2
Sistemas Multi-Agentes e Ambientes de Desenvolvimento de Software

Resumo

Este capítulo apresenta uma caracterização de Sistemas Multi-Agentes, uma breve introdução a Ambientes de Desenvolvimento de Software e trabalhos relacionados.

2.1.
Caracterização de SMA

O mundo do software é um dos mais ricos e diversos. Milhares de produtos de software estão disponíveis aos usuários hoje, fornecendo uma larga variedade de informações e de serviços em uma larga variedade de domínios. Enquanto a maioria destes programas fornecidos aos usuários são usados isoladamente, está aumentando a demanda para os programas interoperáveis (para troca de informações e serviços com outros programas e, desse modo, resolução dos problemas que não podem ser resolvidos sozinhos).

Parte do que dificulta a interoperabilidade é a heterogeneidade (os sistemas são distribuídos e incluem uma combinação de hardware e software). Os programas são escritos por pessoas diferentes, em horas diferentes, em línguas diferentes; e, conseqüentemente, freqüentemente fornecem interfaces diferentes. As dificuldades criadas pela heterogeneidade prejudicam a dinâmica no ambiente de software. Os programas são freqüentemente reescritos; programas novos são adicionados e programas velhos removidos.

Atualmente, a maioria dos sistemas de software são concorrentes e distribuídos, esperando-se interagir com componentes e serviços que são encontrados dinamicamente na rede. Os sistemas de software estão tornando-se entidades que não podem ser paradas, restauradas, e mantidas de maneira tradicional. E como uma conseqüência natural, os sistemas de software tendem a

ser abertos, existindo um ambiente de operação dinâmico onde novos componentes são incorporados e componentes existentes saem em uma base contínua, e onde as condições de operação podem mudar de uma maneira imprevisível.

Desta forma, utilizamos sistemas multi-agente (SMA) como um paradigma da tecnologia de programação para projetar e desenvolver sistemas de software complexos [15; 34], facilitando a criação de software interoperável. Em SMA, as aplicações são projetadas e desenvolvidas nos termos das entidades autônomas (isto é, a habilidade para decidir que ações devem ser tomadas e em que hora [31]) do software (agentes) que podem atingir seus objetivos interagindo com outros agentes em termos de protocolos e linguagens de alto-nível. Os agentes podem ser tão simples quanto sub-rotinas; mas tipicamente são entidades maiores com algum controle de persistência (por exemplo, threads de controle distinto dentro de um único espaço de endereço, processos distintos em uma única máquina, ou processos separados em máquinas diferentes).

Uma característica importante da linguagem usada por agentes é a expressividade. Permite a troca de dados e informação lógica, comandos individuais e scripts (isto é programas). Usando esta linguagem, ocorre a comunicação de informações e objetivos complexos, diretamente ou indiretamente entre os agentes.

A tecnologia de programação baseada em Agente é comparada freqüentemente à programação orientada a objetos. Como um “objeto", um agente fornece uma interface baseada na mensagem independente da sua estrutura de dados e algoritmos internos. A principal diferença entre as duas aproximações está na linguagem da interface. Em geral, na programação orientada a objetos, o significado da mensagem pode variar de um objeto para outro. Na tecnologia de programação baseada em agentes, os agentes usam uma linguagem comum com uma semântica independente do agente.

2.1.1. Conceitos Básicos

O primeiro conceito chave é agente, visto aqui como uma entidade do software que exibe as seguintes características para atingir seus objetivos de projeto [31]:

• Autonomia. Um agente não é passivamente sujeito a um fluxo global, ou a um fluxo externo de controle em suas ações. Isto é, um agente tem sua própria thread interna de execução, orientada tipicamente à realização de uma tarefa específica, e decide-se que ações deve executar em que hora.

• Localidade (Situatedness). Os agentes executam suas ações quando localizados em um ambiente particular. O ambiente pode ser computacional (por exemplo, um Web site) ou físico (por exemplo, manufacturing pipeline), e um agente pode detectar e efetuar algumas parcelas dele.

• Proatividade. A fim de realizarem seus objetivos de projeto em um ambiente dinâmico e imprevisível o agente pode necessitar agir para se assegurar de que seus objetivos foram atingidos e que os novos objetivos serão oportunamente atingidos sempre que apropriado.

