *t_read*, *IO_readWrite_t_write* para a invocação da tarefa *t_write* e *IO_readWrite_t_ref* para a invocação da tarefa *t_ref*. Após estes autómatos estarem sincronizados, é iniciada a sua execução de forma independente até que seja atingido o término do LET de cada um. À medida que isso acontece são novamente feitas sincronizações com o autómato do módulo. Isto actualiza o contador *count* para garantir que o modo só é reiniciado quando o valor do seu período for atingido e todas as tarefas tiverem acabado o seu LET.

Figura 6.2: Módulo IO

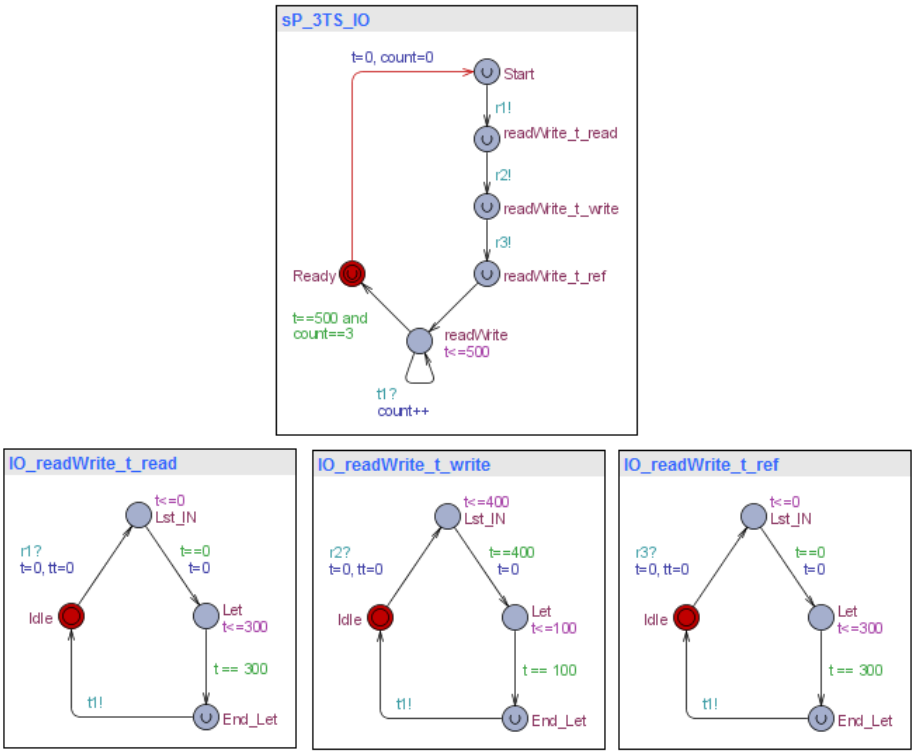
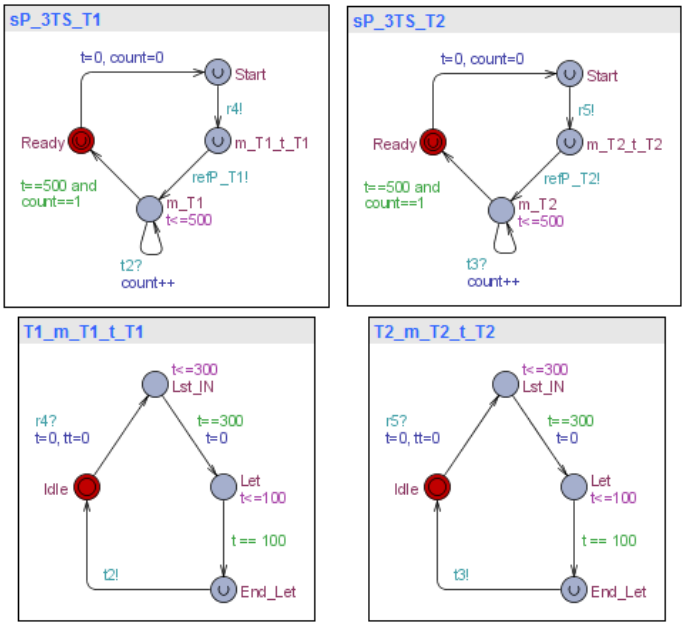


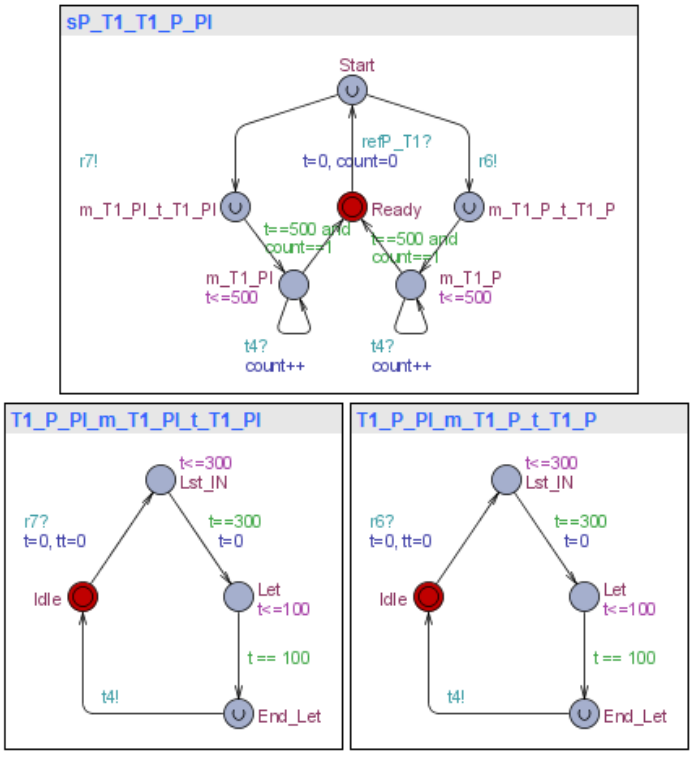
Figura 6.3: Módulo T1 e Módulo T2



A figura 6.3 apresenta os autómatos dos módulos $T1$ (sP_3TS_T1) e $T2$ (sP_3TS_T2), bem como as tarefas invocadas em cada um deles. Cada um destes módulos é constituído por um único modo no qual é invocada uma tarefa. O facto das tarefas invocadas serem abstractas não tem qualquer influência nos seus autómatos de tarefas. Assim a execução da tarefa t_T1 é modelada pelo autómato $T1\_m\_T1\_t\_T1$ e a t_T2 pelo autómato $T2\_m\_T2\_t\_T2$, à semelhança de qualquer outra invocação de tarefa. A questão dos refinamentos de cada um dos modos é apenas tratada ao nível do autómato de módulo com as sincronizações $refP\_T1!$ e $refP\_T2!$.

Os três autómatos de módulo anteriormente apresentados são executados em paralelo. Uma vez que estes módulos pertencem ao programa principal, os seus estados iniciais são urgentes de modo a obrigar que os mesmos sejam sempre executados após o término do período anterior.

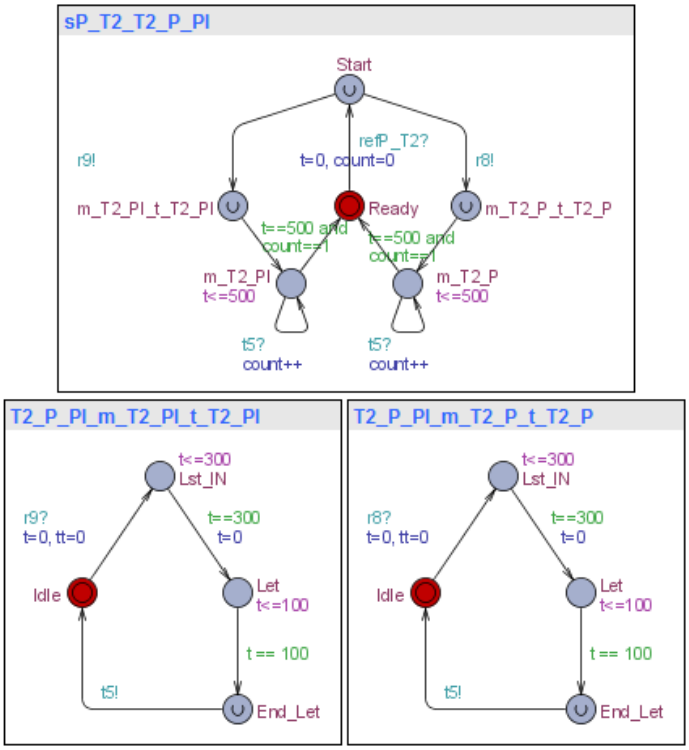
Figura 6.4: Módulo T1_P_PI



Na figura 6.4 é retratado o módulo $T1\_P\_PI$ através do autómato temporizado $sP\_T1\_T1\_P\_PI$, no qual são modelados dois modos distintos. Os modos de um mesmo módulo não podem ser executados em paralelo pelo que existem dois

caminhos distintos neste autómato. Um dos caminhos representa a execução do modo *m_T1_P* com a invocação da tarefa *t_T1_P*, cujo autómato de tarefa é *T1_P_PI_m_T1_P_t_T1_P*. Relembra-se que esta tarefa controla o fluxo da bomba *P1* quando não existe perturbação no reservatório *T1*, assim sempre que o estado *Let* deste autómato for o estado de controlo tem-se a modelação da execução da tarefa tendo em conta o LET da mesma. O caminho alternativo representa a execução do modo *m_T1_PI* com a invocação da tarefa *t_T1_PI*, cujo autómato de tarefa é *T1_P_PI_m_T1_P_t_T1_PI*. Neste caminho, o fluxo da bomba *P1* é gerido tendo em consideração perturbação no reservatório *T1*.

