SARA^{MR}: Uma Arquitetura de Referência para Facilitar Manutenções em Sistemas Robóticos Autoadaptativos

Marcos H. de Paula[1], Marcel A. Serikawa[1], André de S. Landi[1], Bruno M. Santos[1]
Renato S. Costa[1], Valter V. de Camargo[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brasil

{marcos.paula,marcel.serikawa,andre.landi}@dc.ufscar.br

{bruno.santos,renato.costa,valter}@dc.ufscar.br

Abstract. *The Self-Adaptive Systems (SAS) architecture has two semantically distinct parts: (i) the managed system, which contains the domain specific features; and (ii) the system manager that monitors and changes the managed system behavior when necessary. In many systems these parts are not modularized difficulting maintenances. This article presents the reference architecture SARA^{MR}, which aims to separate the SAS parts in order to facilitated the management system maintenance. The architecture has been implemented in Java and a case study was developed using a Lego robot. A preliminary assessment was conducted to quantify the maintenance impact on the system. We realized that the reference architecture tends to minimize the maintenance activities produced by evolution impacts of self-adaptatives robotic applications.*

Resumo. *A arquitetura de Sistemas AutoAdaptativos (SAA) possui duas partes semanticamente distintas: (i) o sistema gerenciado, que representa as funcionalidades específicas do domínio; e (ii) o sistema gerenciador, que monitora e altera o comportamento do sistema gerenciado quando necessário. Em muitos sistemas essas partes não estão modularizadas, dificultando manutenções. Este artigo apresenta a arquitetura de referência SARA^{MR}, cujo objetivo é modularizar essas partes de um SAA de forma que as manutenções no sistema gerenciador sejam facilitadas. A arquitetura foi implementada em Java e foi desenvolvido um estudo de caso utilizando-se um robô Lego. Uma avaliação preliminar foi conduzida para quantificar o impacto de uma manutenção no sistema. Percebeu-se que a arquitetura de referência tende a minimizar as atividades de manutenção produzidas por impactos de evolução em aplicações robóticas autoadaptativas.*

1. Introdução

Robôs Móveis Autônomos (RMAs) são capazes de realizar atividades com pouca ou nenhuma intervenção externa. Para que um robô seja considerado autônomo, ele precisa de um controlador que seja um Sistema AutoAdaptativo (SAA) [Baker et al. 2011]. Muitos estudos demonstram que SAAs são baseados na teoria do controle do campo de Engenharia de controle (*Control Theory*). Sendo assim, intrinsecamente usam *loops* de controle (*Control Loops*) em sua arquitetura para realizar as adaptações [Cheng et al. 2005, Brun et al. 2009]. Um *loop* de controle tem as seguintes responsabilidades [Weyns et al.

2013, Kokar et al. 1999]: (i) monitorar e capturar dados de um determinado processo/-contexto; (ii) analisar os dados capturados desse processo/contexto perante um valor de referência; (iii) planejar uma adaptação; e (iv) realizar adaptações no processo/contexto de forma que se aproxime os próximos dados capturados da referência. Geralmente esses *loops* de controle são compostos por quatro elementos: Monitores, Analisadores, Planejadores e Executores. Atualmente, um dos modelos conceituais mais difundidos para esses *loops* de controle é o MAPE-K idealizado pela IBM [IBM 1994].

Em consequência dessas características, a arquitetura de um SAA pode ser denominada como duas partes distintas, sendo: (i) um subsistema controlado e um subsistema controlador. O subsistema controlado representa o sistema propriamente dito, já o subsistema controlador representa o *loop* de controle que realiza adaptações no subsistema controlado. Apesar dessa divisão conceitual de um SAA, esses subsistemas usualmente não se encontram modularizadas e separados no código-fonte. Vários autores apontam que uma das formas de melhorar os níveis de manutenibilidade de SAAs pode ser por meio da modularização e separação desses subsistemas, bem como dos *loops* de controle em entidades de primeira classe, já que diversas atividades de manutenção e evolução em SAAs são voltadas a esses quatro componentes [Weyns et al. 2013, Cheng et al. 2005, Garlan et al. 2004]. Isso significa que esses *loops* de controle devem ser implementados de forma modular e evidente no código-fonte ao invés de ficar espalhado e entrelaçado com outras classes do sistema. Embora alguns autores apresentem propostas de arquitetura para sistemas robóticos autônomos, apenas Albus [Albus 2000] e Affonso e Nakagawa [Affonso et al. 2013] apresentam soluções claras e concretas.

