Verificação de Modelos de Programas HTL

Joel Silva Carvalho e Simão Melo de Sousa

RELiablE And SEcure Computation Group
Departamento de Informática da
Universidade da Beira Interior,
Portugal

Resumo Neste artigo apresenta-se uma *Tool-Chain* para extensão da verificação temporal de programas HTL. Neste processo foi desenvolvida uma ferramenta de tradução automatizada designada por HTL2XTA. Partindo da especificação de um programa HTL, esta ferramenta constrói um modelo Uppaal e um conjunto de propriedades desse modelo. As propriedades inferidas do modelo baseiam-se em propriedades conhecidas do HTL, no entanto é possível acrescentar a essa especificação propriedades que se correlacionem com os requisitos temporais previamente definidos.
This paper introduces a tool-chain that extends the verification of HTL programs. This tool-chain is based on an automated translation tool, called HTL2XTA. This translator builds, from a HTL program, an Uppaal and infers a set of properties that the model (and thus, the source code) should meet. Such properties state the compliance of the model with temporal constraints that can be deduced from HTL source code. In addition to these automatically inferred properties, the Uppaal model can also be completed by any other property that the user may consider.

1 Contexto

Novas exigências surgem com a evolução dos 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[1]. 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.

Este artigo resulta do estudo da fiabilidade de um subconjunto de sistemas computacionais, mais precisamente os Sistemas de Tempo Real Críticos[2][3]. Comparativamente com os sistemas tradicionais, os sistemas de tempo real acrescentam, à questão da fiabilidade, a necessidade intrínseca de garantir que as tarefas que compõem tais sistemas são executadas individualmente num intervalo de tempo bem determinado. Para este tipo de sistemas, não se conseguir

[1] Este trabalho é parcialmente suportado pelo projecto RESCUE (PTDC/EIA/65862/2006) financiado pela FCT (Fundação para a Ciência e a Tecnologia).

finalizar uma tarefa, no tempo que é devido, corresponde sumariamente a uma falha do sistema.

1.1 Uppaal

O Uppaal[4] foi desenvolvido pelas universidades de **Upp**sala e de **Aal**borg, e consiste numa aplicação de modelação (com redes de autómatos temporizados[5]), simulação e verificação (com um subconjunto da lógica TCTL[4]) de sistemas de tempo real. Uma vez que o motor do verificador de modelos[6] é independente da interface gráfica é possível verificar um modelo tendo apenas a especificação textual do mesmo. A especificação do modelo pode ser feita no formato *ta*, *xta* ou *xml*, e a especificação das propriedades no formato *q*. Esta abordagem foi utilizada na *Tool-Chain* para permitir verificação sem necessitar recorrer à interface gráfica do Uppaal.

1.2 HTL

O HTL (Hierarchical Timing Language)[7][8][9] é uma linguagem de coordenação[10] hierárquica para sistemas de tempo real críticos, com tarefas periódicas, que permite verificação da *time-safety* na vertente do escalonamento. As linguagens de coordenação 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. Assim, o princípio de base é que, num sistema crítico que contemple uma camada HTL, esta sirva de especificação do comportamento temporal das funções definidas em C/C++. A descrição temporal é separada da descrição funcional das tarefas que compõem o sistema.

Na base do HTL está uma abstração, que separa a execução física das tarefas da sua execução lógica, designada por LET (Logical Execution Time). Sumariamente o LET considera um intervalo de tempo lógico no qual a tarefa pode ser executado independentemente da forma como o sistema operativo distribui os recursos para essa tarefa. O LET de uma tarefa só é iniciado após a última leitura de variável e é finalizado antes da primeira escrita de variável. Esta abstração tem um papel fundamental no esquema de verificação descrito adiante.

1.3 Motivação

Do estudo feito aos mecanismos de verificação, aplicados nos sistemas de tempo real críticos, constatou-se que existem linguagens derivadas do Giotto[11] capazes de verificar estaticamente o escalonamento de programas. Este tipo de linguagens, apesar de ser académico, aufere um conjunto de propriedades interessantes e distintas das ferramentas mais industrializadas. Essas características são o reaproveitamento eficiente do código, a facilidade teórica de adaptação de um mesmo programa a diversas plataformas, a construção dos programas por hierarquias, a possível utilização de todas as características da linguagem funcional sem qualquer limitação, etc. O nosso interesse recaiu para a mais recente

das linguagens derivadas do Giotto, o HTL que culminou com a publicação de duas teses de doutoramento [7][9] no decorrer de 2008, uma vez que a mesma usufrui das diversas evoluções do Giotto. No entanto, esta linguagem ainda requer algum desenvolvimento para que a sua verificação seja mais completa e considerada suficiente no meio industrial.