Figura 6.5: Módulo T2_P_PI



Neste autómato de módulo, bem como no equivalente para a bomba *P2* (figura 6.5), o estado inicial não é urgente. Uma vez que estes módulos não pertencem ao programa principal a abstracção adoptada assim o determina. No entanto dada a simplicidade do modelo, torna-se óbvio que a existência das sincronizações de refinamento obriga à execução em contínuo destes autómatos, sendo escolhido um caminho de forma não determinista. No entanto, num sistema mais complexo, pode acontecer que um autómato de módulo não seja obrigatoriamente executado em contínuo, podendo permanecer alguns períodos no estado inicial, daí estes

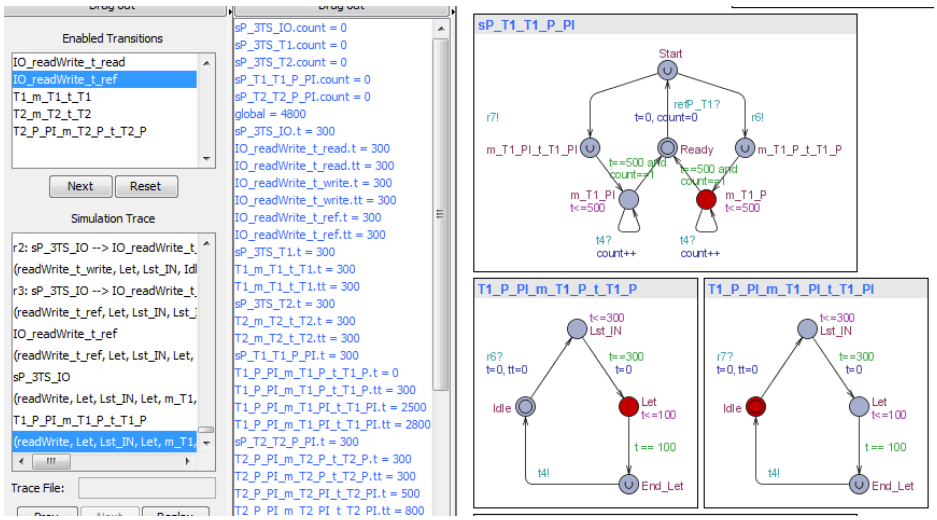
estados iniciais não serem urgentes.

Relativamente à tradução de outros casos de estudos, verificou-se que a especificação do modelo não cresce exponencialmente à medida que a especificação do programa HTL aumenta. No entanto a complexidade do modelo pode tornar-se facilmente intratável pelo verificador de modelos Uppaal. Isto deve-se ao facto da complexidade da verificação do modelo ser exponencial em relação ao crescimento do modelo. É preciso não esquecer que estes autómatos são desdobrados e o verificador necessita explorar todos os caminhos possíveis para produzir um resultado.

6.1.3 Simulação Uppaal

Apesar de não ser o aspecto fundamental, foi sempre feita simulação dos modelos traduzidos. Este caso de estudo até acabou por ser um dos que foi mais alvo de simulação uma vez que serviu de suporte na implementação do tradutor. A complexidade deste modelo é standard e permite que a verificação seja feita com facilidade. No entanto a simulação revela-se mais importante para casos nos quais a complexidade é tal que não se consegue fazer verificação.

Figura 6.6: Simulação do 3TS



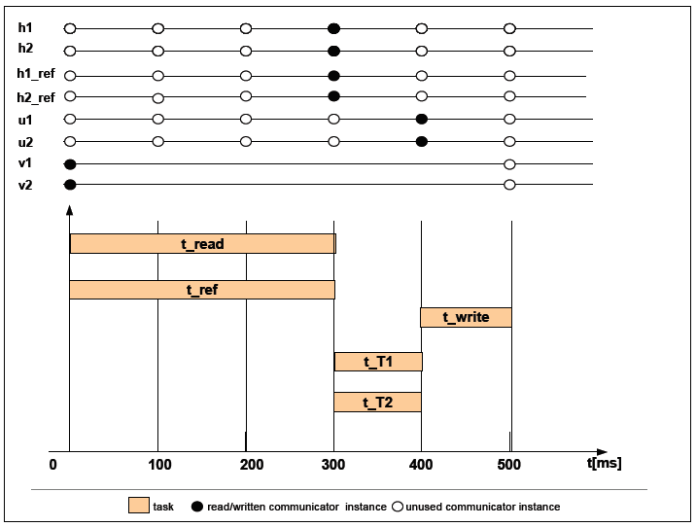
Uma vez que o modelo é construído automaticamente pelo tradutor a simulação

do mesmo não requer grande interacção humana. É sempre necessário abrir o modelo através da interface gráfica e executar o mesmo não deterministicamente ou realizando uma execução passo a passo seleccionando as transições a realizar em cada um deles. No entanto não é preciso mais nenhuma interacção por parte do utilizador. A não ser que, devido à disposição desordenada dos estados, o modelo se torne um quanto ilegível graficamente e que o utilizador queira optar por compor graficamente o mesmo para ter uma maior percepção. A composição gráfica apresentada nos exemplos foi feita manualmente para este caso de estudo.

6.1.4 Verificação Uppaal

No apêndice D é apresentado a especificação das propriedades para o caso de estudo aqui descrito. Esta especificação foi verificada quer por interface gráfica (figura 6.8) como pelo motor do verificador de modelos (passando a redundância). No apêndice E é apresentado um relatório da verificação feita pelo verifyta. Apesar de este relatório não ser considerado um certificado de prova, porque não contem informação concreta sobre as provas das propriedades, o mesmo tem alguma relevância porque é composto pelo output do verifyta. Inclusivamente no caso de uma propriedade não ser validada é possível obter um trace do contra-exemplo, tal como na interface gráfica.

Figura 6.7: Modelo Temporal [29]

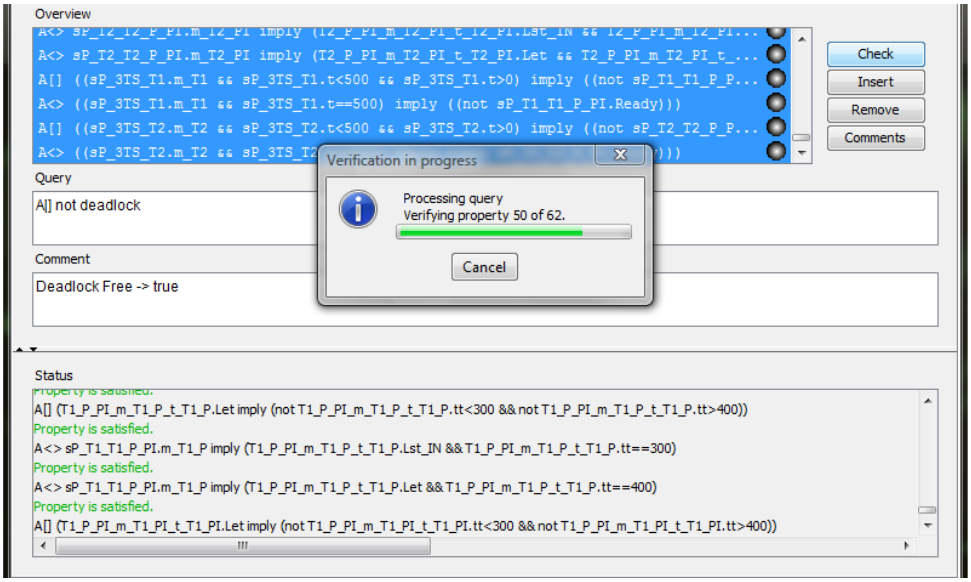


Na figura 6.7 é apresentado o modelo temporal do caso de estudo. A partir dele

é possível determinar propriedades interessantes e verificar as mesmas no Uppaal. No entanto parte dessas propriedades já são automaticamente especificadas pelo HTL2XTA. Por exemplo, a especificação do LET de cada tarefa é feita da linha 163 até à linha 314 do ficheiro no apêndice D.

Nessa especificação constam propriedades que indicam que o LET das tarefas *t_ref* e *t_read* é [0;300], o LET da tarefa *t_write* é [400;500], o LET das restantes tarefas *t_T1*, *t_T2*, *t_T1_P*, *t_T1_PI*, *t_T2_P*, *t_T2_PI* é [300;400]. Comparando estes intervalos com aqueles apresentados na figura 6.7 corrobora-se o valor do LET de cada tarefa. Convém ainda referir que, das 62 propriedades geradas automaticamente, para este caso de estudo, todas foram devidamente validadas.