A principal motivação para este trabalho é que, assim como sistemas tradicionais, sistemas robóticos autônomos também necessitam de manutenções, muitas vezes no sentido de evoluir o software que controla o robô. Entretanto, apesar de vários autores reconhecerem que manutenções, nesse contexto, muitas vezes são mais desafiadoras do que em sistemas tradicionais, porém, pouco deles concentram-se em facilitar essas manutenções [Brugali 2007, Georgas and Taylor 2008, Edwards et al. 2009]. Neste artigo é proposta uma arquitetura de referência, chamada SARA^{MR}, para estruturar softwares robóticos autônomos de forma que sua manutenção seja facilitada. Isto é, vislumbra-se que o emprego da SARA^{MR} faz com que atividades de inclusão, alteração e remoção de elementos relacionados aos *loops* de controle sejam realizados de forma mais concentrada e controlada. A arquitetura SARA^{MR} está implementada como um *framework* Java, com classes abstratas e concretas, sendo assim para usá-la cria-se classes e métodos concretos que estendem os elementos abstratos. Os estudos de caso apresentados foram implementados utilizando-se a SARA^{MR} e testados com um robô Lego do *kit Mindstorm NXT*.

2. Arquiteturas de Referência

Arquitetura de referência é uma estrutura que fornece a caracterização das funcionalidades de um sistema a partir de um domínio específico [Eickelmann et al. 1996]. Dessa forma, uma arquitetura de referência serve como um guia para aumentar as chances de um desenvolvimento com sucesso de um sistema e pode ser considerado o primeiro passo essencial para o desenvolvimento de frameworks de aplicação.

De acordo com Eickelmann e Richardson [Eickelmann et al. 1996], a proposta de uma arquitetura de referência para sistemas de um domínio específico não é uma tarefa

trivial, pois, requer um profundo conhecimento sobre o domínio que a arquitetura de referência está sendo criada. Além disso, uma arquitetura de referência permite o reúso do projeto arquitetural de uma área específica [Affonso et al. 2013].

3. A Arquitetura SARA^{MR}

A proposta da arquitetura SARA^{MR} é fazer com que todas as funcionalidades de autoadaptação (monitorar, analisar, planejar, agir) sejam modularizadas e externalizadas para fora da aplicação base do robô. Além disso, também é a criação de uma camada de integração que forneça propriedades de baixo acoplamento entre o sistema controlado e o sistema controlador. Dessa forma, os módulos de *loop* de controle podem ser implementados, incluídos ou excluídos com menor impacto de manutenção. Além disso, a arquitetura propõe uma abstração para modularizar quatro funções básicas de um robô: (i) comportamento; (ii) representação do ambiente; (iii) implementação de sensores e (iv) atuadores. Com isso, essas funcionalidades também podem ser incluídas ou excluídas com menor impacto de manutenção. A Figura 1 apresenta a proposta da arquitetura mencionada.