Por exemplo, um componente de software para filtrar emails pode ser visto como um (simples) agente [16]. É autônomo se for implementado como uma thread que detecta a caixa de correiro dos emails do cliente e se a ele estiver atribuída a tarefa específica de analisar e de modificar o índice de algumas caixas postais do usuário. É proativo pois pode atrair a atenção do usuário aos correios novos específicos ou às situações específicas que ocorrem em suas pastas. Localizado pois vive no mundo das caixas postais, podendo detectar mudanças nas caixas postais e afetar seu estado.

Os agentes podem ser úteis como entidades autônomas a que são delegadas tarefas particulares em nome de um usuário (como no exemplo acima). Entretanto, na maioria dos casos, os agentes existem nos ambientes que contêm outros agentes. Nestes sistemas multi-agentes, o comportamento global deriva-se da interação entre os agentes constituintes. Isto traz-nos ao segundo conceito chave da computação baseada em agente, a sociabilidade [31]:

• agentes interativos cooperam, coordenam ou negociam um com o outro, seja para atingir um objetivo comum ou porque isto é necessário para ele atingir seus próprios objetivos.

Amplamente falando, é possível distinguir entre duas classes principais de SMAs:

• problema de resolução de sistemas distribuídos em que os agentes componentes são projetados explicitamente para atingir cooperativamente um dado objetivo;

• sistemas abertos em que os agentes não são projetados para compartilhar um objetivo comum, e foram desenvolvidos possivelmente por pessoas diferentes para atingir objetivos diferentes. Além disso, a composição do sistema pode dinamicamente variar enquanto os agentes incorporam e saem do sistema.

2.2. Ambiente de Desenvolvimento de Software (ADS)

Todos os engenheiros de software usam ferramentas, desde os primeiros programas em assembly. Algumas pessoas usam ferramentas autônomas (stand-alone), enquanto outras usam um conjunto de ferramentas integradas, chamadas ambientes. Com o tempo, o número e a variedade de ferramentas cresceu tremendamente, variando desde ferramentas tradicionais como editores, compiladores e depuradores, às ferramentas que ajudam na especificação de requisitos, design, construindo GUIs, gerando consultas, definindo mensagens, arquitetando sistemas e conectando componentes, testando, gerenciando o controle de versão e da configuração, administrando bases de dados, fazendo reengenharia, engenharia reversa, análise, programação visual, e recolhendo métricas, aos ambientes de processo centralizado da tecnologia de programação que cobrem o ciclo de vida inteiro, ou ao menos às parcelas significativas dele. Certamente, a tecnologia de programação moderna não pode ser realizada sem sustentação razoável de ferramentas.

O papel dos computadores, seu poder, e a sua variedade, estão aumentando em um ritmo dramático. A competição é afiada durante todo a indústria de computadores, e o momento do mercado determina freqüentemente o sucesso. Com isso, conseqüentemente, os softwares são produzidos rapidamente e a custos razoáveis. Isto envolve geralmente alguma mistura da escrita, da adaptação e da integração de novo software com o software existente. Ferramentas e ambientes de suporte podem ter um efeito dramático sobre o quão rapidamente estas podem ser feitas, quanto custarão, e na qualidade do resultado. Determinam freqüentemente se pode ser feita, dentro da realidade econômica e outros pontos, tais como a segurança e a confiabilidade.

Alguns temas comuns são aparentes durante toda a história do estudo de ferramentas e ambientes de tecnologia de programação. Não surpreendentemente, talvez o mais comum é o tema da integração. A necessidade para a integração – de

ferramentas, processos, artefatos, e visões – foi a principal causa de mudanças para novas linhas de pesquisas [7]. A integração foi realizada usando uma grande quantidade de aproximações – texto ASCII, repositório compartilhado, representações e interfaces padronizadas, transformação e adaptação, integração baseada em evento, estruturas, etc. – e ferramentas e plataformas que os facilitam.

A razão chave é que cada ferramenta ou ambiente é ainda altamente específico a algum contexto [7]. Pode requerer um software que seja escrito em uma linguagem particular, ou ser representado usando um tipo particular de formulário intermediário ou programa de base de dados. Pode funcionar somente num hardware particular, sob um sistema operacional particular, ou usar uma plataforma particular do compilador, do ambiente ou da integração. Pode requerer a presença de uma variedade de outras ferramentas ou componentes de infra-estrutura que o desenvolvedor pode nem ou não desejar usar. Não pode conter "âncoras" necessárias para permitir o trabalho correto com outro software que o desenvolvedor deve usar.

O principal desafio para a comunidade de ferramentas e ambientes é, conseqüentemente, encontrar maneiras para construir e integrar ferramentas, de modo que as potencialidades dentro delas possam facilmente ser adaptadas para o uso em contextos novos [7]. É improvável que os contextos se tornem padronizados suficientemente para permitir o "plug-and-play", a história demonstra que a padronização não pode ser confiada como uma solução completa, porque, entre outras razões, os novos domínios são identificados como bons padrões necessários somente depois que os desenvolvedores em mais de um grupo já tenham construído software para eles. Apesar disso, a adaptação e a integração são potencialidades chaves. As ferramentas possuem um papel crítico em permitir adaptação e integração flexíveis, assim isso se transforma num desafio dobrado para a comunidade das ferramentas:

• Desenvolvendo e usando ferramentas arquiteturas e outras aproximações que facilitam a adaptação e a integração.

• Construindo ferramentas para ajudar com o processo de adaptação e integração.

2.3. Trabalhos Relacionados

Muitas metodologias e linguagens de modelagem foram propostas para modelagem de sistemas multi-agente e diversas plataformas e frameworks foram propostos para implementação de tais sistemas. Entretanto, pouco trabalho tem sido feito no propósito de mapear os modelos orientados a agente em código. Metodologias, tais como Gaia [6] e MaSE [17], não fornecem nenhuma orientação para a implementação. Em [6], os autores afirmam que Gaia não trata diretamente das questões de implementação. Embora os autores de MaSE [17] afirmem que o foco preliminar de MaSE é ajudar nas fases de requerimentos, análise, modelagem, e implementação, a metodologia não descreve como os modelos do projeto são implementados em qualquer plataforma existente.

Em [1], os autores propõem um mapeamento dos conceitos i * usados pela metodologia Tropos a um ambiente de desenvolvimento orientado a Belief Desire Intention (BDI) chamado Jack [21]. Cada conceito i * é mapeado em um conceito BDI que é mapeado consecutivamente para Jack. Embora o trabalho descreva os mapeamentos, não os exemplifica. Não demonstra o mapeamento de algum conceito i * usado para modelar uma aplicação orientada a agente. Além disso, os autores também não detalham a implementação dos planos e dos agentes da aplicação que estendem as abstrações de Plano e Agente propostas em Jack.

A metodologia Prometheus [1] fornece também uma sustentação de "início-a-fim" da especificação para o detalhamento da modelagem e da implementação. Em [21], os autores também propõem o uso de Jack para implementar os modelos de Prometheus. Em [22], o ambiente do desenvolvimento Jack Development Environment (JDE) é apresentado como um instrumento de apoio para modelagem de sistemas Prometheus e para implementação de sistemas orientado a agentes usando Jack. Entretanto, nenhum mapeamento dos artefatos do sistema para implementação fornecidos por Jack é descrito, nem um exemplo de sistema modelado usando Prometheus e implementado usando Jack é apresentado.

Em [1], o autor demonstra a geração do código dos diagramas de seqüência de AUML. Os autores focalizam em exemplificar o mapeamento de um protocolo de interação do agente ao código de Java. Entretanto, o mapeamento não é baseado em nenhuma arquitetura ou estrutura de implementação orientada a

agente. Estendendo e customizando frameworks, o processo de implementação de uma aplicação torna-se fácil e mais rápido desde que a parte do código da aplicação já seja escrita e compilada no framework.

Neste trabalho, é apresentado e demonstrado o mapeamento da modelagem do sistema até a implementação utilizando a linguagem de modelagem ANote para a especificação do sistema e a arquitetura de desenvolvimento ASYNC para a fase de implementação.