Figura 6.8: Verificação do 3TS



Em termos práticos, a conjugação da compilação do programa com a validação destas propriedades permite ter mais algumas garantias sobre o comportamento temporal do programa. Para além de se poder dizer que uma implementação deste programa com os devidos WCET é escalonável, pode-se também dizer que garantidamente as tarefas referidas vão ser sempre executadas dentro dos intervalos referidos. Não se sabe fisicamente quando elas são executadas, mas sabe-se que a execução física é sempre feita dentro da janela do LET definido. Mesmo sem uma especificação explícita (feita manualmente), pode-se concluir que as tarefas *t_read* e

Tabela 6.1: Resultados

Ficheiro	Níveis	HTL	Modelo	Verificações	Estados
3TS-simulink.htl	0	75	263	62/62	8'241
	1	75	199	30/30	684
3TS.htl	0	90	271	72/72	23'383
	1	90	207	40/40	1'216
3TS-FE2.htl	0	134	336	106/106	365'587
	1	134	208	42/42	1'584
3TS-PhD.htl	0	111	329	98/98	214'083
	1	111	201	34/34	1'116
flatten_3TS.htl	0	60	203	31/31	448
steer-by-wire.htl	0	873	1043	617/0	N/A
	1	873	690	394/0	N/A

t_ref nunca se sobrepõem à tarefa *t_write*, etc..

6.2 Outros Resultados

Utilizando a versão actual do tradutor (v0.4 de 24/04/2009) conseguiu-se gerar, com sucesso, modelos e propriedades para diversos programas HTL. Na tabela 6.1 constam algumas informações pertinentes sobre esses resultados. Nomeadamente o número de níveis aplicados na tradução (0=todos, 1=programa principal), o número de linhas do respectivo ficheiro HTL, o número de linhas do ficheiro da especificação do modelo, o número de propriedades especificadas contra o número de propriedades correctamente verificadas bem como o número de estados explorados por cada verificação.

De notar que, como seria espectável, sempre que apenas é modelado o programa principal, todos os valores associados ao modelo e à verificação do mesmo tornam-se inferiores. O quanto são inferiores apenas depende do grau de abstracção do programa principal. Exceptuando no programa do *Steer By Wire*, foram verificadas todas as propriedades automaticamente produzidas.

Não se deve esquecer que o número de linhas de um programa HTL não

corresponde ao número de linhas de um programa funcional. Uma vez que a especificação funcional é feita fora do HTL, o facto de um programa ter um número de linhas aparentemente reduzido não implica que se trate de um programa trivial. Por exemplo no caso de estudo retratado existe algum grau de complexidade (aqui considerado intermédio ou standard) e nenhuma das implementações possui mais de 150 linhas de código HTL. Na realidade estes programas coordenam funções que por si podem ser bastante complexas.

É ainda curioso constatar que o programa da planta real $3TS - FE2$, por ter mais dois refinamentos do que o caso de estudo apresentado, tem um aumento considerável na sua complexidade. Revela-se necessário estudar melhor a relação entre as hierarquias e o aumento de complexidade da verificação do modelo para concluir algo que possa beneficiar o tradutor. No entanto não se deve esquecer que esta foi uma primeira abordagem na construção de um tradutor automatizado da linguagem HTL para Uppaal.

6.3 Conclusão

Neste capítulo foi descrito o caso de estudo 3TS e foram apresentados os resultados da aplicação da *Tool-Chain* HTL2XTA ao mesmo caso de estudo bem como, de forma mais superficial, a outros casos de estudo. No capítulo seguinte são apresentadas as conclusões finais da dissertação.

Capítulo 7

Conclusão

7.1 Contributo

O contributo directo da dissertação prende-se com a construção de uma *Tool-Chain* para extensão da verificação de programas HTL. Neste processo foi desenvolvida uma ferramenta de tradução automatizado, designada por HTL2XTA, capaz de construir o modelo Uppaal bem como a especificação de um conjunto de propriedades a partir da especificação de um programa HTL. Em momento oportuno serão publicados os binários e o código fonte do tradutor HTL2XTA no seguinte URL `http://htl2xta.sourceforge.net`.

Relativamente aos contributos indirectos, destacam-se as apresentações feitas nas reuniões do RESCUE. Nestas reuniões foram expostas, o estudo dos mecanismos de verificação para sistemas tempo real (22/10/2008) e a pré-apresentação do tradutor (12/05/2009). Outro contributo, relaciona-se com o apoio dado nas palestras teórico-práticas realizadas pelo Professor Simão Melo de Sousa na Critical Software (12/03/2009) e na universidade do Minho (06/05/2009). Estas palestras foram sobre o Uppaal e a minha participação activa foi feita na componente prática, com a demonstração da ferramenta e elaboração de um documento de apoio (`http://floyd.di.ubi.pt/release/jcarvalho/TutorialPT_Uppaal.pdf`).

Para além destes, ambiciona-se ainda a publicação de alguns artigos relacionados com o tradutor.

7.2 Desafio

Esta dissertação implicou o estudo e descoberta de novas matérias. Antes da dissertação, os conhecimentos sobre a existência do HTL eram nulos. O estudo efectuado é que levou à descoberta da linguagem e à escolha desta como base para um processo de verificação automatizado. Apesar de convencido da possibilidade da construção de um tradutor de HTL para o Uppaal, não existiam garantias sobre a viabilidade de tal tarefa.

O tradutor foi alvo de algumas correcções e adaptações ao longo do desenvolvimento, algumas das quais resultado da detecção de anomalias no modelo produzido quer por simulação como por verificação. No entanto este esforço não está descrito na dissertação.

7.3 Trabalho Futuro

Apesar de o tradutor estar numa versão estável, este não foi alvo de verificação formal. Um dos trabalhos futuros passará, de alguma forma, pela verificação formal da correcção do mesmo. No entanto, este esforço deve ser feito numa versão que não seja alvo de grandes alterações futuras. Uma vez que se verificou que esta versão não é capaz de lidar com programas HTL de maior dimensão ainda é cedo para se ter esse esforço. Neste momento é mais imperativo estudar formas de simplificar o modelo gerado pelo tradutor, ou formas de dividir a complexidade do mesmo por sub-modelos. Mantendo sempre a capacidade de verificar o maior número de propriedades temporais.

Outro trabalho futuro poderá passar por estender o próprio HTL com anotações que permitam introduzir mais algumas informações sobre os *switch*, bem como outras anotações que se possam considerar pertinentes em termos de verificação. Mas antes disso, deve ser estudada a forma como lidar com essas anotações e o impacto das mesmas no modelo. Pode ser interessante juntar algumas anotações para verificação estática, no compilador HTL, e outras para melhorar a tradução do modelo, permitindo a especificação de mais algumas propriedades.No entanto, convêm não esquecer que o modelo actualmente produzido ainda não é completo.

O HTL2XTA traduz a maior parte dos comportamentos dos programas HTL, mas é necessário estudar uma forma simples de lidar com a situação de múltiplos *ports* numa só invocação de tarefa.

Por fim, outro potencial trabalho futuro consiste na transposição deste modelo ou de um modelo muito similar para o Ada/SPARK. Isto é, utilizar os ensinamentos aqui adquiridos para construir um tradutor capaz de abstrair a coordenação das tarefas declaradas nesta linguagem e utilizar o modelo desta coordenação para realizar uma análise temporal estendida.

Apêndice A

Árvore de Sintaxe Abstracta do HTL2XTA

```
1  (**
2   ast.ml of htl2xta Translator (HTL TO UPPAAL)
3   @author Joel Carvalho m2381 at ubi.pt;
4   MSc Student at Universidade da Beira Interior
5   @version Beta 0.4 - 24/04/2009
6  *)
7
8  open Lexing
9
10 type pos =
11         position * position
12
13 type cinit =
14         | Zero
15         | Def_double
16         | TF of bool
17
18 type ctype =
19         | Int
20         | Bool
21         | Char
22         | String
23         | Double
24
25 (*  A port consists of a integer value *)
26 type pt =
27         Port of int
28
29 (* One ip consists of a string *)
30 type ip =
31         IP of string
32
33 (* One name consists of a string *)
34 type name =
35         Name of string
36
37 (* One host consists of a name, a ip and a port *)
38 type host =
39         name * ip * pt
40
41 (* One communicator consists of a name, a type, a period, a initial value and the information about the declaration
       position in the HTL file *)
42 type communicator =
```

```
43          name * ctype * int * cinit * pos
44
45  (*  One port consists of a name, a type, a initial value and the information about the declaration position in the HTL
        file *)
46  type port =
47          name * ctype * cinit * pos
48
49  (* One variable declaration consts a name, a type and a possible initial value *)
50  type var =
51          name * ctype * cinit option
52
53  (* One task consists of a name, a list of input port, a list of states, a list of output port, a possible function name,
        a possible WCET and the information about the declaration position in the HTL file *)
54  type task =
55          name * var list * var list * var list * string option * int option * pos
56
57  (* One port consists of a name and a possible instance *)
58  type aport =
59          name * int option
60
61  (* One invocation consists of a name, a list of input ports, a list of output ports, a possible parent name and the
        information about the declaration position in the HTL file *)
62  type invoke =
63          name * aport list * aport list * string option * pos
64
65  (* One switch consists of a name, a name of condition function and the information about the declaration position in the
        HTL file *)
66  type switch =
67          name * string * pos
68
69  (* One block mode consists of a list of invocations and a list of switch's *)
70  type bmode =
71          invoke list * switch list
72
73  (* One mode consists of a name, a period, a possible name of refined program, a block called bmode and the information
        about the declaration position in the HTL file *)
74  type mode =
75          name * int * string option * bmode * pos
76
77  (* One block module consists of a list of ports, a list of tasks and a list of modes *)
78  type bmodule =
79          port list * task list * mode list
80
81  (*  One module (renamed to module_ because it's a reserved word of ocaml) consists of a name, a list of hosts, a name of
        start mode, a block called bmodule and the information about the declaration position in the HTL file *)
82  type module_ =
83          name * host list * string * bmodule * pos
84
85  (* One program consists of a name, a list of communicators, a list of modules and the information about the declaration
        position in the HTL file *)
86  type program =
87          name * communicator list * module_ list * pos
88
89  (* Abstract Syntax Tree (Ast.prog), consists of a list of programs *)
90  type prog =
91          program list
92  ;;
```