Tendo em atenção este aspecto, constatou-se ainda que a verificação temporal do HTL podia ser complementada com verificação de modelos[6]. O tipo de verificação do Uppaal complementa muito bem a análise estática realizada pelo compilador HTL. 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 em simultâneo com a situação B, e sabendo quais as tarefas que implementam essas situações, então o Uppaal pode verificar esta propriedade no modelo do sistema.

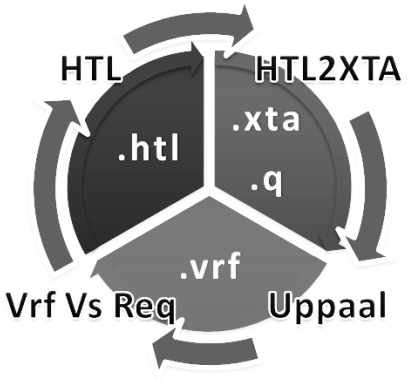
Inspirado em [12] e de forma análoga a [13], mas recorrendo a uma abstração diferente e executando o processo de tradução de forma automatiza, o tradutor que se apresenta no artigo constrói modelos com base em programas HTL e especifica propriedades esperadas sobre os modelos. Após serem verificadas pelo verificador de modelos Uppaal[4], as propriedades permitem estabelecer concordância entre os requisitos temporais e o que foi realmente programado.

2 HTL2XTA Tool-Chain

O tradutor (HTL2XTA) enquadra-se numa *Tool-Chain* (figura 1) delineada com o objectivo de estender a verificação de programas HTL. O HTL2XTA recebe uma especificação HTL (ficheiro .htl) e devolve o respectivo modelo (ficheiro .xta) juntamente com as propriedades automaticamente inferidas (ficheiro .q). É possível, com base nestes ficheiros, utilizar a interface gráfica do Uppaal ou o motor do verificador de modelos (verifyta) para apurar se as propriedades são ou não satisfeitas. De modo a 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.

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 em simultâneo com 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.

Figura 1. HTL2XTA Tool-Chain



2.1 Tradução do Modelo

Sendo o tradutor o meio para atingir o principal objectivo da *Tool-Chain* (a verificação temporal do sistema), decidiu-se evitar a construção de um esquema de tradução que produzisse modelos muito complexos. A razão essencial prende-se com uma limitação clássica da verificação de modelos, nomeadamente o problema da explosão do espaço de estados[6], que não consegue verificar modelos de complexidade alta[1]. Uma vez que o HTL já faz uma análise de escalonamento, a abstração utilizada ignorou por completo a execução física das tarefas. Ao contrário de [13], decidiu-se que a abstração a ter em consideração não devia, nem podia, considerar a execução física das tarefas. Como não interessa verificar o escalonamento das tarefas, o LET do HTL é mais que suficiente para verificar as restantes propriedades temporais.

Fez-se assim 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 abstração adoptada é ignorado por completo o tipo dos comunicadores bem como o driver de inicialização.

Sumariamente todo o esquema de tradução rege-se por esta abstração, i.e. autómatos temporizados que representam o LET das tarefas e autómatos temporizados que representam cada módulo. O tradutor foi testado com diversos

[1] O tamanho do modelo global dum sistema é exponencial comparativamente ao tamanho das suas componentes. A verificação destes modelos obriga a uma exploração, muitas vezes exaustiva, do modelo global

programas HTL e constatou-se que os modelos de programas de complexidade mais elevada não permitem verificação completa (por exemplo a complexidade de um dos modelos não permitiu a verificação da ausência de bloqueio), apesar da simplicidade da abstração. Todos os casos de complexidade intermédia foram alvo de verificação bem sucedida, mas a quantidade de refinamentos aumenta substancialmente a complexidade do modelo. Para aliviar esta situação, o tradutor permite a construção de modelos tendo em conta os níveis de refinamento desejados.

2.2 Inferência de Propriedades

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. 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.

Listing 1.1. Exemplo de propriedades comentadas

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

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. Exemplificando com a listagem 1.1, na linha quatro e seis tem-se que $P = P\_3TS$, $M = IO$, $m = readWrite$ e na linha oito $t = t\_write$.

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.

