Comunicação entre agentes no *framework* Embedded-BDI*

Vitor Luis Babireski Furio [1], Maiquel de Brito[1], Carlos Roberto Moratelli[1]

[1]Universidade Federal de Santa Catarina (UFSC)
Rua João Pessoa, 2750 – 89036-002 – Blumenau – SC – Brasil

vitorluis.babireskifurio@gmail.com,{maiquel.b,carlos.moratelli}@ufsc.br

Abstract. *The Embedded-BDI framework provides resources for the implementation and execution of BDI agents on hardware with limited resources, typical of cyber-physical systems, without providing resources for implementing communication among the agents. This work extends the framework by adding features to specify communication between agents using high-level elements related to the BDI model and to transmit messages using protocols and resources consolidated in cyber-physical systems. The experimentally evaluated results demonstrate that the extension adds communication capability to agents.*

Resumo. *O* framework Embedded-BDI *provê recursos para a implementação e execução de agentes BDI em* hardwares *com recursos limitados, típicos de sistemas ciberfísicos, sem fornecer recursos para implementar comunicação entre agentes. Este trabalho estende o* framework *adicionando recursos para especificar a comunicação entre agentes usando elementos de alto nível relacionados ao modelo BDI e para transmitir as mensagens usando protocolos e recursos consolidados em sistemas ciberfísicos. Os resultados, avaliados experimentalmente, demonstram que a extensão proposta adiciona capacidade de comunicação aos agentes.*

1. Introdução

Sistemas ciberfísicos (ou CPS, do inglês *Cyber-Physical Systems*) combinam programas de computador, embarcados em diferentes dispositivos, com sensores e atuadores, para, assim, interagir com o ambiente físico. Aplicações complexas podem requerer que CPS tenham características de agentes BDI, como (i) autonomia, para atuar sem a intervenção de um operador humano, (ii) proatividade, para tomar a iniciativa de atingir objetivos de longo prazo, (iii) reatividade, para adaptar-se a novas circunstâncias de operação, e (iv) habilidades sociais, para interagir com outros CPS. A programação orientada a agentes (ou AOP, do inglês *Agent-Oriented Programming*) fornece modelos e ferramentas para o desenvolvimento de agentes que, normalmente, requerem mais recursos computacionais (processamento, memória etc) do que dispõem os *hardwares* típicos de CPS. Para tratar desta limitação, o *framework Embedded-BDI*[1] [Santos 2022] fornece ferramentas para implementar e executar agentes BDI neste tipo de *hardware*.

Agentes desenvolvidos com o *framework Embedded-BDI* possuem autonomia, proatividade e reatividade. Este trabalho propõe uma extensão para que o *framework* forneça suporte a comunicação entre agentes. Os resultados são avaliados experimentalmente através de uma aplicação desenvolvida com o *framework* estendido.

[1]https://embedded-bdi.github.io/index.html

2. Fundamentação

2.1. Agentes, AOP e o *framework Embedded-BDI*

Um *agente* é um sistema que percebe o ambiente em que está situado e atua de forma autônoma para satisfazer os objetivos para os quais foi projetado [Wooldridge 2009]. Um agente que segue a arquitetura BDI atua a partir de crenças, que são as informações que ele possui sobre o ambiente, desejos (ou objetivos), que são os estados do mundo que ele gostaria de atingir, e intenções, que são os estados do mundo que ele está atuando para atingir [Rao and Georgeff 1995]. Agentes BDI executam um ciclo de raciocínio contínuo em que (i) percebem o ambiente e atualizam suas crenças a partir disso, (ii) avaliam, a partir das crenças atuais, quais objetivos são factíveis, passando a ter intenções de realizá-los e (iii) executam planos de atuação para satisfazer as intenções.

