Análise sobre testes automatizados para sistemas multiagentes BDI

Martin Fabichak[1], **Jomi F. Hübner**[1]

[1]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina – (UFSC)
Florianópolis, SC, Brasil

mfabichak@gmail.com, jomi.hubner@ufsc.br

Abstract. *Many researches have being conducted related to automated testing in multiagent systems. Some are related to testing methodologies, other regarding to automatic test generation but only a few are related to automated test tools. However, none of those involve muti-agent systems written in BDI languages. This article analyses past projects and proposes a future work of developing an automated testing framework for BDI languages.*

Resumo. *Diversos tipos de estudos já foram feitos relacionados a testes automatizados em sistemas multiagentes. Alguns relacionados a metodologias de testes, outros relacionados a geração automática de testes e poucos relacionados a ferramentas de testes. Porém, nenhum destes foram pensando em sistemas de agentes escritos em linguagens BDI. Este artigo tem como propósito analisar o que já foi feito e propor um trabalho futuro visando o desenvolvimento de um framework para testes automatizados em linguagens BDI.*

1. Introdução

Sistemas multiagentes (SMA) estão sendo explorados em diversas vertentes nas últimas duas décadas. Seja em definição de metologias [Brinkkemper 1996, Bernon et al. 2003], exploração de sua organização ou na definição mais complexa de seu funcionamento com o paradigma BDI [Bordini and Hübner 2006, Bratman 1999]. Porém, pouca atenção tem sido dada em como eles podem ser testados [Cernuzzi et al. 2005].

Testes são difíceis e consomem tempo. Frequentemente mais de 50% do custo de desenvolvimento é gasto em testes [Kit and Finzi 1995]. Ao mesmo tempo, testar é crítico para assegurar a qualidade e reduzir gastos de problemas de software. Segundo a NIST [Tassey 2002] cerca de $ 59.5 bilhões de dólares são gastos anualmente para corrigir problemas de software e mais de um terço poderia ser economizado se melhores testes fossem feitos. Utilizar testes usuais em sistemas multiagentes é uma tarefa desafiadora porque estes sistemas são distribuídos, autônomos e deliberativos [Nguyen et al. 2011]. Há problemas relacionados a interoperabilidade, comunicação e coordenação, que são funcionalidades difíceis de criar e programar [Omicini et al. 2004], e também de testar.

Dentre os poucos projetos que exploraram este tema, alguns autores como Zhang [Zhang et al. 2011], Coelho [Coelho et al. 2006] e Nguyen [Nguyen et al. 2008] publicaram trabalhos expressivos na área de testes automatizados para SMA. Porém, todos esses projetos foram desenvolvidos com a utilização de linguagens orientadas a objetos com *frameworks* orientados a agente (como o JADE) [Bellifemine et al. 1999]

e não linguagens específicas para agentes BDI (como o Jason, 2APL, entre outros) [Bordini and Hübner 2006].

Enquanto a seção 2 deste artigo detalha o estado da arte de testes em *software*, a seção 3 analisa diversos artigos e publicações sobre o tópico em sistemas multiagentes. Finalmente, a seção 4 apresentará uma proposta de um projeto nesta área.

2. Testes Automatizados

Testar um software é o processo que revela inconsistências entre o comportamento esperado e o apresentado de um sistema [Eytani et al. 2008]. Testes automatizados consistem em criar programas que testam outros programas a fim de descobrir tais inconsistências. Testes automatizados estão sendo cada vez mais utilizados como um método de teste em sistemas de software para aumentar a eficiência e eficácia do processo de teste [Runeson 2006].

Teste não são mais vistos como uma extensa fase do projeto e sim algo que permeia todo processo de desenvolvimento. A figura 1 ilustra a relação entre fases do desenvolvimento e testes, que, segundo [Myers et al. 2011], são:

1. Testes de aceitação focam em achar defeitos em requerimentos;
2. Testes de sistema focam em achar defeitos na especificação do sistema;
3. Testes de integração focam em achar inconsistências entre todas interfaces;
4. Testes unitários focam se um pedaço de código (métodos, classes ou até mesmo partes de um agente) funcionam da maneira correta.

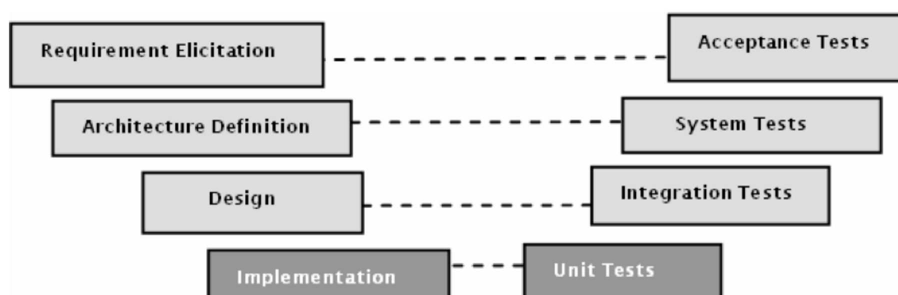

Figura 1. Relação entre diversas fases de desenvolvimento e testes em *software*. [Myers et al. 2011]

[Kaner 1988] define dois termos sempre utilizados:

- Suites de teste, que consiste em um conjunto de casos de teste e de operações que são realizadas antes e depois dos testes;
- Casos de teste, que são efetivamente um teste que será feito e faz um afirmação (*assert*) de como esse teste deveria terminar.

Muitas metodologias utilizam testes unitários como parte importante do desenvolvimento de todo projeto. Beck [Beck 2002], criador do *Extreme Programming*, diz que testes unitários é uma técnica importante para verificar a acurácia e confiança de sistemas.

Muito da literatura e divulgação de testes automatizados vieram do paradigma de orientação a objetos, das quais o paradigma de agentes pode se inspirar, mas, como resultado da sua natureza específica, novos métodos são necessários [Houhamdi 2011].

Trabalho	Nível de Teste	Principal Contribuição	Tecnologias envolvidas
[Zhang et al. 2011]	Unitário	Teste de objetivos relacionados ao eventos que o disparam; identificação da melhor ordem de execução dos testes	não especificado
[Ekinci et al. 2009]	Unitário	Teste de objetivos; criação de um *framework* que leva em conta a criação, a execução e a verificação do resultado dos testes	SEAGENT (Java)
[Coelho et al. 2006]	Agente	Criação de um *framework* baseado em JADE e JUNIT para se criar um agente *mock* [Mackinnon et al. 2001]	JADE (Java)
[Lam and Barber 2005]	Agente	Proposta de um processo semi-automatizado para compreender o comportamento de um agente, utilizando o que foi aprendido para encontrar comportamentos estranhos	múltiplos
[Serrano and Botia 2009, Botia et al. 2004]	Integração	Proposta de uma análise utilizando *data mining* de todas as comunicações feitas entre agentes, criando um grafo de comunicações e verificando posteriormente se todas foram satisfeitas. Ferramenta criada para JADE	Jade (Java)
[Padgham et al. 2005]	Integração	Criação de testes através de artefatos de design (planos e protocolos de interação) para identificação de erros em tempo real.	Jack (Java)