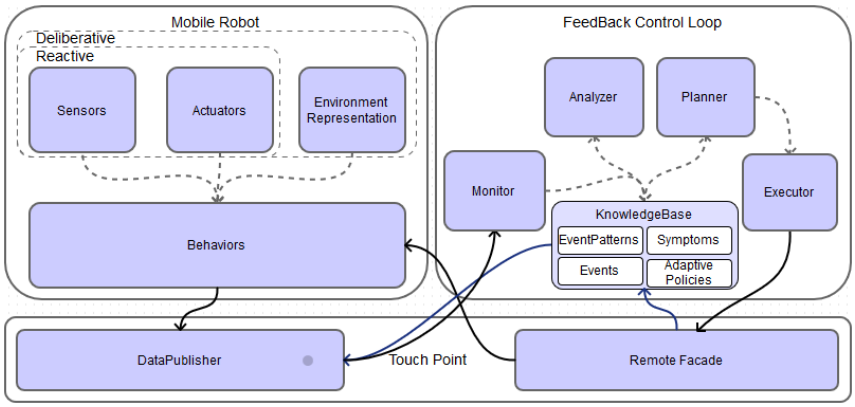

Figura 1. Diagrama de Blocks da arquitetura SARA^{MR}

A arquitetura de referência SARA^{MR} foi implementada em Java na forma de um *framework* caixa branca, possuindo classes e métodos abstratos. É apresentado na Figura 2 a arquitetura de referência criada que é composta por três módulos principais: i) *Mobile Robot Module* (`mobileRobot`) - que representa o sistema gerenciado, isto é, a aplicação base do robô; ii) *Feedback Control Loop Module* (`feedbackControlLoop`) - que representa o sistema gerenciador, o qual agrupa os componentes e as atividades de autoadaptação; e iii) *Touch Point Module* (`touchPoint`) - que representa a interface de comunicação entre os dois módulos citados anteriormente. Na parte superior da Figura 2 encontram-se os principais componentes da arquitetura de referência representados pelo pacote `feedbackControlLoop` e pelo `touchPoint`. Já na parte inferior da figura são representadas as classes concretas da aplicação desenvolvida como estudo de caso (`inDoorMonitoringBiding`, `inDoorMonitoring`). Sendo assim, para desenvolver uma nova aplicação seguindo essa arquitetura é necessário a criação de classes e métodos concretos, como é representado na figura, mais especificamente no pacote `inDoorMonitoring`. Nessa figura algumas partes estão representadas apenas por pacotes como `mobileRobot` e `inDoorMonitoringBinding`. Esses pacotes não serão detalhados pois o enfoque deste artigo e apenas no módulo de autoadaptação. A estrutura de classes da aplicação é apresentada com maiores detalhes na Seção 4.

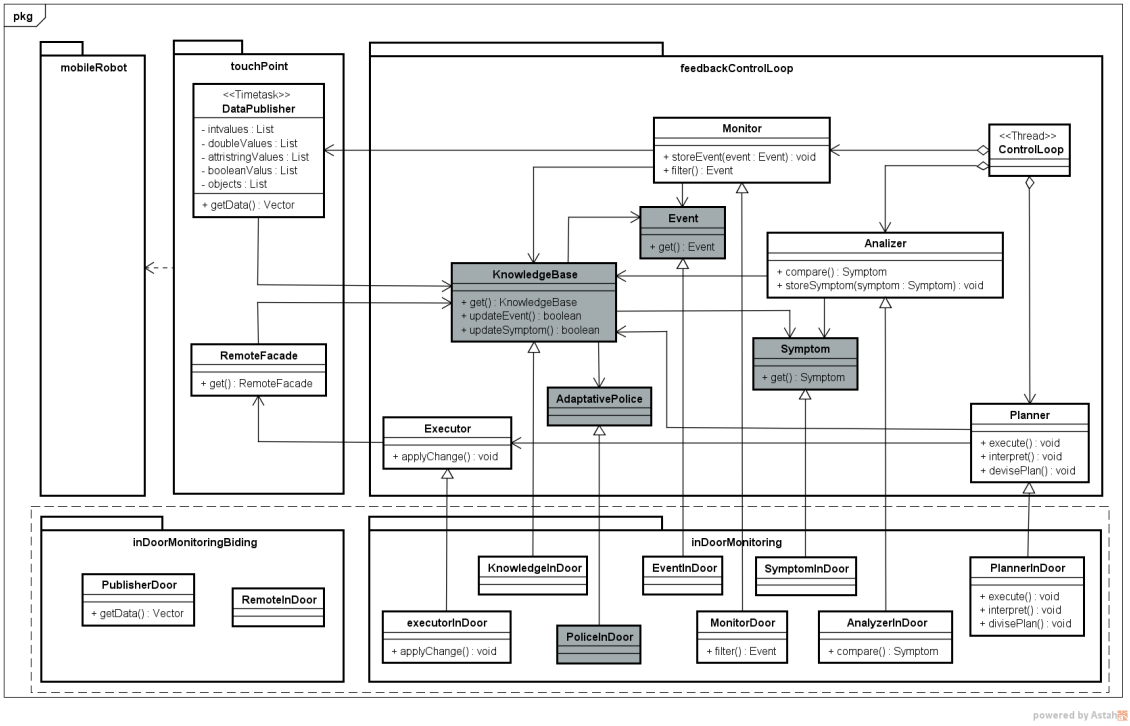

Figura 2. Arquitetura Implementada e Instanciação de uma Aplicação

O pacote `feedbackControlLoop` representa o sistema gerenciador de acordo com o modelo apresentado por Kephart e Chess [Kephart and Chess 2003]. Um ponto importante desse módulo é a presença de classes para modularizar os principais elementos de um *loop* de controle. Dessa maneira, manutenções que envolvem esses elementos podem ser realizadas de forma mais concentrada e controlada que em sistemas nos quais o código-fonte desses elementos encontra-se espalhado pelas classes do sistema. A seguir, seguem simples explicações sobre as classes desse pacote:

- `Monitor`: Detecta o processo gerenciado, filtra os dados coletados e armazena eventos relevantes na base de conhecimento para referencia futura.
- `Analizer`: compara dados de eventos para diagnosticar sintomas e armazena os sintomas na base de conhecimento para referencia futura.
- `Planner`: interpreta os dados analisados e elabora um plano de ação para mudar o processo do subsistema gerenciado.
- `Executor`: aplica esse plano por meio dos atuadores.
- `KnowledgeBase`: a base de conhecimento armazena os dados compartilhados entre os elementos do *loop* de controle.
- `Event`: pode ser um valor ou conjunto de valores recolhidos a partir dos sensores e atuadores que poderiam indicar um estado particular do sistema.
- `Symptom`: é a interpretação resultante da comparação entre um evento do sistema contra um evento padrão. Para cada problema há um conjunto de políticas de adaptação estabelecidas em conformidade com os objetivos do sistema.
- `AdaptivePolicies`: algoritmos reutilizáveis ou blocos de comandos que podem ser executados pelos atuadores, alterando o comportamento do robô.
- `ControlLoop`: é uma *Thread* que coordena a execução do fluxo de dados entre as classes do *loop* de controle.

O pacote `touchPoint` representa uma camada para facilitar a integração e fornece a propriedade de baixo acoplamento entre os módulos que estão sendo integrados. Esse pacote minimiza os efeitos de alterações efetuadas causando pouco ou nenhum impacto sobre outro modulo que seja integrado. A estrutura interna deste pacote é composta por duas classes. A classe `DataPublisher` é o elemento que tem acesso a todos os dados utilizados no comportamento do robô. Ou seja, os dados coletados dos sensores e atuadores são armazenados em sua estrutura de dados interna. E a classe `RemoteFacade` é quem possui acesso aos comandos disponíveis nos comportamentos do robô e da base de conhecimento de um *loop* de controle. O *RemoteFacade* deve ser implementado como o padrão *Facade*, simplificando a utilização dos comandos de comportamentos do robô.

A exemplificação do funcionamento de um SAA aderente a SARA^{MR} ocorre da seguinte forma. De acordo com uma taxa pré-determinada de tempo, os valores obtidos dos sensores e atuadores são atualizados nos `DataPublishers`. Os monitores registram esses valores na base de dados traduzindo-os em eventos, por meio dos métodos `storeEvent()` e `filter()`. Os analisadores inferem sintomas e os registra na base de dados por meio dos métodos `storeSymptom()` e `compare()`. Os planejadores interpretam os sintomas com o método `interpret()`, e preparam um plano de correção. Para isso o método `devisePlan()` realiza uma busca na base de dados pela melhor política de adaptação. Os executores recebem o plano e aplicam a correção nos comportamentos do robô, por meio do padrão *Facade*.

4. Estudo de Caso e Avaliação Preliminar

Como estudo de caso, foi desenvolvida uma aplicação com apoio da arquitetura de referência SARA^{MR} com o objetivo de demonstrar as propriedades de facilidade de evolução e manutenção. A aplicação é um robô de monitoramento de ambientes fechados cuja estratégia é um comportamento que o faz seguir em frente mantendo uma distancia de 20cm da parede. O mecanismo de ajuste de distância é implementado na forma de um *loop* de controle e utiliza um algoritmo de PID como politica de adaptação. O sistema autoadaptativo submete os valores coletados dos sensores ao processo de monitoramento e análise, em seguira um plano de correção é efetuado com base no valor de retorno do algoritmo, finalmente o plano é traduzido em ação por meio de comandos aplicados nos atuadores. Assim, o *loop* de controle atua como um piloto automático sobre o comportamento do robô fazendo os ajustes necessários para que ele continue mantendo a distância desejada.

Entretanto, esse *loop* de controle não ofereceu um bom desempenho, pois os parâmetros de ajuste do algoritmo de PID (Kp, Ki e Kd), são fixos e o robô o acaba fazendo "zig zags", comprometendo o tempo de deslocamento. Dessa forma, optou-se pela estratégia de inclusão de um segundo *loop* de controle atuando com ajustes nos parâmetros do algoritmo do primeiro *loop*. O segundo *loop* também possui o ciclo completo de autoadaptação, com as funções de monitoramento, análise, planejamento e ação.

