LoTuS: uma Ferramenta Gráfica Extensível para Modelagem, Análise e Verificação de Modelos LTS e PLTS

**Emerson Correia, Bruno Barbosa, Lucas Vieira,
Ranniery Jesuíno, Messias Filho, Paulo Henrique M. Maia**

[1]Universidade Estadual do Ceará
Av. Dr. Silas Munguba, 1700, Campus do Itaperi – Fortaleza - CE

{emerson.lima,bruno.barbosa,lucas.vieira}@aluno.uece.br,

{ranniery.jesuino,messias.filho}@aluno.uece.br, pauloh.maia@uece.br

Abstract. *This paper introduces LoTuS, a tool for graphical modeling, analysis and verification of software behavior using LTS and PLTS. Its main contribution is fourfold: facilitating the formal modeling process through a drag and drop mechanism to design both non-probabilistic and probabilistic models; allowing the model generation from other sources, such as UML sequence diagrams and execution traces; providing a set of techniques for model analysis, such as simulation, execution, deadlock detection and reachability properties probabilistic verification; finally, providing an API that allows developers to add new functionalities by creating plugins. The tool has been evaluated in terms of usability, performance and through a case study in which its main features have been exercised.*

Resumo. *Este artigo apresenta LoTuS, uma ferramenta para modelagem gráfica, análise e verificação de comportamento de software usando LTS e PLTS. Suas principais contribuições são: facilitar o processo de modelagem formal através de um mecanismo de drag and drop que permite criar tanto modelos não probabilísticos como probabilísticos; permitir a geração de modelos a partir de outras fontes, como diagramas de sequencia da UML ou rastros de execução; prover um conjunto de técnicas de análise de modelos, como simulação, execução, detecção de deadlock e verificação probabilística de propriedades de alcançabilidade; e por fim, fornecer uma API para que desenvolvedores possam adicionar novas funcionalidades através da criação de plugins. A ferramenta foi avaliada em termos de sua usabilidade e desempenho e através de um estudo de caso no qual suas principais funcionalidades foram exercitadas.*

1. Introdução

Modelagem de comportamento é uma técnica que permite desenvolvedores descrever abstratamente e raciocinar sobre o comportamento desejado de um sistema de software. Por sua simplicidade, o custo do desenvolvimento de modelos é considerado pequeno quando comparado ao custo de construir um sistema por completo. Além disso, a modelagem traz como benefício a facilidade de entendimento sobre como o software deve se comportar em tempo de execução e a possibilidade de antever possíveis falhas em tempo de projeto [Magee and Kramer 2006].

Para modelar e analisar o comportamento do sistema, geralmente é necessário especificá-lo formalmente e usar uma ferramenta automatizada que dê suporte à análise desejada. Métodos formais oferecem uma precisão matemática na análise de programas que dão a possibilidade de certificar a corretude do programa, algo que o teste e a revisão informal não conseguem. O objetivo maior dos métodos formais é aumentar a qualidade do software ao tornar os requisitos, projeto e implementação do sistema mais explícitos e, portanto, deixando os defeitos mais facilmente detectados [Geer 2011].

O comportamento de sistemas é geralmente modelado utilizando um formalismo de máquina de estados, como o Labelled Transition Systems (LTS) [Keller 1976], onde cada modelo se refere a um componente do sistema que interage com os demais componentes através de ações compartilhadas, Probabilistic LTS (PLTS) e Discrete-time Markov Chains (DTMC), no caso de modelagem de comportamento probabilístico. Há várias ferramentas de modelagem de comportamento que usam tanto LTSs, como LTSA [Magee and Kramer 2006], SPIN [Holzmann 1997], and LTSMin [Blom et al. 2010], ou DTMCs, como PRISM [Kwiatkowska 2007]. Apesar de existirem outros formalismos, como as Redes de Petri [Peterson 1981] e suas derivações estocásticas, este trabalho foca no uso de LTS e PLTS para a modelagem e análise de comportamento tradicional e probabilístico de software, respectivamente.

A modelagem formal é útil para sistemas grandes e complexos, para os quais métodos formais têm sido aplicados com sucesso [Woodcock et al. 2009][Broadfoot 2005]. Contudo, esses métodos são geralmente entendidos como de difícil aplicação e requerem significante experiência matemática quando considerados sistemas não-críticos, como sistemas de informação *web* [Hinchey et al. 2008]. Além disso, os usuários necessitam aprender a linguagem de especificação formal implementada por essas ferramentas, como FSP e PROMELA. Isto pode ser particularmente desmotivante em cursos introdutórios de modelagem de software ou quando o usuário deseja apenas fazer um esboço do possível comportamento do sistema e realizar algumas análises, uma vez que essas linguagens são difíceis de ler e entender.

Para amenizar esse problema, algumas ferramentas fornecem um ambiente de análise e modelagem gráficas, como UPAAL [Behrmann et al. 2004], MaTeLo [Feliachi and Le Guen 2010] e PIPE2 [Dingle et al. 2009]. Contudo, essas ferramentas apresentam algumas limitações, como o fato de não disponibilizarem API para que desenvolvedores possam estendê-las, apresentar poucas formas de geração de modelos (usualmente apenas através de interface gráfica ou linguagem de espeficicação formal), o que restringe sua utilização, e possuir interface de difícil uso.

Para preencher essa lacuna, este trabalho propõe LoTuS, uma ferramenta de código aberto e estensível para modelagem gráfica e análise de comportamento de software utilizando modelos LTS e PLTS. Os principais benefícios da ferramenta são: (i) oferecer um mecanismo *drag and drop* que torna a modelagem mais fácil e intuitiva, possibilitando também ao usuário incrementar as transições com guardas e probabilidades, o que permite facilmente evoluir os modelos para se tornarem probabilísticos; (ii) permitir a geração de modelos a partir de outras fontes, como a diagramas de sequência UML, rastros de execução, ou outras ferramentas de modelagem, como o LTSA; (iii) disponibilizar algumas técnicas de análises de modelo, como detecção de *deadlocks*, simulação

e execução, além da verificação probabilísticas de propriedades de alcançabilidade (*reachability properties*); (iv) fornecer uma API que permite desenvolvedores adicionarem novas funcionalidades através de *plugins*.

O restante deste artigo está divido da seguinte forma: na seção 2 apresentamos os principais conceitos relacionados aos modelos LTS e DTMC. Seção 3 dá uma visão geral sobre a ferramenta LoTuS e detalha sua arquitetura. A seção 4 descreve as principais funcionalidades do LoTuS, enquanto a seção 5 apresenta as avaliações realizadas. Os principais trabalhos relacionados a este são discutidos na seção 6. Por fim, a seção 7 traz as conclusões e trabalhos futuros.

2. Fundamentação Teórica

Definição 1 [LTS]. Um *Labelled Transition System* (LTS) é uma estrutura L=(S, A, Δ, q), onde S é um conjunto de estados finitos, A é um conjunto de rótulos (*labels*) de eventos, também conhecido como alfabeto, Δ: ($S \times A \times S$) define as transições rotuladas entre estados, e $q \in S$ é o estado inicial.

Uma execução de um LTS é a sequência de estados e rótulos que podem ser obtidos a partir do estado inicial do LTS. Um rastro (*trace*) é a sequência de rótulos observáveis que são produzidos pela execução de um LTS, enquanto um caminho (*path*) é a sequência de estados resultantes da exclusão dos rótulos de uma execução. Dizemos que um estado s_1 alcança outro estado s_2 se existe um caminho a partir de s_1 até s_2.

Dado dois LTSs L_1 e L_2, denotamos por $P=(L_1 \parallel L_2)$ o LTS resultante da composição paralela de L_1 e L_2 que modela o comportamento conjunto dos dois LTSs. O alfabeto de P é a união do alfabeto de L_1 e L_2 e seus estados podem ser vistos como pares de estados (s_1, s_2) que refletem o fato de L_1 estar no estado s_1 e L_2 estar no estado s_2.

Um LTS probabilístico (*Probabilistic LTS* - PLTS) modela um sistema cujo comportamento em cada ponto no tempo pode ser descrito por uma escolha probabilística discreta sobre várias possíveis saídas. Essencialmente, um PLTS pode ser visto como um sistema de transição de estados no qual cada transição é anotada com um valor entre 0 e 1 indicando a probabilidade de sua ocorrência.

Definição 2 [PLTS] Um PLTS é uma estrutura Q = (L, λ), onde L=(S, A, Δ, q) é um LTS e λ: $\Delta \rightarrow [0,1]$ é a função de probabilidade das transições que atribui um número real entre 0 e 1 para cada transição tal que a soma das probabilidades de todas as transições saindo de um mesmo estado é 1.