AgentSpeak [Rao 1996] é uma linguagem para programação de agentes BDI especificando crenças, objetivos e planos. O *framework Embedded-BDI* utiliza uma versão simplificada de *AgentSpeak* para programação de agentes e para execução em *hardwares* limitados, típicos de CPS. A Figura 2 ilustra um código nesta versão. Ela contém todos os elementos da linguagem original. Porém as crenças e objetivos são expressos através de proposições, enquanto no *AgentSpeak* original, são expressos através de predicados. O código *AgentSpeak*, escrito em arquivos com extensão *.asl*, é traduzido para C++ e compilado junto com outros módulos escritos em C++ que implementam (i) funções para conectar ações e atualização de crenças do agente aos sensores e atuadores físicos; e (ii) uma *engine BDI*, que executa o ciclo de raciocínio BDI. Estes componentes são compilados e embarcados no *hardware* (Figura 1).

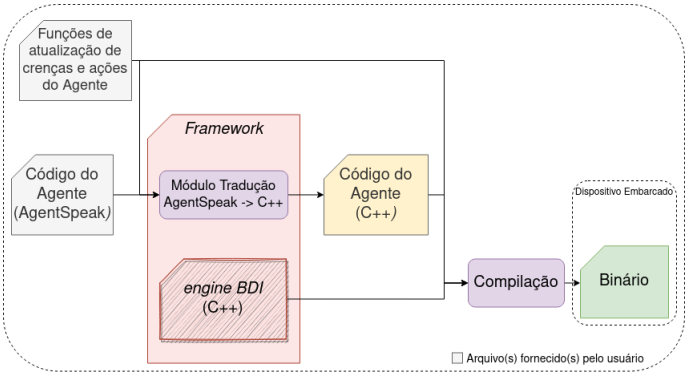

Figura 1. Elementos envolvidos na compilação do agente [Santos 2022]

Para tratar das limitações de memória típicas em CPS, não há alocação dinâmica de memória na *engine BDI*. Por isso, todas as crenças que o agente pode vir a possuir, bem como todos os objetivos que o agente pode buscar atingir, são instanciados quando ele é incializado.[2] Estas crenças e objetivos são aqueles que constam no código fonte do agente (.asl). Por exemplo, o agente implementado pelo código da Figura 2 possuirá somente a crença *happy* e poderá ter somente os objetivos *start* e *hello*. Durante a execução do agente, os objetivos instanciados podem ser ativados e desativados e as crenças instanciadas são associadas ao valor *verdadeiro* quando o agente possui a crença e *falso* caso contrário. Para economizar memória, os identificadores de crenças e objetivos são substituídos por números inteiros no código C++ gerado com a tradução do .asl, seguindo a ordem em que aparecem neste último. Para o código ilustrado na Figura 2, os identificadores *start*, *happy* e *hello* são mapeados, respectivamente, para os valores 0, 1 e 2 (mais detalhes na Seção 3.3).

[2]Há outros elementos que também são instanciados na inicialização do agente, mas que não são relevantes neste trabalho.

```
1  !start. // desejo (ou objetivo)
2  +!start <- +happy. //plano para satisfazer o objetivo ``start''
3  +happy <- !!hello.  //plano para tratar a nova crenca ``happy''
4  +!hello <- say_hello. //plano para satisfazer o objetivo ``hello''
```

Figura 2. Exemplo de código fonte de agente

2.2. MQTT

O *Message Queuing Telemetry Transport* (MQTT) é um protocolo de mensagens leve, projetado para dispositivos IoT (Internet das Coisas). Ele segue o modelo de comunicação de publicação/assinatura. Dispositivos *clientes* se conectam a um servidor central (ou *broker*), que gerencia a comunicação entre eles. Os clientes publicam mensagens em *tópicos*, que são canais de comunicação hierárquicos. Outros clientes podem se inscrever nesses tópicos de interesse. Quando uma mensagem é publicada em um tópico, o *broker* a recebe e a distribui para todos os clientes inscritos nesse tópico.

3. Desenvolvimento

A adição de capacidade de comunicação entre agentes ao *framework Embedded-BDI* requer integrar o ciclo de raciocínio à infraestrutura de comunicação disponível, conciliar as mensagens transmitidas através dessa infraestrutura com as representações do modelo BDI e adaptar as funcionalidades de comunicação adicionadas ao *framework* às suas premissas para otimização de uso de recursos computacionais (cf. seções 3.1 a 3.3).

3.1. Integração do ciclo de raciocínio à infraestrutura de comunicação

Os diferentes *hardwares* usados em CPS têm interfaces e métodos de acesso a redes de comunicação próprios. Para prover alguma generalidade ao *framework*, propõe-se basear a transmissão em algum protocolo para o qual os principais *hardwares* tenham bibliotecas apropriadas. Neste trabalho, considera-se o protocolo MQTT. A integração do ciclo de raciocínio do agente com a infraestrutura de comunicação sobre a qual as mensagens são transmitidas é feita integrando a *engine BDI* ao serviço MQTT. Ela incorpora funcionalidades para ler e escrever em tópicos mantidos pelo *broker*, que é externo ao agente. Ao inicializar, a *engine BDI* se inscreve em (ou *assina*) dois tópicos MQTT. Um deles tem o nome do próprio agente e destina-se a armazenar as mensagens enviadas para o agente em particular. O segundo tópico, de nome *broadcast*, destina-se a armazenar as mensagens que os agentes enviam a todos os demais. No início de cada ciclo de raciocínio, a *engine BDI* detecta novas mensagens nestes tópicos, as interpreta e, na sequência do ciclo, atualiza o estado interno do agente.

3.2. Conciliação das mensagens transmitidas com a semântica do modelo BDI.

As mensagens trocadas entre os agentes têm, inevitavelmente, relação com as abstrações do modelo BDI que representam seu estado interno, podendo, por exemplo, afetar suas crenças e objetivos. É necessário, assim, ligar as mensagens trocadas a essas abstrações. Como é usual em AOP, considera-se, neste trabalho, que os agentes comunicam-se utilizando uma linguagem de comunicação entre agentes (ou ACL, do inglês *agent communication language*). A ACL considerada neste trabalho é KQML (ou *Knowledge Query and Manipulation Language*) [Finin et al. 1994]. Nela, as mensagens são compostas não

só pelo conteúdo a ser comunicado, mas também por uma *performativa* que expressa a intenção que o emissor possui ao enviar a mensagem. As performativas relevantes neste trabalho são *tell*, que denota a intenção do emissor de que o receptor possua uma determinada crença, e *achieve*, que denota a intenção do emissor de que o receptor tenha um determinado objetivo. Por exemplo, um agente que envia o conteúdo *happy* acompanhado da performativa *tell* tem a tem a intenção de que o destinatário passe a ter a crença *happy*. De forma semelhante, um agente que envia o conteúdo *do_something* acompanhado da performativa *achieve* tem a intenção de o receptor passe a ter o objetivo *do_something*.

A comunicação através de KQML e a sua ligação com o ciclo de raciocínio do agente requerem adaptações no *framework* para tratar do envio e recepção de mensagens. Quanto ao envio, considera-se que a capacidade de enviar uma mensagem é parte da implementação do próprio agente, incorporada à *engine BDI*, e não depende de elementos do ambiente. Assim, com inspiração na linguagem *Jason* [Bordini et al. 2007], que também estende *AgentSpeak*, adiciona-se um novo construto, chamado *ação interna*, à linguagem de programação. A linguagem de programação é estendida para incluir as ações internas *send* e *broadcast*, que permitem que o agente envie uma mensagem a outro agente ou a todos os demais, respectivamente. Elas são especificadas na forma de instruções no codigo .asl, como, por exemplo, *.broadcast(tell,happy)*, em que agente emissor envia um *broadcast* com conteúdo *happy* e performativa *tell*; ou *.send(bob,achieve,do_something)*, em que o agente emissor envia uma mensagem diretamente ao agente *bob*, com conteúdo *do_something* e performativa *achieve*.[3] Em tempo de execução, a *engine BDI* executa tudo o que é necessário para que instruções como estas resultem na escrita em tópicos MQTT. A *engine BDI* é estendida também para, ao receber mensagens via MQTT, tratá-las de forma adequada em função da sua performativa: ao processar uma mesagem com a performativa *tell*, atualiza a crença correspondente e, no caso da performativa *achieve*, atualiza os objetivos do agente.