Apêndice B

.HTL - 3TS Java Simulator

```
1  /* 3TS HTL version Working With Java Simulator */
2
3  program P_3TS{
4          communicator
5                  c_double h1 period 100 init c_zero;
6                  c_double h1_ref period 100 init c_zero;
7                  c_double h2 period 100 init c_zero;
8                  c_double h2_ref period 100 init c_zero;
9                  c_double u1 period 100 init c_zero;
10                 c_double u2 period 100 init c_zero;
11                 c_bool v1 period 500 init c_false;
12                 c_bool v2 period 500 init c_false;
13
14         module IO start readWrite{
15                 task t_read input() state() output(c_double p_h1, c_double p_h2,c_bool p_V1, c_bool p_V2) function f_read
                       ;
16                 task t_write input(c_double p_u1:=def_double,c_double p_u2:=def_double) state() output() function f_write
                       ;
17                 task t_ref input() state() output(c_double p_h1_ref, c_double p_h2_ref) function f_ref;
18
19                 mode readWrite period 500{
20                         invoke t_read input() output((h1,3), (h2,3), (v1,1), (v2,1));
21                         invoke t_write input((u1,4), (u2,4)) output();
22                         invoke t_ref input() output((h1_ref,3), (h2_ref,3));
23                 }
24         }
25
26         module T1 start m_T1{
27                 task t_T1 input(c_double p_h1,c_double p_h1_ref) state() output(c_double p_u1);
28
29                 mode m_T1 period 500 program P_T1{
30                         invoke t_T1 input((h1,3),(h1_ref,3)) output((u1,4));
31                 }
32         }
33
34         module T2 start m_T2{
35                 task t_T2 input(c_double v_h2,c_double v_h2_ref) state() output(c_double v_u2);
36
37                 mode m_T2 period 500 program P_T2{
38                         invoke t_T2 input((h2,3),(h2_ref,3)) output((u2,4));
39                 }
40         }
41 }
42
43 program P_T1{
44         module T1_P_PI start m_T1_P{
45                 task t_T1_P input(c_double v_h1,c_double v_h1_ref) state() output(c_double v_u1) function f_T1_P;
```

```
46                  task t_T1_PI input(c_double v_h1,c_double v_h1_ref) state() output(c_double v_u1) function f_T1_PI;
47
48                  mode m_T1_P period 500{
49                          invoke t_T1_P input((h1,3),(h1_ref,3)) output((u1,4)) parent t_T1;
50                          switch(withPerturbation(v1)) m_T1_PI;
51                  }
52
53                  mode m_T1_PI period 500{
54                          invoke t_T1_PI input((h1,3),(h1_ref,3)) output((u1,4)) parent t_T1;
55                          switch(withoutPerturbation(v1)) m_T1_P;
56                  }
57          }
58  }
59
60  program P_T2{
61          module T2_P_PI start m_T2_P{
62                  task t_T2_P input(c_double v_h2,c_double v_h2_ref) state() output(c_double v_u2) function f_T2_P;
63                  task t_T2_PI input(c_double v_h2,c_double v_h2_ref) state() output(c_double v_u2) function f_T2_PI;
64
65                  mode m_T2_P period 500{
66                          invoke t_T2_P input((h2,3),(h2_ref,3)) output((u2,4)) parent t_T2;
67                          switch(withPerturbation(v2)) m_T2_PI;
68                  }
69
70                  mode m_T2_PI period 500{
71                          invoke t_T2_PI input((h2,3),(h2_ref,3)) output((u2,4)) parent t_T2;
72                          switch(withoutPerturbation(v2)) m_T2_P;
73                  }
74          }
75  }
```