A probabilidade de um caminho de um PLTS é o produto das probabilidades de todas as transições que ligam cada par de estados consecutivos no caminho. Dado um PLTS P, é possível construir um DTMC não rotulado removendo os rótulos das transições de P.

3. Visão Geral do LoTuS

LoTuS [1] é uma ferramenta para criação, análise e verificação de modelos LTS e PLTS. Sua interface gráfica permite a construção de modelos de comportamento através de um mecanismo de *drag and drop*, ao invés do processo de compilação de uma linguagem

[1] http://gesad.uece.br/ferramentas/lotus/

formal, permitindo que usuários modelem seus sistemas de maneira simples, rápida e intuitiva. Toda a aplicação foi implementada em Java, utilizando a sua nova tecnologia para construção de interfaces ricas JavaFX, e seu código-fonte é aberto para estudo e melhorias e está disponível publicamente no GitHub[2]. Nesta seção descrevemos o ambiente de modelagem da ferramenta e sua arquitetura.

3.1. O Ambiente de Modelagem

A tela principal do LoTuS é composta pelo menu principal, a barra de ferramentas, e três painéis: o de projeto, o de edição, e o de propriedades, como mostra a Figura 1. O painel de projeto mostra os projetos abertos e seus respectivos componentes. Um projeto do LoTuS é uma coleção de componentes que, por sua vez, são modelados como LTSs ou PLTSs. No exemplo da Figura 1, o projeto "SistemaEnvioMensagem" representa um sistema de envio de mensagens, cujo componentes que o constitui são "Sistema" e "Usuario".

O duplo clique em um componente mostra sua modelagem no painel de edição. A modelagem consiste em escolher os botões da barra de ferramentas localizadas na parte superior do painel que irão realizar a tarefa desejada (inserir estado ou transição, apagar, mover, etc). Por exemplo, para adicionar um estado, clica-se no botão de estado e, em seguida, no local da área de edição no qual se deseja colocá-lo. Transições entre estados são criadas selecionando-se a opção de transição, clicando no estado de origem, segurando o mouse e soltando-o sobre o estado destino da transição.

Na Figura 1, o comportamento do componente "Sistema" é mostrado, no qual a tela de login é apresentada inicialmente. O usuário informa o login e, se a senha digitada for diferente de 123, volta à tela de login. Caso seja igual, então ele pode enviar mensagens até sair do sistema ou até que uma falha no envio aconteça. Por padrão, LoTuS identifica o estado inicial com rótulo 0 e cor amarela, enquanto os estados comuns são identificados por rótulos com numeração crescente e cor azul. Além desses estados, a ferramenta também permite identificar um estado como final, que representa um ponto de término da execução do sistema com sucesso, ou erro, indicando que uma falha ou algo errado aconteceu. O estado final é representado por um estado com rótulo E, cor cinza, e um círculo interno, enquanto o estado de erro possui rótulo -1 e cor vermelha. Não é permitido que esses estados tenham transições de saída. Para alterar o tipo de estado, basta clicar com o botão direito sobre o estado e escolher a opção desejada no menu *popup*. Também é possível alterar a cor do estado sem, conteudo, alterar o seu tipo.

Na Figura 1, a transição *(2,loginValido,3)* está selecionada e suas propriedades são mostradas no painel de propriedades. O campo *action* representa o rótulo da transição, o campo *guard* representa a condição que deve ser satisfeita para que a ação ocorra, e o campo *probability* indica a probabilidade com a qual a transição ocorre. Nesse exemplo, a ação *loginValido* só acontece se a senha digitada for igual a 123. O modelo não contém probabilidades.

Um diferencial do LoTuS é a possibilidade de se criar um estado composto, que resulta do agrupamento de outros estados. Essa funcionalidade é útil para abstrair comportamentos específicos, tornando o modelo mais abstrato ainda. Para tanto, basta selecionar

[2]https://github.com/lotus-tool/lotus-tool

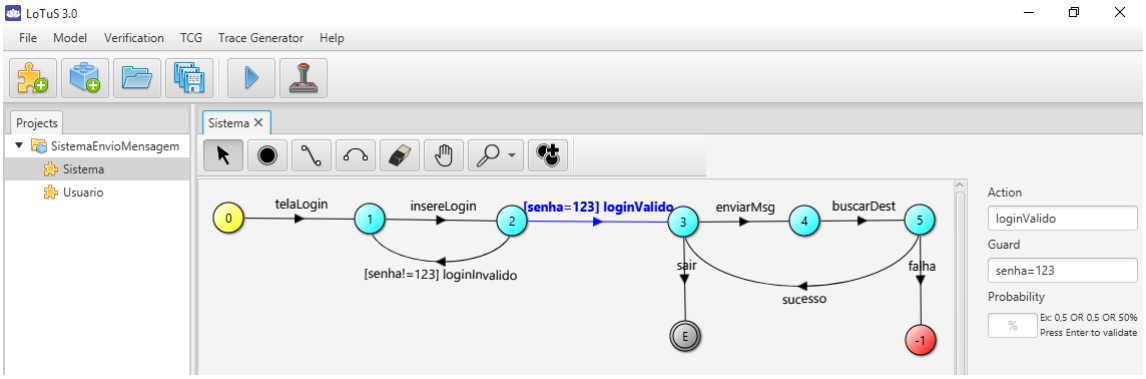

Figura 1. Tela principal do LoTuS

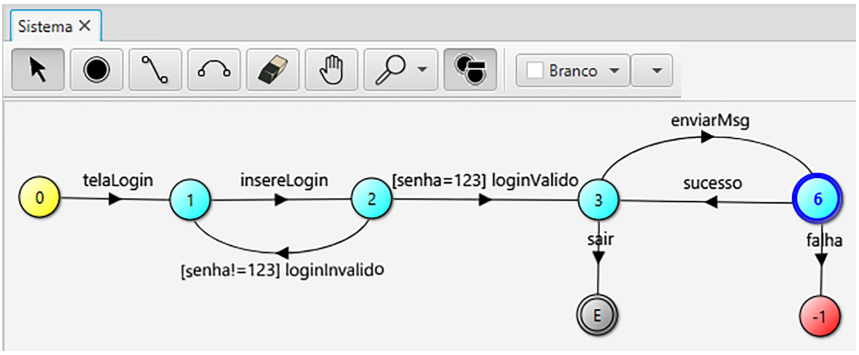

Figura 2. LTS resultante do agrupamento dos estados 4 e 5

o conjunto de estados que se deseja compor e clicar no botão de composição de estado na barra de ferramentas (o mais à direita, que contém dois círculos e o símbolo "+"). Por exemplo, na Figura 2, o estado 6 resulta do agrupamento dos estado 4 e 5 da Figura 1.

Note que o estado 6 é um pouco maior que os outros e possui um outro círculo no seu interior, indicando que ele é um estado composto. Neste caso, ele encapsulou o comportamento de buscar o destinatário após enviar um mensagem, representado pela transição *(4,buscarDest,5)*. Agora, uma vez que a mensagem é enviada, apenas sabe-se que ela falha ou que é entregue com sucesso. Para voltar ao comportamento incial, basta selecionar o estado composto e clicar novamente no botão de composição de estado, cuja imagem foi alterada para dois círculos e o símbolo "-", como mostra a Figura 2.

Um projeto LoTuS é salvo como um arquivo XML, cujo esquema pode ser encontrado no *website* da ferramenta. Desta forma, um desenvolvedor pode também criar modelos a partir de uma outra ferrmenta ou aplicação através da conversão para o formato XML utilizado pelo LoTuS.

3.2. Arquitetura

O projeto LoTuS visa, dentre outros aspectos, a construção de uma comunidade formada por estudantes e profissionais em torno do tópico de modelagem e análise de comportamento de software. Por isso, é um projeto de código-aberto, no qual os membros da comunidade podem contribuir com diferentes técnicas para análise, simulação e transformação de modelos por meio de *plugins*.