3.3. Adaptação às representações internas do *framework Embedded-BDI*

A representação interna de crenças e objetivos através de números inteiros (cf. Sec. 2.1) impõe um desafio à comunicação entre agentes pois as mesmas crenças e objetivos podem ser mapeados para valores diferentes em agentes cujo código .asl é também diferente. Por exemplo, no caso do caso o agente codificado na Figura 2, a ação interna *.broadcast(tell,happy)*, ao ser traduzida para C++, resultará em uma instrução para enviar uma mensagem com a performativa *tell* e o conteúdo 1, que é associado à crença *happy* naquele agente. Não e possível afirmar, no entanto, que a crença será associada valor 1 no agente receptor. Para tratar deste problema, optou-se em transmitir, na mensagem, o identificador das crenças e objetivos em vez de seu valor inteiro correspondente. Para isso, a *engine BDI* contém uma estrutura de dados chamada *msg_list* que armazena a lista de identificadores e os números inteiros correspondentes.

O processo de envio da mensagem segue as seguintes etapas: (i) busca-se na *msg_list* o identificador correspondente ao número da crença ou objetivo a enviar, e (ii) envia-se esse identificador ao destinatário pelo protocolo MQTT. Para tratar as mensagens recebidas, verifica-se inicialmente se o conteúdo da mensagem, que é um identificador de

[3]No código C++ gerado a partir da tradução do asl, os identificadores de crenças e objetivos (como *happy* e *do_something* neste exemplo) são substituídos por números inteiros (cf. Sec. 2.1). Assim, as mensagens enviadas contém uma performativa e um número inteiro.

crença ou objetivo, está instanciado na *engine BDI*. Caso afirmativo, processa-se a mensagem, convertendo seu conteúdo para o número inteiro correspondente, e então atualizando o estado da crença ou objetivo em função da performativa enviada na mensagem.

4. Resultados

A comunicação entre agentes é ilustrada através de um exemplo envolvendo os agentes *bob* e *alice*.[4] *Bob*, cujo código é ilustrado na Figura 3-a, inicia com o objetivo *start* e a crença *its_night* (linhas 1–2). Ele tem um plano para satisfazer o objetivo *start* (linha 3), no qual envia um *broadcast* com a intenção de que os receptores tenham o objetivo *hello*. *Alice*, cujo código é ilustrado na Figura 3-b, recebe a mensagem e adquire tal objetivo. Para satisfazê-lo, executa o plano das linhas 1–3, em que (i) executa a ação *say_hello* e (ii) envia uma mensagem a *bob* com a intenção de que este tenha o objetivo *is_day*. Ao receber a mensagem, *bob* adquire tal objetivo e, para satisfazê-lo, executa o plano das linhas 6–8, em que (i) executa a ação *say_night* e envia para *alice* uma mensagem com a intenção de que ela possua a crença *its_night*. *Alice* recebe a mensagem, adquirindo a referida crença e, como reação à aquisição, executa o plano da linha 4, disparando a ação *say_its_night*. O *log* da execução dos agentes é exibido nas figuras 4 e 5.