Apêndice C

.XTA - 3TS Java Simulator

```
1  /*
2   XTA (Uppaal), generated by htl2xta
3   Joel Carvalho m2381 at ubi.pt
4   MSc Student at Universidade da Beira Interior
5   Version: Beta 0.4 - 24/04/2009
6  */
7
8  process taskTA(urgent chan &release, urgent chan &termination, const int period, const int lst_in){
9    clock
10     t,tt;
11
12   state
13     Idle, Lst_IN{t<=lst_in}, Let{t<=period}, End_Let;
14
15   urgent
16     End_Let;
17
18   init
19     Idle;
20
21   trans
22     Lst_IN -> Let {guard t==lst_in; assign t=0;},
23     Idle -> Lst_IN {sync release?; assign t=0, tt=0;},
24     Let -> End_Let {guard t == period;},
25     End_Let -> Idle {sync termination!;};
26 }
27
28 process taskTA_S(urgent chan &release, urgent chan &termination, urgent chan &directCom, const int period, const int
       lst_in) {
29   clock
30     t,tt;
31
32   state
33     Idle, Lst_IN {t<=lst_in}, Let {t<=period}, End_Let;
34
35   urgent
36     End_Let;
37
38   init
39     Idle;
40
41   trans
42     End_Let -> Idle {sync termination!;},
43     Let -> End_Let {sync directCom!;},
44     Lst_IN -> Let {guard t==lst_in; assign t=0;},
45     Idle -> Lst_IN {sync release?; assign t=0, tt=0;};
46 }
```

```
47
48  process taskTA_R(urgent chan &release, urgent chan &termination, urgent chan &directCom, const int period, const int
        lst_in) {
49    clock
50      t,tt;
51
52    const int
53      end_time = lst_in+period;
54
55    bool
56      flag;
57
58    state
59      Idle, Lst_IN {t<=lst_in}, dCom, Let {t<=end_time}, End_Let;
60
61    urgent
62      End_Let;
63
64    init
65      Idle;
66
67    trans
68      dCom -> Let {sync directCom?;},
69      Lst_IN -> dCom {guard t==lst_in && flag==false;},
70      Lst_IN -> Lst_IN {sync directCom?; assign flag=true;},
71      Lst_IN -> Let {guard t==lst_in && flag==true;},
72      End_Let -> Idle {sync termination!;},
73      Let -> End_Let {guard t==end_time;},
74      Idle -> Lst_IN {sync release?; assign t=0, tt=0, flag=false;};
75  }
76
77  process taskTA_SR(urgent chan &release, urgent chan &termination, urgent chan &directCom1, urgent chan &directCom2,const
        int period, const int lst_in) {
78    clock
79      t,tt;
80
81    const int
82      end_time = lst_in+period;
83
84    bool
85      flag;
86
87    state
88      Idle, Lst_IN {t<=lst_in}, dCom, Let {t<=end_time}, End_Let;
89
90    urgent
91      End_Let;
92
93    init
94      Idle;
95
96    trans
97      dCom -> Let {sync directCom1?;},
98      Lst_IN -> dCom {guard t==lst_in && flag==false;},
99      Lst_IN -> Lst_IN {sync directCom1?; assign flag=true;},
100     Lst_IN -> Let {guard t==lst_in && flag==true;},
101     End_Let -> Idle {sync termination!;},
102     Let -> End_Let {sync directCom2!;},
103     Idle -> Lst_IN {sync release?; assign t=0, tt=0, flag=false;};
104 }
105
106 process P_3TS_IO(urgent chan &RreadWrite_t_read, urgent chan &RreadWrite_t_write, urgent chan &RreadWrite_t_ref, urgent
        chan &termination){
107   clock
108     t;
109
110   int
111     count;
112
113   state
```

```
114       Ready, Start, readWrite{t<=500}, readWrite_t_read, readWrite_t_write, readWrite_t_ref;
115
116     urgent
117       Start, Ready, readWrite_t_read, readWrite_t_write, readWrite_t_ref;
118
119     init
120       Ready;
121
122     trans
123       Ready -> Start{assign t=0, count=0;},
124       readWrite -> readWrite {sync termination?; assign count++;},
125       readWrite -> Ready {guard t==500 and count==3;},
126       Start -> readWrite_t_read {sync RreadWrite_t_read!;},
127       readWrite_t_read -> readWrite_t_write {sync RreadWrite_t_write!;},
128       readWrite_t_write -> readWrite_t_ref {sync RreadWrite_t_ref!;},
129       readWrite_t_ref -> readWrite{};
130   }
131
132   process P_3TS_T1(urgent chan &P_T1, urgent chan &Rm_T1_t_T1, urgent chan &termination){
133     clock
134       t;
135
136     int
137       count;
138
139     state
140       Ready, Start, m_T1{t<=500}, m_T1_t_T1;
141
142     urgent
143       Start, Ready, m_T1_t_T1;
144
145     init
146       Ready;
147
148     trans
149       Ready -> Start{assign t=0, count=0;},
150       m_T1 -> m_T1 {sync termination?; assign count++;},
151       m_T1 -> Ready {guard t==500 and count==1;},
152       Start -> m_T1_t_T1 {sync Rm_T1_t_T1!;},
153       m_T1_t_T1 -> m_T1{sync P_T1!;};
154   }
155
156   process P_3TS_T2(urgent chan &P_T2, urgent chan &Rm_T2_t_T2, urgent chan &termination){
157     clock
158       t;
159
160     int
161       count;
162
163     state
164       Ready, Start, m_T2{t<=500}, m_T2_t_T2;
165
166     urgent
167       Start, Ready, m_T2_t_T2;
168
169     init
170       Ready;
171
172     trans
173       Ready -> Start{assign t=0, count=0;},
174       m_T2 -> m_T2 {sync termination?; assign count++;},
175       m_T2 -> Ready {guard t==500 and count==1;},
176       Start -> m_T2_t_T2 {sync Rm_T2_t_T2!;},
177       m_T2_t_T2 -> m_T2{sync P_T2!;};
178   }
179
180   process P_T1_T1_P_PI(urgent chan &ref, urgent chan &Rm_T1_P_t_T1_P, urgent chan &Rm_T1_PI_t_T1_PI, urgent chan &
          termination){
181     clock
182       t;
```

```
183
184   int
185     count;
186
187   state
188     Ready, Start, m_T1_P{t<=500}, m_T1_PI{t<=500}, m_T1_P_t_T1_P, m_T1_PI_t_T1_PI;
189
190   urgent
191     Start, m_T1_P_t_T1_P, m_T1_PI_t_T1_PI;
192
193   init
194     Ready;
195
196   trans
197     Ready -> Start{sync ref?; assign t=0, count=0;},
198     m_T1_P -> m_T1_P {sync termination?; assign count++;},
199     m_T1_P -> Ready {guard t==500 and count==1;},
200     m_T1_PI -> m_T1_PI {sync termination?; assign count++;},
201     m_T1_PI -> Ready {guard t==500 and count==1;},
202     Start -> m_T1_P_t_T1_P {sync Rm_T1_P_t_T1_P!;},
203     m_T1_P_t_T1_P -> m_T1_P{},
204     Start -> m_T1_PI_t_T1_PI {sync Rm_T1_PI_t_T1_PI!;},
205     m_T1_PI_t_T1_PI -> m_T1_PI{};
206 }
207
208 process P_T2_T2_P_PI(urgent chan &ref, urgent chan &Rm_T2_P_t_T2_P, urgent chan &Rm_T2_PI_t_T2_PI, urgent chan &
      termination){
209   clock
210     t;
211
212   int
213     count;
214
215   state
216     Ready, Start, m_T2_P{t<=500}, m_T2_PI{t<=500}, m_T2_P_t_T2_P, m_T2_PI_t_T2_PI;
217
218   urgent
219     Start, m_T2_P_t_T2_P, m_T2_PI_t_T2_PI;
220
221   init
222     Ready;
223
224   trans
225     Ready -> Start{sync ref?; assign t=0, count=0;},
226     m_T2_P -> m_T2_P {sync termination?; assign count++;},
227     m_T2_P -> Ready {guard t==500 and count==1;},
228     m_T2_PI -> m_T2_PI {sync termination?; assign count++;},
229     m_T2_PI -> Ready {guard t==500 and count==1;},
230     Start -> m_T2_P_t_T2_P {sync Rm_T2_P_t_T2_P!;},
231     m_T2_P_t_T2_P -> m_T2_P{},
232     Start -> m_T2_PI_t_T2_PI {sync Rm_T2_PI_t_T2_PI!;},
233     m_T2_PI_t_T2_PI -> m_T2_PI{};
234 }
235
236 clock
237   global;
238
239 urgent chan
240   refP_T1, refP_T2, r1, r2, r3, r4, r5, r6, r7, r8, r9, t1, t2, t3, t4, t5;
241
242 IO_readWrite_t_read = taskTA(r1,t1,300,0);
243 IO_readWrite_t_write = taskTA(r2,t1,100,400);
244 IO_readWrite_t_ref = taskTA(r3,t1,300,0);
245 sP_3TS_IO = P_3TS_IO(r1,r2,r3,t1);
246
247 T1_m_T1_t_T1 = taskTA(r4,t2,100,300);
248 sP_3TS_T1 = P_3TS_T1(refP_T1,r4,t2);
249
250 T2_m_T2_t_T2 = taskTA(r5,t3,100,300);
251 sP_3TS_T2 = P_3TS_T2(refP_T2,r5,t3);
```

```
252
253  T1_P_PI_m_T1_P_t_T1_P = taskTA(r6,t4,100,300);
254  T1_P_PI_m_T1_PI_t_T1_PI = taskTA(r7,t4,100,300);
255  sP_T1_T1_P_PI = P_T1_T1_P_PI(refP_T1,r6,r7,t4);
256
257  T2_P_PI_m_T2_P_t_T2_P = taskTA(r8,t5,100,300);
258  T2_P_PI_m_T2_PI_t_T2_PI = taskTA(r9,t5,100,300);
259  sP_T2_T2_P_PI = P_T2_T2_P_PI(refP_T2,r8,r9,t5);
260
261
262  system
263    sP_3TS_IO, IO_readWrite_t_read, IO_readWrite_t_write, IO_readWrite_t_ref, sP_3TS_T1, T1_m_T1_t_T1, sP_3TS_T2,
         T2_m_T2_t_T2, sP_T1_T1_P_PI, T1_P_PI_m_T1_P_t_T1_P, T1_P_PI_m_T1_PI_t_T1_PI, sP_T2_T2_P_PI, T2_P_PI_m_T2_P_t_T2_P
         , T2_P_PI_m_T2_PI_t_T2_PI;
```

Apêndice D

.Q - 3TS Java Simulator

```
/*
 Queries File (Uppaal), generated by htl2xta
 Joel Carvalho m2381 at ubi.pt
 MSc Student at Universidade da Beira Interior
 Version: Beta 0.4 - 24/04/2009
*/

/*
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))

/*
P2 mode readWrite period 500 @ Line 19 -> true
*/
sP_3TS_IO.readWrite   > (sP_3TS_IO.Ready && (sP_3TS_IO.t==0  ||  sP_3TS_IO.t==500))

/*
P1 mode m_T1 period 500 @ Line 29 -> true
*/
A[] sP_3TS_T1.m_T1 imply ((not sP_3TS_T1.t>500)  && (not sP_3TS_T1.t<0))

/*
P2 mode m_T1 period 500 @ Line 29 -> true
*/
sP_3TS_T1.m_T1   > (sP_3TS_T1.Ready && (sP_3TS_T1.t==0  ||  sP_3TS_T1.t==500))

/*
P1 mode m_T2 period 500 @ Line 37 -> true
*/
A[] sP_3TS_T2.m_T2 imply ((not sP_3TS_T2.t>500)  && (not sP_3TS_T2.t<0))

/*
P2 mode m_T2 period 500 @ Line 37 -> true
*/
sP_3TS_T2.m_T2   > (sP_3TS_T2.Ready && (sP_3TS_T2.t==0  ||  sP_3TS_T2.t==500))

/*
P1 mode m_T1_P period 500 @ Line 48 -> true
*/
A[] sP_T1_T1_P_PI.m_T1_P imply ((not sP_T1_T1_P_PI.t>500)  && (not sP_T1_T1_P_PI.t<0))

```

```
48  /*
49  P2 mode m_T1_P period 500 @ Line 48 -> true
50  */
51  sP_T1_T1_P_PI.m_T1_P   > (sP_T1_T1_P_PI.Ready && (sP_T1_T1_P_PI.t==0  ||  sP_T1_T1_P_PI.t==500))
52
53  /*
54  P1 mode m_T1_PI period 500 @ Line 53 -> true
55  */
56  A[] sP_T1_T1_P_PI.m_T1_PI imply ((not sP_T1_T1_P_PI.t>500)  && (not sP_T1_T1_P_PI.t<0))
57
58  /*
59  P2 mode m_T1_PI period 500 @ Line 53 -> true
60  */
61  sP_T1_T1_P_PI.m_T1_PI   > (sP_T1_T1_P_PI.Ready && (sP_T1_T1_P_PI.t==0  ||  sP_T1_T1_P_PI.t==500))
62
63  /*
64  P1 mode m_T2_P period 500 @ Line 65 -> true
65  */
66  A[] sP_T2_T2_P_PI.m_T2_P imply ((not sP_T2_T2_P_PI.t>500)  && (not sP_T2_T2_P_PI.t<0))
67
68  /*
69  P2 mode m_T2_P period 500 @ Line 65 -> true
70  */
71  sP_T2_T2_P_PI.m_T2_P   > (sP_T2_T2_P_PI.Ready && (sP_T2_T2_P_PI.t==0  ||  sP_T2_T2_P_PI.t==500))
72
73  /*
74  P1 mode m_T2_PI period 500 @ Line 70 -> true
75  */
76  A[] sP_T2_T2_P_PI.m_T2_PI imply ((not sP_T2_T2_P_PI.t>500)  && (not sP_T2_T2_P_PI.t<0))
77
78  /*
79  P2 mode m_T2_PI period 500 @ Line 70 -> true
80  */
81  sP_T2_T2_P_PI.m_T2_PI   > (sP_T2_T2_P_PI.Ready && (sP_T2_T2_P_PI.t==0  ||  sP_T2_T2_P_PI.t==500))
82
83  /*
84  P1 mode readWrite in module IO invokes: t_read @ Line 20, t_write @ Line 21, t_ref @ Line 22 -> true
85  */
86  A[] (sP_3TS_IO.readWrite_t_read imply (not IO_readWrite_t_read.Idle)) && (sP_3TS_IO.readWrite_t_write imply (not
        IO_readWrite_t_write.Idle)) && (sP_3TS_IO.readWrite_t_ref imply (not IO_readWrite_t_ref.Idle)) &&  (sP_3TS_IO.Ready
         imply (IO_readWrite_t_read.Idle && IO_readWrite_t_write.Idle && IO_readWrite_t_ref.Idle))
87
88  /*
89  P1 mode m_T1 in module T1 invokes: t_T1 @ Line 30 -> true
90  */
91  A[] (sP_3TS_T1.m_T1_t_T1 imply (not T1_m_T1_t_T1.Idle)) &&  (sP_3TS_T1.Ready imply (T1_m_T1_t_T1.Idle))
92
93  /*
94  P1 mode m_T2 in module T2 invokes: t_T2 @ Line 38 -> true
95  */
96  A[] (sP_3TS_T2.m_T2_t_T2 imply (not T2_m_T2_t_T2.Idle)) &&  (sP_3TS_T2.Ready imply (T2_m_T2_t_T2.Idle))
97
98  /*
99  P1 mode m_T1_P in module T1_P_PI invokes: t_T1_P @ Line 49 -> true
100 */
101 A[] (sP_T1_T1_P_PI.m_T1_P_t_T1_P imply (not T1_P_PI_m_T1_P_t_T1_P.Idle)) &&  (sP_T1_T1_P_PI.Ready imply (
        T1_P_PI_m_T1_P_t_T1_P.Idle))
102
103 /*
104 P1 mode m_T1_PI in module T1_P_PI invokes: t_T1_PI @ Line 54 -> true
105 */
106 A[] (sP_T1_T1_P_PI.m_T1_PI_t_T1_PI imply (not T1_P_PI_m_T1_PI_t_T1_PI.Idle)) &&  (sP_T1_T1_P_PI.Ready imply (
        T1_P_PI_m_T1_PI_t_T1_PI.Idle))
107
108 /*
109 P1 mode m_T2_P in module T2_P_PI invokes: t_T2_P @ Line 66 -> true
110 */
111 A[] (sP_T2_T2_P_PI.m_T2_P_t_T2_P imply (not T2_P_PI_m_T2_P_t_T2_P.Idle)) &&  (sP_T2_T2_P_PI.Ready imply (
        T2_P_PI_m_T2_P_t_T2_P.Idle))
112
```

```
113 /*
114 P1 mode m_T2_PI in module T2_P_PI invokes: t_T2_PI @ Line 71 -> true
115 */
116 A[] (sP_T2_T2_P_PI.m_T2_PI_t_T2_PI imply (not T2_P_PI_m_T2_PI_t_T2_PI.Idle)) &&  (sP_T2_T2_P_PI.Ready imply (
        T2_P_PI_m_T2_PI_t_T2_PI.Idle))
117
118 /*
119 P2 mode readWrite in module IO invokes: t_read @ Line 20 -> true
120 */
121 A[] (IO_readWrite_t_read.Let && IO_readWrite_t_read.tt!=0) imply sP_3TS_IO.readWrite
122
123 /*
124 P2 mode readWrite in module IO invokes: t_write @ Line 21 -> true
125 */
126 A[] (IO_readWrite_t_write.Let && IO_readWrite_t_write.tt!=0) imply sP_3TS_IO.readWrite
127
128 /*
129 P2 mode readWrite in module IO invokes: t_ref @ Line 22 -> true
130 */
131 A[] (IO_readWrite_t_ref.Let && IO_readWrite_t_ref.tt!=0) imply sP_3TS_IO.readWrite
132
133 /*
134 P2 mode m_T1 in module T1 invokes: t_T1 @ Line 30 -> true
135 */
136 A[] (T1_m_T1_t_T1.Let && T1_m_T1_t_T1.tt!=0) imply sP_3TS_T1.m_T1
137
138 /*
139 P2 mode m_T2 in module T2 invokes: t_T2 @ Line 38 -> true
140 */
141 A[] (T2_m_T2_t_T2.Let && T2_m_T2_t_T2.tt!=0) imply sP_3TS_T2.m_T2
142
143 /*
144 P2 mode m_T1_P in module T1_P_PI invokes: t_T1_P @ Line 49 -> true
145 */
146 A[] (T1_P_PI_m_T1_P_t_T1_P.Let && T1_P_PI_m_T1_P_t_T1_P.tt!=0) imply sP_T1_T1_P_PI.m_T1_P
147
148 /*
149 P2 mode m_T1_PI in module T1_P_PI invokes: t_T1_PI @ Line 54 -> true
150 */
151 A[] (T1_P_PI_m_T1_PI_t_T1_PI.Let && T1_P_PI_m_T1_PI_t_T1_PI.tt!=0) imply sP_T1_T1_P_PI.m_T1_PI
152
153 /*
154 P2 mode m_T2_P in module T2_P_PI invokes: t_T2_P @ Line 66 -> true
155 */
156 A[] (T2_P_PI_m_T2_P_t_T2_P.Let && T2_P_PI_m_T2_P_t_T2_P.tt!=0) imply sP_T2_T2_P_PI.m_T2_P
157
158 /*
159 P2 mode m_T2_PI in module T2_P_PI invokes: t_T2_PI @ Line 71 -> true
160 */
161 A[] (T2_P_PI_m_T2_PI_t_T2_PI.Let && T2_P_PI_m_T2_PI_t_T2_PI.tt!=0) imply sP_T2_T2_P_PI.m_T2_PI
162
163 /*
164 P1 Let of t_read = [0;300] @ Line 20 ->true
165 */
166 A[] (IO_readWrite_t_read.Let imply (not IO_readWrite_t_read.tt<0 && not IO_readWrite_t_read.tt>300))
167
168 /*
169 P2 Let of t_read = [0;300] @ Line 20 ->true
170 #Last input port of t_read in mode readWrite inevitably reach 0
171 */
172 A<> sP_3TS_IO.readWrite imply (IO_readWrite_t_read.Lst_IN && IO_readWrite_t_read.tt==0)
173
174 /*
175 P3 Let of t_read = [0;300] @ Line 20 ->true
176 #Let of t_read in mode readWrite inevitably reach 300
177 */
178 A<> sP_3TS_IO.readWrite imply (IO_readWrite_t_read.Let && IO_readWrite_t_read.tt==300)
179
180 /*
181 P1 Let of t_write = [400;500] @ Line 21 ->true
```

```
182 */
183 A[] (IO_readWrite_t_write.Let imply (not IO_readWrite_t_write.tt<400 && not IO_readWrite_t_write.tt>500))
184
185 /*
186 P2 Let of t_write = [400;500] @ Line 21 ->true
187 #Last input port of t_write in mode readWrite inevitably reach 400