Tabela 1. Análise entre trabalhos relacionados

3. Testes automatizados em sistemas multiagentes

Existem alguns trabalhos relacionados a testes automatizados em SMA, nos diversos níveis apresentados na figura 1. [Nguyen et al. 2011] define um novo nível de testes relacionados a agentes e categoriza diversos trabalhos relacionados dentre os níveis e também em dois aspectos: passivo e ativo. Sendo testes passivos relativos a trabalhos que observam o *output* do sistema, geralmente tendo os dados de entrada pré-programados; Testes ativos são aqueles que analisam o *output* para gerar novos e melhores *inputs* para que os testes sejam melhorados. Já [Houhamdi 2011] analisou diversos trabalhos verificando o quão usáveis são os produtos das pesquisas, também dividindo pelos níveis de teste. Dos trabalhos estudados pelos dois autores, muitos são relacionados a metodologia de desenvolvimento (ou seja, identificar quais testes precisam ser feitos, geralmente sendo parte de metologias usuais de SMA como MASE, PROMETHEUS ou Tropos) enquanto poucos geraram ferramentas ou métodos de se testar um SMA ou partes dele. Nenhum teste compreendeu ambientes e organização [Boissier et al. 2013, Omicini et al. 2008], que, segundo [Weiss 1999], aumentam o impacto na performance de curto e longo prazo de um sistema multiagente.

Na tabela 1 estão listados alguns dos diversos trabalhos relacionados ao tema, mais próximos do projeto de mestrado apresentado na seção 4. Após a análise de tais trabalhos, é fácil de notar que não houve estudos relacionados diretamente testes automatizados em linguagens de agentes *BDI*. Alguns trabalhos são direcionados a *BDI*, e, por exemplo, a base de crenças até é utilizada para se testar os agentes, porém nunca para uma linguagem *BDI*.

4. Trabalho futuro

Trabalhos futuros serão feitos no programa de mestrado e espera-se que a contribuição acadêmica seja na criação de uma ferramenta e de métodos para testes automatizados utilizando linguagens *BDI*.

Partiremos da hipótese de que é possível definir um *script* (como no teatro) na qual todo o sistema precisa seguir para passar nos testes. Um mecanismo de testes receberá todas as comunicações de todos agentes e a percepção de todos artefatos do ambiente. Para o desenvolvimento deste *script* de teste, algumas perguntas precisam ser respondidas: como seriam tais testes? Como criar diferentes casos de testes? Como os agentes seriam testados? Como forçar em tempo real as situações descritas no *script*? Como deixar a criação e manutenção do *script* fácil o suficiente para que seja realmente produtivo utilizar o método em aplicações? Essa hipótese foi criada se pensando na facilidade de se criar testes e de executá-los. Utilizar uma maneira mais convencional como na de engenharia de *software* é a melhor opção para manter a simplicidade para o desenvolvedor.

Por exemplo, suponho uma situação onde, se um agente *a1* com papel *p1* passa a acreditar na crença *c* graças ao evento *e*. outro agente *a2* de papel *p2* também precisa ter em sua base de crenças *c*. Uma possível abordagem seria:

```
1.      agent a1(p1), a2(p2);
2.      define_event(e):
3.              message(a1, tell, c);
4.      when(e):
```

```
5.              assert ? a1 believes c;
6.              assert ? a2 believes c;
7.      trigger(e);
```

Na linha 1 ocorre a definição dos agentes com seus determinados papeis; Nas linha 2 e 3, define-se o evento *e* caracterizado pelo envio de uma mensagem para *a1* com o comando *tell* [Bellifemine et al. 1999] a crença *c*; Na linha 4 define-se o que deve acontecer quando o evento *e* ocorrer; Nas linhas 5 e 6, há o teste de fato através do comando *assert*: verifica-se se os agentes *a1* e *a2* acreditam em *c*; Na linha 8, o teste é iniciado com a execução do evento *e*.

Se a hipótese for correta e este pequeno trecho de código puder ser feito, será possível descrever diversos tipos de testes que se iniciam de diversos pontos da aplicação.

Referências

Beck (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bellifemine, F., Poggi, A., and Rimassa, G. (1999). JADE: A FIPA-compliant agent framework. In *Proceedings of the fourth conference on the practical application of intelligent agents and multi-agent technology*, pages 97–108, London, UK.

Bernon, C., Gleizes, M.-P., Peyruqueou, S., and Picard, G. (2003). Adelfe: A methodology for adaptive multi-agent systems engineering. In *Proceedings of the 3rd International Conference on Engineering Societies in the Agents World III*, ESAW'02, pages 156–169, Berlin, Heidelberg. Springer-Verlag.

