Este documento apresenta uma introdução ao uso da ferramenta de Model Checking UPPAAL e também alguns conceitos relacionados. Para tal são apresentados alguns exemplos de utilização.

Sumário

1 Introdução

O Model Checker UPPAAL constitui um ambiente integrado onde é possível criar modelos (*autômatos temporais*), os quais podem ser simulados e analisados por meio de especificação de propriedades. Por sua vez, essas propriedades são expressadas em uma lógica similar a TCTL.

A ferramenta UPPAAL possui uma interface gráfica implemantada em Java, onde podem ser construídos modelos (*autômatos temporais*), tais modelos podem ser simulados e ter propriedades verificadas. O modelo construído na ferramenta é de fato constituído por uma rede de autômatos temporais.

Além disso, o verificador do UPPAAL pode ser usado por uma interface de linha de comando, `verifyta`. Essa estratégia pode ser útil quando se utiliza a ferramenta remotamente por meio de um servidor alto desempenho computacional, por exemplo.

2 Conceitos Básicos

Antes de apresentar a utilização da ferramenta faz-se necessário introduzir alguns conceitos básicos como Lógica e Autômato Temporal.

2.1 Lógica Temporal

As seções que seguem apresentam as lógicas temporais utilizadas no processo de especificação de propriedades. A questão *tempo* (ou *temporal*) não está necessariamente relacionada com a capacidade de adicionar a medição de tempo, mas sim com a possibilidade de estabelecer sucessão temporal na ordenação dos eventos.

A Subseção 2.1.1 apresenta a lógica temporal linear (*Linear Temporal Logic*, LTL), que é utilizada para a especificação de propriedades temporais. Após a Subseção 2.1.2 apresenta lógica de árvore de computação (*Computation Tree Logic*, CTL), a qual especifica propriedades através de caminhos e estados. Por fim, tem-se a Subseção 2.1.3 que descreve uma variação da CTL com suporte a fórmulas temporais, a lógica de árvore de computação temporal (*Timed Computation Tree Logic*, TCTL).

Observação: na TCTL tem-se realmente o sentido de progressão temporal, e não somente na ordenação (temporal) dos eventos.

2.1.1 LTL - Lógica Temporal Linear

A lógica temporal linear (*Linear Temporal Logic*, LTL) possui os operadores básicos da lógica clássica proposicional ($\neg, \wedge$) e adiciona dois operadores temporais: o operador *próximo* (*next*, $\bigcirc$) e o *até* (*until*, $\cup$).

A gramática da LTL é definida em (1):

$$\varphi ::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2 \tag{1}$$

- O operador *next* $\bigcirc$ é unário e indica que no próximo momento a fórmula φ será verdadeira.
- O operador *until* $\cup$ é um operador binário e especifica que, em todos os momentos antes que φ_2 torne-se verdadeiro, φ_1 é sempre verdadeiro.

- a é um elemento do conjunto de proposições atômicas AP (*Atomic Propositions*).

Ainda é possível derivar dois novos operadores unários a partir do operador $\cup$:

- o operador *agora ou em algum momento* no futuro ($\Diamond$)
- e o *agora e sempre* no futuro ($\Box$).

O primeiro operador é derivado da seguinte forma: $\Diamond\varphi \equiv true \cup \varphi$.
O segundo operador é dado por $\Box\varphi \equiv \neg\Diamond\neg\varphi$.
Conforme visto anteriormente, os operadores $\Box$ e $\Diamond$ são oriundos da **lógica modal** e são conhecidos também como operadores: *necessariamente* e *possivelmente*.
A Figura 1 ilustra uma representação da semântica dos operadores da LTL e abaixo estão alguns exemplos de fórmulas LTL.

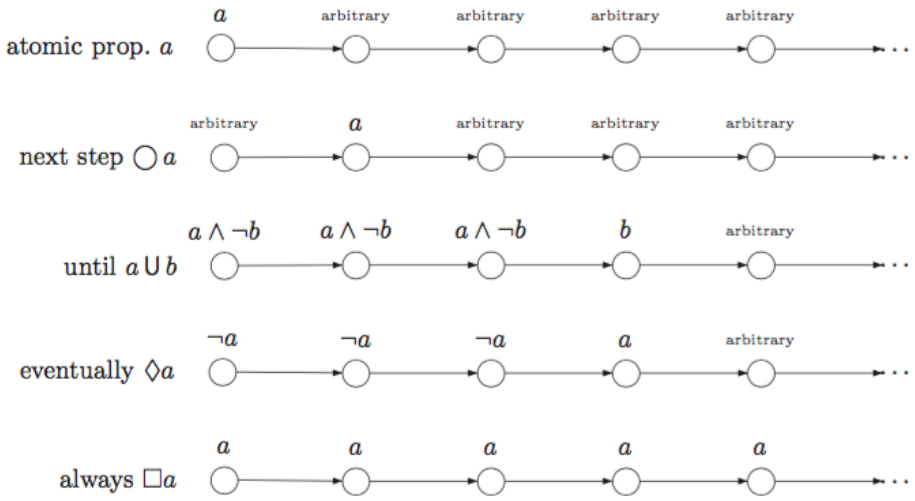

Figura 1: Representação do significado dos operadores da LTL

Operadores primitivos × Operadores derivados

Note que nos exemplos que seguem o operador ($\rightarrow$) é utilizado nas fórmulas LTL. Porém, o operador $\rightarrow$ não está definido na gramática da LTL, como um operador primitivo. Isso acontece por que operadores, como $\rightarrow$, $\vee$ podem ser obtidos a partir dos operadores $\neg$, $\wedge$. Assim, a gramática é definida com o fragmento mínimo de operadores e os demais operadores podem ser derivados do fragmento mínimo. Seguem as definições de equivalência entre alguns operadores lógicos:

$$\varphi_1 \vee \varphi_2 =^{def} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$$
$$\varphi_1 \rightarrow \varphi_2 =^{def} \neg\varphi_1 \vee \varphi_2$$
$$\varphi_1 \leftrightarrow \varphi_2 =^{def} (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$$

Exemplos - fórmulas LTL

- fórmula LTL: $\Box(\neg crit_1 \vee \neg crit_2)$
  - Essa fórmula especifica uma propriedade de *segurança*, onde dois processos disputando acesso às suas regiões críticas **mutuamente exclusivas** jamais podem ocupá-las simultaneamente. Ou seja, sempre deve ser garantido que somente um processo estará em sua região crítica.
- fórmula LTL: $(\Box\Diamond crit_1) \wedge (\Box\Diamond crit_2)$
  - Essa fórmula expressa uma propriedade de *vivacidade*. Cada processo P_i é **infinitamente frequente** em sua seção crítica.
  - Ainda nessa fórmula é possível identificar o uso da combinação de operadores modais: $\Box\Diamond a$. Essa fórmula descreve a propriedade que em qualquer momento j, existe um momento $i \geq j$ onde um estado a é visitado.
- fórmula LTL: $\Box(tremPerto \rightarrow abrirCancela)$
  - a qual especifica um cruzamento de uma linha férrea.
- fórmula LTL: $\Box(requisicao \rightarrow \Diamond resposta)$
  - a qual especifica uma propriedade de progresso.

2.1.2 CTL - Lógica de Árvore de Computação

A LTL tem uma certa limitação devido ao fato de que sempre há somente um sucessor temporal. A lógica de árvore de computação (*Computation Tree Logic*, CTL) é uma lógica temporal com *ramificações* que permite especificar propriedades relacionadas a vários caminhos possíveis a partir de um estado s.

A sintaxe da CTL para a especificação de propriedades é dividida em duas partes: uma para estados e outra para caminhos.

A gramática de **estados** é definida em (2):

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi \tag{2}$$

onde:

- $\exists$ e $\forall$ são quantificadores de caminhos, existencial e universal, respectivamente;

- φ é uma fórmula de caminho, a qual é definida em 3, na gramática de **caminhos**;
- $a \in AP$ (*conjunto de proposições atômicas*).

Adicionalmente, a CTL possui o operador *next* $\bigcirc$. Desse modo é possível derivar os operadores temporais presentes na LTL ($\Box$, $\Diamond$, $\cup$), com exatamente os mesmos significados, *sempre*, *em algum momento*, *until*.

Para a especificação de fórmulas de **caminhos** a gramática é a seguinte (3):

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \cup \Phi_2 \tag{3}$$

onde Φ, Φ_1, e Φ_2 são fórmulas de **estados** (definidas por meio da primeira gramática – 2).