188 */
189 A<> sP_3TS_IO.readWrite imply (IO_readWrite_t_write.Lst_IN && IO_readWrite_t_write.tt==400)
190
191 /*
192 P3 Let of t_write = [400;500] @ Line 21 ->true
193 #Let of t_write in mode readWrite inevitably reach 500
194 */
195 A<> sP_3TS_IO.readWrite imply (IO_readWrite_t_write.Let && IO_readWrite_t_write.tt==500)
196
197 /*
198 P1 Let of t_ref = [0;300] @ Line 22 ->true
199 */
200 A[] (IO_readWrite_t_ref.Let imply (not IO_readWrite_t_ref.tt<0 && not IO_readWrite_t_ref.tt>300))
201
202 /*
203 P2 Let of t_ref = [0;300] @ Line 22 ->true
204 #Last input port of t_ref in mode readWrite inevitably reach 0
205 */
206 A<> sP_3TS_IO.readWrite imply (IO_readWrite_t_ref.Lst_IN && IO_readWrite_t_ref.tt==0)
207
208 /*
209 P3 Let of t_ref = [0;300] @ Line 22 ->true
210 #Let of t_ref in mode readWrite inevitably reach 300
211 */
212 A<> sP_3TS_IO.readWrite imply (IO_readWrite_t_ref.Let && IO_readWrite_t_ref.tt==300)
213
214 /*
215 P1 Let of t_T1 = [300;400] @ Line 30 ->true
216 */
217 A[] (T1_m_T1_t_T1.Let imply (not T1_m_T1_t_T1.tt<300 && not T1_m_T1_t_T1.tt>400))
218
219 /*
220 P2 Let of t_T1 = [300;400] @ Line 30 ->true
221 #Last input port of t_T1 in mode m_T1 inevitably reach 300
222 */
223 A<> sP_3TS_T1.m_T1 imply (T1_m_T1_t_T1.Lst_IN && T1_m_T1_t_T1.tt==300)
224
225 /*
226 P3 Let of t_T1 = [300;400] @ Line 30 ->true
227 #Let of t_T1 in mode m_T1 inevitably reach 400
228 */
229 A<> sP_3TS_T1.m_T1 imply (T1_m_T1_t_T1.Let && T1_m_T1_t_T1.tt==400)
230
231 /*
232 P1 Let of t_T2 = [300;400] @ Line 38 ->true
233 */
234 A[] (T2_m_T2_t_T2.Let imply (not T2_m_T2_t_T2.tt<300 && not T2_m_T2_t_T2.tt>400))
235
236 /*
237 P2 Let of t_T2 = [300;400] @ Line 38 ->true
238 #Last input port of t_T2 in mode m_T2 inevitably reach 300
239 */
240 A<> sP_3TS_T2.m_T2 imply (T2_m_T2_t_T2.Lst_IN && T2_m_T2_t_T2.tt==300)
241
242 /*
243 P3 Let of t_T2 = [300;400] @ Line 38 ->true
244 #Let of t_T2 in mode m_T2 inevitably reach 400
245 */
246 A<> sP_3TS_T2.m_T2 imply (T2_m_T2_t_T2.Let && T2_m_T2_t_T2.tt==400)
247
248 /*
249 P1 Let of t_T1_P = [300;400] @ Line 49 ->true
250 */
251 A[] (T1_P_PI_m_T1_P_t_T1_P.Let imply (not T1_P_PI_m_T1_P_t_T1_P.tt<300 && not T1_P_PI_m_T1_P_t_T1_P.tt>400))
```

```
252
253 /*
254 P2 Let of t_T1_P = [300;400] @ Line 49 ->true
255 #Last input port of t_T1_P in mode m_T1_P inevitably reach 300
256 */
257 A<> sP_T1_T1_P_PI.m_T1_P imply (T1_P_PI_m_T1_P_t_T1_P.Lst_IN && T1_P_PI_m_T1_P_t_T1_P.tt==300)
258
259 /*
260 P3 Let of t_T1_P = [300;400] @ Line 49 ->true
261 #Let of t_T1_P in mode m_T1_P inevitably reach 400
262 */
263 A<> sP_T1_T1_P_PI.m_T1_P imply (T1_P_PI_m_T1_P_t_T1_P.Let && T1_P_PI_m_T1_P_t_T1_P.tt==400)
264
265 /*
266 P1 Let of t_T1_PI = [300;400] @ Line 54 ->true
267 */
268 A[] (T1_P_PI_m_T1_PI_t_T1_PI.Let imply (not T1_P_PI_m_T1_PI_t_T1_PI.tt<300 && not T1_P_PI_m_T1_PI_t_T1_PI.tt>400))
269
270 /*
271 P2 Let of t_T1_PI = [300;400] @ Line 54 ->true
272 #Last input port of t_T1_PI in mode m_T1_PI inevitably reach 300
273 */
274 A<> sP_T1_T1_P_PI.m_T1_PI imply (T1_P_PI_m_T1_PI_t_T1_PI.Lst_IN && T1_P_PI_m_T1_PI_t_T1_PI.tt==300)
275
276 /*
277 P3 Let of t_T1_PI = [300;400] @ Line 54 ->true
278 #Let of t_T1_PI in mode m_T1_PI inevitably reach 400
279 */
280 A<> sP_T1_T1_P_PI.m_T1_PI imply (T1_P_PI_m_T1_PI_t_T1_PI.Let && T1_P_PI_m_T1_PI_t_T1_PI.tt==400)
281
282 /*
283 P1 Let of t_T2_P = [300;400] @ Line 66 ->true
284 */
285 A[] (T2_P_PI_m_T2_P_t_T2_P.Let imply (not T2_P_PI_m_T2_P_t_T2_P.tt<300 && not T2_P_PI_m_T2_P_t_T2_P.tt>400))
286
287 /*
288 P2 Let of t_T2_P = [300;400] @ Line 66 ->true
289 #Last input port of t_T2_P in mode m_T2_P inevitably reach 300
290 */
291 A<> sP_T2_T2_P_PI.m_T2_P imply (T2_P_PI_m_T2_P_t_T2_P.Lst_IN && T2_P_PI_m_T2_P_t_T2_P.tt==300)
292
293 /*
294 P3 Let of t_T2_P = [300;400] @ Line 66 ->true
295 #Let of t_T2_P in mode m_T2_P inevitably reach 400
296 */
297 A<> sP_T2_T2_P_PI.m_T2_P imply (T2_P_PI_m_T2_P_t_T2_P.Let && T2_P_PI_m_T2_P_t_T2_P.tt==400)
298
299 /*
300 P1 Let of t_T2_PI = [300;400] @ Line 71 ->true
301 */
302 A[] (T2_P_PI_m_T2_PI_t_T2_PI.Let imply (not T2_P_PI_m_T2_PI_t_T2_PI.tt<300 && not T2_P_PI_m_T2_PI_t_T2_PI.tt>400))
303
304 /*
305 P2 Let of t_T2_PI = [300;400] @ Line 71 ->true
306 #Last input port of t_T2_PI in mode m_T2_PI inevitably reach 300
307 */
308 A<> sP_T2_T2_P_PI.m_T2_PI imply (T2_P_PI_m_T2_PI_t_T2_PI.Lst_IN && T2_P_PI_m_T2_PI_t_T2_PI.tt==300)
309
310 /*
311 P3 Let of t_T2_PI = [300;400] @ Line 71 ->true
312 #Let of t_T2_PI in mode m_T2_PI inevitably reach 400
313 */
314 A<> sP_T2_T2_P_PI.m_T2_PI imply (T2_P_PI_m_T2_PI_t_T2_PI.Let && T2_P_PI_m_T2_PI_t_T2_PI.tt==400)
315
316 /*
317 P1 mode m_T1 from module T1 of program P_3TS is refined by ProgramP_T1 @ Line 29 -> true
318 */
319 A[] ((sP_3TS_T1.m_T1 && sP_3TS_T1.t<500 && sP_3TS_T1.t>0) imply ((not sP_T1_T1_P_PI.Ready)))
320
321 /*
```

```
322 P2 mode m_T1 from module T1 of program P_3TS is refined by ProgramP_T1 @ Line 29 -> true
323 */
324 A<> ((sP_3TS_T1.m_T1 && sP_3TS_T1.t==500) imply ((not sP_T1_T1_P_PI.Ready)))
325
326 /*
327 P1 mode m_T2 from module T2 of program P_3TS is refined by ProgramP_T2 @ Line 37 -> true
328 */
329 A[] ((sP_3TS_T2.m_T2 && sP_3TS_T2.t<500 && sP_3TS_T2.t>0) imply ((not sP_T2_T2_P_PI.Ready)))
330
331 /*
332 P2 mode m_T2 from module T2 of program P_3TS is refined by ProgramP_T2 @ Line 37 -> true
333 */
334 A<> ((sP_3TS_T2.m_T2 && sP_3TS_T2.t==500) imply ((not sP_T2_T2_P_PI.Ready)))
```