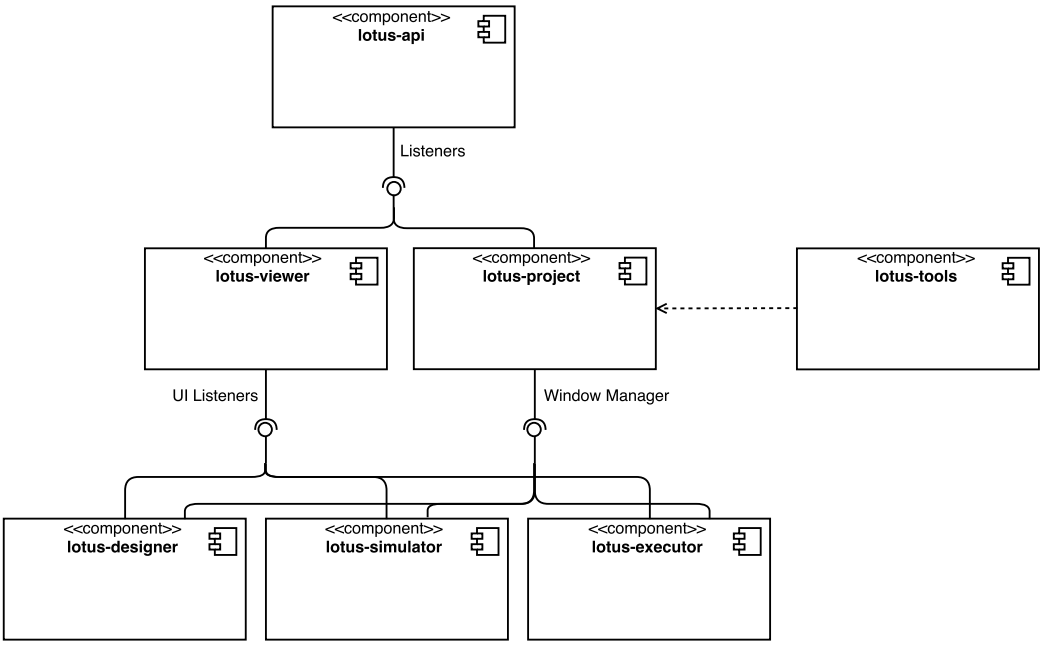

Figura 3. Arquitetura do LoTuS

Para viabilizar essa contribuição, a arquitetura do LoTuS (Figura 3) foi desenvolvida já visando a extensibilidade da ferramenta, concretizada por meio da modularização e disponibilização de uma API para *plugins*. Diversos plugins foram construídos e já vêm embutidos na própria ferramenta, como o gerenciamento de projetos, a visualização, edição, simulação, e execução de modelos, e a integração com outras ferramentas através de importação e exportação de formatos. O gerenciamento dos *plugins* é bastante simples, de modo que *plugins* são ativados pela sua presença no diretório de extensões da ferramenta.

A construção de *plugins* consiste basicamente em desenvolver um módulo Java (JAR) utilizando as classes do componente *lotus-api*. Também é possível estender ou reutilizar funcionalidades de *plugins* já existentes. Um breve tutorial de como desenvolver *plugins* pode ser encontrado no *website* da ferramenta, porém a melhor forma de aprender a construir um plugin é observando o código-fonte dos *plugins* já existentes.

O componente *lotus-api* possui todas as classes que modelam um projeto LoTuS e as classes básicas comuns para definição de *plugins* e interação com a interface gráfica principal. Outros componentes irão observar mudanças nos projetos, modelos, estados e ações, registrando *listeners*. O componente *lotus-viewer* apenas disponibiliza um componente gráfico JavaFX que exibe um modelo LTS ou PLTS que reflete automaticamente toda e qualquer mudança nas classes do tipo *Plain Old Java Object* (POJO) do modelo. O componente *lotus-project* possui vários *plugins* relacionados ao gerenciamento de projetos pela aplicação, como abrir e salvar um projeto e adicionar ou remover um modelo ao projeto.

O componente *lotus-tools* possui *plugins* relacionados à importação e exportação de projetos construídos no LoTuS para ferramentas relacionadas. O componente *designer* é responsável pelo editor *WYSIWYG* (*What You See Is What You Get*) que permite que o usuário construa ou modifique um modelo com recursos *drag and drop*. Os componentes

lotus-simulator e *lotus-executor* são responsáveis pela simulação e execução, respectivamente, de um modelo.

A arquitetura do LoTuS utiliza o padrão *Model-View-Controller* [Buschmann et al. 1996]. Os modelos são definidos nas classes Project, Component, State e Transition como POJOs, enquanto as visões da interface gráfica para o usuário são definidas em arquivos *FXML* ou em classes Java que estendem objetos gráficos do *JavaFX*. Por sua vez, os controladores são responsáveis pela interoperabilidade dessas duas camadas.

Vários padrões de projeto [Gamma et al. 1995] também foram utilizados para a implementação da ferramenta. O padrão *Observer*, através do qual objetos interessados em informações se registram em objetos publicadores de conteúdo, foi utilizado para desacoplar a camada de visualização do modelo permitindo que modificações nos modelos sejam propagadas aos objetos gráficos. O padrão *Command*, no qual cada ação é modelada como um objeto, foi utilizado para viabilizar o histórico das ações tomadas pelo usuário na simulação e possibilitar a opção de desfazê-los. O padrão *Strategy* foi utilizado na classe Designer para que que comportamentos (subclasses de Behavior) sejam isolados e modificados sem trazer efeitos colaterais aos outros componentes.

4. Principais funcionalidades

Esta seção descreve os principais recursos da ferramenta proposta.

4.1. Composição paralela

Dois ou mais componentes não probabilísticos de um projeto podem ser compostos em paralelo para que se obtenha o comportamento conjunto dos mesmos. LoTuS assume que se os LTSs possuem ações com o mesmo rótulo, essas ações são compartilhadas (*shared actions*) e, portanto, devem ser executadas sincronamente no comportamento paralelo. Para realizar a composição, basta selecionar no painel de projeto os componentes que serão compostos, clicar sobre a seleção com o botão direito do mouse, e escolher a opção *Parallel Composition* no menu *popup*.

4.2. Simulação

A simulação é um processo interativo pelo qual o usuário pode navegar pelo modelo. Consiste de, a partir do estado inicial, habilitar somente as transições possíveis de serem realizadas no estado selecionado, deixando para o usuário a escolha de qual transição seguir. A partir de então, o estado que recebe a transição escolhida é selecionado e o mesmo processo se repete até que um estado final ou de erro seja alcançado ou o usuário encerre a simulação. O rastro produzido pela simulação é exibido abaixo do painel de edição. As transições selecionadas na simulação são realçadas no modelo, permitindo que o usuário visualize o caminho percorrido pela simulação até então.

O usuário pode também deixar que a ferramenta escolha a próxima ação a ser realizada clicando no botão “Random Step”. Neste caso, se o modelo for probabilístico, a escolha da ação leva em conta a probabilidade de execução de cada transição de saída. Caso o modelo seja não probabilístico, então a escolha é feita não deterministicamente. A Figura 4 mostra um trecho da simulação do modelo da Figura 1.

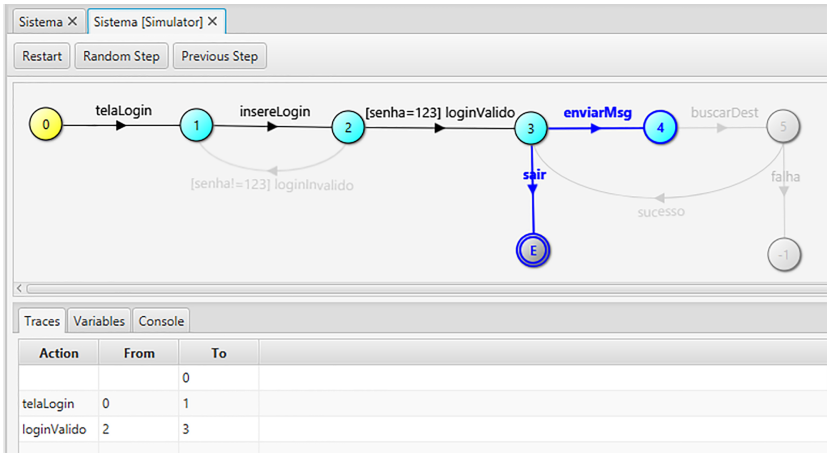

Figura 4. Tela de simulação do LoTuS

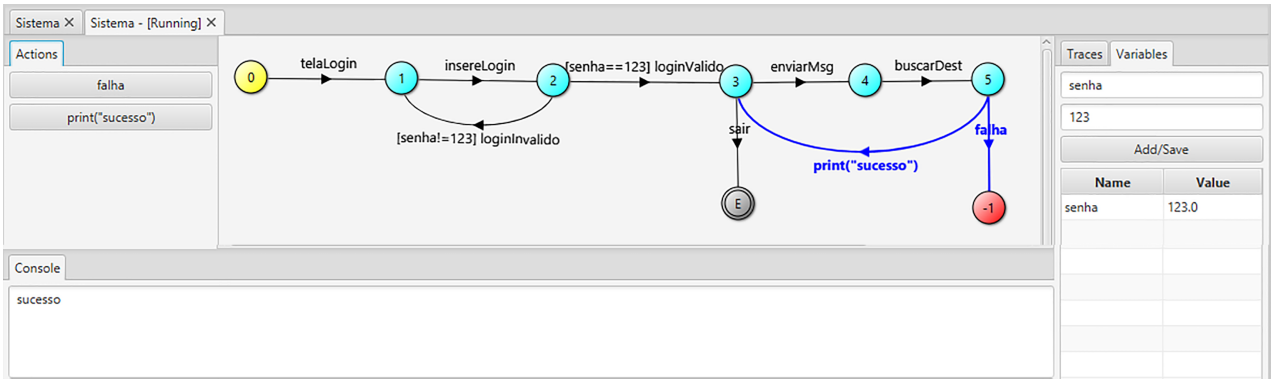

Figura 5. Tela de execução do LoTuS

4.3. Execução

Esse processo é semelhante à simulação, porém com três diferenças: a primeira é que o usuário pode especificar variáveis, que são usadas nas guardas ou nas transições, e determinar seus valores. A segunda diz respeito à habilitação das transições de saída do estado selecionado em dado momento da execução, pois só são habilitadas as transições cujas guardas forem verdadeiras ou não possuam guardas. Por fim, LoTuS também permite que transições sejam realmente executadas durante esse processo. Por exemplo, se o usuário declara uma variável `cont` e inicializa seu valor para 1, e realiza uma transição cuja ação é `cont++`, então após essa ação o valor de `cont` será 2. Além disso, LoTuS considera duas ações especiais: `print`, para impressão de valores em um console que aparece abaixo do painel de edição durante a execução, e `read`, que habilita um campo de leitura no console para que o usuário possa entrar um determinado valor.

A Figura 5 mostra a execução do modelo da Figura 1, o qual teve a ação *sucesso* alterada para *print("sucesso")* para mostrar sua execução no console na parte de baixo da figura. Note que o valor da guarda da ação *loginValido* também foi alterada para usar o sinal "==", pois ele é necessário para realizar a comparação de valores na execução. Por fim, foi inserida uma variável *senha* com valor 123, como mostrado no canto direito da imagem.

4.4. Extensão

Um recurso importante do LoTuS é sua extensibilidade. A ferramenta fornece uma API que permite que usuários contribuam implementando suas próprias funcionalidades através de *plugins*. Um conjunto de *plugins* já foi desenvolvido e vêm com a ferramenta no momento do *download*. Os principais são:

- *Test Case Generator (TCG)* [Muniz et al. 2015]: permite a geração e seleção de casos de testes abstratos funcionais e estatísticos de modelos tradicionais e probabilísticos, respectivamente. Um caso de teste abstrato é um rastro de execução de um modelo. O *plugin* implementa vários algoritmos como *All Paths*, *All Transitions* e *All One-loop Paths*. Com ele, o usuário pode selecionar a técnica que melhor se adequa para cada sistema em teste. Além disso, o *plugin* fornece uma variedade de técnicas de seleção, como o caminho mais provável, similaridade de caminhos, e propósito de testes.
- *Trace Generator*: permite a geração automática de um conjunto de rastros de execução para o modelo. Se o modelo for probabilístico, então cada rastro será gerado de acordo com as probabilidade das transições, caso contrário a próxima ação será escolhida de forma não determinística. O usuário indica a quantidade de traces desejados que serão então armazenados em um arquivo *Comma Separated Values* (CSV), e um *timer* pode ser configurado para determinar o intervalo no qual cada trace será produzido. Esse recurso é útil para simular o uso real de um sistema.
- *Probability Annotator*: permite que o usuário selecione um arquivo CSV contendo um conjunto de rastros de execução. A partir desses rastros, a probabilidade de ocorrência de cada transição é calculada e anotada no modelo, transformando-o em um PLTS. O algoritmo percorre o modelo LTS de entrada seguindo cada rastro do arquivo CSV. A partir do estado inicial e da primeira ação do rastro, o algoritmo seleciona a transição que possui ação correspondente à ação atual do rastro, incrementando o contador daquela transição. Esse processo se repete para cada ação do rastro. Ao percorrer todos os rastros do arquivo CSV, o algoritmo contou o número de ocorrências para cada transição do modelo. A probabilidade de ocorrência de cada transição é determinada pela divisão do toal de ocorrência de uma transição pela soma das ocorrências de todas as transições que saem do mesmo estado.

4.5. Transformação, importação e geração de modelos a partir de outras fontes

Um importante diferencial do LoTuS é a possibilidade de geração de modelos LTS a partir de outras fontes, dentre as quais destacamos:

- Diagrama de Sequência da UML: LoTuS permite transformar um diagrama de sequência da UML em um LTS. É possível que o diagrama contenha também os blocos adicionais *loop*, representando uma repetição, e *alt*, que representa um fluxo que só é realizado se uma certa condição for satisfeita. Além disso, a transformação do diagrama pode gerar um formato especial de LTS, usado em [Cartaxo et al. 2008] para a geração de casos de testes, podendo ser utilizado pelo plugin TCG (apresentado na seção 4.4). Por enquanto, a importação só funciona

caso o diagrama tenha sido modelado utilizando a ferramenta Astah[3] e tenha sido salvo no formato XMI (XML Metadata Interchange[4]).

- Rastros de execução: nesta funcionalidade, o usuário pode selecionar um arquivo CSV contendo diversos rastros de execução do sistema e, a partir deles, LoTuS irá gerar um possível modelo de comportamento do sistema. A heurística para a geração do modelo é bem simples e consiste em tentar criar um novo caminho no modelo para cada rastro do arquivo. Caso o rastro contenha caminhos que já estejam no modelo, então os caminhos são sobrescritos, até que uma nova ação apareça. Caso haja ações repetidas no caminho, a segunda ação é retornada para o estado final da primeira, constituindo um ciclo.
- Formatos de outras ferramentas: permite importar modelos LTS especificados em outras ferramentas. Atualmente, apenas o formato FSP, utilizado pela ferramenta LTSA [Magee and Kramer 2006], é suportado como formato de entrada. Contudo, LoTuS importa apenas modelos já compilados pela ferramenta LTSA.

Vale salientar que, através da criação de *plugins*, o usuário pode criar seu próprio mecanismo de importação, transformação e geração de modelos para LoTuS.

4.6. Exportação

LoTuS permite que os modelos gerados na ferramenta sejam exportados para outros formatos. No caso de um modelo não probabilístico, é disponibilizada a exportação para FSP, enquanto é possível exportar para o PRISM [Kwiatkowska 2007] no caso do modelo ser probabilístico.

4.7. Análise e Verificação de Modelos

Além dos recursos de modelagem, LoTuS fornece ao usuário algumas técnicas de análise e verificação: detecção de *deadlock*, detecção de estados (probabilisticamente) inalcançáveis e verificações de propriedades de alcançabilidade (*reachability properties*).

O *deadlock* em um modelo é caracterizado por um estado qualquer sem transições de saídas. A detecção de *deadlock* é particularmente útil na composição paralela de modelos, uma vez que a execução conjunta pode causar comportamentos inesperados. Para representar a correta terminação da execução de um sistema, o usuário pode mudar o tipo do estado para final, pois caminhos que terminam em um estado final não são considerados caminhos de *deadlock*.

A verificação da existência de estados inalcançáveis é importante porque o usuário pode cometer erros quando cria um modelo graficamente. Um estado inalcançável significa que o comportamento iniciado por ele nunca irá ocorrer, ou seja, é um comportamento impossível. LoTuS informa ao usuário a existência de estados inalcançáveis e pergunta a ele se os estados devem ser removidos. Se o usuário concordar, os estados e suas transições serão removidos do modelo.

Um situação similar pode ocorrer na análise de um modelo probabilístico. Embora um estado possa ser alcançável em um caminho regular no modelo, ele pode ser probabilisticamente inalcançável, isto é, há pelo menos um caminho que termina naquele estado

[3]http://astah.net

[4]http://www.omg.org/spec/XMI/

que possui probabilidade 0. Isso indica que mesmo que o comportamento representado por aquele rastro possa eventualmente acontecer, ele nunca é exercitado de fato. Esse cenário pode ser identificado, por exemplo, quando o modelo é anotado com perfis operacionais [Musa 1993], mostrando que determinadas funções podem nunca ser acionadas pelo usuário. De modo semelhante à detecção não probabilística, LoTuS irá solicitar ao usuário qual ação tomar com os estados inalcançáveis, podendo removê-los ou não do modelo.

Com respeito à verificação de modelos, LoTuS trabalha com requisitos de sistemas que podem ser descritos como propriedades de confiabilidade. O mais importante caso de uma propriedade desse tipo é a propriedade de alcançabilidade [Filieri et al. 2011], que diz que, a partir de um estado inicial, há um outro estado alcançável onde a propriedade é satisfeita. Esse estado pode ser um estado de falha ou de sucesso que indicam, respectivamente, um terminação desejada ou indesejada da execução do sistema modelado.

Cadeias de Markov de tempo discreto (DTMC) tradicionalmente são usadas para análise probabilística de alcançabilidade [Cheung et al. 2008][Filieri et al. 2011]. Como mencionado na Seção 2, um DTMC pode ser obtido removendo-se as ações de um PLTS. Portanto, LoTuS oferece verificação probabilística de propriedades de alcançabilidade em PLTSs. O usuário pode representar a propriedade a ser verificada de duas formas: (i) indicando o estado inicial e o estado alvo, ou (ii) indicando a ação inicial e a ação final. A ferramenta calcula a probabilidade de alcançar o estado alvo do estado inicial transformando o PLTS em uma matriz de probabilidade e realizando multiplicações de matrizes [Kwiatkowska 2007].

O usuário também pode especificar o valor desejado e o operador de comparação (menor ou maior que, igual ou diferente). Nesse caso, além do valor calculado da propriedade de alcançabilidade, LoTuS também informa se a propriedade é satisfeita ou não. Outra possibilidade é informar que estados ou ações não devem ser consideradas na propriedade. Vale salientar que a verificação probabilística não funciona caso o modelo contenha um estado composto.

5. Avaliação

Nesta seção, descrevemos três tipos de avaliação da ferramenta proposta que foram realizadas: usabilidade, desempenho, e um estudo de caso que exercita algumas de suas principais funcionalidades.

5.1. Usabilidade

Uma análise de usabilidade de LoTuS é apresentada em [Cavalcante 2015]. O experimento realizado foi baseado em um estudo de observação, onde o experimentador observa os participantes enquanto estes realizam atividades específicas com o objetivo de coletar informações sobre como estas atividades são desempenhadas. Foram selecionados para a realização do teste de usabilidade da ferramenta 10 alunos de graduação do curso de bacharelado em Ciências da Computação da Universidade Estadual do Ceará (UECE), todos entre o sexto e o nono semestre do curso. A instrumentação do experimento incluiu os seguintes elementos: um formulário de pré-experiência, caracterizando os participantes, a fim de adquirir um conhecimento prévio sobre eles; a ficha de descrição do teste, no qual são descritas as tarefas que os participantes devem realizar; a ferramenta LoTuS; um questionário de pós-experiência, visando à avaliação da usabilidade e interface da ferramenta;

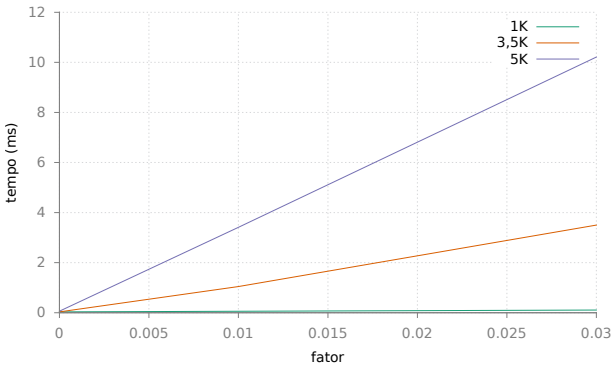

Figura 6. Experimentos de avaliação da desempenho

e um questionário de observação do usuário, onde o observador vai indicar comentários acerca do uso da ferramenta pelo participante do teste.

As seguintes variáveis foram consideradas no experimento: a quantidade de tarefas realizadas corretamente, tarefas não realizadas e tarefas incompletas; o grau de satisfação do participante em relação a interface da ferramenta; a quantidade de vezes que o grupo responsável pelo experimento foi acionado para tirar dúvidas; o tempo de execução de cada tarefa; o grau de satisfação geral da ferramenta. Foram analisadas 14 tarefas que incluem o uso das seguintes funcionalidades: importar diagramas de sequência, desenhar modelos, fazer uma composição paralela, simular o modelo, executar o modelo utilizando valores de guarda, gerar *traces*, e gerar um modelo através de um arquivo de traces.

Como resultado, a autora destaca que 90% dos participantes acharam a interface da ferramenta bastante agradável e fizeram elogios acerca desta e de sua aparência. Ao perguntar se a ferramenta atingiu seu objetivo para a qual foi desenvolvida, 100% dos participantes concordaram, afirmando que a ferramenta é muito boa e que os usuários poderão utilizá-la para grandes proveitos. Por outro lado, foi possível identificar os principais problemas relacionados à usabilidade da ferramenta, os quais foram solucionados pela equipe de desenvolvimento e já incoporados na versão atual proposta neste trabalho.

5.2. Desempenho

Para avaliar a capacidade e os limites do LoTuS ao trabalhar com modelos que possuem uma grande quantidade de estados e transições, foram realizados testes de performance nos quais modelos gerados aleatoriamente em XML são carregados visualmente e descarregados pela ferramenta, simulando as operações de abrir e salvar realizadas pelos usuários. Os componentes envolvidos nessas operações são utilizados para automatizar a avaliação que consiste em medir o tempo médio necessário para abrir e salvar os modelos.

O gráfico da Figura 6 (obtidos em uma máquina com processador Intel Core i5, 6GB de memória RAM, rodando a distribuição GNU/Linux Debian 8 Wheezy e Oracle JDK 8u43) mostra o tempo médio para as operações de abrir e salvar para modelos com 1000 (1K), 3500 (3,5K) e 5000 (5K) estados. O eixo das abcissas representa a quantidade de transições para cada estado do modelo medido em percentual do total de estados, onde 0 representa um modelo sem transições, apenas com estados, e 1 representa um grafo completo, com todos os estados possuindo transições para todos os outros estados. O

eixo das ordenadas representa o tempo médio para realizar a operação em milisegundos (ms).

O gráfico mostra que modelos com até 1000 estados são carregados e salvos muito rapidamente, com tempo de carga bem menor do que 1ms. Para os outros tamanhos, nota-se que o tempo de carga cresce linearmente à quantidade de transições presentes. Para modelos com até 3500 estados, no cenário onde cada estado tem transições para 3% dos outros estados (ou seja, um estado conectado a outros 105 estados), o tempo médio de carga foi menor que 4ms. Já no modelo de 5000 estados, esse mesmo cenário (porém agora cada estado está conectado a outros 150 estados) foi carregado em pouco mais de 10ms.

Como conclusão, podemos ver que mesmo para modelos relativamente grandes, LoTuS apresenta um tempo médio de carga baixo. Contudo, esse tempo pode ser bastante diferente caso outras operações sejam realizadas, como a verificação probabilística do modelo. Pretende-se realizar esse teste de desempenho em um trabalho futuro.

5.3. Estudo de caso

A terceira forma de avaliação consistiu em um estudo de caso onde LoTuS foi utilizado para modelar e analisar uma aplicação chamada TeleAssistance (TA), um sistema distribuído para assistência remota de pacientes [Epifani et al. 2009] [Weyns and Calinescu 2015]. O TeleAssistance oferece três possibilidades ao usuário: enviar os sinais vitais do paciente, enviar um alarme de pânico e parar a aplicação. Os dados da primeira mensagem são encaminhados para o serviço de um laboratório para serem analisados. O laboratório responde enviando uma das seguintes mensagens: mudar medicamento, mudar a dose, ou enviar um alarme. Esta útima envia um chamado a uma equipe de pronto socorro (*First Aid Squad* - FAS) cuja função é assistir o paciente em domicílio no caso de emergência.

Quando um paciente aperta o botão de pânico, a aplicação também envia um alarme para o pronto socorro. Finalmente, o paciente pode decidir por parar o TA. O sistema pode falhar nas seguintes situações: ao enviar um alarme para o pronto socorro, ao receber a análise dos dados enviados pelo laboratório, ou ao enviar uma notificação de mudança de dose ou de medicamento para o paciente. Em todos esses casos, o sistema vai para um estado de erro indicando que uma falha aconteceu.

5.3.1. Verificação de propriedades de alcançabilidade

Estamos interessados em verificar as seguintes propriedades: (P1) a probabilidade de sucesso ao usar o dispositivo (nenhum falha ocorrer) é maior ou igual a 0.7 e (P2) a probabilidade de falha após um envio de alarme para o pronto socorro é menor que 0.006. Ambas propriedades podem ser vistas como sendo de alcançabilidade, onde a primeira visa atingir um estado de sucesso (estado 2), enquanto a outra consiste em alcançar um estado de falha (estado 5).

Diferentemente do exemplo mostrado em [Epifani et al. 2009], onde o modelo é originalmente probabilístico, nós assumimos que as probabilidades de ocorrência de cada transição não eram sabidas em tempo de projeto. Então, foi criado um outro modelo do

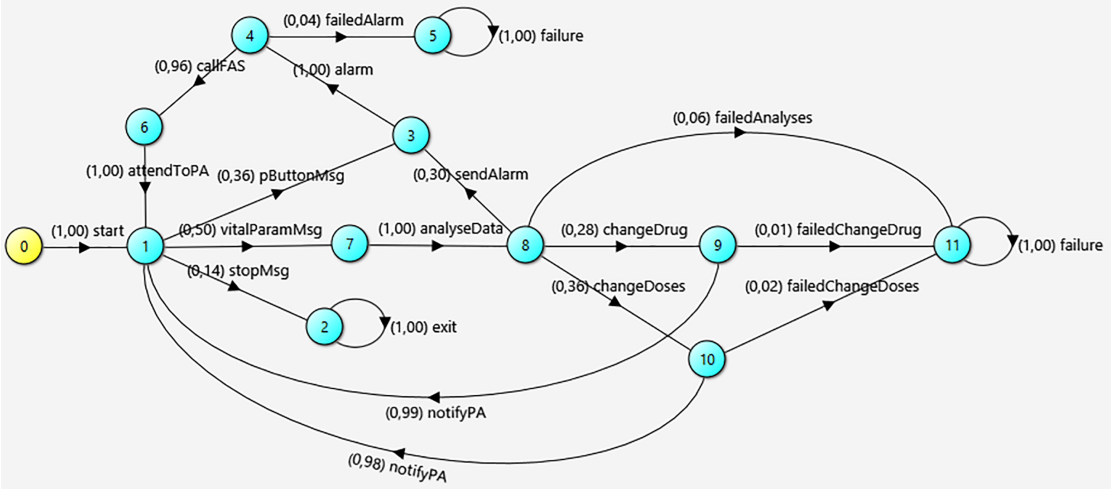

Figura 7. Modelo probabilístico do TA

TA, equivalente ao primeiro em bisimilaridade[5] [Milner 1989], porém com mais estados e transições, com o objetivo de simular o comportamento "real" do usuário e as probabilidades de falhas do serviço. Para simular o uso do sistema, utilizamos o plugin *Trace Generator*, o qual gerou 5000 rastros de execução.

Em seguida, os *traces* foram utilizados como entrada para a anotação de probabilidades de execução no modelo original utilizando o plugin *Probability Annotator*, que gerou como saída o PLTS mostrado na Figura 7. Com isso, é possível calcular as probabilidades das propriedades P1 e P2 utilizando a verificação probabilística de LoTuS. Uma vez que o valor obtido para a propriedade P1 foi 0.71519, podemos dizer que a propriedade é satisfeita. Por outro lado, P2 não é satisfeita, pois seu resultado retornou o valor 0.10237. A Figura 8 mostra a tela de verificação de propriedades do LoTuS para o modelo TA e as propriedades P1 e P2.

Para verificar se os valores calculados pelo LoTuS para P1 e P2 estão corretos, o modelo probabilístico anotado do TA foi exportado para a ferramenta de checagem de modelos PRISM [Kwiatkowska 2007] usando a funcionalidade de exportação do LoTuS. As propriedades foram especificadas em PCTL e verificadas nessa ferramenta. Os resultados foram bastante próximos aos obtidos pela verificação probabilística do LoTuS, uma vez que a diferença entre os resultados foi de apenas 0.0006 para ambas propriedades. Essa diferença pode ser devido à aproximações realizadas durante o procedimento de multiplicação de matrizes.

Em uma segunda análise, consideramos o cenário onde não há um modelo inicial do sistema (ou não se tem acesso a ele), mas deseja-se gerar esse modelo através de rastros de execução da aplicação. Para tanto, a o código fonte da implementação existente do TA[6] foi anotado e executado 1000 vezes (gerando um rastro para cada execução), com o intuito de exercitar todos os comportamentos posíveis.

Os rastros foram então organizados em um arquivo CSV, que foi usado como

[5]Dois LTSs são considerados bisimilares se, quando um realiza uma ação a partir de um estado s, o outro realiza a mesma ação a partir de um estado q equivalente a s

[6]Disponível em https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/tas/

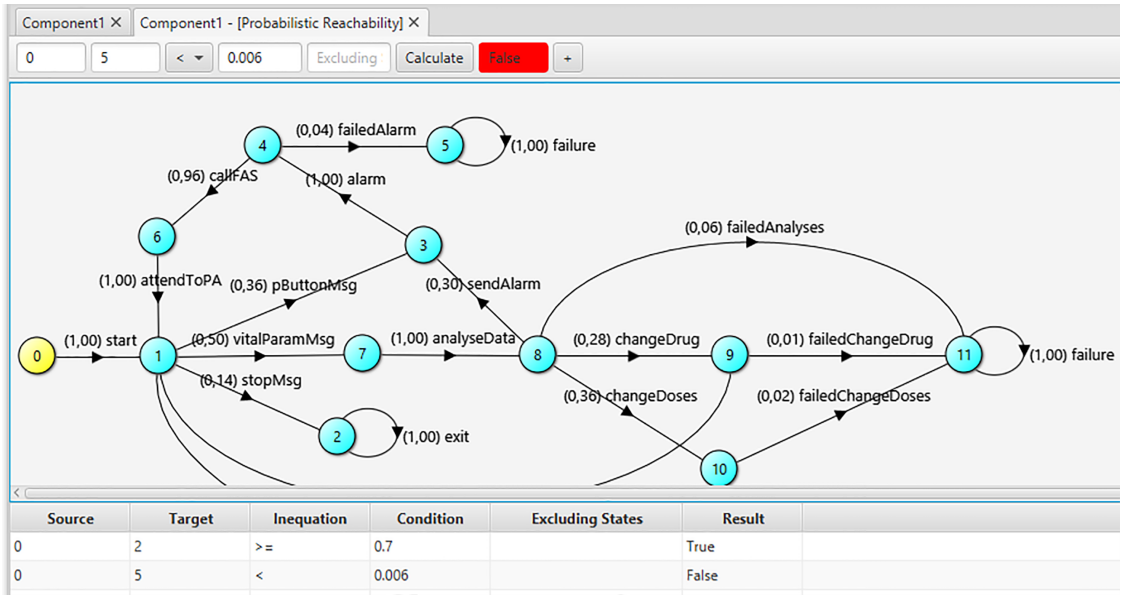

Figura 8. Tela de verificação do LoTuS referente ao modelo TA

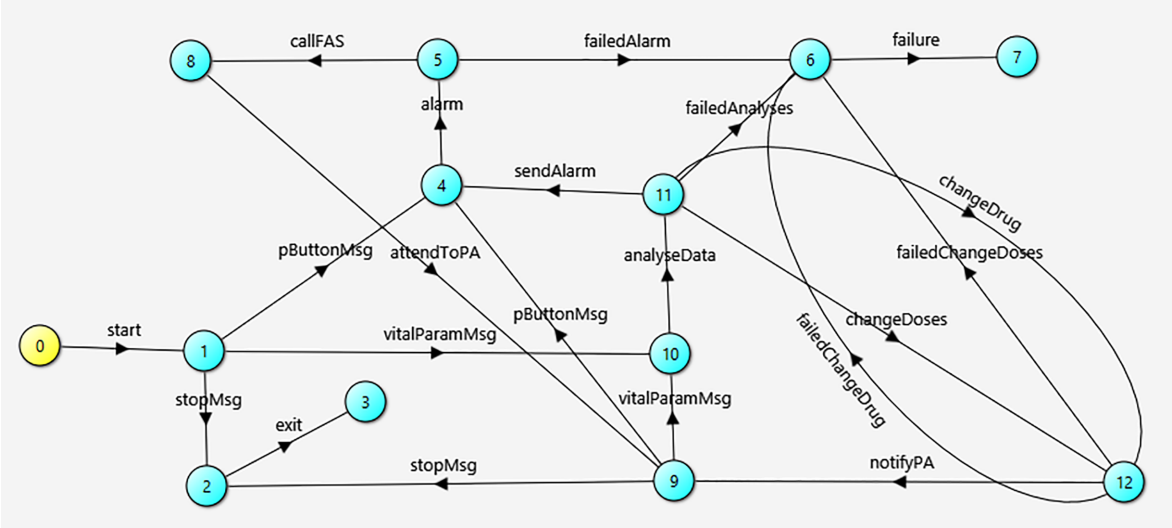

Figura 9. LTS do TA a partir de rastros

entrada para a funcionalidade do LoTuS de geração de modelos a partir de rastros, gerando como saída um possível modelo de comportamento do sistema[7], conforme ilustrado na Figura 9. Comparando o modelo gerado com o modelo original do sistema (o mesmo da Figura 7, porém sem probabilidades), nota-se que os modelos, apesar de diferentes, representam o mesmo comportamento do sistema. Com isso, podemos ver que a implementação do sistema está de acordo com o comportamento modelado originalmente.

6. Trabalhos Relacionados e Discussão

Esta seção compara e discute alguns trabalhos relacionados aos principais benefícios de LoTuS: a modelagem gráfica de modelos tradicionais e probabilísticos, a importação a

[7]A disposição dos estados foi alterada para facilitar a visualização do modelo

partir de outros formatos, as formas de análise e a possibilidade de extensão.

Sobre modelagem de comportamento visual, a abordagem mais comum é o uso de modelos UML dinâmicos, como diagramas de sequência e de estados. Para estes, há um grande conjunto de ferramentas, sejam acadêmicas ou profissionais, gratuitas ou pagas, tais como Astha, Rational Rose[8], e Visual Paradigm[9]. O maior problema de modelos UML é a falta de uma semântica formal que permite a verificação de modelos. Nesta direção, alguns trabalhos foram propostos [Ng and Butler 2002] [Bolton and Davies 2000] para integrar partes de CSP [Hoare 1985], uma álgebra de processo para especificação de comportamento de componentes que interagem, e UML, onde CSP serve como semântica formal para a notação gráfica não-formal, e a UML fornece uma interface amigável para CSP. Uma abordagem similar modela o sistema como Árvores de Comportamento (Behaviour Trees [Dromey 2003]), uma notação não-formal gráfica que permite ao usuário primeiro modelar os requisitos individualmente que são, em seguida, integrados no modelo de projeto do sistema, e transforma-os em processos CSP para permitir análise de modelos [Winter 2004][Colvin and Hayes 2011].

É importante mencionar que nem sempre uma notação gráfica é mais adequada para especificar o comportamento de um sistema do que uma linguagem formal, como por exemplo, na modelagem de sistemas complexos que demandam milhares de estados. Por outro lado, ela pode facilitar o entendimento e aprendizado de modelagem de software através de visualização instantânea do modelo, isto é, o comportamento desejado pode ser visto em tempo de projeto, e não apenas após a compilação do modelo especificado em uma linguagem formal. Além disso, mesmo sendo uma notação gráfica, a modelagem usando LTSs pode consumir muito tempo e introduzir erros, particularmente para modelos de porte médio ou grande. Por isso, a transformação de modelos de comportamento UML, como os diagramas de sequência, atividades e estados, em modelos LTSs pode aumentar a utilização da abordagem gráfica. Isto já vem sendo feito na literatura para diferentes propósitos, como teste [Cartaxo et al. 2008], análise de performance [Bennett et al. 2004], e confiabilidade [Rodrigues et al. 2005].

Outro formalismo bastante utilizado para modelar o comportamento de software são as Redes de Petri [Peterson 1981], para as quais há uma vasta lista de ferramentas disponíveis[10]. Embora seja possível realizar várias formas de análise utilizando Redes de Petri tradicionais ou estocásticas, como simulação [Dingle et al. 2009], detecção de *deadlock* [Dingle et al. 2009], verificação de propriedades de alcançabilidade [Dingle et al. 2009], geração de modelos a partir de código [Dedova and Petrucci 2013] e de diagramas UML [López-Grao et al. 2004], e geração de casos de teste [Xu 2011], similarmente ao que LoTuS também faz, nossa escolha por usar LTS e PLTS se deu principalmente pela experiência dos autores em lidar com esses formalismos e ferramentas relacionadas. Pretendemos implementar em LoTuS outras formas de análise já presentes em trabalhos que usam Redes de Petri, como por exemplo a diferenciação de modelos [Dingle et al. 2009]. Por outro lado, o uso de LTS facilitou a implementação de algumas funcionalidades, como a modelagem de estados

[8]http://www-03.ibm.com/software/products/en/ratirosefami

[9]http://www.visual-paradigm.com/

[10]Petri Nets Tool Database: http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html

compostos. Portanto, acreditamos que LoTuS pode tanto se beneficiar de técnicas de modelagem e análise já existentes para Redes de Petri, quanto servir de motivação para a extensão de ferramentas que usam esse formalismo.

Ferramentas de modelagem gráfica de comportamento que suportem ambos modelos tradicionais e probabilisticos não são fáceis de encontrar. Por exemplo, MaTeLo (Markov Test Logic) [Feliachi and Le Guen 2010] fornece modelagem gráfica de Cadeias de Markov e máquinas de estados finitos para realizar teste baseados em modelos. PIPE2 [Dingle et al. 2009] é uma ferramenta Java para a construção e análise de modelos Generalised Stochastic Petri Nets. Outras ferramentas fornecem linguagens formais que suportam a especificação de ambos os tipos de modelos. Por exemplo, LTSA-PCA [Rodrigues et al. 2014] é uma extensão da ferramenta LTSA que suporta a especificação, visualização e análise de falhas de sistemas baseados em componentes modelados como Probabilistic Component Automata (PCA).

Sobre extensibilidade, Mobius[Courtney et al. 2009] é uma ferramenta para modelagem de desempenho e dependabilidade cuja arquitetura foi projetada para permitir a integração de novos formalismos e técnicas de maneira fácil. Similar ao LoTuS, PIPE2 também é extensível via plugins e suporta interoperabilidade com outras ferramentas de modelagem de Redes de Petri. LTSA também é uma ferramenta extensível, mas sua API não está disponibilizada para o desenvolvimento por terceiros.

A ferramenta mais próxima à nossa é UPPAAL [Behrmann et al. 2004], uma ferramenta para modelagem, simulação e verificação de sistemas de tempo real modelados como redes de autômatos temporais (*timed automata*) estendidos com relógios e variáveis de dados. UPAAL permite tanto a modelagem gráfica como disponilibiliza uma linguagem de especificação formal. Além disso, existem extensões de UPPAAL para realizar checagem de modelos estatísticos (UPPALL-SMC [David et al. 2015b]), explorar estratégias para *stochastic priced timed games* (UPPAAL Stratego [David et al. 2015a]), e algoritmos *on-the-fly* eficientes para análise de jogos baseados em *timed game automata* com respeito a propriedades de segurança (*safety*) e alcançabilidade (UPPAAL-TIGA [Cassez et al. 2005]).

Assim como LoTuS, UPPAAL também permite simulação e checagem de modelo probabilístico, porém LoTuS permite também a geração de modelos a partir da importação de diagramas de sequência ou de rastros de execução. Adicionalmente, o mecanismo de extensão de LoTuS é simples e está disponível no site da ferramenta, enquanto não há informações sobre como estender UPPAAL com novas funcionalidades. Por fim, LoTuS disponibiliza outras formas de manipulação e análise de modelos, como a criação de estados compostos, a anotação de probabilidades a partir de *traces*, e algoritmos de testes baseados em modelos.

7. Conclusão

Este artigo apresentou LoTuS, uma ferramenta de modelagem gráfica, análise e verificação de comportamento de software utilizando LTS e PLTS. Por ser estensível e de código aberto, LoTuS permite que a comunidade contribua para a evolução da ferramenta em termos de melhoria interna de sua estrutura de código e adicione novas funcionalidades através da criação de *plugins*. A arquitetura e principais funcionalidades do LoTuS foram apresentadas, bem como foi uma avaliação da ferramenta, que levou em

conta usabilidade, desempenho e utilização dos seus principais recursos em um estudo de caso.

Apesar da modelagem gráfica ser intuitiva e rápida, ela traz como desvantagem a dificuldade de modelar sistemas complexos, com centenas ou milhares de estados. Para estes casos, LoTuS pode ser utilizada em conjunto com outras ferramentas que usem, por exemplo, uma linguagem de especificação formal. Além disso, a verificação probabilística de LoTuS pode ser considerada simples se comparada com outras ferramentas que usam uma abordagem simbólica (*symbolic model checking*), na qual o espaço de estados do modelo é representado utilizando estruturas de dados especiais, como o *binary decisions diagrams*, permitindo uma manipulação mais eficiente.

Atualmente, está sendo implementado um plugin para modelagem de Message Sequence Charts que permitirá que o comportamento dinâmico do sistema possa ser modelado usando aquele formalismo e, em seguida, seja transformado para LTS, conforme proposto em [Uchitel et al. 2003]. Isso permitirá novas formas de análise, como a detecção de cenários implícitos [Uchitel et al. 2001]. Como trabalhos futuros, pretendemos disponibilizar outros *plugins* de transformação de modelos e de importação de e exportação para outros formatos, aumentando assim a interoperabilidade de LoTuS. Adicionalmente, planejamos permitir a especificação de outros tipos de propriedades, como *safety* e *liveness*, através do uso de uma linguagem de lógica temporal. Também pretendemos realizar uma análise de desepenho mais aprofundada, levando em conta outras funcionalidades da ferramenta, como simulação, execução e composição paralela. Por fim, pretendemos realizar outros estudos de caso com sistemas reais.

Referências

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. In Bernardo, M. and Corradini, F., editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg.

Bennett, A., Field, A., and Woodside, C. (2004). Experimental evaluation of the uml profile for schedulability, performance, and time. In Baar, T., Strohmeier, A., Moreira, A., and Mellor, S., editors, *UML 2004 The Unified Modeling Language. Modeling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 143–157. Springer Berlin Heidelberg.

Blom, S., van de Pol, J., and Weber, M. (2010). Ltsmin: Distributed and symbolic reachability. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 354–359. Springer-Verlag.

Bolton, C. and Davies, J. (2000). Activity graphs and processes. In Grieskamp, W., Santen, T., and Stoddart, B., editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 77–96. Springer Berlin Heidelberg.

Broadfoot, G. (2005). Asd case notes: Costs and benefits of applying formal methods to industrial control software. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 548–551. Springer Berlin Heidelberg.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

Cartaxo, E. G., Andrade, W. L., Neto, F. G. O., and Machado, P. D. L. (2008). Lts-bt: A tool to generate and select functional test cases for embedded systems. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 1540–1544. ACM.

Cassez, F., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2005). Efficient on-the-fly algorithms for the analysis of timed games. In Abadi, M. and de Alfaro, L., editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin Heidelberg.

Cavalcante, N. A. (2015). Análise da usabilidade da ferramenta de modelagem lotus. Monografia de graduação, Universidade Estadual do Ceará (UECE), Fortaleza, Brasil.

Cheung, L., Roshandel, R., Medvidovic, N., and Golubchik, L. (2008). Early prediction of software component reliability. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 111–120, Leipzig, Germany. ACM.

Colvin, R. J. and Hayes, I. J. (2011). A semantics for behavior trees using csp with specification commands. *Sci. Comput. Program.*, 76(10):891–914.

Courtney, T., Gaonkar, S., Keefe, K., Rozier, E., and Sanders, W. (2009). Mobius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 353–358.

David, A., Jensen, P. G., Larsen, K. G., Mikučionis, M., and Taankvist, J. H. (2015a). Uppaal stratego. In Baier, C. and Tinelli, C., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer Berlin Heidelberg.

David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Poulsen, D. B. (2015b). Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415.

Dedova, A. and Petrucci, L. (2013). From code to coloured petri nets: Modelling guidelines. In *Transactions on Petri Nets and Other Models of Concurrency VIII*, volume 8100 of *Lecture Notes in Computer Science*, pages 71–88. Springer Berlin Heidelberg.

Dingle, N. J., Knottenbelt, W. J., and Suto, T. (2009). Pipe2: A tool for the performance evaluation of generalised stochastic petri nets. *SIGMETRICS Perform. Eval. Rev.*, 36(4):34–39.

Dromey, R. (2003). From requirements to design: formalizing the key steps. In *Software Engineering and Formal Methods, 2003.Proceedings. First International Conference on*, pages 2–11.

Epifani, I., Ghezzi, C., Mirandola, R., and Tamburrelli, G. (2009). Model evolution by run-time adaptation. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 111–121, Vancouver, Canada. ACM.

Feliachi, A. and Le Guen, H. (2010). Generating transition probabilities for automatic model-based test generation. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 99–102.

Filieri, A., Ghezzi, C., and Tamburrelli, G. (2011). Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 341–350. ACM.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Geer, P. A. (2011). *Formal Methods in Practice: Analysis and Application of Formal Modeling to Information Systems*. PhD thesis, State University of New York Institute of Technology at Utica/Rome.

Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J. P., and Margaria, T. (2008). Software engineering and formal methods. *Commun. ACM*, 51(9):54–59.

Hoare, C. A. R. (1985). *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Holzmann, G. J. (1997). The model checker spin. *IEEE Transaction on Software Engineering*, 23(5):279–295.

Keller, R. M. (1976). Formal verification of parallel programs. *Commun. ACM*, 19:371–384.

Kwiatkowska, M. (2007). Quantitative verification: models, techniques and tools. In *ESEC/FSE'07: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 449–458, Dubrovnik, Croatia. ACM.

López-Grao, J. P., Merseguer, J., and Campos, J. (2004). From uml activity diagrams to stochastic petri nets: Application to software performance engineering. *SIGSOFT Softw. Eng. Notes*, 29(1):25–36.

Magee, J. and Kramer, J. (2006). *Concurrency: State Models and Java Programs*. Wiley.

Milner, R. (1989). *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Muniz, L. L., Netto, U. S. C., and Maia, P. H. M. (2015). TCG - A model-based testing tool for functional and statistical testing. In *ICEIS 2015 - Proceedings of the 17th International Conference on Enterprise Information Systems, Volume 2, Barcelona, Spain, 27-30 April, 2015*, pages 404–411.

Musa, J. D. (1993). Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32.

Ng, M. Y. and Butler, M. J. (2002). Tool support for visualizing csp in uml. In *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '02, pages 287–298. Springer-Verlag.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Rodrigues, G., Rosemblum, D., and Uchitel, S. (2005). Using scenarios to predict the reliability of concurrent component-based software systems. In *FASE'05: 8th International Conference on Fundamental Approaches to Software Engineering*, pages 111–126, Edinburgh, Scotland.

Rodrigues, P., Lupu, E., and Kramer, J. (2014). Ltsa-pca: Tool support for compositional reliability analysis. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 548–551. ACM.

Uchitel, S., Chatley, R., Kramer, J., and Magee, J. (2003). Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 597–601, Berlin, Heidelberg. Springer-Verlag.

Uchitel, S., Kramer, J., and Magee, J. (2001). Detecting implied scenarios in message sequence chart specifications. In *ESEC/FSE '09: Proceedings of the Joint Meeting of the 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 74–82, New York, NY, USA. ACM.

Weyns, D. and Calinescu, R. (2015). Tele assistance: A self-adaptive service-based system examplar. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 88–92. IEEE Press.

Winter, K. (2004). Formalising behaviour trees with csp. In Boiten, E., Derrick, J., and Smith, G., editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 148–167. Springer Berlin Heidelberg.

Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36.

Xu, D. (2011). A tool for automated test code generation from high-level petri nets. In *Proceedings of the 32Nd International Conference on Applications and Theory of Petri Nets*, PETRI NETS'11, pages 308–317. Springer-Verlag.

Prezados editores,

Segue abaixo relatório destacando as modificações realizadas na versão final do artigo.

Att.

Revisão: "Justificar o diferencial da abordagem com relação a Redes de Petri. Isso foi mencionado, conforme o revisor solicitou, mas não fica clara a comparação e o motivo da escolha pelas abordagens utilizadas. Não há problema se o artigo exceder em uma página para fazer a inclusão."

R: Foi introduzido o seguinte parágrafo na seção 6 (Discussão): "Outro formalismo bastante utilizado para modelar o comportamento de software são as Redes de Petri [Peterson 1981], para as quais há uma vasta lista de ferramentas disponíveis10. Embora seja possível realizar várias formas de análise utilizando Redes de Petri tradicionais ou estocásticas, como simulação [Dingle et al. 2009], detecção de deadlock [Dingle et al. 2009], verificação de propriedades de alcançabilidade [Dingle et al. 2009], geração de modelos a partir de código [Dedova and Petrucci 2013] e de diagramas UML [López-Grao et al. 2004], e geração de casos de teste [Xu 2011], similarmente ao que LoTuS também faz, nossa escolha por usar LTS e PLTS se deu principalmente pela experiência dos autores em lidar com esses formalismos e ferramentas relacionadas. Pretendemos implementar em LoTuS outras formas de análise já presentes em trabalhos que usam Redes de Petri, como por exemplo a diferenciação de modelos [Dingle et al. 2009]. Por outro lado, o uso de LTS facilitou a implementação de algumas funcionalidades, como a modelagem de estados compostos. Portanto, acreditamos que LoTuS pode tanto se beneficiar de técnicas de modelagem e análise já existentes para Redes de Petri, quanto servir de motivação para a extensão de ferramentas que usam esse formalismo."

Com isso, também foram acrescentadas as seguintes referências:

- Dedova, A. and Petrucci, L. (2013). From code to coloured petri nets: Modelling guidelines. In Transactions on Petri Nets and Other Models of Concurrency VIII, volume 8100 of Lecture Notes in Computer Science, pages 71–88. Springer Berlin Heidelberg.
- López-Grao, J. P., Merseguer, J., and Campos, J. (2004). From uml activity diagrams to stochastic petri nets: Application to software performance engineering. SIGSOFT Softw. Eng. Notes, 29(1):25–36.
- Xu, D. (2011). A tool for automated test code generation from high-level petri nets. In Proceedings of the 32Nd International Conference on Applications and Theory of Petri Nets, PETRI NETS'11, pages 308–317. Springer-Verlag.