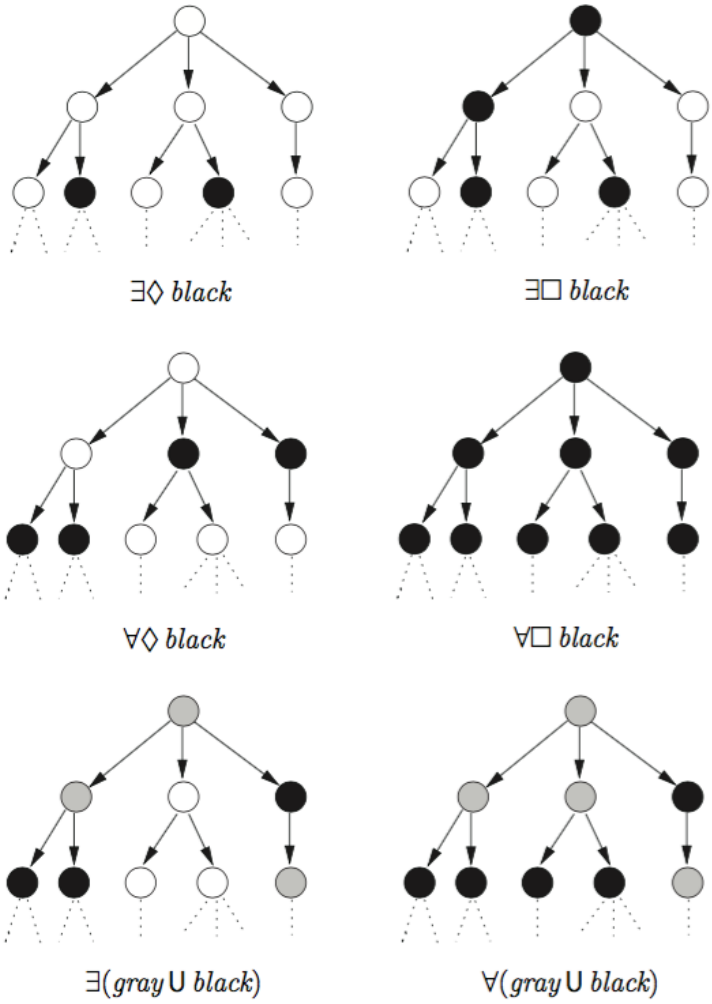

Figura 2: Representação do significado dos operadores da CTL

A Figura 2 ilustra o significado das fórmulas em CTL.

- A árvore superior à esquerda indica que existe ($\exists$) uma ramificação onde *black* é em algum momento ($\Diamond$) verdadeira.
- A árvore superior à direita mostra o caso onde há ($\exists$) uma ramificação na qual a proposição *black* é sempre ($\Box$) verdadeira.
- A árvore intermediária à esquerda representa que para todos os caminhos ($\forall$) *black* é em algum momento verdadeira ($\Diamond black$).

- A árvore intermediária à direita define que para todas as ramificações possíveis ($\forall$) *black* é sempre ($\Box$) satisfeita.
- A árvore inferior à esquerda mostra que há caminhos ($\exists$) nos quais os estados são *gray* até ($\cup$) *black*.
- A árvore inferior à direita define que para todos os caminhos possíveis ($\forall$) os estados são *gray* até que ($\cup$) a propriedade *black* seja satisfeita.

Exemplos - fórmulas CTL

- A fórmula CTL: $\forall\Box(\neg crit_1 \vee \neg crit_2)$

  Representa: para todas as computações possíveis, somente um estado está na sua região crítica.

- A fórmula CTL: $\forall\Box(requisicao \rightarrow \forall\Diamond resposta)$

  Representa: toda requisição será em algum momento (*agora ou num futuro*) respondida.

- A fórmula CTL: $\forall\Box(yellow \rightarrow \forall \bigcirc red)$

  Representa: para todos os casos sempre o sinal amarelo do semáforo irá anteceder um sinal vermelho do semáforo.

2.1.3 TCTL - Lógica de Árvore de Computação Temporal

Para a especificação de propriedades que necessitam da especificação de limites temporais a lógica de árvore de computação temporal (TCTL, aqui no sentido também de medir o tempo) pode ser usada. A TCTL estende a CTL com a inclusão da possibilidade de uma propriedade ser satisfeita dentro de um intervalo de tempo.

A sintaxe da TCTL para a especificação de propriedades também é dividida em duas partes: uma para estados e outra para caminhos.

A gramática para **estados** é definida em (4):

$$\Phi ::= true \mid a \mid g \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi \tag{4}$$

- onde $a \in AP$ (conjunto de proposições atômicas),
- $g \in ACC(C)$ (conjunto de restrições sobre o conjunto de relógios C),
- e φ é uma fórmula de caminho.

Fórmulas de **caminhos** são especificadas pela seguinte gramática (5):

$$\varphi ::= \Phi_1 \cup^J \Phi_2 \tag{5}$$

- onde $J \subseteq \mathbb{R}_{\geq 0}$.

Como o operador *until* ($\cup$) agora é temporal no sentido de medição de tempo, os operadores dele derivados ($\Diamond$ e $\Box$) também passam a ter esse aspecto temporal.

Exemplo - fórmula TCTL

Um exemplo de fórmula TCTL é

$$\forall\Box((on \wedge x = 0) \rightarrow (\forall\Box^{\leq 1} on \wedge \forall\Diamond^{>1} off))$$

Esta fórmula especifica que:

- para todos os caminhos possíveis e sempre nestes ($\forall\Box$),
- um interruptor, uma vez ligado ($on \wedge x = 0$),
- irá permanecer ligado por, pelo menos uma unidade de tempo ($\forall\Box^{\leq 1} on$),
- antes de desligar ($\forall\Diamond^{>1} off$).
- Note ainda que:
  - *on*, *off* são proposições atômicas.
  - $x = 0$ é restrição em relação ao relógio x.

2.2 Autômato Temporal

Nesta Seção será visto o conceito de Autômato Temporal seguido por exemplos.

Embora os Sistemas de transição descritos até o momento caracterizem um formalismo expressivo e representativo, eles não têm a capacidade de medir a progressão do tempo nos sistemas modelados.

Assim, acabam não sendo adequados para a modelagem de sistemas onde o tempo é crítico, por exemplo, máquinas de café, protocolos de comunicação, máquinas bancárias, sistemas de controle de tráfego, entre outros. Sem a característica temporal não é possível especificar por quantas unidades de tempo uma dada cancela deve permanecer aberta, por exemplo.

Quando é possível adicionar e expressar as questões temporais é possível escrever proposições, como:

O semáforo passará para o sinal verde dentro dos próximos 30 segundos.

Para lidar com sistemas onde seja possível admitir unidades de tempo foi criado o conceito de **autômatos** estendido por meio de **relógios**.

Para contextualizar a questão envolvendo autômatos temporais o exemplo do Trem, Controlador e Cancela será revisto.

A Fig. 3 ilustra os sistemas de transição para cada componente: Trem, Controlador e Cancela.

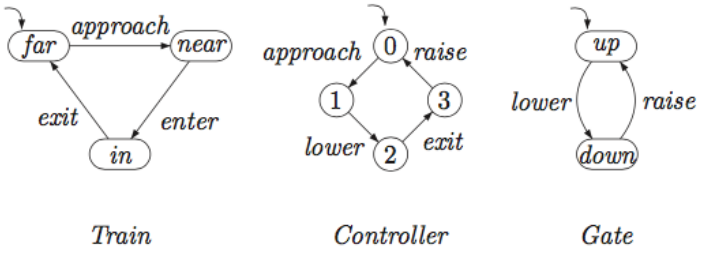

Figura 3: Sistemas de Transição para Trem, Controlador e Cancela

Para o correto funcionamento do sistema é esperado que a seguinte **propriedade de segurança** seja satisfeita:

Porém, a composição dos sistemas de transição não garante tal propriedade de segurança. Isto pode ser visto por uma análise do fragmento inicial (ver Figura 4), onde não é possível deduzir o que o sistema de transição irá executar após o envio do sinal "approach", ou a cancela será fechada ou o trem irá ingressar no cruzamento (sem garantias do fechamento da cancela).

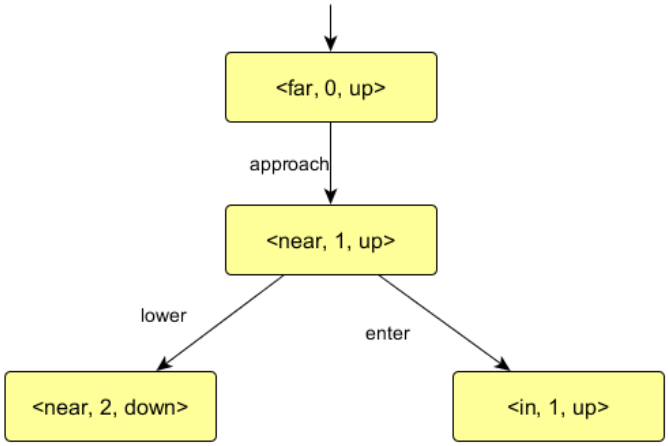

Figura 4: Fragmento inicial do sistemas de Transição $Trem \mid\mid Controlador \mid\mid Cancela$

A solução neste caso é adicionar aspectos temporais para controlar o sistema.