3 Algoritmo de Tradução do Modelo

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 (Abstract Syntax Tree) do tradutor foi inicialmente pensada para suportar a descrição da totalidade da linguagem HTL, a mesma possui informação que não é analisada ou traduzida pelo algoritmo.

Considera-se a definição da função T, que aceita um programa HTL (de facto a AST do HTL) e que devolve a RAT (Rede de Autómatos Temporizados) 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 auxiliar A, que aceita um programa HTL e que devolve informação necessária para construção da RAT.

3.1 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 abstração 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 $A_{communicator}(n, dt, pd, p) = pd$.

3.2 Transposição do LET

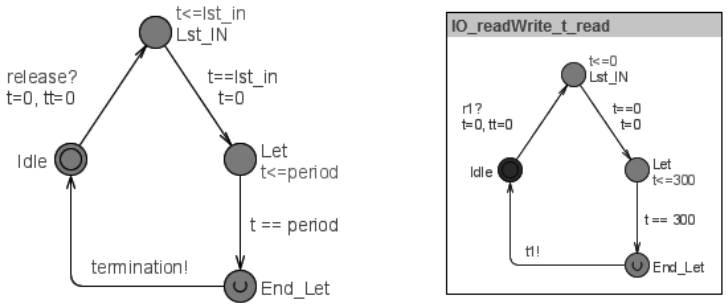
Na base do algoritmo de tradução está uma implementação do LET. Esta implementação é retratada pelos autómatos temporizados *taskTA*, *taskTA_S*, *taskTA_R*, *taskTA_SR*. Existem quatro implementações devido à utilização de portos concretos nas invocações das tarefas. As instanciações do autómato temporizado, $taskTA$ 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.

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.

taskTA Seja $Port$ o conjunto de todos os portos concretos, cp um porto concreto 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 2. Autómato $taskTA$ à esquerda e instanciação à direita



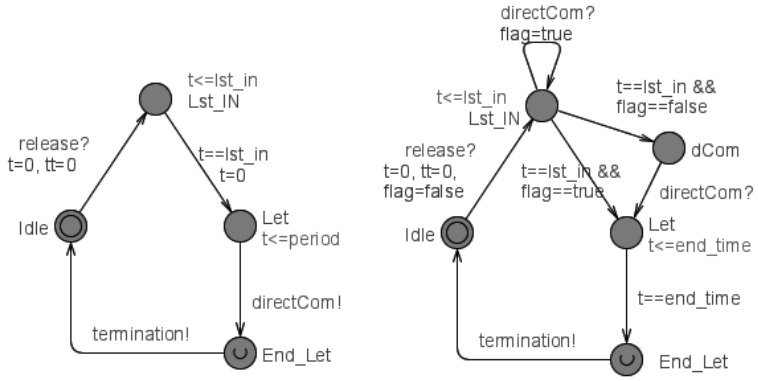
A cada invocação de tarefa, onde não exista nenhum porto concreto, quer nas variáveis de entrada como de saída, corresponde uma instanciação de um autómato $taskTA$ (figura 3.2). Os canais de sincronização urgentes r e t são determinados na declaração do sistema. O nome do canal r de cada instanciação 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 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.

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

Figura 3. Autómato $taskTA\_S$ à esquerda e $taskTA\_R$ à direita



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.

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.

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

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 $\forall mode \in module, \exists subModule \in modulteTA, T_{mode}(n, p, refP, bmo, pos) = subModule(e, t)$.

4 Algoritmo de Tradução das Propriedades

Considera-se 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.

4.1 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*).

4.2 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.

4.3 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.

4.4 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 Validação Experimental

Utilizando a versão actual do tradutor (v0.4 de 24/04/2009) conseguiu-se gerar, com sucesso, modelos e propriedades para diversos programas HTL apresentados em [7][9]. Na tabela 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.

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 abstração do programa principal. Exceptuando no programa do *Steer By Wire* onde se verificou explosão de estados, foram verificadas todas as propriedades automaticamente produzidas.

Tabela 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

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 do 3TS[7][9] 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.

O programa da planta real $3TS - FE2$, por ter mais dois refinamentos do que a versão da planta simulada $3TS - simulink$, 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.

6 Conclusão e Trabalho Futuro

O tradutor HTL2XTA apresentado foi desenvolvido em ocaml, no âmbito de uma dissertação de mestrado, seguindo o modelo de construção de um tradicional compilador, deixando a verificação de tipos ao compilador clássico do HTL. A totalidade da *Tool-Chain* (compilador HTL, tradutor HTL2XTA, Uppaal) funcionam em linux. A linguagem HTL surge num contexto académico e ainda lhe resta completar o desafio da sua transferência de tecnologia para um contexto industrial. Entendemos este nosso trabalho como uma contribuição à resolução deste desafio.

Relativamente às vantagens da verificação proporcionada pela *Tool-Chain HTL2XTA* pode-se dar um dos exemplos testados no caso do 3TS[7][9]. Adulterando a descrição HTL para que o modo $readWrite$ tenha um período de 5000 em vez de 500 isto tem por efeito que o novo executável faça dez vezes mais cálculos com base nas mesmas leituras dos sensores. Em termos de escalonamento a consequência desta alteração é nula na medida que não inviabiliza o escalonamento do programa. Todavia isto infringe os requisitos temporais que exigem que por cada leitura do mesmo sensor seja feita uma actualização do valor do actuador da respectiva bomba de água.

A *Tool-Chain HTL2XTA* não identifica automaticamente esta situação, através da falha da verificação de uma das propriedades, no entanto após comparação dos requisitos temporais com a especificação de cada propriedade gerada é possível constatar a discrepância existente entre os LET's das tarefas e os períodos dos diversos modos. Outra forma de identificar o erro directamente consiste na especificação manual da propriedade apresentada na listagem 1.2. Esta propriedade exige que no instante imediatamente antes da execução do LET da tarefa t_T1 o Let da tarefa t_read esteja em execução. No caso do programa HTL com o período do modo $readWrite$ certo esta propriedade é satisfeita, no caso do período ser maior (independentemente do valor) esta propriedade não vai ser satisfeita.

Listing 1.2. Exemplo de especificação de uma propriedade suplementar

```
1  A[] (T1_m_T1_t_T1.Lst_IN && (T1_m_T1_t_T1.tt>T1_m_T1_t_T1.lst_in-1) &&
       T1_m_T1_t_T1.tt<T1_m_T1_t_T1.lst_in) imply IO_readWrite_t_read.Let
```

Apesar de o tradutor estar numa versão estável e do mesmo conseguir traduzir efectivamente os diversos programas referidos na tabela 1, este não foi alvo de verificação formal. Um dos trabalhos futuros passará, de alguma forma, pela verificação formal da correção do mesmo. No entanto, tendo em consideração que se constatou que esta versão não é capaz de lidar com programas HTL de maior dimensão ainda é cedo para se ter esse esforço.

Outro trabalho futuro contemplará a extensão do próprio HTL com anotações que permitam introduzir regras comportamentais suplementares, como por exemplo o caso dos *switch*. Mas antes disso, deve ser estudada a forma como lidar com essas anotações e o impacto das mesmas no modelo. Por fim pretende-se transpor este modelo, ou um modelo muito similar, para o Ada/SPARK. Isto é, quer-se 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.

Referências

1. J. Rushby, "Formal methods and their role in the certification of critical systems," tech. rep., Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop, 1995.
2. S.-T. Levi and A. K. Agrawala, *Real-time system design*. New York, NY, USA: McGraw-Hill, Inc., 1990.
3. P. J. Stankovic and A. Stankovic, "Real-time computing," 1992.
4. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *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* (M. Bernardo and F. Corradini, eds.), no. 3185 in LNCS, pp. 200–236, Springer–Verlag, September 2004.
5. J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," 2004.
6. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.
7. A. Ghosal, T. A. Henzinger, D. Iercan, C. Kirsch, and A. L. Sangiovanni-Vincentelli, "Hierarchical timing language," Tech. Rep. UCB/EECS-2006-79, EECS Department, University of California, Berkeley, May 2006.
8. A. Ghosal, *A Hierarchical Coordination Language for Reliable Real-Time Tasks*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2008.
9. D. Iercan, *Contribuitions to the Development of Real-Time Programming Techniques and Technologies*. PhD thesis, EECS Department, University of California, Berkeley, Set 2008.
10. D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, 1992.
11. T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, (London, UK), pp. 166–184, Springer-Verlag, 2001.
12. K. Lundqvist and L. Asplund, "A ravenscar-compliant run-time kernel for safety-critical systems*," *Real-Time Syst.*, vol. 24, no. 1, pp. 29–54, 2003.
13. R. K. Poddar and P. Bhaduri, "Verification of giotto based embedded control systems," *Nordic J. of Computing*, vol. 13, no. 4, pp. 266–293, 2006.