Apêndice E

.VRF - 3TS Java Simulator

```
1  Options for the verification:
2    Generating no trace
3    Search order is breadth first
4    Using aggressive space optimisation
5    Using reuse optimisation
6    Seed is 1242044147
7    State space representation uses minimal constraint
         systems
8
9  Verifying property 1 at line 11
10     Property is satisfied.
11     States stored : 3551 states
12     States explored : 8241 states
13
14 Verifying property 2 at line 16
15     Property is satisfied.
16     States stored : 3551 states
17     States explored : 8241 states
18
19 Verifying property 3 at line 21
20     Property is satisfied.
21     States stored : 3551 states
22     States explored : 8241 states
23
24 Verifying property 4 at line 26
25     Property is satisfied.
26     States stored : 3551 states
27     States explored : 8241 states
28
29 Verifying property 5 at line 31
30     Property is satisfied.
31     States stored : 3551 states
32     States explored : 8241 states
33
34 Verifying property 6 at line 36
35     Property is satisfied.
36     States stored : 3551 states
37     States explored : 8241 states
38
39 Verifying property 7 at line 41
40     Property is satisfied.
41     States stored : 3551 states
42     States explored : 8241 states
43
44 Verifying property 8 at line 46
45     Property is satisfied.
46     States stored : 3551 states
47     States explored : 8241 states
48
49 Verifying property 9 at line 51
50     Property is satisfied.
51     States stored : 3551 states
52     States explored : 8241 states
53
54 Verifying property 10 at line 56
55     Property is satisfied.
56     States stored : 3551 states
57     States explored : 8241 states
58
59 Verifying property 11 at line 61
60     Property is satisfied.
61     States stored : 3551 states
62     States explored : 8241 states
63
64 Verifying property 12 at line 66
65     Property is satisfied.
66     States stored : 3551 states
67     States explored : 8241 states
68
69 Verifying property 13 at line 71
70     Property is satisfied.
71     States stored : 3551 states
72     States explored : 8241 states
73
74 Verifying property 14 at line 76
75     Property is satisfied.
76     States stored : 3551 states
77     States explored : 8241 states
78
79 Verifying property 15 at line 81
80     Property is satisfied.
81     States stored : 3551 states
82     States explored : 8241 states
83
84 Verifying property 16 at line 86
85     Property is satisfied.
86     States stored : 3551 states
87     States explored : 8241 states
88
89 Verifying property 17 at line 91
90     Property is satisfied.
91     States stored : 3551 states
```

```
92      States explored : 0 states
93
94  Verifying property 18 at line 96
95      Property is satisfied.
96      States stored : 3551 states
97      States explored : 0 states
98
99  Verifying property 19 at line 101
100     Property is satisfied.
101     States stored : 3551 states
102     States explored : 0 states
103
104 Verifying property 20 at line 106
105     Property is satisfied.
106     States stored : 3551 states
107     States explored : 0 states
108
109 Verifying property 21 at line 111
110     Property is satisfied.
111     States stored : 3551 states
112     States explored : 0 states
113
114 Verifying property 22 at line 116
115     Property is satisfied.
116     States stored : 3551 states
117     States explored : 0 states
118
119 Verifying property 23 at line 121
120     Property is satisfied.
121     States stored : 3551 states
122     States explored : 0 states
123
124 Verifying property 24 at line 126
125     Property is satisfied.
126     States stored : 3551 states
127     States explored : 0 states
128
129 Verifying property 25 at line 131
130     Property is satisfied.
131     States stored : 3551 states
132     States explored : 0 states
133
134 Verifying property 26 at line 136
135     Property is satisfied.
136     States stored : 3551 states
137     States explored : 0 states
138
139 Verifying property 27 at line 141
140     Property is satisfied.
141     States stored : 3551 states
142     States explored : 0 states
143
144 Verifying property 28 at line 146
145     Property is satisfied.
146     States stored : 3551 states
147     States explored : 0 states
148
149 Verifying property 29 at line 151
150     Property is satisfied.
151     States stored : 3551 states
152     States explored : 0 states
153
154 Verifying property 30 at line 156
155     Property is satisfied.
156     States stored : 3551 states
157     States explored : 0 states
158
159 Verifying property 31 at line 161
160     Property is satisfied.
161     States stored : 3551 states
162     States explored : 0 states
163
164 Verifying property 32 at line 166
165     Property is satisfied.
166     States stored : 3551 states
167     States explored : 0 states
168
169 Verifying property 33 at line 172
170     Property is satisfied.
171     States explored : 0 states
172
173 Verifying property 34 at line 178
174     Property is satisfied.
175     States explored : 0 states
176
177 Verifying property 35 at line 183
178     Property is satisfied.
179     States stored : 3551 states
180     States explored : 8241 states
181
182 Verifying property 36 at line 189
183     Property is satisfied.
184     States explored : 0 states
185
186 Verifying property 37 at line 195
187     Property is satisfied.
188     States explored : 0 states
189
190 Verifying property 38 at line 200
191     Property is satisfied.
192     States stored : 3551 states
193     States explored : 8241 states
194
195 Verifying property 39 at line 206
196     Property is satisfied.
197     States explored : 0 states
198
199 Verifying property 40 at line 212
200     Property is satisfied.
201     States explored : 0 states
202
203 Verifying property 41 at line 217
204     Property is satisfied.
205     States stored : 3551 states
206     States explored : 8241 states
207
208 Verifying property 42 at line 223
209     Property is satisfied.
210     States explored : 0 states
211
212 Verifying property 43 at line 229
213     Property is satisfied.
214     States explored : 0 states
215
216 Verifying property 44 at line 234
217     Property is satisfied.
218     States stored : 3551 states
219     States explored : 8241 states
220
221 Verifying property 45 at line 240
222     Property is satisfied.
223     States explored : 0 states
224
225 Verifying property 46 at line 246
226     Property is satisfied.
227     States explored : 0 states
228
229 Verifying property 47 at line 251
230     Property is satisfied.
231     States stored : 3551 states
```

```
    States explored : 8241 states

Verifying property 48 at line 257
    Property is satisfied.
    States explored : 0 states

Verifying property 49 at line 263
    Property is satisfied.
    States explored : 0 states

Verifying property 50 at line 268
    Property is satisfied.
    States stored : 3551 states
    States explored : 8241 states

Verifying property 51 at line 274
    Property is satisfied.
    States explored : 0 states

Verifying property 52 at line 280
    Property is satisfied.
    States explored : 0 states

Verifying property 53 at line 285
    Property is satisfied.
    States stored : 3551 states
    States explored : 8241 states

Verifying property 54 at line 291
    Property is satisfied.
    States explored : 0 states

Verifying property 55 at line 297
    Property is satisfied.
    States explored : 0 states

Verifying property 56 at line 302
    Property is satisfied.
    States stored : 3551 states
    States explored : 8241 states

Verifying property 57 at line 308
    Property is satisfied.
    States explored : 0 states

Verifying property 58 at line 314
    Property is satisfied.
    States explored : 0 states

Verifying property 59 at line 319
    Property is satisfied.
    States stored : 3551 states
    States explored : 8241 states

Verifying property 60 at line 324
    Property is satisfied.
    States explored : 0 states

Verifying property 61 at line 329
    Property is satisfied.
    States stored : 3551 states
    States explored : 8241 states

Verifying property 62 at line 334
    Property is satisfied.
    States explored : 0 states
```

Referências

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] P. N. Amey and B. J. Dobbing. Static analysis of ravenscar programs. In *IRTAW '03: Proceedings of the 12th international workshop on Real-time Ada*, pages 58–64, New York, NY, USA, 2003. ACM.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[5] 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.

[6] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools, 2004.

[7] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. 200–236. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

[8] Augusto Burgueño, Kim G. Larsen, Luca Aceto, and Luca Aceto. Model checking via reachability testing for timed automata, 1997.

[9] A. Burns. Scheduling hard real-time systems: A review, 1991.

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

[11] Alan Burns. The ravenscar profile. *ACM Ada Letters*, 4:49–52, 1999.

[12] Alan Burns, Brian Dobbing, and George Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of Ada-Europe 98, LNCS*, pages 263–275. Springer-Verlag, 1998.

[13] Bernard Carré and Jonathan 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.

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

[15] Thomas Henzinger Christoph, Christoph M. Kirsch, and Slobodan Matic. Schedule-carrying code. In *In Proc. EMSOFT, LNCS 2855*, pages 241–256. Springer, 2003.

[16] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[17] Arkadeb Ghosal. *A Hierarchical Coordination Language for Reliable Real-Time Tasks*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2008.

[18] 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.

[19] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. Event-driven programming with logical execution times. In *Proc. of HSCC 2004, Lecture Notes in Computer Science*, pages 357–371, 2004.

[20] Arkadeb Ghosal, Tom Henzinger, Daniel Iercan, Christoph Kirsch, and Alberto Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT*, October 2006. http://htl.cs.uni-salzburg.at/.

[21] Joao Gomes, Daniel Martins, Simao Melo de Sousa, and Jorge Sousa Pinto. Lissom, a source level proof carrying code platform. *CoRR*, abs/0803.2317, 2008.

[22] R. "Hamberg and F.W."Vaandrager. "Using Model Checkers in an Introductory Course on Operating Systems". Technical Report "ICIS–R07031", "Radboud University Nijmegen", "December2007".

[23] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Embedded control systems development with giotto. In *OM '01: Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, pages 64–72, New York, NY, USA, 2001. ACM.

[24] 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.

[25] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. Distributed schedule carrying code. Technical Report UCB/CSD-04-1360, EECS Department, University of California, Berkeley, 2004.

[26] Thomas A. Henzinger, Christoph Meyer Kirsch, Rupak Majumdar, and Slobodan Matic. Time-safety checking for embedded programs. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 76–92, London, UK, 2002. Springer-Verlag.

[27] Tom Henzinger, Christoph Kirsch, and Slobodan Matic. Composable code generation for distributed giotto. In *Proceedings of LCTES 2005*, pages 21–30, June 2005.

[28] Tom Henzinger, Christoph Kirsch, Marco Sanvido, and Wolfgang Pree. From control models to real-time code using giotto. *IEEE Control Systems Magazine*, 23(1):50–64, January 2003.

[29] Daniel Iercan. *Contribuitions to the Development of Real-Time Programming Techniques and Technologies*. PhD thesis, EECS Department, University of California, Berkeley, Set 2008.

[30] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[31] Yann-Hang Lee, Gerald Gannod, Karam Ghata, and Eric Wong. A comparative study of formal methods for state based systems. Technical report, Arizona State University and University of Texas at Dallas, 2002.

[32] Shem-Tov Levi and Ashok K. Agrawala. *Real-time system design*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[33] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. *Lecture Notes in Computer Science*, 1384:281–??, 1998.

[34] Kristina Lundqvist and Lars Asplund. A ravenscar-compliant run-time kernel for safety-critical systems*. *Real-Time Syst.*, 24(1):29–54, 2003.

[35] Andreas Naderlinger, Johannes Pletzer, Wolfgang Pree, and Josef Templ. Model-driven development of flexray-based systems with the timing definition language (tdl). In *SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[36] Luís Miguel Pinho. *Sistemas de Tempo Real: Conceitos, Características e Componentes*. ISEP, v0.5 edition, Janeiro 2004.

[37] Rajiv Kumar Poddar and Purandar Bhaduri. Verification of giotto based embedded control systems. *Nordic J. of Computing*, 13(4):266–293, 2006.

[38] 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.

[39] Marco A. A. Sanvido, Arkadeb Ghosal, and Thomas A. Henzinger. xgiotto language report. Technical Report UCB/CSD-03-1261, EECS Department, University of California, Berkeley, Jul 2003.

[40] P. Shin, K.G. ; Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE 82,*, pages 6–24, 1994.

[41] John A. Stankovic, Sang Son, and Jorgen Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32:29–36, 1998.

[42] Professor John Stankovic and A. Stankovic. Real-time computing, 1992.

[43] Josef Templ. Tdl specification and report. Technical report, Department of Computer Science, University of Salzburg, Austria, November 2003.

[44] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.