- Primeiro, o trem necessita mais de **dois** minutos para alcançar o cruzamento após a emissão do sinal de aproximação (*approach*).
- Quando recebe o sinal de aproximação, após exatamente **um** minuto o Controlador irá sinalizar a Cancela para ser fechada.
- O processo de fechamento da cancela é ≤ 1.

Com isso, o trem irá **entrar** no cruzamento somente após mais que **dois** minutos. Por outro lado, o fechamento da cancela leva no máximo **dois** minutos (após a sinalização de aproximação).

Consequentemente, sempre que o trem atingir o cruzamento com a rodovia, a cancela já estará fechada. Portanto, a propriedade de **segurança** agora é assegurada.

2.2.1 Definições

Aqui são descritas as definições relacionadas com Autômato Temporal. Autômatos temporais modelam o comportamento de sistemas de tempo crítico. De fato, um autômato temporal é um *grafo de programa* "equipado" com um conjunto finito de relógios, denotados: x, y e z. Esses relógios diferem de variáveis comuns, já que o acesso é limitado, relógios podem apenas ser inspecionados (testados) e "resetados" a zero. Todos relógios evoluem sob uma taxa, ou seja após passar d **unidades de tempo**, todos relógios avançam conforme o valor d. Os relógios podem ser verificados de forma independente.

Condições nos valores dos relógios são usadas como **condições** ("guards") nas ações: somente se a condição é satisfeita é que a respectiva ação é habilitada e pode ser escolhida; caso contrário a ação é desabilitada. Aquelas condições que dependem dos valores dos relógios são

chamadas **restrições de relógios** (ou *clock constraints*). Por questões de simplicidade, tem-se que para habilitar condições depende-se apenas dos relógios e não de outros valores de variáveis. As restrições de relógios também são usadas para limitar a quantidade de tempo que é possível "passar" em uma dada *location*.

Na definição seguinte tem-se como as restrições sobre relógios são escritas:

Definition 1 (Restrição de Relógio). *Uma restrição de relógio sobre o conjunto C de relógios é formada conforme a seguinte gramática:*

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g$$

onde $c \in \mathcal{N}$ e $x \in C$. Tendo que $CC(C)$ denota o conjunto de restrições de relógios sobre C.

As restrições de relógio que não possuem qualquer conjunção são ditas atômicas. E, portanto $ACC(C)$ denota o conjunto de todas restrições de relógio atômicas sobre C.

■

Intuitivamente, um **autômato temporal** é um *grafo de programa* (levemente modificado), cujas variáveis são relógios. Os relógios são usados para formular suposições de tempo real no comportamente do sistema. Uma aresta em um autômato temporal é rotulada por meio de três componentes: (**i.**) um *guard* (condição, quando é permitido que uma dada aresta seja escolhida?); (**ii.**) uma **ação** (o que é realizado quando uma dada aresta é selecionada?); e (**iii.**) um conjunto de relógios (quais relógios devem ser "zerados"?). Uma *location* possui uma invariante que restringe a quantidade de tempo que é possível permanecer em uma dada *location*.

A definição formal de um **Autômato Temporal** segue abaixo:

Definition 2 (Autômato Temporal). *Um autômato temporal é uma tupla $AT = (Loc, Act, C, \longrightarrow, Loc_0, Inv, AP, L)$, onde:*

- *Loc é o conjunto finito de* ***locations****.*
- *$Loc_0 \subseteq Loc$ é o conjunto de* ***locations iniciais****.*
- *Act é o conjunto finito de* **ações**.
- *C é conjunto finito de* **relógios**.
- *$\longrightarrow \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ é uma relação de* **transição**.
- *$Inv : Loc \to CC(C)$ é uma função de atribuição de* **invariante** *(a uma dada location).*
- *AP é um conjunto finito de* **proposições atômicas**.
- *$L : Loc \to 2^{AP}$ é uma função de aplicação de* **rótulos** *para locations.*

*$ACC(AT)$ denota o conjunto de restrições nos relógios atômicos que ocorrem num guard ou em uma invariante de location em um Autômato Temporal (**AT**).*

■

10 Como dito anteriormente, um autômato temporal é um grafo de programa com um conjunto finito C de relógios. Arestas são rotuladas com tuplas (g, α, D), onde g é uma restrição de relógio, α é uma ação e $D \subseteq C$ é um conjunto de relógios.

A interpretação intuitiva de $loc \longrightarrow^{g:\alpha,D} loc'$ é que o autômato temporal pode mover-se de uma *location* loc para outra loc', quando a restrição de relógio g é *satisfeita*. Além disso, quando for realizada essa transição, qualquer relógio D será "zerado" e uma dada ação α será realizada.

A função Inv atribui a cada *location* uma invariante que especifica por quanto (tempo) é possível permanecer na respectiva *location*. Ou seja, a *location* loc deve ser "deixada" antes da invariante $Inv(loc)$ tornar-se **inválida**. Caso isso não seja possível, não é possível progredir no autômato temporal. Esta situação é conhecida como **timelock**.

Observação: como esperado o aspecto *temporal* tem papel fundamental neste tipo de autômato. Note que tal aspecto pode aparecer em componentes distintos do autômato temporal: nas *locations* e/ou nas *arestas*.

2.2.2 Exemplos

A Figura 5 ilustra um autômato temporal para a *Cancela*, considerando o sistema de controle de um trem, visto anteriormente.

Observe que para fechar a cancela leva no máximo **uma** unidade de tempo. Ao passo que para abrir a cancela leva no mínimo **uma** e no máximo **duas** unidades de tempo.

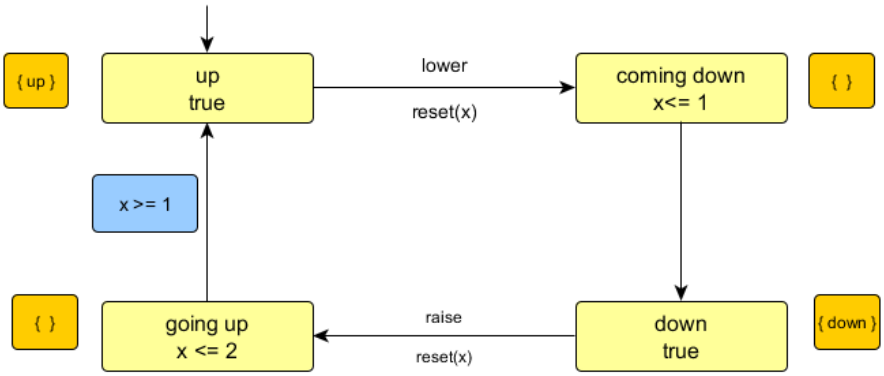

Figura 5: Autômato Temporal para *Cancela*

3 Exemplo: Alocação de Recursos em uma Linha de produção

Nesta seção é visto um exemplo de uma simples **linha de produção**, onde serão apresentadas algumas características do UPPAAL. Esse exemplo é baseado em Milner (1989).

3.1 Descrição do problema

Supondo que existem dois trabalhadores (ver ilustração na Fig. 6[1]) compartilhando o uso de duas ferramentas: um **martelo** e uma **marreta**. Essas duas ferramentas são usadas para fabricar objetos a partir de componentes simples.

Cada **objeto** é feito pregando um prego em um bloco. Um par consistindo em um *prego* e um *bloco* é chamada um **trabalho**. Os trabalhos chegam de forma sequencial em uma correia transportadora.

A linha de produção pode envolver qualquer número de pessoas, chamadas *trabalhadores*, os quais compartilham mais ou menos ferramentas. Porém, aqui é considerado um sistema com apenas **dois trabalhadores** e duas ferramentas: **martelo** e **marreta**.

Ainda existe uma especificação quanto a natureza do serviço, a qual influencia a escolha das ferramentas. Existem dois predicados em relação aos trabalhos: **fácil** e **difícil**. Um trabalhador realizará um trabalho **fácil** com suas mãos, um trabalho **difícil** com o **martelo** e outros trabalhos com o **martelo**, ou com a **marreta**.

Figura 6: Ilustração de uma linha de produção

3.2 Editor

Na Fig. 7 tem-se uma visão geral da ferramenta. Especificiamente da aba "Editor", onde é possível utilizar ferramentas básicas para criar o modelo de um autômato temporal no UPPAAL.

Quando a ferramenta é iniciada tem-se a princípio um estado (com duplo círculo) que é uma estado inicial do sistema. A partir desse estado é possível construir os demais estados e transições do sistema por meio de ícones específicos no menu superior. Existem ícones para criação das *locations* e arestas. Depois de adicionar esses componentes é possível editar as informações, como: nome, *location* inicial, entre outros.