Boissier, O., Bordini, R. H., Hübner, J., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761. Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

Bordini, R. H. and Hübner, J. F. (2006). Bdi agent programming in agentspeak using jason. In *Proceedings of the 6th International Conference on Computational Logic in Multi-Agent Systems*, CLIMA'05, pages 143–164, Berlin, Heidelberg. Springer-Verlag.

Botia, J. A., Lopez-Acosta, A., and Gomez-Skarmeta, A. F. (2004). Aclanalyser: A tool for debugging multi-agent systems. In de MÃ¡ntaras, R. L. and Saitta, L., editors, *ECAI*, pages 967–968. IOS Press.

Bratman, M. (1999). *Intention, plans, and practical reason*. The David Hume series of philosophy and cognitive sciences reissues. Center for the Study of Language and Information.

Brinkkemper, S. (1996). Method engineering: engineering of information systems development methods and tools. *Information and Software Technology*, 38(4):275–280.

Cernuzzi, L., Cossentino, M., and Zambonelli, F. (2005). Process models for agent-based development. *Eng. Appl. Artif. Intell.*, 18(2):205–222.

Coelho, R., Kulesza, U., von Staa, A., and Lucena, C. (2006). Unit testing in multi-agent systems using mock agents and aspects. In *Proc. SELMAS '06*, pages 83–90, New York, NY, USA. ACM.

Ekinci, E. E., Tiryaki, A. M., Çetin, O., and Dikenelli, O. (2009). Agent-oriented software engineering ix. pages 173–186. Springer.

Eytani, Y., Tzoref, R., and Ur, S. (2008). Experience with a concurrency bugs benchmark, validation workshop. ICSTW '08, pages 379–384. IEEE Computer Society.

Houhamdi, Z. (2011). Multi-Agent System Testing: A Survey. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 2(6).

Kaner, C. (1988). *Testing Computer Software*. TAB Books, USA.

Kit, E. and Finzi, S. (1995). *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Lam, D. and Barber, K. (2005). Debugging agent behavior in an implemented agent system. In *Programming Multi-Agent Systems*, volume 3346 of *Lecture Notes in Computer Science*, pages 104–125. Springer Berlin Heidelberg.

Mackinnon, T., Freeman, S., and Craig, P. (2001). Extreme programming examined. chapter Endo-testing: Unit Testing with Mock Objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*.

Nguyen, C. D., Perini, A., Bernon, C., Pavón, J., and Thangarajah, J. (2011). Testing in multi-agent systems. In Gleizes, M. P. and Gómez-Sanz, J. J., editors, *Agent-Oriented Software Engineering X*, pages 180–190. Springer.

Nguyen, D. C., Perini, A., and Tonella, P. (2008). A goal-oriented software testing methodology. *Agent-Oriented Software Engineering VIII*, pages 58–72.

Omicini, A., Ossowski, S., and Ricci, A. (2004). Coordination infrastructures in the engineering of multiagent systems. In *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11, chapter 14.

Omicini, A., Ricci, A., and Viroli, M. (2008). Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456.

Padgham, L., Winikoff, M., and Poutakidis, D. (2005). Adding debugging support to the prometheus methodology. *Engineering Applications of Artificial Intelligence*, 18(2):173 – 190. Agent-oriented Software Development.

Runeson, P. (2006). A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29.

Serrano, E. and Botia, J. (2009). Infrastructure for forensic analysis of multi-agent systems. In Hindriks, K., Pokahr, A., and Sardina, S., editors, *Programming Multi-Agent Systems*, volume 5442 of *Lecture Notes in Computer Science*, pages 168–183. Springer.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology.

Weiss, G., editor (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Zhang, Z., Thangarajah, J., and Padgham, L. (2011). Automated testing for intelligent agent systems. In *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 66–79. Springer.