A abordagem escolhida para a avaliação da arquitetura se baseia na demonstração de atividades de manutenção no código necessárias para evolução da aplicação robótica. Simulou-se uma atividade de manutenção, que foi a inclusão de um novo *loop* de controle, conforme Tabela 1. A motivação foi a percepção de que o primeiro loop não estava sendo suficiente para fazer com que o robô obtivesse um bom desempenho. Assim, percebeu-se a necessidade de um outro loop de controle que atuasse como um sistema gerenciador

sobre o primeiro *loop*, o qual passou a fazer o papel de sistema gerenciado.

Tabela 1. Atividades de Manutenção na Aplicação 1

Evolução	Descrição da Necessidade de Evolução	Atividade	Qtd
Inclusão de um novo *loop* de controle	Em consequência do mau desempenho do robô, opto-se por incluir um novo *loop* de controle para melhorar o desempenho.	Criação de Classes	10
		Implementação de Métodos	10
		Declaração de Atributos	7

As atividades de manutenção demonstraram que a arquitetura SARA^{MR} cumpre a função de guia e orienta o mantenedor do código a identificar facilmente os pontos de alteração. Tanto nas tarefas de inclusão ou exclusão de *loop* de controle como na implementação do ciclo de autoadaptação. Todos esses pontos estão referenciados na arquitetura. A estratégia de utilização de classes espelho e políticas de adaptação facilitou no cenário de exclusão de sensores. No caso de aplicações que não utilizam a arquitetura SARA^{MR} ou/e que não levam em conta a estratégias de classes espelho, bem como separação do módulo de *loop* de controle, as atividades de manutenção podem ser bem mais complexas e custosas.

Com o objetivo de avaliar o esforço gasto com as atividades de manutenção evolutiva, realizou-se uma avaliação do impacto que as alterações de códigos efetuadas em um determinado componente podem causar aos outros componentes da arquitetura. O termo impacto aqui se refere ao fato de que uma determinada alteração em um componente pode disparar necessidades de alterações em outros componentes.

Dessa forma, foi elaborada uma matriz de relacionamentos, conforme é mostrada na Tabela 2, contrastando as atividades de manutenção com o impacto de alterações que eventualmente possam se propagar pela arquitetura. Na primeira linha estão os módulos e as atividades de manutenção (Inclusão, Alteração, Exclusão). Na primeira coluna são apresentados os módulos que recebem algum impacto. Nas intersecções são mostradas as letras N (Não) para assinalar que não há impacto e a letra S (Sim) para assinalar que há impacto entre atividade e módulo. Conforme é mostrado na Tabela 2, as atividades de inclusão e alteração de sensores não causam impacto de manutenção em outros módulos da aplicação. Já a atividade de exclusão de sensores pode ocasionar impactos nos módulos de Comportamentos e *Loops* de Controle. As atividades de inclusão, alteração ou exclusão de *Loops* de Controle não causam impactos nos outros módulos da aplicação.

Nas duas colunas da direita são indicadas as quantidades que apontam uma relação entre o número total de atividades de manutenção e o total de impactos causados. Observa-se que do total de 60 atividades de manutenção, apenas 10 causam impactos de manutenção, ou seja, um total de 17%.

5. Trabalhos Relacionados

A arquitetura 4D/RCS [Albus 2000] fornece um modelo de referência para veículos militares não tripulados. É uma arquitetura híbrida com capacidade de planejamento e ação em tempo real para responder e reagir aos estímulos do ambiente. Cada camada possui vários *loops* de controle que são chamados de nós computacionais. Apenas os níveis mais baixos foram totalmente implementados. Os elementos são organizados na estrutura de um *feedback loop*. Onde cada nó funciona como um *loop* de controle, lendo dados dos sensores e enviando comandos aos atuadores.