Após, a Fig. 8 ilustra um simples exemplo de uma linha de produção, onde são produzidos serviços de três níveis: *easy*, *average* e *hard*. Especificiamente, um serviço dito fácil (*easy*)

[1]Referência para imagem: `https://www.wannapik.com/vectors/77380`.

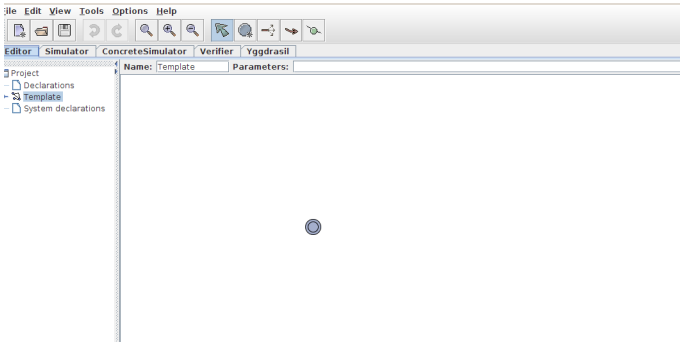

Figura 7: Visão geral do Editor

pode ser feito com as mãos. Ao passo que um serviço médio (*average*) pode ser feito com uma *marreta* ou então com *martelo*. E ainda um serviço dito difícil (*hard*) somente será feito com uso do *martelo*.

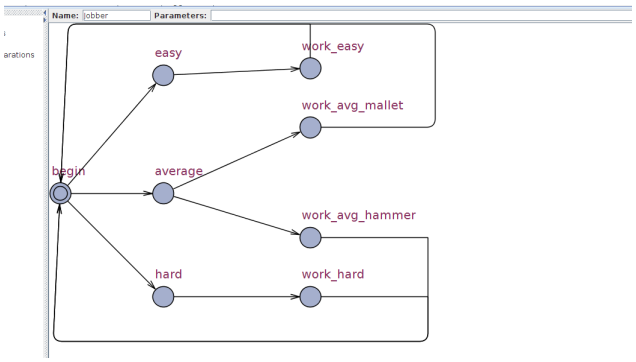

Figura 8: Exemplo do Jobber

3.3 Simulador

Tendo criado o modelo por meio do Editor do UPPAAL, agora é possível definir a simulação do modelo. Para tal, inicialmente criamos as declarações do sistema (ver Fig. 9). Nesta etapa são criadas as declarações referentes aos modelos que constituem todo o sistema. Por enquando, são duas cópias do modelo `Jobber`: `Jobber1` e `Jobber2`. **Observação**: é possível usar a tecla de atalho **F7** (ou seguir o caminho: *Tools* > *Check Syntax*) para fazer a verificação de sintaxe do modelo e suas respectivas declarações.

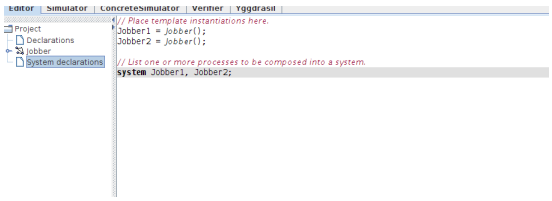

Figura 9: Declarações do Sistema

Assim, na Fig. 10 aparecem as duas respectivas cópias do modelo, as quais serão utilizadas na simulação do sistema. Nessa figura é possível perceber que as *locations* marcadas em *vermelho* são as *locations* que identificam o ponto da simulação corrente. Nas opções à direita é possível verificar quais são as transições possíveis a cada instante. Para a transição selecionada, a aresta respectiva fica em *vermelho* no modelo.

Adicionalmente são disponíveis algumas opções de simulação:

- `Next`: próximo passo na simulação do sistema.
- `Prev`: passo anterior na simulação do sistema.
- `Replay`: repetir algum passo na simulação do sistema.
- `Random`: seguir uma estratégia aleatória para simulação do sistema.
- `Slow -- Fast`: ajuste fino na velocidade de simulação.

Ainda será visto que é possível salvar um traço de simulação.

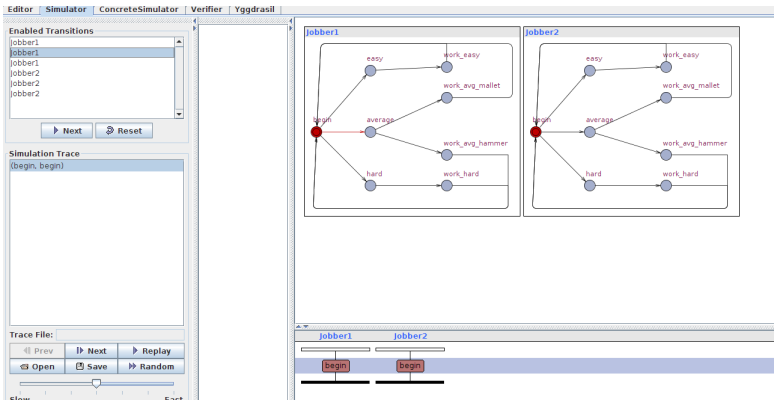

Figura 10: Simulação do Modelo

3.4 Canais

Um recurso bastante usado no UPPAAL são os canais de sincronização, eles auxiliam na comunicação entre sistemas. No caso, do modelo da linha de produção existe apenas um recurso (martelo) para ser usado por dois trabalhadores. Neste caso, faz-se necessário refinar o modelo de forma que sejam usados os canais de sincronização conforme apresenta a Fig. 11.

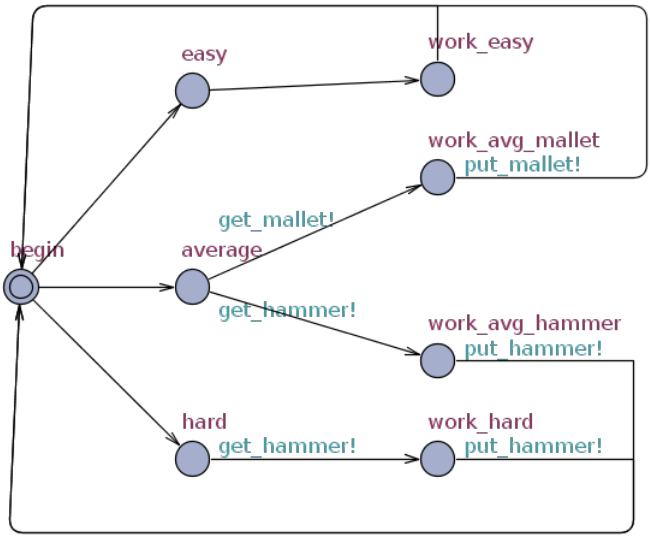

Figura 11: Exemplo do Jobber com Canais de Sincronização

Para tal são criados os canais: `get_mallet!`, `get_hammer!`, `put_mallet!`, `put_hammer!`. Na Fig. 11 estão as transições ativas de sincronização, ao passo que nas Figuras 12 e 13 estão os complementos dessas quatro ações de sincronização.

[14] Perceba ainda que os modelos das Figuras 12 e 13 são bem simples, ambos possuem apenas duas *locations*: `free` e `taken`, as quais respectivamente indica que o recurso (marreta ou martelo) está livre ou sendo usado.

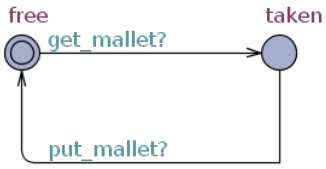

Figura 12: Modelo para o *Mallet* - marreta

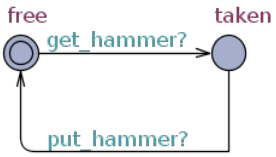

Figura 13: Modelo para o *Hammer* - martelo

3.5 Verificador

Tendo criado o modelo (com o Editor), definido a Simulação do sistema, ainda é possível especificar propriedades que possam ser verificadas pela ferramenta. As propriedades escolhidas servem para assegurar o correto funcionamento do sistema. O UPPAAL retorna se a propriedade especificada é **satisfeita** ou não no sistema em questão, caso a propriedade não seja satisfeita a ferramenta pode retornar um traço de simulação que indique o caminho que mostrar por que a propriedade em questão não é satisfeita.

Importante! Para especificar as propriedades, o UPPAAL utiliza a lógica TCTL. Portanto, faz uso dos seguintes operadores:

- **Operadores básicos da lógica proposicional**:

  && operador de conjunção.

  || operador de disjunção.

  == operador de equivalência.

  `imply` operador de implicação.

  `not` operador de negação.

- **Operadores da lógica temporal (de caminho e estado)**:

  `A` quantificador de caminho (**universal**), verificar todos caminhos possíveis.

  `E` quantificador de caminho (**existencial**), verificar pelo menos um caminho possível.

  `[]` operador de estado **sempre** em qualquer possível estado do sistema.

  `<>` operador de estado **agora ou em algum momento no futuro**.

A Fig. 14 ilustra a verificação da propriedade de ausência de *deadlock*.

```
A[] not deadlock
```

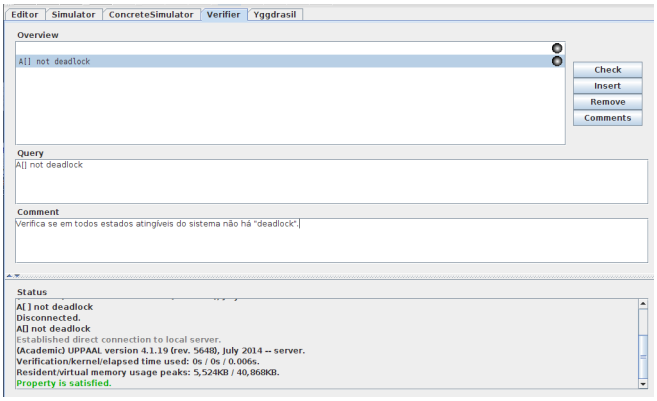

Figura 14: Especificação de propriedades - *Deadlock*

A Fig. 15 ilustra a verificação da seguinte propriedade:

```
E<> (Jobber1.work_hard && Jobber2.work_hard)
```

Essa propriedade verifica se existe algum caminho possível onde tanto `Jobber1`, como `Jobber2` estejam no estado `work_hard`. Observe que realmente é desejado que essa propriedade **não** seja **satisfeita**. Pois assim é assegurado os dois trabalhadores não fazem uso do recurso *martelo* ao mesmo tempo.

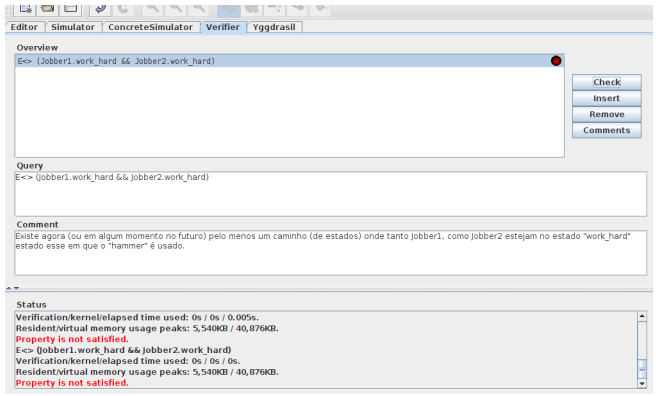

Figura 15: Especificação de propriedades - trabalho *difícil*

A Fig. 16 ilustra a verificação da seguinte propriedade:

```
E<> (Jobber1.work_avg_mallet && Jobber2.work_avg_hammer)
```

Essa propriedade verifica se há algum estado no sistema em que `Jobber1` esteja trabalhando com a *marreta* e `Jobber2` esteja trabalhando com a *martelo*. Tal propriedade é **satisfeita**.

Ainda é possível constatar na Fig. 17 a exibição de um traço de diagnóstico da propriedade anterior na ferramenta de simulação do UPPAAL.

Atenção! Para utilizar a opção de traço de diagnóstico é necessário seguir o caminho no menu: `Options` > `Diagnostic Trace` > `Shortest` (ou outra opção ao invés de *Shortest*). Selecionando essa opção a ferramenta de simulação exibe o traço de diagnóstico, o qual pode ser salvo em um *trace file* com a extensão XTR. Essa opção é útil para uma análise detalhada do comportamento do sistema.

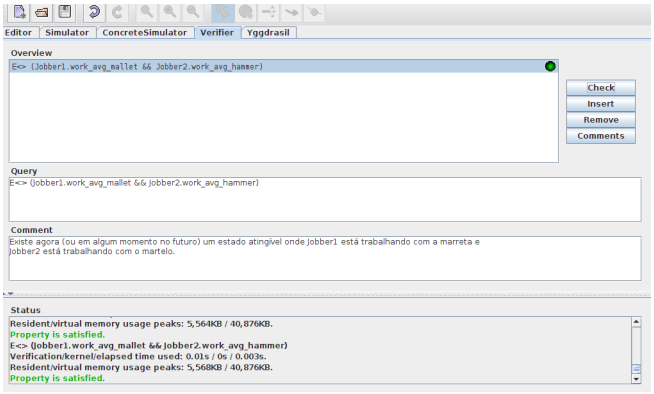

Figura 16: Especificação de propriedades - trabalho *médio*

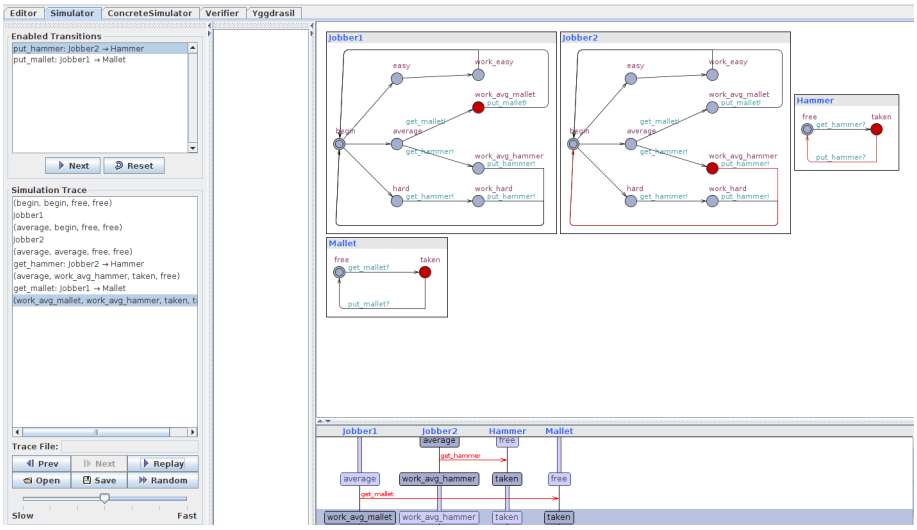

Figura 17: Especificação de propriedades - traço de verificação

Salvando consultas/propriedades

É possível salvar as consultas ou propriedades escritas na ferramenta de **verificação**. Para tal utiliza-se o caminho: `File > Export Queries`.

Assim as consultas são salvas em um arquivos com extensão `Q`. Quando um dado arquivo de modelo de um sistema chamado JOBBER.XML for aberto, o respectivo arquivo JOBBER.Q (caso exista) será aberto. Alternativamente é possível abrir separadamente um arquivo de consulta específico.

3.6 Variáveis

Outra característica do UPPAAL é a possibilidade de incluir variáveis nos sistemas. As variáveis podem ser declaradas e atualizadas. Para tal, imagine o seguinte cenário no exemplo da linha de produção:

> Suponha que os trabalhadores param quando completam **10** trabalhos realizados (em conjunto).

Para tal o modelo anterior é estendido com a criação de variáveis: `J`, `jobs`, conforme apresentado pelas Figuras 18 e 19. Observa-se que as variáveis criadas são **globais**. Caso desejado é possível criar variáveis **locais**, por exemplo variáveis visíveis apenas dentro do modelo

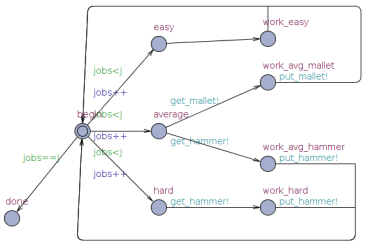

Figura 18: Criação de variáveis - modelo

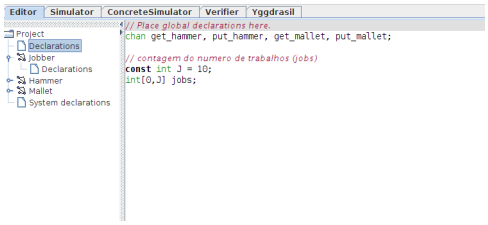

Figura 19: Criação de variáveis - declaração

`Jobber`. Para tal, basta usar o *Declarations* dentro do modelo `Jobber`, conforme ilustra a coluna à esquerda na Fig. 19.

Por meio da criação dessas variáveis é possível contar quando qualquer tipo de serviço é realizado por qualquer um dos dois trabalhadores. Note ainda que a condição `jobs < J` é colocada no campo *Guard* de cada aresta do modelo. Ao passo, que o incremento de variável: `jobs++` é colocado no campo *Update* de cada aresta do modelo (conforme ilustra a Fig. 18).

Outra alteração no modelo ocorre com a adição da *location* denominada **done**. Essa *location* é atingível quando a quantidade de trabalhos é igual a **10**. A Fig. 20 ilustra a simulação do modelo com **variáveis**, onde agora é possível observar as mudanças que ocorrem nas variáveis do sistema. No caso do modelo aqui apresentado, a variável `jobs` é observada.

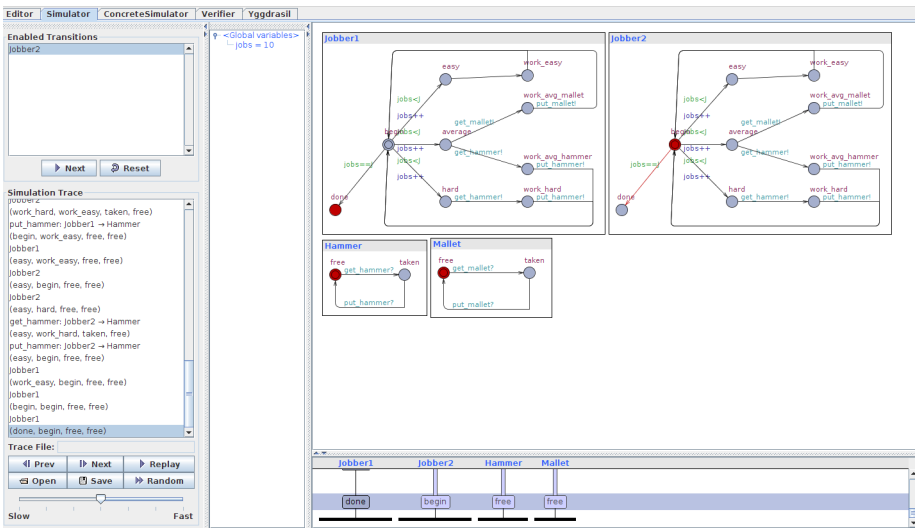

Figura 20: Criação de variáveis - simulação

3.7 Tempo e relógios

Para diversos tipos de problemas mais complexos faz-se necessário utilizar restrições de relógios.

A Fig. 21 ilustra o Autômato criado para modelar uma esteira que recebe 10 serviços (ou trabalhos). Neste novo cenário tem-se as seguintes suposições para o modelo:

- Um trabalhador necessita (no mínimo) **5** segundos para realizar um trabalho **fácil**,
- **10** segundos para um trabalho **médio** com **martelo**,
- **15** segundos para um trabalho **médio** com **marreta**,
- e **20** segundos para um trabalho **difícil** (necessariamente com **martelo**).
- Os trabalhos chegam via esteira na seguinte ordem: `H, A, H, H, H, E, E, A, A, A`
- onde, `E` significa *Easy* (fácil), `A` significa *Average* (médio), `E` significa *Hard* (difícil).

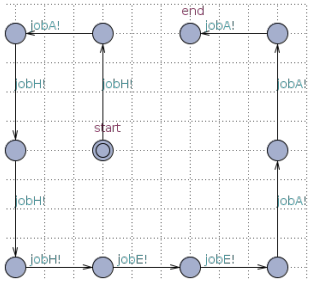

Figura 21: Autômato para modelagem de uma esteira com 10 trabalhos

Relógios e limitantes inferiores

A Fig. 22 ilustra uma extensão do autômato temporal, considerando agora a inclusão de relógios (ou *clocks*) para determinar aspectos temporais na realização dos serviços.

Em relação aos limitantes inferiores utilizam-se *guards* para especificá-los no **UPPAAL**. Note na Fig. 22 a definição dos limitantes inferiores (*representados pelas condições que saem das arestas de cada tipo de trabalho*). Esses valores (*5, 10, 15, 20*) são os tempos estabelecidos na descrição do modelo logo no ínicio da Seção 3.7.

Adicionalmente, para utilizar os limitantes inferiores e superiores no UPPAAL é necessário definir os *clocks* do sistema. Um *clock* é um tipo especial de variável, cujo domínio consiste nos números reais não-negativos. Assim como outras variáveis, os *clocks* podem ser declarados tanto como variáveis globais (de forma que podem ser testadas e atualizadas por todos autômatos), como variáveis locais (sendo testadas e atualizadas apenas um autômato).

No estado inicial (do sistema) todos *clocks* valem 0. Quando um autômato está esperando em uma dada *location* e o tempo passa, então todos valores dos *clocks* são incrementados. Quando t unidades de tempo passam, então os valores de todos os *clocks* no modelo são incrementados com t.

A Fig. 22 apresenta o modelo com extensões, novos canais de sincronização e também com a declaração de uma variável de *clock* x. Para tal, a linha de código (abaixo) deve ser adicionada na parte de de *Declarações* do projeto.

```
clock x;
```

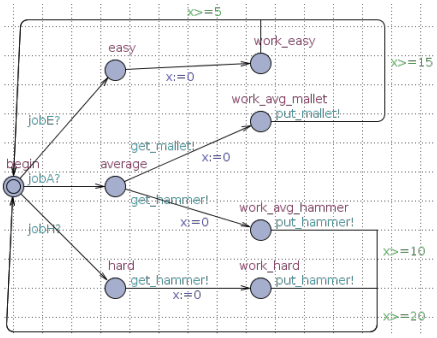

Figura 22: Autômato com utilização de clocks (relógios)

No modelo resultante, `Jobber1` e `Jobber2` possuem um *clock* local x, o qual armazena quanto tempo é usado para concluir um determinado serviço. Cada momento que um trabalhador inicia um novo serviço, o seu *clock* local é iniciado com *zero*. Para conseguir obter a quantidade de tempo total que passou, faz-se necessário criar uma variável de *clock* global, nomeada `now`, a qual nunca é "resetada". Com isso, tem-se as seguintes declarações globais:

```
chan JobE, JobA, JobH, get_mallet, get_hammer, put_mallet, put_hammer;

clock now;
```

Verificação de Propriedade

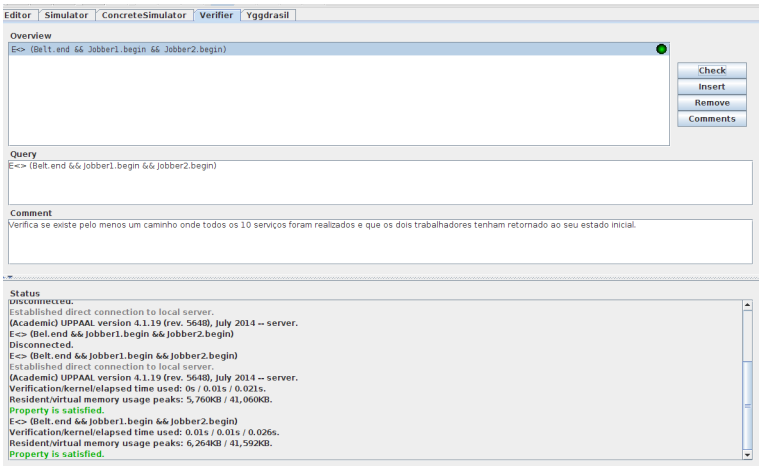

Figura 23: Verificação de propriedade formal

A Fig. 23 apresenta a verificação formal da seguinte propriedade:

```
E<> (Belt.end && Jobber1.begin && Jobber2.begin)
```

Essa propriedade representa a seguinte consulta:

existe um estado, onde todos os 10 serviços foram realizados, e os dois trabalhadores retornaram ao seus estados iniciais?

As Figuras 24 e 25 apresentam os respectivos Diagramas de Sequência para o escalonamento de serviços entre os trabalhadores do modelo aqui apresentado. Por meios desses diagramas é possível identificar quais os serviços foram atribuídos a cada um dos trabalhadores.

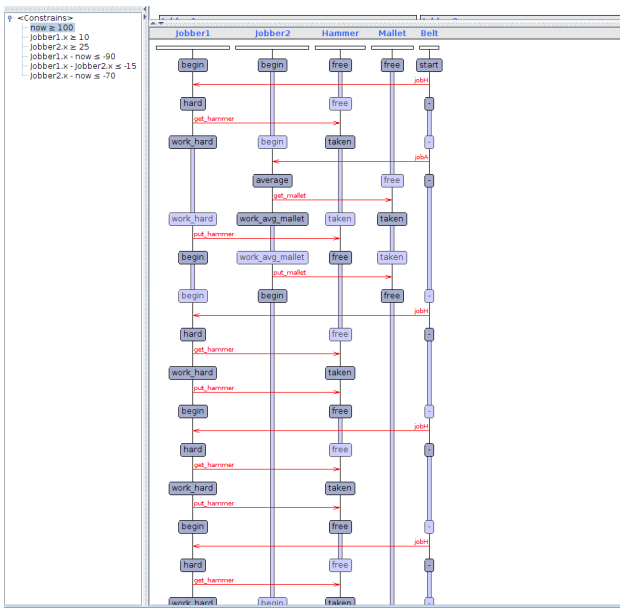

Figura 24: Escalonamento de serviços - Diagrama de sequência - parte 1

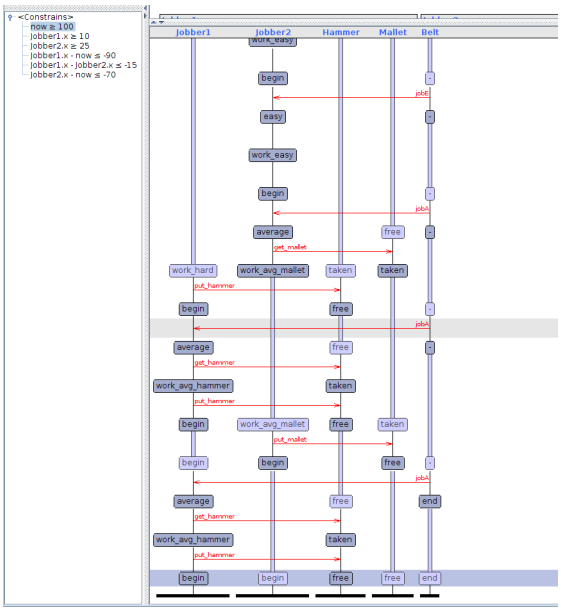

Figura 25: Escalonamento de serviços - Diagrama de sequência - parte 2

Limitantes superiores de tempo

Foi visto que é possível usar limitante inferior de tempo por meio de restrições específicas nos *guards*. Por exemplo, a restrição $x \geq 5$ na transição de `work_easy` para `begin` indica que esta transição só pode ser realizada quando trabalhador passar, pelo menos, 5 unidades de tempo no *location* `work_easy`.

Porém é possível perceber que também são necessários os limitantes superiores. Considere o seguinte cenário no exemplo da linha de produção:

- Suponha que um trabalhador necessita de no máximo 7 segundos para executar um *trabalho fácil*,
- 12 segundos para um trabalho *médio* com martelo,
- 17 segundos para um trabalho *médio* com marreta,

- 22 segundos para um trabalho *difícil* (necessariamente com martelo).

A partir disso é possível questionar-se: *Quanto tempo (no máximo) os dois trabalhadores necessitam para completar os 10 serviços*?

No UPPAAL é possível especificar limitantes superiores de tempo por meio de **invariantes**. Clicando em uma *location*, uma janela com o campo *Invariant* aparece. Seguindo o exemplo anterior é possível definir a invariante $x \leq 7$ na *location*: `work_easy`. Se mais de 7 unidades de tempo já passaram, então o trabalhador já deixou a *location*: `work_easy`.

Na Fig. 26, os limitantes superiores de tempo: $7, 12, 17$ e 22 foram adicionados, respectivamente, nas *locations*: `work_easy`, `work_avg_hammer`, `work_avg_mallet` e `work_hard`.

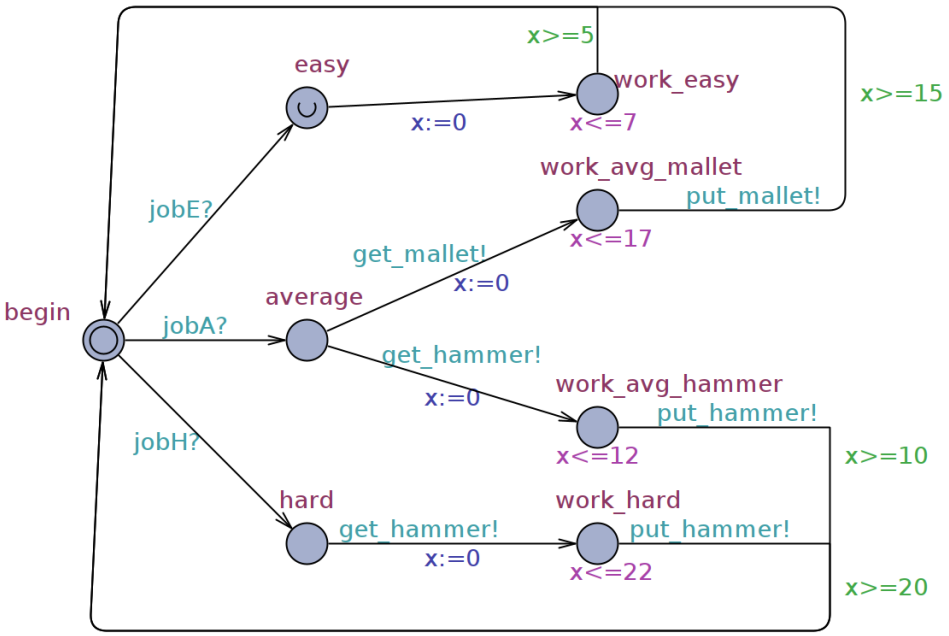

Figura 26: Exemplo - Jobber com invariantes (limitante superior)

Adicionalmente, deseja-se colocar um limitante superior igual a 0 na *location* `easy`. Isto modela a situação onde um trabalhador que tem um "trabalho fácil", pode começa-lo imediatamente. Mas, a ferramenta UPPAAL ainda disponibiliza outra maneira de estabelecer limitantes de tempo. Para tal, é necessário colocar a respectiva *location* como `Urgent`.

- Se uma *location* é dita **urgente** (`Urgent`), então isso significa que o tempo não pode passar quando se encontra neste tipo de *location*.
- Portanto, uma dada transição para outra *location* (a partir da *location*: `Urgent`) irá ocorrer imediatamente (sem passagem de tempo).

De forma similar ao que ocorre com *locations*, os canais de sincronização também podem ser ditos **urgentes**.

- Quando se tem um canal de sincronização **urgente**, isso significa que sempre que uma sincronização ocorre por meio desse canal,
- o tempo **não** pode avançar,
- e uma transição deve ser "tomada" **imediatamente**.

22 Considerando o modelo visto na Fig. 26. É possível especificar essa alteração nos canais por meio de mudanças nas respectivas declarações, como segue abaixo:

```
urgent chan jobE, jobA, jobH, get_mallet, get_hammer;

chan put_mallet, put_hammer;
```

Essa mudança para utilizar canais de sincronização urgentes faz com que quando um dado serviço estiver na linha de produção, ele necessariamente será atribuído a um trabalhador.

É importante observar que há uma diferença sutil entre *locations* **urgentes** e *canais de sincronização* **urgentes**. Caso uma *location* seja definida como **urgente**, mas não tenha um canal de sincronização na aresta dessa *location* **urgente**, o sistema pode entrar em "*time deadlock*", onde não é possível progredir em relação ao tempo no sistema. Por isso, a determinação de *locations* e *canais de sincronização* **urgentes** deve ser feita com certo cuidado em relação ao comportamento completo do sistema.

Outra observação em relação ao modedo diz respeito as "janelas de tempo" definidas para cada tipo de serviço. Por exemplo, um trabalho **fácil** pode ser realizado entre **5** e **7** unidades de tempo. Note que essa escolha do tempo exato para realização de um dado trabalho **fácil** é completamente *não determinística*. Um dado trabahador pode realizar o serviço em **6** unidades de tempo, outro em **5.764** e ainda um outro poderia levar **6.8** unidades de tempo. Essa capacidade de ter transições de tempo *não determinísticas* é uma característica muito útil em autômatos temporais, já que é possível descrever sistemas com um alto nível de abstração e especificar propriedades de comportamento do sistema modelado.

Verificaçãos de Propriedades Relacionadas com Tempo

Agora é possível fazer verificação de propriedades considerando a última versão do modelo. Note que a verificação das propriedades a seguir tem como objetivo verificar qual é o tempo utilizado para fazer a lista de trabalhos na esteira da linha de produção.

O mecanismo usado aqui é simples, onde a propriedade é escrita estimando um valor de tempo para a variável global do sistema `now`.

Na primeira tentativa o valor é igual a **200** unidades de tempo e a propriedade é satisfeita.

✓ `A[] now>=200 imply (Belt.end && Jobber1.begin && Jobber2.begin)`

A segunda tentativa tem `now` igual a **150** e igualmente é satisfeita.

✓ `A[] now>=150 imply (Belt.end && Jobber1.begin && Jobber2.begin)`

Em uma terceira tentativa, tendo `now` igual a **100 não** é satisfeita.

X `A[] now>=100 imply (Belt.end && Jobber1.begin && Jobber2.begin)`

O processo estende-se até encontrar os valores de `now` nos limites iguais a **127** e **126**, respectivamente.

✓ `A[] now>=127 imply (Belt.end && Jobber1.begin && Jobber2.begin)`

Para `now` igual a **127** a propriedade é satisfeita. Porém, para `now` igual a **126** a propriedade **não** é satisfeita.

X `A[] now>=126 imply (Belt.end && Jobber1.begin && Jobber2.begin)`

4 Exemplo: Controle de Trens

Nesta Seção é representado um problema envolvendo o acesso de trens a uma dada ponte. Na ilustração deste problema no UPPAAL tem-se a descrição de algumas características e funcionalidades da ferramenta.

4.1 Descrição

Neste problema tem-se uma certa quantidade de trens em trilhos separados, mas que devido a questões de economia devem cruzar uma ponte, a qual possui um único trilho. E, portanto, faz-se necessário controlar o acesso dos diferentes trens a essa ponte.

Almeja-se com modelo no UPPAAL criar uma representação do comportamento dos trens, de forma que um controlador determine quando um dado trem deve parar ou prosseguir pela ponte, de forma que sejam evitadas colisões entre os trens.

Este problema foi originalmente apresentado por Yi, Petterson e Daniels (1994), denominado como **Simple Railway Control System** (ou *Sistema de Controle para Ferrovias Simples*).

A descrição do sistema envolve a noção de tempo, no sentido de limitantes para acessar a ponte, cruzar ponte e outras características do modelo, conforme as definições que seguem.

- Inicialmente, os trens estão longe o suficiente da ponte e o sistema está num estado seguro (*Safe*).
- Em algum momento, um trem aproxima-se da ponte (estado *Approaching*).
- O Controlador tem **10** unidades de tempo para parar o trem. Após isso não é seguro mandar o trem parar. Logo, o mesmo deve prosseguir (adiante).
- Prosseguindo em direção ao cruzamento, o trem levará no máximo **20** unidades de tempo para chegar na ponte.
- Caso o trem seja parado (estado *Stop*), então em algum momento o trem deverá reiniciar o movimento para cruzar a ponte (estado *Start*) e levará entre **7** e **15** unidades de tempo para chegar no cruzamento.
  - Note que o trem pode ser parado pelo Controlador em qualquer tempo menor que **10** unidades, logo existe um comportamento não determinístico do sistema.
- O trem leva de **3** a **5** unidades de tempo para cruzar a ponte.
- Por razões de segurança, apenas um único trem deve cruzar a ponte por vez.
- Após cruzar a ponte, o trem retorna ao estado seguro (*Safe*).

O modelo a ser construído possui um *template* **Trem**, do qual são derivadas instâncias de Trens, conforme a quantidade de trens do sistema. Além disso, há um controlador de acesso a ponte.

4.2 Definição de Tipos

Na definição do autômato Controlador será utilizada uma fila para gerenciar a chegada dos trens que devem permanecer parados. Os trens são distinguidos por um único identificador cujos valores são definidos pelo número de total de trens. Para essas definições são usados tipos de dados e estruturas específicas para representação dos dados. A fila é representada por uma estrutura contendo um vetor e o respectivo tamanho.

Na linguagem do UPPAAL os tipos são declarados da seguinte forma:

```
typedef type name;
```

onde o tipo (*type*) pode ser um valor inteiro (`int[min, max]`) ou, então uma estrutura declarada da seguinte forma:

```
struct { type1; typ2; ... }
```

Declarações Globais

Neste modelos tem-se as seguintes definições globais, referentes à variáveis e canais.

```
const int N = 2;
typedef int[0,N-1] id_t;

// canais
chan appr[N], stop[N], leave[N];
urgent chan go[N];
```

Declarações do Sistema

O modelo no UPPAAL possui as seguintes declarações de sistema, as quais dizem respeito ao modelo Train e Gate (autômato Controlador). Note que o modelo Train tem um parâmetro, o qual é uma constante inteira do tipo `id_t`, que representa o identificador do respectivo trem.

```
Train1 = Train(0);
Train2 = Train(1);

system Train1, Train2, Gate;
```

As Figuras 27 e 28 ilustram os autômatos temporais, respectivamente, referentes ao Trem e ao Controlador de acesso à ponte.

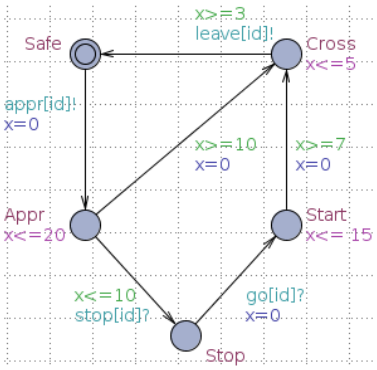

Figura 27: Autômato Temporal - Modelo de Trem

4.3 Definição de Funções

Este modelo possui a construção de funções específicas no autômato do Controlador de acesso à ponte. No modelo são criadas funções para manipular a fila de trens que desejam cruzar a ponte. Para tal são criadas duas variáveis: um vetor `list` com os identificadores dos trens e uma variável inteira `len` com a quantidade máxima de trens no modelo.

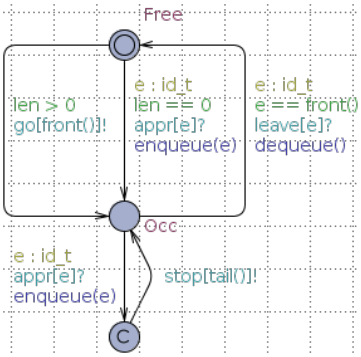

Figura 28: Autômato Temporal - Controlador

```
id_t list[N];
int[0,N] len;

// Put an element at the end of the queue
void enqueue(id_t element)
{
        list[len++] = element;
}

// Remove the front element of the queue
void dequeue()
{
        int i = 0;
        len -= 1;
        while (i < len)
        {
                list[i] = list[i + 1];
                i++;
        }
        list[i] = 0;
}

// Returns the front element of the queue
id_t front()
{
   return list[0];
}

// Returns the last element of the queue
id_t tail()
{
   return list[len - 1];
}
```

4.4 Propriedades

Aqui são descritas algumas propriedades que podem ser verificadas por meio do UPPAAL.

Atingibilidade

As propriedades de atingibilidade (do inglês, *reachability*) são da forma:

```
E<> p
```

e significam: *existe algum caminho onde p ocorre em algum estado.*

Essas propriedades são usadas para verificar os estágios iniciais de modelos e propriedades básicas de comportamente e funcionamento.

Para o exemplo do Controle de Trens seguem alguns exemplos de propriedades que podem ser verificadas:

```
E<> Gate.Occ
E<> Train1.Cross
E<> Train2.Cross
E<> Train1.Cross and Train2.Stop
```

Segurança

As propriedades de segurança (do inglês, *safety*) são da forma:

```
A[] p
E[] p
```

e significam, respectivamente: *para todos caminhos e para todos estados ocorre p*; *existe um caminho onde p sempre ocorre.*

Essas propriedades são usadas para verificar propriedades consideradas críticas ao correto funcionamento do sistema. É necessário atenção especial em definir tais tipos de propriedades, em virtude do problema de explosão de estados da técnica de Model Checking.

Para o exemplo do Controle de Trens é possível formular a seguinte propriedade de segurança:

```
A[] not deadlock
```

Vivacidade

As propriedades de vivacidade (do inglês, *liveness*) expressam que algo bom necessariamente irá ocorrer e possuem a seguinte forma:

```
A<> p
p --> q
```

A primeira forma significa: *para todos caminhos necessariamente ocorre p.*

A segunda deve ser lida como p tem como consequência q. Essa propriedade é na verdade uma equivalência da seguinte fórmula

```
A[] (p imply A<>q)
```

A qual significa: *em qualquer momento que p ocorre para algum estado, então q irá sempre ocorrer em algum momento para todos caminhos que iniciam pelo estado com p.*

Para o exemplo do Controle de Trens as seguintes propriedades podem ser verificadas:

```
A<> Train1.Appr imply Train1.Cross
```

```
A<> Train2.Appr imply Train2.Cross
```

5 Exemplos de Sistemas Autônomos

Os exemplos dessa seção encontram-se no seguinte repositório: `https://github.com/laca-is/uppaal`

6 Referências

DAVID, A. LARSEN, K. G. **More Features in Uppaal**.

MILNER, R. **Communications and Concurrency**. Prentice-Hall International, Englewood Cliffs. 1989.

VAANDRAGER, F. **A First Introduction to Uppaal** (Representation and Mind Series). The MIT Press. 2008.

BAIER, Christel; KATOEN, Joost-Pieter. **Principles of Model Checking** (Representation and Mind Series). The MIT Press. 2008.

YI, W. PETTERSON, P. DANIELS, M. **Automatic Verification of Real-Time Communicating Systems by Constraint-Solving**. In Dieter Hogrefe and Stefan Leue, editors, Proc of 7th Int. Conf. on Formal Description Techniques, pages 223-238, North-Holland, 1994.