O exemplo demonstra capacidade de comunicação entre agentes. A troca de mensagens é viabilizada pelo protocolo MQTT. Mas, em vez de programar-se mera troca de mensagens naquele protocolo, programa-se comunicação entre agentes, utilizando construtores de alto nível que relacionam os atos de comunicação às intenções do agente ao enviar a mensagem e com as representações do modelo BDI que compõem o estado interno do agente. As ações dos agentes neste exemplo são realizadas por meio de mensagens impressas na tela em vez de atuações feitas por atuadores físicos. Isso é suficiente para avaliar a possibilidade de comunicação entre agentes, considerando-se que a utilização de atuadores físicos é suportada pelo *framework* desde sua versão inicial [Santos 2022].

```
1 !start.
2 its_night.
3 +!start
4   <- say_hello;
5      .broadcast(achieve,hello).
6 +!is_day
7   <- say_night;
8     .send(alice,tell,its_night).
```

(a) bob

```
1 +!hello
2   <- say_hello;
3      .send(bob,achieve,is_day).
4 +its_night <- say_its_night.
```

(b) alice

Figura 3. Códigos dos agentes

5. Conclusões

Os resultados satisfazem o objetivo deste trabalho, que é adicionar funcionalidades para comunicação entre agentes ao *framework Emebdded-BDI*. Comunicação entre agentes em sistemas embarcados é tratada em outros trabalhos (ex.: [de Jesus et al. 2023, Ortiz et al. 2022]). Quanto ao *framework Emebdded-BDI*, a funcionalidade de *broadcast*

[4]O código completo do exemplo, bem como instruções para execução, encontram-se em `http://github.com/VitorFurio/Embedded-bdi`

```
Message arrived on topic broadcast: ACHIEVE/hello
Hello Everyone, I'm Alice!
Sending message to topic bob: ACHIEVE/is_day
Message arrived on topic alice: TELL/its_night
Meh, it's night, I'm going to sleep...
```

Figura 4. Saída do console de Alice

```
Hello, I'm Bob!!
Sending message to topic broadcast: ACHIEVE/hello
Message arrived on topic bob: ACHIEVE/is_day
It's night now, Alice.
Sending message to topic alice: TELL/its_night
```

Figura 5. Saída do console de Bob

é adicionada em [William 2022]. No entanto, aquela abordagem é fortemente acoplada a um *hardware* específico, enquanto este trabalho inclui um certo nível de generalidade ao usar MQTT. Testes preliminares indicam que os agentes implementados com a nova versão do *framework* podem ser executados em dispositivos ESP32 ou equivalentes. Trabalhos futuros incluem testes com outros dispositivos, bem como a medição dos recursos computacionais demandados pelas funcionalidades adicionadas neste trabalho.

Referências

Bordini, R. H., Hübner, J. F., and Wooldridge, M. J. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason*. J. Wiley.

de Jesus, V. S., Lazarin, N. M., Pantoja, C. E., Manoel, F. C. P. B., Alves, G. V., and Viterbo, J. (2023). A middleware for providing communicability to embedded MAS based on the lack of connectivity. *Artif. Intell. Rev.*, 56(Supplement 3):2971–3001.

Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). Kqml as an agent communication language. In *CIKM'94*, page 456–463, New York, USA. ACM.

Ortiz, G., Zouai, M., Kazar, O., de Prado, A. G., and Boubeta-Puig, J. (2022). Atmosphere: Context and situational-aware collaborative iot architecture for edge-fog-cloud computing. *Computer Standards Interfaces*, 79:103550.

Rao, A. S. (1996). Agentspeak(l): BDI agents speak out in a logical computable language. In *MAAMAW 1996*, volume 1038 of *LNCS*, pages 42–55. Springer.

Rao, A. S. and Georgeff, M. P. (1995). BDI agents: From theory to practice. In *ICMAS'1995*, pages 312–319.

Santos, M. (2022). Programação orientada a agentes bdi em sistemas embarcados. Dissertação (mestrado), Universidade Federal de Santa Catarina, Florianópolis, Brasil.

William, J. (2022). Multi-agent systems on ultra low power platforms. Dissertação (mestrado), ETH Zurich, Suiça.

Wooldridge, M. J. (2009). *An Introduction to MultiAgent Systems, Second Edition*. Wiley.