Tabela 2. Matriz de relacionamentos (Manutenção x Impacto)

		Manutenção (Inclusão, Alteração, Exclusão)															Total de Impactos	
		Sensores			Atuadores			Comportamentos			Ambiente			*Loops* de Controle				
		I	A	E	I	A	E	I	A	E	I	A	E	I	A	E	Sim	Não
Impacto	Sensores				N	N	N	N	N	N	N	N	N	N	N	N	0	12
	Atuadores	N	N	N				N	N	N	N	N	N	N	N	N	0	12
	Comportamentos	N	N	S	N	N	S				N	S	S	N	N	N	4	8
	Ambiente	N	N	N	N	N	N	N	N	N				N	N	N	0	12
	Loops de Controle	N	N	S	N	N	S	N	S	S	N	S	S				6	6
Totais =																	**10**	**50**
																	17%	**83%**

A RA4SaS [Affonso et al. 2013] e uma arquitetura de referência baseada no recurso de reflexão para inspeção e modificação de entidades de software em tempo de execução. Apesar de não ser uma arquitetura para o domínio de RMAs, essa proposta é interessante para o contexto deste trabalho, pois essa arquitetura utiliza *loops* de controle para realizar a autoadaptação. Outro ponto importante é que o contexto de adaptação da RA4SaS ocorre diretamente na estrutura das entidades de software diferenciando do método de adaptação apresentado na proposta deste trabalho, criando uma perspectiva interessante de comparação.

6. Conclusão

Neste artigo foi apresentada uma arquitetura de referência para apoiar construção e manutenção de software autoadaptativo para robôs móveis autônomos. Com o apoio da arquitetura SARA^{MR}, conforme é demonstrado no decorrer do trabalho as atividades de manutenção evolutiva são facilitadas de acordo com os aspectos a seguir:

- Inclusão ou alteração de sensores e atuadores não geram impactos de manutenção nos outros módulos da aplicação.
- Os comportamentos são blocos individuais de código e podem ser adicionados sem gerar impacto. No caso de alteração e remoção os impactos são mínimos, podendo ocorrer apenas no módulo de *loop*.
- A representação de elementos no ambiente é efetuada por meio de informações métricas, facilitando as atividades de manutenção.
- Inclusão de novos elementos não geram impactos de manutenção. Alteração ou exclusão de elementos podem gerar impactos porem apenas nos módulos de comportamentos e *loops* de controle.
- Inclusão, alteração ou exclusão de *loops* de controle não geram impactos na aplicação, as funcionalidades de autoadaptação são separadas da aplicação base do RMA.

Embora a atual versão da arquitetura SARA^{MR} esteja implementada em um *framework* Java e somente tenha sido testada em um robô Lego. Acredita-se que as abstrações oferecidas na estrutura da arquitetura possam servir de guia para a implementação em qualquer outra linguagem orientada a objetos. Sua implementação em C++, por exemplo, poderia propiciar seu uso em outros tipos de robôs. Os resultados obtidos demonstram que a arquitetura de referência minimiza as atividades de manutenção produzidas por atividades de evolução das aplicações robóticas autoadaptativas. A arquitetura de referencia fornece uma padronização de módulos para as principais funcionalidades do domínio de RMAs. Fornece também uma estrutura com diretrizes, técnicas e padrões para melhorar as atividades ligadas a construção e manutenção evolutiva no software de RMAs.

7. Agradecimentos

O presente trabalho possui relação direta com o processo Fapesp 2012/00494-0

Referências

Affonso, F. J. et al. (2013). A reference architecture based on reflection for self-adaptive software. In *Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium on*, pages 129–138.

Albus, J. S. (2000). 4-d/rcs reference model architecture for unmanned ground vehicles. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 4, pages 3260–3265 vol.4.

Baker, C. R. et al. (2011). Toward adaptation and reuse of advanced robotic software. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 6071–6077.

Brugali, D. (2007). Software abstractions for modeling robot mechanisms. In *2007 IEEE/ASME international conference on advanced intelligent mechatronics*, pages 1–6.

Brun, Y. et al. (2009). *Software Engineering for Self-Adaptive Systems*, chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2005). *Self-star Properties in Complex Information Systems*, chapter Making Self-adaptation an Engineering Reality, pages 158–173. Springer-Verlag, Berlin, Heidelberg.

Edwards, G. et al. (2009). Architecture-driven self-adaptation and self-management in robotics systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 142–151.

Eickelmann, N. S. et al. (1996). An evaluation of software test environment architectures. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 353–364.

Garlan, D. et al. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.

Georgas, J. C. and Taylor, R. N. (2008). Policy-based self-adaptive architectures: A feasibility study in the robotics domain. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 105–112, New York, NY, USA. ACM.

IBM (1994). *Autonomic computing white paper: An architectural blueprint for autonomic computing*. IBM White Paper.

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

Kokar, M. M., Baclawski, K., and Eracar, Y. A. (1999). Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45.

Weyns, D. et al. (2013). *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, pages 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg.