Lotus@Runtime: Uma Ferramenta para Monitoramento e Verificação em Tempo de Execução para Sistemas Autoadaptativos

Davi Monteiro Barbosa[1], **Paulo Henrique Mendes Maia**[1], **Evilásio Costa Júnior**[1]

[1]Universidade Estadual do Ceará (UECE)
Centro de Ciências e Tecnologia - CCT
Av. Dr. Silas Munguba, 1700, Campus do Itaperi – Fortaleza – CE

davi.monteiro@aluno.uece.br, pauloh.maia@uece.br, junior.facanha@gmail.com

Abstract. *Self-adaptive systems have the ability to adapt their behavior during its execution in response to changes in the environment where it operates. To ensure the success of an adaptation, traditional approaches such as testing and software verification are insufficient. Therefore, it should adopt monitoring and runtime verification approaches to ensure that adaptations have the expected result. In this paper, we propose the Lotus@Runtime to perform, in an integrated manner, monitoring, verification, and violations notification found during execution of a self adaptive system.*

Resumo. *Sistemas autoadaptativos possuem a capacidade de adaptar seu comportamento durante sua execução em resposta às mudanças no ambiente onde está inserido. Para garantir o sucesso de uma adaptação, abordagens tradicionais como testes e verificações de software são insuficientes. Por isso, deve-se adotar abordagens de monitoramento e verificação em tempo de execução para assegurar que as adaptações tenham o resultado esperado. Neste trabalho, é proposta uma ferramenta para realizar, de forma integrada, o monitoramento, a verificação e a notificação de violações encontradas durante a execução de um sistema autoadaptativo.*

1. Introdução

O crescimento da complexidade de sistemas, junto com o custo para mantê-los após o seu desenvolvimento, foram fatores apontados por Kephart e Chess (2003) que serviram de motivação para a proposta de sistemas autoadaptativos. Para Huebscher e McCann (2008), características como autoconfiguração, auto-otimização, autocorreção e autoproteção fazem parte da essência dos sistemas autoadaptativos, fazendo com que essa nova classe de sistemas tenha autonomia suficiente para operar com o mínimo de intervenção humana.

Um sistema autoadaptativo possui a capacidade, em tempo de execução, de modificar o seu comportamento ou sua estrutura através de adaptações em resposta a mudanças no ambiente, que podem representar uma simples reconfiguração ou até uma modificação no modelo do sistema, resultando em uma alteração de sua arquitetura [Chen et al. 2014]. A adaptação pode acontecer para que uma funcionalidade do sistema não deixe de ser executada (por exemplo, a troca de um serviço por outro similar caso o primeiro tenha

ficado indisponível) ou para garantir que requisitos não funcionais sejam atendidos, como níveis de confiabilidade, segurança ou disponibilidade, dentre outros.

Uma técnica bastante utilizada no desenvolvimento de sistemas autoadaptativos é a construção de modelos arquiteturais ou de comportamento em tempo de projeto para que sejam atualizados durante a execução do sistema. Tais modelos são conhecidos como *models@runtime* [Blair et al. 2009]. Esse cenário acontece, por exemplo, em sistemas que devem garantir requisitos expressos em termos de propriedades e que são altamente influenciados pela forma como o ambiente se comporta (por exemplo, levando em conta o perfil de comportamento do usuário). Com isso, é possível verificar propriedades utilizando uma técnica quantitativa, como a checagem de modelos (*model checking*), para prever e identificar violações dessas propriedades, bem como planejar os passos da adaptação para prevenir ou recuperar o sistema das violações [Calinescu et al. 2012].

Para endereçar esses desafios, trabalhos como Goldsby *et al.* (2008), Arcaini *et al.* (2012) e Calinescu *et al.* (2013) apresentam propostas para realizar o monitoramento e a verificação em sistemas autoadaptativos. Porém, essas soluções geralmente são específicas para uma categoria de sistemas, como os baseados em serviço, o que torna difícil sua reutilização para outros domínios. Além disso, a maioria faz uso de uma linguagem formal para especificar o comportamento do sistema e para definir as propriedades de interesse, o que também dificulta sua aplicação por usuários que não têm familiaridade com métodos formais. Por fim, esses trabalhos não disponibilizam um ambiente integrado, onde o usuário possa, na mesma ferramenta, realizar a modelagem, o monitoramento e a verificação do sistema.

Este trabalho apresenta Lotus@Runtime, uma ferramenta para realizar, de forma integrada, o monitoramento, a verificação e a notificação de violações encontradas durante a execução de um sistema autoadaptativo. O Lotus@Runtime realiza o monitoramento dos rastros de execução (*traces*) gerados por um sistema autoadaptativo para anotar o modelo do sistema baseado em *Labelled Transition System* (LTS) probabilístico. Em seguida, as verificações em tempo de execução são realizadas a partir do modelo atualizado e um conjunto de propriedades de alcançabilidade. Caso uma propriedade seja violada, o sistema autoadaptativo será notificado para realizar uma adaptação. A ferramenta foi utilizada em um sistema autoadaptativo existente que não fazia o uso de modelos em tempo de execução. Por isso, foi necessário customizar o estudo de caso para que a aplicação pudesse ser adaptada de acordo com violações encontradas pelo Lotus@Runtime. Por fim, por ser de código-fonte aberto, ela permite que a comunidade possa contribuir com melhorias e ajustes.

O restante deste artigo encontra-se organizado da seguinte forma: a seção 2 apresenta o referencial teórico. Em seguida, a seção 3 descreve a ferramenta proposta, sua arquitetura e seu funcionamento. Posteriormente, a seção 4 mostra um estudo de caso que foi realizado na ferramenta Lotus@Runtime. A seção 5 apresenta os principais trabalhos relacionados. Por fim, a seção 6 apresenta as considerações finais.

2. Fundamentação Teórica

Sistemas de software autoadaptativos modificam seu próprio comportamento em resposta às mudanças de contexto [Oreizy et al. 1999]. Pelo contexto, entende-se o ambiente no qual o sistema está inserido, ou seja, quaisquer itens observáveis do sistema, tais como:

entradas de usuário, sensores, outras aplicações, entre outros. No âmbito de sistemas autoadaptativos, as adaptações têm papel fundamental no sucesso de tais aplicações. Por isso, adaptações devem ocorrer dentro de um ciclo de vida interminável durante a execução do sistema para que possam implantar as solicitações de mudanças requeridas, tanto pelo ambiente quanto pelos usuários.

O *loop* de controle MAPE-K (do inglês, *Monitor, Analyze, Plan, Execute over Knowledge base*) [Huebscher and McCann 2008, Cheng et al. 2009] tem sido utilizado como uma alternativa para viabilizar a autoadaptação em sistemas de software [Salehie and Tahvildari 2009]. Em resumo, na proposta apresentada em [Oreizy et al. 1999], um sistema de software adquire comportamento autoadaptativo através das adaptações providas pelo *loop* de controle. Os sensores são componentes de software ou hardware responsáveis por coletar informações de um sistema autoadaptativo. Os atuadores também são componentes de software ou hardware, que são responsáveis por aplicar as adaptações provenientes do loop de controle diretamente em um sistema autoadaptativo. Por fim, Salehie e Tahvildari (2009) apresentam uma taxonomia sobre duas abordagens de adaptação: interna e externa. Na primeira a lógica de adaptação está embutida no sistema de software. Na segunda, a lógica de adaptação está isolada do sistema de software.

Uma vez que uma adaptação modifica o comportamento de um sistema autoadaptativo, verificações realizadas durante a fase de desenvolvimento são insuficientes para garantir a conformidade no comportamento desses sistemas. Por isso, deve-se realizar verificações em tempo de execução com o objetivo de identificar violações em propriedades para que se possa planejar adaptações que previnam ou recuperem um sistema de tais violações [Calinescu et al. 2012]. Desse modo, faz-se necessário representar um sistema autoadaptativo em abstrações que possam acompanhar sua evolução no decorrer do tempo. Para tanto, modelos em tempo de execução são fortes candidatos para representar a evolução de tais sistemas [Bencomo et al. 2013]. Um modelo em tempo de execução pode ser definido como uma abstração de uma representação de um sistema, incluindo sua estrutura, comportamento e objetivos, que visa atender um propósito especifico durante a execução do sistema [Blair et al. 2009, Bencomo et al. 2013].

3. Lotus@Runtime

Lotus@Runtime foi desenvolvido com o objetivo de estender as funcionalidades do LoTuS[1], ferramenta para modelagem gráfica e análise de comportamento de software utilizando LTS. LoTuS fornece aos usuários um mecanismo de *drag and drop* para a criação dos modelos, o que torna a modelagem mais fácil e intuitiva. Além disso, a ferramenta também disponibiliza algumas técnicas de análise de modelo, como detecção de *deadlocks*, simulação e execução, além da verificação probabilísticas de propriedades de alcançabilidade (*reachability properties*), que são especificadas através do estado de origem e estado destino no modelo que se deseja alcançar.

Lotus@Runtime aproveita esses benefícios do LoTuS e fornece suporte para o monitoramento e verificação de software utilizando modelos probabilísticos em tempo de execução.

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

3.1. Arquitetura da ferramenta

O Lotus@Runtime foi projetado com uma arquitetura extensível baseada em componentes, possibilitando, dessa forma, um maior desacoplamento do seu código. A interface de programação disponibilizada pela ferramenta permite que componentes sejam instalados, removidos, ou recuperados por outro componente. A Figura 1 ilustra os principais componentes que fazem parte da arquitetura do Lotus@Runtime: `MonitorComponent`, `LotusModelComponent`, `ModelCheckerComponent`, `NotifierComponent` e `ConfigurationComponent`. A seguir, cada componente é melhor detalhado.

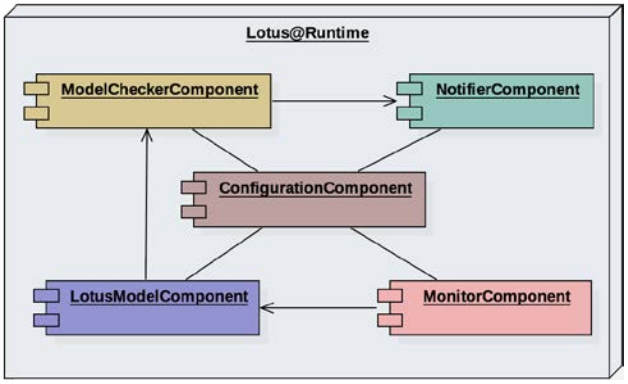

Figura 1. Arquitetura do Lotus@Runtime

3.1.1. MonitorComponent

Para realizar o monitoramento do sistema, optou-se por utilizar uma abordagem de monitoramento dos rastros de execução gerados pela aplicação [Maoz 2009]. Dessa forma, pode-se extrair informações relevantes que serão essenciais para manter o modelo em tempo de execução sempre atualizado em relação à evolução do sistema. Cada rastro é formado por uma sequência de ações visíveis que representam eventos responsáveis pela mudança de estado do sistema. Assim, um *log* é um conjunto finito de rastros que uma aplicação pode gerar durante sua execução.

O `MonitorComponent` é o componente responsável por monitorar os rastros que são gerados por um sistema autoadaptativo. Dessa forma, o monitoramento da aplicação é realizado através da leitura do arquivo de *log*, o qual deve estar no formato CSV. Cada linha no arquivo representa um rastro da aplicação e cada ação de mudança de estado é separada por uma vírgula em seus respectivos rastros. Existem duas estratégias para monitorar os rastros: na primeira, verifica-se a existência de um novo rastro de acordo com uma periodicidade definida pelo usuário (em milisegundos), enquanto na segunda verifica-se apenas a existência de um novo rastro caso ocorra uma alteração no arquivo de *log*.

3.1.2. LotusModelComponent

Para atender à necessidade de manter o modelo de um sistema autoadaptativo atualizado em relação às constantes adaptações que ocorrem ao longo do tempo, foi criado o compo-

nente `LotusModelComponent`, o qual é responsável por manter o modelo atualizado em relação às mudanças no comportamento da aplicação e oferecer serviços para recuperar o modelo atualizado para outros componentes do Lotus@Runtime.

O Lotus@Runtime usa o modelo do sistema representado como um LTS e criado de forma gráfica através da ferramenta LoTuS. A partir das informações extraídas pelo `MonitorComponent` dos rastros de execução da aplicação, o `LotusModelComponent` atualiza as probabilidades de ocorrência de cada ação do modelo e as anota em suas respectivas transições. Dessa forma, o modelo de comportamento do sistema passa a ser um LTS probabilístico, que pode ser visto como uma Cadeia de Markov de Tempo Discreto (DTMC) rotulada.

3.1.3. ModelCheckerComponent

Uma vez que o modelo de uma aplicação autoadaptativa esteja atualizado, pode-se então realizar verificações em tempo de execução com o objetivo de encontrar possíveis violações de alguma propriedade especificada pelo usuário. No Lotus@Runtime, uma propriedade é representada pela classe `Property`, composta por atributos que identificam um estado de origem e estado de destino de um modelo LTS, uma condição lógica que se deseja verificar e uma probabilidade de ocorrência.

Após definir um conjunto de propriedades que devem ser satisfeitas, o componente `ModelCheckerComponent` pode realizar verificações probabilísticas no modelo. Em uma verificação, o componente utiliza um algoritmo de alcance probabilístico para calcular a probabilidade do sistema ir de um estado inicial para um estado final, onde cada estado está definido em uma propriedade.

Em seguida, a probabilidade que foi calculada será comparada com a probabilidade que foi especificada na propriedade. Caso o resultado não satisfaça o valor por ela esperado, o que constitui uma violação, o `ModelCheckerComponent` deve invocar o serviço de notificação do `NotifierComponent`.

3.1.4. NotifierComponent

As possíveis violações de propriedades são publicadas pelo `NotifierComponent` em um *EventBus*, seguindo o padrão de arquitetura *publish subscriber* [Eugster et al. 2003]. Em seguida, para receber as violações que foram publicadas, deve-se implementar a interface `ViolationHandler` e sobrescrever o método `handler(property)`. Recomenda-se que a implementação concreta de um `ViolationHandler` faça parte da etapa de planejamento, pois, dessa forma, o planejador terá informações para projetar adaptações que possam evitar possíveis danos ao sistema.

O componente `ModelCheckerComponent` faz o uso do serviço disponibilizado pelo `NotifierComponent` toda vez que uma verificação em tempo de execução encontra uma violação no modelo de uma aplicação autoadaptativa.

3.1.5. ConfigurationComponent

O componente `ConfigurationComponent` foi desenvolvido com o objetivo de centralizar os parâmetros de configuração do Lotus@Runtime e fornecer, aos outros componentes, um mecanismo para recuperar tais parâmetros. Pode-se configurar o Lotus@Runtime através da interface de programação oferecida pelo `ConfigurationComponent` ou com o auxílio do *plugin Lotus@Runtime Configuration*, que está disponível para a ferramenta LoTuS e permite a configuração desses parâmetros através de uma interface gráfica. Por intermédio do *plugin* pode-se adicionar, remover, importar e exportar as configurações em formato JSON que serão lidas no momento da inicialização do Lotus@Runtime.

Os parâmetros de configuração do Lotus@Runtime são: (i) nome do arquivo de configuração; (ii) caminho do arquivo de *log* da aplicação autoadaptativa. (iii) caminho do arquivo do modelo LoTuS. (iv) tempo, em milissegundos, para verificar a existência de novos rastros de execução; (v) lista de propriedades que deseja-se verificar em tempo de execução.

3.2. Fluxo de uma adaptação

No Lotus@Runtime, o fluxo de uma adaptação, ilustrado na Figura 2, inicia-se pelo monitoramento dos rastro de execução gerados por um sistema autoadaptativo. Os rastros de execução são capturados durante a fase de monitoramento pelo *MonitorComponent*. Em seguida, durante o processo de atualização do modelo, o *LotusModelComponent* calcula as probabilidades de cada transição de estado e atualiza o modelo com as novas probabilidades em tempo de execução. Após a atualização do modelo, *ModelCheckerComponent* realiza as verificações em tempo de execução durante o processo de verificação. Caso uma propriedade seja violada, então *NotifierComponent* deve notificar a etapa de planejamento sobre a violação que ocorreu. Em seguida, independente de ocorrer ou não uma violação, o processo de monitoramento deve ser executado novamente de forma contínua. Para que as violações sejam recebidas, deve-se implementar a interface *ViolationHandler* na etapa de planejamento, como descrito na seção 3.1.4. Caso contrario, as notificações serão perdidas.

Como pode ser visto na Figura 2, em comparação com o ciclo MAPE-K, os componentes do Lotus@Runtime implementam apenas as fases de monitoramento e análise, fornecendo uma notificação em caso de violação de propriedades. As outras fases (planejamento e execução) devem ser implementadas pelo usuário fora da ferramenta.

4. Estudo de Caso

O estudo possui o objetivo de avaliar o uso da ferramenta proposta neste trabalho através da implementação das fases de monitoramento e análise utilizando o Lotus@Runtime em uma aplicação que não foi desenvolvida pelos autores deste artigo. A aplicação escolhida para realizar o estudo de caso foi a implementação de referência Tele Assistente System (TAS) proposta por Weyns and Calinescu (2015). A aplicação segue os princípios propostos na arquitetura de referência MAPE-K, o que motivou a escolha desta aplicação no estudo de caso. O material utilizado nesse estudo de caso está disponível na conta de um dos autores autor no GitHub[2].

[2]`https://github.com/davimonteiro`

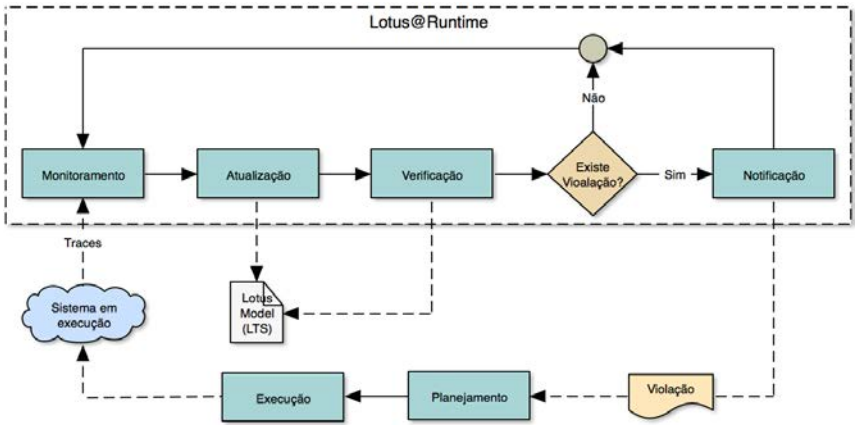

Figura 2. Fluxo de uma adaptação

A implementação de referência da aplicação TAS foi desenvolvida utilizando a plataforma ReSeP [Weyns and Calinescu 2015], que seleciona serviços durante a etapa de planejamento de acordo com uma política de menor custo ou maior confiabilidade que deve ser selecionada antes da inicialização de uma aplicação autoadaptativa. Assim, caso uma aplicação autoadaptativa seja executada utilizando a política de maior confiabilidade, o planejador deverá selecionar os serviços com menor taxa de falhas.

4.1. Implementação do estudo de caso

O presente estudo de caso concentra seus esforços em realizar as etapas de monitoramento e verificação na construção do TAS utilizando o Lotus@Runtime. Dessa forma, para validar o estudo de caso da ferramenta proposta, foi necessário realizar as seguintes atividades: (i) criar o modelo probabilístico com o auxílio da ferramenta LoTuS; (ii) configurar o Lotus@Runtime através do *plugin* de configuração; (iii) adicionar ao TAS a capacidade de gerar rastros de execução; (iv) receber as notificações sobre as violações na etapa de planejamento e modificar a estratégia de seleção dos serviços.

Como resultado da primeira atividade, foi criado na ferramenta LoTuS o modelo LTS probabilístico, ilustrado na Figura 3. Esse modelo foi baseado no modelo em DTMC apresentado pelos autores Calinescu *et al*. (2013). Em seguida, a configuração do Lotus@Runtime foi feita com o auxílio do *plugin* Lotus@Runtime Configuration. O arquivo de configuração é composto pelo nome da configuração, o caminho onde está localizado o arquivo de *logs* do sistema, o caminho onde está localizado o arquivo contendo o modelo probabilístico e as 2 propriedades que se desejam verificar, que são: (P1) a probabilidade de uma falha após o serviço de alarme deve ser menor que 20%, representado por alcançar o estado 5 a partir do estado 0, e (P2) a probabilidade de uma falha acontecer após o serviço de farmácia deve ser menor que 10%, o que é representado pelo alcance do estado 11 a partir do estado 0.

Posteriormente, para adicionar ao TAS a capacidade de gerar rastros de execução, foi necessário instrumentar o código fonte para que cada funcionalidade do TAS que corresponde a uma ação no modelo de comportamento da aplicação fosse gravada num *log*. Um exemplo de rastro de execução gerado pela aplicação é *start, vitalParamMsg, analyzeData, changeDrug, notifyPA, stopMsg, exit*. O arquivo de *logs* gerado foi utilizado como entrada para a o *plugin* Lotus@Runtime Configuration, conforme descrito anteriormente.

A estratégia de monitoramento adotada no estudo de caso foi a checagem de novos rastros a cada alteração no arquivo de *logs*.

Por último, para receber as notificações sobre as violações, foi implementada a classe `LotusRuntimeHandler` dentro da etapa de planejamento. Para realizar a troca de uma estratégia de seleção de serviços dentro do TAS, foi necessário alterar o comportamento do *framework* ReSeP. Assim, foi possível selecionar qual estratégia de seleção de serviços deve ser utilizada, de acordo com as informações recebidas após a etapa de análise realizada pelo Lotus@Runtime. A estratégia de seleção de serviços utilizada foi: caso P1 seja seja violada, então será selecionado os serviço de menor custo; caso P2 não seja satisfeita, então será selecionado o serviço com maior confiabilidade. É importante ressaltar que o presente trabalho não se propõe a comparar a estratégia de seleção de serviços adotada neste estudo de caso com a estratégia adotada no trabalho de Weyns e Calinescu (2015).

Fazendo uma análise sobre o estudo de caso, podemos relatar que não foi complicado adaptar o código-fonte do TAS para utilizar o Lotus@Runtime nas fases de monitoramento e análise. Além disso, pelo fato das estratégias de planejamento e execução das adaptações já estarem implementadas no *framework* ReSeP, o esforço para também adequar essas etapas ao estudo de caso foi pequeno. Contudo, vale ressaltar que o estudo de caso introduziu duas técnicas que não haviam na aplicação original: o monitoramento em tempo real da aplicação através dos rastros de execução e o uso de um modelo atualizado em tempo de execução para verificação de propriedades.

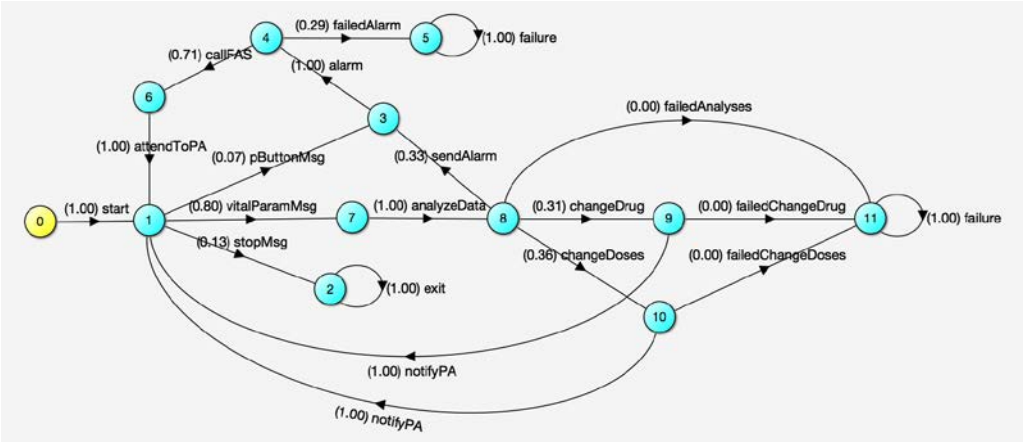

Figura 3. Modelo do Tele Assistence System (TAS)

5. Trabalhos Relacionados

No trabalho de Goldsby *et al.*(2008), os autores descrevem a ferramenta AMOEBA-RT que foi desenvolvida com o objetivo de realizar verificações em tempo de execução de sistemas autoadaptativos. A ferramenta faz o uso de uma abordagem não intrusiva baseada em programação orientada a aspectos para coletar informações sobre o estado atual do sistema. Após serem coletadas, essas informações são enviadas para um servidor remoto, onde um verificador de modelo confere se o estado atual do sistema satisfaz as propriedades que foram especificadas em formalismo baseado em LTL. Em resposta às violações detectadas, a ferramenta grava o caminho de execução das violações em um relatório de erros no qual é processado *offline*.

Em Arcaini *et al.*(2012), os autores apresentam o CoMA, uma ferramenta para monitoramento em tempo de execução de aplicações Java. Nessa proposta, são utilizadas anotações Java para associar o código-fonte do sistema monitorado com o modelo descrito no formalismo máquinas de estados abstratos (ASM). O monitor da ferramenta consegue extrair as informações sobre o estado do sistema em tempo de execução utilizando a ferramenta AspectJ. Após a extração dessas informações, o monitor verifica se o comportamento do sistema está em conformidade com o comportamento esperado que foi especificado no formalismo ASM. Caso o monitor encontre uma não conformidade, será reportado um relatório de erro que poderá ser analisado quando o sistema estiver *offline*.

No trabalho de Filieri *et al.*(2011) é proposta uma abordagem para verificação de modelo em tempo de execução mais eficiente do que as soluções tradicionais como o PRISM. O autor propõe uma abordagem que utiliza DTMC para expressar o modelo do sistema e representá-lo através da linguagem formal PCTL (*Probabilistic Computation Tree Logic*) para expressar as propriedades do sistema. Os resultados apresentados pelo autor mostram que sua abordagem, diferente das ferramentas PRISM e MRMC, obteve uma performance constante mesmo com o aumento do número de estados do DTMC.

No trabalho de Calinescu *et al.*(2013), os autores apresentam o *framework* COVE para criar sistemas autoadaptativos baseados em serviços. Os sistemas autoadaptativos construídos com base no COVE devem, de preferencia, ter serviços redundantes com diferentes taxas de confiabilidade e custo. Dessa forma, o COVE pode realizar verificações formais no modelo a fim de garantir que o sistema utilize serviços confiáveis com o mínimo de custo possível. Para tal, o framework realiza verificações utilizando a ferramenta PRISM [Hinton et al. 2006] para analisar o modelo do sistema definido em DTMC.

Os trabalhos acima utilizam verificadores externos e formalismos para representar o comportamento de sistemas. Diferentemente, o presente trabalho realiza a verificação em um ambiente integrado que abrange desde a construção do modelo de comportamento de forma gráfica até a notificação de violações encontradas em tempo de execução.

6. Considerações Finais

Este trabalho apresentou o Lotus@Runtime para realizar, de forma integrada, o monitoramento, a verificação e a notificação de violações encontradas durante a execução de um sistema autoadaptativo. Para validar a ferramenta proposta, foi desenvolvido um estudo de caso para a aplicação de referência Tele Assistance System [Weyns and Calinescu 2015]. Uma limitação para adoção da ferramenta proposta é a capacidade do sistema autoadaptativo gerar rastros de execução que permitam popular o modelo em tempo de execução. Outra limitação atualmente é necessidade de implementar as etapas de planejamento e execução para completar o ciclo MAPE-K. Como trabalhos futuros, pretende-se adicionar o monitoramento para aplicações em tempo real e suportar outras etapas do ciclo MAPE-K. Além disso, pretende-se desenvolver outros estudos de caso e realizar comparativos de desempenho, custo e confiabilidade com outras abordagens.

Referências

Arcaini, P., Gargantini, A., and Riccobene, E. (2012). Coma: conformance monitoring of java programs by abstract state machines. In *Runtime Verification*, pages 223–238. Springer.

Bencomo, N., Bennaceur, A., Grace, P., Blair, G., and Issarny, V. (2013). The role of models@ run. time in supporting on-the-fly interoperability. *Computing*, 95(3):167–190.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@ run. time. *Computer*, 42(10):22–27.

Calinescu, R., Ghezzi, C., Kwiatkowska, M., and Mirandola, R. (2012). Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77.

Calinescu, R., Johnson, K., and Rafiq, Y. (2013). Developing self-verifying service-based systems. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 734–737. IEEE.

Chen, B., Peng, X., Yu, Y., Nuseibeh, B., and Zhao, W. (2014). Self-adaptation through incremental generative model transformations at runtime. In *36th International Conference on Software Engineering, Hyderabad*. ACM/IEEE.

Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., et al. (2009). *Software Engineering for Self-Adaptive Systems*, chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131.

Filieri, A., Ghezzi, C., and Tamburrelli, G. (2011). Run-time efficient probabilistic model checking. In *Proceedings of the 33rd international conference on software engineering*, pages 341–350. ACM.

Goldsby, H. J., Cheng, B. H., and Zhang, J. (2008). Amoeba-rt: Run-time verification of adaptive software. In *Models in Software Engineering*, pages 212–224. Springer.

Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. (2006). Prism: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer.

Huebscher, M. C. and McCann, J. A. (2008). A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Maoz, S. (2009). Using model-based traces as runtime models. *Computer*, 42(10):28–36.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, 14(3):54–62.

Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14.

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*, pages 88–92. IEEE Press.