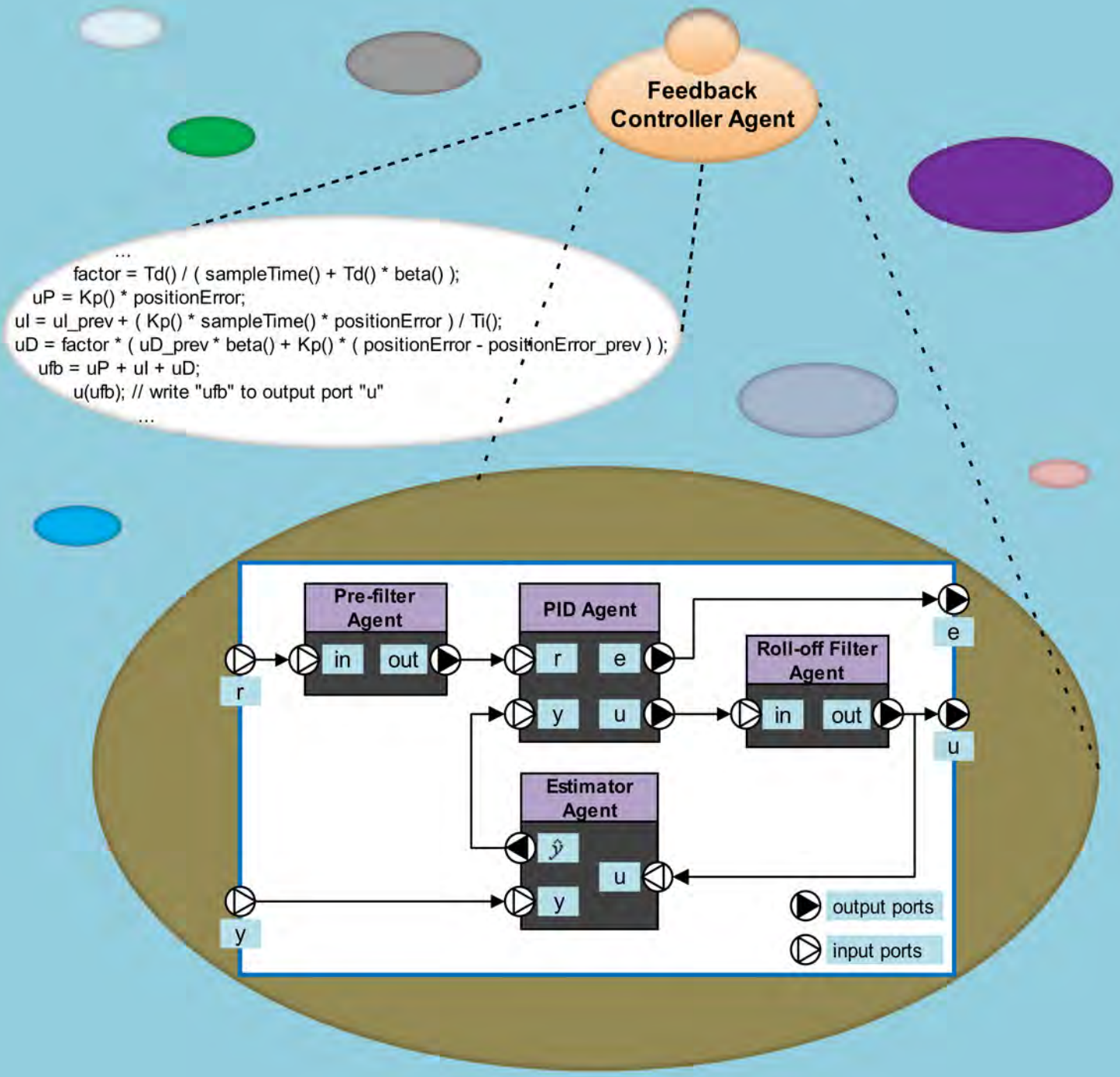
Safe-Guarded Multi-Agent Control for Mechatronic Systems

Implementation Framework and Design Patterns



Dao Ba Phong

Safe-Guarded Multi-Agent Control for Mechatronic Systems

Implementation Framework and Design Patterns

Ph.D. thesis of Dao Ba Phong

Graduation committee:

Chairman:	prof.dr.ir. A.J. Mouthaan	University of Twente
Secretary:	prof.dr.ir. A.J. Mouthaan	University of Twente
Promotor:	prof.dr.ir. J. van Amerongen	University of Twente
Asst. promotor:	dr.ir. T.J.A. de Vries	University of Twente
Members:	prof.dr.ir. M. Akşit	University of Twente
	prof.dr.ir. F.J.A.M. van Houten	University of Twente
	prof.dr.ir. P.P. Jonker	Delft University of Technology
	dr.ir. M.J.G. van de Molengraft	University of Eindhoven

This research was financially supported by the Vietnamese Government through the 322 Project for four years (5/2006 - 4/2010) and then by Imotec B.V. for eight months (6/2010 - 1/2011) and carried out at the Control Engineering group at the University of Twente, Enschede, The Netherlands.

ISBN: 978-90-365-3148-1

The front cover picture of this thesis is about a polymorphic feedback controller agent. The back one is about the System Agent design pattern.

Printed by PrintPartners Ipskamp, Enschede, The Netherlands.

SAFE-GUARDED MULTI-AGENT CONTROL FOR MECHATRONIC SYSTEMS

IMPLEMENTATION FRAMEWORK AND DESIGN PATTERNS

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
Prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, 3rd of February 2011 at 14:45

by

Dao Ba Phong
born on 11th of September 1978
in Haiduong, Vietnam

This dissertation is approved by:
prof.dr.ir. J. van Amerongen, promotor
dr.ir. T.J.A. de Vries, assistant promotor

Summary

This thesis addresses two issues: (i) developing an implementation framework for Multi-Agent Control Systems (MACS); and (ii) developing a pattern-based safe-guarded MACS design method.

The Multi-Agent Controller Implementation Framework (MACIF), developed by Van Breemen (2001), is selected as the starting point because of its capability to produce MACS for solving complex control problems with two useful features:

- MACS is *hierarchically structured* in terms of a coordinated group of elementary and/or composite controller-agents;
- MACS has an *open architecture* such that controller-agents can be added, modified or removed without redesigning and/or reprogramming the remaining part of the MACS.

However, this framework still had some shortcomings that give room for improvement. An enhancement scheme has been realized: developing a new implementation framework for MACS that inherits and improves the advantages of the MACIF and simultaneously provides the missing features for the MACIF. Through evaluating four possible approaches, that can be applied to develop real-time MACS using concepts and operation mechanisms of the MACIF, the solution using the OROCOS framework (Orocos, 2009a) has been selected for developing a new implementation framework for MACS. After studying the resemblance between the MACIF and the OROCOS framework, a functional combination of the two frameworks has been realized. As a result, we obtain an OROCOS-based Implementation Framework for MACS (OROMACS framework), which supports the development of *multi-threaded MACS* with deterministic *real-time* control behavior, *thread-safe* real-time inter-process communication mechanism, and the capability of handling events. The way of integration used in this combination results in a low coupling between these frameworks. Hence, change of the OROCOS framework will not require much modification of the MACS developed by using the OROMACS framework.

In addition, the port-based polymorphic modeling approach (De Vries, 1994) has been brought to the OROMACS framework. Polymorphic modeling is the division of a subsystem description into a subsystem type and a subsystem specification, and the expression of a subsystem type in terms of one or more designated other types. This approach has been applied to the OROMACS framework in such a way that a controller-

agent with a particular Type can be implemented in the form of different Elementary and/or Composite Specifications. This "one Type with multiple Specifications" approach makes the controller-agent and MACS polymorphic. This property, called *polymorphism*, opens the possibility to create libraries of structures for which the detailed implementation is unspecified. As a result, with a sufficiently rich library of multiple specifications, the design and programming of a control system becomes a matter of *configuration and composition of controller-agents*. Moreover, the OROMACS framework allows designers to decide beforehand a desired control strategy by selecting suitable coordinators.

Although the OROMACS framework brings with it the improvements, it still faces two shortcomings: (i) the trade-off between the desire to achieve a MACS design with good control performances and a short development time; and (ii) the lack of support for reusability of design results from previous projects into new projects. These shortcomings are tackled by using a combination of the OROMACS framework with the pattern-based design method, which results in *a pattern-based safe-guarded MACS design method*. This design method is demonstrated by means of two case studies.

First, we design a safe-guarded MACS for the DemoLin setup, a simple single-axis electro-mechanical motion system with the dominant compliance in the transmission. The design is required to meet three particular requirements (multi-operation modes, good control performances, and safe-guarded control equiped with capabilities: error detection, error handling, graceful degradation, and error recovery). Based on this design, we have formulated *a generalized safe-guarded control solution for simple mechatronic systems*, i.e. motion systems with one degree-of-freedom (1-DoF).

Next, we design a safe-guarded MACS for the TriPod setup, a complex three-axis electro-mechanical motion system. This design reuses the design results of the DemoLin setup. This reusability is proven through reusing two parts of the MACS design: the operation control and the safe-guarded control. The only thing that remains to be done is to modify application-specific settings (e.g. trajectory, controller parameters, coordinators, etc.). Based on this design, we have formulated *a generalized safe-guarded control solution for complex mechatronic systems*, i.e. motion systems with multiple degrees-of-freedom (n-DoF). The design method makes the design and programming of real-time safe-guarded MACS become *a matter of configuration and composition of the whole design*. This is done through the application of proper design patterns and selection of suitable specifications for controller-agents to quickly build up a complete MACS. As a result, the *short time-to-market objective* with regard to the control system development can be obtained.

This thesis has developed control system design patterns in which the Safe-Guarded Agent is one of main design patterns. This design pattern can flexibly handle faults and particularly fault propagations that may happen in n-DoF motion systems. Specifically, the Safe-Guarded Agent deals with two possibilities of fault propagations: (i) the propagations of influence spheres of faults, i.e. from faults occurring on a single axis to faults involving multiple axes; and (ii) the propagations of criticality levels of faults, i.e. from warning to serious, from serious to dangerous, and from warning to dangerous.

Samenvatting

Dit proefschrift behandelt twee onderwerpen: (i) de ontwikkeling van een implementatie-raamwerk voor regelsystemen die zijn opgebouwd uit meerdere "agenten" (Engels: Multi-Agent Control Systems, MACS); en (ii) de ontwikkeling van een patroon-gebaseerde, veiligheid-bewakende MACS ontwerpmethode.

Het Multi-Agent Controller Implementation Framework (MACIF), dat is ontwikkeld door Van Breemen (2001), is gekozen als startpunt vanwege het vermogen van dit raamwerk om MACS voor complexe regelproblemen te bouwen die twee nuttige kenmerken hebben:

- een MACS is *hiërarchisch georganiseerd* in termen van gecoördineerde groepen van elementaire en/of samengestelde regelaar-agenten;
- een MACS heeft een *open architectuur*, waardoor regelaar-agenten kunnen worden toegevoegd, veranderd of verwijderd zonder herontwerp en/of herprogrammering van het overblijvende deel van de MACS.

Echter, het MACIF raamwerk bezit ook enkele tekortkomingen die ruimte bieden voor vooruitgang. Een verbeterplan is opgesteld en gerealiseerd: de ontwikkeling van een nieuw implementatie-raamwerk dat de voordelen van MACIF overerft en uitbouwt en tegelijkertijd ontbrekende eigenschappen toevoegt. Via evaluatie van vier mogelijke werkwijzen die kunnen worden ingezet voor het ontwerp van rekentijd-begrensde MACS is de oplossing waarbij gebruik wordt gemaakt van het OROCOS raamwerk (Orocos, 2009a) uitgekozen voor de ontwikkeling van een nieuw implementatie-raamwerk voor MACS. Na bestudering van de overeenkomsten en verschillen tussen MACIF en het OROCOS raamwerk is een functionele combinatie van de twee raamwerken gerealiseerd. Als gevolg hiervan is verkregen een OROCOS-gebaseerd Implementatie-Raamwerk voor MACS (OROMACS), dat de ontwikkeling ondersteunt van in meerdere threads executerende MACS met deterministisch tijd-begrensd regelgedrag, met mechanismen die tijd-begrensde en beschermde communicatie tussen processen in verschillende threads toestaan en met het vermogen om events af te handelen. De manier van integratie die is gekozen zorgt ervoor dat de koppeling tussen de raamwerken laag is. Veranderingen in het OROCOS raamwerk zullen daardoor weinig aanpassingen vergen van de MACS die zijn ontwikkeld met OROMACS.

Bovendien is de polymorfe modelvormings-aanpak (De Vries, 1994) aan het OROMACS raamwerk toegevoegd. De polymorfe modelvormings-aanpak houdt in dat de beschrijving

van een subsysteem wordt opgedeeld in een subsysteem-type en een subsysteem-specificatie en dat het subsysteem-type wordt uitgedrukt in termen van één of meer aangewezen andere types. Deze aanpak is zodanig toegepast in het OROMACS raamwerk dat een regelaar-agent met een bepaald type kan worden geïmplementeerd door middel van verschillende elementaire en/of samengestelde specificaties. Deze "één type met meerdere specificaties" aanpak zorgt ervoor dat een regelaar-agent, en dus een MACS, 'veelvormig' worden. Deze eigenschap, die *polymorfisme* wordt genoemd, maakt het mogelijk bibliotheken met structuren te maken waarvan de gedetailleerde implementatie nog niet is vastgelegd. Wanneer een voldoende rijk gevulde bibliotheek beschikbaar is, heeft dit tot gevolg dat het ontwerp en de programmering van een regelsysteem een zaak wordt van *samenstelling en configuratie van regelaar-agenten*. Bovendien maakt het OROMACS raamwerk het voor de ontwerper mogelijk om vooraf de gewenste regelstrategie te beïnvloeden door gebruik te maken van verschillende coördinatie-mechanismen.

Hoewel het OROMACS raamwerk verbeteringen met zich meebrengt, zijn er nog twee tekortkomingen: (i) de afweging tussen de wens om een MACS ontwerp met een goede prestatie te bereiken versus een korte ontwikkeltijd; en (ii) het gebrek aan ondersteuning voor hergebruik van ontwerpresultaten uit vorige projecten in nieuwe projecten. Deze tekortkomingen zijn onderhanden genomen door het OROMACS raamwerk te combineren met de patroon-gebaseerde ontwerpmethode, wat heeft geresulteerd in een patroon-gebaseerde, veiligheid-bewakende MACS ontwerpmethode. Deze ontwerpmethode is gedemonstreerd door middel van twee casussen.

Als eerste is een veiligheid-bewakende MACS ontworpen voor de DemoLin opstelling, een eenvoudig, enkel-assig elektro-mechanisch bewegings-systeem met de dominante compliantie in de overbrenging. Het ontwerp dient te voldoen aan drie vereisten (meerdere modi van operatie, goede regelprestaties, en veiligheid-bewakende regeling met als kwaliteiten: fout-herkenning, fout-afhandeling, geleidelijke vermindering van werking, en herstel na foutsituaties). Op basis van dit ontwerp is een algemene oplossing geformuleerd voor *veiligheid-bewakende regeling van eenvoudige mechatronische systemen*, dat wil zeggen, bewegings-systemen met één graad van vrijheid (1-DoF).

Vervolgens is een veiligheid-bewakende MACS ontworpen voor de TriPod opstelling, een complex, drie-assig elektro-mechanisch bewegings-systeem. Dit ontwerp herbruikt het ontwerpresultaat van de DemoLin opstelling. Dit hergebruik is bewezen door twee delen van het MACS ontwerp te hergebruiken: de operationele regeling en de veiligheid-bewakende regeling. Het enige dat nog overblijft om te doen is het aanpassen van de applicatie-specifieke instellingen (bijvoorbeeld te volgen pad, regelaar-parameters, coördinatoren, enzovoort). Op basis van dit ontwerp is een algemene oplossing geformuleerd voor *veiligheid-bewakende regeling van complexe mechatronische systemen*, dat wil zeggen, bewegings-systemen met meerdere graden van vrijheid (n-DoF). De MACS ontwerpmethode zorgt ervoor dat het ontwerp en de programmering van tijd-begrensde veiligheid-bewakende regelaars een zaak wordt van configuratie en samenstelling van het gehele ontwerp. Dit wordt gedaan door het toepassen van goede ontwerp-patronen en de selectie van geschikte specificaties voor regelaar-agenten om zo

snel een complete MACS te bouwen. Het gevolg is, dat met betrekking tot de ontwikkeling van regelsystemen *een korte tijd-tot-vermarkting* kan worden verkregen.

In dit proefschrift zijn ontwerp-patronen voor regelsystemen ontwikkeld, waarbij de veiligheid-bewakende agent één van de belangrijkste ontwerp-patronen is. Dit patroon is flexibel in de manier van fout-afhandeling, in het bijzonder ook in fout-propagaties die voorkomen in n-DoF bewegings-systemen. De veiligheid-bewakende agent handelt met name twee soorten fout-propagatie af: (i) de propagatie van de invloedssfeer van fouten, dat wil zeggen, van fouten die ontstaan in één as naar fouten die betrekking hebben op meerdere assen; en (ii) de propagatie van het gevaar-niveau van fouten, dat wil zeggen, van waarschuwing naar ernstig, en van ernstig naar gevaarlijk.

Tóm tắt

Luận văn này đề cập đến hai vấn đề: (i) phát triển một công cụ thực thi các hệ thống điều khiển đa agent, Multi-Agent Control Systems (MACS); và (ii) phát triển một phương pháp thiết kế MACS bảo đảm an toàn (safe-guarded MACS) dựa trên các mẫu thiết kế (design patterns).

Công cụ thực thi bộ điều khiển đa agent, Multi-Agent Controller Implementation Framework (MACIF), được phát triển bởi Van Breemen (2001), đã được lựa chọn làm điểm khởi đầu của nghiên cứu bởi vì MACIF có khả năng tạo ra MACS để giải quyết các vấn đề điều khiển phức tạp với hai ưu điểm:

- thứ nhất là, MACS được hình thành dưới dạng *cấu trúc có thứ bậc* của một nhóm phối hợp của các bộ điều khiển agent cơ sở (elementary controller-agents) và phức hợp (composite controller-agents);
- thứ hai là, MACS được tạo ra với một *kiến trúc mở* vì thế các bộ điều khiển agent (controller-agents) có thể được bổ sung thêm, chỉnh sửa hoặc loại bỏ khỏi hệ thống mà không phải thiết kế hay lập trình lại phần còn lại của MACS.

Tuy nhiên, MACIF còn có một vài điểm chưa hoàn thiện; vì vậy, một kế hoạch cải tiến MACIF đã được thực hiện: phát triển mới một công cụ thực thi MACS với quan điểm thừa kế và nâng cao những điểm mạnh vốn có của MACIF và đồng thời bổ sung những đặc tính còn thiếu hay chưa hoàn thiện cho MACIF. Thông qua việc đánh giá bốn giải pháp khả thi có thể được áp dụng để phát triển MACS thời gian thực (real-time MACS) với ràng buộc sử dụng các khái niệm và cơ chế hoạt động của MACIF, giải pháp sử dụng OROCOS framework (Orocos, 2009a) đã được lựa chọn làm cơ sở để phát triển một công cụ thực thi mới cho MACS. Sau khi nghiên cứu sự tương đồng giữa MACIF và OROCOS, một giải pháp kết hợp chức năng giữa hai công cụ (frameworks) này đã được thực hiện. Kết quả là, nghiên cứu đã tạo ra một công cụ thực thi mới dựa trên OROCOS cho MACS (OROCOS-based Implementation Framework for MACS). Chúng tôi gọi công cụ thực thi mới này là OROMACS. Công cụ thực thi OROMACS cho phép người thiết kế phát triển MACS đa nhiệm (multi-threaded MACS) với đặc tính điều khiển thời gian thực tiền định (deterministic real-time control behavior), cơ chế giao tiếp liên quá trình bảo đảm tính thời gian thực và an toàn dữ liệu trong khi truy xuất (thread-safe real-time inter-process communication mechanism) và khả năng tương tác với các sự kiện rời rạc (the capability of handling events). Phương thức tích hợp mà chúng tôi sử dụng trong quá trình kết hợp giữa MACIF và OROCOS không tạo ra nhiều sự ràng buộc giữa hai công cụ (frameworks) này.

Vì vậy, sự thay đổi của OROCOS sẽ không gây ra nhiều ảnh hưởng đối với các hệ thống MACS thời gian thực được phát triển bằng công cụ thực thi OROMACS.

Công việc tiếp theo mà nghiên cứu đã triển khai đó là áp dụng lý thuyết mô hình hóa đa hình thái dựa trên cổng, the port-based polymorphic modeling approach (De Vries, 1994), để nâng cao tính năng cho công cụ thực thi OROMACS. Mô hình hóa đa hình thái là ứng dụng kết hợp giữa mô đun hóa với định hình kiểu (subtyping) trong quá trình xây dựng mô hình, nghĩa là, mô tả của một hệ thống sẽ được phân tách thành mô tả kiểu hệ thống (subsystem type) và đặc tả hệ thống (subsystem specification), trong đó sự mô tả kiểu hệ thống được thể hiện dưới dạng của một hoặc nhiều kiểu hệ thống đã biết khác. Lý thuyết mô hình hóa đa hình thái này đã được triển khai vào OROMACS theo cách thức như sau: một bộ điều khiển agent với một kiểu cụ thể có thể được thực thi dưới dạng các đặc tả cơ sở (Elementary Specifications) và/hoặc các đặc tả phức hợp (Composite Specifications) khác nhau. Phương pháp "một kiểu hệ thống với nhiều dạng đặc tả" (one Type with multiple Specifications) này đã tạo ra các bộ điều khiển agent đa hình thái (polymorphic controller-agents) và từ đó tạo thành các hệ thống MACS đa hình thái (polymorphic MACS). Tính chất này được gọi là *hiện tượng đa hình thái* (polymorphism), là cơ sở để hình thành các thư viện cấu trúc điều khiển với đặc điểm là không cần phải cụ thể hóa việc thực thi tại thời điểm thiết kế. Thay vào đó, người sử dụng các thư viện này sẽ quyết định việc thực thi chi tiết cho phù hợp với mục đích và ứng dụng của họ. Kết quả là, với một thư viện phong phú của các dạng đặc tả (multiple specifications), việc thiết kế và lập trình một hệ thống điều khiển sẽ trở thành vấn đề *cấu hình và soạn thảo các bộ điều khiển agent*. Điều này có nghĩa là hiện tượng đa hình thái (polymorphism) cung cấp cho người sử dụng các lựa chọn cấu hình (configuration options) và vì vậy nó chính là phương tiện để biến khả năng phát triển hệ thống điều khiển dựa trên cấu hình và soạn thảo trở thành hiện thực. Ngoài ra, công cụ thực thi OROMACS còn cho phép người thiết kế quyết định trước một chiến lược điều khiển như mong muốn bằng cách lựa chọn các các cơ chế phối hợp (coordination mechanisms) phù hợp.

Mặc dù công cụ thực thi OROMACS đã có nhiều cải tiến đáng kể so với MACIF, tuy nhiên nó vẫn còn tồn tại hai mặt hạn chế trong quá trình sử dụng: một là, sự mâu thuẫn (trade-off) giữa mong muốn đạt được một thiết kế MACS với chất lượng điều khiển tốt và thời gian phát triển ngắn. Hai là, thiếu sự hỗ trợ cần thiết để đạt được khả năng tái sử dụng các kết quả thiết kế của các dự án đã hoàn thành sang các dự án mới. Tuy nhiên, những điểm yếu này đã được chúng tôi khắc phục bằng một giải pháp kết hợp công cụ thực thi OROMACS với phương pháp thiết kế dựa trên mẫu (pattern-based design method). Sự kết hợp này đã tạo thành *một phương pháp thiết kế MACS bảo đảm an toàn dựa trên các mẫu thiết kế* (pattern-based safe-guarded MACS design method). Phương pháp thiết kế này được minh họa thông qua hai ví dụ ứng dụng.

Trong ví dụ đầu tiên, chúng tôi đã thiết kế một hệ thống MACS bảo đảm an toàn cho mô hình thực nghiệm DemoLin, một hệ thống chuyển động cơ-điện một trục đơn giản với đặc tính động hoc chủ yếu tập trung tại bộ phận truyền chuyển động. Thiết kế cần đạt được ba yêu cầu cụ thể sau: vận hành với đa chế độ hoạt động, đáp ứng tốt các chỉ tiêu chất lượng điều khiển, và điều khiển bảo đảm an toàn với các khả năng: phát hiện lỗi, xử lý lỗi, duy trì hoạt động hạn chế trong khi xử lý lỗi, và phục hồi hoạt động bình thường sau khi xử lý lỗi.

Dựa trên thiết kế này, chúng tôi đã đưa ra *một giải pháp điều khiển bảo đảm an toàn tổng quát có thể áp dụng cho các hệ thống cơ điện tử đơn giản*, nghĩa là các hệ thống chuyển động với một bậc tự do (1-DoF).

Ở ví dụ tiếp theo, chúng tôi đã thiết kế một hệ thống MACS bảo đảm an toàn cho mô hình thực nghiệm TriPod, một hệ thống chuyển động cơ-điện ba trục phức tạp. Thiết kế này đã tái sử dụng các kết quả thiết kế mà chúng tôi đã thực hiện cho mô hình DemoLin. Chức năng tái sử dụng này được minh chứng thông qua việc tái sử dụng hai phần của thiết kế MACS: phần điều khiển hoạt động (operation control) và phần điều khiển bảo đảm an toàn (safe-guarded control). Điều duy nhất mà người thiết kế cần phải thực hiện đó là thay đổi các thông số cấu hình phụ thuộc từng ứng dụng cụ thể (ví dụ như quỹ đạo điều khiển mong muốn, các thông số bộ điều khiển, kiểu cơ chế phối hợp, etc.). Dựa trên thiết kế này, chúng tôi đã đưa ra *một giải pháp điều khiển bảo đảm an toàn tổng quát có thể áp dụng cho các hệ thống cơ điện tử phức tạp*, nghĩa là các hệ thống chuyển động với đa bậc tự do (n-DoF). Phương pháp thiết kế dựa trên mẫu này đã biến việc thiết kế và lập trình các hệ thống MACS bảo đảm an toàn thời gian thực (real-time safe-guarded MACS) trở thành vấn đề *cấu hình và soạn thảo đối với toàn bộ thiết kế*. Tính năng này được thực hiện thông qua áp dụng các mẫu thiết kế (design patterns) thích hợp và lựa chọn các đặc tả (specifications) phù hợp cho các bộ điều khiển agent để từ đó nhanh chóng tạo thành một thiết kế MACS hoàn chỉnh. Kết quả là, *mục tiêu "short time-to-market"* đối với việc phát triển hệ thống điều khiển có thể đạt được.

Luận văn này đã phát triển các mẫu thiết kế hệ thống điều khiển, trong đó Safe-Guarded Agent là một trong những mẫu thiết kế chính. Mẫu thiết kế này có thể xử lý linh hoạt các lỗi và đặc biệt là sự lan truyền của lỗi có thể xảy ra trong các hệ thống chuyển động đa bậc tự do. Cụ thể là, mẫu thiết kế Safe-Guarded Agent có thể xử lý hai khả năng lan truyền của lỗi: (i) một là, sự lan truyền phạm vi ảnh hưởng của lỗi, nghĩa là từ các lỗi xảy ra trên một trục thành các lỗi liên quan đến đa trục; (ii) hai là, sự lan truyền mức độ nguy hiểm của lỗi, nghĩa là từ mức độ cảnh báo chuyển thành nghiêm trọng, từ mức độ nghiêm trọng chuyển thành nguy hiểm, và từ mức độ cảnh báo chuyển thành nguy hiểm.

Contents

Chapter 1

Introduction

1.1 Mechatronic Design Approach

While solving (complex) control problems with the *traditional design approach*, a subsequent design of domain specific subsystems is typically used that starts with the design of mechanical modules then followed by the design of electrical and electronic parts, and finally the control software components in the form of embedded hardware and software co-design are realized (Van Brussel, 1996; Van Amerongen, 2007). Using this sequential design method, designers face a lot of difficulties. Especially at the end of projects, many problems are left to be solved by the control and software engineers. But when they get involved too late, they are blamed for missing the deadline and for huge development costs (Van Amerongen, 2007).

To overcome this problem, a so-called *mechatronic design approach* is used. A mechatronic design philosophy considers the design of controlled systems as a whole. The design trajectory is based on a multi-disciplinary design methodology that enables concurrent design and interactions between the different engineering disciplines involved: mechanical engineering, control engineering, electrical engineering, (micro)electronics, and computer science (Van Brussel, 1996; Tomizuka, 2002). Isermann (1997) discussed that the integration of mechatronic systems can be performed by the components, i.e. hardware-integration and by information processing, i.e. software-integration. The mechatronic design approach results in degrees of freedom for the designers in all underlying domains; for example, changes in the mechanical design and the controller design can be evaluated

simultaneously. Moreover, solutions in different disciplines are considered or allowed; for example, by evaluating the possibility of a more expensive mechanical construction against a more sophisticated control system (Van Amerongen, 2007). In other words, because the interrelations during the design play an import role, simultaneous engineering from the very beginning has to take place (Isermann, 1997). So, mechatronic design actually is teamwork where specialists from the underlying disciplines participate in the design team must have the ability to look beyond the design problem within their own field, in order to profit from the advantages of a mechatronic design approach (Van Amerongen et al., 2000). Optimized solutions in all underlying disciplines are being sought simultaneously, thus a global optimization design may be achieved. In summary, mechatronics may be interpreted as the best practice for synthesis of engineering systems, and it covers a broad area and scope (Tomizuka, 2002).

Mechatronics, instead of one unique definition, has a variety of definitions. Next, we introduce three useful definitions that are generally used.

- Mechatronics is a technology which combines mechanics with electronics and information technology to form both functional interaction and spatial integration in components, modules, products and systems (Buur, 1990).
- Mechatronics is a synergistic combination of precision mechanical engineering, electronics, control, and systems thinking in the design of products and manufacturing processes (IRDAC, 1986).
- Mechatronics is a synergistic approach to the integrated and optimal design of a mechanical system and its embedded control system, where solutions are sought that cross the borders of the different domains (Van Amerongen, 2003).

The first definition stresses that mechatronics considers functional interaction and spatial integration of subsystems and disciplines. The second definition indicates that mechatronics is not a conventional engineering discipline or a technology, but a design approach. Van Brussel (1996) defined that mechatronics is a synergistic cross-fertilization between the different engineering disciplines. The third definition addresses a number of important aspects of mechatronics. Synergistic implies that such a solution is not sought in a sequential way, where each domain is optimized in itself, but that solutions in all domains are sought simultaneously. Optimal design implies that a best solution in terms of performance or price is desired (Van Amerongen et al., 2000). In our research, the third definition will be used as a system-level approach to designing the electro-mechanical motion systems that merges mechanical, electrical, control, and embedded software design.

In mechatronic systems, *controlled electro-mechanical motion systems* are the specific class that plays an important role in the manufacturing process. Van Brussel (1996) stated that mechatronics can be defined as the science of motion control, i.e. it encompasses the knowledge base and technologies for the flexible generation of controlled motion. Such systems have been used in the industrial environment; for example, in robotic and manipulator systems, packaging processes, printing and textile industries, and other

industrial applications. For this reason, we aim at applying the research results of this thesis to controlled electro-mechanical motion systems.

A controlled electro-mechanical motion system is defined as an electrically actuated mechanical plant that requires control of the position of the end-effector (Coelingh, 2000). In figure 1.1, the main parts of a controlled electro-mechanical motion system are depicted. The reference path generator indicates a desired path for the end-effector. The actuator, normally an AC or DC linear/rotary electric motor or a piezo actuator, is used to drive the end-effector through a mechanical transmission which optionally includes shafts, gears, belts, linkages, rotational bearings, etc. The control system processes information from appropriate sensors that are generally located at the actuator and/or the end-effector, to create a stable closed-loop feedback control system and to obtain the desired behavior and performance for the end-effector under all relevant conditions. Besides these issues, *safe-guarded control* is important in mechatronic systems. It should be considered at a high priority level. The reason is that while performing a certain task, a robot or manipulator system can encounter a variety of serious malfunctions and faults that can occur in an unwanted manner. If these problems are not identified correctly and handled strictly, they can result in dangerous situations for both human and machine.

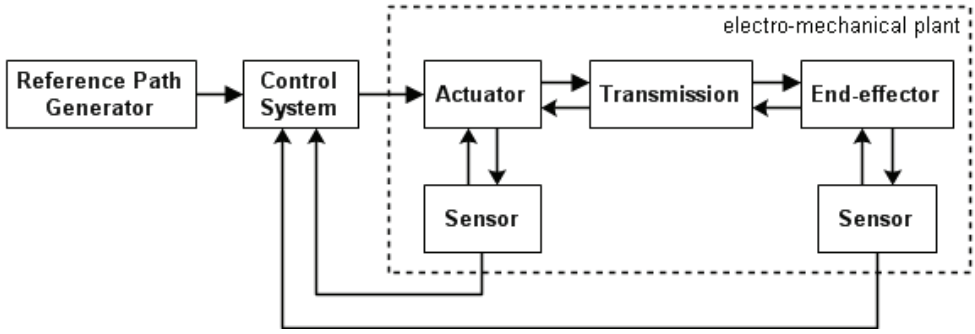


Figure 1.1 Block diagram of a controlled electro-mechanical motion system

In robot systems that exist the interaction between robots and their environment, especially where humans cooperate with robots, force and/or impedance control is generally used (Kamnik et al., 1998, Morinaga and Kosuge, 2003). This thesis will not study this control method and be limited to position control. However, force and impedance control can be applied with the controller design method that will be proposed in this thesis.

The current research on the safe-guarded control issue for mechatronic systems generally addresses some main aspects: how to detect faults, how to handle faults, how is the system's behavior in the error state, and how to recover from faults. The safe-guarded control issue concerns two major aspects. First of all, the safety of humans (or operators) working with the machine has to be guaranteed. Secondly, the machine must not damage itself during operation. The safety for humans is more important. In this research, we are

interested in the development of a safe-guarded control system for mechatronic systems where our main goal is to maximize the reusability in all phases of the control system development process, which are: reuse of controller design and plant model in the design phase, reuse of control software in the implementation phase, and reuse of controller code and device driver in the realization phase. As a result, the control system development time will be shortened, thus the time-to-market of mechatronic products can be reduced.

1.2 Problem Statement and Approach

1.2.1 Introduction

While designing a safe-guarded control system for a mechatronic system (in particular, a controlled electro-mechanical motion system), the designer has to consider several requirements. As these requirements originate from the mechatronic design philosophy (i.e. a multi-disciplinary design methodology), they naturally consist of a set of integrated requirements involving multiple disciplines. In this thesis, requirements related to control and software engineering will get most attention. In the following, *seven integrated requirements* with respect to the development of safe-guarded control systems for mechatronic systems are addressed:

1. The control system is required to have timing performances such as: it meets the deterministic (hard) real-time behavior; it is possible to build multi-threaded controllers with thread safety; it can react to events in both synchronous and asynchronous manner. This requirement is named *deterministic real-time multi-threaded control behavior*.
2. The control system is desired to have an inter-process communication (IPC) mechanism with features: hard real-time, thread-safe, and lock-free data exchange. The IPC mechanism is required to support a port-based asynchronous data flow between realtime and non-realtime components in a deterministic way. This requirement is named *thread-safe and real-time IPC mechanism*.
3. The control system should enable multi-operation modes like start-up, homing, normal operation, shut-down, safe-guarded, etc. in which each operation mode has a different priority level (i.e. priority-based operational sequence), different motion trajectory (such as periodic or non-periodic path), different control system configuration (for example, simple or advanced controller), and different control mission (e.g. accurate position control or safe-guarded control). In addition, operation modes should be designed such that multiple functionalities can be deployed in each mode. Hence, this kind of control systems is typically governed by continuous-time dynamic equations within operation modes and by discrete state transitions between these modes whenever certain events occur. The systems are referred to as hybrid systems which can be described under a hierarchy of

continuous-time systems (operation modes) and finite state machines (FSM). In this thesis, we aim at developing hybrid control systems in which both periodic and aperiodic controllers can be realized. Therefore, a discrete-event system's approach should be applied to provide control systems with the capability to handle discrete-events. This requirement is named *capability of handling events*.

4. The control system should be developed with an efficient support from design support toolchains in multiple stages, from the anlysis, design and (co)simulation in a computer to the implementation, realization and application with a real setup. This requirement is named *support toolchain based development*.

5. The control system should be designed to achieve good performances (e.g. fast response speed, large bandwidth, robust stability, small overshoot, low sensitivity to disturbances and parameter variations) while at the same time require only a short development time, basic knowledge of control theory, and essential skill of control engineering of the designer. However, in practice, depending on the behavior of the plant and the qualitative control requirements that the system has to achieve, the designer can choose a suitable controller configuration. It is because the best controller is the simplest one that is powerful enough to fulfill the system requirements (Cuong, 2008). This requirement is named *achievable good control performances in a short development time*.

6. The control system is required to guarantee safe-guarded control for both operators and machines. The safe-guarded control issue has to cover and deal with a variety of fault sources with different criticality levels. It should also be equiped with functions such as error detection, error handling, graceful degradation, and error recovery together with different degrees of fault tolerance. This requirement is named *safe-guarded control*.

7. The control system should be developed based on reusability of the design results from previous projects into new projects, rather than from scratch. In other words, the control system can be built in terms of reusable, modular-oriented, and library-based components. This requirement is named *reusability of design results*.

In fact, meeting these seven integrated requirements is challenging in the face of complex control problems. Before going into further discussions, the concept of a complex control problem should be clearly understood. This is addressed hereafter.

What is a complex control problem?

In general, the answer for questions such as what is a complex control problem, or what kind of control problem can be considered as complex, etc. is not easy to be formulated. Even if one comes to an answer, this is a subjective choice. In this research, *the complex control problems we are working with are limited in a scope of the definition proposed by Van Breemen (2001)*: "A complex control problem is a hierarchical organized structure of elementary and compound control problems. The latter consists of a set of partial control problems and a set of coupling relationships that exists between these partial control

problems". An example is a swarm robotic system with multiple mobile robots that is required to perform multiple functions such as detect targets, move to targets, follow a target and concurrently avoid obstacles and collision while moving. For more information about this issue, the readers are referred to chapter 2 of the PhD thesis of Van Breemen (2001). Moreover, we will only focus on solving complex control problems in the context of mechatronic systems. Specifically, complex control problems that are considered in this thesis are electro-mechanical motion systems with *multiple degrees-of-freedom (N-DoF)* that requires *multiple operation modes* like start-up, homing, normal operation, shutdown, and safe-guarded mode.

In the next sections, we will discuss in detail these seven integrated requirements in the form of two topics:

- Developing an implementation framework for solving complex control problems.
- Dealing with the trade-off while designing control systems: how to solve the trade-off between the desire to achieve good control performances and a short development time?

1.2.2 An implementation framework for solving complex control problems

Because of the multiple design objectives and constraints, the integrated requirements cannot be solved easily and completely by a single (traditional) control algorithm, i.e. a control system with only one controller such as a conventional PID controller. A number of practical solutions have been proposed such that complex control problems can be solved by using multiple models of computation, heterogeneous design techniques, and an integrated approach of multi-disciplinary engineering (e.g. control engineering, mechanical engineering, electrical engineering, and software engineering) while taking multiple control objectives into account (Van Breemen and De Vries, 2001; Fregene et al., 2001; Waarsing et al., 2003a; Chang et al., 2003; Masina et al., 2004). The current solutions for complex control problems generally result in *a multi-controller system* including a set of subcontrollers that can be combined into an overall solution (Hilhorst, 1992; Saffiotti, 1997; Narendra et al., 1995; Narendra and Balakrishnan, 1997). Therefore, when the multi-controller system is executed the overall performance specification can be obtained. For subcontrollers, several classical and advanced control system design techniques can be appropriately applied, for example:

- PID control (Astrom and Hagglund, 1995; Burns, 2001; Visioli, 2006), state feedback control (Friedland, 1996; Bosgra et al., 2006).
- Fuzzy-logic control (Takagi and Sugeno, 1985; Lin and Lee, 1996), neuro-fuzzy control (Miller et al., 1995; Czogala and Leski, 2000).
- Adaptive and learning control, such as: Model Reference Adaptive Control (Van Amerongen, 1981; Cook, 1994; Kamnik et al., 1998), Iterative Learning Control

(Arimoto et al., 1984; Moore et al, 1992; Jang et al., 1995; Bien and Xu, 1998; Verwoerd, 2005), Learning Feed-Forward Control (Kawato et al., 1987; Miyamoto et al., 1988; Jacobs and Jordan, 1993; Starrenburg et al., 1996; Velthuis, 2000; De Vries et al., 2001), Self-Tuning Regulator (Astrom and Wittenmark, 1995).

The above-mentioned strategy of solving a complex control problem by decomposing it into partial control problems is called *the divide-and-conquer approach* (Johansen and Murray-Smith, 1997). This approach basically consists of three steps (Van Breemen, 2001):

1. Decomposing the overall control problem into a complete set of well-defined partial control problems.
2. Solving the partial control problems.
3. Integrating the partial solutions into an overall solution.

Recently, the field of *agents* and *multi-agent systems* (see section 2.2) have brought promising advantages in structuring and solving complex control problems. The multi-agent systems approach is a way of solving a complex problem through dividing it into many well-defined sub-problems that can be handled by multiple well-structured agent-based solutions. The agent-based solutions are then coordinated to obtain an overall solution. Because of the advantages of agent technology, agents have been combined with controllers in the control engineering discipline to form a new concept named *controller-agents*. It brings the best of both fields together in the form of an *agent-based multi-controller system*.

Pursuing this approach, the research on Agent-Based Multi-Controller Systems (Van Breemen and De Vries, 2000; Van Breemen, 2001) resulted in an agent-based controller design framework for complex control problems named *Multi-Agent Controller Implementation Framework (MACIF)*. The MACIF helps the designer to solve complex control problems by offering concepts to structure the problem in terms of partial control problems and their interdependencies, and integrating partial solutions into an overall solution. It also offers different coordination mechanisms to deal with these interdependencies. After the research of Van Breemen, several MSc projects (Bajracharya, 2003; Eglence, 2003; Bijl, 2006; Flinkers, 2006; Bustani, 2008) using the MACIF were carried out at the Control Engineering Lab (http://www.ce.utwente.nl). These projects created a concept called *Multi-Agent Control Systems (MACS)*.

The MACIF provides a good solution to develop MACS for solving complex control problems with two useful features:

- MACS has an *open architecture* such that “parts” can be added, modified or removed without redesigning and reprogramming the remaining “parts” of the MACS. The “parts” mentioned here are elementary and/or composite controller-agents.
- MACS is *hierarchically structured* in terms of a coordinated group of elementary and composite controller-agents in which each composite controller-agent

generally contains several elementary controller-agents and probably other composite controller-agents.

So, the main benefit of the MACIF is the capability to produce a hierarchically structured MACS with an open architecture. However, we believe that it can be enhanced to provide a better support for MACS designs. Moreover, the MACIF has just provided a theoretical base to develop MACS and its applicability has been proved through only a few simple applications (Van Breemen and De Vries, 2001; Van Breemen, 2001; Eglence, 2003; Flinkers, 2006). Therefore, one of the objectives of this research is *to improve the applicability of the MACIF into practical applications*. Based on the suggestions proposed in section 7.2 (Suggestions for future research) of the PhD thesis of Van Breemen (2001), we emphasize three important aspects of the MACIF that can be improved:

- The MACIF provides a solution to develop discrete-time controller-agents that can run in a single thread and on a single processor. However, it does not support building multi-threaded control systems with thread-safe behavior and (hard) real-time requirements. Moreover, the MACIF does not support an inter-process communication (IPC) mechanism between controller-agents with hard real-time and thread-safe data exchange performance.
- The controller-agents, developed by the MACIF, are only executed periodically. However, solving a (complex) control problem sometimes requires implementing aperiodic tasks. Hence, a discrete-event system's approach should be brought to the MACIF to provide possibility of developing hybrid control systems in which both periodic and aperiodic controllers can be realized.
- The MACIF does not have sufficient support tools to enhance the presented agent-based controller design method. Van Breemen emphasized that the MACIF needs computer-based support tools (like a graphics-based software environment, an editor to construct a MACS by using components) and other debugging tools for MACS. Pursuing the suggestions, two design support tools were developed at the Control Engineering Lab. Bajracharya (2003) implemented the Integrated Design and Implementation Tool for Multi-Agent Controllers (IDITmac); Bijl (2006) developed the 20-sim Multi-Agent Controller Specification (20-Macs). However, the tools have encountered drawbacks that were recorded in (Bijl, 2006; Flinkers, 2006). Although some problems were solved, the others still exist and cannot be easily fixed, thus decreasing the useful role of the IDITmac and 20-Macs tool with regard to the MACIF.

In summary, the MACIF is lacking the following important features, which actually are the integrated requirements (*number 1, 2, 3, and 4*) mentioned in section 1.2.1:

- Deterministic real-time multi-threaded control behavior;
- Thread-safe and real-time inter-process communication (IPC) mechanism;
- Capability of handling events;
- Efficient design support toolchain.

In this research, developing an implementation framework for solving complex control problems in mechatronic systems is one of the objectives we have pursued. To obtain this objective, our approach is to develop a new implementation framework for MACS by means of inheriting and improving the advantages of the MACIF and simultaneously providing the above-mentioned missing features for the MACIF. Specifically, this research is going to study the answer to two research questions:

> The 1st research question: How to provide the deterministic real-time multi-threaded control behavior, thread-safe and real-time IPC mechanism, and the capability of handling events for the MACIF?
>
> The 2nd research question: How to improve the capability of developing the hierarchically structured MACS with an open architecture of the MACIF?

To solve the 1st research question, the approach that we advocate is a combination between the MACIF and the OROCOS framework (Orocos, 2009a): the main advantage of the MACIF, i.e. capability to produce the hierarchically structured MACS with an open architecture, will be combined with the strong point of the OROCOS framework, i.e. a generic feedback control architecture that can be combined easily with several hard real-time targets like RTAI/LXRT (www.rtai.org) or Xenomai (www.xenomai.org) to develop multi-threaded control systems with deterministic real-time behavior and thread safety. As a result, a new implementation framework named **ORO**COS-based Implementation Framework for **MACS** (*OROMACS framework*) has been formed to support the development of real-time multi-threaded MACS. The approach to solving the 2nd research question is an extension of the port-based polymorphic modeling approach (De Vries et al., 1993; De Vries, 1994) into the OROMACS framework in such a way that it makes controller-agents become *polymorphic*, i.e. a controller-agent with a specific Type, which defines its interface, can have multiple Specifications, which define different implementations of this interface. As a result, we obtain *polymorphism in the specification and realization of MACS*. Hence, the capability of developing the hierarchically structured MACS with an open architecture is well improved. In addition, the OROMACS framework enables the development of hybrid control systems with multi-operation modes. This work will be presented in chapter 3.

After these two research questions have been solved, we obtain the OROMACS framework for solving complex control problems. However, the deployment of the OROMACS framework into practical applications could be realized in an easier way if it is efficiently assisted by a design support toolchain. Because the IDITmac and 20-Macs tool cannot be reused for the OROMACS framework, a new design support toolchain has been developed: OROCOS - 20sim Cosimulation (Bozlak, 2009), OROMACS Browser (Tadele, 2009), and MACS Editor & Code Generation (Bozlak, 2010).

1.2.3 Trade-off while designing control systems

The field of control engineering is concerned with solving (complex) control problems where the objective is to obtain desired dynamic behavior and performance specifications. In practice, depending on the knowledge and experience of the designers about the controlled plant and problem-related domains, two possible design trajectories of a control system can be used (Coelingh, 2000):

Short trajectory: a ready-to-use controller is applied, which can be tuned for each particular problem. The control system obtained by the short trajectory normally gives non-optimal performance or a certain acceptable performance level only, but the development time is relatively short and the required designer's skills in control engineering and specific problems are not high.

Elaborate trajectory: a custom-made controller is obtained through an iterative design process that involves seven development steps:

1. Problem specification: the controlled problem is transformed into (partial) specifications and/or requirements that can be handled by available design methods and techniques.
2. Modeling and identification of the plant: analytical modeling techniques can be used to obtain a plant model. Modeling techniques, such as equations, block diagrams, bond graphs and iconic diagrams, are generally used. An analytical plant model also can be obtained by identification techniques (experimental modeling methods). In this case a finite number of measured data from the physical plant is used to estimate characteristic parameter values for the analytical model.
3. Linearization and reduction of plant model: the obtained plant model is often complex because it may contain nonlinear and high-order terms. Therefore, linearization techniques can be used to get a linearized model and then reduction techniques are used to obtain a linear model with an appropriate order.
4. Controller design: the control problem specification and the suitable plant model resulting from the previous steps are now used to design the control system. In general, many available controller design techniques can be applied at this step.
5. Simulation and evaluation of the designed control system: the behavior of the controlled system can be predicted through simulation with both the reduced-order linearized plant model and with the full-order nonlinear plant model. The simulation result give a sufficient evaluation with regard to the designed control system to decide if a design iteration should be taken or an implementation of the designed controller will be proceeded. At this step, the technique Model-In-the-Loop (MIL) simulation is generally used.
6. Implementation of the controller: the successfully evaluated control system is now implemented as a computer-based control algorithm, i.e. forming an embedded

control system in the form of executable code on a general hardware platform. At this step, the code-based controller is desired to be tested before officially being deployed with a real plant. Techniques such as Software-in-the-Loop (SIL) simulation, Hardware-in-the-Loop (HIL) simulation, and Processor-in-the-Loop (PIL) simulation are generally applied for this purpose.

7. Realization of the embedded control system with a real plant: the embedded control software is realized for an experimental plant with hardware connection. The interaction between the embedded controller and the experimental plant is done through input and output interfaces of the embedded hardware with sensors and actuators of the plant. Hence, the realization of embedded controller requires a co-design and co-implementation process involving software components and hardware parts.

This research aims at solving complex control problems in mechatronic systems by using advanced design techniques so that the elaborate trajectory will be followed. The elaborate trajectory can lead to better solutions, but only if the designer has sufficient knowledge of control theory, good skill of control engineering and a deep understanding of the problem; thus it normally causes a long development time. It is *the well-known trade-off while designing a high performance robust control system* (Graebe, 1999; Coelingh, 2000). This trade-off actually is the integrated requirement (*number 5*) mentioned in section 1.2.1. We remark here that, modeling and identification of the plant (step2) and linearization and reduction of plant model (step 3) are not the topic of this thesis. We will consider the steps (1, 4, 5, 6, and 7) in the context of two research questions which are discussed hereafter.

Although the OROMACS framework has the advantages in solving complex control problems, it still faces the trade-off in a specific case, i.e. *the trade-off while designing MACS*. Solving this trade-off is one of the objectives we have pursued. Particularly, we will solve it together with strict requirements of the *safe-guarded control*, i.e. the integrated requirement (*number 6*). However, we limit ourselves to the safe-guarded control of mechatronic systems that will be categorized into two classes: simple mechatronic systems, i.e. electro-mechanical motion systems with one degree-of-freedom (1-DoF); and complex mechatronic systems, i.e. electro-mechanical motion systems with multiple degrees-of-freedom (N-DoF). Specifically, this thesis will study to handle faults and particularly fault propagations that may happen in N-DoF motion systems. We will consider two possibilities of fault propagations: (i) the propagations of influence spheres of faults, i.e. from faults occurring on a single axis to faults involving multiple axes; and (ii) the propagations of criticality levels of faults. Note that, *error/fault detection itself is not the topic of this thesis; it is assumed that faults are reliably detected*. This research is going to study the answer to the following research question.

> The 3rd research question: How to solve *the trade-off* between the desire to achieve a real-time safe-guarded MACS having good performances and a short development time?

Moreover, like most other implementation frameworks, the OROMACS framework has another problem that needs to be improved: the MACS design process has to be largely repeated whenever the designer moves to new applications or other mechatronic systems, even when these control problems resemble each other. The main reason is *the lack of support for reusability of the design results from previous projects into new projects*. This problem in fact is the integrated requirement (*number 7*), i.e. reusability of design results. Dealing with this shortcoming leads to the 4th research question of this thesis.

> The 4th research question: How to support *the reusability* of the real-time safe-guarded MACS design results from previous projects into new projects?

We are going to tackle these two research questions through building up a control system design method that can be used to develop high performance intelligent controllers for mechatronic systems while at the same time decreasing the development time and requirements of control theory knowledge and control engineering skills of the designers. The approach that we use is a combination of two aspects: (i) the OROMACS framework, and (ii) the pattern-based design method. This combination will result in *a pattern-based safe-guarded MACS design method* (see chapter 4) that provides two advantages:

- The designers with less professional understanding of control theory and particular knowledge of multi-agent control systems can design the real-time safe-guarded MACS for mechatronic systems with good control performances and in a short development time (i.e. the 3rd research question is solved).
- The design of a real-time safe-guarded MACS for a complex mechatronic system can be created by reusing the design results of more simple mechatronic systems (i.e. the 4th research question is solved).

1.3 Contributions of Research

Two main contributions of this thesis are:

1. *The OROMACS framework* for solving complex control problems that supports the development of multi-threaded MACS with deterministic real-time control behavior and thread-safe real-time IPC mechanism. This framework makes MACS designs polymorphic. In other words, it creates polymorphism in the specification and realization of MACS. As a result, it makes the design and programming of a MACS become a matter of configuration and composition of controller-agents. The OROMACS framework can pre-schedule the operationality of MACS by using different coordination mechanisms and provides the capability to develop the hybrid control system with multi-operation modes that can handle discrete-

events. In addition, this framework creates the kind of composite components for the OROCOS framework.

2. *A pattern-based safe-guarded MACS design method* that solves the trade-off between the desire to obtain a real-time safe-guarded MACS with good control performances and a short development time. The design method also supports the reusability of MACS design results from previous projects into new projects. This research contributes nine control system design patterns which are well organized to formulate two reusable generalized safe-guarded control solutions, one for simple mechatronic systems and one for complex mechatronic systems. As a result, the designer can quickly build up such a complete safe-guarded MACS for a mechatronic system; thus making the short time-to-market objective obtainable for control system development.

1.4 Outline of Thesis

This thesis is organized in five chapters whose contents are briefly described hereafter:

Chapter 2: A review of related research

This chapter gives an overview of issues related to this thesis. After introducing the general information of agents and multi-agent systems (MAS), we discuss the applicability of agent and MAS technology in a general view. Next, we present a particular case which is the application of MAS in the control engineering discipline. The next topic that we consider is the pattern-based design method in software and control engineering. Finally, we discuss safety issues in Human-Robot Interaction (HRI).

Chapter 3: An OROCOS-based implementation framework for MACS

This chapter presents the development of a MACS implementation framework for solving complex control problems. This chapter starts with summarizing the fundamental idea of coordination and main contents of the MACIF to provide a background for the next parts. We assess and compare several feasible approaches that can be used for developing real-time MACS. After addressing the resemblance between the MACIF and the OROCOS framework, we describe the proposed functional combination of the two frameworks. The result of this combination is the OROMACS framework. Next, we explain how a composite controller-agent is formed in the OROMACS framework. Another topic of this chapter is an extension of the port-based polymorphic modeling approach into the OROMACS framework that makes the controller-agent and MACS polymorphic. Two new concepts, being OROMACS TaskContext and OROMACS Root-Agent, will be introduced. Finally, we discuss roles of the polymorphism approach and coordination principles.

Chapter 4: A pattern-based safe-guarded MACS design method

This chapter presents a pattern-based safe-guarded MACS design method that is developed based on a combination between the OROMACS framework and the pattern-based design method. Hence, the work in this chapter is considered as a next step that inherits and develops the result obtained in chapter 3. This chapter starts with the description of safety issues in mechatronic systems. Next, the design of a safe-guarded MACS for the DemoLin setup (a simple mechatronic system) is presented; and then a reusable generalized safe-guarded control solution for simple mechatronic systems is formulated. After that, the design of a safe-guarded MACS for the TriPod setup (a more complex mechatronic system) is presented that reuses the safe-guarded MACS design results of the DemoLin setup; and then a reusable generalized safe-guarded control solution for complex mechatronic systems is formulated. Finally, a summary of the control system design patterns is given.

Chapter 5: Discussion

This chapter presents a review of the previous chapters, conclusions, and suggestions for future research.

Chapter 2

A Review of Related Research

2.1 Introduction

This chapter presents some background knowledge and reviews issues related to this thesis. In section 2.2, we introduce an overview of agents and multi-agent systems (MAS). Next, section 2.3 discusses applicability of agent and MAS technology in a general view. In section 2.4, we present a particular case which is the application of MAS in the control engineering discipline. Several typical applications involving Multi-Agent Control Systems (MACS) and control systems that combine hybrid control and MAS will be considered. Section 2.5 addresses the pattern-based design method in software and control engineering. Finally, section 2.6 reviews safety issues in Human-Robot Interaction (HRI).

2.2 Agents and Multi-Agent Systems

2.2.1 Agents

In the early 90's the concept of agents appeared at the same time in both information and communication technology with some typical kinds of agents such as mobile agents, interface agents, and information agents (Monostori et al., 2006). In general, an agent is

regarded as an autonomous entity responsible for performing a certain task in coordination with its community (Van Breemen and De Vries, 2000). However, since the term "agent" has been used largely in various domains with different purposes, a unanimous precise and technical definition of this concept cannot be formulated easily (Wooldridge, 1999; Nwana, 1996). In spite of the lack of a technical definition, researchers managed to come up with a notion of what an agent is, and is not. A definition that captures the essential aspects of being an agent, what is approved by most researchers, was given in (Franklin and Graesser, 1996): "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future". This definition of an agent resembles the concept of a single closed-loop controller, as a single closed-loop controller is a system that also senses and acts and is designed to meet some objective. However, the concept of an agent is strongly associated with localization and abstraction: all aspects related to a particular subproblem should be contained in one object (Van Breemen and De Vries, 2000).

Another definition of agents given by Schoop et al. (2001), adapted from Jennings and Wooldridge (1998), used in flexible production systems: "An agent is considered a software entity situated in a flexible production environment, with enough intelligence that is capable of autonomous control actions in this environment and of co-operation relationships by participating in associations agreements with other entities in order to meet its design objectives". According to this definition, an agent should be able to act without the direct intervention of humans and/or other agents, and should have control over its own actions and internal states.

The interaction between an agent and its environment is described in figure 2.1 with properties: an agent (i) makes observations about its environment, (ii) has its own knowledge and beliefs about its environment, (iii) has preferences regarding the states of the environment, and finally, (iv) initiates and executes actions to change the environment.

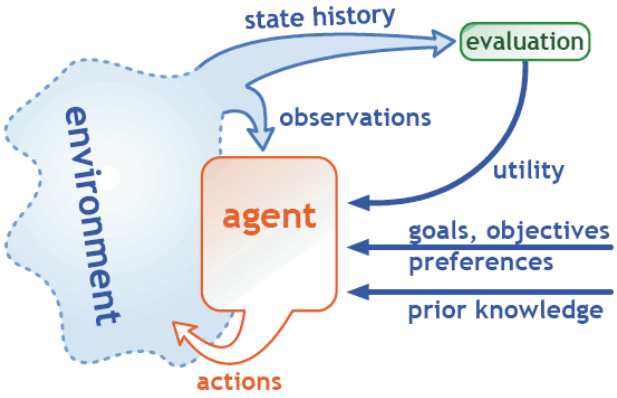


Figure 2.1 An agent and its environment (Russel and Norvig, 1995; Monostori et al., 2006)

Besides the definitions for agents, it is also practical to characterize an agent by having a list of characteristics that an entity must have in order to be denoted as agent. The list contains the following main attributes (Jennings and Wooldridge, 1998):

- *Autonomous:* an agent is an intelligent entity that can perform its local task independently without assistance from other systems and/or direct intervention from humans.
- *Social ability:* an agent is able to communicate with its environment, which is a group of other agents and the physical environment of the whole system, in order to complete its own local task and to help others with its activity to achieve the global objective of the whole system.
- *Reactivity:* an agent should perceive its environment and respond in a positive manner to the changes that occur in the environment.

Particularly from the control engineering point of view, an agent can be seen as an object with its own thread of control, which is capable of autonomous decision making. It means that an agent can decide for itself whether it should undertake some actions (Van Breemen and De Vries, 2001).

2.2.2 Multi-Agent Systems (MAS)

Although an agent is presented as an entity that solves problems in order to achieve its own goal, many problems are far too complex to be handled by an individual agent. Only a "society of agents" is capable of solving such problems (Van Breemen, 2001). Minsky (1986) described an architecture of mind in which the problem solving capability of the human mind is represented in terms of such a society of agents. In a more technical meaning, a society of agents is called a multi-agent system (MAS) and studying of MAS has become an exciting field of research. Similar to the agent definition problem, various definitions from different disciplines have been proposed for the term MAS. A notion of MAS was given in (Durfee and Lesser, 1989). They defined a MAS as a loosely coupled group of problem solvers that work simultaneously to solve problems that are beyond the individual capability or knowledge of each problem solver. Wooldridge (2002) stated that a MAS consists of a number of autonomous, intelligent agents, which can interact with each other to pursue their own goals or solve cooperatively common problems. The Foundation for Intelligent Physical Agents introduced another definition (FIPA, 2003): "A Multi-Agent System is a system composed of a great number of autonomous entities, named agents, having a collective behavior that allows to obtain the desired function/service".

The term MAS has been given a more general meaning. It is now used for all types of systems composed of multiple autonomous components with the following characteristics (Jennings et al., 1998; Van Breemen, 2001):

- Each autonomous component has incomplete information and/or capabilities to solve the problem and therefore it has a limited viewpoint;

- There is no global control system;
- Data are decentralized;
- Computation is asynchronous.

Ren and Anumba (2004) stated that agent and MAS technology is ideally suited to solving complex problems that require multiple problem solving methods, multiple perspectives and/or multiple problem solving entities because of its ability to provide advantages such as: efficiency, reusability, robustness, flexibility, adaptability, and scalability. The overall operation of a MAS is governed by an organization that manages the interactions of individual agents. Although there may be no global control or centralized data and the computations are asynchronous, some organizational rules always exist (Monostori et al., 2006). The organization determines the activity scope of agents, as well as their potential interactions (see figure 2.2). The various organization patterns of MAS such as: (i) star (centralized structure, see figure 2.3a); (ii) ring (decentralized structure, see figure 2.3b); (iii) chain (hierarchy structure, see figure 2.3c), and (iv) network (democratic structure, see figure 2.3d), provide different ways to obtain system-level design objectives and/or to facilitate the deployment of desired properties in multi-agent systems. In summary, the research field of MAS aims at putting forward the principles for the construction of complex systems involving multiple agents and providing tools and mechanisms for the coordination of independent agent behaviors. The MAS paradigm provides structures for building systems with high autonomy and for specifying interaction and coordination rules among agents (Fregene et al., 2005). The reason is that the solution of complex problems is too large for just one centralized agent to deal with; hence, a MAS is planned to interconnect separately developed agents so that the entire system performs better than any one of its members.

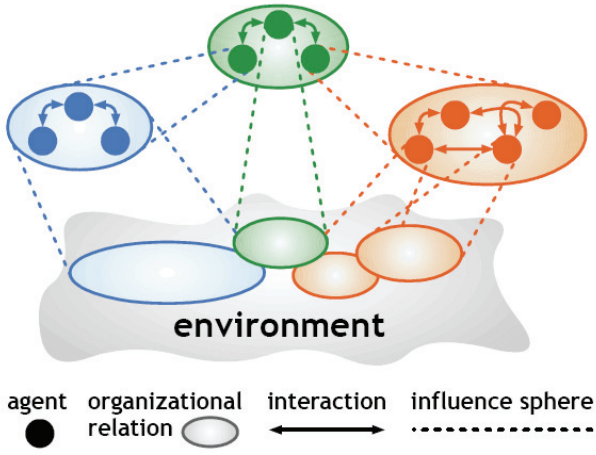


Figure 2.2 Generic scheme of a MAS (Jennings, 2001; Monostori et al., 2006)

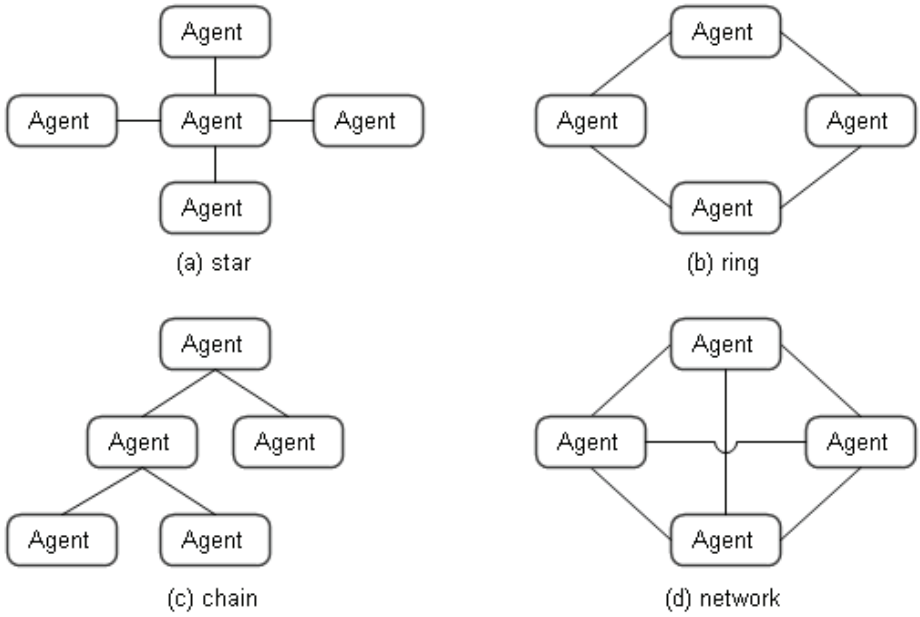


Figure 2.3 Organization patterns of MAS (adopted from Lockemann et al., 2006)

2.3 Applicability of Agent and MAS Technology

2.3.1 Introduction

At present, many agent platforms and agent-oriented development environments, which are available either commercially or in open source, are ready to be applied. This provides an essential basis for the transfer of research results from university laboratories and research institutes into practical applications. Pechoucek et al. (2006) addressed three reasons that brought the successful application of agent technology:

- Academic research of agent technology has been deployed with the support of advanced techniques from the Artificial Intelligence (AI) areas. The techniques have enabled the design and implementation of agent-based applications having an obvious advantage in comparison with traditional systems.
- The width and depth development of human resource considered as the second generation of agent researchers who understand thoroughly the principles and possibilities of agent and AI techniques with the high skills to implement these techniques into practical solutions.

- The rapid development of computer hardware and considerably increased availability of open source components, especially in the software domain, have made development of agent-based applications cheaper, faster and more reliable.

Besides advantageous points, Pechoucek and Marik (2008) pointed out the main bottlenecks in fast and massive adoption of the agent-based solutions which are based on the common knowledge in the community and also by observation they gathered through involvement in AgentLink III (http://www.agentlink.org/index.php):

- Limited awareness about the potentials of agent technologies: agents have only been used in a few specialized disciplines, while they could have been deployed in other domains as well.
- Limited publication of the successful applications using agent technologies.
- Misunderstandings about the agent technology capabilities: people often either have over-expectations of agent potentiality or sometimes abuse agent terminology in the domains where it is not fit.
- Being afraid of risks that can arise when a new technology is applied.
- Lack of mature design and development tools for the design, implementation and deployment of agent-based solutions.

Moreover, a problem arisen repetitively at many agent conferences and workshops is the lack of agreement over what actually constitutes an agent. Interestingly, this has both helpful and deterrent effects on the development of agent-based applications (Luck, 1999). On the one hand, it causes difficulties in communication, confusion in terminology, overuse of the concept, and a proliferation of agent labeled systems without obvious justification for doing so. On the other hand, it leads to the popularisation of agent-based systems in the public community. In order to have an overall view of application domains of MAS, it is necessary to have a reasonable classification of agent-based application types. Amongst a lot of others, Pechoucek et al. (2006) discussed that the available agent techniques have performed well in five application groups:

1. Systems where data required for automated decision making are not centrally available.
2. Systems with requirements for a time-critical response and high robustness in a distributed environment.
3. In simulation and modeling scenarios.
4. Systems with restrictions on information sharing that prevent a centralized decision-making architecture.
5. In open systems scenarios.

The following sections discuss this classification and present some typical examples.

2.3.2 Systems where data required for automated decision making are not centrally available

According to Pechoucek et al. (2006), this agent-based application group has been formed because of three distribution types of knowledge and control:

- *Geographical distribution*, like applications in logistics, collaborative exploration, mobile and collective robotics, and pervasive systems.
- *Temporal distribution*, such as the application of satellite networks where satellites have different views of the earth at different times of the day.
- *Conceptual distribution*, e.g. applications with layered hierarchies, where entities at one layer might have no knowledge of events or processes at other layers.

The first representative example in this group is the application of agent technology in electricity infrastructure control by the ECN (Energy Research Center of the Netherlands). Kok et al. (2005) presented the *PowerMatcher*, a market-based control concept for supply and demand matching in electricity networks, which has been developed in cooperation with industrial and academic partners. The PowerMatcher uses a combination of multi-agent systems technology with distributed algorithms and electronic markets to form an appropriate technology needed for control and coordination tasks in the future electricity network (see figure 2.4).

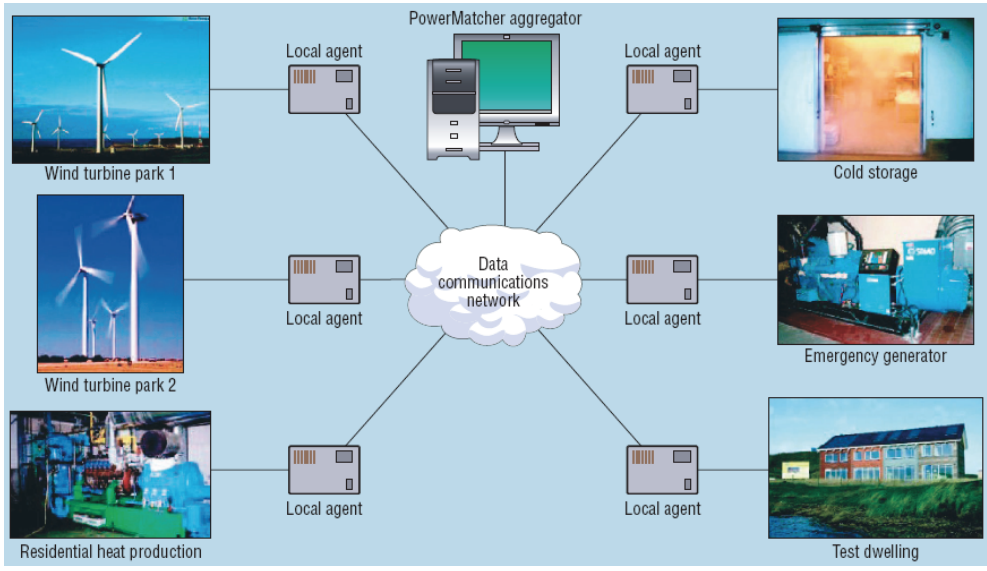


Figure 2.4 Model of the PowerMatcher using the market-based control concept and MAS technology (Kok et al., 2005; Pechoucek et al., 2006)

In this system, a local agent represents each device, which performs two jobs: (i) operating the device process in an economically optimal way within the system's constraints and (ii) negotiating their electricity consumption or production on an electronic exchange market. The resulting market price determines the power volume allocated to each device. Agent reactions on price fluctuations make the balance between production and consumption of electricity by devices in a subnetwork increase. As a result, the net import profile of the subnetwork is smoothed and peak demand is reduced, which meets the desire of electricity distribution networks (Kok et al., 2005). Experiment results indicate a decrease of the total power imbalance by approximately 25 percent. Moreover, reduction of unpredictability in the trade portfolio reduces imbalance costs charged to the trader by the independent network operator (Pechoucek et al., 2006).

The second example is the application of MAS for controlling teams of unmanned aerial vehicles (UAVs) (Baxter and Horn, 2005). This MAS architecture is depicted in figure 2.5.

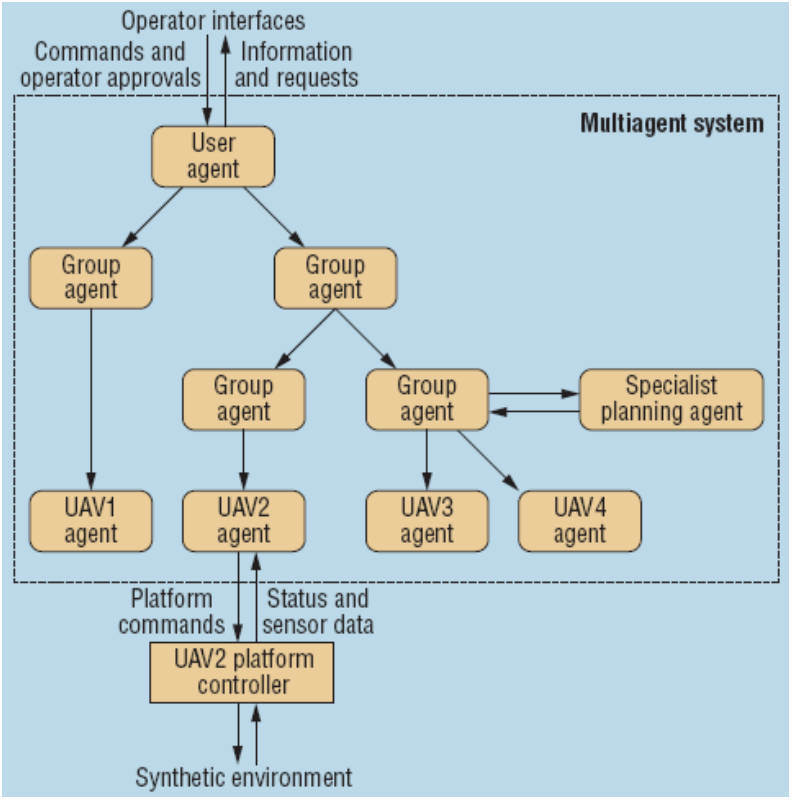


Figure 2.5 MAS architecture for controlling teams of UAVs (Baxter and Horn, 2005; Pechoucek et al., 2006)

The MAS contains four types of agents: (i) *User agent* allocates individual UAVs to the tasks set by the operator and provides information for the operator; (ii) *Group agents* plan and coordinate the execution for tasks; (iii) *UAV agents* interact with individual platforms, command the autopilot to undertake specific maneuvers and receive status and sensor information; (iv) *Specialist planning agent* directly supports the Group agents. The MAS controls the UAVs and can self-organise to achieve the tasks set by the operator with interaction by means of a variable autonomy interface (Baxter and Horn, 2005). The authors describe how the agents are integrated with the rest of the system and present a number of system integration issues that have arisen.

2.3.3 Systems with requirements for a time-critical response and high robustness in a distributed environment

Typical examples in this group are manufacturing and industrial control systems that require time-critical constraints and fast local reconfiguration capabilities to handle problems instantaneously.

A successful application of agent technology in the manufacturing control by the Rockwell Automation company is selected to introduce. They have presented *an agent-based industrial control architecture* (Marik et al., 2005) that provides a higher degree of flexibility and reconfigurability of manufacturing solutions as well as higher robustness of industrial systems. In this control architecture, they implement each agent as a module that encapsulates a real-time control agent and a software agent (see figure 2.6), with features:

- The *real-time control agent* directly handles the information from physical sensors and actuators by using a low-level language (e.g. the ladder logic programming language). At this low-level control, the IEC 1131-3 communication standard (http://www.software.rockwell.com/corporate/reference/iec1131/) is applied.
- The *software agent* is implemented in a high-level programming language (like C++ or JAVA) and normally handles decision making and negotiation. At this high-level control, the FIPA communication standard, e.g. Agent Communication Language (http://www.fipa.org/specs/fipa00061/SC00061G.html), can be used.

In order to simplify the integration of this agent-based control architecture with existing industrial automation control architectures which is based on Programmable Logic Controller (PLC), Rockwell Automation modified ControlLogix PLC's firmware so that the software agents (i.e. C++ or Java code) can run directly inside the PLC in parallel with the ladder logic code. This runtime interface solution allows the information transfer among the software agents and the real-time control agents in the PLC.

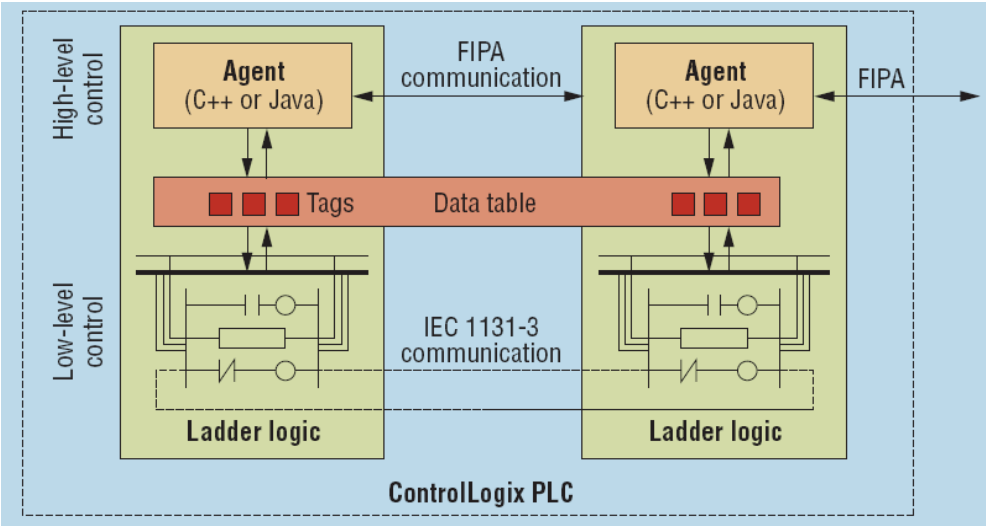


Figure 2.6 Real-time agent-based industrial control architecture used in the ControlLogix PLC (Marik et al., 2005; Pechoucek et al., 2006)

2.3.4 In simulation and modeling scenarios

Agent technology has been deployed in simulation and modeling scenarios with the objectives to get an easy migration to the real environment and replace traditional simulation techniques which are expensive. The common trend in this application group is to develop agent-oriented simulation tools that combine the emulation of the controlled manufacturing equipment with the emulation of agent-control activities (Vrba, 2006).

A remarkable application in this group is an agent-based control and simulation environment called the *Manufacturing Agent Simulation Tool (MAST)*, developed by Rockwell Automation for material handling applications in flexible manufacturing systems (Vrba, 2003). The MAST is completely implemented in JAVA language and uses the JADE (**J**ava **A**gent **DE**velopment) framework (http://jade.tilab.com/) as the development and run-time environment for the agents. An agent library, involving basic components of material handling systems such as work cell, conveyor belt, switch, etc., has been developed. The cooperation of agents aims at finding the optimal transportation routes between the work cells which are interconnected via a network of the conveyor belts and intersections (Vrba and Marik, 2005). An important feature of the system is the fault tolerance and structure flexibility. A failure of any component causes the agents to start negotiations on the alternative transportation paths. Moreover, new components can be added to the system or the existing ones can be removed while the rest of the system still continues its operation.

The MAST simulation environment consists of the following parts (Vrba, 2006):

- The *agent control part* that contains a library of Java/JADE classes representing basic material handling components.
- The *emulation part* that provides the emulation of the physical-manufacturing environment.
- The *run-time interface* that supports the exchange of sensor and actuator values between agents and the emulation part, as well as between agents and the physical manufacturing equipments when they are connected to the real hardware.
- The *graphical user interface (GUI)* that supports the graphical drag-and-drop design of the material handling system and the visualization of the simulation.

The main advantage of the MAST is that it allows a smooth shift from the simulation to real-world control that completely reuses the agent algorithms. The MAST was applied to simulate the holonic packing cell of the Center for Distributed Automation and Control at the Cambridge University (Fletcher et al., 2003).

2.3.5 Systems with restrictions on information sharing that prevent a centralized decision-making architecture

This application group mainly focuses on the design and implemetation of web-based systems where competitive and non-cooperative coordinations used, such as: e-commerce applications, supply-chain management, e-business, etc. In e-commerce applications, software agents have offered a promise to change e-commerce trading by helping traders to purchase products based on their interests and preferences. Buying and selling become an ideal market for intelligent agents. It is a reason to bring comparison-shopping agents into e-commerce applications and web-based business systems. The world-first comparison-shopping agent is *Bargain Finder* (Krulwich, 1996). It can search a number of online music stores for the best price of an album. Another one is *ShopBot* (Doorenbos et al., 1997), a fully-implemented, domain-independent comparison-shopping agent. The ShopBot agent can autonomously learn how to shop at software and CD vendors. After learning, it can speedily visit the vendors to extract product information, and summarize the results for the user. ShopBot enables users to find superior prices and substantially reduce shopping time. The current research in this domain focuses on the goal to create collaborative multi-agent based e-commerce frameworks that allow autonomy, proactivity, and personalization.

2.3.6 In open systems scenarios

In such open environments like the internet, agents invariably coordinate with others in order to meet designer's objectives. Agents which participate in a coordination (either cooperative or competitive) usually face two major challenges: firstly, they must be able to

find each other in such an open environment where agents might appear and disappear unpredictably; and secondly, after finding and locating each other, they must be able to coordinate or negotiate. To address the issue of finding agents in an open environment, middle agents (Decker et al., 1997) have been proposed. The workround is that each agent advertises its capability to some middle agent and so they can find each other. Many different types of middle agents have been identified and implemented, such as: matchmakers or yellow-page agents that match advertisements to requests for advertised capabilities, blackboard agents that collect requests, and and brokers that can process both (Jennings et al., 1998).

2.4 Applications of MAS in Control Engineering

As discussed in chapter 1, the research of Van Breemen (2001) resulted in an agent-based controller design framework for complex control problems, named Multi-Agent Controller Implementation Framework (MACIF). As this thesis will inherit and improve the MACIF, we will first discuss its current situation and then review several representative studies which are relevant to the MACIF or directly apply the MAS technology in the control engineering field. We are particularly interested in applications involving Multi-Agent Control Systems (MACS) and control systems combining hybrid control approach and MAS technology. Before going into further discussions, we answer a question that would be given: which one of five application groups discussed in the previous section that MACS should be classified into? Because MACS have the attributes of systems where data required for automated decision making are not centrally available, and with requirements of time-critical response systems, it can be classified into a mixture group which is a combination of the first and the second application group.

2.4.1 Current situation of the MACIF

To enhance applicability of the MACIF into practical applications, follow-up research focused on the development of design support tools. Bajracharya (2003) developed a specification language, called Multi-Agent Controller Specification Markup Language (*MacsML*) for MACS, which is based on eXtensive Markup Language (XML). He also created a software tool named Integrated Design and Implementation Tool for Multi-Agent Controllers (*IDITmac*) that supports the C++ code generation for MACS from the MacsML-based specifications (see figure 2.7). The generated code can be compiled and linked to generate: (i) *a dynamic link library* that can be linked with the 20-sim software (Controllab Products, 2009) to simulate the MACS; and (ii) *an executable file* for implementing the MACS in a real system with 20-works. Although the IDITmac is a useful tool in the design of MACS, it still requires developers to write code since the MACS specifications need to be provided in the form of MacsML code. The way to specify a

MACS by directly writing the MacsML code is a repetitive, tedious, time consuming and error-prone task. To solve this problem, Bijl (2006) developed a software tool named 20-sim Multi-Agent Controller Specification (*20-Macs*) that can automatically convert the MACS specifications in the form of 20-sim models into the MacsML-based specifications (see figure 2.7). As a result, developers can design MACS using the hierarchical and user-friendly graphical interface of the 20-sim environment, instead of using the plain text editor of the IDITmac tool.

Based on two design support tools, some practical applications of MACS were developed. Eglence (2003) realized a safe control system for a parallel manipulator (the TriPod setup) including a setpoint generator. The safe control system was implemented in the form of a MACS that includes various operation modes such as Startup, Alarm, Emergency and Standard in which the latest one consists of a pool of controller-agents like Stop agent, Operate agent, Shutdown agent and HoldZero agent. This MACS is actually implemented in terms of MacsML files that contain the MacsML-based specifications. The IDITmac tool was used to generate C++ code for the designed MACS and then create the dynamic link library and executable file for the simulation purpose and real-time execution, respectively. Flinkers (2006) developed a learning multi-agent controller for a linear motor (the DemoLin setup) by using both IDITmac and 20-Macs tool. First, he designed and implemented a MACS using a PID controller-agent and then integrated a learning feed-forward controller using key sample machines which was developed by De Kruif (2004) into the existing MACS, thus forming a learning multi-agent controller.

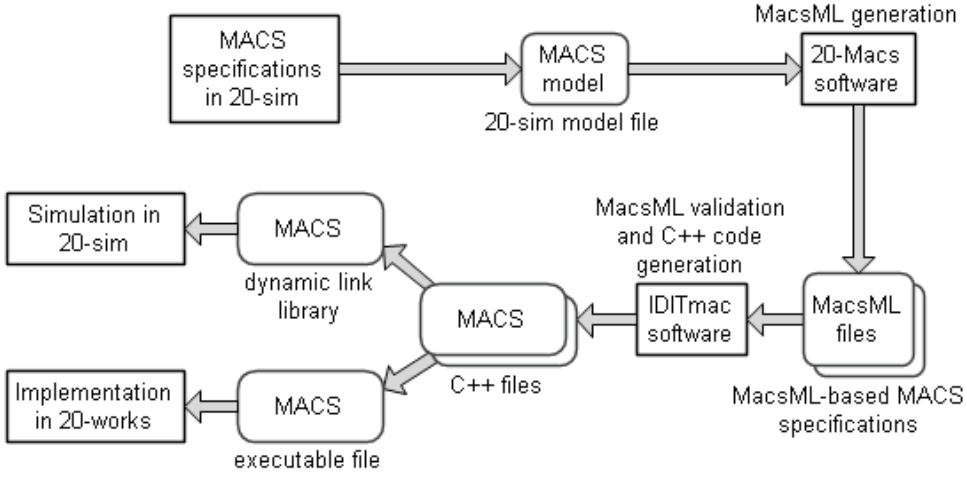


Figure 2.7 Overview of the design support tools and steps for developing a MACS (adopted from Flinkers, 2006)

2.4.2 An architecture of MAS for control of fossil-fuel power unit

Chang et al. (2003) and Masina et al. (2004) proposed a MAS architecture applied to a fossil-fuel power unit (FFPU). The model consists of a Central Agent (CA), a Setpoint Agent (SA), a Feedforward Agent (FeFA), a Feedback Agent (FeBA), a Limiter & Scaling agent (LSA), and a Threat/Emergency Agent (TA). This MAS architecture is depicted in figure 2.8.

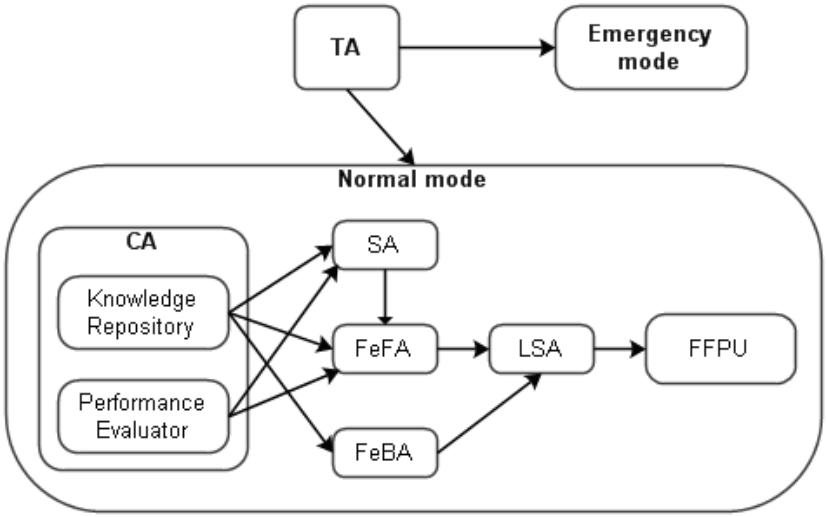


Figure 2.8 MAS architecture applied to the FFPU (adopted from Masina et al., 2004)

In Masina et al. (2004), the authors described the goal and task of agents:

- The *CA* is the key agent that monitors and runs the entire system by sending the message requests to different agents and monitoring their status for an efficient and proper operation of the plant at different operating regions.
- The *FeFA* facilitates a wide-range set-point driven operation for the FFPU and provides off-line operator-requested system adaptability to achieve optimal operation. The FeFA includes three neuro-fuzzy controllers for power, pressure and level and it makes decisions using the system database provided by the CA.
- The *FeBA* provides corrective control actions along the commanded setpoint trajectories to overcome the effect of disturbances and uncertainties in the whole operating scope of the FFPU. It consists of three feedback controllers using conventional PID control for power, pressure and level.
- The *SA* decomposes the task of the multi-agent control system by multi-objective optimization. The essential goal of the SA is to harmonize the slow response of the

boiler with faster response of the turbine generator and achieve fast and stable response during load changes and disturbances.

- The *TA* ensures that the plant is always in the safe operating region and takes care of any emergency situation. It does not get activated unless there is serious fault in the system, which neither agent can solve nor have any immediate solution.
- The *LSA* ensures that the plant gets inputs in the safe range specified through introducing scaling factors in the command signals to the actuators.

The communication protocols between agents are defined by using the finite state machine theory (FSM). The Matlab-Simulink and Stateflow toolbox are used to investigate the strategies, to develop the architecture of MAS for the FFPU and to simulate the communication protocols. The FSM is represented using state transition diagrams, which are developed using the Stateflow toolbox. To simulate and demonstrate the multi-agent systems architecture, the entire communication protocols are implemented in Stateflow representation. The agents are essentially Stateflow and Simulink blocks. The Stateflow and the Simulink blocks blend together seamlessly, thus running a simulation automatically executes both Stateflow and Simulink portions of the model (Masina et al., 2004).

Here, we consider the communication protocols between agents that are used in this MAS architecture. The CA coordinates the agents in the MAS according to different operating conditions. It perceives the environment of the MAS to make system level decisions and gets different information from every agent in the MAS and also from the operator by passing data and control messages. It analyzes and decides what kind of message should be sent to which agent (Chang et al., 2003). Therefore, the CA acts like a supervisory coordinator to decide which agent to be (in)activated based on measured information. This supervisory architecture can be applied to coordinate several locally operating controllers or agents and decides when and how to switch between them (Johansen and Murray-Smith, 1997). Although the supervisory architecture solves the problem of coordinating a set of agents, in general it is not an open solution. Adding, removing or changing any agents cannot be done without redesigning the supervisor (Van Breemen, 2001).

2.4.3 A MAS-based embedded control system design method

Ning and Yang (2006) presented a MAS-based embedded control system design method that is principally adapted from the approaches of Johansen and Murray-Smith (1997) and Van Breemen (2001) and focuses on the implementation of MAS-based embedded control systems for complex mechatronic systems in terms of software components.

The implementation of MAS-based embedded control systems can be summarized as follows: a practical system is first partitioned into some *agent-tasks* based on the controller-agent concept and MAS viewpoint. Then, by using the multitasking capability of a real-time operating system (RTOS) to realize the agent-tasks and using the system services of this RTOS to construct the coordination and communication mechanisms, a MAS can be

implemented in the form of an embedded system. The “agent-task”, which has the same role as the elementary controller-agent, is considered as the elementary unit of the MAS-based embedded system. The authors presented a transition scheme, from the elementary controller-agent to the agent-task, that involves the interfaces, operating states, state transition diagram, and functions. For implementing the controller-agency concept (i.e. the coordination and communication of a group of agent-tasks), the RTOS kernel and its system services are used. Specifically, the communication between these agent-tasks is based on *the shared memory pool (SMP)*. The SMP, which is a shared data structure created in the memory, becomes the black board model of the MAS (Ning and Yang, 2006).

The authors generalized the implementation of MAS-based embedded control systems into a design method involving three phases:

- *MAS design phase:* partitioning the entire system functions into controller-agents, thus constructing the system’s hierarchy and coordination mechanism according to the system’s property and the function requirements.
- *MAS implementation phase on RTOS:* transforming the controller-agents into the agent-tasks. This is the implementation of the agent-tasks and the coordination mechanism at software-level. Each agent-task is coded in a separated task of the RTOS, and is assigned a fixed priority based on the hierarchical structure.
- *MAS deployment phase on device:* debugging and testing of the MAS that was designed and implemented on RTOS. Modification, expansion, or optimization of the foregoing phases can be required. Finally, the MAS runing on an embedded device is made complete.

Based on this design method, the development procedure of MAS-based embedded control systems is high efficiency and clear hierarchy (Ning and Yang, 2006). We see that this work can be considered as a typical effort in order to bring the research result proposed by Van Breemen (2001) into practical applications. However, the authors have mentioned that the way to implement the coordination and communication mechanisms of MAS using the system services of RTOS (e.g. semaphore management, mailboxes, queues, time delays, etc.) is crucial and requires high embedded system development skills. Bruyninckx (2002b) has also judged that using the powerful but dangerously unstructured API (application programming interface) of an RTOS can make designers miss the chance to develop more structured, and, hence, more deterministic and more portable software systems.

2.4.4 An agent-based framework for control of multi-sensor and multi-actuator systems

Waarsing et al. (2003a) introduced a software framework for control of multi-sensor, multi-actuator systems which is based on an agent-based philosophy. This framework is closely linked to the work done by Van Breemen (2001). The authors use basic components that

they call *BasicAgents* to implement the communication protocols and function needed in a control system. The BasicAgents are coordinated and combined into an *Agency* by an appropriate *CoordinationObject*. The interface that an Agency offers to the outside is the same as the interface of a BasicAgent, thus providing the possibility of building even larger agencies from basic agents and agencies. It is similar to what Van Breemen did, the authors also defined *MultiAgentController (MAC)* component that is used to incrementally build up a control system. This software framework aimed at developing behavior-based control systems. It was used for the implementation of a door opening algorithm on a behavior-based mobile manipulation (Waarsing et al., 2003b). It was also used in the Ambience project to program the navigational algorithm of an indoor mobile robot (Van Breemen et al., 2003). It can be seen that there was a research cooperation between Van Breemen and the software framework's authors in the Ambience project. In the future, the authors have planned to develop a learning algorithm that can learn new behaviors from a small basic collection of actions. This may lead to a system that improves its own performance and learns completely new tasks (Waarsing et al., 2003a).

2.4.5 Control systems combining hybrid control and MAS

Lygeros et al. (1997) proposed *a control scheme that combines a hybrid control approach and MAS* for solving complex control systems. The approach they use is based on optimal control and game theory. At the continuous level, each agent is designed with its own optimal strategy and conditions under which they satisfy the closed loop requirements. In the discrete design, a coordination mechanism is used to resolve conflicts between the continuous designs (i.e. agents). The authors consider a hybrid design as a game between two players: one is the disturbance that enters the system; the second player is the control which is implemented by the designer. In this game, the control seeks to improve system performance while the disturbance seeks to make it worse. By setting a threshold on the cost function to distinguish acceptable from unacceptable performance, it is possible to decide that the control wins the game if the requirements are satisfied for any allowable disturbances, otherwise the disturbance wins (Lygeros et al., 1997). They distinguish two classes of disturbances: Class 1 consists of known disturbances such as unmodeled forces and torques, measurement noise, etc.; Class 2 consists of actions of other neighbour agents that is considered as uncontrollable disturbances. The authors suggested that Class 1 can be handled by classical or advanced control theory; while, Class 2 should be solved by means of communication and coordination between the agents in a MAS.

Fregene et al. (2001) developed the Hybrid Intelligent Control Agent (HICA) framework for the synthesis of intelligent controllers in problem domains which are inherently distributed and may require multi-mode and real-time control. The key idea of this framework is that *it combines hybrid control theory with MAS to create HICA agents*. Particularly, the HICA framework conceptually wraps an intelligent agent around a core that is itself a hybrid control system (Fregene et al, 2001). The authors illustrated that the HICA framework can be used as a skeletal control agent to synthesize agent-based controllers for inherently distributed multi-mode problems. In the HICA framework, a state

is represented as a control mode and a HICA agent has the same role as a controller-agent in the MACIF (Van Breemen, 2001).

2.5 Pattern-Based Design Method in Software and Control Engineering

As introduced in chapter 1, the pattern-based design method has the key role in solving the trade-off while designing MACS and providing the reusability of the design results between projects. Hence, in this section we will address important aspects of this method and introduce several typical examples that have successfully applied the pattern-based design method and design patterns.

2.5.1 Pattern-based design method in software engineering

The design pattern technique that is widely used nowadays was inspired by the work of Christopher Alexander and colleagues. He is the first person who used what he called "a pattern language" in the architectural work to obtain better design solutions. Alexander defined a pattern as "a three-part rule, which expresses a relation between a certain context, a problem, and a solution." (Alexander, 1979).

Software engineering is one of the first specialized domains that made use of design patterns. During the 1990s, pattern-based designs were successfully adapted by the software engineering community. In a software development process, a design pattern is a reusable solution for a set of particular problems. A design pattern is not a complete design that can be directly transformed into code; it is a generic description or template that shows how to solve a problem that may occur in different situations. One of the most influential works on software patterns for object-oriented systems is the contribution of Gamma et al. (1995). They stated that a design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. In a similar way, Buschmann et al. (1996) defined: "A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution". After almost 20 years of development, design patterns have been widely used in many programming languages, software systems and software development processes.

Software design patterns are usually considered as architectural building blocks and they are typically described by the following properties (Sanz and Zalewski, 2003):

- The name of the pattern.
- The problem the pattern is trying to solve.

- The context for which a pattern is designed.
- The forces that make clear the intricacies of the problem.
- The solution that describes the structure and behavior of a pattern.
- The rationale tells where the pattern came from, why it works, and why it is used.

With the same approach, Gamma et al. (1995) stated that the software pattern description contains the following parts: name and classification, intent and motivation, applicability, structure, participants, collaborations, implementation, sample code, known uses, and related patterns. In practice, the collection of these parts is optional and varies somewhat according to personal opinion of the designer and depends on a specific software project.

2.5.2 Pattern-based design method in control engineering

Through a literature review on design patterns, we see that research and application of the design pattern technique in the control engineering domain is rather rare. It is clearly proved by the little number of relevant publications in journals and conference papers. The main reason is that *pattern-based design method* is fairly new in the control theory and control engineering community. Another reason is the insufficient awareness of advantages of using design patterns. Because of that, some experienced researchers on *pattern-based control engineering* tried to highlight the main benefits of the design pattern technique to motivate the applicability of pattern-based design method in the control engineering field. Sanz and Zalewski (2003) defined the pattern-based approach as "a method of generating solutions based on existing design knowledge". They stated that pattern-based control engineering is not a control design method in the classic sense but a new way of managing and exploiting existing design knowledge for control systems, leading to better solutions. With the same judgement, Selic (1996) stated that design patterns capture proven solutions, which, if applied intelligently, can result in significant benefits in terms of productivity and reliability. In summary, researchers believe that using this method leads to control systems designed better, i.e. they are more modular, adaptable, understandable, and evolvable.

Moreover, design patterns can be used to document designs at any level in any domain (Sanz and Zalewski, 2003). This feature is especially important in the control engineering domain because control systems usually deal with practical problems that go well beyond software issues. Patterns are useful for documenting design decisions in any aspect of a control system (such as documenting the designs of controller and plant, electric and electronic systems, sensing and actuating systems).

In practice, we see that the pattern-based design method has been implicitly used to capture design solutions for control systems for years. In a popular form, many design examples (practical experience and best practices) that have been presented in classic control textbooks (Friedland, 1996; Burns, 2001; Dorf and Bishop, 2004; Nise, 2004) used by control engineers throughout the world, but they are not described by using pattern

schemata (name, context, problem, etc.). Control system designs have recently been documented by using the pattern-based design method. However, in most of the cases, designers have a tendency to address controller design issues in a certain concrete domain and to enhance the reusability of code by using reusable components and domain dependent design frameworks. In the following, we will address some typical examples.

Lea (1994) was probably one of the first authors to use design patterns in the control engineering discipline. He introduced patterns related to the observations, reinterpretations, rational reconstructions, and redesigns of Avionic Control Systems (ACSs) which is the main navigation system of an aircraft. First, he described a general operation mechanism of the ACS which collects sensor data to estimate actual state of an aircraft, computes desired aircraft state with respect to guidance modes, and performs actions that advise pilots to manipulate aircraft effectors. Next, Lea presented a general system architecture, which is composed of a set of interacting patterns (models in Lea's terminology, such as *Navigation Models, Objective Models, Error Models, Action Models, Guidance Models*). Finally, design guidelines in the form of design steps were given to build these models, which may be viewed as process patterns (Sanz and Zalewski, 2003).

The work in (Rubel, 1995) is interesting as the author presented a set of design patterns to address the layered architecture of a mechanical control system. The main idea of this work is a natural decomposition of system requirements into a layered architecture through four design patterns: Pedestal, Bridge, Symmetrical Reuse, and Elevate References to Enhance Reuse. The *Pedestal* pattern is the main one that is used to create a layered architecture in four steps. Given the layered architecture, the *Bridge* pattern is used to create a separate world for each domain. The *Symmetrical Reuse* pattern supports the reusability in layers of this layered architecture by hiding the details of how components use objects in lower layers. The *Elevate References to Enhance Reuse* pattern is used to model a relationship between objects.

Molin and Ohlsson (1996) described a pattern language in terms of design patterns related to the design of fire alarm systems. This pattern language was used to document an architecture of an object-oriented framework for a family of fire alarm systems. This framework can cover a variety of fire alarm systems ranging from small office systems to large distributed systems for industrial multi-building plants. The pattern language consists of six design patterns such as *Deviation, Point, Pool, Lazy State, Periodic Object, Data Pump*, in which the Point and the Deviation pattern are the most important. The *Point* pattern covers the abstraction of sensors and actuators, whereas the *Deviation* pattern deals with alarms, faults, and other abnormal conditions. All design patterns are built with the following parts: name, context, problem, forces, solution and related patterns.

All these above-mentioned works focus on concrete application domains. However, in order to take full advantage of the pattern-based design method, research on design patterns for control systems should cover a large range of application domains, or in other words design patterns should be domain-independent. Sanz and Zalewski (2003) stated that design patterns for control systems should focus on three issues: (i) the development of adequate methods to document the patterns, (ii) the elaboration of domain-independent pattern

languages, and (iii) the definition of control engineering processes based on design patterns. In the following, we will look at *two well-known researches on domain-independent design patterns for generic real-time control systems*.

Selic (1996) presented a high-level design pattern for generic real-time control software. This pattern, which he calls *Recursive Control*, provides a systematic method for dealing with software functions such as system start-up and shut-down, failure detection and recovery, on-line maintenance, etc.. The design pattern relies on separating the control part from the functional part, as well as on separating control policies from control mechanisms of a real-time system. The clear separation between the control aspects and service providing aspects allows each part to be defined and modified independently. The pattern has a recursive structure, which means that each functional component, just like the system as a whole, has a control interface and one or more functional interfaces. Therefore, complex functional components can be further decomposed in the same way into more simple functional components. Moreover, the design pattern can be applied recursively which means that it is applicable across a wide range of levels and scopes, starting from the highest system architectural level down to individual components (Selic, 1996). In this way, composite components can be built and therefore the hierarchical architecture of a real-time control system can be formed.

Following this approach, Selic introduced architectural patterns for real-time systems using UML as an architectural description language (Selic, 2003). He stated that there are only three fundamental architecture forms (*structural micro-patterns*); out of which, other architectures can be constructed. Three structural micro-patterns can be combined in various ways to produce different architectural patterns that suit different domains and requirements. The three structural micro-patterns are shortly described hereafter:

- The *Peer-to-Peer* micro-pattern captures the situation where two run-time entities collaborate via the communication channel between them to accomplish some joint purpose.
- The *Container* micro-pattern is used when one entity is contained within the other. A primary role of this pattern is to encapsulate its parts (i.e. the implementation) to eliminate potential dependencies between its environment and its implementation.
- The *Layering* micro-pattern addresses interaction of the adjacency of the upper- and lower-layer entities in which the upper-layer entity depends on the presence of the lower-layer entity for its proper functioning. On the other hand, the lower-layer entity can exist independently of the upper.

Selic demonstrated the usefulness of structural micro-patterns in defining architectures of complex real-time systems through the example of a typical real-time component responsible for controlling an individual communications line that is part of a complex telecom switching system. Furthermore, he stated that the combination of the *Recursive Control* pattern with three structural micro-pattern (*Peer-to-Peer*, *Container* and *Layering*) can be applied to design complex real-time systems, including heterogeneous “systems of

systems". The features make it suitable as the top-level architecture for realtime embedded systems that needs to be dynamically controlled (Selic, 2003).

We see that the work of Selic (1996, 2003) is rather similar to the one of Van Breemen (2001). The Container and Recursive Control design pattern have almost the same role as the *Controller-Agency* component proposed by Van Breemen. Selic (2003) presented an idea about a "super" controller that coordinates the operation of the individual controllers. This super controller is similar to the coordinator object in Van Breemen's framework.

Another remarkable contribution on design patterns in the control engineering domain is the work done by Bruyninckx, Soetens and colleagues at the Katholieke Universiteit Leuven in Belgium. With establishing OROCOS project (Bruyninckx, 2001), the Orocos team has set an ambitious long-term goal of building a fully generic real-time control and signal processing system. They have focused on the design of an *application-independent control kernel* that covers the needs of many more application areas (Bruyninckx, 2002a). To obtain this goal, they have used a design approach that addresses aspects such as loose coupling between components, separation between structure and functionality, event-driven interaction, etc. (Bruyninckx et al., 2003).

Continuing on this design approach, Soetens (2006) finally presented *a design pattern for Feedback Control* which can be used to structure feedback control applications based on reusable components for sensing (and estimating), planning, process control and regulation, which are common activities in any feedback controller. This design pattern serves as a design template towards feedback control applications. However, to implement advanced hybrid control applications, a supporting infrastructure is required to provide services (e.g. reconfiguration, real-time state changes, distribution, on-line inspection and interaction, etc.) for the components of the Feedback Control design pattern. In this context, the above-mentioned control kernel is used as the infrastructure. Soetens named the combination of the Feedback Control design pattern and the infrastructure the *Feedback Control Kernel*.

Soetens defines the Feedback Control design pattern as a structural design pattern that constitutes feedback control applications. It captures the application-independent structure that presents in all feedback control systems. He identifies three principal participants in the design pattern, being *Data Flow Components*, *Process Components* and *Control Kernel Infrastructure* (Soetens, 2006), which are described hereafter:

- *Data Flow Components* is the main functionality of a digital feedback control system to execute a given set of actions (i.e. sense, calculate, actuate, update) once per sampling interval. The data flow components have the responsibility to create and manipulate the data flow forming a feedback control loop, and implement control algorithms such as state space controllers, PID controllers, etc. The data flow components are classified with *five stereotypes: Generator, Controller, Sensor, Estimator and Effector*. The connection network of the data flow components is displayed in figure 2.9. The rectangular components contain activities, the oval connectors contain data and constitute the data flow. The

arrows indicate either "writes" or "reads" direction. The dashed-line command connector is only present in cascaded control architectures. The data flow activity of the participants is typically a synchronous periodic activity: the sequence of actions is deterministically executed in a (non)-distributed application; it can be run in one single thread or multi-thread of the operating system. The stereotypes are optional; therefore, it depends on the specific control application to decide how to construct the feedback control loop (Soetens, 2006).

- *Process Components* contain the execution flow that is used to present the continuous evaluation and execution of data flow components. The process components keep the role of supervisors (i.e. operating plan makers) of data flow components as they know of the presence, interface and connections of data flow components (Soetens, 2006). For example, a process component for homing the axes of a robot or machine will command the setpoint generator to set out homing trajectories and monitor the inputs or home switch events to control the homing process. A process component can read all the data flows to be able to make appropriate decisions on the execution flow level. The decisions can be active (controlling) or passive (serving). With this approach, the process components and data flow components can structure intelligent control systems.
- *Control Kernel Infrastructure* is a collection of services in which data flow and process components perform their activities. It provides a deployment environment for components, connects the data flow between components using connectors, manages component deployment and executes the activities of the components.

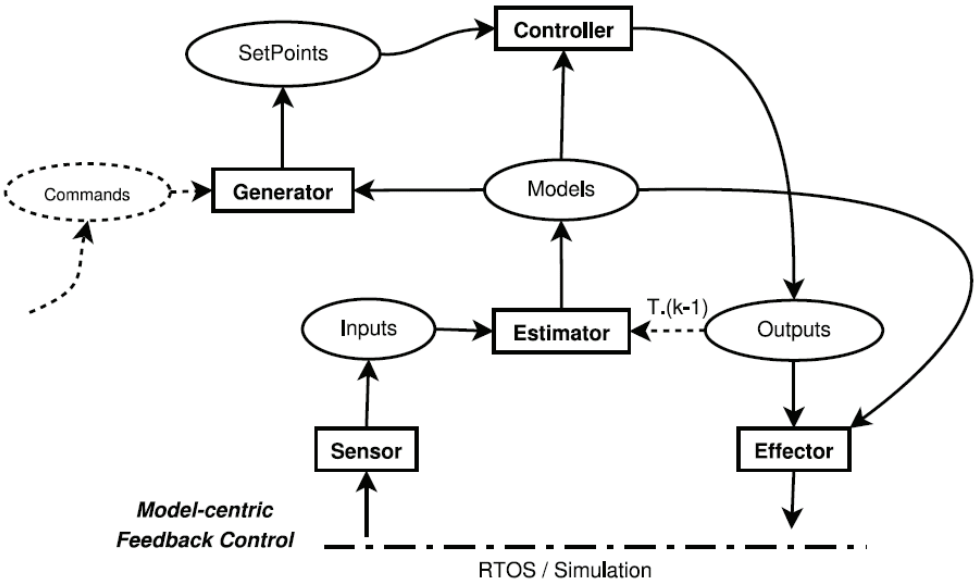


Figure 2.9 The connection network of the data flow components (Soetens, 2006)

Not like other research on design patterns in the control engineering field, the result of the Orocos project has been widely applied in a variety of control applications ranging from designing a simple feedback control for a moving mass to complex sensor-driven robot tasks. Many applications have been reported in theses and papers. Some typical applications can be mentioned here such as real-time hybrid task-based control for robots and machine tools (Soetens and Bruyninckx, 2005; Smits et al., 2008) and learning feed-forward control (Rezola, 2009).

For the ending of this review on the pattern-based design method in software and control engineering, we would like to quote the point of view of Sanz and Zalewski "The pattern way of thinking is not the search for some strange, golden entities but the continuous effort of thinking generic when designing, creating designs patterns that will last because they demonstrate wide applicability". Some useful tips should be also taken into account while working with patterns-based design: search for commonality, try to be generic, keep things simple, and choose good names because pattern names will constitute part of the vocabulary of the design team (Sanz and Zalewski, 2003).

2.6 Safety Issues in Human-Robot Interaction (HRI)

2.6.1 Introduction

The object "robots" that we will discuss is industrial robots. Hence, a concept of the industrial robots is first given. The ANSI/RIA R15.06-1999 (R2009) defines *an industrial robot* as an automatically controlled, reprogrammable, multi-purpose manipulator, either fixed in place or mobile, programmable in three or more axes, for industrial automation applications. This standard also defines *an industrial robot system* as a piece of equipment that includes the robot's hardware and software, manipulator power supply and control system, end-effectors, and any other associated machinery and equipment within the safeguarded space.

Early industrial robots and manipulators were large, hydraulically powered machines. They boasted significant strength, but no intelligence beyond their human operators (Schuster and Winrich, 2009). By the 1980s, robots were equiped with electrically driven units, which have improved accuracy and performance. Recently, increases in microprocessing power, and innovations in artificial intelligence techniques, automation and control technology have resulted in a new age of robotics, made robots mission-critical industrial tools. Today, robots operate in many kinds of applications ranging from manufacturing to medicine and in various environments including industries, domesticities, warehouses, hospitals and laboratories. Robots have been used on tasks that can endanger humans, e.g. manipulating toxic substances and working in extreme temperatures. However, without the proper precautions, robots can cause serious injuries to people and damages to equipment.

Especially, since robots have been provided with the stronger power and higher flexibility in operation, they pose more potential danger to people. Generally speaking, an unreliable robot can also be unsafe (Dhillon and Anude, 1993). Hence, a considerable amount of research has been directed toward producing more reliable and safer robots.

When a robot operates automatically without any interaction with human, there might not be any accidents. However, in tasks such as set-up programming, fine-tuning, repairs, troubleshooting, and maintenance for robots and manipulators, operators are normally required to enter the robot workspace. Accident analysis for injuries caused by industrial robots indicates that these tasks are the most dangerous activities (Jiang and Gainer, 1987) and they are the most difficult tasks to automate (Rahimi and Karwowski, 1990). In practice, many of the accidents occurred when workers placed or handled the work piece that the robot was processing or during special situations when operators needed to perform a task alongside a robot. In general, the closer the physical interaction between human and robot is, the greater the potential for human injury is. Since there will be more robot-based applications, and more tasks where the humans and robots need to cooperate in close proximity, the safety issues in HRI become more challenging. For example, the use of safety measures in HRI applications may be less efficient when a person and a robot are in close proximity. The close proximity of the robots means that there is little time to escape from crushing hazard when an accident occurs. In order to increase robot safety, De Santis and Siciliano (2008) suggested that all aspects of manipulator design, including mechanics, electronics, and control software, should be taken into account.

In the next section, we will introduce a recent study in this field. The author approached the safety issues in HRI through a practical method, i.e. learning from robot-related accidents.

2.6.2 Safety issues in HRI: learning from robot-related accidents

Malm et al. (2010) have reviewed safety issues of interactive robotics in the course of learning from robot-related accidents in Finland, where there are a total of about 6000 robots used in various applications such as handling operations, welding and assembly. The authors used the information source of 25 severe robot-related accidents from the Accident Report Database of the Safety Administration in Finland.

They studied these accidents based on the operation tasks and causes of the accidents:

- Five different operating modes: (i) troubleshooting; (ii) repairing and maintenance; (iii) production; (iv) setting, programming, adjustments, cleaning, tool change; and (v) undefined task. The review indicated that *troubleshooting* is the most common operating mode for severe robot-related accidents (48%). The accidents that occurred during production, and while an operator was programming or making settings and adjustments, were significant, with approximately 20% for each operating mode.

- Some main causes of the accidents: unexpected start-up, mishap, dangerous working method, inadequate safeguarding, inadequate design, insufficient work experience, insufficient warnings/instructions. Because each accident had several causes, the total amount of the causes is significantly higher than the amount of the accidents. The review showed that *inadequate safeguarding* was related to almost all of the accidents (80%). The insufficient warnings/instructions represented a significant cause of the accidents (60%).

The review has revealed that most of the serious robot-related accidents involved crushing a person against a rigid object (specifically, 23 of all 25 severe injuries were caused by crushing). In these accidents, the danger is critical, even at low speeds. Especially with HRI applications, these accidents are usual because of the close presence of humans to robots. Hence, the authors emphasized that special attention must be paid to the locations where a crushing possibility exists. Another noticeable information is that a significant number of accidents (64%) resulted in a variety of hand injuries.

With experience learned from this review, Malm et al. have discussed and presented some guidelines on the control issue and decisions to be undertaken to avoid robot-related accidents in HRI applications:

- Proper *safeguarding* would have prevented many of the studied accidents during production and troubleshooting. However, during programming, setting, and maintenance, the worker needs to get close to the robot, and perhaps even into its working area. The traditional safeguarding with fences and light barriers are not designed to protect a person working inside a robot cell while the robot is moving, and thus are not suitable for any kind of HRI applications. Therefore, new safeguarding methods should be developed for these special application.
- The *reaction of a robot and human after a collision* should be carefully considered. When a collision occurs, a quick stop is normally activated. This protective stop means a crushing situation continues because brakes keep the robot in the crash position. This problem leads to a question: brakes should be released or not in these situations? Since quick actions to diminish the effects of crushing hazard are important, many people approve that the brakes should be released. The authors gave a proof: “in 2 out of the 3 lethal robot accidents in Finland, the human body was in a crushing situation for a considerable period and a pressure release could have saved the persons”. This brake-released activity, however, can make the robot arm fall down, and thus causing an additional danger. The authors have discussed that it may be safe to release the brake of the first joint, however this solution does not always sufficiently minimize the risk.
- About *roles of the robot control system*, they have judged that software-based solutions for collision detection exist in some robot controllers, which can detect abnormal forces directed at the robot end-effector in a crushing situation. However, these systems are not considered as safety measures, but they have features that can lessen the damage caused by crushing. Malm et al. (2010) illustrated a useful feature: control software can make a robot move backward

along the trajectory it has moved before the crushing situation becomes worse. This trajectory may be considered as the safest direction in a generic case. The robot will automatically stop its backward movement after traveling a preset distance. Concerning with such backward movement, the authors have discussed that this involves an automated active movement after a protective stop situation. They have stated that a risk assessment may prove that a backward movement is safe and justifiable. However, it is possible that in several cases, any active safe movement could have hazardous results. Therefore, some adjustments or new interpretations to the current safety requirements and standards are needed to justify *safe movements after a collision* (Malm et al., 2010).

- Because an automated active backward movement of robot after a protective stop situation is not always safe, Malm et al. have suggested that a robot's backward movement could be activated manually using a separate safety button, pressed by a colleague. However, this feature will only work well when employees are not working alone. This is *an essential role of human supervision* in manufacturing systems. The authors emphasized that in all of the fatal accidents the victim was working alone. If a colleague had been present at the time of the accident, the consequences could have been milder.

In the next section, we introduce some typical studies in this field. The review will address research, both in the past and at present.

2.6.3 Typical research of safety issues in HRI

The study of Dhillon and Anude (1993) has been considered one of the best overall reviews relating to robot safety and reliability. The authors reviewed selective literature: over 200 articles published during the two decades 70's and 80's of the last century, which was obtained from more than 70 journals and conference proceedings of the leading world robotic associations. These papers addressed a wide range of issues ranging from safety measures relating to the design and installation of robots and software reliability to human factor considerations including the legal aspects concerning robot technology. The authors classified articles into three broad categories: Robot Reliability, Robot Safety and Maintenance, and General. The common efforts of research were directed to ensure the emergence of safer and more reliable robot systems. Dhillon and Anude acknowledged several main research topics that were studied during this time:

- Robot safety in terms of the safe design of equipment and workplace.
- Developing standard safety measures with regard to the testing, inspection, installation, and maintenance of robots.
- The safety assessment and reliability of the software that controls the robot's movements. In this topic, efforts were directed at software checking and the verification of robot programmes to ensure collision-free tasks.

- Robot sensory system development and artificial intelligence. In this direction, the goals were greater safety, faster emergency and error recovery, and enhanced collision avoidance.
- Human factor issues relating to robot use incorporating the physical, mental, psychological and legal aspects.
- Development of robotic safety standards.
- Practical examples of safety in robotic systems.
- The reliability of robot systems using methods such as Fault Tree Analysis (FTA) and Failure Mode and Effect Analyses (FMEA).

Rahimi and Karwowski (1990) reviewed critical issues in HRI, and proposed a research framework to study human aspects of robotic system design. The main emphasis was on system design whereby machine and human can interact in the context of task performance. They suggested a macro-systems approach to indicate different categories of behavior related to human and robot components. The authors presented a general human-robot model, which shows the representation of the functional performance of a human-robot system. In this model, the human physical and cognitive contributions are described through the use of a number of devices such as pendant, workstations displays, visual detection, teach pendant functions, and keyboard entries. Moreover, with an attempt to integrate the interaction between humans and robots into an activity taxonomy, they introduced so-called human-robot taxonomies. The authors expected that past and current research in this area can be mapped against this taxonomy. Hence, areas of emphasis and lack of activity can be identified for future research. Rahimi and Karwowski (1990) have defined the human-robot interaction system (HRIS) as a quintuple:

HRIS = (T, U, R, E, I) where:
T = task requirements (cognitive and physical);
U = (UC, UP} - user characteristics (UC: cognitive, UP: physical);
R = (RS, RH} - robot characteristics (RS: software, RH: hardware);
E = an environment;
I = a set of interactions.

Rahimi and Karwowski described: a set of interactions (I) contains all feasible interactions between task requirements (T), user characteristics (UC, UP), and robot characteristics (RS, RH) in some environment E. The interactions can be elemental, i.e. one to one association, or complex such as an interaction between an operator, software used to perform on-line programming to accomplish a given task (requirements of the task), and particular robot's range of motion capabilities (Rahimi and Karwowski, 1990). The elemental interaction between U and R reflects the traditional scheme of HRI applications. However, the elemental interactions may not necessarily involve the user.

In relation to the perception of arm speed and the human reaction to press the emergence stop, Collins (1989) studied the optimum button location through evaluating ten general button locations and two different button sizes on a typical teach pendant. The results have indicated that the slowest reaction time was obtained from one-half inch button located on the left hand side of the pendant, and the fastest was obtained from one-inch button located on the front of the pendant.

Helander (1988) proposed a model for monitoring and controlling robots, with the following six possible scenarios of human-robot interaction:

- robot controls task autonomously with feedback from sensors;
- robot controls task autonomously without feedback;
- operator controls robot arm directly by moving it;
- operator controls robot arm indirectly through keyboard or teach pendant;
- operator monitors task directly; and
- operator monitors task indirectly through display.

Amongst a number of research topics of safety issues in robotics, much research has been focused on the topic of *collision detection and avoidance*. In general, it is not easy to detect and/or avoid obstacles in the unstructured real environment of HRI applications. The reason is that a detailed description of the unstructured environment is difficult, if not impossible, to obtain (De Santis and Siciliano, 2008). In order to overcome this problem, some researchers have developed dynamic collision detection systems for industrial robots to avoid fatal accidents with the objective that these robots have the ability to detect collisions *without using external sensors* and hardware modifications. Yamada et.al. (1997) proposed a model-based collision detection scheme for robots. This system can detect collisions based on the comparison of the actual input torques and the reference input torques calculated from a dynamic model of the manipulator. The proposed system assumes that the dynamic parameters of the manipulator are known precisely. However, this assumption is not always satisfied for the robots used in human and industrial environment because the robot dynamics changes with its load and configuration. Matsumoto and Kosuge (2001) developed a collision detection method of a manipulator based on the nonlinear adaptive control law proposed by Slotine and Li (1988). The collision of a manipulator with its environment is detected by the difference between the actual input torques to the manipulator and the reference input torques calculated based on the manipulator dynamics. Matsumoto and Kosuge employed an adaptive control scheme for the manipulator control and the parameter estimation of the manipulator, and calculated the reference input torques by using the estimated manipulator parameters. The authors demonstrated the validity of the proposed collision detection scheme for a three degrees-of-freedom industrial manipulator. However, manipulators using these proposed collision detection methods (Yamada et.al., 1997; Matsumoto and Kosuge, 2001) could only execute positioning tasks whereas many tasks which are expected to be done by robots need to have interaction with its environment such as cooperation to handle an object with a human, opening and shutting

a door, and so on. Hence, another collision detection system, which is based on an adaptive impedance control law (Morinaga and Kosuge, 2003), was proposed that allows the manipulator to have interaction with its environment.

Using another approach, De Santis and Siciliano (2008) suggested to use reactive control *in the presence of an efficient sensory system*. The authors proved that interaction control strategies as impedance control, together with reactive collision avoidance may increase safety by means of control. They also used virtual reality for realistic simulations of HRI tasks, including collisions and injected errors.

Schuster and Winrich (2009) discussed *new software-based safety systems, supported by modern sensory and vision systems*, that have revolutionized the way robots and people interact. These safety systems can slow a robot to a safe speed, direct a robot's motion to a safe position, or bring it to a safe state, thus allowing people and robots to share the same workspace with less risk. This makes robots work collaboratively hand-in-hand and side-by-side with people. For example, a robot can lift and position a heavy sheet of metal while an operator hands weld parts onto the larger piece (Schuster and Winrich, 2009). These safety systems are supported by environmental awareness sensors and vision-based guard systems that allows robots to "see" the co-worker and other people. Hence, when a human moves closely into a workspace of a robot, the safety system triggers the robot to go into a safe position or safe state, and wait in safe mode until the human moves out of this range and an operator resets the motion. The authors presented a typical example of the vision-based guard systems, which is a new 3-D safety-rated vision intrusion system. This system can keep robots and people separate in the workspace without using traditional perimeter fencing. This electronic and programmable perimeter guarding system includes three video cameras mounted overhead in the workspace, which can detect when someone enters the hazard zone. The safety system then cautions the intruder about the danger by issuing a visual or audio warning. This system also controls robots to slow down or stop, thus helping reduce risk. When the hazard zone is clear, the robots are reset and normal operations can be safely resumed (Schuster and Winrich, 2009).

De Santis and Siciliano (2008) have stated that robots designed to cooperate with humans must fulfil the requirements of conventional robot systems and applications: fast motions and absolute accuracy, without external sensing, provided that the operational environments are perfectly known; additionally, they are required to meet the optimality criteria, *safety and dependability*. These criteria have been considered as the keys for direct interaction, and the way to successfully introduce robots into the human and industrial environment (Yamada et.al., 1997; Heinzmann and Zelinsky, 2003; Ikuta et.al., 2003; Kulic and Croft, 2005; De Santis and Siciliano, 2008).

In safety critical systems like HRI, where dependability is a legal requirement, robots will require certification before they can be used (Harper and Virk, 2010). Standards play an important part in this process since they provide safety requirements and design assurance guidelines for manufacturers and users. In the next section, this issue will be addressed.

2.6.4 Robot safety standards

In this section, we introduce two well-known safety standards in robotics and then discuss if these safety standards are suitable and can be applied for HRI applications:

- The ANSI/RIA R15.06-1999 (R2009) is the American National Standard for Industrial Robots and Robot Systems - Safety Requirements, which is first published in 1999, According to the information published on the website of the ANSI/RIA R15.06-1999 (R2009): this standard provides requirements for industrial robot manufacture, remanufacture and rebuild; robot system integration and installation; and methods of safeguarding to enhance the safety of personnel associated with the use of robots and robot systems. The second review (released in 2009) further limits the potential requirements for any retrofit of existing systems, revises the description of control reliable circuitry, and reorganizes several clauses to enhance understanding.
- The ISO 10218 (International Organization for Standardization: Robots for Industrial Environment - Safety Requirements) has two parts: (i) ISO 10218-1 (2006), entitled "Part 1: Robot", published in 2006, is the initial standard. This part specifies requirements and provides guidance for the assurance of safety in design and construction of the robot itself, not the entire robot system (Schuster and Winrich, 2009). It is intended to be fully compliant with the European Machinery Directive (2006/42/EC) and expected to replace the existing EN775 standard (Manipulating Industrial Robots - Safety). (ii) ISO 10218-2 (2008), entitled "Part 2: Industrial Robot Systems and Integration", which is undergoing development and is expected to be published in 2011. This second part covers the integration and installation of a robot system, thereby providing a more comprehensive set of requirements for robot safety. It is intended to address robot workplace safety requirements and is directed more to the end-user than the manufacturer (De Santis and Siciliano, 2008).

HSG43 (2000) is not a safety standard, but a guidance to the safeguarding of industrial robots, which is published by the Health and Safety Executive (HSE). This guidance includes a list of relevant standards, thus providing a wealth of useful guidance in applying safety standards for both manufacturers and users of industrial robots. According to Pilz Automation Technology (2007), the HSG43 covers aspects: safety during installation, commissioning, testing and programming, as well as during use and maintenance. Other topics range from the principles of safeguarding robot systems and safety at the design stage, to hazard identification, risk assessment, training and interfacing with the robot controller. There is also a useful appendix with seven case studies and another that outlines the relevant health and safety laws. However, the HSG43 was published in 2000 so it does not include all of the latest standards.

The draft ISO 10218-2 has fairly directed safety issues in HRI. It defines a collaborative workspace as a "workspace within the safeguarded space of the robot work cell, where the robot and a human can perform tasks simultaneously during production operation". It also

addresses the safety principles for collaboration. However, De Santis and Siciliano (2008) have remarked that the modification of this draft standard is not as effective as desired. It specifies human-robot cooperation, however, with prescribed limits with respect to speed and power of robot. Moreover, the cases when robots and people have to share the operational workspace are not clearly discussed. Actually, the standard poses human-robot segregation in the workplace as the way to obtain safety (De Santis and Siciliano, 2008).

Schuster and Winrich (2009) have judged that current robot safety standards do not cover and meet the increasing sophistication, complexity and needs of robotic systems. De Santis and Siciliano (2008) remarked that the international standards for robotics do not address directly the safety in HRI. Malm et al. (2010) also indicated: currently, no international standards specifically address HRI and therefore developers need to apply industrial robot standards. Hence, stakeholders in the robotics and automation industries have promoted to establish new international safety standards for robots and robot systems integration through the International Organization for Standardization (ISO). Establishing the ISO standard that provides detailed guidelines to the safe design, production, and application in HRI will open the door for HRI applications around the world. Specifically, this effort is addressed in detail in the paper titled "Towards the development of international safety standards for human robot interaction" by Harper and Virk (2010). The authors provide the necessary view on new safety standards for the future of HRI. They present a survey of the work being performed by the ISO committee TC184/SC2: Robots and Robotic Devices. This committee has performed two main works: (i) developing safety standards for robotic applications in personal and medical care, and (ii) revising existing industrial robot standards with requirements for new applications. The new standards should meet the need for safety guidelines for new HRI applications in which extensive human-robot interaction behavior is present much more than previous generations of industrial robots (Herrmann and Melhuish, 2010).

Some remarks about safety issues this thesis is going to handle:

1. We limit ourselves to the safe-guarded control of mechatronic systems (e.g. robots and manipulators). Particularly, we consider mechatronic systems in the form of two classes: simple and complex system. The simple mechatronic systems are motion systems with one degree-of-freedom (1-DoF). An example of this class is the DemoLin setup (see section 4.3.1), a simple single-axis electro-mechanical motion system, which has the dynamic behavior of motion systems with the dominant compliance in the transmission. The complex mechatronic systems are motion systems with multiple degrees-of-freedom (N-DoF). The TriPod setup (section 4.4.1) will be used as an example of this class. It is a three-axis electro-mechanical motion system.
2. In such a N-DoF motion system, it always exists the strong coupling between axes (i.e. between the joint spaces of axes) as well as between the end-effector space and these joint spaces. The coupling involves the analysis, design and realization of mechanical, electrical, and control software components for each axis and for the whole machine. Especially, this coupling presents clearly in controller design

issue for a robot or manipulator with more than one axis; the coupling between multiple axes can be seen as disturbances with respect to the SISO control system of each axis. Hence, there is also a coupling between the safe-guarded control of a single axis and the one of multiple axes. How to properly handle this coupling will be one of research topics of this thesis.

3. However, as mentioned in chapter 1, the error/fault detection itself is not the topic of this thesis; it is assumed that faults are reliably detected. Moreover, this thesis is not going to develop (advanced) control algorithms to control robots, or to detect and avoid collision. The main point is that dangerous or serious situations, which can be detected based on methods and techniques, for example, that we have reviewed in this section, will be properly dealt with by using control system design patterns developed in chapter 4. Specifically speaking, this research will develop a pattern-based safe-guarded MACS design method that provides highly generic safe-guarded control patterns suitable for mechatronic systems with various levels of complexity.

Chapter 3

An OROCOS-Based Implementation Framework for MACS

3.1 Introduction

As discussed, the divide-and-conquer approach (Johansen and Murray-Smith, 1997) has been considered as one of the best strategies to deal with complex control problems. It is also judged that the multi-agent system is a good realization of this divide-and-conquer approach. Additionally, the MACIF (Van Breemen and De Vries, 2000; Van Breemen, 2001) has proven to be a good solution to synthesize the hierarchically structured MACS with an open architecture. However, the MACIF still faces some shortcomings that give room for improvement. Dealing with the shortcomings leads to the two research questions formulated in section 1.2.2 which will be the main topic of this chapter.

> The 1st research question: How to provide the deterministic real-time multi-threaded control behavior, thread-safe and real-time IPC mechanism, and the capability of handling events for the MACIF?
>
> The 2nd research question: How to improve the capability of developing the hierarchically structured MACS with an open architecture of the MACIF?

In this chapter, we will develop a new implementation framework for MACS that inherits and improves the advantages of the MACIF and simultaneously provides the missing features for the MACIF.

To have a broad view about the MACS development issue, we first study and evaluate some possible solutions that can be applied to develop real-time MACS (see section 3.3). The evaluation is based on three aspects: (i) the method used to implement a MACS using the concepts (such as controller-agent, coordination object, controller-agency, etc.) and operation mechanisms of the MACIF, (ii) the deterministic real-time communication and control behavior provided by the solution, and (iii) the MACS architecture type, i.e. centralized or decentralized, supported by the solution. This evaluation points out the best potential solution and gives us ideas to realize the selected solution (see section 3.4).

The approach to solving the 1st research question: bring the advantages of the MACIF and the OROCOS framework (Orocos, 2009a) together. As a result, a new implementation framework named **ORO**COS-based Implementation Framework for **MACS** (*OROMACS framework*) will be designed (see section 3.5, section 3.6 and section 3.8). The OROMACS framework is developed based on a functional combination between the MACIF with the Real-Time Toolkit (RTT) of the OROCOS framework.

The approach to solving the 2nd research question: deploy the port-based polymorphic modeling approach (De Vries et al., 1993; De Vries, 1994) into the OROMACS framework such that it makes controller-agents and the whole MACS design polymorphic, i.e. we obtain *polymorphism in the specification and realization of MACS*. As a result, the capability of developing the hierarchically structured MACS with an open architecture is well enhanced (see section 3.7 and section 3.9).

This chapter is organized as follows. In section 3.2, we summarize the central idea of coordination and main contents of the MACIF to give a background for the next sections. For more information about the MACIF and coordination, the readers are recommended to read the PhD thesis of Van Breemen (2001) and the papers (Van Breemen and De Vries, 2000; Van Breemen and De Vries, 2001). Next, section 3.3 presents an evaluation and comparison between several feasible approaches that can be used to develop real-time MACS. In section 3.4, we address the resemblance between an elementary controller-agent of the MACIF and a TaskContext component of the OROCOS framework. In section 3.5, we describe the proposed combination between two frameworks by means of mapping the elementary controller-agent into the extended state transition diagram of the TaskContext component. Next, section 3.6 explains how a composite controller-agent is formed in the OROMACS framework. In section 3.7, we present an extension of the port-based polymorphic modeling approach into the OROMACS framework and then discuss polymorphism in the specification and realization of MACS. In section 3.8, we introduce two concepts being OROMACS TaskContext and OROMACS Root-Agent. Section 3.9 discusses roles of the polymorphism approach and coordination. Finally, some concluding remarks are given in section 3.10.

3.2 Multi-Agent Controller Implementation Framework

3.2.1 Central idea of coordination

In this thesis, we only consider agent-based systems with explicit coordination mechanism. Coordinating controller-agents involves making decisions about which controller-agents should be (in)active at a particular moment. In general, decision making consists of two parts (Van Breemen and De Vries, 2001): first, *arguments* need to be collected through some form of communication, and secondly, *a judgement* needs to be made by interpreting the arguments. A judgement is generally made at some central place that has authority. However, the arguments may be formulated in a centralized or decentralized way (see figure 3.1). This results in two general types of coordination architectures:

- *Centralized coordination (or supervisory)*: a single planner or supervisor decides when controller-agents should become (in)active. It formulates the arguments based on information about states, inputs and outputs of the controller-agents involved and makes a judgement. As a supervisor has an overview of the whole system, this coordination architecture may result in the global optimal behavior. However, a disadvantage of the centralized coordination is that it often results in a closed architecture in which it is difficult to add or remove controller-agents.
- *Decentralized coordination:* each controller-agent decides locally whether it wants to become active, but the final decision whether a controller-agent becomes (in)active is made by a central judgement-making process. This decentralized coordination architecture may result in local optimal behavior, but creates an open environment in which it is easy to add, remove and modify controller-agents.

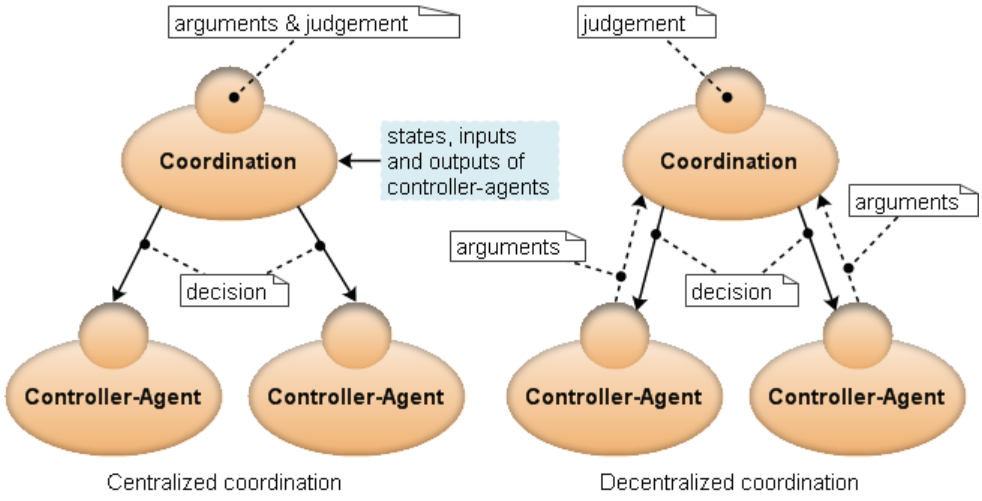


Figure 3.1 Two general types of coordination mechanisms (adopted from Van Breemen and De Vries, 2001)

The Multi-Agent Controller Implementation Framework (MACIF) proposed by Van Breemen (2001) *currently uses the decentralized coordination architecture* that is described as following: a multi-agent control systems (MACS) includes a group of controller-agents and a coordinator. A controller-agent is a locally operating controller that encapsulates a local control algorithm and a local operating regime. The local operating regime represents the self-intention of the controller-agent. A coordinator handles the dependencies between controller-agents. It decides, based on the intentions of the controller-agents, which one has to be (in)actived. A second responsibility of the coordinator is to blend (if necessary) the control actions of the controller-agents involved. It means that the operating regimes of controller-agents and control decision logic of the MACS are cooperatively manipulated by the coordinator and controller-agents as well.

Hereafter, some coordination mechanisms are introduced in which the first five types will get most attention in this thesis.

- *Master-Slave* is a subordination dependency in which the Slave-controller-agent depends on the Master-controller-agent, i.e. the Slave can be active only when the Master is active.
- *Fixed-Priority*: each controller-agent within a group is assigned a fixed priority. The operation of controller-agents is determined relying on their priority levels. The controller-agent with the highest priority which wants to become active, becomes active. If a controller-agent with a lower priority was active, this one gets inactive. It means that, only one controller-agent within a group of controller-agents may be active at a particular moment.
- *Parallel*: all controller-agents within a group can be concurrently active. Hence, activity of a controller-agent is independent of other controller-agents.
- *Sequential* makes controller-agents within a group active in succession, for a single round only.
- *Cyclic* makes controller-agents within a group active in succession repeatedly.
- *First Stays Active*: the first controller-agent that wants to become active, will become active. It stays active until it decides by itself to get inactive. No other controller-agent can become active until the first becomes inactive.
- *Last Stays Active*: the last controller-agent in timing aspect that wants to become active, will become active. If there is a previously active controller-agent, this controller-agent is inactivated.
- *Dynamic Priority*: each controller-agent within a group determines its own priority by sending an activation intention signal to the coordination object. The controller-agent that sends the signal with the highest value will become active.
- *Addition*: each controller-agent may become active independently from the other controller-agents within a group. The control actions of the active controller-agents are added. Other extension forms of the Addition coordination mechanism are the Weighted Addition and Fuzzy Logic Addition (Van Breemen, 2001).

For more information about the coordination mechanisms, the readers are recommended to read the PhD thesis of Van Breemen (2001).

3.2.2 The MACIF

In the MACIF, there are six functional components used to construct a MACS. The components are classified into two types: *the composite components* which contain other components and *the elementary components* which do not consist of other components.

- *Elementary controller-agent* is a component that implements a locally operating controller.
- *Coordination object* is a component that implements a coordination mechanism.
- *Controller-agency* is a composite component that consists of a coordination object and a group of controller-agents.
- *Sensor component* converts a value read from outside world, e.g. an AD-converter, into a meaningful number.
- *Actuator component* converts a meaningful number into a value that is written to outside world, e.g. an DA-converter.
- *Multi-agent controller* (or the main controller-agent) is a composite component that implements the overall controller and consists of sensor, actuator, and controller-agents, as well as one coordination object.

The class diagram and relations between the six different components are shown in figure 3.2 and figure 3.3. The multi-agent controller and controller-agency are composite components because they consist of other components. These functional components of the MACIF are defined in terms of attributes, methods, subcomponents, and connections:

- *Attributes* are particular variables that a component may have. Four types of attributes, which are input ports, output ports, parameters and state variables, can be specified for each component.
- *Methods* are functions that are executed by a controller-agent.
- *Subcomponents* are only present in the main controller-agent or controller-agency. A subcomponent can be an elementary controller-agent or a controller-agency.
- *Connections* are made between ports of components to exchange data.

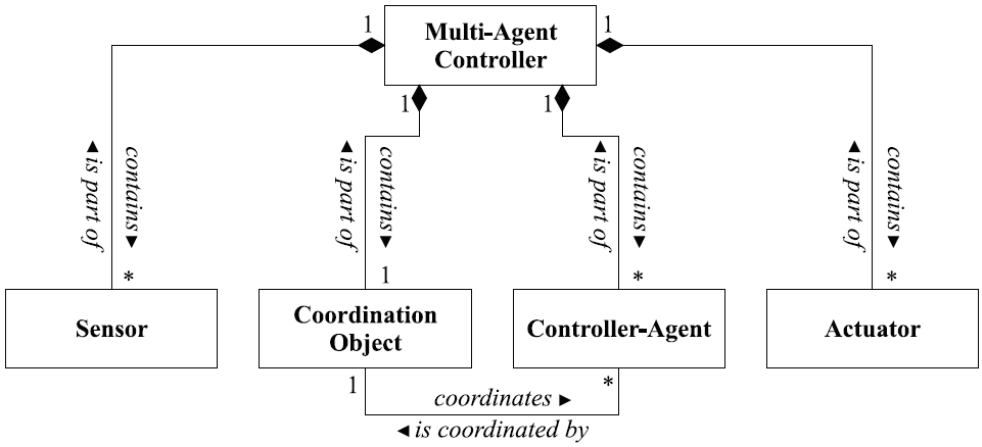


Figure 3.2 Class diagram of a multi-agent controller (Van Breemen, 2001)

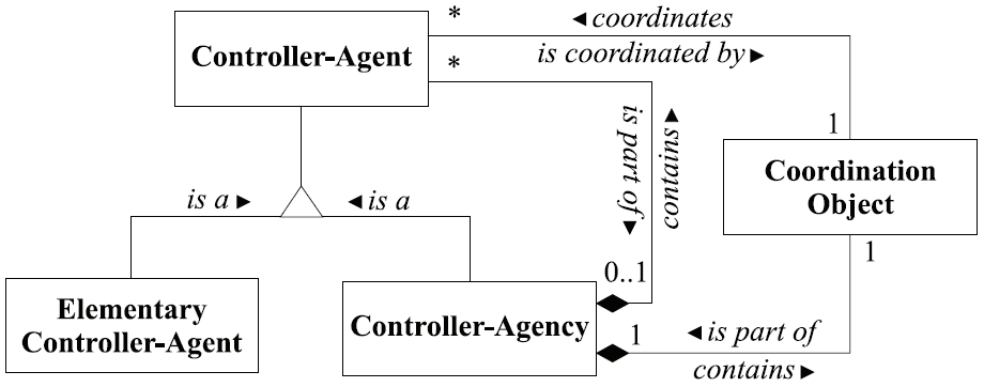


Figure 3.3 Class diagram of a controller-agency (Van Breemen, 2001)

There are two information flows between components of a multi-agent controller:

- *Message flow:* messages are sent and received among controller-agents and also between a coordination object and coordinated controller-agents. Based on passing messages, particular methods of components are executed and hence the messages can be considered as a control flow. The message/control flow between components is fixed and provided as a ready-to-use communication service, thus need not be defined again when implementing a multi-agent controller.
- *Data flow:* the connections between input and output ports are used to exchange data between controller-agent components, thus forming the data flow of a multi-

agent controller. The data flow is specified by the designer while making connections between ports of components.

3.3 Evaluation of MACS Development Solutions

3.3.1 Introduction

In this section, we are going to evaluate four possible approaches to develop real-time MACS using concepts and operation mechanisms of the MACIF:

1. Use a general programming language.
2. Use an agent-oriented programming language.
3. Use the Matlab-Simulink software, Stateflow toolbox and Real-Time Workshop (The MathWorks Inc., 2009).
4. Use the OROCOS framework.

The evaluation is based on three criteria:

1. How easy- and ready-to-deploy is the solution to implement MACS using the operation mechanisms and concepts of the MACIF? This criterion is considered as a basic requirement; thus it will be evaluated first.
2. How good and ready-to-use is the deterministic real-time communication and control behavior provided by the solution?
3. How good is the support for the decentralized coordination architecture?

3.3.2 Solution using a general programming language

This solution uses a programming language such as C/C++ to develop a MACS. It means that the developer must do everything from scratch, i.e. this solution does not provide any ready-to-deploy feature to implement the operation mechanisms and concepts of the MACIF. Moreover, since this solution is not a high-level development method, the work will be laborious. The reason is that the way to specify such concepts of the MACIF by means of direct programming is freedom but also repetitive, tedious, time consuming, and prone to errors, thus requiring a lot of debugging. A general programming language also provides no design support, such as semantic model checks, validation of designed MACS structure and correctness of inter-connections between controller-agent components of the agent-based control system. Finally, this solution requires extensive programming skills and experience from the designer.

3.3.3 Solution using an agent-oriented programming language

In the research community of multi-agent systems, the following are agent-oriented programming languages, integrated development environments and frameworks (platforms) which have been widely used in the design, implementation, and deployment of agent-based applications:

- The FLUX (Thielscher, 2005; FLUX website, 2010) is a high-level programming system for cognitive agents of all kinds, including autonomous robots.
- The CLAIM - **C**omputational **L**anguage for **A**utonomous, **I**ntelligent and **M**obile Agents (Seghrouchni and Suna, 2004).
- The JAL - **J**ACK **A**gent **L**anguage (Howden et al., 2001; JAL website, 2010) with support of the JDE - **J**ACK **D**evelopment **E**nvironment (Winikoff, 2005).
- The 3APL - **A**n **A**bstract **A**gent **P**rogramming **L**anguage (Hindriks et al., 1999; 3APL website, 2010) with support of the 3APL-IDE.
- The AF-APL (**A**gent **F**actory - **A**gent **P**rogramming **L**anguage) (Collier, 2002) with support of the Agent Factory Framework (Agent Factory website, 2010).
- Jason IDE (Bordini et al., 2005; Jason website, 2010) is a platform for programming agents in AgentSpeak (Rao, 1996), a logic-based agent-oriented programming language.
- The FIPA (**F**oundation for **I**ntelligent **P**hysical **A**gents) is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies (FIPA website, 2010).
- The JADE - **J**ava **A**gent **DE**velopment (Bellifemine et al., 2005; JADE website, 2010) is a middleware agent platform which supports the development of distributed intelligent multi-agent applications in compliance with the FIPA specifications.
- The Jadex Agents (Braubach et al., 2005; Jadex website, 2010) is a software framework for programming intelligent software agents in XML and Java and can be deployed on different kinds of middleware such as JADE.

Agent-oriented programming languages normally provide a high-level development method for expressing the high-level abstractions associated with agent-based system design. They provide structures to specify agents and support to distribute the agent-based system over different hardware platforms, which is beneficial if the target hardware consists of multiple (heterogeneous) processors. However, almost all agent languages provide agents with a fixed architecture, or aim at programming for logical reasoning systems. Therefore, the concepts like controller-agent, coordination object and controller-agency can not be directly specified (Van Breemen, 2001).

Agent frameworks or platforms are created to simplify the development of agent-based applications by providing an infrastructure where agents can live and operate. The infrastructure consists of necessary management services for hosting agents in a uniform environment and additionally supply ready-to-use communication services for the agents. Technically, an agent-oriented platform is represented by a programming language for realizing agents and the available tools for development, administration and debugging. However, almost all of the agent platforms focus on business applications, possibly run on the internet, and so that the deterministic real-time behavior is hard to guarantee.

So we can conclude that in comparison with the solution using a general programming language, the solution using an agent-oriented programming language has more advantages of deploying the MACIF. However, this solution still cannot directly specify the concepts and operation mechanisms of the MACIF. Moreover, the deterministic real-time behavior provided by this solution is not ready-to-use.

3.3.4 Solution using Matlab-Simulink, Stateflow toolbox and RTW

In this part, we will discuss that the combination of Matlab-Simulink software environment, Stateflow toolbox, and Real-Time Workshop (RTW) might be a possible solution to develop real-time MACS using the operation mechanisms and concepts of the MACIF. We refer to this solution shortly as *Stateflow-based MACS* that has some main advantages:

- This solution supports the development of MACS with *the centralized/supervisory coordination architecture* that consists of one or more Stateflow components and several Simulink blocks. Control algorithms are situated inside Simulink blocks which are considered as local controllers (or controller-agents). A Simulink block can be either an elementary controller-agent or a controller-agency. Stateflow components keep a role of the supervisory coordination objects (i.e. coordinators) which decide every sampling period when and how to switch between the local controllers based on user command and input information. We call this coordination mechanism the *Stateflow-based supervisory coordination*. Therefore, this solution enables the concepts of the MACIF to be usable at a basic ready-to-deploy level. However, the operation mechanisms and decentralized coordination architecture of the MACIF are not supported.
- A controller-agency or composite controller-agent is formed through grouping some Simulink blocks to create a pool of controller-agents that are coordinated by a Stateflow-based supervisory coordinator. Hence, the Stateflow-based solution can be applied to develop hierarchically structured control systems.
- Matlab-Simulink provides the designer with analysis and design toolboxes that support the modeling of plant dynamics and physical systems, the design and simulation of control algorithms of local controllers (i.e. Simulink blocks) of MACS. To deploy the designed Stateflow-based MACS in the form of software code that can be executed on computer, the RTW can be used to automatically

generate stand-alone C code from the Simulink models. To generate C code from Stateflow models, the Real-Time Workshop Embedded Coder is used as an add-on product with regard to the RTW. The resulting code can be used for real-time and non-real-time control applications. Hence, the deterministic real-time behavior provided by this solution seems to be good and ready-to-use.

Besides these advantages, this Stateflow-based MACS development solution also has some disadvantages:

- The Stateflow-based solution uses the centralized coordination architecture therefore the openness of MACS designs is limited. Whenever a certain controller-agent (i.e. a Simulink block) is added, modified or removed, the designer needs to modify the design, i.e. ports, connections between Simulink blocks, the internal behavior of the Stateflow-based supervisory coordinator.
- A coordination mechanism (such as Master-Slave, Sequential, FixedPriority or Parallel, etc.) of a controller-agency in the Stateflow-based MACS is formed by using the state transition rules which are based on the complex combination of state positions, geometry of the outgoing transitions, clockwise progression, transition-label writing syntax, etc. It is not easy to understand and debug in case of errors. Moreover, when the designer wants to change the coordination mechanism, for example from Sequential to FixedPriority, then almost all parts of the current design need to be redesigned, particularly the state transition rules inside the Stateflow-based supervisory coordinator of the controller-agency.
- In the Stateflow-based solution, it is hard to support reusable coordinator components and library-based design. The reason is the strong coupling between the Stateflow-based supervisory coordinator and Simulink blocks. A coordination mechanism is always designed based on operational logic or switching condition between Simulink blocks, which depends on each specific control problem. It means that the same coordination mechanism, when it is used for two controller-agencies, may be implemented differently. Hence, the Stateflow-based supervisory coordination is an application-dependent mechanism. As a result, the coordination types such as Master-Slave, Sequential, or FixedPriority cannot be generalized to be reusable between applications. A possible solution for solving this disadvantage is to modify the Stateflow toolbox such that it can be used to build the reusable generalized coordination types. However, Matlab-Simulink and Stateflow toolbox are commercial products so that they are not open to accept any modification. Additionally, the strong dependence upon the Matlab-Simulink environment is also a negative point of this solution.

A typical application that uses Matlab-Simulink and Stateflow toolbox to develop MACS is the research on developing an architecture of multi-agent control system applied to Fossil-Fuel Power Unit (Chang et al., 2003; Masina et al., 2004). Detailed information and discussion about this research is given in section 2.4.2.

3.3.5 Solution using the OROCOS framework

An overview of the OROCOS framework is given in (Orocos, 2009a). In this part, we focus on an overall evaluation with respect to the OROCOS framework and particularly to the RTT to study the advantages and disadvantages and to assess the feasibility of this solution.

Firstly, the RTT and OROCOS framework offer users several advantages:

- The OROCOS framework supports *a generic feedback control architecture*. It allows the designer to create feedback control systems for advanced robotics and mechatronic systems from basic components such as feedback controller, feedforward controller, path generator, estimator (filter), sensor and actuator. Building of these basic components is based on C++ libraries of the Real-Time Toolkit. An example is presented in (Rezola, 2009) in which he developed a control system for a moving mass plant model that consists of a path generator, a PID controller, and a time-index learning feedforward controller.
- The RTT of the OROCOS framework *can be combined easily with several hard real-time targets or platforms* such as RTAI/LXRT (www.rtai.org) or Xenomai (www.xenomai.org) to develop multi-threaded control systems with performances such as deterministic real-time behavior and reliable thread safety. It means that the OROCOS framework could be a good run-time environment for the MACS.
- The OROCOS framework is *computer platform and application independent*. Hence, it can be used with different operating systems (e.g. Linux OS, Window OS, and Mac OS X, etc.) and it is applicable to develop control systems for various kinds of mechatronic systems.
- The OROCOS framework has been maintained and developed in the form *a free software project*, i.e. source code and documentation are released under a Free Software license, allowing free use for academic as well as industrial applications. The OROCOS project has a ambitious, long-time target and it is currently under development. Therefore, OROCOS-based applications can be efficiently supported by users and the development community.

Secondly, there is a resemblance between an *elementary controller-agent* of the MACIF and a *TaskContext component* of the OROCOS framework. This resemblance is presented through three aspects hereafter. A detailed description about the resemblance will be presented in section 3.4.

- They are *basic primitives* of the frameworks that constitute either the MACS or OROCOS-based control system.
- In the MACIF, the operation of an elementary controller-agent is managed through a set of *user functions* such as start(), initialize(), activation(), calculate(), update(), finalize(), and stop(). In the OROCOS framework, the operation of a TaskContext component is controlled by a list of *user 'Hook' functions* such as configureHook(), activateHook(), startHook(), updateHook(), stopHook(),

cleanupHook(), resetHook(), and errorHook(). We see that there are functional equivalences between the user functions and the user 'Hook' functions.

- In the MACIF, an elementary controller-agent has *two operating states* (Active and Inactive). In the OROCOS framework, a TaskContext component has a finite state machine which consists of *six operating states* (Init, PreOperational, Stopped, Active, Running, and FatalError). We also notice that there are some functional equivalences between the operating states of an elementary controller-agent and a TaskContext.

Additionally, both frameworks have strong and weak points which can reciprocally complement each other through a functional combination between two frameworks:

- *The MACIF enables to create hierarchically structured control systems* that consist of several elementary and composite controller-agents coordinated by a coordinator. In the OROCOS framework, it is currently not possible to do this because the construction of a composite component is not supported.
- *The OROCOS framework supports a generic hard real-time kernel* that can be used to build control applications with deterministic (hard) real-time behaviors. Therefore, it is an ideal complement for the MACIF.

Finally, the MACIF and OROCOS framework use the same implementation approach in some aspects hereafter:

- *Modularity and reusability:* both MACIF and OROCOS have followed the same design approach in which a control system is developed based on modular and reusable components.
- *Decoupling between components:* both frameworks can be used to develop control systems in which the implementation of one component does not rely on knowing something about the internal structure and implementation of another component. Decoupling between components makes the designed control system easily scalable. The decoupling is assured by the port-based communication mechanism that is supported in both MACIF and OROCOS.
- *Port-based communication:* the OROCOS framework supports various kinds of communication mechanisms between components such as port-based data flow, commands, methods and events. Whereas the MACIF uses the port-based approach to make connections between input and output ports of controller-agents, thus also forming the port-based data flow inside the MACS.

3.3.6 Concluding remarks

According to the evaluation with regard to four possible solutions that can be used to develop real-time MACS using the concepts and operation mechanisms of the MACIF, we come to some conclusions:

- The solution using either a general programming language or an agent-oriented programming language cannot directly realize the concepts and operation mechanisms of the MACIF. Moreover, the deterministic real-time behavior provided by these solutions is not ready-to-use. Hence, they are not used.
- The solution using Matlab-Simulink software environment, StateFlow toolbox and Real-Time Workshop is a candidate to develop multi-controller systems operating based on the switching control strategy. The deterministic real-time behavior provided by this solution is acceptable. However, instead of using the decentralized coordination architecture, the supervisory/centralized coordination mechanism is used to coordinate controller-agents. Therefore, it results in control systems with weak openness, reduces the scalability and limits the reusability in the design process, and produces over-complicated supervisors. With this solution, the concepts of the MACIF can be specified but only in context of the supervisory architecture. Moreover, the operation mechanisms and decentralized coordination architecture of the MACIF are not directly deployable. In comparison with the solution using the OROCOS framework, this solution has more disadvantages and is thus not being used.
- The solution using the OROCOS framework brings the best features of both frameworks together. The resemblance between the elementary controller-agent and the TaskContext component opens the possibility to deploy the concepts, operation mechanisms and decentralized coordination architecture of the MACIF into OROCOS. Moreover, besause this solution uses the RTT of the OROCOS framework as the run-time environment for the MACS, the deterministic real-time communication and control behavior provided by this solution is good and ready-to-use. As a result, this solution is chosen to develop a new implementation framework for MACS.

3.4 Controller-Agent and TaskContext

3.4.1 Operation mechanism of an elementary controller-agent

In the MACIF, a Multi-Agent Controller (MAC) starts operating or running when it receives a start() message. This start() message is then forwarded to all subcomponents of the MAC (such as elementary controller-agents, controller-agencies, coordination objects,

sensor-agents, and actuator-agents) to execute their start() functions. Subcomponents start running afterward. The same pattern is applied for passing the stop() message in the MAC. While an elementary controller-agent is running, it has two operating states: *Active* and *Inactive*. The transitions from one operating state to another are controlled by an operating regime evaluation object (i.e. a coordination object), by use of an acknowledge variable, denoted by *ack* (Van Breemen, 2001). Figure 3.4 illustrates the switching behavior of an elementary controller-agent through a state transition diagram. It can be seen that after getting started, the elementary controller-agent starts running and initially it enters the Inactive state. Hereafter, definitions of the Active and Inactive state are given.

- *Active* is the state in which an elementary controller-agent is *running* and executes the *calculate()* function to calculate control samples and then the *update()* function to update the (local) state variables.
- *Inactive* is the state in which an elementary controller-agent is *running* and only executes the *update()* function to update the (local) state variables.

While an elementary controller-agent *is running and in the Inactive state*, it can:

- stay in Inactive state if the coordination object returns *ack = false*;
- switch to the Active state if the coordination object returns *ack = true*. While switching from the Inactive to Active state, the initialize() function is executed.

While an elementary controller-agent *is running and in the Active state*, it can:

- stay in Active state if the coordination object returns *ack = true*;
- switch to the Inactive state if the coordination object returns *ack = false*. While switching from the Active to Inactive state, the finalize() function is executed.

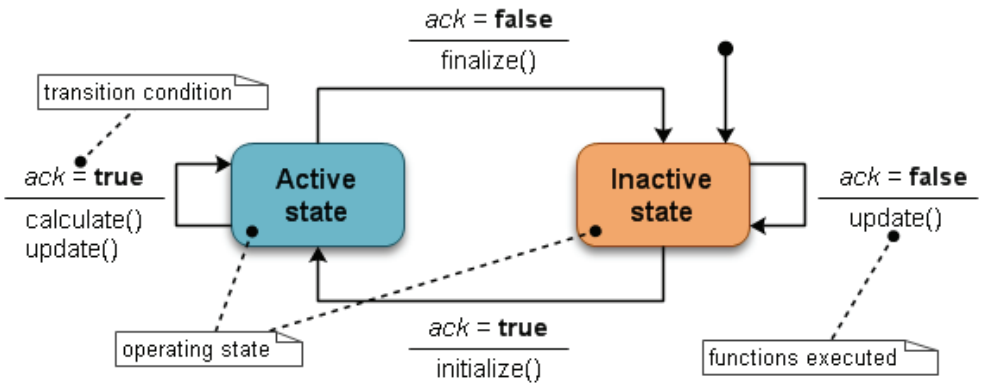


Figure 3.4 Switching behavior of an elementary controller-agent (adopted from Van Breemen, 2001)

In the MACIF, each elementary controller-agent has seven user functions whose role and meaning are explained hereafter.

```
void start()
{
    Piece of code is executed while the elementary controller-agent is being started.
}

void initialize()
{
    Piece of code is executed while the elementary controller-agent is running and
    switching from Inactive to Active state (i.e. becomes active).
}

real activation()
{
    Piece of code determines the activation intention of the elementary controller-agent.
    - if return value > 0.0, the controller-agent wants to be active.
    - if return value = 0.0, the controller-agent wants to be inactive.
}

void calculate()
{
    Piece of code is executed on every sampling interval while the elementary controller-
    agent is running and active.
}

void update()
{
    Piece of code is executed while the elementary controller-agent is running, both when
    it is in the Active and Inactive state, to update its state variables.
}
void finalize()
{
    Piece of code is executed while the elementary controller-agent is running and
    switching from Active to Inactive state (i.e. becomes inactive).
}

void stop()
{
    Piece of code is executed while the elementary controller-agent is being stopped.
}
```

3.4.2 Operation state diagram of a TaskContext component

In the OROCOS framework, components are implemented by subclassing the C++ TaskContext class that defines the "context" in which application-specific tasks are executed. Hence, an Orocos component is also called a TaskContext component (see figure 3.5). Each TaskContext component is described through five Orocos primitives: *Attributes and Properties, Commands, Methods, Events and Data-Flow ports*. The interface between TaskContexts is implemented through these primitives. When a TaskContext is running, it accepts commands or events using its *Execution Engine*. The Execution Engine will check periodically for new commands in its queue and execute programs which are running in the task. When a TaskContext is started, the Execution Engine is running (Orocos, 2009b). Data flows through ports and is manipulated by algorithms in the TaskContext.

A TaskContext component consists of six states: *Init, PreOperational, Stopped, Active, Running, and FatalError*. The figure 3.6 shows that for each API function, a user 'Hook' function is available for customization by the designer. When an API function is called, an appropriate user 'Hook' function will be executed. The user application code is filled in by inheriting from the C++ TaskContext class and implementing the user 'Hook' functions. In OROCOS, there are eight user 'Hook' functions such as *configureHook(), cleanupHook(), activateHook(), startHook(), updateHook(), stopHook(), resetHook(), and errorHook()* that are called when the TaskContext's states change. In the following, a brief introduction into the user 'Hook' functions, operational states, and state transition diagram of a TaskContext component is presented.

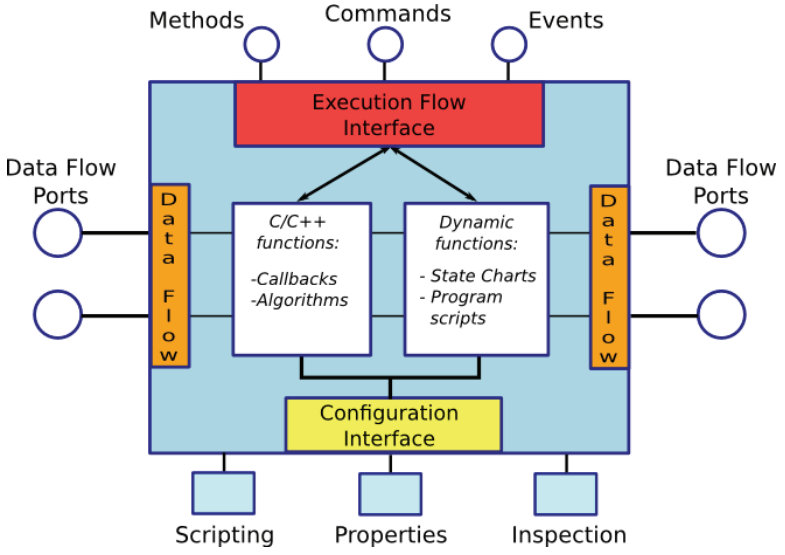


Figure 3.5 Schematic overview of a TaskContext component (Orocos, 2009b)

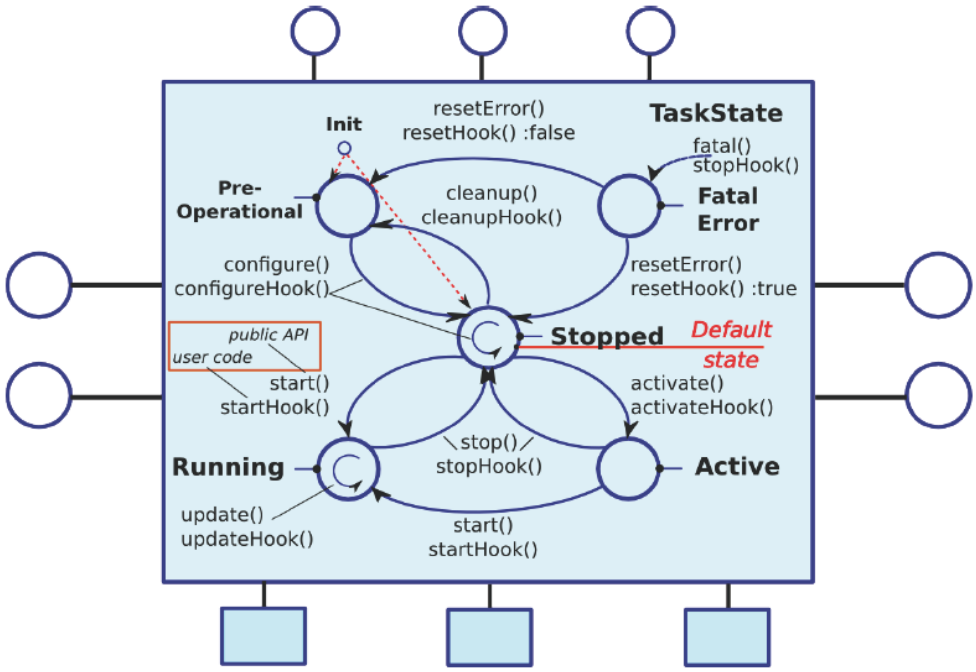


Figure 3.6 Extended TaskContext state transition diagram (Orocos, 2009b)

In the configureHook() function, users can insert such configuration code (for example read XML files, print status messages, etc.). In the cleanupHook() function, users can write some cleanup code (such as write XML files, free resources, etc.). Therefore, the configureHook() and cleanupHook() functions are used to implement the following tasks:

- The *configureHook(): bool* function will be executed when *configure()* is called to make a state transition of the TaskContext component from the PreOperational state to the Stopped state. If the configureHook() function returns false, the TaskContext stays in the PreOperational state. If it returns true, the TaskContext enters the Stopped state and is ready to be started.
- The *cleanupHook()* function will be executed when *cleanup()* is called to implement a state transition of the TaskContext component from the Stopped state to the PreOperational state.

The user real-time application programs are generally implemented in the 'Hook' functions like activateHook(), startHook(), updateHook(), stopHook(), resetHook(), and errorHook() where each function undertakes its own specific task:

- The *activateHook(): bool* function will be executed when *activate()* is called to realize a state transition of the TaskContext from the Stopped state to the Active

state. If the activateHook() function returns false, the TaskContext remains in the Stopped state. If it returns true, the TaskContext enters the Active state and is ready to be started.

- The *startHook(): bool* function will be executed when *start()* is called to perform a state transition of the TaskContext component from the Stopped or Active state to the Running state. If the startHook() function returns false, the TaskContext remains in the Stopped or Active state. If it returns true then the TaskContext enters the Running state.
- The *updateHook()* function will be executed periodically or aperiodically when *update()* is called appropriately by the Execution Engine while the TaskContext component is in the Running state.
- The *stopHook()* function will be executed when *stop()* is called to make a state transition of the TaskContext component from the Running or Active state to the Stopped state. Note that, the stopHook() function is executed just after the last updateHook() execution.
- The *resetHook(): bool* function will be executed when *resetError()* is called to implement an error recovery from the FatalError state. If the resetHook() function returns true, the recovery is possible and the TaskContext enters the Stopped state. In case it returns false, the recovery is impossible and the TaskContext goes to the PreOperational state. Then it requires a new configuration through calling the configure() which calls the configureHook() function, the user function, in turn.
- The *errorHook()* function must be executed, instead of the updateHook() function, while the TaskContext is in the RunTimeError substate of the Running state.

The *Init, PreOperational and Stopped state*: when created, a TaskContext component is in the Init state. After construction, it enters the PreOperational state or Stopped (the default state), depending on what the designer has chosen. If the TaskContext enters the PreOperational state, it requires an additional configure() call that makes the configureHook() function executed. If this execution succeeds, the TaskContext enters the Stopped state (see figure 3.6).

The *Active state* is for processing incoming programs, commands, state machines, and events, but not yet running the updateHook() function. It is used for TaskContext components that require to accept commands before they are running. Hence, the Active state is optional and can be skipped (Orocos, 2009b).

The *FatalError state* is entered whenever the TaskContext's fatal() function is called that indicates an unrecoverable error occured, possibly in the updateHook() or in any other 'Hook' function. The Execution Engine is stopped immediately and the stopHook() function is called when the TaskContext enters this state.

The *Running state* consists of three substates: Running, RunTimeWarning, and RunTimeError. The state transitions between the substates are described as following: when user application code calls error(), the RunTimeError state is entered and the errorHook() function is executed instead of the updateHook(). If the TaskContext detects that it can resume normal operation, it calls recovered(), which leads to the Running state again and in the next iteration, the updateHook() function is again executed. When warning() is called, the RunTimeWarning state is entered and the updateHook() function is still executed. Then, it calls recovered() to go back to the Running state (Orocos, 2009b).

3.4.3 Resemblance between elementary controller-agent and TaskContext component

As mentioned in section 3.3.5, there is a resemblance between an elementary controller-agent of the MACIF and a TaskContext component of the OROCOS framework. Based on the information presented in the two previous sections, in this part we will discuss this resemblance in detail.

As we want to realize the concepts, operation mechanisms and decentralized coordination architecture of the MACIF into OROCOS, the approach that we will use is mapping an elementary controller-agent into a TaskContext component. By comparing the switching behavior of an elementary controller-agent (see figure 3.4) and the extended TaskContext state transition diagram (see figure 3.6), it can be seen that there are some functional equivalences with regard to the operating states between an elementary controller-agent and a TaskContext. Starting from this viewpoint, we will discuss how to map two operating states of the elementary controller-agent into the extended state transition diagram of the TaskContext component.

We start with using *the Running state of the TaskContext for the Active state of the elementary controller-agent*. The reason is because Running is the only state in which a TaskContext is running and can update its state variables by executing the updateHook() function. Therefore, it meets the basic specification of the Active state of the elementary controller-agent. After this option has been selected, it opens up three possible choices for implementing *the Inactive state of the elementary controller-agent:* (i) use the Stopped state of the TaskContext, (ii) use the Active state of the TaskContext, (iii) use another substate of the Running state of the TaskContext.

In summary, the resemblance between an elementary controller-agent and a TaskContext component discussed above leads to a feasible combination of two elements such that the strong points of both are merged to be better, whereas their weak points are minimized significantly. In the next part, this combination will be realized in the form of mapping the elementary controller-agent into the extended state transition diagram of the TaskContext.

3.5 Mapping the MACIF into OROCOS Framework

3.5.1 Some remarks

Before discussing the possible solutions, *four remarks* are given:

1. According to the state transition diagram of an elementary controller-agent (see figure 3.4), it is noticed again that after being started the elementary controller-agent starts running and it should first enter the Inactive state, not the Active state.
2. An elementary controller-agent should be stopped only when it is in the Inactive state, not in the Active state. It means that if the elementary controller-agent is in the Active state, it should first be switched to the Inactive state, and then stopped.
3. As the function with name update() is used in both elementary controller-agent and TaskContext, this update() function of the elementary controller-agent is renamed to refresh() so as to avoid misunderstanding while mapping. So from now on, *the refresh() function* will be used to update state variables of the elementary controller-agent while it is running, both in the Active and Inactive state.
4. The mapping is performed with a special attention that it should keep the extended TaskContext state transition diagram (figure 3.6) changed or modified as little as possible. So that it does not cause much problem with regard to *the pre-defined operation mechanisms* of the RTT of the OROCOS framework.

It is also remarked that there are *configuration activities* required to be accomplished before the MASC starts running. The configuration activities are:

- Select the sampling interval for MACS.
- Select specifications and parameters for controller-agents of MACS.
- Configure application-specific hardware drivers for the hardware access.
- Configure interfaces between MACS and the plant by means of connections between I/O ports of MACS with virtual I/O ports of the simulated plant or with physical I/O ports of the real plant.
- Start the cosimulation engine between MACS running under Linux OS and the plant running under Windows OS.

We need a “not running” state for accomplishing the configuration activities. *The Stopped or Active state of the TaskContext* can be used for this purpose. It means that at least one of the two states should be reserved, i.e. kept unchanged as mush as possible. This issue will be taken into account while mapping.

3.5.2 Solutions for the Inactive state of the elementary controller-agent

Solution using the Stopped state of the TaskContext

This solution has some disadvantages: in the Stopped state, the TaskContext is not running and the update() function is not available here. Therefore, this solution does not meet the basic specification with regard to the elementary controller-agent, i.e. running and updating the state variables in the Inactive state. In practice, the extended TaskContext state transition diagram (figure 3.6) could be modified to meet this requirement. However, the *semantics* of the Stopped state and other states will be changed much in this case. Furthermore, making the update() function available in the Stopped state seems to be an unreasonable semantic. This clearly is a huge modification with respect to the pre-defined operation mechanisms of the RTT of the OROCOS framework. Because of the disadvantages, this solution will not be further investigated.

Note: in the OROCOS framework, the semantic concept can be thought as the original purpose and design logic, which is properly assigned for each state or function. Modification of semantics normally leads to changes with respect to the pre-defined operation mechanisms of the RTT.

Solution using the Active state of the TaskContext

This solution utilizes the start() function of the TaskContext component for implementing the initialize() function of the elementary controller-agent. It has some disadvantages, or in other words, it requires the modifications to the RTT as follows:

- It has the same problem as the previous solution that the update() function is not available in the Active state of TaskContext. However, compared with the case of the Stopped state, modifying the Active state of TaskContext to overcome this problem does not cause much change to the semantic of the Active state.
- In the extended TaskContext state transition diagram (figure 3.6), there is no transition path from the Running state to the Active state. Hence, this solution does not provide a ready-to-use state transition path for implementing the finalize() function of elementary controller-agent (see figure 3.7).
- The onward and backward transition between the Running and Stopped state of TaskContext should be prevented because they cause two undesired switching behaviors for the elementary controller-agent as mentioned in section 3.5.1 (the first two remarks).

Because the Active state of the TaskContext is used for the Inactive state of the elementary controller-agent, the configuration activities, which were discussed in section 3.5.1, can be implemented in the Stopped state of the TaskContext.

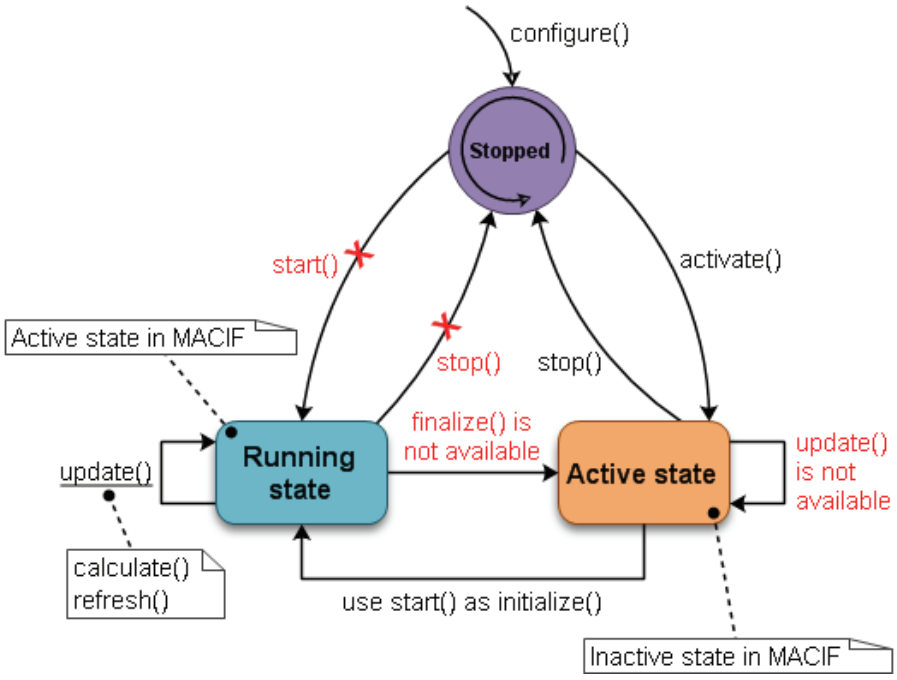


Figure 3.7 Solution using the Active state of TaskContext for the Inactive state of elementary controller-agent

Solution using another substate of the Running state of the TaskContext

This solution does not exploit neither the Stopped nor Active state of the TaskContext so that the configuration activities discussed above can be deployed in one of these states. Particularly, it is desired to have a transition from the Stopped to Running state via the Active state by calling in turn the activate() and then start() function (see figure 3.8). There is another direct path from the Stopped to Running state by just calling the start() function, but we will not use this path as it skips the useful Active state of TaskContext.

Because both the Active and Inactive state of elementary controller-agent use the Running state of TaskContext. Hence, a modification with respect to this Running state is required. A decision is made to modify the Running state of the TaskContext component into two new substates called *Operational* and *Idle* which respectively hold the roles of the *Active* and *Inactive* state of the elementary controller-agent. This modified Running state thus becomes *a composite state*. As a result, besides five operating states (Init, PreOperational, Stopped, Active, and FatalError) inherited from the TaskContext component, an elementary controller-agent will have the composite Running state that consists of the Operational and Idle state. We define the Running, Operational and Idle state as follows.

Running state definition

The Running state of the elementary controller-agent is a composite state that consists of two substates: the Operational and Idle state.

Idle state definition

The Idle state is a substate of the Running state in which the elementary controller-agent is running and can update state variables by executing the refresh() function, but is not able to calculate any control samples.

Operational state definition

The Operational state is a substate of the Running state in which the elementary controller-agent is running and can calculate control samples and update state variables by executing the calculate() and refresh() function, respectively.

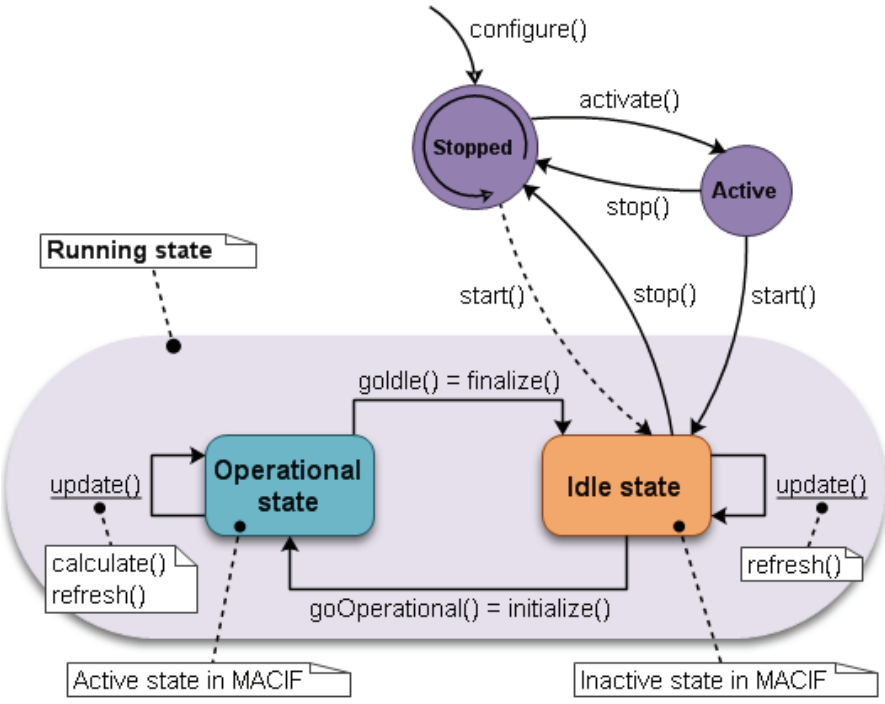


Figure 3.8 Solution using two substates of the Running state of the TaskContext for the Active and Inactive state of the elementary controller-agent

As this solution does not provide any ready-to-use state transition path for implementing the initialize() and finalize() function of the elementary controller-agent, two new functions named goOperational() and goIdle() are needed. The goOperational() and goIdle() function

will respectively have the same role as the initialize() and finalize() function used in the MACIF (see figure 3.8). As a result, an elementary controller-agent can switch between the Operational and Idle state by calling the goOperational() and goIdle() function.

- The *goOperational()* function is called while the elementary controller-agent is running and switching from the Idle state to the Operational state (i.e. becomes Operational).
- The *goIdle()* function is called while the elementary controller-agent is running and switching from the Operational state to the Idle state (i.e. becomes Idle).

Discussion: in comparison with the other two solutions, the solution using two substates of the Running state of the TaskContext is the only one that provides the elementary controller-agent with the update() function available in both the Active and Inactive state (see figure 3.8). Therefore, this solution meets the basic requirements of the elementary controller-agent: (i) running in both states, (ii) updating state variables in the Inactive state, (iii) calculating control samples and then updating state variables in the Active state. Moreover, it does not require much modification with regard to the extended TaskContext state transition diagram as other two solutions do. As a result, this solution is finally selected for the mapping. This combination results in a so-called *TaskContext-based elementary controller-agent* that will be the basic primitive of the new implementation framework named **ORO**COS-based Implementation Framework for **MACS** (called *OROMACS framework* in short).

3.5.3 TaskContext-based elementary controller-agent

The state transition diagram of a TaskContext-based elementary controller-agent is given in figure 3.9. As presented in section 3.4.2, the TaskContext-based elementary controller-agent inherits seven user 'Hook' functions of the TaskContext component being configureHook(), activateHook(), startHook(), stopHook(), cleanupHook(), resetHook() and errorHook(). Besides this, the updateHook() function of TaskContext will be modified to meet the requirements of the OROMACS framework (see section 3.8). The TaskContext-based elementary controller-agent has six new user 'Hook' functions:

- The *goOperationalHook()* function is executed when *goOperational()* is called.
- The *goIdleHook()* function is executed when *goIdle()* is called.
- The *calculateHook()* function is executed when *calculate()* is called while the elementary controller-agent is running in the Operational state to calculate control samples.
- The *refreshHook()* function is executed when *refresh()* is called while the elementary controller-agent is running in the Operational or Idle state to update state variables.

- The *operationHook(): double* function is implemented to respond to a request from a coordination object to indicate whether a controller-agent (either elementary or composite) wants to become Operational. The return value indicates the desire to become Operational: (i) if return value > 0.0, this controller-agent wants to be Operational or stays being Operational; (ii) if return value = 0.0, this controller-agent wants to be Idle or stays being Idle.
- The *acknowledgeHook (bool ack)* function is implemented to contain additional behaviour of a controller-agent (either elementary or composite) when receiving a response from a coordination object indicating whether this controller-agent is to be Operational. The *ack* is an acknowledge variable, i.e. an answer of the coordination object, specifying the activation status of this controller-agent.

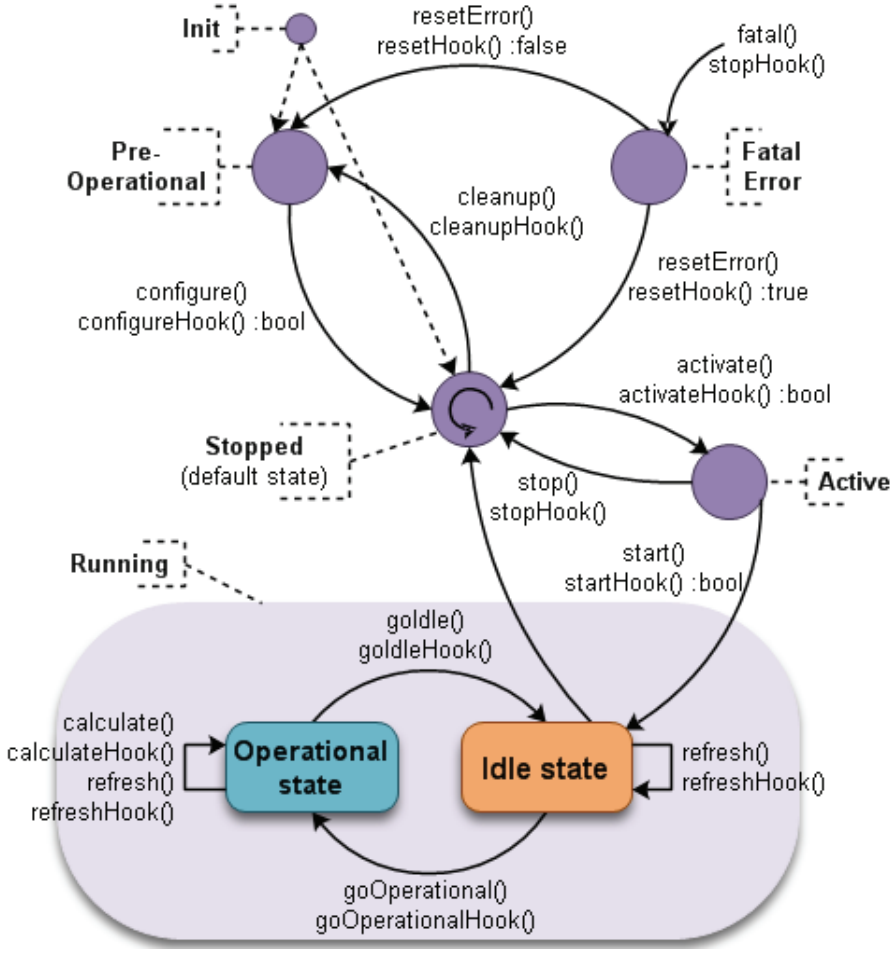


Figure 3.9 State transition diagram of a TaskContext-based elementary controller-agent

A summary of names of the operating states and user functions used in the three different frameworks (i.e. MACIF, OROCOS, and OROMACS) is presented in the following table.

The MACIF	The OROCOS framework	The OROMACS framework
not available	Init state PreOperational state Stopped state Active state FatalError state	Init state PreOperational state Stopped state Active state FatalError state
not available	Running state	composite-Running state
Active state	not available	Operational (sub)state
Inactive state	not available	Idle (sub)state
void start()	bool configureHook() bool activateHook() bool startHook()	bool configureHook() bool activateHook() bool startHook()
void initialize()	not available	void goOperationalHook()
real activation() void calculate() void update()	void updateHook()	double operationHook() void calculateHook() void refreshHook()
void finalize()	not available	void goIdleHook()
void stop()	void stopHook()	void stopHook()
not available	void cleanupHook()	void cleanupHook()
not available	bool resetHook() void errorHook()	bool resetHook() void errorHook()

Table 3.1 Summary of operating states and user functions used in three frameworks

3.6 Composite Controller-Agent in OROMACS

A composite controller-agent consists of a group of elementary and/or composite controller-agents and a coordination object in which the coordination object has a role to coordinate the activity behavior of the whole group. In this section, we will present how a composite controller-agent is formed and operates in the OROMACS framework.

The main idea formulating the composite controller-agent is that *a group of coordinated controller-agents can be dealt with as if it were an individual controller-agent*. If this is

possible, the composite controller-agents could be used in other groups, thus leading to hierarchical structures of the MACS. In the OROMACS framework, this is done as both the elementary and composite controller-agents have the same interface (see figure 3.10). Hence, seen from the outside a composite controller-agent behaves in the same way as an elementary controller-agent. The only difference between two types is the internal architecture (Van Breemen, 2001). This common interface of a general controller-agent (either elementary or composite) is made up of input and output ports, operation request and acknowledge signal(s) or message(s). The operation request signal(s) are execution results of the *operationHook(): double* function. The acknowledge signal(s) are the ones of the *acknowledgeHook (bool ack)* function.

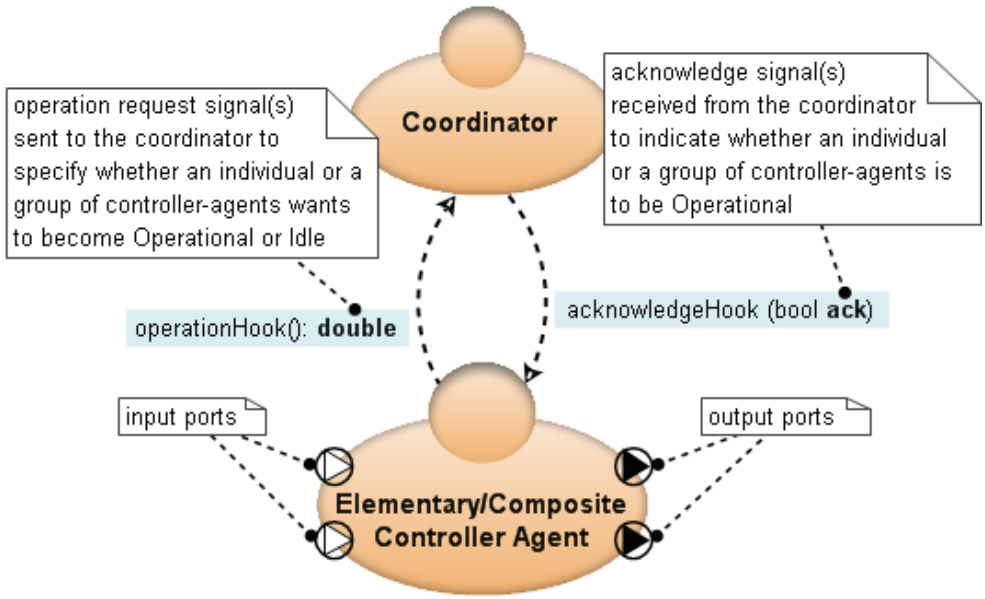


Figure 3.10 The common interface of a controller-agent (adopted from Van Breemen and De Vries, 2001)

Besides the *operationHook(): double* and *acknowledgeHook (bool ack)* function, in case of the composite controller-agent, it uses the following additional functions to build up the interface and operation mechanism of subcontroller-agents and the coordinator:

- The *resoluteHook(): double* function is implemented to combine the individual operation request signals of all subcontroller-agents into an operation request signal that represents the group's operation intention. This group's operation request signal is then sent to a higher level coordinator.
- The *decideHook(bool ack)* function is implemented to decide which subcontroller-agent(s) of the composite controller-agent is/are to be Operational. This depends

on four things: (i) the acknowledge signal received from a higher level coordinator indicating that this group (i.e. the composite controller-agent) may become Operational; (ii) the operation self-intention of the controller-agent; (iii) operation intentions of other controller-agents in the composite controller-agent; and (iv) the coordination mechanism used in the group. Note that, if a negative acknowledge signal is received from a higher level coordinator, then none of the subcontroller-agents in the group may become Operational. If the received acknowledge signal is positive, then the normal coordination procedure will be followed, with a condition that at least one subcontroller-agent should be activated.

- The *combineHook()* function is implemented to combine outputs of subcontroller-agents of the composite controller-agent.

The sequence diagram of an elementary controller-agent is depicted in figure 3.11. The sequence diagram of a composite controller-agent is given in figure 3.12. The sequence diagram of a coordination object is described in figure 3.13. The flow of messages between components in these diagrams describes the operation mechanism of the composite controller-agent. We will not discuss these diagrams here. For detailed information, the readers are referred to chapter 5 of the PhD thesis of Van Breemen (2001).

Some remarks are given:

- The *activation()* function used in the MACIF is now replaced by the *operationHook()* function with the same role and functionality.
- The *start()* function used in the MACIF is equivalent to the *sequence of three functions: configureHook(), activateHook() and startHook()*.

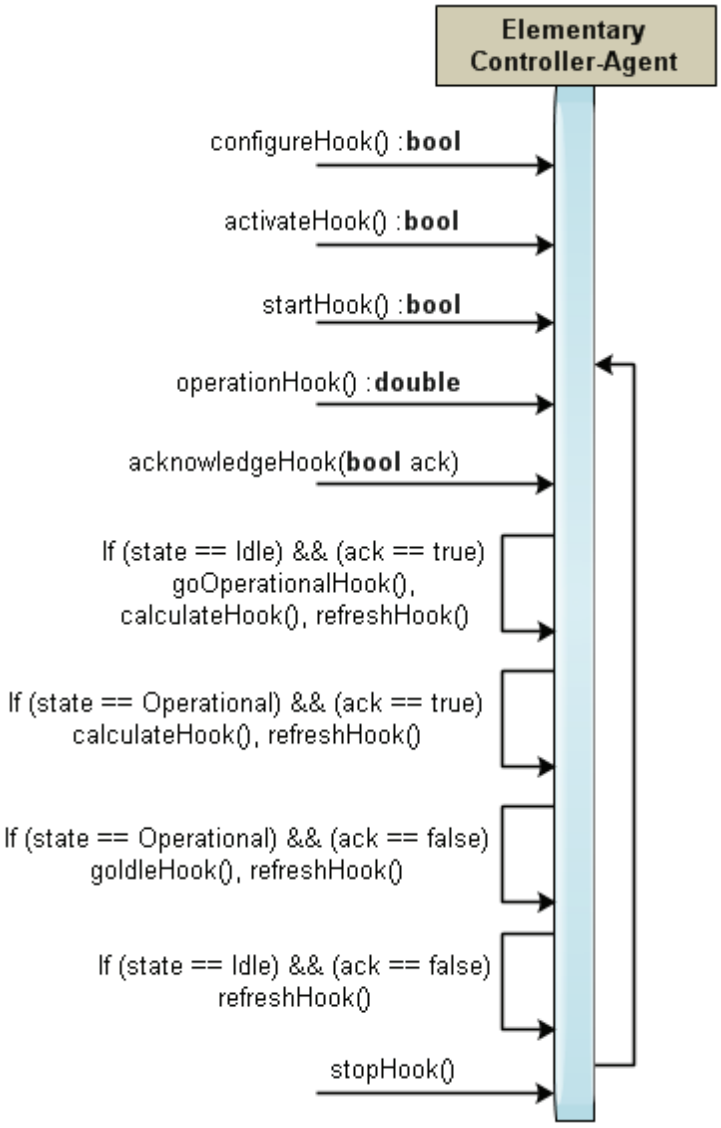


Figure 3.11 Sequence diagram of an elementary controller-agent

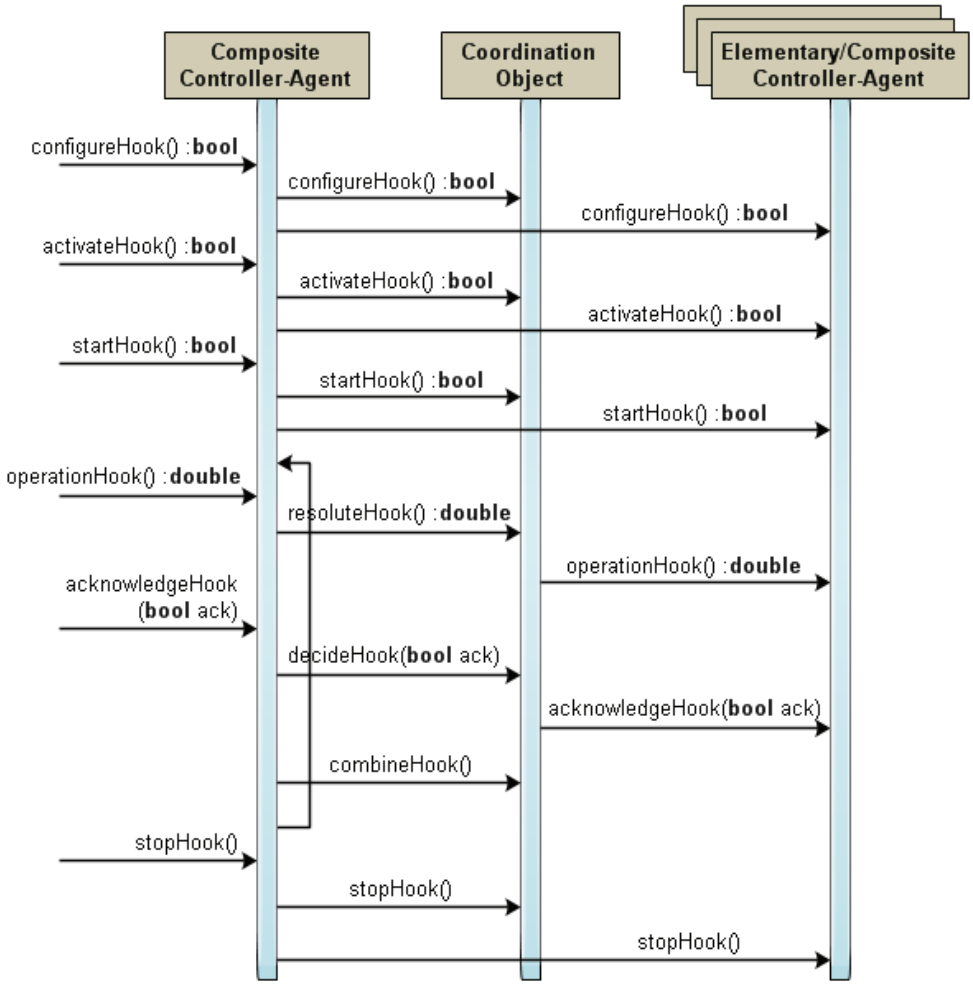


Figure 3.12 Sequence diagram of a composite controller-agent

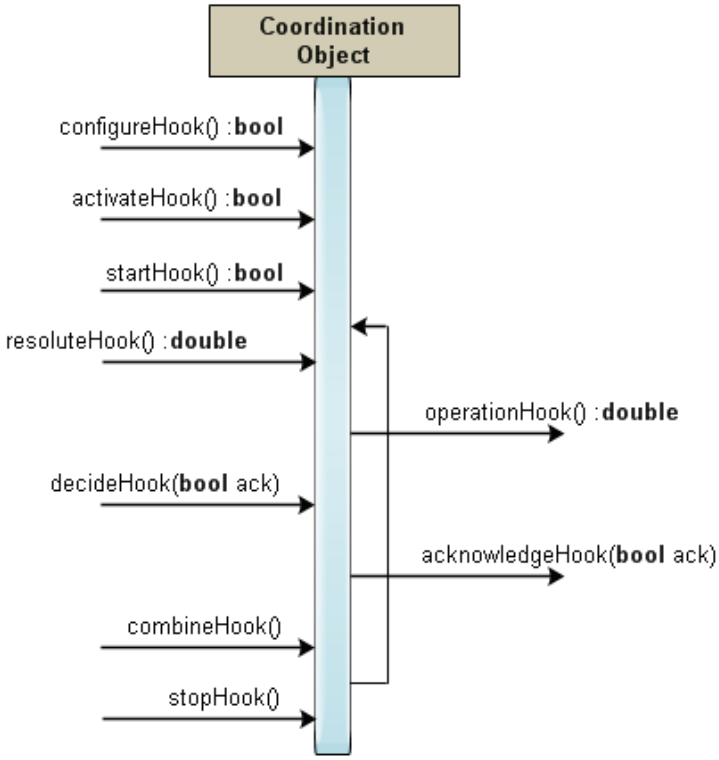


Figure 3.13 Sequence diagram of a coordination object

3.7 Polymorphism in OROMACS

In this section, we present an extension of the port-based polymorphic modeling approach (De Vries et al., 1993; De Vries, 1994) into the OROMACS framework, which makes the controller-agent polymorphic.

3.7.1 Port-based polymorphic modeling approach

Port-based approach

The *concept of a "port"* was first introduced in electrical circuit theory by Wheeler and Dettinger (1949), then extended to arbitrary power ports: electrical ports, mechanical ports, hydraulic ports, thermal ports, etc.. Inspired by the port concept, Henry Paynter, a civil

engineer, invented a so-called *"bond"* that is a notation based on the efficient representation of the relation between two power ports by just one line. The *"bond graph"* notation was finally formulated when he introduced the concept of the junction (Paynter, 1961). In comparison with other modeling methods using transfer functions, state space descriptions, or block diagram representations, bond graphs are a good modeling environment as it offers support for obtaining good models of complex systems and providing much insight of systems, thus be easily understood (Breedveld et al., 1991). The notation provided by Paynter and others (Karnopp and Rosenberg, 1968; Breedveld, 1982) allows a graphical representation of systems that enables direct reasoning and is unambiguous (De Vries et al., 1993).

The port-based modeling and control has gained much interest. The reason, as discussed in Van Amerongen (2007) is: a port-based approach in all domains has major advantages with regard to mechatronic designs which use the multi-disciplinary design methodology. In this research, we will use the port concept defined by Van Amerongen (2007): "A port is the interface of a component to the outside world that allows the component to be connected with other components, such that the actual contents of the component is not relevant and could be of different degrees of complexity". This definition of port shows that *a port-based design* enables components to be easily expandable, thus leading to a good scalability for mechatronic designs. Through connections between ports, components can be constructed from other (sub)components in hierarchical structures. Moreover, as components are characterized by the ports through which they interact with the rest of the system, their contents can vary in complexity during different phases of the design process. The components themselves can have many forms or realizations without the need to change their interaction with the rest of the system. De Vries (1994) refers to this as *polymorphic modeling* and addresses that the port-based approach directly supports the concept of polymorphic modeling as well as hierarchy. In the following, the research of De Vries will be discussed in detail.

Port-based polymorphic modeling approach

De Vries (1994) defined: "Polymorphic modeling is the combined application of modularization and subtyping during model building, that is, the division of a subsystem description into a subsystem type and a subsystem specification, and the expression of a subsystem type in terms of one or more designated other types". Initially, there are two techniques applied to the polymorphic modeling approach:

- The first technique is *typing*, which enables the encapsulation and parametrisation of components and hierarchical modeling. Therefore, this technique improved decomposition and classification possibilities of modeling.
- The second technique is *port-based interfaces*, which makes it possible to model with a component-oriented approach instead of a process-oriented one. This technique improved the representation of models by enabling a component to be depicted as a network of lower level components.

De Vries discussed that typing and port-based interfaces together allow the decomposition and its representation as done in bond graph modeling, but they do not support classification properly. To solve this, he proposed to use the *technique of modularisation*. Modularisation involves the division of component definitions into two parts: a type, defining essential properties of a component, and a specification, defining the incidental ones. One type may have more than one specification, i.e. component types become polymorphic. Additionally, modularisation can be combined with subtyping to support classification, and thus extending hierarchical modeling into polymorphic modeling (De Vries et al., 1993). To highlight the two aspects, i.e polymorphic modeling and port-based interfacing, the name *"port-based polymorphic modeling approach"* will be used.

In summary, the port-based polymorphic modeling approach is useful because it

- improves the classification of subsystems by means of generic as well as specific typing;
- further enhances the reusability, because subsystem types and subsystem specifications can be reused separately;
- enables the subsystem library to be organized in a kind-of-hierarchy.

Because of the advantages, we decide to deploy this port-based polymorphic modeling approach into the OROMACS framework in such a way that the controller-agent will be modularized into two parts: (i) *a Type*, defining its interface, i.e. input and output ports, and (ii) *a Specification*, defining the implementation of this interface by means of specifying name, parameters, instance variables, connections, coordination mechanisms, internal behavior, etc. For the same Type, different Specifications can be implemented. This *makes the controller-agent polymorphic* and opens up the possibility to create libraries of structures for which the detailed implementation is unspecified.

3.7.2 Type and Specifications

As discussed in the previous parts, in the OROMACS framework, controller-agents are classified into two forms: either composite components which can contain other components, or elementary components which do not consist of other components. Both elementary and composite controller-agents can be considered as being general controller-agents. In the following, we present ideas that lead to a so-called polymorphism.

- First, in the scope of this thesis we define the interfaces of a controller-agent as the communication vehicles between the internals of the controller-agent and the environment. In addition, as the controller-agent is based on the TaskContext component, there are five distinct ways in which a controller-agent can be interfaced: through its properties, events, methods, commands and data flow ports. These are all optional interfaces.

- Second, we advocate that the interface of an elementary controller-agent and a composite controller-agent should be the same, i.e. the interface should contain common information only. Hence, we decide that for now the interface will only use the data flow port primitives. In other words, the interface of a controller-agent will be defined in terms of input and output ports. As a result, the controller-agent becomes *a "black-box"*. It means, from the outside you only see its interface, i.e. ports. The designers should understand the essential concept of the controller-agent but they do not need to know its content in detail. This kind of interface reduces the coupling between controller-agents and provides extra flexibility with regard to the MACS development.
- Third, we define the interface (i.e. ports) of controller-agents by means of a term named *Type*. We consider the elementary controller-agents having an Elementary Controller-Agent Specification (called *Elementary Specification* in short) and the composite controller-agents having a Composite Controller-Agent Specification (called *Composite Specification* in short). It is emphasized here that a controller-agent with a particular Type can be implemented in the form of different Elementary and/or Composite Specifications. We call this property *polymorphism*. In other words, a controller-agent becomes polymorphic by having *one Type with multiple Specifications*.

Comments: the concept of interface used in the discussion of the composite controller-agent (section 3.6) involves input and output ports, operation request and acknowledge signals or messages. However, these signals/messages actually are the control flow inside the composite controller-agent or MACS, which is fixed, thus need not to be defined again when implementing a MACS (see section 3.2.2). Hence, the concept of interface discussed in this section will only involve ports.

Next, the Type, Elementary and Composite Specification, and the polymorphism concept will be presented and illustrated by means of some examples.

Type

A Type of a controller-agent is defined through specifying its interface, i.e. ports.

- *Ports:* for each port we specify port type (input or output), port name and data type (double, bool, etc.).

Elementary Specification

To implement an Elementary Specification of a Type, i.e. an implementation of an elementary controller-agent, the following parts need to be specified:

- *Name:* each specification has a unique name. The name should clearly represent the main point of the specification.

- *Parameters:* for each parameter we specify information like name, data type (double, bool, etc.) and a default value.
- *Instance variables:* for each instance variable we specify information such as name and data type.
- *User 'Hook' functions:* the internal behavior of an elementary controller-agent is specified in terms of user 'Hook' functions (see section 3.5.3). However, it is not required to specify all the user 'Hook' functions because they are optional and application-specific. Amongst the set of user 'Hook' functions, the five main ones being operationHook(), goOperationalHook(), goIdleHook(), calculateHook() and refreshHook() are used at most to implement a specification.

Composite Specification

To implement a Composite Specification of a Type, i.e. an implementation of a composite controller-agent, the following parts need to be specified:

- *Name:* each specification has a unique name. The name should clearly represent the main point of the specification.
- *Subcontroller-agents:* each composite controller-agent consists of at least one or several subcontroller-agents which can be either elementary or composite controller-agents.
- *Coordinator:* there are at least five coordination mechanisms being Master-Slave, Fixed-Priority, Parallel, Sequential and Cyclic used in the OROMACS framework. Which particular coordination mechanism to use depends on the type of coupling (i.e. interdependencies) between the partial control problems.
- *Connections:* ports are connected to make up the data flow of a composite controller-agent. Connections can be made between subcontroller-agents, and also between the composite controller-agent and its subcontroller-agents. There are three kinds of connections present in a composite controller-agent. First, connections from input ports of the composite controller-agent to input ports of subcontroller-agents. Second, connections from output ports of subcontroller-agents to output ports of the composite controller-agent. Thrid, connections between output and input ports of subcontroller-agents themselves.

3.7.3 Examples

First, we present the Type "Filter Agent" together with two Elementary Specifications. Figure 3.14 shows block diagrams of the 1st-order and 2nd-order filter agent. In table 3.2, this Type and the two elementary specifications being "1st-order roll-off" and "2nd-order roll-off" are described. It can be seen that the information of Type (i.e. the interface or

ports) is generic, whereas the Elementary Specifications (i.e. name, parameters, instance variables, internal behavior or realization) are variable. The parameter *Ts*, which is used in the elementary specifications, is the sampling time that is specified by the OROMACS TaskContext (see section 3.8). Note that: pre-filter agents, roll-off filter agents and state variable filter agents have the same Type "Filter Agent".

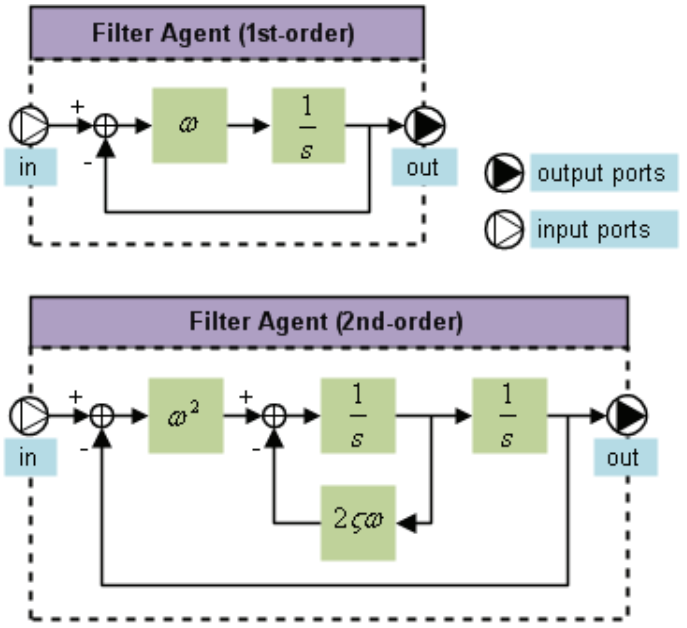


Figure 3.14 Block diagrams of the 1st-order and 2nd-order filter agent

Type: Filter Agent		
	input ports	`in`
	output ports	`out`
Elementary Specification		
Name	**`1st-order roll-off`**	
Parameters	ω : `roll-off frequency {rad/s}`	

Instance variables	`double u; // variable stores signal before filtering`
	`double s1y; // state variable`
	`double y; // variable stores signal after filtering`	

User 'Hook' functions	`/* only code in calculateHook() function */` `// read input port(s)`	

	`u = in();`
	`// calculate filtered signal`
	`s1y = ω * ( u - y );`
	`y = y + s1y * Ts;`
	`// write results to output port(s)`
	`out(y);`

Elementary Specification	
Name	**`2nd-order roll-off`**
Parameters	`ω : roll-off frequency {rad/s}` `ς : damping ratio`

Instance variables	`double u; // variable stores signal before filtering`
	`double s1y; // state variable`
	`double s2y; // state variable`
	`double y; // variable stores signal after filtering`

User 'Hook' functions	`/* only code in calculateHook() function */`
	`// read input port(s)`
	`u = in();`
	`// calculate filtered signal`
	`s2y = ω * ω * ( u - y ) - 2 * ς * ω * s1y;`
	`s1y = s1y + s2y * Ts;`
	`y = y + s1y * Ts;`
	`// write results to output port(s)`
	`out(y);`

Table 3.2 Type "Filter Agent" and two elementary specifications "1st-order roll-off" and "2nd-order roll-off"

Next, we present the Type "Feedback Controller Agent" and several Composite Specifications. In figure 3.15, a block diagram of the feedback controller agent is described. Composite specifications of the Type "Feedback Controller Agent" can be formed flexibly based on the combination of four subcontroller-agents such as: Pre-filter Agent, PID Agent, Roll-off Filter Agent, and Estimator Agent. Some typical composite specifications of the Type "Feedback Controller Agent" are:

- single PID
- PID with roll-off filter
- PID with estimator
- PID with roll-off filter and estimator
- PID with pre-filter, roll-off filter and estimator

In table 3.3, the Type “Feedback Controller Agent” and two composite specifications being “single PID” and “PID with roll-off filter” are described. It can be seen that the information of Type (i.e. input and output ports) is generic, whereas the Composite Specifications (i.e. name, subcontroller-agents, coordinator, and connections) are variable.

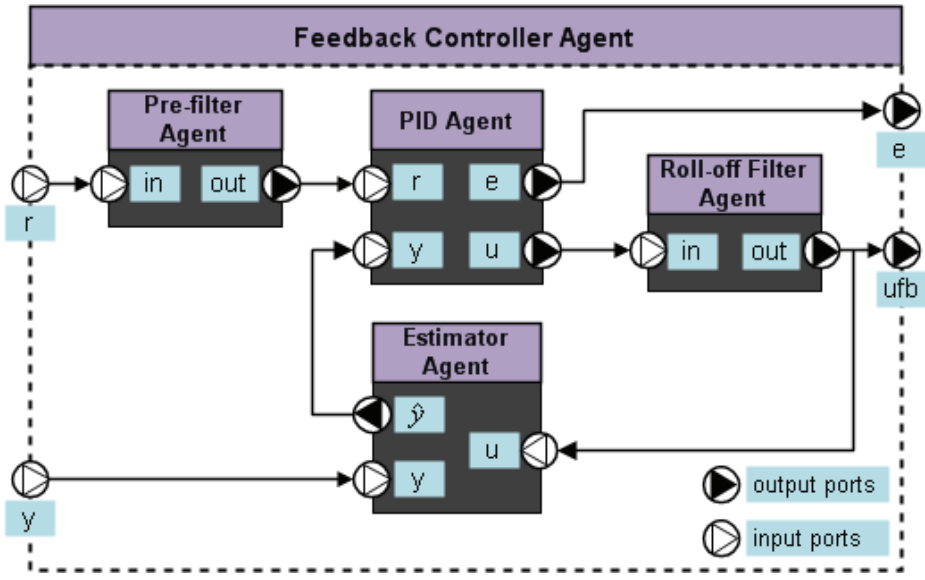
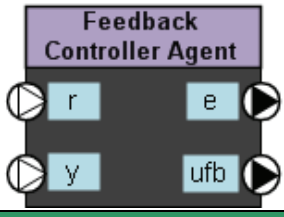


Figure 3.15 Block diagram of the feedback controller agent

Type: Feedback Controller Agent		
	input ports	`r: reference signal` `y: feedback signal`
	output ports	`e: error signal` `ufb: feedback control signal`
Composite Specification		
Name	**`single PID`**	
Subcontroller-agents	`PID Agent`	
Coordinator	`Parallel`	

Connections	from container().r to PID_Agent.r from container().y to PID_Agent.y from PID_Agent.e to container().e from PID_Agent.u to container().ufb
Composite Specification	
Name	**PID with roll-off filter**
Subcontroller-agents	PID Agent Roll-off Filter Agent
Coordinator	Master-Slave (PID Agent is the Master)
Connections	from container().r to PID_Agent.r from container().y to PID_Agent.y from PID_Agent.e to container().e from PID_Agent.u to Rolloff_Filter_Agent.in from Rolloff_Filter_Agent.out to container().ufb

Table 3.3 Type "Feedback Controller Agent" and two composite specifications "single PID" and "PID with roll-off filter"

Finally, we describe the Type "Operation Controller Agent" and the Composite Specification. In figure 3.16, a block diagram of the operation controller agent is described. This Type has two composite specifications being "only feedback" and "feedback and feedforward" which are presented in table 3.4.

Some discussions:

- Through these examples, we summarize the polymorphism concept: *a Type can have multiple specifications in which each specification can be implemented in the form of an Elementary Specification or a Composite Specification*.
- In the example of the Type "Feedback Controller Agent", besides multiple composite specifications, this Type can be also deployed in terms of elementary specifications. For example, the composite specification "single PID" can be modified to become one of the elementary specifications such as "Parallel PID Controller", "PI-D Controller", and "PID Compensator".
- In the example of the Type "Operation Controller Agent", the essence is that we can specify a feedback and feedforward control structure without being explicit about the kind of feedback controller, i.e. we have *configuration options*.

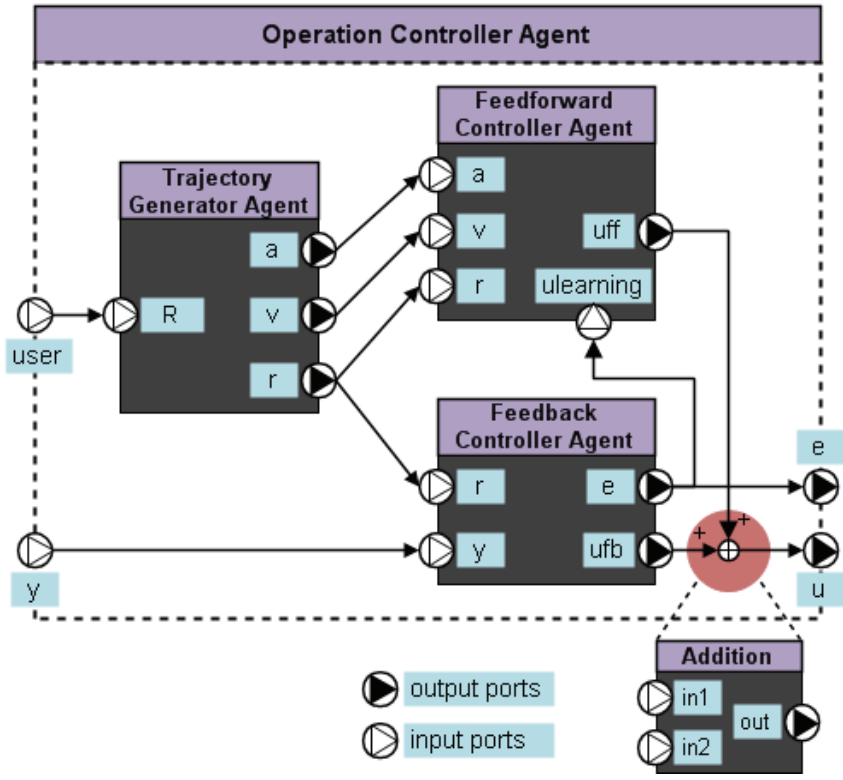
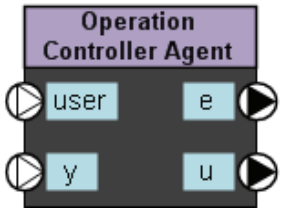


Figure 3.16 Block diagram of the operation controller agent

Type: Operation Controller Agent		
	input ports	`user: command signal` `y: feedback signal`
	output ports	`e: error signal` `u: control signal`
Composite Specification		
Name	**`only feedback`**	
Subcontroller-agents	`Trajectory Generator Agent` `Feedback Controller Agent`	
Coordinator	`Master-Slave (Trajectory Generator Agent is the Master)`	

Connections	from container().user to Trajectory_Agent.R from container().y to Feedback_Agent.y from Trajectory_Agent.r to Feedback_Agent.r from Feedback_Agent.e to container().e from Feedback_Agent.ufb to container().u
Composite Specification	
Name	**feedback and feedforward**
Subcontroller-agents	Trajectory Generator Agent Feedback Controller Agent Feedforward Controller Agent Addition Agent
Coordinator	Master-Slave (Trajectory Generator Agent is the Master)
Connections	from container().user to Trajectory_Agent.R from container().y to Feedback_Agent.y from Trajectory_Agent.r to Feedback_Agent.r from Trajectory_Agent.r to Feedforward_Agent.r from Trajectory_Agent.v to Feedforward_Agent.v from Trajectory_Agent.a to Feedforward_Agent.a from Feedback_Agent.e to Feedforward_Agent.u_learning from Feedback_Agent.e to container().e from Feedback_Agent.ufb to Addition.in1 from Feedforward_Agent.uff to Addition.in2 from Addition.out to container().u

Table 3.4 Type “Operation Controller Agent” and two composite specifications “only feedback” and “feedback and feedforward”

3.7.4 Realization of Types

The previous part shows that a conventional control system such as PID controller or an advanced control system like Model Reference Adaptive Systems (MRAS)-based Learning Feed-Forward Controller (Van Amerongen, 2006; Cuong, 2008) can be realized in the form of the Type “Operation Controller Agent”. Depending on each specific application, an appropriate control system configuration will be selected. As a control system configuration actually is an elementary specification or a composite specification of a Type, *selecting a control system configuration is considered as a realization of a Type*.

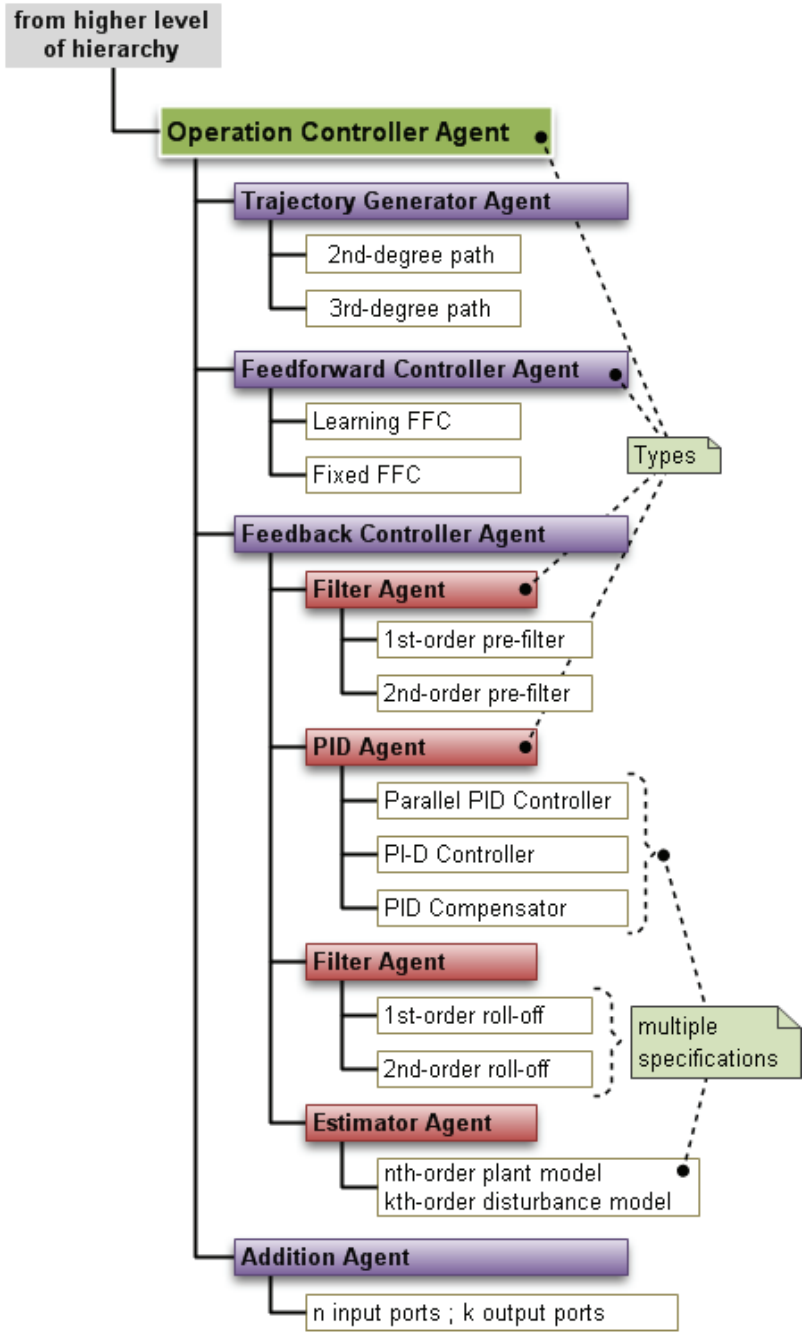


Figure 3.17 Overall hierarchy of the Type “Operation Controller Agent”

In figure 3.17, an overall hierarchy of the Type “Operation Controller Agent” with multiple specifications is presented. Based on this hierarchy, different realizations of the Type “Operation Controller Agent”, varying from simple to advanced control system configurations, can be easily obtained. Particularly, a realization of a Type is based on selecting or specifying appropriate specifications. For example, a realization of the Type “Operation Controller Agent” in figure 3.18 is accomplished by specifying five particular specifications:

- The composite specification “only feedback” for the Type “Operation Controller Agent”.
- The elementary specification “2nd-degree path” for the Type “Trajectory Generator Agent”.
- The composite specification “PID with roll-off filter” for the Type “Feedback Controller Agent”.
- The elementary specification “PID Compensator” for the Type “PID Agent”.
- The elementary specification “2nd-order roll-off”for the Type “Filter Agent”.

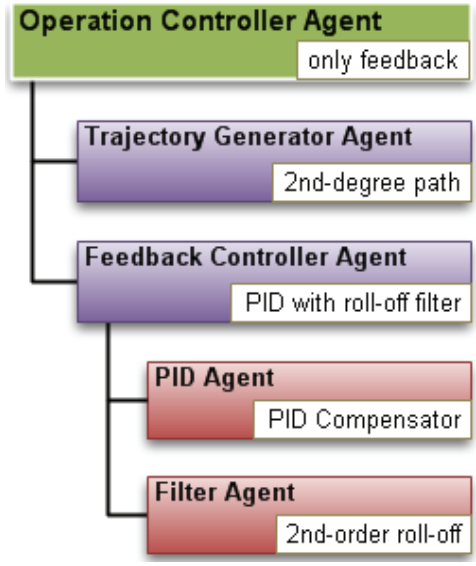


Figure 3.18 A realization of the Type “Operation Controller Agent”

3.8 OROMACS TaskContext and OROMACS Root-Agent

In this section, we discuss how the OROMACS framework operates within the OROCOS framework. In an overall hierarchy of MACS (figure 3.19), OROMACS TaskContext is the most top component that communicates directly with the outside world, i.e. either with the controlled plant through sensors and actuators or with other OROMACS TaskContexts. The OROMACS TaskContext's ports are categorized into two types:

- *Sensor ports* are the OROCOS input ports which are connected with sensors of the controlled plant or with other OROMACS TaskContext's ports to read information required by MACS.
- *Actuator ports* are the OROCOS output ports which are connected with actuators of the controlled plant or with other OROMACS TaskContext's ports to write information produced by MACS.

It means that sensor and actuator ports of the OROMACS TaskContext use *OROCOS data ports*, thus being suitable for hard real-time, thread-safe, and lock-free data exchange (Orocos, 2009a). When multiple OROMACS TaskContexts are present, communication between them is done through OROCOS data ports. Moreover, because the OROMACS framework uses the OROCOS framework as a run-time environment, it is possible to build *multi-threaded MACS* with performances being deterministic real-time behavior and thread safety in which *each thread is an OROMACS TaskContext* (see figure 3.19).

Each OROMACS TaskContext holds a single OROMACS Root-Agent. The OROMACS Root-Agent contains a control system (i.e. a MACS) which can be in the form of an elementary or a composite controller-agent. However, the OROMACS Root-Agent and all its subcontroller-agents use *OROMACS data ports*. The question is *how to make the connections between OROCOS data ports and OROMACS data ports easy and convenient?* We start this discussion by first giving an example: applying results of the realization of the Type "Operation Controller Agent" presented in section 3.7.4, we obtain a MACS design in which the Operation Controller Agent is deployed as an OROMACS Root-Agent (see figure 3.20).

Because the OROMACS Root-Agent has two different kinds of data ports: (i) controller-agents' ports used for data flow in the MACS; and (ii) other ports containing information that are used to communicate with the outside world (i.e. with plants or other OROMACS TaskContexts). The OROMACS TaskContext however contains only the ports that are related to the outside world. As discussed, the ports are called sensor and actuator ports. Therefore, a solution that can extract the ports of the OROMACS Root-Agent which involve the outside world and then makes them available for the OROMACS TaskContext is desired.

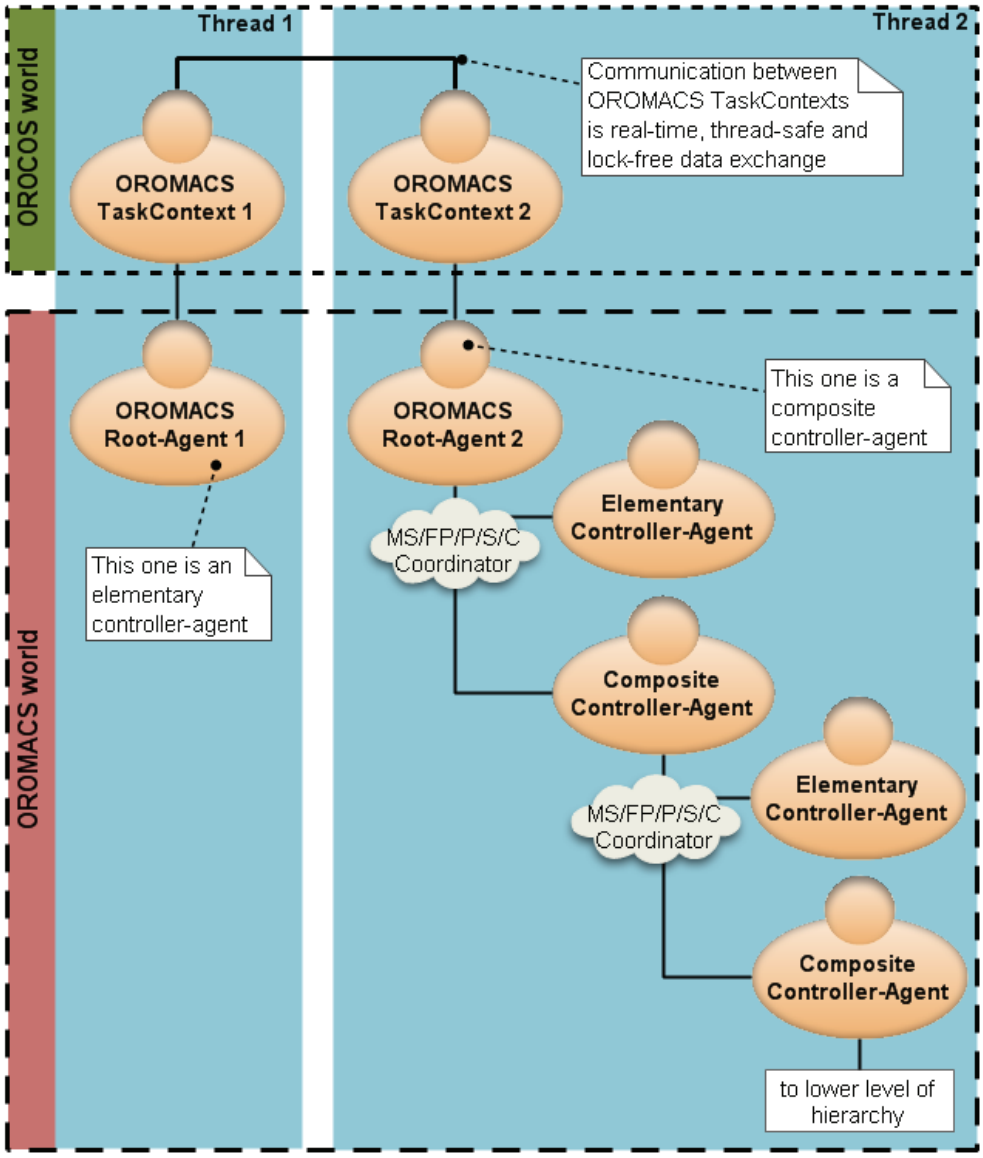


Figure 3.19 General organization of the OROMACS TaskContext(s) and OROMACS Root-Agent(s)

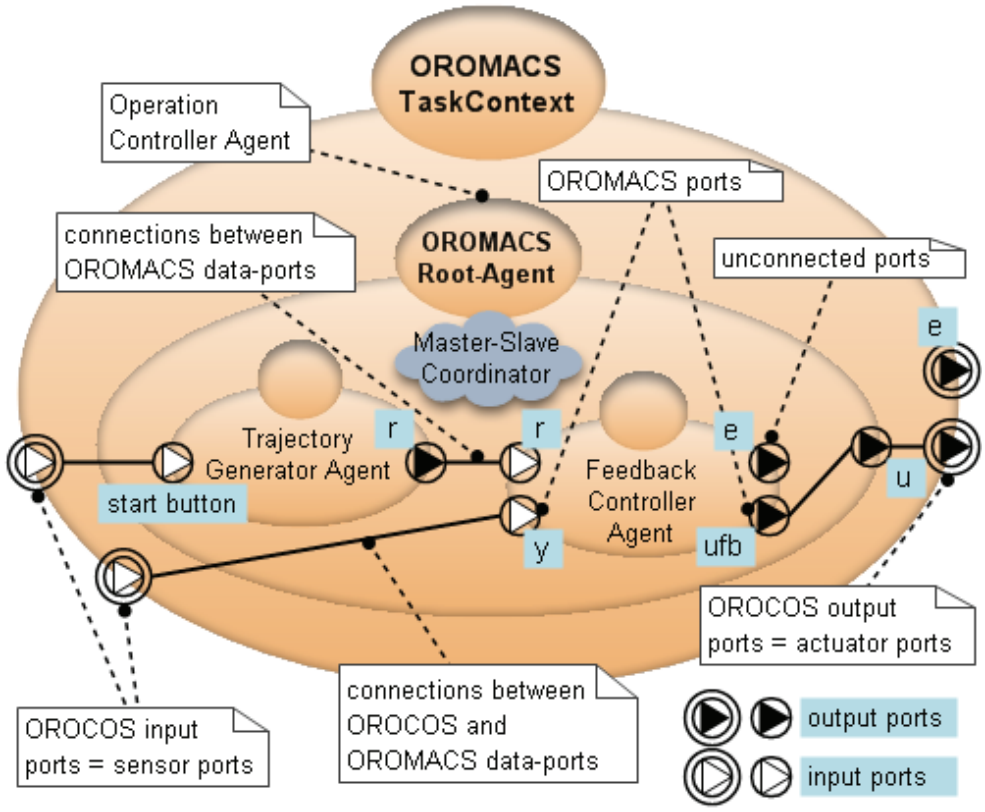


Figure 3.20 Operation Controller Agent is the OROMACS Root-Agent

We propose a solution to automatically produce sensor and actuator ports for the OROMACS TaskContext: all *unconnected* input and output ports of the OROMACS Root-Agent and all its subcontroller-agents are considered as candidate sensor ports and actuator ports, respectively. It means that the unconnected ports are already intended to communicate with the outside world. These ports will be converted from OROMACS to OROCOS data ports and then mapped to the OROMACS TaskContext as *"clones"*. Finally, the cloned ports can be (automatically) connected with the corresponding OROMACS data ports. By this way, the connections between OROCOS data ports and OROMACS data ports have the features:

- The connections from OROCOS input ports (i.e. sensor ports) can go through several hierarchical levels and can be directly connected with input ports of controller-agents that need the information. For example, a sensor port of the OROMACS TaskContext is directly connected with the input port "y" of the Feedback Controller Agent.

- Output ports of controller-agents of the OROMACS Root-Agent can be connected or unconnected. For example, the output port “e” of the Feedback Controller Agent is not connected in this example (figure 3.20). When the Feedforward Controller Agent is present in the control system, this port will be connected with the input port that requires learning signals. However, as the output port “e” is unconnected, it will be made available as an actuator port of the OROMACS TaskContext. This port can be connected with other OROMACS TaskContexts such as GUI or Cosimulation TaskContext for special purposes.
- However, output ports of controller-agents should be connected through each hierarchical level of the OROMACS Root-Agent in case they are required to be coordinated by coordinators. An example for this case is the output port “u” that is connected with an actuator port of the OROMACS TaskContext through all hierarchical levels (see figure 3.20). This port is used to steer the controlled plant. Coordination of output ports of controller-agents will be discussed in section 3.9.2.

OROMACS TaskContext acts as an intermediary between the designer and MACS. It means that users interact with MACS by specifying configurations or sending commands to the OROMACS TaskContext. The configurations and commands are then forwarded to the OROMACS Root-Agent to control the operation of MACS. The main interactions are:

- Specify the execution engine of MACS by selecting or configuring: sampling interval, specifications and parameters for controller-agents, I/O device drivers, interfaces between MACS and plant, execution context (i.e. in a cosimulation setting with a virtual plant or in an experiment setting with a real plant). The configuration activities are usually done in the configureHook() function.
- Start the operation of MACS by forwarding in succession the configureHook(), activateHook() and startHook() function to the OROMACS Root-Agent.
- Handle the actuating and sensing issue inside the updateHook() function by forwarding periodically the actuateHook() and senseHook() function, respectively.
- Stop the operation of MACS through forwarding the stopHook() function.

The default *updateHook()* function of the TaskContext component will be modified to meet the mentioned roles of the OROMACS TaskContext: it is implemented as *a sequence of three functions: actuate(), sense() and operate()* that is shown in figure 3.21. This sequence is chosen to ensure that the actuating and sensing are done as simultanously as possible, before any activity of the OROMACS Root-Agent that will be implemented in the *operate()*. Each of these functions invokes a user ‘Hook’ function that allows the user to implement application-specific functionality.

The role of three user ‘Hook’ functions are explained hereafter:

- The *actuateHook()* function will be executed when *actuate()* is called to make the latest data on output ports of the OROMACS Root-Agent available on actuator ports of the OROMACS TaskContext.

- The *senseHook()* function will be executed when *sense()* is called to make the latest data on sensor ports of the OROMACS TaskContext available on input ports of the OROMACS Root-Agent and all its subcontroller-agents.
- The *operateHook()* function will be executed when operate() is called to implement the behavior of the OROMACS Root-Agent. This is done by means of the *operationHook(): double* and *acknowledgeHook (bool ack)* function that have been discussed in section 3.5.3.

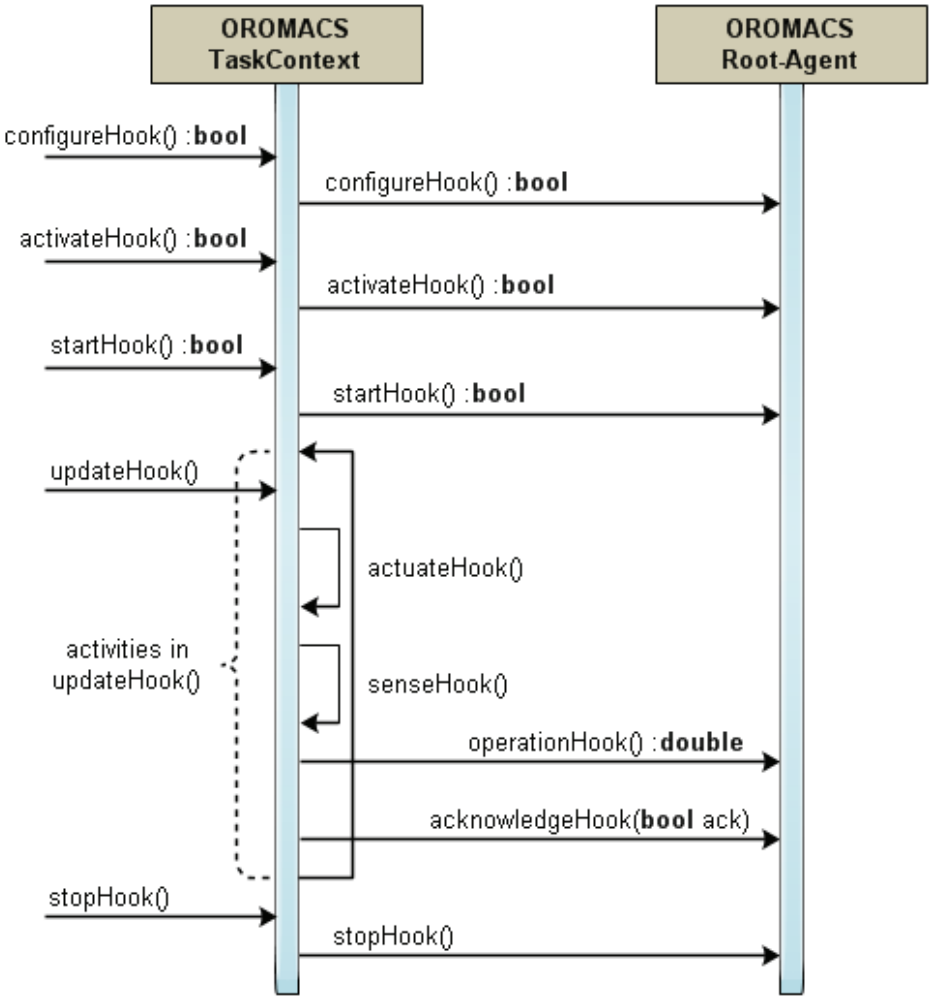


Figure 3.21 Sequence diagram of an OROMACS TaskContext

As discussed, either an elementary or a composite controller-agent can become the OROMACS Root-Agent. The reason is that the elementary controller-agent (see figure 3.11), the composite controller-agent (see figure 3.12) and the OROMACS Root-Agent (see figure 3.21) have the same interface as depicted in figure 3.22.

Discussion: although the OROCOS framework can support distributed real-time control systems, i.e. a real-time MACS can be distributed in several PCs. However, this thesis will only consider implementing MACS in one PC, but with multithreading behavior. In addition, this thesis will not make any change to the basic libraries of the OROCOS framework as well as system libraries of the core linux. We also do not introduce any function that can harm the proper operation of the OROCOS framework. Hence, the real-time behavior, task management and scheduling, IPC, etc. are kept unchanged. Specifically, a real-time multi-threaded MACS will be developed based on the OROMACS framework that runs on top of the OROCOS framework; the OROCOS framework will run on top of either Linux OS or optionally on top of hard real-time targets such as RTAI/LXRT (www.rtai.org) or Xenomai (www.xenomai.org).

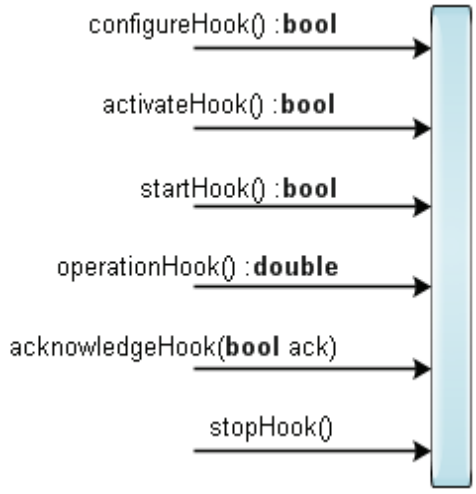


Figure 3.22 Common interface of the OROMACS Root-Agent

3.9 Roles of Polymorphism and Coordination

3.9.1 Polymorphism provides configuration options

Although the open architecture of a MACS design, provided by the MACIF, enables controller-agents to be added, modified or removed from the MACS without redesigning and reprogramming the remaining parts, some modification is still needed in practice:

- When a designer wants to change a control algorithm implemented in an elementary controller-agent, he needs to modify parameters, instance variables and user functions (i.e. internal behavior) of this elementary controller-agent.
- When a designer wants to change a control system configuration implemented in a composite controller-agent, he needs to modify the list of subcontroller-agents, coordination mechanism and inner connections of this composite controller-agent.

By using the OROMACS framework, *the open architecture and hierarchical structure of MACS is additionally featured with polymorphism*. As a result, any change in the design of a MACS now requires minimum effort and time to be spent on the modification. We demonstrate this advantage through extending the previous example: because practical applications usually require an extra safety layer, the Operation Controller Agent described in figure 3.20 has to be improved. However, instead of performing a lot of modifications as mentioned above, the only thing that has to be done is to select or specify appropriate specifications to obtain this desired MACS design.

Figure 3.23 presents a realization of the Type “Safe-Guarded MACS” that actually is the realization of the Type “Operation Controller Agent” (see figure 3.18) supplemented with a realization of the Type “Safe-Guarded Agent”. In this case, five specifications which have been specified while realizing the Type “Operation Controller Agent” are kept unchanged and reused into the realization of the Type “Safe-Guarded MACS”. Specifically, the designer only needs to specify two other specifications:

- The elementary specification “single axis” for the Type “Safe-Guarded Agent”.
- The composite specification “single application” for the Type “Safe-Guarded MACS”.

This design method is possible because control algorithms and control system configurations are already realized in terms of elementary specifications and composite specifications, respectively. It means:

- for each elementary specification, the implementation with regard to parameters, instance variables, and user ‘Hook’ functions; and
- for each composite specification, the implementation with respect to the list of subcontroller-agents, coordinator, and inner connections

are complete in the design phase; thus specifications are ready to be selected or specified in the realization phase.

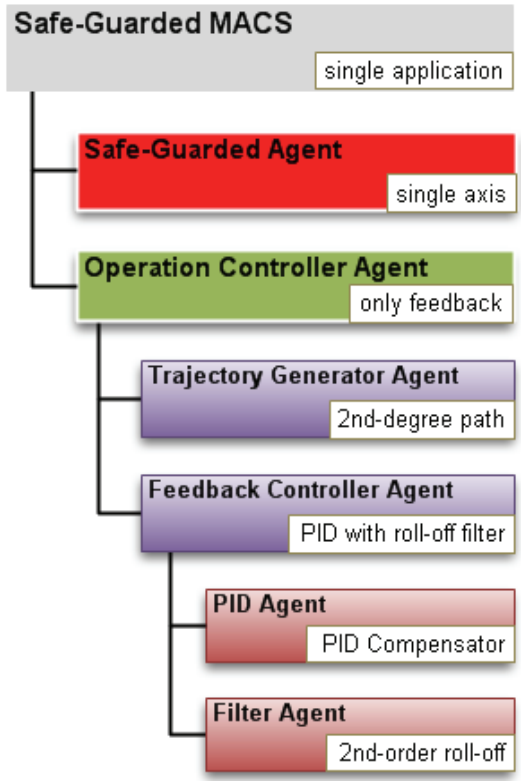


Figure 3.23 A realization of the Type "Safe-Guarded MACS"

Based on this approach, a set of elementary specifications can constitute a library of control algorithms; and a set of composite specifications can form a library of control system configurations. As a result, with a sufficient library of multiple specifications, the *design and programming* of a control system (i.e. a MACS) becomes *a matter of configuration and composition of controller-agents*. We conclude here that polymorphism provides users configuration options.

3.9.2 Pre-schedule operationality of MACS by coordinations

The realization of the Type "Safe-Guarded MACS" (figure 3.23) results in a safe-guarded MACS design that is depicted in figure 3.24. It is remarked that the Safe-Guarded MACS now has the role of an OROMACS Root-Agent. Particularly, this safe-guarded MACS

design has the same structure as the "OROMACS Root-Agent 2" which is shown in figure 3.19. Through this example, the role of coordinators in coordinating multiple output ports of controller-agents will be discussed.

An example is given: two output ports with name "u", one of the Safe-Guarded Agent and another one of the Operation Controller Agent, are both connected with the output port "u" of the OROMACS Root-Agent (i.e. the Safe-Guarded MACS). We will show that by using different coordination mechanisms, the designers can decide or schedule in advance the operationality of a control system or MACS. It is emphasized that the schedule is made at the design time, but it will have influence on the MACS operationality during run-time. Supposing that a cooperative coordination type (like a Parallel coordinator) is used then the Safe-Guarded Agent and Operation Controller Agent have the same role and priority. Hence, they will run in parallel and concurrently contribute control signals to the output port "u" of the OROMACS Root-Agent.

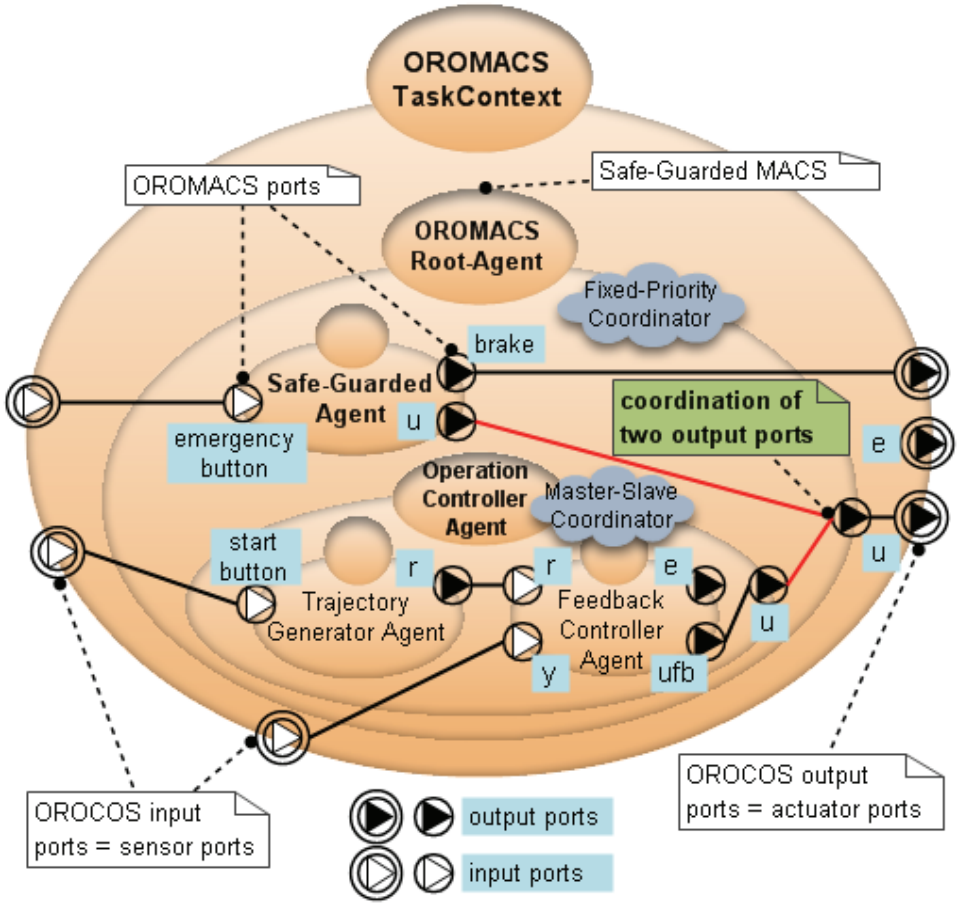


Figure 3.24 The Operation Controller Agent is supplemented with a Safe-Guarded Agent

However, because the role of the Safe-Guarded Agent is always more important than the one of the Operation Controller Agent, a Fixed-Priority coordinator (i.e. a competitive coordination type) is used in this example. As a result, the Safe-Guarded Agent is assigned a higher priority than the Operation Controller Agent. This coordination mechanism pre-schedules the operationality of the Safe-Guarded MACS as follows:

- In normal operating conditions, i.e. the Operation Controller Agent is Operational and the Safe-Guarded Agent is Idle, only the Operation Controller Agent contributes control signals to the output port "u" of the OROMACS Root-Agent.
- When a certain fault occurs, the Safe-Guarded Agent wants to become Operational to handle the fault. The Fixed-Priority coordinator will solve this competitive coordination. It immediately deactivates the Operation Controller Agent and concurrently activates the Safe-Guarded Agent. As a result, the Operation Controller Agent becomes Idle and the Safe-Guarded Agent becomes Operational. Hence, only the Safe-Guarded Agent contributes control signals to the output port "u" of the OROMACS Root-Agent. In this case, it can be thought that the output of the Safe-Guarded Agent overrides the output of the Operation Controller Agent.

Discussion: obviously, if other coordination mechanisms are used then the operationality of the Safe-Guarded MACS will be pre-scheduled differently. In other words, the designers can decide beforehand a desired control strategy by selecting a proper coordinator. In general, the selection of coordination should be appropriate to the specifications of the overall control problem and dependent on the type of coupling between partial control problems. Typically, the designer's experience might have an important role in selecting coordinators. However, the examples discussed in this chapter are quite simple to highlight the roles of coordination. In chapter 4, some control system design patterns using different types of coordination are developed, which will highlight the benefits of coordination. The design patterns can be considered as a reference with respect to the use and selection of coordinators.

3.10 Concluding Remarks

This chapter has presented the development of the OROMACS framework for solving complex control problems. In summary, two major works have been realized: (i) the functional combination between the MACIF and the OROCOS framework that creates the OROMACS framework; and (ii) the extension of the port-based polymorphic modeling approach into the OROMACS framework that improves the performance of the MACS designs made by using the OROMACS framework.

The main benefits of the OROMACS framework are:

1. It has the capability of developing the hierarchically structured MACS with an open architecture as the MACIF does. Moreover, the OROMACS framework has

improved the shortcomings of the MACIF, i.e. it allows users to develop multi-threaded MACS with deterministic real-time control behavior and thread-safe real-time inter-process communication mechanism; and it provides the MACS with the capability of handling events.

2. It makes the kind of composite components, which is currently not supported in the OROCOS framework, available in the OROMACS framework in terms of composite controller-agents.
3. It makes not only the controller-agent, but also the entire MACS polymorphic, i.e. we have obtained polymorphic controller-agents, polymorphic MACS. We call this property *polymorphism*, i.e. one controller-agent or a MACS with a particular Type can have multiple Specifications. This approach opens the possibility to create libraries of structures for which the detailed implementation is unspecified. Moreover, it makes the design and programming of a MACS become a matter of configuration and composition of controller-agents. This is done by selecting or specifying suitable specifications for elementary and composite controller-agents. In other word, polymorphism provides users configuration options. As a result, the open architecture and hierarchical structure of MACS has been improved with polymorphism.
4. It can pre-schedule the operationality of a MACS by using different coordination principles. Hence, the designers can decide beforehand a desired control strategy by selecting suitable coordination mechanisms.

Some discussions:

- By constructing an OROMACS Root-Agent (i.e. a whole MACS) residing in an OROMACS TaskContext (see figure 3.19), we *reduce the coupling between two frameworks*. The benefit is that any change of the OROCOS framework does not cause much modification with respect to the designed MACS. The only thing that probably needs to be adapted is the OROMACS TaskContext and its interfaces with the OROMACS Root-Agent.
- The OROMACS framework that we have developed, cooperates well with the OROCOS framework to provide a good development environment for real-time multi-threaded MACS. Specifically, the OROCOS framework provides a communication and computation mechanism for MACS; whereas, the OROMACS framework supports a configuration and coordination mechanism for MACS. As a result, *a MACS architecture involving four layers (Communication, Computation, Coordination, and Configuration)* is produced. This architecture is somewhat similar to the specification of a distributed system in term of four layers that was presented by Radestock and Eisenbach (1996).

Chapter 4

A Pattern-Based Safe-Guarded MACS Design Method

4.1 Introduction

The work in chapter 3 results in the OROMACS framework for solving complex control problems in mechatronic systems. However, as discussed in section 1.2.3, this framework still faces the shortcomings which can be improved by solving the two following research questions. This will be the main topic of this chapter.

> The 3rd research question: How to solve *the trade-off* between the desire to achieve a real-time safe-guarded MACS having good performances and a short development time?
>
> The 4th research question: How to support *the reusability* of the real-time safe-guarded MACS design results from previous projects into new projects?

We propose to tackle the two research questions with a combination of two aspects: (i) the OROMACS framework, and (ii) the pattern-based design method. As a result, nine control system design patterns are formed in which Safe-Guarded Agent, Single Function Agent,

Single Application Agent, and System Agent are four main design patterns. These design patterns are well organized to formulate two reusable generalized safe-guarded control solutions, one for simple mechatronic systems and one for complex mechatronic systems. In this thesis, we consider simple mechatronic systems as the motion systems with one degree-of-freedom (1-DoF) and complex mechatronic systems as the motion systems with multiple degrees-of-freedom (N-DoF). In summary, the main contribution of this chapter is *a pattern-based safe-guarded MACS design method* that solves these research questions.

This chapter is organized as follows. Section 4.2 describes the safety issues in mechatronic systems. Section 4.3 deals with the DemoLin setup, a simple mechatronic system: we start with the design of a safe-guarded MACS for the DemoLin and then formulate a reusable generalized safe-guarded control solution for simple mechatronic systems. Next, section 4.4 deals with the TriPod setup, a more complex mechatronic system. The design of a safe-guarded MACS for the TriPod reuses the safe-guarded MACS design results of the DemoLin setup. As a result, another reusable generalized safe-guarded control solution for complex mechatronic systems is formed. Section 4.5 summarizes nine control system design patterns. Finally, some concluding remarks are given in section 4.6.

4.2 Safety Issues in Mechatronic Systems

Safety issues in mechatronic systems (e.g. robots and manipulators) generally involve two aspects being safety for humans or operators and safety for machines themselves.

- The safety issue for humans can be guaranteed by always placing robots and manipulators in an area where people do not work closely to or directly with; and also by preventing people from entering the machine's working area. However, in some special cases, people have to work closely to the manipulator to perform experimental research or to test the system. In these cases, safety issues for operators become more complex.
- Regarding the safety issue for machines themselves, the problem is generally complicated because many possible sources of faults or errors have to be identified and handled strictly. Hence, designing safe-guarded control can be challenging and laborious.

While performing a certain task, a robot or manipulator system usually has to cope with various levels of criticality of different error/fault sources that can occur in an unwanted manner. If these faults are not identified correctly and handled strictly, they can bring dangerous situations for both human and machine. Because of that, we identify *18 fault sources* that are common in mechatronic systems (see table 4.1). We also classify the fault sources into three criticality levels: *dangerous, serious, and warning*. Dangerous is the highest hazardous level, the next one is serious, and warning is the lowest hazardous level. The list of fault sources and the categorization of criticality levels have multiple solutions

that can vary according to personal opinion and also depend on the specific system. However, the main point is that in general, fault sources that can appear in a particular system will be organized in a limited set of ordered criticality levels.

Table 4.1 distinguishes:

- *8 fault sources at dangerous level*: can result in hazardous accidents for operator and critical damages for machine; therefore cut-off the power supply as fast as posible is generally used to prevent faults from growing worse. The dangerous fault sources are signed with prefix "D".
- *7 fault sources at serious level*: can cause serious damages for machine only; so that emergency-stop as fast as possible is the desired response to prevent worse damage for the machine. The serious fault sources are signed with prefix "S".
- *3 fault sources at warning level*: do not cause much danger for operator and machine; therefore normal stop or enter standby mode with warning messages is the reasonable solution. The warning fault sources are signed with prefix "W".

Faults / Errors	Criticality level
D1. Control computer gets crashed totally	dangerous
D2. Failure of interface cards	
D3. Interconnecting wiring gets broken	
D4. Mechanical part is broken	
D5. Human collision	
D6. Human's unauthorized access into the working area	
D7. Failure of the power supply	
D8. Active emergency-stop	
S1. Exceeding end-effector working area	serious
S2. Exceeding joint working area	
S3. Failure of joint(s) or motor's transmission part(s)	
S4. The moving direction of motor(s) is intercepted by obstacle	
S5. Failure of motor's power amplifier	
S6. Obstacle collision	
S7. Self-collision	
W1. Over-heating of the motor's armature coils	warning
W2. Large tracking error	
W3. Overweight load or actuator saturation	

Table 4.1 A list of 18 common fault sources of mechatronic systems

4.3 DemoLin Setup

4.3.1 Introduction of DemoLin

The DemoLin setup is a simple mass-spring-mass system which was developed at Imotec BV (http://www.imotec.nl/) for the demonstration purpose of controller performances. It has a base plate (motor mass) which is driven by a linear motor and a load mass (end-effector mass) which is connected on the top of the base plate through two flexible iron plates (stiffness). The masses are attached to pretension belts and these belts are supported by pulleys mounted on two shafts that drive encoders. At the left shaft, the pulley of the lower belt is fixed while the pulley of the upper belt is connected with bearings. At the right shaft, the pulley of the upper belt is fixed and the pulley of the lower belt is connected with bearings. Therefore, the DemoLin setup can be considered as a simple single-axis electro-mechanical motion system. Note that, the DemoLin setup has the dynamic behavior of a common class of motion systems, namely those that have the dominant compliance in the transmission. A photo of the DemoLin setup is shown in figure 4.1. More information about the mechanical structure, electrical and electronic components, and the modeling of the setup are given in appendix A.1.

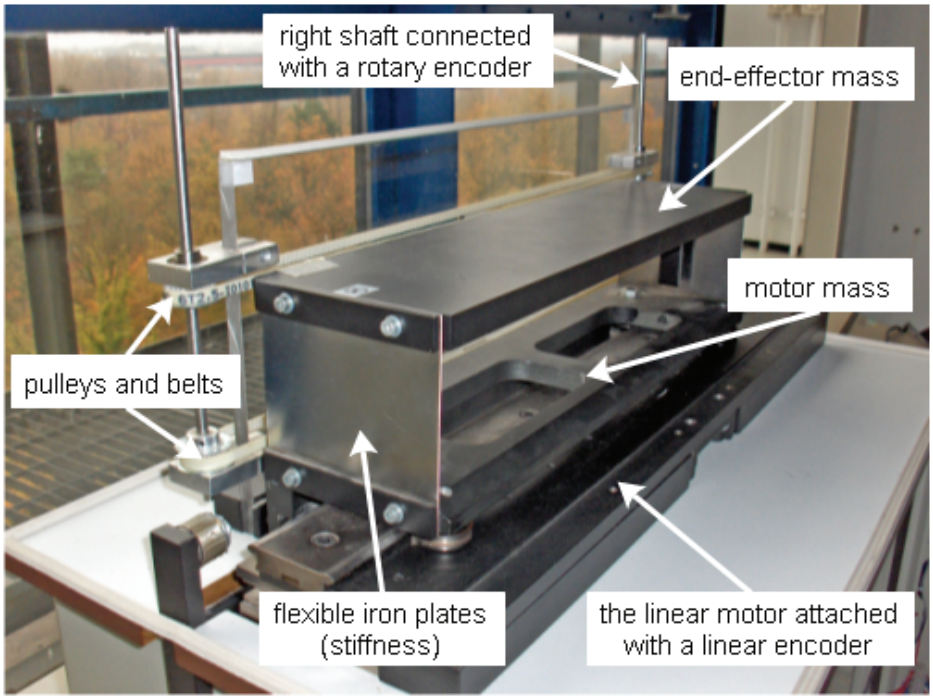


Figure 4.1 Photo of the DemoLin setup

4.3.2 Design a safe-guarded MACS for DemoLin

Before designing a safe-guarded control system for the DemoLin setup, *three particular requirements* with regard to the control system are given:

1. The control system should have multi-operation modes in which each operation mode can have a different priority level (i.e. priority-based operation), motion trajectories (periodic or non-periodic path), control system configurations (simple or advanced controller), and control missions (accurate position control or safe-guarded control). Moreover, operation modes should be designed such that multiple functionality can be deployed in each mode.

   - *Startup* (with two sub-operation modes: Initial and Homing): the Initial mode is used for calculating or locating the index pulses of incremental encoders by means of a special motion. The Homing mode has a function to bring the linear motor of the DemoLin setup to a "home position" after the index pulses are located.
   - *Normal Operation* is the mode in which several periodic strokes are performed between the home position and a certain end position with the goal to accurately position the end-effector (i.e. load mass).
   - *Shutdown* (with three sub-operation modes: Stop, Standby, and PowerOff): at the beginning of the shutdown process, the Stop mode is used to move the linear motor from a current position to a safe stop-position (which can be either the same as or different than the homing position); and then the Standby mode is realized. Its function is to keep the linear motor in standstill state at the safe stop-position for a short moment; finally, the PowerOff mode is active to cut the power supply off. After the Shutdown mode is fully completed, the system can be restarted by the operator.
   - *Safe-Guarded* is the operation mode that always has the highest operational priority level to ensure the safety for the DemoLin setup and humans working around it.

2. The control system is required to meet the desired control performances (speed of response, bandwidth, stability, overshoot, sensitivity for disturbances and parameter variations) and to guarantee safe-guarded control for both operator and machine.

3. In each operation mode, the control system should perform its mission intelligently and autonomously. In case there is not any fault, the operation modes are normally active in the sequence: Initial, Homing, Normal Operation, Stop, Standby, and PowerOff. However, in case a certain fault occurs, the Safe-Guarded mode is immediately activated to handle the fault. The Safe-Guarded mode should be equiped with capabilities such as error detection, error handling, graceful degradation, and error recovery along with different degrees of fault tolerance. Depending on the potential criticality level (dangerous, serious or warning) of the present fault, an appropriate safe-guarded activity could be applied.

A safe-guarded MACS is designed for the DemoLin setup that meets the above-mentioned requirements (see figure 4.2). We apply *a design procedure including four control system design patterns* that will be described in a top-down approach as follows:

Firstly, we use the *Single Application Agent design pattern* to initially generate the hierarchically structured safe-guarded MACS. As a result, we obtain the "MACS for DemoLin setup" that consists of a "Safe-Guarded Agent" and a "Multi-Operation Mode Agent", coordinated by a Fixed-Priority Coordinator in which the "Safe-Guarded Agent" has a higher priority level than the one of the "Multi-Operation Mode Agent".

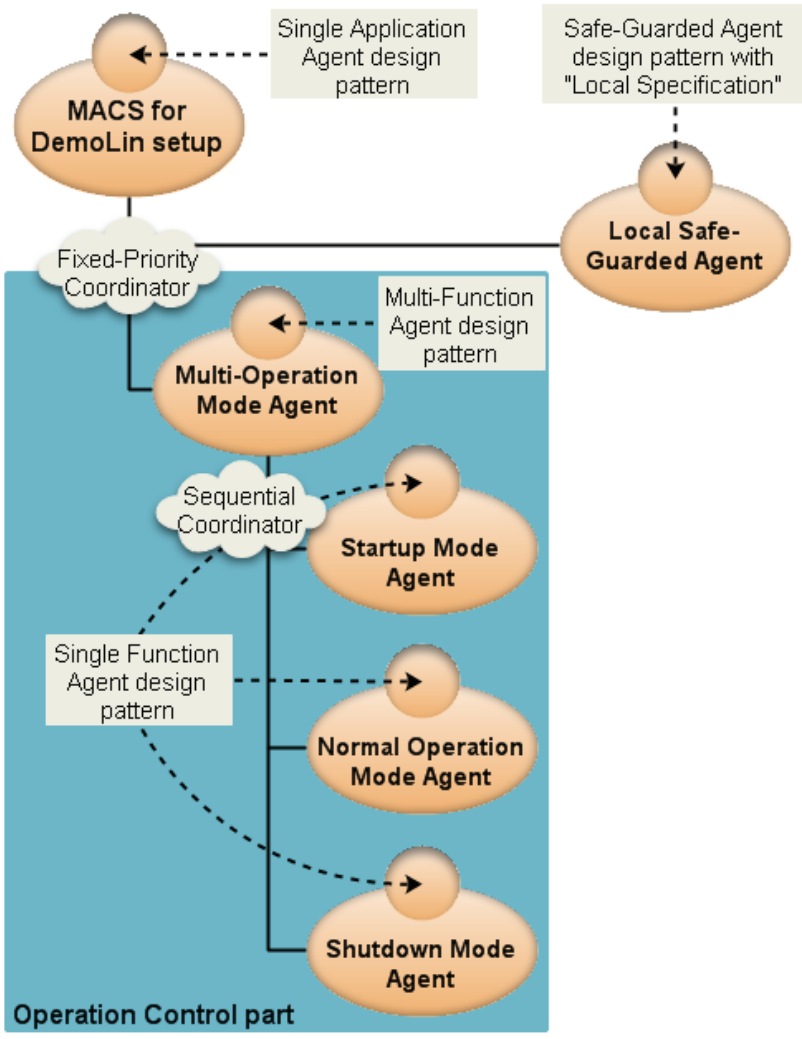


Figure 4.2 Safe-guarded MACS for the DemoLin setup

Secondly, we use the *Safe-Guarded Agent design pattern* with the *"Local Specification"* chosen to generate the hierarchical structure for the "Local Safe-Guarded Agent". As a result, the "Local Safe-Guarded Agent" consists of a "Dangerous Problem Handler", a "Serious Problem Handler", and a "Warning Problem Handler", coordinated by a Fixed-Priority Coordinator (see figure 4.3). The "Dangerous Problem Handler" has the highest priority level; the "Serious Problem Handler" is the next one; and the "Warning Problem Handler" has the lowest priority level.

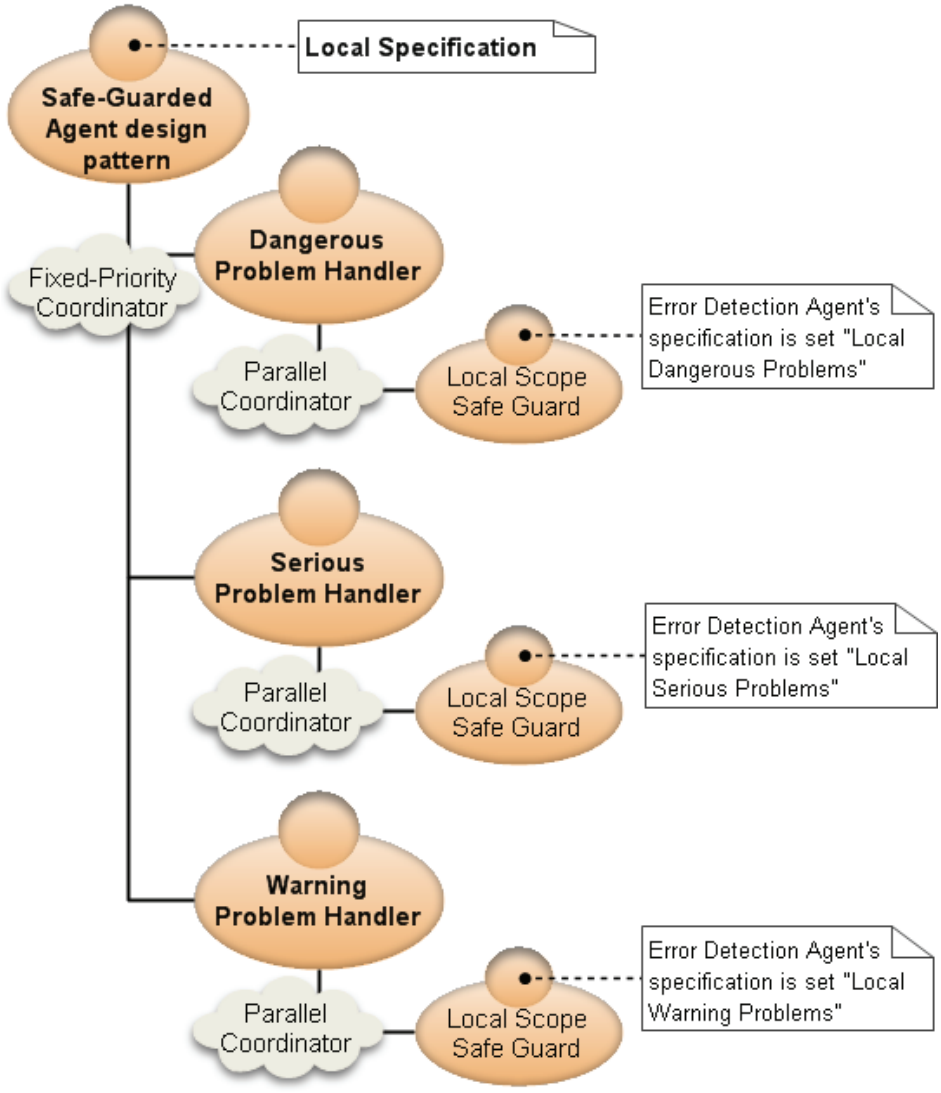


Figure 4.3 Local Specification of the Safe-Guarded Agent design pattern

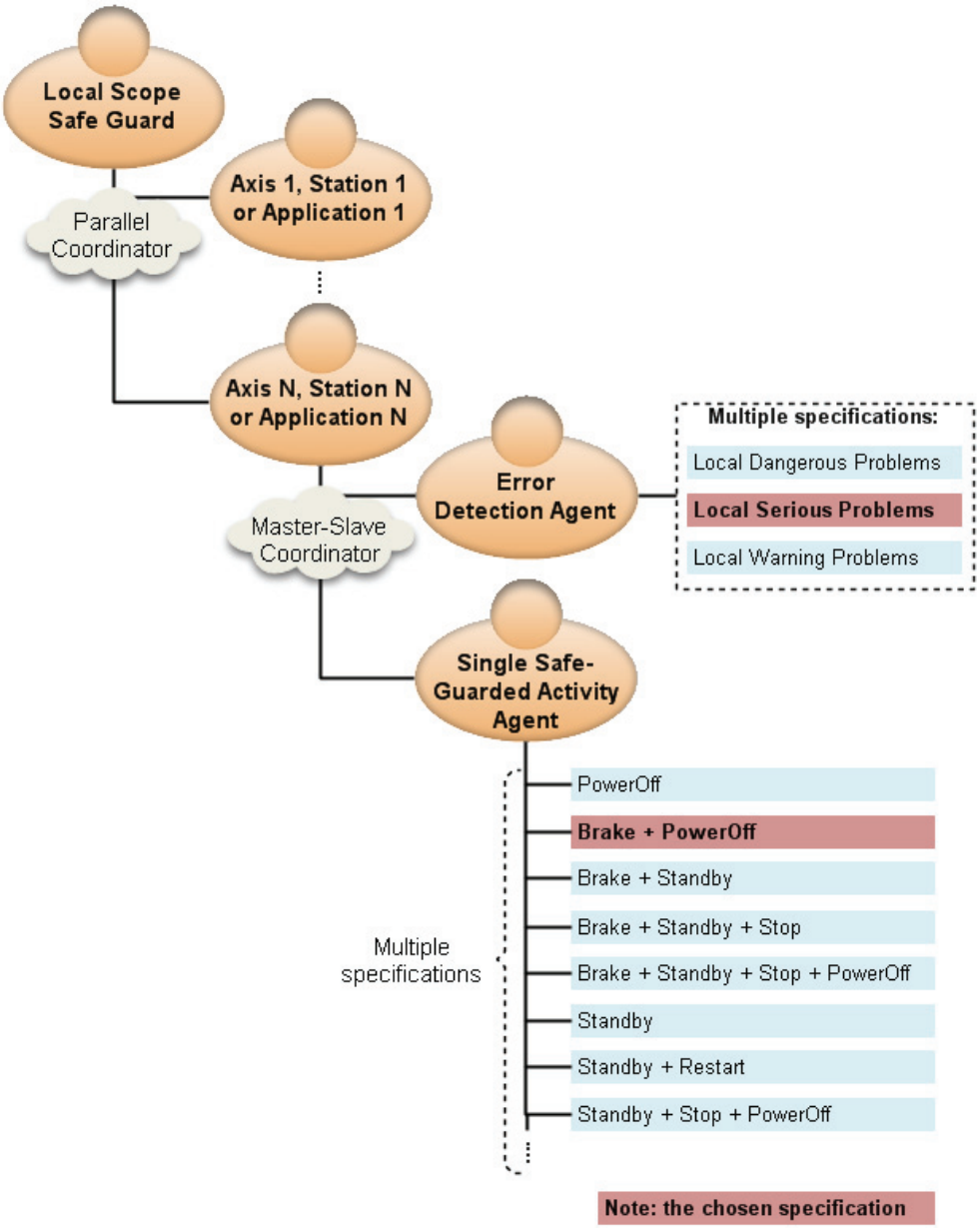


Figure 4.4 General hierarchical structure of the Local Scope Safe Guard

Moreover, each Problem Handler includes only one “Local Scope Safe Guard” with the general hierarchical structure given in figure 4.4 and it uses the default coordination type,

i.e. Parallel Coordinator. The Local Scope Safe Guard should consist of at least one Axis, Station, or Application; it can contain multiple Axes, Stations, or Applications, coordinated by a Parallel Coordinator. It means that the Local Scope Safe Guard can be applied for mechatronic systems with single or multiple axes, functions, etc. The number of Axes/Stations/Applications running in parallel depends on each specific application. In section 4.3.1, we discussed that the DemoLin setup is considered as a simple single-axis electro-mechanical motion system. Hence, in this case the Local Scope Safe Guard will be designed for only one Axis.

In the hierarchical structure of the Local Scope Safe Guard, each Axis, Station, or Application contains an "Error Detection Agent" and a "Single Safe-Guarded Activity Agent" that are coordinated by a Master-Slave Coordinator (see figure 4.4). The Error Detection Agent is the *Master-Controller Agent* with the mission to detect faults and to classify them into three criticality levels. The Single Safe-Guarded Activity Agent is the *Slave-Controller Agent* with the mission to deal with graceful degradation and error recovery issues. Therefore, it can be understood that whenever a certain fault occurs the Error Detection Agent will "wake up" the Single Safe-Guarded Activity Agent to solve the problem, i.e. the Error Detection Agent has a special role of a decision maker with regard to safe-guarded activity.

In figure 4.3, three different Problem Handlers are present but only one Local Scope Safe Guard is really existing. It is because we use *polymorphism*, which has been developed in chapter 3, to make the Local Scope Safe Guard polymorphic. It means, the Local Scope Safe Guard is a generic Type and it can hold multiple specifications or realizations. Afterwards, depending on each specific case, an appropriate specification will be selected. In this design pattern, we rely on the special role of the Error Detection Agent to formulate polymorphism of the Local Scope Safe Guard. We make the Error Detection Agent as a Type with three different specifications, being "Local Dangerous Problems", "Local Serious Problems", and "Local Warning Problems" which deal with three criticality levels of faults, i.e. dangerous, serious, and warning, respectively (see figure 4.4). Polymorphism is also applied for the Single Safe-Guarded Activity Agent such that the graceful degradation and error recovery issues can be specified flexibly through a plentiful set of multiple specifications.

In summary, the design and programming of the "Local Safe-Guarded Agent" for the DemoLin setup become a matter of configurations with regard to the Local Scope Safe Guard of three different Problem Handlers. Each configuration is done by means of selecting suitable specifications for the Error Detection Agent and Single Safe-Guarded Activity Agent. We remark here that, the selection of specifications for the Single Safe-Guarded Activity Agent varies according to personal opinion and depends on each specific application. In case of the DemoLin setup, some selections are made hereafter:

- For the Local Scope Safe Guard of the Dangerous Problem Handler: the specification "Local Dangerous Problems" is selected for the Error Detection Agent; the specification "PowerOff" is selected for the Single Safe-Guarded Activity Agent.

- For the Local Scope Safe Guard of the Serious Problem Handler: the specification "Local Serious Problems" is selected for the Error Detection Agent; the specification "Brake + PowerOff" is selected for the Single Safe-Guarded Activity Agent.
- For the Local Scope Safe Guard of the Warning Problem Handler: the specification "Local Warning Problems" is selected for the Error Detection Agent; the specification "Standby" is selected for the Single Safe-Guarded Activity Agent.

In the Safe-Guarded Agent design pattern, Problem Handlers are optionally configurable. That means, it is not required that all three Problem Handlers participate in the Local Safe-Guarded Agent structure, i.e. this structure only needs at least one Problem Handler to be operable. Nevertheless, we recommend users to design the Local Safe-Guarded Agent with all Problem Handlers because it provides such a control system with a flexible capability to handle a variety of faults with different criticality levels.

In the next step of the application of design patterns, we use the *Multi-Function Agent design pattern* to generate the hierarchical structure for the "Multi-Operation Mode Agent". As a result, it consists of a "Startup Mode Agent", a "Normal Operation Mode Agent", and a "Shutdown Mode Agent" that are coordinated by a Sequential Coordinator (see figure 4.2). The "Startup Mode Agent" is the first operation mode to be active; the next one is "Normal Operation Mode Agent"; and the last one is "Shutdown Mode Agent".

Finally, we use the *Single Function Agent design pattern* to generate the hierarchical structure for the above-mentioned operation modes. As a result, the three operation modes have the same structure as depicted in figure 4.5. The Single Function Agent consists of a "Trajectory Generator Agent", a "Feedback Controller Agent", and a "Feedforward Controller Agent" that are coordinated by a Master-Slave Coordinator. The "Trajectory Generator Agent" is the Master and the others are the Slave.

In case of the DemoLin setup, all operation modes use the same specification "only feedback" with regard to the Single Function Agent design pattern, i.e. the Feedforward Controller Agent is not present. The specification "PID with roll-off filter" is selected for the Feedback Controller Agent. The Trajectory Generator Agent consists of two Path Generator Agents, being Rising-up Path and Rising-down Path in which both paths use the same specification "2nd-degree path". However, depending on each operation mode, a different coordinator will be used in the Trajectory Generator Agent to coordinate the Rising-up Path and Rising-down Path. For example, in case of the Normal Operation mode, such a periodic motion is normally used so that the Cyclic Coordinator is applied; whereas, because of special requirements during the Startup and Shutdown mode, the Sequential Coordinator will be applied.

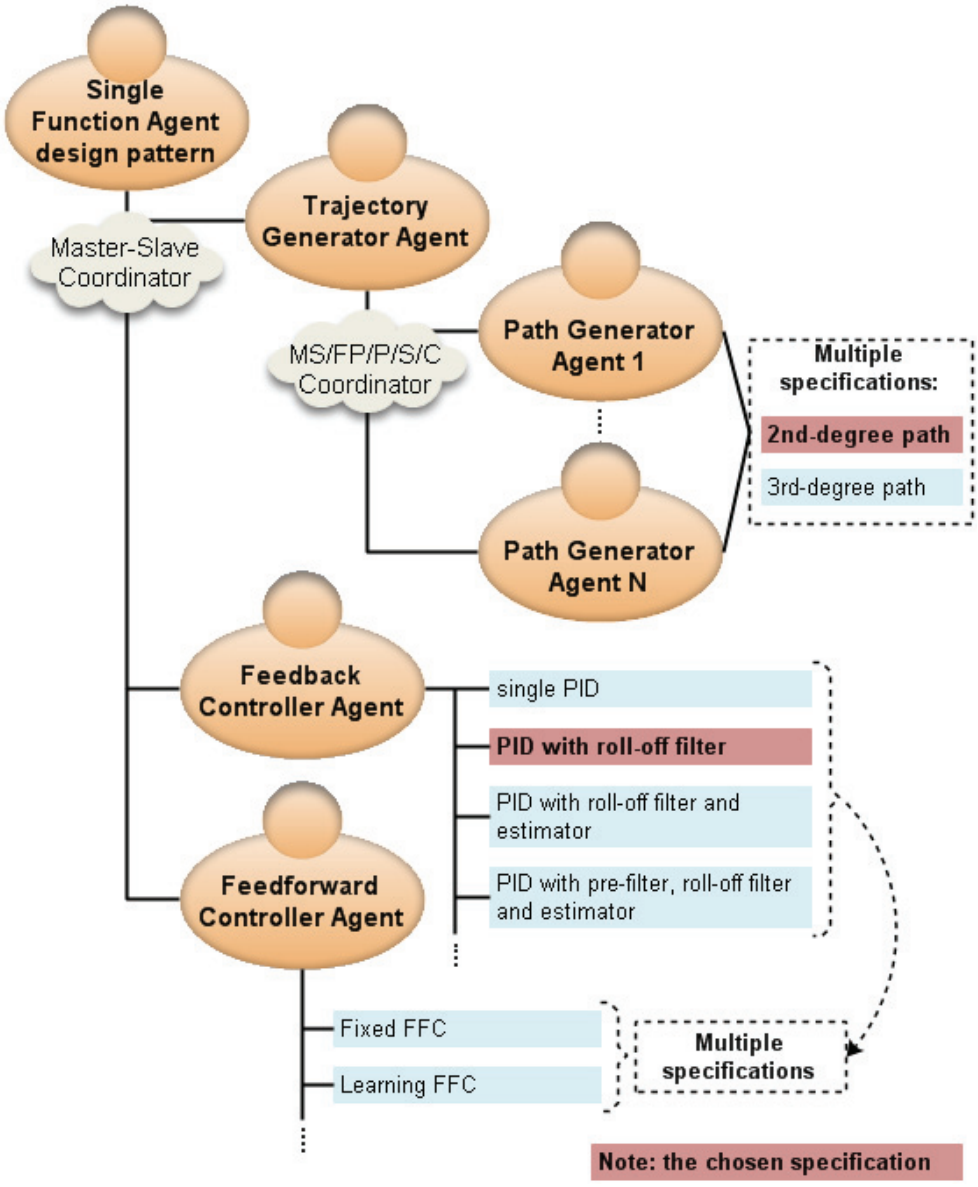


Figure 4.5 Single Function Agent design pattern

4.3.3 Cosimulation results

While designing a (safe-guarded) control system, it is desirable to test and evaluate the control system on a virtual plant (i.e. in a simulation setting) before applying it for a real plant. For the purpose of modeling and simulation of dynamic systems, some software packages such as Matlab-Simulink (The MathWorks Inc., 2009), LabVIEW (National Instruments Corporation, 2009), and 20-sim (Controllab Products B.V., 2009), can be used. In this thesis, the 20-sim software, which runs under Windows OS, is used to model and simulate plants. With 20-sim, the designer can simulate the behavior of dynamic systems, such as electrical, mechanical and hydraulic systems or any combination of these by using high-level input of models in the form of iconic diagram, bond graph, block diagram and equation models. On the side of implementing control systems, the real-time safe-guarded MACS has been developed based on TaskContext components and using the Real-Time Toolkit (RTT) of the OROCOS framework as the run-time environment. Hence, it has to run under Linux OS. As a result, an Orocos-20sim Cosimulation tool (Bozlak, 2009) was developed that forms a cosimulation environment where a real-time safe-guarded MACS, running under Linux OS, can be tested with a 20-sim simulated plant running under Windows OS. This cosimulation environment uses socket-based TCP/IP communication. In figure 4.6, the 20-sim model and additional parts of the DemoLin setup are shown. To demonstrate the safe-guarded MACS design results, two test cases are performed.

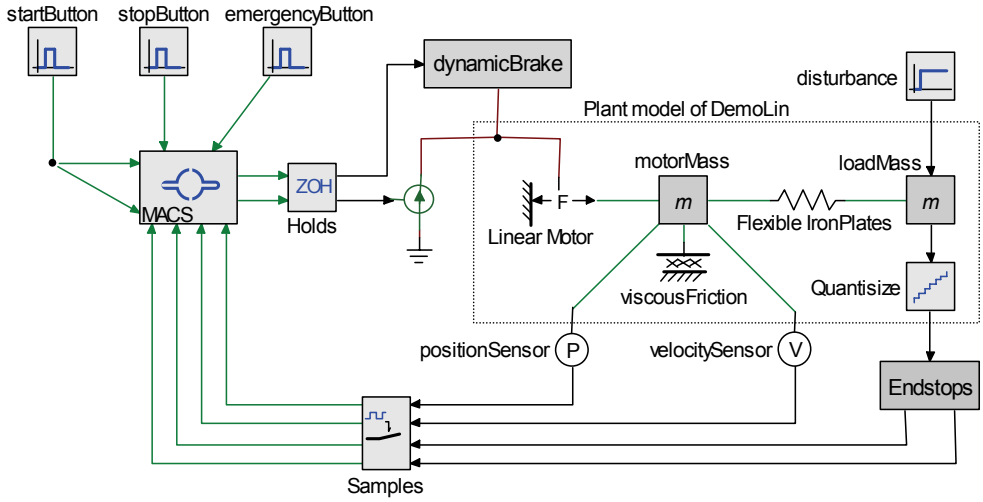


Figure 4.6 20-sim model of the DemoLin setup with safe-guarded MACS

The first test is performed in the case where the DemoLin setup runs in the normal operation condition, i.e. without any fault/error. The cosimulation results with respect to the position control issue are presented in figure 4.7. The sequence of operation modes is shown in figure 4.8. To easily observe the status transition (or switching) between

operation modes of the safe-guarded MACS, these operation modes are illustrated by mean of numbers. We implemented the safe-guarded MACS, from which it generates a unique number that is corresponding to each operation mode. This is a good way to supervise if the designed safe-guarded MACS runs correctly.

In the normal operation condition, the sequence of operation modes with the assigned numbers is:

- From t = 0 [s] to 4 [s] : Initial mode (number 1)
- From t = 4 [s] to 6 [s] : Homing mode (number 2)
- From t = 6 [s] to 24 [s] : Normal Operation mode (number 3)
- From t = 24 [s] to 27 [s] : Stop mode (number 4)
- From t = 27 [s] to 29 [s] : Standby mode (number 5)
- From t = 29 [s] to 30 [s] : PowerOff mode (number 6)

The second test case is realized in a scheme where the end-effector hits against an end-limit switch caused by a wrong position reference. After the Error Detection Agent identifies this fault as “S2” (exceeding joint working area) and classifies it as serious (see table 4.1), the Local Scope Safe Guard of the Serious Problem Handler is immediately activated to handle the fault. Note that, this fault “S2” should be detected by the fault detection algorithms which are programmed in the specification “Local Serious Problems” of the Error Detection Agent. In this case, a graceful degradation is realized by the Single Safe-Guarded Activity Agent with the specification “Brake + PowerOff” selected (see figure 4.4). The cosimulation results with regard to the position control issue are presented in figure 4.9. The sequence of operation modes is shown in figure 4.10. In this case, the Safe-Guarded mode is assigned number 7.

In case of the fault “S2”, the sequence of operation modes is:

- From t = 0 [s] to 4 [s] : Initial mode
- From t = 4 [s] to 6 [s] : Homing mode
- From t = 6 [s] to 9.62 [s] : Normal Operation mode
- From t = 9.62 [s] to 9.87 [s] : Safe-Guarded mode with dynamic braking
- From t = 9.87 [s] to end : PowerOff mode

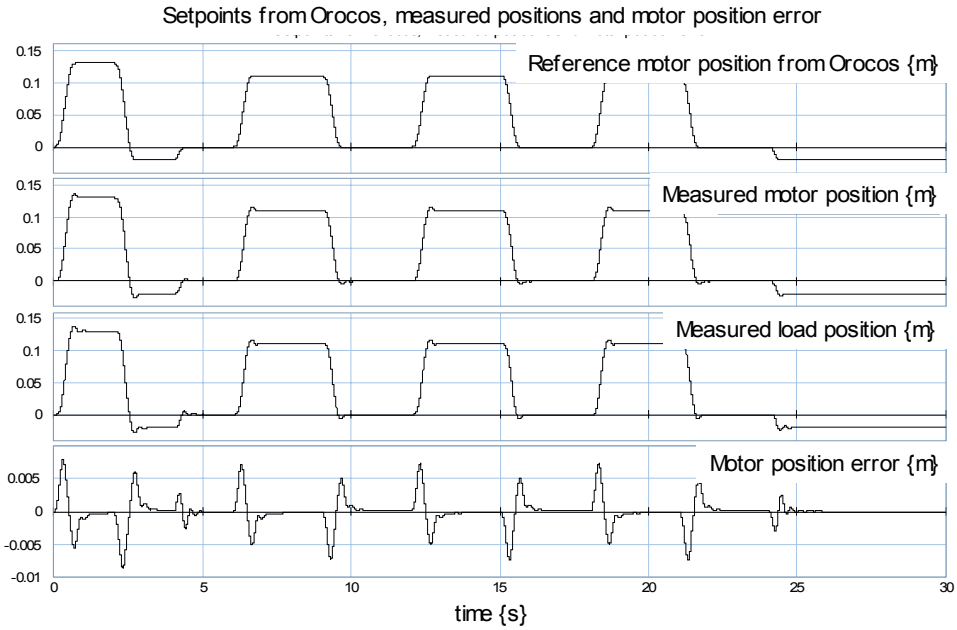


Figure 4.7 Cosimulation results in the normal operation condition

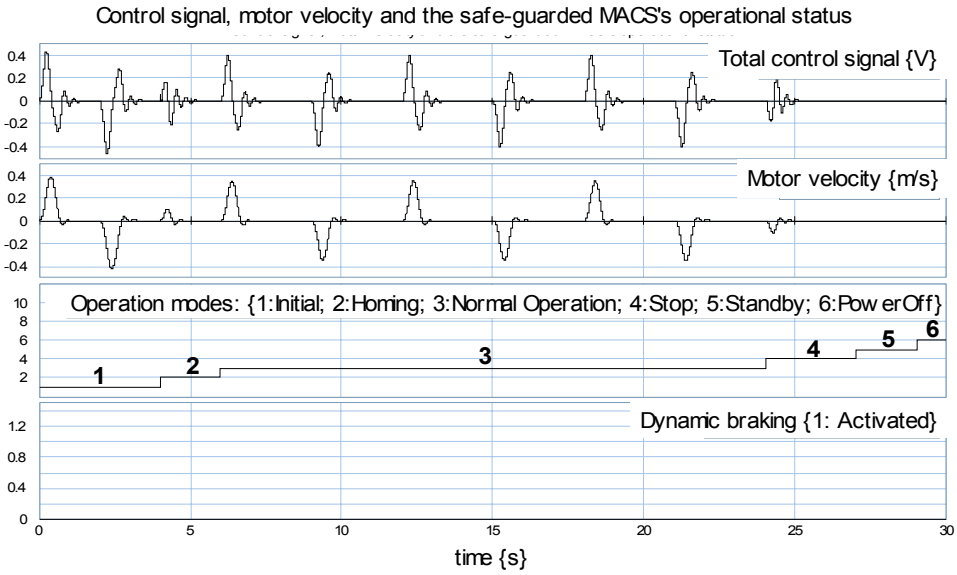


Figure 4.8 Status transitions between operation modes in the normal operation condition

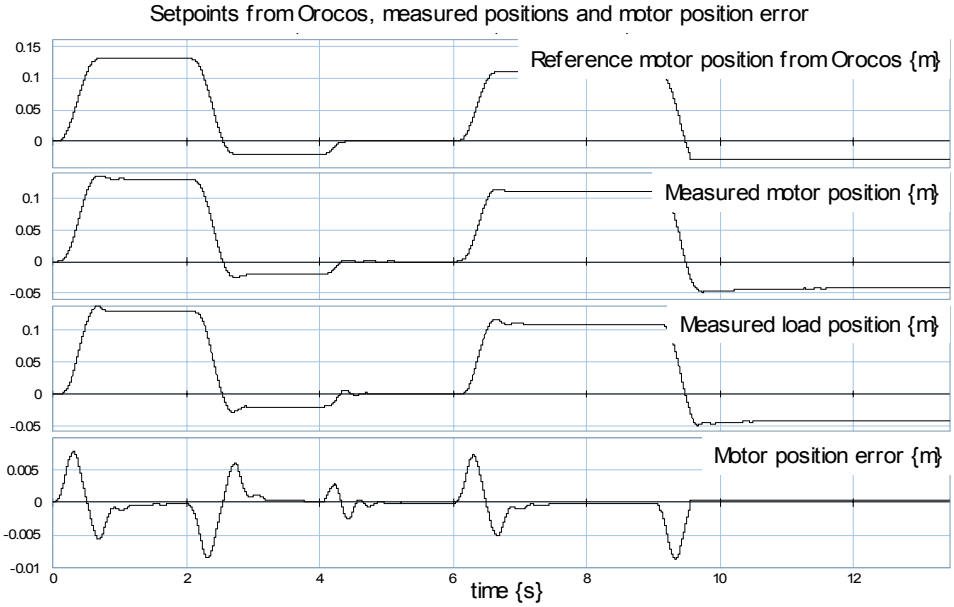


Figure 4.9 Cosimulation results in case of the fault "S2" (exceeding joint working area)

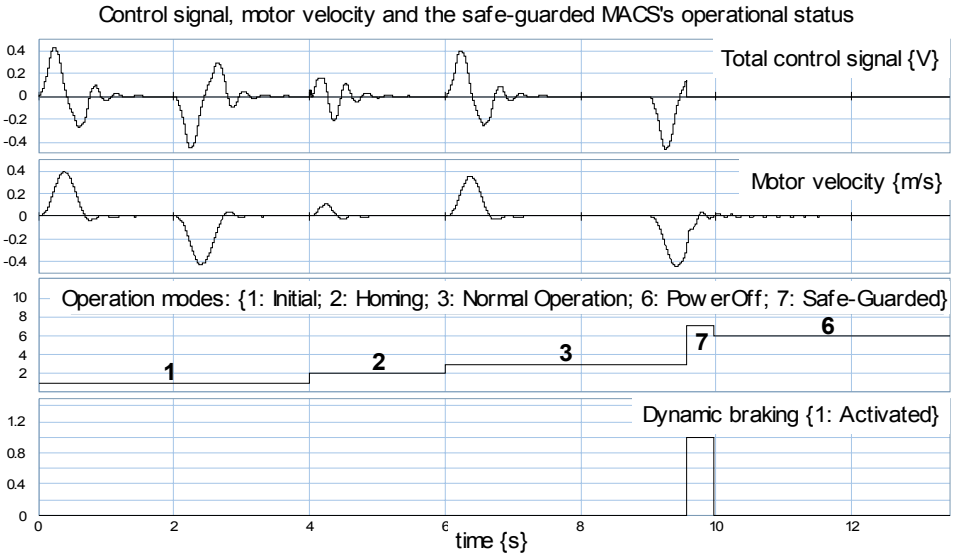


Figure 4.10 Status transitions between operation modes in case of the fault "S2" (exceeding joint working area)

4.3.4 A generalized safe-guarded control solution for simple mechatronic systems

Based on the safe-guarded MACS design procedure for the DemoLin setup, we propose a reusable generalized safe-guarded control solution (see figure 4.11) for *simple mechatronic systems* (such as single-axis manipulators or single-functionality motion systems, etc.) that is based on the structured application of four control system design patterns as follows:

Single Application Agent design pattern consists of a Safe-Guarded Agent and a Multi-Function Agent coordinated by a Fixed-Priority Coordinator in which the Safe-Guarded Agent always has a higher priority level than the Multi-Function Agent.

Safe-Guarded Agent design pattern using the *"Local Specification"* that consists of a Dangerous Problem Handler, a Serious Problem Handler, and a Warning Problem Handler coordinated by a Fixed-Priority Coordinator.

Multi-Function Agent design pattern consists of several Single Function Agents coordinated by one of the coordination types: Master-Slave (MS), Fixed-Priority (FP), Parallel (P), Sequential (S), or Cyclic (C).

Single Function Agent design pattern consists of a Trajectory Generator Agent, a Feedback Controller Agent, and a Feedforward Controller Agent coordinated by a Master-Slave Coordinator in which Trajectory Generator Agent is the Master and others are the Slave.

Note: in the Multi-Function Agent design pattern, one of the *"MS/FP/P/S/C coordinators"* is used. However, this design pattern is not limited to these five coordinators. Users can select other coordination types that are suitable for their applications (see section 3.2.1). This statement is also applied for other design patterns, which use the "MS/FP/P/S/C coordinators", are presented later in this chapter.

Discussion: the Single Function Agent design pattern can be considered as a basic control system configuration. Each realization of this design pattern creates a controlled motion profile that can be simple or complex. The complexity of a motion profile depends on the implementation of the Trajectory Generator Agent. When several realizations of the Single Function Agent are coordinated, it actually forms a realization of the Multi-Function Agent design pattern. It is possible to create a library of executable programs for machine and robot control applications in which each executable program is based on a specific realization of this Multi-Function Agent design pattern. Moreover, for each realization of the Multi-Function Agent, the designer can flexibly customize multiple specifications of the Single Function Agent and the coordination types. So that, the advantages of polymorphism are again demonstrated. At the realization phase, an executable program or a realization of the Multi-Function Agent can be used by means of selecting it from the library. We believe that the way of deploying executable programs can meet requirements of practical applications. For this reason, the Multi-Function Agent in figure 4.11 is represented by an icon with multiple layers.

An example is given to illustrate this discussion: in figure 4.2, the Multi-Operation Mode Agent of the safe-guarded MACS for DemoLin consists of three operation modes (i.e. Startup, Normal Operation and Shutdown) in which each mode is a realization of the Single Function Agent design pattern. This Multi-Operation Mode Agent, i.e. a realization of the Multi-Function Agent design pattern, is an executable program suitable for testing, initializing or calibrating a machine at first-time runs. After the calibration process finished or when the DemoLin setup is in the middle of a stable and repetitive operating process, another specific Normal Operation mode, which depends on each application, is the only one that should be present in the Multi-Operation Mode Agent. Hence, at least two versions of the Multi-Operation Mode Agent should be designed and realized.

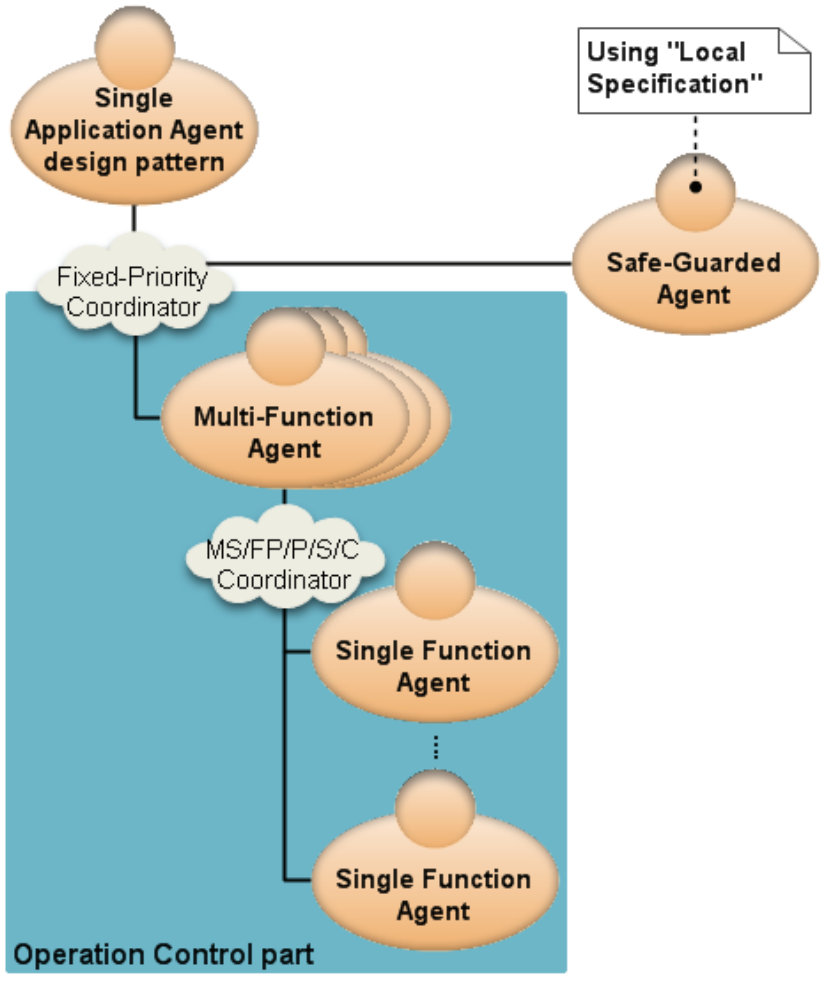


Figure 4.11 Hierarchical structure of the generalized safe-guarded control solution for simple mechatronic systems

4.4 TriPod Setup

4.4.1 Introduction of TriPod

The TriPod setup is a pick-and-place machine which was also developed at Imotec BV (http://www.imotec.nl/) for testing different types of advanced controllers. It consists of three linear motors, which can move up and down within their safe operating regions. A pair of rods is connected to each linear motor, and the other side of these rods is connected to a platform (end-effector with load) at the top. Due to the constrained movement of the rods, the platform cannot rotate but only translate. The constraints on the rods make them form a parallelogram. In the TriPod, only the positions of the three linear motors are measured. The position of the platform is determined by the positions of the three linear motors. So, the TriPod has three identical parts. Each part consists of one linear motor attached to the platform through a leg, thus forming a fourth order plant model which can be categorized as a Flexible Mechanism of type AR (Eglence, 2003). As a result, each leg of the TriPod setup has the same plant model as the DemoLin setup. Therefore, the TriPod setup can be considered as a complex multi-axis electro-mechanical motion system (three axes). Moreover, because of its specific structure, the TriPod setup has some special properties: variable load mass and variable springs due to the coupling between three axes make the load forces variable as well as the strong coupling between the end-effector space and the joint spaces of three legs. A photo and schematic view of the Tripod setup is given in figure 4.12. More information about the mechanical structure, electrical and electronic components, and the modeling of the TriPod setup are given in appendix A.2.

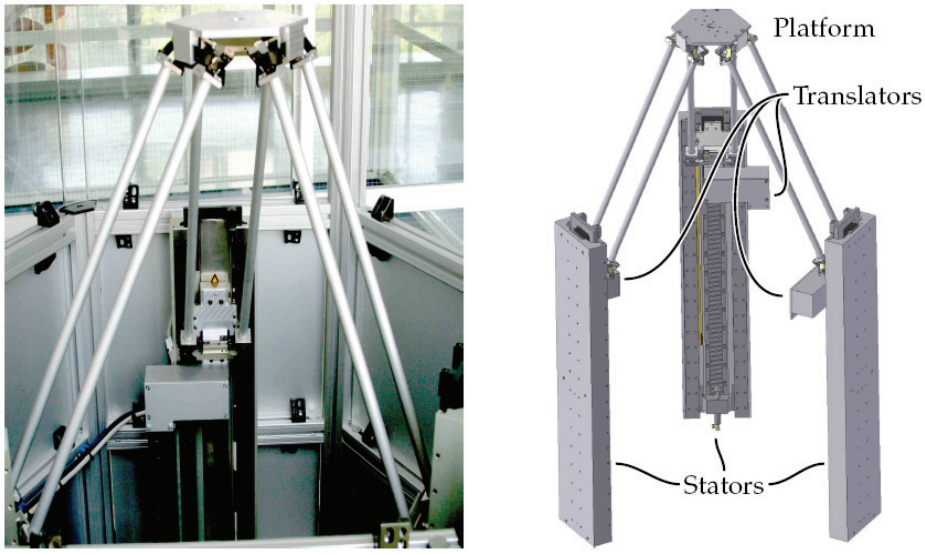


Figure 4.12 Photo and schematic view of the TriPod setup (De Kruif, 2004)

4.4.2 Design a safe-guarded MACS for TriPod

In general, the particular requirements in designing a safe-guarded control system for the TriPod setup are almost the same as the ones for the DemoLin setup (see section 4.3.2). The reason is that the safe-guarded control problem of each axis of the TriPod setup can be considered the same as the one of the DemoLin setup. However, in this case there is a new issue that should be taken into account: the safe-guarded control for multi-axis operations that is related to three axes of the TriPod setup. In order to distinguish this kind of safe-guarded control from the one for single-axis operations (like the case of the DemoLin setup), we study the safe-guarded control for multi-axis operations in a new context, i.e. a *Global influence sphere*. It means, designing a safe-guarded control system for the TriPod setup will involve both *Local safe-guarded control* and *Global safe-guarded control*.

In figure 4.13, we present a safe-guarded MACS design for the TriPod setup that meets the particular requirements. The safe-guarded MACS is implemented based on the design procedure: *apply three control system design patterns and reuse the safe-guarded MACS design results of the DemoLin setup into the design for the TriPod setup*. Next, the design procedure is explained with a top-down approach.

Firstly, we use the *System Agent design pattern* to initially generate the hierarchically structured safe-guarded MACS. As a result, we obtain "MACS for TriPod setup" that consists of a "Safe-Guarded Agent" and a "Multi-Axis Controller Agent", coordinated by a Fixed-Priority Coordinator in which the "Safe-Guarded Agent" has a higher priority level than the one of the "Multi-Axis Controller Agent".

Secondly, we use the *Safe-Guarded Agent design pattern* with the *"Global Specification"* chosen to generate the hierarchical structure for the "Global Safe-Guarded Agent". As a result, the "Global Safe-Guarded Agent" consists of a "Dangerous Problem Handler", a "Serious Problem Handler", and a "Warning Problem Handler", coordinated by a Fixed-Priority Coordinator (see figure 4.14).

Here, we make a comparison between two specifications of the Safe-Guarded Agent design pattern: both specifications have the same hierarchical structure with three different Problem Handlers coordinated by the Fixed-Priority Coordinator. However, they also have differences:

- In case of the Local Specification (figure 4.3), each Problem Handler includes only one "Local Scope Safe Guard" and uses the default coordination type, i.e. Parallel Coordinator.
- In case of the Global Specification (figure 4.14), each Problem Handler consists of a "Global Scope Safe Guard" and a "Local Scope Safe Guard" that are coordinated by a Fixed-Priority Coordinator in which the "Global Scope Safe Guard" has a higher priority level than the one of the "Local Scope Safe Guard".

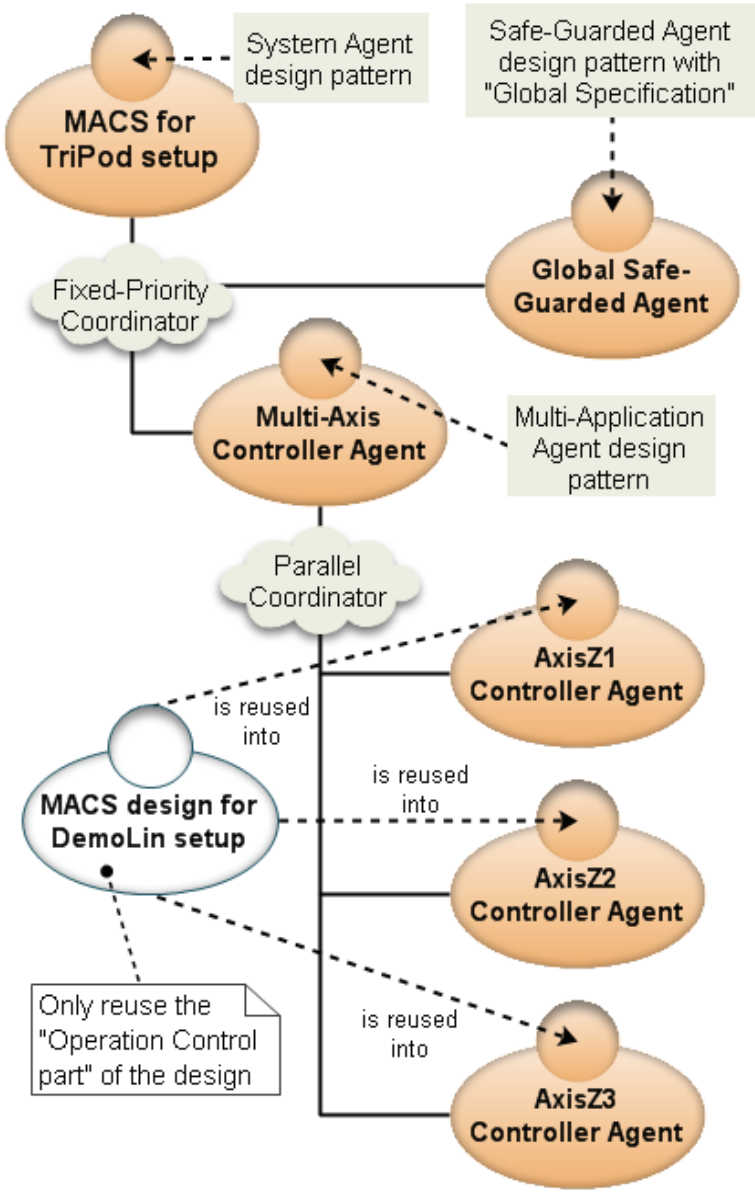


Figure 4.13 Safe-guarded MACS for the TriPod setup

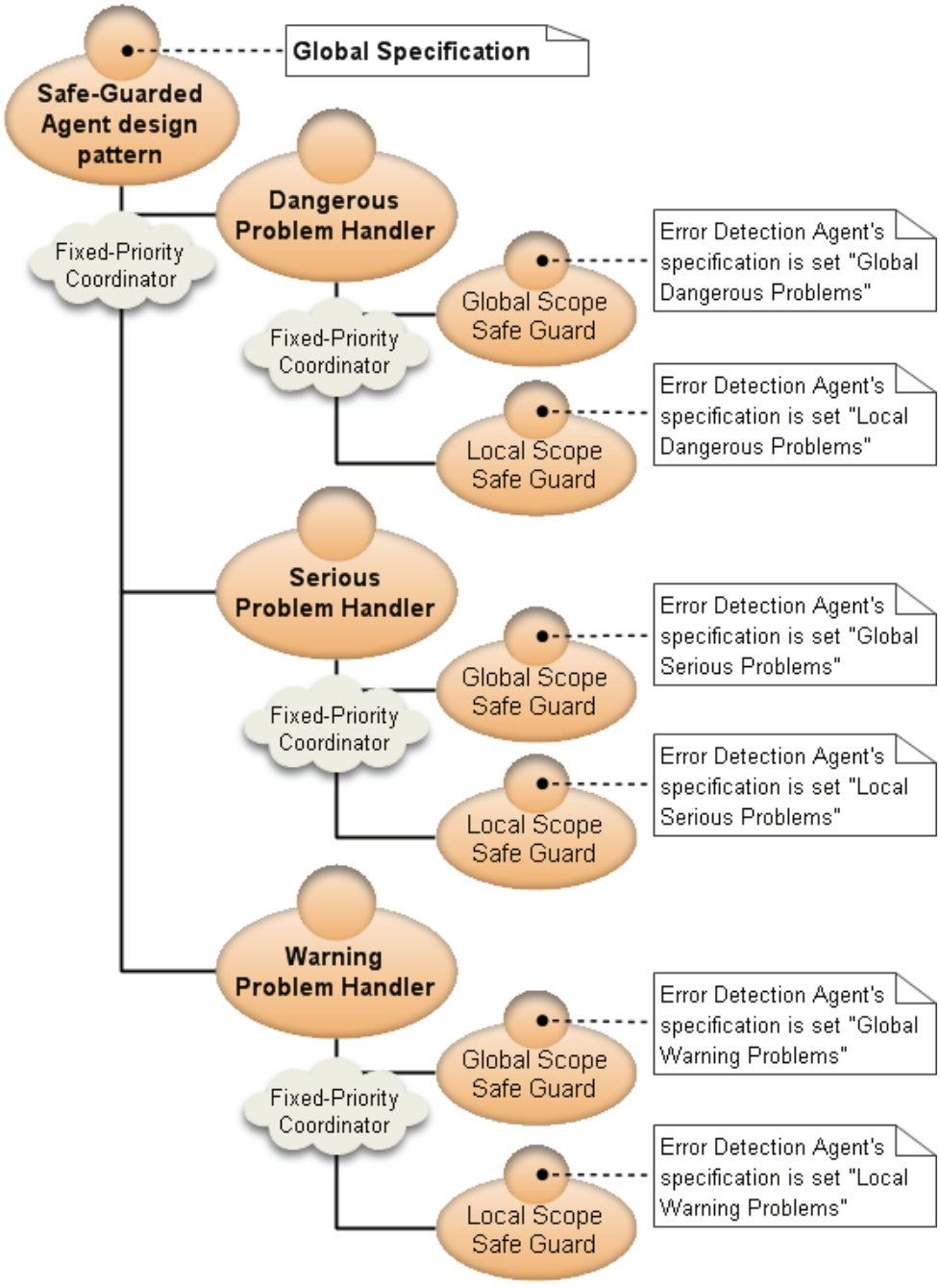


Figure 4.14 Global Specification of the Safe-Guarded Agent design pattern

The general hierarchical structure of the "Local Scope Safe Guard" is given in figure 4.4. In case of the DemoLin setup, the Local Scope Safe Guard is designed for only one Axis. In section 4.4.1, we discuss that the TriPod setup is considered as a complex three-axis electro-mechanical motion system. Therefore, in this case the Local Scope Safe Guard will be designed for three Axes, being AxisZ1, AxisZ2 and AxisZ3, in which each Axis consists of an "Error Detection Agent" and a "Single Safe-Guarded Activity Agent" that are coordinated by a Master-Slave Coordinator. The operation mechanism of the Local Scope Safe Guard and the role of the Error Detection Agent and Single Safe-Guarded Activity Agent were presented while designing the safe-guarded MACS for the DemoLin setup. In the following, we will discuss the operation mechanism of the Global Scope Safe Guard.

The Global Scope Safe Guard contains an "Error Detection Agent" and a "Multi-Safe-Guarded Activity Agent", coordinated by a Master-Slave Coordinator (see figure 4.15). The Error Detection Agent is the *Master-Controller Agent* with the mission to detect faults and to classify them into three criticality levels. The Multi-Safe-Guarded Activity Agent is the *Slave-Controller Agent* with the mission to deal with graceful degradation and error recovery issues. The operation mechanism of the Global Scope Safe Guard is the same as the one of the Local Scope Safe Guard. However, the Global Scope Safe Guard uses the Multi-Safe-Guarded Activity Agent instead of the Single Safe-Guarded Activity Agent as the Local Scope Safe Guard does. In figure 4.15, the Multi-Safe-Guarded Activity Agent is a pool of Single Safe-Guarded Activity Agents coordinated by one of five coordination types: Master-Slave (MS), Fixed-Priority (FP), Parallel (P), Sequential (S) and Cyclic (C). The different types of coordinators allow designers to flexibly choose an appropriate safe-guarded control strategy for each specific application.

Polymorphism is also used in the Global Scope Safe Guard and its all subcomponents like Error Detection Agent, Multi-Safe-Guarded Activity Agent, and Single Safe-Guarded Activity Agents. We also rely on the special role of the Error Detection Agent to formulate polymorphism of the Global Scope Safe Guard. In this case, besides three specifications already made for the Local Scope Safe Guard, other three specifications being "Global Dangerous Problems", "Global Serious Problems", and "Global Warning Problems" are created for the Global Scope Safe Guard. As a result, *the Error Detection Agent has six specifications* (see figure 4.15).

Here, we explain the difference between the Local Scope Safe Guard and the Global Scope Safe Guard:

- In the Local Scope Safe Guard: because three Axes run in parallel, the Error Detection Agent has the mission to detect and classify faults separately for each Axis. The Single Safe-Guarded Activity Agent also has the mission to deal with graceful degradation and error recovery issues for separate Axis.
- In the Global Scope Safe Guard: the Error Detection Agent has the mission to detect and classify faults that are related to all three Axes. The Multi-Safe-Guarded Activity Agent has the mission to deal with graceful degradation and error recovery issues for all three Axes.

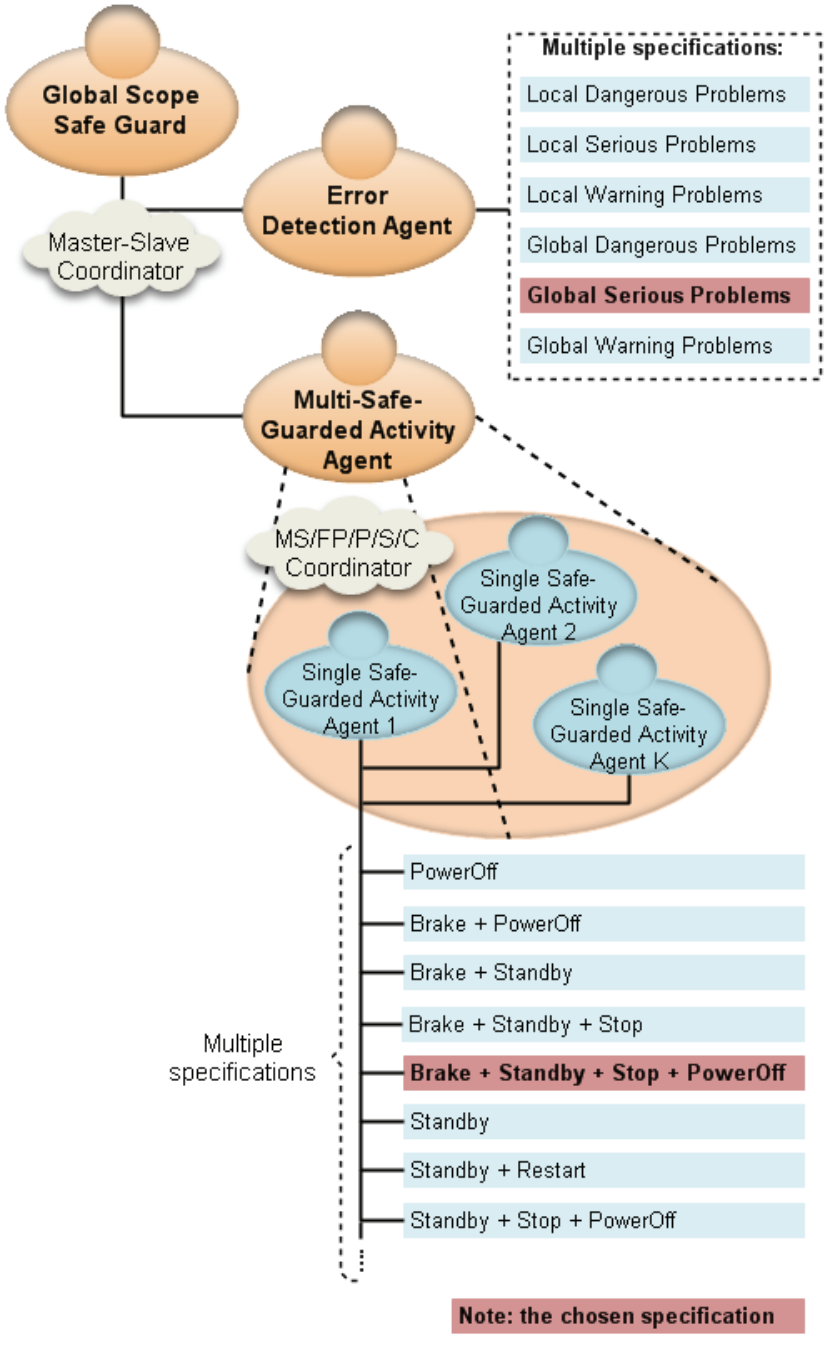


Figure 4.15 General hierarchical structure of the Global Scope Safe Guard

In summary, the design and programming of a safe-guarded MACS for the TriPod setup become a matter of configurations with regard to the Local Scope Safe Guard and Global Scope Safe Guard of three different Problem Handlers, in which:

- A configuration of the Local Scope Safe Guard (figure 4.4) is done by means of selecting suitable specifications for the Error Detection Agent and Single Safe-Guarded Activity Agent for each Axis (AxisZ1, AxisZ2 and AxisZ3).
- A configuration of the Global Scope Safe Guard (figure 4.15) is done through selecting appropriate specifications for the Error Detection Agent and three Single Safe-Guarded Activity Agents of the Multi-Safe-Guarded Activity Agent.

In case of the TriPod setup, the configurations of the Local Scope Safe Guard are selected the same as the ones for the DemoLin setup and thus will be reused into the current design for the TriPod setup. As a result, only the configurations of the Global Scope Safe Guard need to be specified:

- For the Global Scope Safe Guard of the Dangerous Problem Handler: the specification "Global Dangerous Problems" is selected for the Error Detection Agent; the same specification "Brake + PowerOff" is selected for three Single Safe-Guarded Activity Agents of the Multi-Safe-Guarded Activity Agent.
- For the Global Scope Safe Guard of the Serious Problem Handler: the specification "Global Serious Problems" is selected for the Error Detection Agent; the same specification "Brake + Standby + Stop + PowerOff" is selected for three Single Safe-Guarded Activity Agents of the Multi-Safe-Guarded Activity Agent.
- For the Global Scope Safe Guard of the Warning Problem Handler: the specification "Global Warning Problems" is selected for the Error Detection Agent; the same specification "Standby + Restart" is selected for three Single Safe-Guarded Activity Agents of the Multi-Safe-Guarded Activity Agent.

In the next step of the application of design patterns, we use the *Multi-Application Agent design pattern* to generate the hierarchical structure for the "Multi-Axis Controller Agent". As a result, the "Multi-Axis Controller Agent" consists of a "AxisZ1 Controller Agent", a "AxisZ2 Controller Agent", and a "AxisZ3 Controller Agent", coordinated by a Parallel Coordinator.

Finally, we *reuse the safe-guarded MACS design results of the DemoLin setup into the design for the TriPod setup*. Particularly, the "Operation Control part" of the design can be fully reused into each axis of the TriPod setup (see figure 4.13). It means that AxisZ1 Controller Agent, AxisZ2 Controller Agent, and AxisZ3 Controller Agent can use this Operation Control design with the same hierarchical structure as the DemoLin setup. *The only thing that remains to be done is to modify application-specific settings* (e.g. error bounds, trajectory, controller parameters, coordinators).

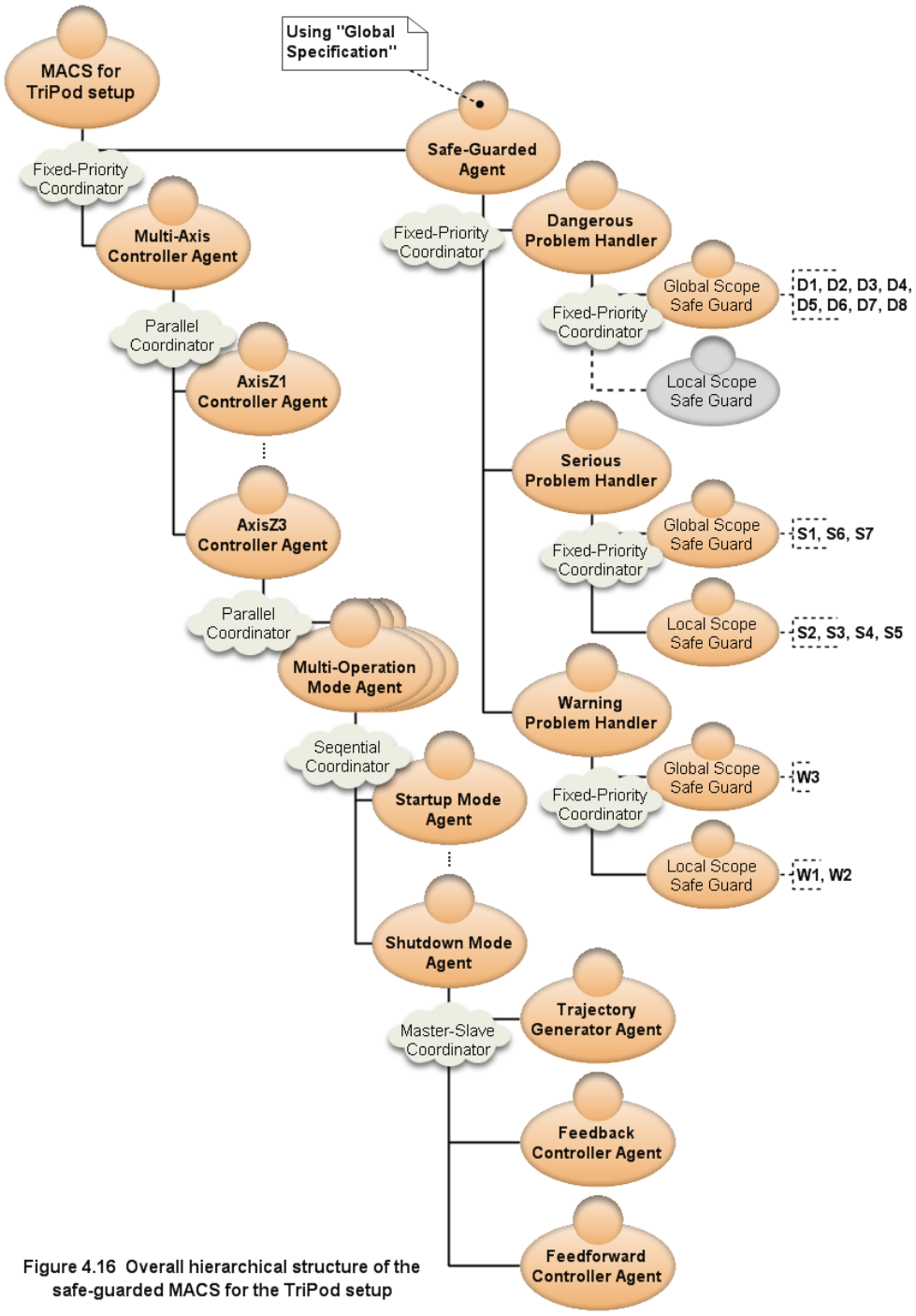


Figure 4.16 Overall hierarchical structure of the safe-guarded MACS for the TriPod setup

Here, we will discuss how to map the list of 18 common fault sources of mechatronic systems (see table 4.1) into Problem Handlers of the Safe-Guarded Agent. As above discussed, we know that the Error Detection Agent has the role of the Master-Controller Agent with the missions to detect faults and to classify them into *three criticality levels* (Dangerous, Serious, or Warning) and *two influence spheres* (Local or Global); and then it decides which Single or Multi-Safe-Guarded Activity Agent should be activated to handle the fault. As a result, this activity indirectly defines the Local or Global Scope Safe Guard of which Problem Handler will become active. It means that the categorization of fault sources decides how algorithms for detecting and classifying faults are implemented in six specifications of the Error Detection Agent. In table 4.2, we present a particular categorization of 18 common fault sources for the TriPod case study. When moving to a new application, the categorization may be different. The design procedure and approach that we use make it easy for designers to re-categorize fault sources for other mechatronic systems. In figure 4.16, an overall hierarchical structure of the safe-guarded MACS for the TriPod setup is presented together with the categorization of fault sources in table 4.2.

Fault sources	Six specifications of the Error Detection Agent
D1, D2, D3, D4, D5, D6, D7, D8	Global Dangerous Problems
S1, S6, S7	Global Serious Problems
W3	Global Warning Problems
not in use in this case	Local Dangerous Problems
S2, S3, S4, S5	Local Serious Problems
W1, W2	Local Warning Problems

Table 4.2 Categorization of 18 common fault sources into six specifications of the Error Detection Agent in case of the TriPod setup

Discussion of the propagation between Problem Handlers

Before starting this discussion, we remark here one thing: as presented, each Problem Handler consists of a Global Scope Safe Guard and a Local Scope Safe Guard. To easily distinguish between them, we use a way of naming that combines both the influence sphere and criticality level. For example, the *Global* and *Local* Scope Safe Guard of the *Dangerous Problem Handler* will be called Global Dangerous Problem Handler and Local Dangerous Problem Handler, respectively. So, a remark is given: because of the Global and Local Scope, the Safe-Guarded Agent actually has six Problem Handlers instead of three.

The Safe-Guarded Agent has *six different Problem Handlers* which are represented by six specifications of the Error Detection Agent. Hence, a maximum propagation can run from the Local Warning Problem Handler (i.e. thought as the lowest level) to the Global Dangerous Problem Handler (i.e. considered as the highest level). It is not required that all six Problem Handlers participate in the Safe-Guarded Agent structure because they are optionally configurable. However, we recommend users to design the Safe-Guarded Agent

with all Problem Handlers because it can provide the safe-guarded MACS with a flexible capability to handle a variety of faults with different criticality levels through the propagation between Problem Handlers.

In summary, the Safe-Guarded Agent provides: (i) *the local measure* involving the Local Warning Problem Handler, Local Serious Problem Handler and Local Dangerous Problem Handler; and (ii) *the global measure* involving the Global Warning Problem Handler, Global Serious Problem Handler and Global Dangerous Problem Handler.

Propagations between Problem Handlers depend on two things:

- The propagations of influence spheres of faults, i.e. from Local to Global Scope.
- The propagations of criticality levels of faults, i.e. from Warning to Serious, from Serious to Dangerous, and from Warning to Dangerous.

Comments:

- The Safe-Guarded Agent is implemented in the form of six different Problem Handlers, thus being *clear to understand, simple to apply, and easy to reuse*. Moreover, compared with a control system whose safe-guarded part is placed in a single component or module, this solution helps to *reduce the complexity* in designing and implementing the safe-guarded part. The reason is because *safe-guarded activities* are normally based on four steps: (i) gathering the information from all sensors (i.e. input ports); (ii) analyzing the data to detect and classify faults (advanced algorithms are sometimes needed for this work); (iii) making a decision of which safe-guarded solution (i.e. graceful degradation and/or error recovery measures) should be used; (iv) selecting suitable actuators (i.e. output ports) on which the safe-guarded solution will be activated. The steps really are huge work, particularly in complex control systems with many inputs and many outputs (i.e. *MIMO systems*). Therefore, instead of being struggling with the whole complex work in a single component, *the divide-and-conquer approach* we apply here for the Safe-Guarded Agent is a good choice.
- The Safe-Guarded Agent is not fixed or closed; it is *flexible and open* for users to decide their safe-guarded control strategy. Depending on specific requirements of each application, the user can design all safe-guarded activities (such as error detection, error handling, graceful degradation, and error recovery) at the local measure or distribute them at the local and global measure. However, using both local and global measure helps to handle faults in a structured way: *the safe-guarded activities are hierarchically organized with respect to the hierarchy of faults* (i.e. the influence sphere and criticality level). When a certain fault occurs, if the local measure can handle this fault then the global measure does not need to be activated. On the contrary, if the local measure cannot handle the problem then the global measure will be immediately activated. For example, in a system with several machines operating concurrently; when a fault occurs at a machine, if the local measure of this machine can detect, classify and solve the fault by means of a

graceful degradation and might be successful in an error recovery then other machines can keep operating as normal.

In the next section, we present the TriPod case study that will show a special propagation from the Local Serious Problem Handler to the Global Serious Problem Handler.

4.4.3 Cosimulation results

In this part, we present cosimulation results between the designed real-time safe-guarded MACS running under Linux OS and the 20-sim simulated plant of the TriPod setup (see figure 4.17) running under Windows OS. Two test cases are performed.

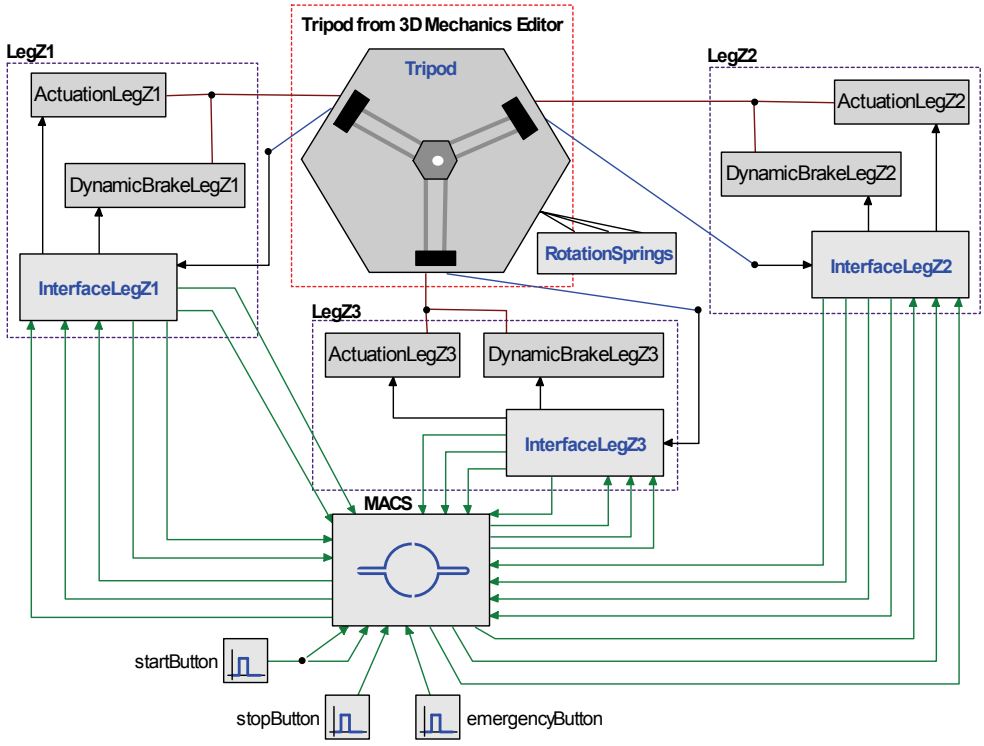


Figure 4.17 20-sim model of the TriPod setup with safe-guarded MACS

It is similar to the DemoLin case study, the status transition (or switching) between operation modes of the safe-guarded MACS for the TriPod setup is also illustrated by mean of numbers. The purposes of this method is to supervise if the designed safe-guarded MACS runs correctly and to distinguish between the cases where the same operation mode

but running in either the Local or Global context. It is also useful to distinguish between different criticality levels of faults (Dangerous, Serious, or Warning). For example, the Single Safe-Guarded Activity Agent of the Local Scope Safe Guard operates with the specification *"Brake + PowerOff"* (see figure 4.4); whereas, three Single Safe-Guarded Activity Agents of the Multi-Safe-Guarded Activity Agent of the Global Scope Safe Guard operates with the specification *"Brake + Standby + Stop + PowerOff"* (see figure 4.15). Therefore, in this case the operation of Brake and PowerOff mode should be distinguished between the Local and Global context. In table 4.3, each operation mode is assigned a unique number and with a brief explanation. A detailed information about the operation modes is given in section 4.3.2.

no.	Operation modes	no.	Operation modes
01	Initial mode	09	StandBy mode
02	Homing mode	10	StandBy mode runs as a part of the Local Warning Problem Handler
03	Normal Operation mode	11	StandBy mode runs as a part of the Global Warning Problem Handler
04	Stop mode	12	Dynamic Brake is activated in the Local Dangerous Problem Handler
05	PowerOff mode runs as a part of the Local Dangerous Problem Handler	13	Dynamic Brake is activated in the Global Dangerous Problem Handler
06	PowerOff mode runs as a part of the Global Dangerous Problem Handler	14	PowerOff mode runs in the normal operation sequence
07	Dynamic Brake is activated in the Local Serious Problem Handler	15	Restart mode is activated in the Local context
08	Dynamic Brake is activated in the Global Serious Problem Handler	16	Restart mode is activated in the Global context

Table 4.3 The operation modes are assigned numbers

The first test is implemented in the case where the TriPod setup runs in the normal operation condition, i.e. without any fault/error. The cosimulation results with regard to the position control issue are presented in figure 4.18. The sequence of the operation modes of all three axes in the case "free of error" is shown in figure 4.19 with the information:

- From t = 0 [s] to 2 [s] : Initial mode
- From t = 2 [s] to 7 [s] : Homing mode
- From t = 7 [s] to 17 [s] : Normal Operation mode
- From t = 17 [s] to 19 [s] : Stop mode
- From t = 19 [s] to 20 [s] : Standby mode
- From t = 20 [s] to 21 [s] : PowerOff mode

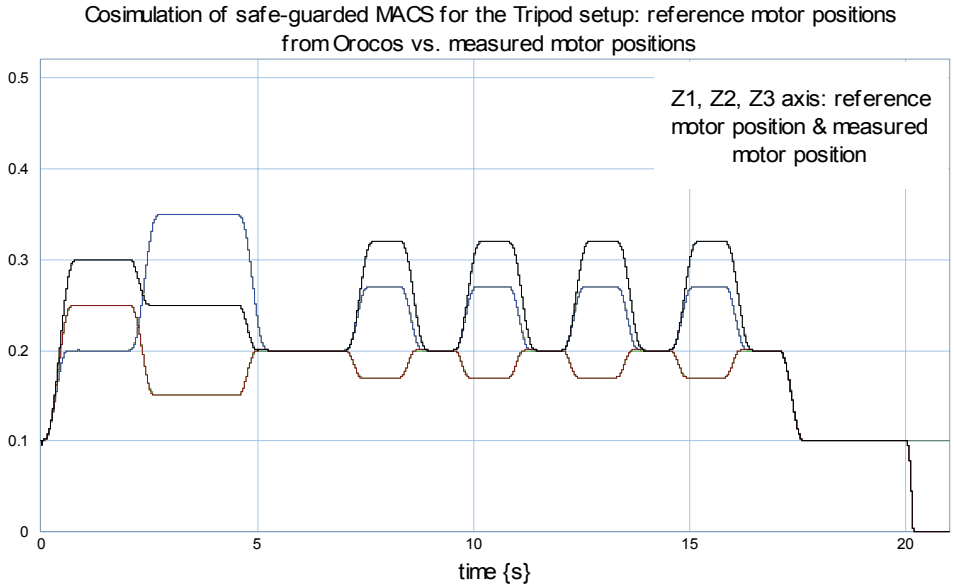


Figure 4.18 Cosimulation results in the normal operation condition

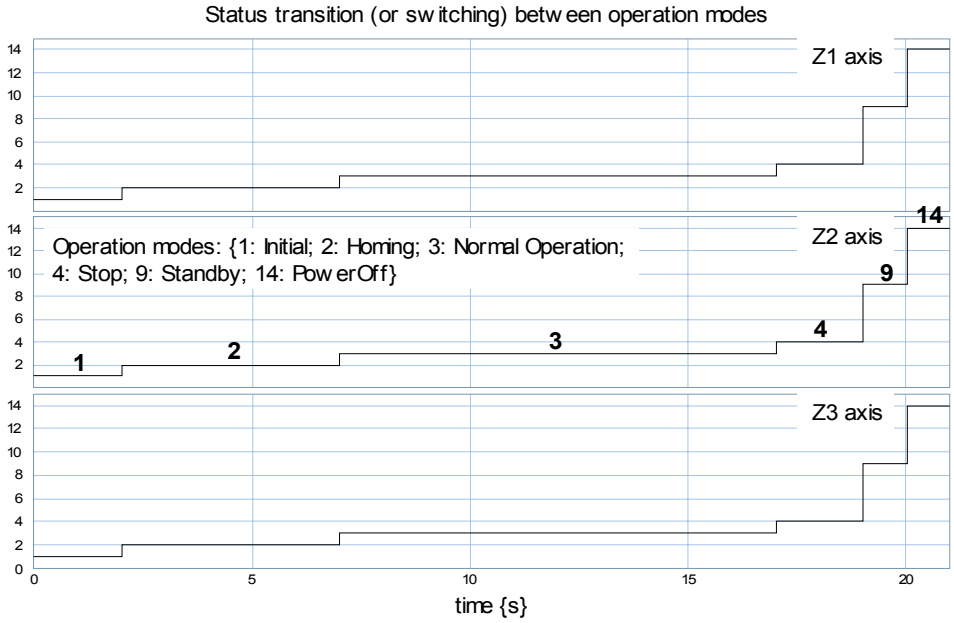


Figure 4.19 Status transitions between operation modes in the normal operation condition

The second test case is realized in a scheme where the 1st linear motor works with a wrong trajectory (it goes down too much and hits against the lower end-limit switch). First, the Error Detection Agent detects this fault as "S2" (exceeding joint working area) occuring to the 1st linear motor and classifies it as *serious* (see table 4.1) with the *Local influence sphere*. Note that, this fault "S2" should be caught by the fault detection algorithms which are implemented in the specification "Local Serious Problems" of the Error Detection Agent (see table 4.2). Then, the Error Detection Agent decides that the *Local* Scope Safe Guard of the *Serious Problem Handler* (i.e. the Local Serious Problem Handler) should be immediately activated to handle the fault (see figure 4.16). In this case, a graceful degradation is performed on the first axis of the TriPod setup through the Single Safe-Guarded Activity Agent of the Local Scope Safe Guard with the specification "Brake + PowerOff" chosen (see figure 4.4).

However, the fault "S2", which has occured to the first axis, indirectly causes another fault for the TriPod setup: it makes the end-effector move out of the safe working area. The Error Detection Agent identifies this fault as "S1" (exceeding end-effector working area) and classifies it as *serious* (see table 4.1) with the *Global influence sphere*. It means that the fault "exceeding end-effector working area" must be related to all three axes of the TriPod setup. Note that, this fault "S1" should be detected by the fault detection algorithms which are implemented in the specification "Global Serious Problems" of the Error Detection Agent (see table 4.2). Then, the Error Detection Agent decides that the *Global* Scope Safe Guard of the *Serious Problem Handler* (i.e. the Global Serious Problem Handler) should be immediately activated to handle this fault. In this case, a graceful degradation is concurrently performed with respect to all three axes of the TriPod setup through the Multi-Safe-Guarded Activity Agent of the Global Scope Safe Guard with the specification "Brake + Standby + Stop + PowerOff" chosen for three Single Safe-Guarded Activity Agents (see figure 4.15).

To illustrate the propagation between Problem Handlers of the safe-guarded MACS designed for the TriPod setup, we use a "zoom in" figure that shows a switching from the Local Serious Problem Handler (assigned number 7) to the Global Serious Problem Handler (assigned number 8) on the Z1 axis (see figure 4.20). This special propagation actually is a switching in the Serious Problem Handler with regard to the Z1 zxis, from the Local Scope Safe Guard, that handles the fault "S2", to the Global Scope Safe Guard, that handles the fault "S1" (see figure 4.16).

On the Z2 and Z3 axis, there is a direct transition from the Normal Operation mode (assigned number 3) to the Global Serious Problem Handler (assigned number 8) of the Safe-Guarded mode. This switching is because of the fault "S1" (exceeding end-effector working area).

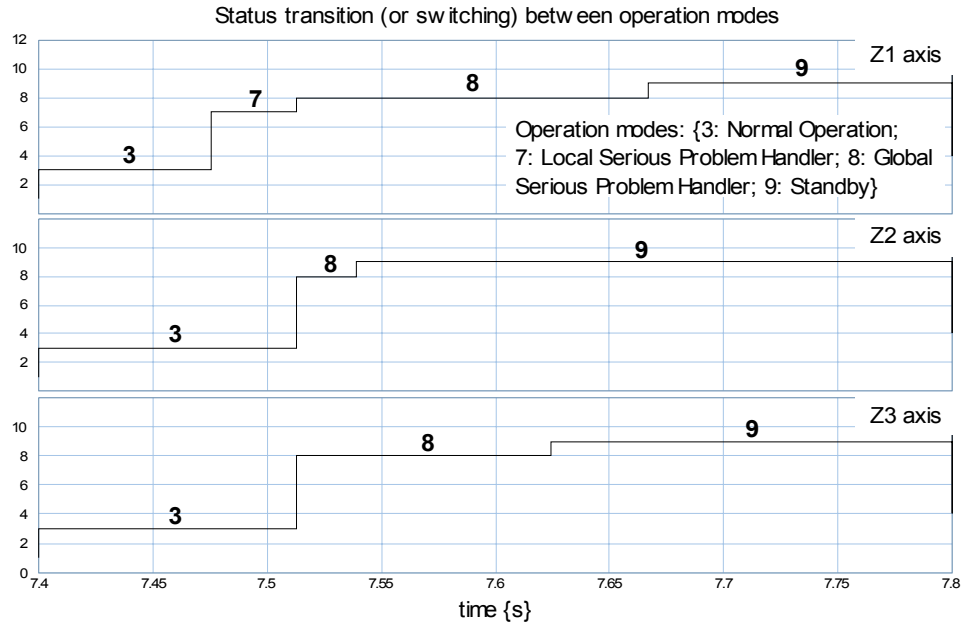


Figure 4.20 Switching between operation modes in case of the two consecutive faults "exceeding joint working area" and "exceeding end-effector working area"

4.4.4 A generalized safe-guarded control solution for complex mechatronic systems

Based on the safe-guarded MACS design procedure for the TriPod setup, we propose a reusable generalized safe-guarded control solution (see figure 4.21) for *complex mechatronic systems* (e.g. multi-robot or multi-manipulator stations with multi-operation modes, multiple functionality, etc.) that is based on two things:

1. The structured application of three control system design patterns hereafter:

System Agent design pattern consists of a Safe-Guarded Agent and a Multi-Application Agent coordinated by a Fixed-Priority Coordinator in which the Safe-Guarded Agent always has a higher priority level than the Multi-Application Agent.

Safe-Guarded Agent design pattern using the *"Global Specification"* that consists of a Dangerous Problem Handler, a Serious Problem Handler, and a Warning Problem Handler coordinated by a Fixed-Priority Coordinator.

Multi-Application Agent design pattern consists of several Single Application Agents coordinated by one of five coordinator types: Master-Slave (MS), Fixed-Priority (FP), Parallel (P), Sequential (S), or Cyclic (C).

2. Reuse the *"Operation Control part"* of the generalized safe-guarded control solution for simple mechatronic systems (see figure 4.11) into the current safe-guarded MACS design for a new complex mechatronic system. Specifically, this Operation Control part, which actually is the realizations of the Multi-Function Agent design pattern, can be fully reused into the Single Application Agents of a new design.

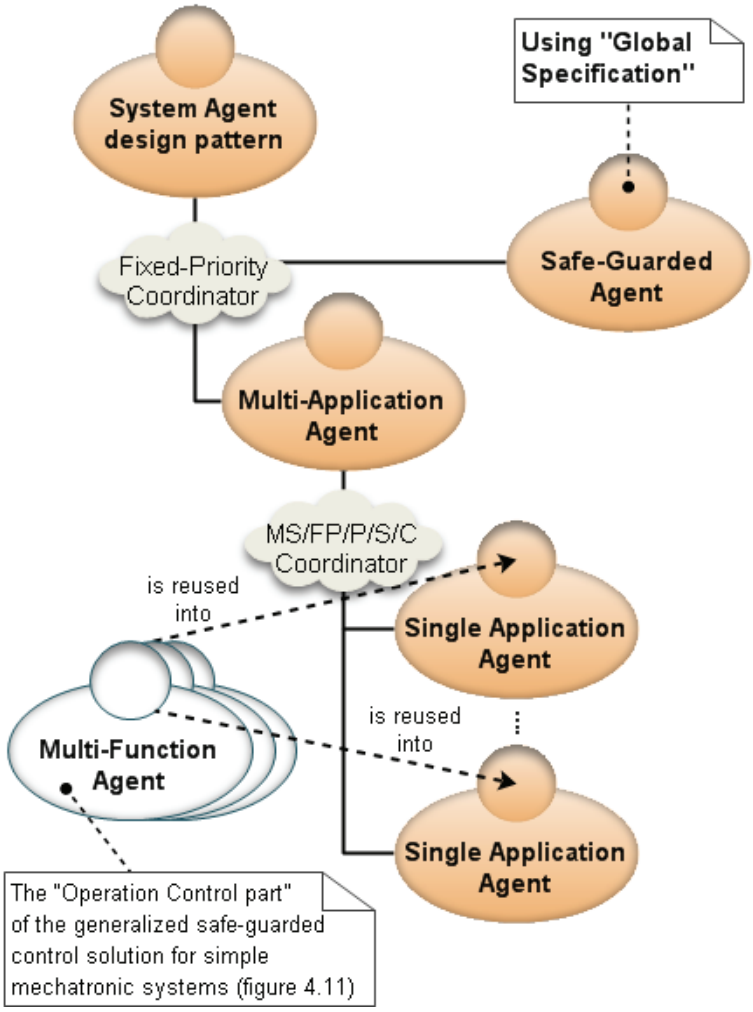


Figure 4.21 Hierarchical structure of the generalized safe-guarded control solution for complex mechatronic systems

4.5 Control System Design Patterns

In summary, nine control system design patterns are formulated and categorized into four levels in terms of a *structure-based classification*. The four levels are clearly described in the form of a hierarchical structure in figure 4.22.

1. **System level**, which consists of one design pattern:
   - System Agent
2. **Application level**, which consists of two design patterns:
   - Single Application Agent
   - Multi-Application Agent
3. **Function level**, which consists of two design patterns:
   - Single Function Agent
   - Multi-Function Agent
4. **Control Algorithm level**, which consists of four design patterns:
   - Safe-Guarded Agent
   - Trajectory Generator Agent
   - Feedback Controller Agent
   - Feedforward Controller Agent

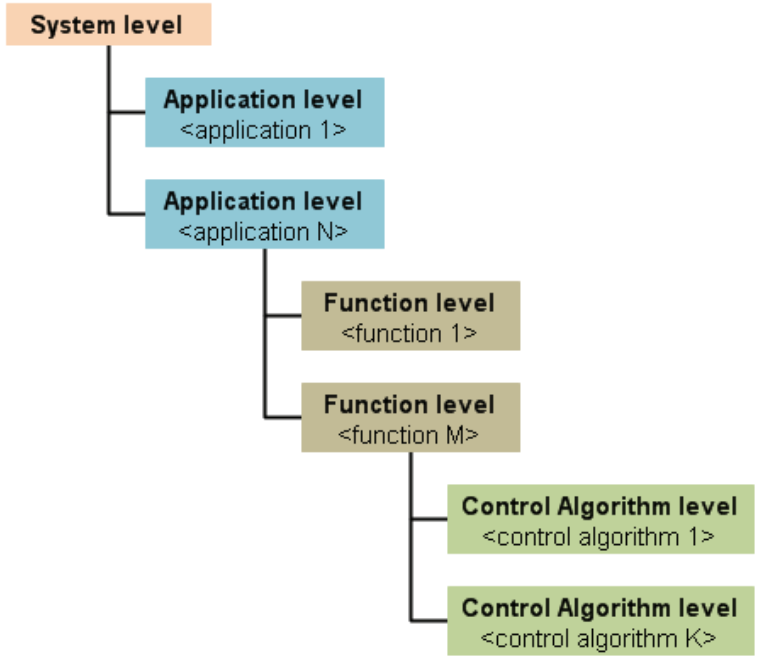


Figure 4.22 Hierarchical four-level structure of control system design patterns

Amongst nine control system design patterns, the Safe-Guarded Agent, System Agent, Single Application Agent and Single Function Agent are considered as four main design patterns, in which:

- The *Safe-Guarded Agent* is the core design pattern; it aims at providing *a generic and flexible safe-guarded control solution for mechatronic systems*. However, the Safe-Guarded Agent should not be used separately; it should be incorporated with other design patterns being the Multi-Function Agent or Multi-Application Agent to form a complete safe-guarded control solution, i.e. Single Application Agent or System Agent, respectively.
- The *System Agent, Single Application Agent and Single Function Agent* are used as *a starting point of safe-guarded MACS designs* for various types of mechatronic systems varying from simple to complex. Practically, which one of three main design patterns is applied depends on desired specifications, particular requirements, and level of complexity of each safe-guarded control problem. The Single Function Agent design pattern is generally used for simple mechatronic systems without any safe-guarded control requirements. The Single Application Agent design pattern is applied for simple mechatronic systems with safe-guarded control required at Local influence sphere. The System Agent design pattern is

intended for complex mechatronic systems with safe-guarded control required at both Local and Global influence sphere.

To support the applicability of control system design patterns, we put forward a design procedure that includes three steps:

- *Step 1:* decompose the overall safe-guarded control problem into a hierarchical structure of elementary and compound control problems, together with their interdependencies. For each control problem, a well-defined specification of the (safe-guarded) control objective is given.
- *Step 2:* compare the hierarchical structure of elementary and compound control problems with the control system design patterns and then select the best appropriate design pattern to apply. As discussed, three design patterns (Single Function Agent, Single Application Agent, and System Agent) are generally applied. However, depending on specific applications, the Multi-Function Agent or Multi-Application Agent design pattern may be used.
- *Step 3:* follow the instructions of the selected design pattern.

To make the control system design patterns easily readable and understandable, they are documented according to the pattern schema that consists of the following parts in which some are *required* properties and the others are *optional* properties.

- The *name* (required) to identify the design pattern and to present its goal briefly.
- The *problem* (required) describes problems the design pattern is trying to solve.
- The *context* (required) gives the context for which the design pattern is designed.
- The *solution* (required) presents a detailed description of the design pattern that consists of the structure, specifications and behavior of the design pattern. Some directions for use can be given.
- The *applicability* (optional) gives a recommendation of the situations where the design pattern can be applied.
- The *parent pattern (optional)* shows name of the design pattern at higher hierarchical level that holds this design pattern as a subcomponent.
- The *child patterns (optional)* shows names of design patterns at lower hierarchical level which are subcomponents of this design pattern.
- The *related references* (optional) provides some useful documents related to the design pattern such that users can refer to understand it more clearly.

To make a clear overview of nine control system design patterns and the relation between them, an overall hierarchical structure of a safe-guarded MACS is given in figure 4.23.

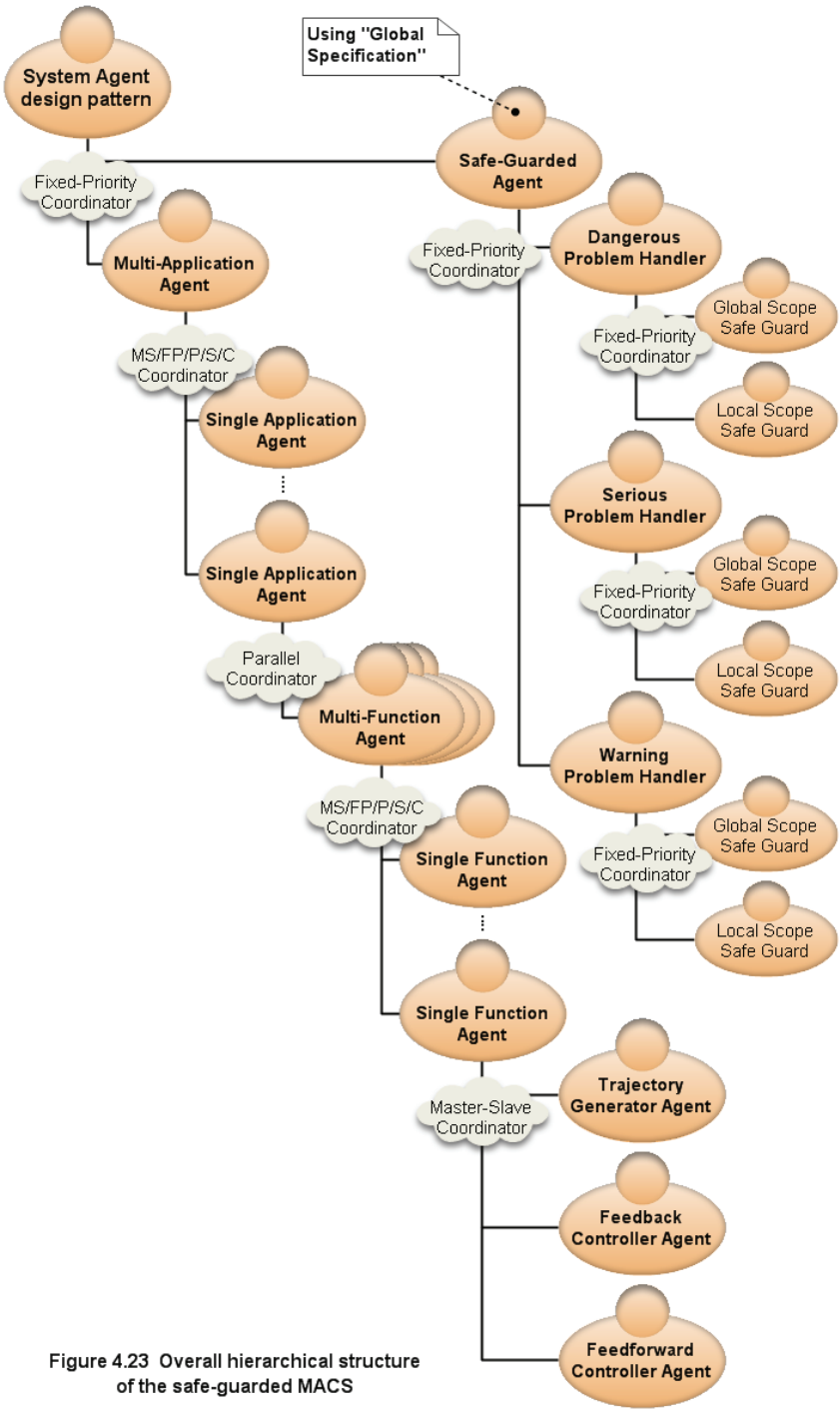


Figure 4.23 Overall hierarchical structure of the safe-guarded MACS

4.6 Concluding Remarks

This chapter has presented *the pattern-based safe-guarded MACS design method* which was developed by using a combination of two aspects: (i) the OROMACS framework, and (ii) the pattern-based design method. The polymorphism approach, which has been developed in chapter 3, was applied to formulate the control system design patterns. Through the analysis and design process of the safe-guarded MACS for the DemoLin and TriPod setup, *two reusable generalized safe-guarded control solutions, one for simple mechatronic systems and one for complex mechatronic systems*, have been realized. The design method and design patterns have answered well the two proposed research questions because they provide three following advantages for designing the safe-guarded MACS for mechatronic systems:

1. It enables to *quickly generate the hierarchically structured safe-guarded MACS* for mechatronic systems with various levels of complexity.
2. It supports *reusability at three levels*:
   - reuse coding parts containing control algorithms of a controller-agent into another controller-agent.
   - reuse controller-agents of a safe-guarded MACS design into another safe-guarded MACS design.
   - reuse the safe-guarded MACS design results of simple mechatronic systems (e.g. the DemoLin setup) into the safe-guarded MACS designs for more complex mechatronic systems (e.g. the TriPod setup).
3. It makes the design and programming of real-time safe-guarded MACS become *a matter of configuration and composition of the whole design*. This is done through the application of proper design patterns and selection of suitable specifications for controller-agents to quickly build up a complete safe-guarded MACS. As a result, the short time-to-market objective with respect to the control system development can be obtained.

Hereafter, we discuss some important aspects:

1. The Safe-Guarded Agent design pattern that we introduced in the paper (Dao et al., 2010) has *a drawback*: because the Global Safe-Guarded Agent always has a higher priority level than the Local Safe-Guarded Agent, the Global Problems Handlers have right to take over the active authority from the Local Problems Handlers. For example, the Global Warning Problems Handler can take over the active authority from the Local Dangerous Problems Handler which maybe is handling a critical fault. We have seen this situation as “a bad design”; thus studying to bring out a better solution, i.e. the Safe-Guarded Agent design pattern presented in this chapter. We emphasize here that *the new design pattern solves this drawback*. The overall hierarchical structure of a safe-guarded MACS, given in figure 4.23, clearly shows our new solution. Specifically, the priority of six

different Problem Handlers are ordered based on two aspects: (i) the criticality levels of faults, i.e. from Dangerous to Serious and finally to Warning; and (ii) the influence spheres of faults, i.e. from Global Scope to Local Scope. Hence, there is no irrational propagation between Problem Handlers.

2. The control system design patterns that we have developed provide *a general principle for designers to develop safe-guarded control systems*. As the design patterns are highly generic, they can be applied to build not only safe-guarded MACS that use the decentralized coordination architecture (i.e. the subject of this thesis), but also safe-guarded control systems that use the supervisory coordination architecture like the solution using Matlab-Simulink and StateFlow toolbox. In a general view, we expect that these design patterns will be a useful reference for developing such traditional safe-guarded control systems which are either a controller-agent-based solution or not.

3. To demonstrate benefits of the pattern-based safe-guarded MACS design method and applicability of the control system design patterns, the DemoLin and TriPod setup are used as two case studies. However, the setups in an academic environment are rather simple; therefore, they cannot fully highlight the design method and design patterns. Hence, further tests in an industrial environment are needed to fully demonstrate the usefulness of the proposed approach.

Chapter 5

Discussion

5.1 Review and Conclusions

Two main contributions of this thesis are:

1. Through merging of the MACIF and the OROCOS framework, the OROMACS framework has been formed that supports the development of multi-threaded Multi-Agent Control Systems (MACS) with deterministic real-time control behavior, thread-safe real-time inter-process communication (IPC) mechanism, and the capability of handling events. It is a good basic for solving complex control problems.
2. The pattern-based safe-guarded MACS design method solves the trade-off between the desire to obtain a real-time safe-guarded MACS with good control performances and a short development time. It also supports the reusability of MACS design results from previous projects into new projects.

Hereafter, these contributions are presented in detail through reviewing chapter 3 and 4. The design decisions, discussion and conclusions will be given while reviewing. Note that, conclusions are placed in frames to highlight their importance.

5.1.1 The OROMACS framework

This thesis aimed at developing an implementation framework for solving complex control problems in mechatronic systems. The MACIF (Van Breemen, 2001) was selected as the starting point because of its capability to produce *the hierarchically structured MACS with an open architecture*. However, as discussed in section 1.2.2, the MACIF still has some shortcomings that give room for improvement; it lacks four important features: the deterministic real-time multi-threaded control behavior, a thread-safe and real-time inter-process communication mechanism, the capability of handling events, and an efficient design support toolchain.

> An improvement scheme has been realized: developing a new implementation framework for MACS that inherits and improves the advantages of the MACIF and simultaneously provides the missing features for the MACIF.

Evaluation of MACS development approaches

In section 3.3, four possible approaches, which can be applied to develop real-time MACS using concepts and operation mechanisms of the MACIF, were studied and evaluated. The purpose of this evaluation were to get a broad view about the MACS development approaches, to point out the best approach and to give us ideas to improve the MACIF. These approaches are: (i) use a general programming language, (ii) use an agent-oriented programming language, (iii) use the Matlab-Simulink software, Stateflow toolbox and Real-Time Workshop, and (iv) use the OROCOS framework. The evaluation was based on three criteria mentioned in section 3.3.1.

This evaluation pointed out:

- The solution using either a general programming language (section 3.3.2) or an agent-oriented programming language (section 3.3.3) cannot directly specify the concepts and operation mechanisms of the MACIF. Additionally, the deterministic real-time behavior provided by these solutions is not ready-to-use.
- The solution using Matlab-Simulink software environment, StateFlow toolbox and Real-Time Workshop (section 3.3.4) is a candidate to develop multi-controller systems operating based on the supervisory/centralized coordination architecture. The deterministic real-time behavior provided by this solution is acceptable. However, this centralized coordination architecture results in control systems (i.e. MACS) with weak openness, reduces the scalability and limits the reusability in the design process, and produces over-complicated supervisors. This solution can specify the concepts (e.g. controller-agent, controller-agency, etc.) of the MACIF, but it cannot directly specify the operation mechanisms and decentralized coordination architecture of the MACIF. Moreover, coordination mechanisms

cannot be generalized to be reusable between applications. A possible solution for solving these shortcomings is to modify the Stateflow toolbox. However, Matlab-Simulink and Stateflow toolbox are commercial products so that they are not open to accept any modification.

- The solution using the OROCOS framework (section 3.3.5) brings the best features of both frameworks together. The MACIF enables to create hierarchically structured control systems, which are not possible to obtain in the OROCOS framework at the moment because the construction of a composite component is currently not supported. Whereas, the OROCOS framework: (i) supports a generic feedback control architecture, (ii) can be easily combined with several hard real-time targets or platforms to develop multi-threaded control systems with deterministic real-time behavior and thread safety, (iii) is computer platform and application independent, and (iv) is a free software project. These advantages of the OROCOS framework are an ideal complement for the MACIF. In addition, the resemblance between the elementary controller-agent of the MACIF and the TaskContext component of the OROCOS framework opens the possibility to deploy the concepts, operation mechanisms and decentralized coordination architecture of the MACIF into OROCOS.

The solution using the OROCOS framework was selected to develop a new implementation framework for MACS.

Mapping the elementary controller-agent into the TaskContext component

To get a clear understanding of the resemblance between an elementary controller-agent and a TaskContext component, their operation mechanisms were studied (see section 3.4.1 and section 3.4.2). By comparing the switching behavior of an elementary controller-agent (figure 3.4) and the extended TaskContext state transition diagram (figure 3.6), the resemblance is shown in the form of some functional equivalences with respect to the operating states of the elementary controller-agent and the TaskContext (section 3.4.3). Hence, a mapping between those was made. The main issue was to decide which state of the TaskContext should be used for the Active and Inactive state of the elementary controller-agent. Because the Running state of the TaskContext is the only one in which a TaskContext is running and can update its state variables by executing the updateHook() function, we decided to use this Running state for the Active state of the elementary controller-agent. After this option has been selected, it opens three possible choices for implementing the Inactive state of the elementary controller-agent (section 3.5.2): (i) use the Stopped state of the TaskContext, (ii) use the Active state of the TaskContext, (iii) use another substate of the Running state of the TaskContext.

In comparison with the other two solutions, the solution using two substates of the Running state of the TaskContext is the only one that provides the elementary controller-agent with the update() function being available in both the Active and Inactive state (figure 3.8).

Moreover, it does not require much modification with respect to the extended TaskContext state transition diagram as the other two solutions do.

> The solution using the Running state of the TaskContext was selected for mapping. This combination has resulted in the so-called TaskContext-based elementary controller-agent (section 3.5.3) that is the basic primitive of the new implementation framework, i.e. **ORO**COS-based Implementation Framework for **MACS** (OROMACS framework).

Composite controller-agent

In the OROMACS framework, *both the elementary and composite controller-agents have been designed to have the same interface* (section 3.6). This common interface is made up of input and output ports, operation request and acknowledge signal(s)/message(s). Seen from the outside a composite controller-agent behaves in the same way as an elementary controller-agent. In other words, a group of coordinated controller-agents can be dealt with as if it were an individual controller-agent. The only difference between the elementary and composite controller-agent is the internal architecture. That is, in case of the composite controller-agent, it has been extended with three additional functions to build up its interface and operation mechanism involving the subcontroller-agents and coordinator:

- A function to combine the individual operation request signals of all subcontroller-agents into an operation request signal that represents the group's operation intention. This group's operation request signal is then sent to a higher level coordinator.
- A function to decide which subcontroller-agent(s) is/are to be Operational.
- A function to combine outputs of subcontroller-agents.

The operation of a subcontroller-agent in a composite controller-agent depends on four things: (i) the acknowledge signal received from a higher level coordinator indicating that this group (i.e. the composite controller-agent) may become Operational; (ii) the operation self-intention of this subcontroller-agent; (iii) operation intentions of other subcontroller-agents in the composite controller-agent; and (iv) the coordination mechanism used in the group. Note that, if a negative acknowledge signal is received from a higher level coordinator, then none of the subcontroller-agents in the group may become Operational. If the received acknowledge signal is positive, then the normal coordination procedure will be followed, with a condition that at least one subcontroller-agent should be activated.

> This work makes the kind of composite components, which is currently not supported in the OROCOS framework, available in the OROMACS framework in terms of composite controller-agents.

Type and Specifications

The port-based polymorphic modeling approach (De Vries, 1994) has been brought to the OROMACS framework in such a way that a controller-agent is modularized into two parts: (i) *a Type*, defining its interface, and (ii) *a Specification*, defining the implementation of this interface (section 3.7.2). For the same Type, different Specifications can be implemented. In particular, we have considered the elementary controller-agents having an Elementary Controller-Agent Specification (called *Elementary Specification* in short) and the composite controller-agents having a Composite Controller-Agent Specification (called *Composite Specification* in short). As a result, a controller-agent with a particular Type can be implemented in the form of different Elementary and/or Composite Specifications.

Type, Elementary Specification and Composite Specification are described as follows:

- *A Type* of a controller-agent is defined through specifying its interface, i.e. input and output ports.
- *An Elementary Specification of a Type*, i.e. an implementation of an elementary controller-agent, is implemented by specifying name, parameters, instance variables, and user 'Hook' functions (i.e. internal behavior of this elementary controller-agent).
- *A Composite Specification of a Type*, i.e. an implementation of a composite controller-agent, is implemented by specifying name, parameters, list of subcontroller-agents, coordination mechanism, and inner connections of this composite controller-agent.

Three examples of Type definitions and implementations of the Elementary and Composite Specification were given in table 3.2, table 3.3, and table 3.4. Because the interface of a controller-agent is defined just in terms of input and output ports, the controller-agent becomes *a "black-box"*. That is, from the outside you only see its interface, i.e. ports. The designers should understand the essential concept of the controller-agent but they do not need to know its content in detail. This kind of interface reduces the coupling between controller-agents and provides extra flexibility with respect to the MACS development.

> This *"one Type with multiple Specifications"* approach makes the controller-agent and MACS polymorphic, i.e. we have obtained polymorphic controller-agents, polymorphic MACS. This property, called *polymorphism*, opens the possibility to create libraries of structures for which the detailed implementation is unspecified.

Roles of polymorphism

By using polymorphism in the specification and realization of MACS, any change in the design of a MACS requires less effort and time to be spent on the modification (section

3.9.1). For example, we planed to design a simple control system for an electro-mechanical motion system. Starting with the Type "Operation Controller Agent" (figure 3.17), a realization was made by specifying five particular specifications (figure 3.18). As a result, we obtained a control system in the form of an Operation Controller Agent (figure 3.20). However, because the control system required an extra safety layer, this Operation Controller Agent had to be improved. But, instead of making a lot of modifications, the only thing that had to be done is to select or specify appropriate specifications to obtain this required MACS design. Specifically, a realization of the Type "Safe-Guarded MACS" (figure 3.23) was made by reusing the realization of the Type "Operation Controller Agent" presented above, together with specifying two other specifications. As a result, we obtained a new control system in the form of a Safe-Guarded MACS (figure 3.24).

This design method is possible because of polymorphism, i.e. control algorithms and control system configurations have been realized in the form of elementary specifications and composite specifications, respectively. These specifications are complete in the design phase; thus being ready to be selected or specified in the realization phase. Based on this approach, a set of elementary specifications can constitute a library of control algorithms; and a set of composite specifications can form a library of control system configurations.

- With a sufficient library of multiple specifications, the design and programming of a control system (i.e. a MACS) becomes *a matter of configuration and composition of controller-agents*. It means that polymorphism provides users configuration options.
- Polymorphism enhances the open architecture and hierarchical structure of MACS.

Roles of coordination and configuration

The designers can decide or schedule in advance the operationality of a MACS by using different coordination principles (section 3.9.2). It is based on the role of coordinators in coordinating multiple output ports of controller-agents. The schedule is made at the design time, but it will have influence on the MACS operationality during run-time. In general, the selection of coordination should be appropriate to the specifications of the overall control problem and dependent on the type of coupling between partial control problems. Typically, the designer's experience might have an important role in selecting coordinators.

The OROMACS framework that we have developed, cooperates well with the OROCOS framework to provide a good development environment for real-time multi-threaded MACS. Specifically, the OROCOS framework provides a communication and computation mechanism for MACS; whereas, the OROMACS framework supports a configuration and coordination mechanism for MACS. As a result, *a MACS architecture involving four layers (Communication, Computation, Coordination, and Configuration)* has been created that is, if seen from the top, the Configuration is the highest layer, the next ones in turn are

Coordination, Computation, and Communication layer. This architecture is somewhat similar to the specification of a distributed system in the form of four layers that was presented by Radestock and Eisenbach (1996). However, in this four-layer-based distributed system architecture, they considered Coordination being the highest layer, which makes it different to our MACS architecture. The reason might be that, the role of Configuration in two architectures is not identical. This is explained as follows:

- Radestock and Eisenbach (1996) considered using configuration and coordination at the level of a specific design so that its configuration is normally fixed. The coordination is concerned with the interaction of the various components in the configuration; hence it decides how the configuration operates.
- In our architecture, MACS is hierarchically structured and now extended with polymorphism. So that, if considering a MACS design as a Type, we obtain a MACS design with multiple specifications, i.e. configuration options. When a particular configuration is selected, it decides how coordination in the hierarchy is. However, when the MACS runs the coordination decides how this particular configuration works.

- Designers can *pre-schedule the operationality of a MACS* by using different coordination mechanisms, i.e. they can decide beforehand a desired control strategy by selecting suitable coordinators.
- Polymorphism is *an engine* for the configuration.

OROMACS TaskContext and OROMACS Root-Agent

By constructing an OROMACS Root-Agent (i.e. a whole MACS design) residing in an OROMACS TaskContext (figure 3.19), *we have reduced the coupling between the two frameworks*. The benefit is that change of the OROCOS framework will not require much modification with respect to the designed MACS. The only thing that probably needs to be adapted is the OROMACS TaskContext and its interfaces with the OROMACS Root-Agent (section 3.8).

5.1.2 The pattern-based safe-guarded MACS design method

This design method has been developed to solve the trade-off between the desire to achieve a real-time safe-guarded MACS having good performances and a short development time; and to support the reusability of the real-time safe-guarded MACS design results from previous projects into new projects.

A generalized safe-guarded control solution for simple mechatronic systems

We started with the design of a safe-guarded MACS for the DemoLin setup (section 4.3.1), a simple mechatronic system, to meet three particular requirements (multi-operation modes, good control performances, safe-guarded control equiped with capabilities such as error detection, error handling, graceful degradation, and error recovery). We applied *a design procedure including four control system design patterns* (section 4.3.2):

- The Single Application Agent design pattern to initially generate the hierarchically structured safe-guarded MACS (figure 4.2).
- The Safe-Guarded Agent design pattern with the "Local Specification" chosen to generate the hierarchical structure for the Local Safe-Guarded Agent (figure 4.3).
- The Multi-Function Agent design pattern to generate the hierarchical structure for the Multi-Operation Mode Agent which consists of a Startup Mode Agent, a Normal Operation Mode Agent, and a Shutdown Mode Agent (figure 4.2).
- The Single Function Agent design pattern to generate the hierarchical structure for three mentioned operation modes. As a result, these operation modes have the same structure depicted in figure 4.5.

In the Local Safe-Guarded Agent (figure 4.3), the Local Scope Safe Guard (figure 4.4) directly handles faults/errors. It consists of two subcontroller-agents: (i) the Error Detection Agent is the *Master-Controller Agent* with the mission to detect faults and to classify them into criticality levels; and (ii) the Single Safe-Guarded Activity Agent is the *Slave-Controller Agent* with the mission to deal with graceful degradation and error recovery issues. Whenever a certain fault occurs the Error Detection Agent will "wake up" the Single Safe-Guarded Activity Agent to solve the problem, i.e. *the Error Detection Agent has a special role of a decision maker with regard to safe-guarded activity*.

> Based on this safe-guarded MACS design procedure for the DemoLin setup, we have formulated *a generalized safe-guarded control solution for simple mechatronic systems* (section 4.3.4).

Applying polymorphism to design patterns

Polymorphism, which has been developed in chapter 3, was applied in forming these design patterns. For example, we made the Local Scope Safe Guard polymorphic. Hence, it can hold multiple specifications or realizations. And then, three specifications of the Local Scope Safe Guard were appropriately selected for three different Problem Handlers of the Local Safe-Guarded Agent (figure 4.3). Particularly, we rely on the special role of the Error Detection Agent to formulate polymorphism of the Local Scope Safe Guard. We made the Error Detection Agent as a Type with three different specifications, being "Local Dangerous Problems", "Local Serious Problems", and "Local Warning Problems" which

deal with three criticality levels of faults, i.e. dangerous, serious, and warning, respectively (figure 4.4). Polymorphism was also applied for the Single Safe-Guarded Activity Agent such that the graceful degradation and error recovery issues can be specified flexibly through a plentiful set of multiple specifications. This example has proved the usefulness of polymorphism in design patterns. Because of that, all design patterns that we have developed use polymorphism.

> Based on the design patterns we have realized, the design and programming of the safe-guarded control part of the MACS, i.e. the Local Safe-Guarded Agent (figure 4.3), for the DemoLin setup or other simple mechatronic systems become *a matter of configurations* with respect to the Local Scope Safe Guard of three different Problem Handlers. Each configuration is done by means of selecting suitable specifications for the Error Detection Agent and Single Safe-Guarded Activity Agent.

A generalized safe-guarded control solution for complex mechatronic systems

We started with the design of a safe-guarded MACS for the TriPod setup (section 4.4.1), a more complex mechatronic system in comparison with the DemoLin setup. In this case, the particular requirements in designing a safe-guarded control system for the TriPod setup are almost the same as the ones for the DemoLin setup. The reason is that the safe-guarded control problem of each axis of the TriPod setup can be considered the same as the one of the DemoLin setup. However, in this case there was a new issue that should be taken into account: the safe-guarded control for multi-axis operations that is related to three axes of the TriPod setup. Hence, we studied the safe-guarded control for the TriPod setup in two aspects: *the local safe-guarded control for each individual axis and the global safe-guarded control for multiple axes*.

We applied a design procedure (section 4.4.2), that is: *apply three control system design patterns and reuse the safe-guarded MACS design results of the DemoLin setup into the design for the TriPod setup*.

- The System Agent design pattern to initially generate the hierarchically structured safe-guarded MACS (figure 4.13).
- The Safe-Guarded Agent design pattern with the “Global Specification” chosen to generate the hierarchical structure for the Global Safe-Guarded Agent (figure 4.14).
- The Multi-Application Agent design pattern to generate the hierarchical structure for the Multi-Axis Controller Agent which consists of a AxisZ1 Controller Agent, a AxisZ2 Controller Agent, and a AxisZ3 Controller Agent (figure 4.13).
- Finally, the safe-guarded MACS design results of the DemoLin setup was partly reused into the design for the TriPod setup. This reusability is discussed hereafter.

Based on the safe-guarded MACS design procedure for the TriPod setup, we have formulated *a generalized safe-guarded control solution for complex mechatronic systems* (section 4.4.4).

Reusability of the safe-guarded MACS design results of the DemoLin setup into the design for the TriPod setup

First, the "Operation Control part" of the safe-guarded MACS design for the DemoLin setup was reused into each axis of the TriPod setup (see figure 4.2 and figure 4.13). Hence, AxisZ1 Controller Agent, AxisZ2 Controller Agent, and AxisZ3 Controller Agent of the TriPod setup could use this Operation Control design with the same hierarchical structure as the DemoLin setup. *The only thing that remains to be done is to modify application-specific settings* (e.g. trajectory, controller parameters, coordinators, etc.).

Second, the design and programming of the safe-guarded control part of the MACS, i.e. the Global Safe-Guarded Agent (figure 4.14) for the TriPod setup, involve configurations with respect to the Local Scope Safe Guard and Global Scope Safe Guard of three different Problem Handlers, in which:

- A configuration of the Local Scope Safe Guard (figure 4.4) is done by means of selecting specifications for the Error Detection Agent (i.e. application-specific settings involving error detection and error handling) and for the Single Safe-Guarded Activity Agent (i.e. application-specific settings involving graceful degradation and error recovery).
- A configuration of the Global Scope Safe Guard (figure 4.15) is done through selecting specifications for the Error Detection Agent and multiple Single Safe-Guarded Activity Agents of the Multi-Safe-Guarded Activity Agent.

In particular, *the Local Scope Safe Guard of three Problem Handlers that we realized for the DemoLin setup was reused for the TriPod setup*. The only thing that needs to be done is to select specifications as mentioned, which are suitable for the TriPod setup.

In figure 4.16, we present an overall hierarchical structure of the safe-guarded MACS for the TriPod setup. The list of 18 common fault sources of mechatronic systems (table 4.1) was mapped into Problem Handlers of the Safe-Guarded Agent. This map is based on the categorization of fault sources in table 4.2.

We reused the safe-guarded MACS design results of the DemoLin setup into the design for the TriPod setup. This reusability was proved through reusing two parts of the MACS design: *the operation control and the safe-guarded control.*

The propagation between Problem Handlers of the Safe-Guarded Agent

A remark is first given: because of the Global and Local Scope, the Safe-Guarded Agent actually has six different Problem Handlers instead of three. Specifically, six Problem Handlers are represented by six specifications of the Error Detection Agent. We have organized these Problem Handlers in the form of: (i) *the local measure* involving the Local Warning Problem Handler, Local Serious Problem Handler and Local Dangerous Problem Handler; and (ii) *the global measure* involving the Global Warning Problem Handler, Global Serious Problem Handler and Global Dangerous Problem Handler. It is not required that all six Problem Handlers participate in the Safe-Guarded Agent structure as they are optionally configurable. However, we recommend users to design the Safe-Guarded Agent with all Problem Handlers because it provides the safe-guarded MACS with a flexible capability to handle a variety of faults with different criticality levels through the propagation between these Problem Handlers.

Propagations between six Problem Handlers depend on two things: (i) the propagations of influence spheres of faults, i.e. from Local to Global Scope; and (ii) the propagations of criticality levels of faults, i.e. from Warning to Serious, from Serious to Dangerous, and from Warning to Dangerous. It means that, a maximum propagation can run from the Local Warning Problem Handler (i.e. thought as the lowest level) to the Global Dangerous Problem Handler (i.e. considered as the highest level).

Depending on specific requirements of each application, the user can design all safe-guarded activities (such as error detection, error handling, graceful degradation, and error recovery) at the local measure or distribute them at the local and global measure. However, using both local and global measure helps to handle faults in a structured way: the safe-guarded activities are hierarchically organized with respect to the hierarchy of faults (i.e. the influence sphere and criticality level). When a certain fault occurs, if the local measure can handle this fault then the global measure does not need to be activated. On the contrary, if the local measure cannot handle the problem then the global measure will be activated.

- The propagation between these Problem Handlers can be thought of as *the capabilities of flexible error handling and graceful degradation* of the Safe-Guarded Agent.
- Compared with a control system that its safe-guarded part is placed in a single component or module, the Safe-Guarded Agent with six different Problem Handlers can *reduce the complexity* in designing and implementing the safe-guarded control system.
- The Safe-Guarded Agent is not fixed or closed; it is *flexible and open* for users to decide their safe-guarded control strategy.

The pattern-based safe-guarded MACS design method

This design method together with control system design patterns have provided a general principle to develop safe-guarded control systems. As these design patterns are generic, they might be applied for developing such traditional safe-guarded control systems which are either a controller-agent-based solution or not. We expect that these design patterns will be a useful reference for the designers.

The pattern-based safe-guarded MACS design method provides three advantages in designing the safe-guarded MACS for mechatronic systems:

1. It enables to *quickly generate the hierarchically structured safe-guarded MACS* for mechatronic systems with various levels of complexity.
2. It supports *reusability at three levels*:
   - reuse coding parts containing control algorithms of a controller-agent into another controller-agent.
   - reuse controller-agents of a safe-guarded MACS design into another safe-guarded MACS design.
   - reuse the safe-guarded MACS design results of simple mechatronic systems into the designs for more complex mechatronic systems.
3. It makes the design and programming of real-time safe-guarded MACS become *a matter of configuration and composition of the whole design*. This is done through the application of proper design patterns and selection of suitable specifications for controller-agents to quickly build up a complete safe-guarded MACS. As a result, the short time-to-market objective with regard to the control system development can be obtained.

5.2 Suggestions for Future Work

At the end of chapter 3, the MACS architecture involving four layers (Communication, Computation, Coordination, and Configuration) has been brought to discuss. However, this is just a first step. Further research should be performed to get a clear understanding about the roles of layers and the relation between layers, especially between the Coordination and Configuration.

Three software tools have been developed to support the OROMACS framework: OROCOS - 20sim Cosimulation (Bozlak, 2009), OROMACS Browser (Tadele, 2009), and

MACS Editor & Code Generation (Bozlak, 2010). However, these tools have not yet completed that give room for improvement. The idea we recommend is: first, improving the functionality of each tool, and then integrating these tools into a design support toolchain that can facilitate the development of MACS into practical applications in multiple stages, from the anlysis, design and (co)simulation in a computer to the implementation and realization with a real mechatronic system.

At present, we have just used the data flow port primitive in defining a Type. Other primitives of the OROCOS framework such as events, methods, and commands should be considered whether they are suitable for making a Type.

In chapter 4, the DemoLin and TriPod setup were used as two case studies. However, because the setups in an academic environment are rather simple, they cannot fully highlight the design method and design patterns. Hence, further tests in an industrial environment are needed to fully demonstrate the usefulness of the approach that we have developed.

In the future, mechatronic and manufacturing systems will be more and more complex and heterogeneous. Hence, besides the control system design patterns we have developed, further research to develop new design patterns is recommended.

Appendix

A.1 The DemoLin setup

A schematic diagram and dimensions of the DemoLin setup are depicted in figure A.1.

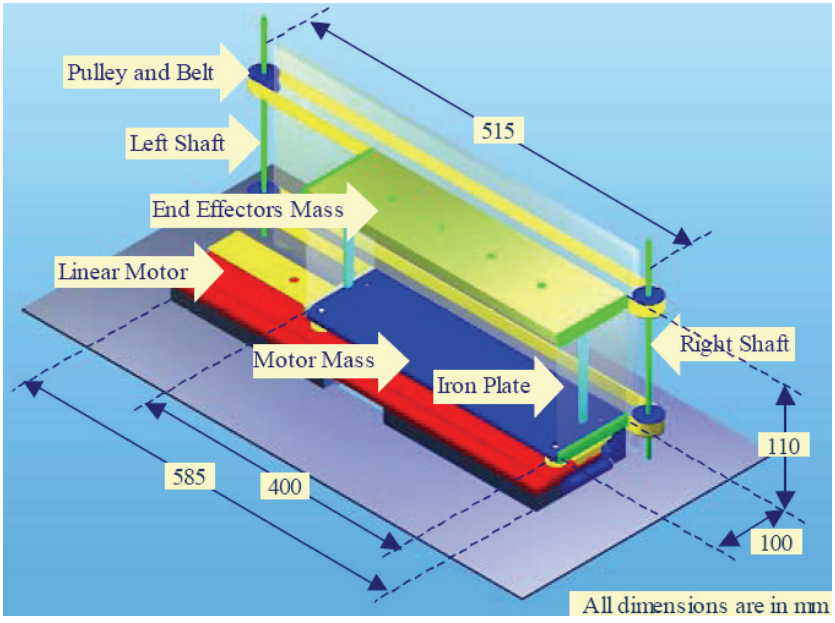


Figure A.1 Schematic diagram of the DemoLin setup (Bajracharya, 2003)

In the following, description of components of the DemoLin setup is given.

1. The electrical and electronic components:

Linear motor:

- Type: ironless synchronous permanent magnet.

- Model: Yaskawa SGLGW60A365A.
- Thrust constant: $k_m = 63$ [N/A].

Servo amplifier:

- Type: Yaskawa Servo Pack SGDH 08AE.
- Gain factor: $k_a = 1.1$ [A/V].

Encoders:

- Linear encoder for motor position: Heidenhain LIDA181.
- Rotational encoder for end-effector position: Heidenhain ROD466.

2. The mechanical parameters:

Elements	Value
End-effector mass (m_1)	5.5 [kg]
Motor mass (m_2)	6.08 [kg]
Stiffness of iron plates (c)	3000 [N/m]
Viscous friction coefficient of the motor mass (μ_v)	2.0 [s/m]

Table A.1 Mechanical parameters of the DemoLin setup

3. Modeling of the DemoLin setup:

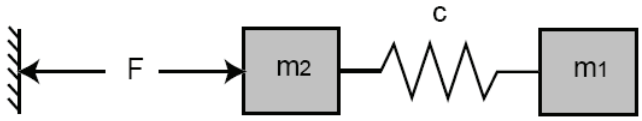


Figure A.2 Iconic diagram of the DemoLin setup

An iconic diagram of the DemoLin setup is presented in figure A.2. Parameters of the load mass (i.e. end-effector mass), motor mass, stiffness of iron plates and viscous friction coefficient of the motor mass are given in table A.1. Because the dominant stiffness is located between motor and end-effector, it forms a Flexible Mechanism (Coelingh, 2000). The feedback signals (position and velocity) of the linear motor are used to design the controller so that the plant model of the DemoLin setup is defined as a flexible mechanism of type AR (Anti Resonance - Resonance) with a transfer function:

$$P_{AR}(s) = \frac{1}{ms^2} \cdot \frac{s^2 + \omega_{ar}^2}{s^2 + \omega_r^2} \cdot \frac{\omega_r^2}{\omega_{ar}^2}, \qquad \omega_{ar} < \omega_r$$

where m is total mass to be displaced; ω_{ar} is anti-resonance frequency; and ω_r is resonance frequency. These parameters are calculated by using data in table A.1. Note that, in the safe-guarded MACS designed for the DemoLin setup (and also for the TriPod setup), PID controllers are used as the basic primitives (see figure 3.17). These PID controllers are synthesized based on the design method presented in Coelingh (2000).

A.2 The TriPod setup

1. Specifications of the TriPod setup:

- Safe work area: a cylinder with radius 200 [mm] and height 250 [mm]
- Max payload: 5 [kg]
- Max speed: 1 [m/s]
- Max acceleration: 30 [m/s^2]
- Max stroke of linear motors: 520 [mm]

2. The electrical and electronic components:

Three motors:

- Type: ironless synchronous permanent magnet (Tecnotion).
- Thrust constant: $k_m = 39.0$ [N/A].

Three servo amplifiers:

- Type: TBL250/10 (MTS Automation).
- Gain factor: $k_a = 2.0$ [A/V].

Three encoders:

- Linear encoders (Numerik Jena)
- Resolution: 20.0 [μm].

Encoder card, DAC card and I/O card:

- Encoder card: PCL-833 (Advantech).
- DAC card: 8 channel x 14-bit (ICP DAS).
- I/O card: 16 optically isolated inputs and 16 relay outputs. The input voltage is 5-24 volt AC or DC.

These amplifiers have a current/voltage ratio of 2 and the linear motors have a force/current ratio of 39; hence the force/voltage ratio is 78.

3. The mechanical parameters of each axis:

Elements	Value
Platform mass ($m_{platform}$)	0.718 [kg]
End-effector mass (m_1) $m_1 \approx m_{platform} + m_{slide} + m_{parallel\_bars}$	1.161 [kg]
Maximum payload (m_{load})	5.0 [kg]
Motor mass (m_2)	2.0 [kg]
Mass of two parallel bars ($m_{parallel\_bars}$)	0.118 [kg]
Slide mass (m_{slide})	0.325 [kg]
Stiffness of iron plates (c)	unknown
Viscous friction coefficient of the motor mass (μ_v)	unknown

Table A.2 Mechanical parameters of the TriPod setup

4. Modeling of the TriPod setup:

In this thesis, we reused the 20-sim model of the TriPod setup, which was made by Controllab Products B.V. (2009), for testing the designed safe-guarded MACS in the cosimulation setting (see section 4.4.3). This model consists of the following parts:

- Three linear motors with their amplifiers that only operate in z-direction. These submodels primarily consist of a modulated source of force and a translator mass.
- Three legs with ball joints on both ends. These legs hold the kinematics of the manipulator and they contain spring and damping effects.
- A platform has a mass and can move with three degrees of freedom (x,y,z).

However, for designing PID controllers, a simplified model of the TriPod setup is needed. As discussed in section 4.4.1, each leg of the TriPod setup has the same plant model as the DemoLin setup. Hence, the TriPod setup is divided into three identical parts, in which each part has the same the plant model as the DemoLin setup. Hence, solving the controller design for one part will give a solution of the total control problem of the TriPod setup.

Bibliography

3APL website (2010), Link: http://www.cs.uu.nl/3apl/.

Agent Factory website (2010), Link: http://sourceforge.net/projects/agentfactory.

Alexander, C. (1979), *The Timeless Way of Building*, New York: Oxford Univ. Press.

ANSI/RIA R15.06-1999 (R2009), *American National Standard for Industrial Robots and Robot Systems - Safety Requirements*, ANSI, Link: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI/RIA+R15.06-1999+(R2009).

Arimoto, S., S. Kawamura, and F. Miyazaki (1984), *Bettering Operation of Robots by Learning*, Journal of Robotic Systems, vol. 1, pp. 123-140.

Astrom, K.J. and T. Hagglund (1995), *PID Controllers: Theory, Design and Tuning*, Research Triangle Park: 2nd edition, Instrumentation, Systems and Automatic Society, NC, USA.

Astrom, K.J. and B. Wittenmark (1995), *Adaptive Control*, Addison-Wesley, Massachusetts Menlo Park, 2nd edition, ISBN: 0201558661.

Bajracharya, G. (2003), *Integrated Design and Implementation Tool for Multi-Agent Controllers [IDITmac]*, Master thesis, report no. 001CE2003, Control Laboratory, University of Twente, Enschede, The Netherlands.

Baxter, J.W. and G.S. Horn (2005), *Controlling Teams of Uninhabited Air Vehicles*, in Proc. the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp.86-95, 27-33 July 2005, The Netherlands.

Bellifemine, F., F. Bergenti, G. Caire, and A. Poggi (2005), *JADE - A Java Agent Development Framework*, chapter 5, pp. 125-147, in Bordini et al., editors, Multi-Agent Programming: Languages, Platforms and Applications, vol. 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer.

Bijl, P. (2006), *Design Support for Multi-Agent Controller Specification in 20-Sim*, Individual Design Assignment, report no. 011CE2006, Control Laboratory, University of Twente, Enschede, The Netherlands.

Bien, Z. and J.X. Xu (1998), *Iterative Learning Control: Analysis, Design, Integration and Applications*, Kluwer Academic Publishers, Boston, USA.

Bozlak, Z. (2009), *Co-simulation of an Orocos-based Controller and a 20-sim Plant*, Pre-doctoral Project, report no. 006CE2009, Control Laboratory, University of Twente, Enschede, The Netherlands.

Bozlak, Z. (2010), *Editor and Code Generation for OroMACS*, Master thesis, report no. 023CE2010, Control Laboratory, University of Twente, Enschede, The Netherlands.

Bosgra, O.H., H. Kwakernaak, and G. Meinsma (2006), *Design Methods for Control Systems*, Lecture Notes, Dutch Institute of Systems and Control (DISC), The Netherlands.

Bordini, R.H., J.F. Hübner, and R. Vieira (2005), *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pp. 3-37, in Bordini et al., editors, Multi-Agent Programming: Languages, Platforms and Applications, vol. 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer.

Breedveld, P.C. (1982), *Proposition for an Unambiguous Vector Bond Graph Notation*, J. Dynamic Systems, Measurement, and Control, vol. 104, nr. 3, pp. 267-270.

Breedveld, P.C., R.C. Rosenberg, and T. Zhou (1991), *Bibliography of Bond Graph Theory and Application*, J. of the Franklin Institute, vol. 328, nr. 5/6, pp. 1067-1109.

Burns, R.S. (2001), *Advanced Control Engineering*, 1st edition, Butterworth-Heinemann, ISBN: 0750651008.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern Oriented Software Architecture - A System of Patterns*, Wiley, New York.

Bustani, A. (2008), *Towards an Integrated Development Environment for Multi-Agent Controllers*, Individual Design Assignment, report no. 018CE2008, Control Laboratory, University of Twente, Enschede, The Netherlands.

Buur, J. (1990), *A Theoretical Approach to Mechatronics Design*, PhD thesis, Institute for Engineering Design, Technical University of Denmark, Lyngby, Denmark.

Braubach, L., A. Pokahr, and W. Lamersdorf (2005), *Jadex: A BDI Agent System Combining Middleware and Reasoning*, pp. 143-168, in R. Unland, M. Calisti, and M. Klusch (ed.), Software Agent-Based Applications, Platforms and Development Kits, Birkhäuser Basel.

Bruyninckx, H. (2001), *Open Robot Control Software: The OROCOS Project*, in Proc. of the IEEE International Conference on Robotics and Automation (May 21-26, 2001), Seoul, Korea, pp. 2523-2528.

Bruyninckx, H. (2002a), *A Free Software Framework for Advanced Robot Control*, in Proc. of the 7th ESA Workshop on Advanced Space Technologies for Robotics and Automation (Nov. 1, 2002), Noordwijk, The Netherlands.

Bruyninckx, H. (2002b), *Real-Time and Embedded Guide*, Katholieke Universiteit Leuven, Belgium. Link: http://people.mech.kuleuven.be/~bruyninc/rthowto/rtHOWTO.pdf

Bruyninckx, H., P. Soetens, and B. Koninckx (2003), *The Real-Time Motion Control Core of the Orocos Project*, in Proc. of the IEEE International Conference on Robotics and Automation (Sept. 14-19, 2003), Taipei, Taiwan, pp. 2766-2771.

Chang, J., K.Y. Lee, and R. Garduno-Ramirez (2003), *Multiagent Control System for a Fossil-Fuel Power Unit*, in Proc. IEEE Power Engineering Society General Meeting, Toronto, Canada, July 13-17, 2003.

Coelingh, H. J. (2000), *Design Support for Motion Control Systems*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-36514118.

Collins, J.W. (1989), *Experimental Evaluation of Emergency Stop Buttons mounted on Hand-held Control Pendants*, in Proc. Human Factors Annual Meeting, Human Factors Society, Santa Monica, CA, pp. 951-955.

Collier, R.W. (2002), *Agent Factory: A Framework for the Engineering of Agent-Oriented Applications*, PhD thesis, University College Dublin, Belfield, Dublin 4, Ireland.

Control Engineering Lab (2009), http://www.ce.utwente.nl/.

Controllab Products B.V. (2009), http://www.20sim.com.

Cook, P.A. (1994), *Application of Model Reference Adaptive Control to a Benchmark Problem*, Automatica, vol. 30, issue 4, April 1994, pp. 585-588.

Cuong, N.D. (2008), *Advanced Controllers for Electromechanical Motion Systems*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN: 978-90-365-2654-8.

Czogala, E. and J. Leski (2000), *Fuzzy and Neuro-Fuzzy Intelligent Systems*, Physica-Verlag, Heidelberg, Germany.

Dao, P.B., T.J.A. De Vries, and J. Van Amerongen (2010), *Safe-Guarded Agent Design Pattern for Mechatronic Systems*, 5th IFAC Symposium on Mechatronic Systems, 13-15 September 2010, Cambridge, MA, USA, pp. 345-354.

De Vries, T.J.A., P.C. Breedveld, and P. Meindertsma (1993), *Polymorphic Modelling of Engineering Systems*, Proceedings Int. Conf. on Bond Graph Modeling and Simulation, Western Simulation MultiConference, SCS, San Diego, California, U.S.A, pp. 17-22.

De Vries, T.J.A. (1994), *Conceptual Design of Controlled Electro-Mechanical Systems - A Modeling Perspective*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-9006876-7.

De Vries, T.J.A., W.J.R. Velthuis, and L.J. Idema (2001), *Application of Parsimonious Learning Feedforward Control to Mechatronic Systems*, IEE Proceedings of Control Theory Applications, vol. 148, nr. 4, pp. 318-322.

De Kruif, B.J. (2004), *Function Approximation for Learning Control: A Key Sample Based Approach*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-365-2050-9.

De Santis, A. and B. Siciliano (2008), *Safety Issues for Human-Robot Cooperation in Manufacturing Systems*, Tools and Perspectives in Virtual Manufacturing, Napoli, Italy, July 2008.

Decker, K., K. Sycara, and M. Williamson (1997), *Middle-Agents for the Internet*, in Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, Nagoya, Japan, August 1997.

Dhillon, B.S. and O.C. Anude (1993), *Robot Safety and Reliability: A Review*, Microelectronics and Reliability, vol. 33, issue 3, February 1993, pp. 413-429.

Doorenbos, R.B., O. Etzioni, and D.S. Weld (1997), *A Scalable Comparison-Shopping Agent for the World-Wide Web*, in Proceedings of the First International Conference on Autonomous Agents.

Dorf, R.C. and R.H. Bishop (2004), *Modern Control Systems*, Prentice Hall, 10th edition, ISBN: 0131457330.

Durfee, E.H. and V.R. Lesser (1989), *Negotiating Task Decomposition and Allocation Using Partial Global Planning*, Distributed Artificial Intelligence, vol. 2, M. Huhns and L. Gasser (ed.), Pitman Publishing Ltd., London, England, pp. 229-244.

Eglence, M. (2003), *Design and Realization of a Safe Control System for a Parallel Manipulator*, Master thesis, report no. 010CE2003, Control Laboratory, University of Twente, Enschede, The Netherlands.

European Machinery Directive (2006), *2006/42/EC*, May 2006.

FIPA (2003), *FIPA Methodology: Glossary of Terms*, The Foundation for Intelligent Physical Agents, rel. 1.0: 2003/11/18.

FIPA website (2010), Link: http://www.fipa.org/.

FLUX website (2010), Link: http://www.fluxagent.org/.

Flinkers, A.B. (2006), *Development of Learning Multi-Agent Controllers for Mechatronic Motion Systems*, Master thesis, report no. 032CE2006, Control Laboratory, University of Twente, Enschede, The Netherlands.

Fletcher, M., D. McFarlane, A. Lucas, J. Brusey, and J. Jarvis (2003), *The Cambridge Packing Cell: A Holonic Enterprise Demonstrator*, in Proc. the 3rd International / Central and Eastern European conference on Multi-Agent Systems, Prague, Czech Republic.

Franklin, S. and A. Graesser (1996), *Is it an agent, or just a program?*, in Proc. the Third International Workshop on Agent Theories, Architectures and Languages, pp. 193-206.

Fregene, K., D.C. Kennedy, and D.W.L. Wang (2001), *HICA: A Framework for Distributed Multiagent Control*, in Proc. Int. Conf. on Intelligent Systems and Control, Tampa, Florida, November 2001, pp. 187-192.

Fregene, K., D.C. Kennedy, R. Madhavan, L.E. Parker, and D.W.L. Wang (2005), *A Class of Intelligent Agents for Coordinated Control of Outdoor Terrain Mapping UGVs*, Engineering Applications of Artificial Intelligence, vol. 18, nr. 5, pp. 513-531.

Friedland, B. (1996), *Advanced Control System Design*, Prentice Hall International, Inc., Englewood Cliffs, New Jersey, ISBN: 0130140104.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN: 0-201-63361-2.

Graebe, S.F. (1999), *Guest editorial special section on "Robust control Benchmark - New Results"*, Eur. J. Control, vol. 5, pp. 183-184.

Harper, C. and G. Virk (2010), *Towards the Development of International Safety Standards for Human Robot Interaction*, International Journal of Social Robotics, vol. 2, nr. 3, pp. 229-234, in special issue: Towards Safety in Human Robot Interaction, ed. by G. Herrmann and C. Melhuish.

Herrmann, G. and C. Melhuish (2010), *Towards Safety in Human Robot Interaction*, International Journal of Social Robotics, vol. 2, nr. 3, pp. 217-219, in special issue: Towards Safety in Human Robot Interaction, ed. by G. Herrmann and C. Melhuish.

Heinzmann, J. and A. Zelinsky (2003), *Quantitative Safety Guarantees for Physical Human-Robot Interaction*, International Journal of Robotics Research, vol. 22, pp. 479-504.

Helander, M.G. (1988), *Ergonomics, Workplace Design*, in: R.C. Dorf (ed.), International Encyclopedia of Robotics, Wiley, New York, pp. 477-487.

Hilhorst, R.A. (1992), *Supervisory Control of Mode-Switch Processes*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-9004829-4.

Hindriks, K.V., F.S. de Boer, W. van der Hoek, and J-J.Ch. Meyer (1999), *Agent Programming in 3APL*, in Int. J. of Autonomous Agents and Multi-Agent Systems, vol. 2, nr. 4, pp. 357-401.

Howden, N., R. Rönnquist, A. Hodgson, and A. Lucas (2001), *JACK^{TM} Intelligent Agents - Summary of an Agent Infrastructure*, in Proc. of the 5th ACM International Conference on Autonomous Agents, Montreal, Canada.

HSG43 (2000), *Industrial Robot Safety: Your Guide to the Safeguarding of Industrial Robots*, Health and Safety Executive, 2nd edition, ISBN: 0-7176-1310-0, Link: http://www.springboardsafetyservices.com/HSG_43_robots.htm

Ikuta, K., H. Ishii, and M. Nokata (2003), *Safety Evaluation Method of Design and Control for Humancare Robots*, International Journal of Robotics Research, vol. 22, pp. 281-297.

IRDAC (1986), *Opinion on R&D needs in the Field of Mechatronics*, Industry R&D Advisory Committee of the Comm. of the EC, Brussels, Belgium.

Isermann, R. (1997), *Mechatronic Systems - A Challenge for Control Engineering*, in Proc. of the 1997 American control conference, June, Albuquerque, New Mexico, vol. 5, pp. 2617-2632, ISBN: 0-7803-3832-4.

ISO 10218-1 (2006), *Robots for Industrial Environment - Safety Requirements - Part 1: Robot*, ISO, 27p.

ISO 10218-2 (2008), *Robots for Industrial Environment - Safety Requirements - Part 2: Industrial Robot Systems and Integration*, ISO, 94p, draft version.

Jacobs, R.A. and M.I. Jordan, (1993), *Learning Piecewise Control Strategies in a Modular Neural Network Architecture*, IEEE Trans. Systems, Man, and Cybernetics, vol. 23, nr. 2, pp. 337-345.

JAL website (2010), Link: http://www.agent-software.com/.

Jang, T.J., C.H. Choi, and H.S. Ahn (1995), *Iterative Learning Control in Feedback Systems*, Automatica, vol. 31, nr. 2, pp. 243-245.

Jason website (2010), Link: http://jason.sf.net/.

JADE website (2010), Link: http://jade.tilab.com/.

Jadex website (2010), Link: http://jadex-agents.informatik.uni-hamburg.de

Jennings, N.R. (2001), *An Agent-Based Approach for Building Complex Software Systems*, Communications of the ACM, vol. 44, nr. 4, pp. 35-41.

Jennings, N.R., K. Sycara, and M. Wooldridge (1998), *A Roadmap of Agent Research and Development*, in Autonomous Agents and Multi-Agent Systems Journal, N.R. Jennings, K. Sycara and M. Georgeff (eds.), Kluwer Academic Publishers, vol. 1, issue 1, pp. 7-38.

Jennings, N.R. and M. Wooldridge (1998), *Applications of Intelligent Agents*, in Agent Technology: Foundations, Applications and Markets, Springer-Verlag, NY, pp. 3-28.

Jiang, B.C. and C.A. Gainer (1987), *A Cause-and-Effect Analysis of Robot Accidents*, Journal of Occupational Accidents, vol. 9, nr. 1, pp. 27-46.

Johansen, T.A. and R. Murray-Smith (1997), *The Operating Regime Approach to Nonlinear Modelling and Control*, Taylor & Francis.

Kamnik, R., D. Matko, and T. Bajd (1998), *Application of Model Reference Adaptive Control to Industrial Robot Impedance Control*, Journal of Intelligent & Robotic Systems, vol. 22, nr. 2, pp. 153-163.

Karnopp, D.C. and R.C. Rosenberg (1968), *Analysis and Simulation of Multiport Systems: The Bond Graph Approach to Physical System Dynamics*, MIT Press, Cambridge.

Kawato, M., K. Furukawa and R. Suzuki (1987), *A Hierarchical Neural-Network Model for Control and Learning of Voluntary Movement*, Biological Cybernetics, vol. 57, pp.169-185.

Kok, J.K., C.J. Warmer, and I.G. Kamphuis (2005), *PowerMatcher: Multiagent Control in the Electricity Infrastructure*, in Proc. the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 75-82, July 25-29, The Netherlands.

Kulic, D. and E.A. Croft (2005), *Real-Time Safety for Human-Robot Interaction*, 12th IEEE International Conference on Advanced Robotics, Seattle, WA.

Krulwich, B. (1996), *The BargainFinder Agent: Comparison Price Shopping on the Internet*, in Agents, Bots, and Other Internet Beasties, J. Williams, ed., Sams.Net Publishing, Macmillan, Indianapolis, Ind., pp. 257-263.

Lea, D. (1994), *Design Patterns for Avionics Control Systems*, DSSA ADAGE Project, State Univ. of New York, Oswego, Tech. Rep. ADAGE-OSW-94-01.

Lin, C.T. and C.S.G. Lee (1996), *Neural Fuzzy Systems: A Neuro-Fuzzy Synergism to Intelligent Systems*, Prentice Hall, Inc., Upper Saddle River, NJ.

Lockemann, P., J. Nimis, L. Braubach, A. Pokahr, and W. Lamersdorf (2006), *Architectural Design*, part 4, chapter 4, pp. 405-429, in Kirn et al., editors, Multiagent Engineering - Theory and Applications in Enterprises, Springer Series: International Handbooks on Information Systems, Publisher: Springer Berlin Heidelberg, ISBN 978-3-540-31406-6 (print), 978-3-540-32062-3 (online).

Luck, M. (1999), *From Definition to Deployment: What next for Agent-Based Systems?*, Knowledge Engineering Review, vol. 14, nr. 2, pp. 119-124.

Lygeros, J., D.N. Godlobe, and S. Sastry (1997), *Hybrid Controller Design for Multi-Agent Systems*, in Control Using Logic Based Switching, Springer Berlin, pp. 59-78. ISSN: 0170-8643.

Malm, T., J. Viitaniemi, J. Latokartano, S. Lind, O. Venho-Ahonen, and J. Schabel (2010), *Safety of Interactive Robotics - Learning from Accidents*, International Journal of Social Robotics, vol. 2, nr. 3, pp. 221-227, in special issue: Towards Safety in Human Robot Interaction, edited by G. Herrmann and C. Melhuish.

Matsumoto, T. and K. Kosuge (2001), *Collision Detection of Manipulator Based on Adaptive Control Law*, IEEE/ASME International Conference on Advanced Intelligent Mechatronics, pp.177-182.

Marik, V., P. Vrba, K.H. Hall, and F.P. Maturana (2005), *Rockwell Automation Agents for Manufacturing*, in Proc. the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 107-113, July 25-29, The Netherlands.

Masina, S., K.Y. Lee, and R. Garduno-Ramirez (2004), *An Architecture of Multi Agent System Applied to Fossil Power-Fuel Power Unit*, in Proc. of the IEEE Power Engineering Society General Meeting, Denver, CO, June 6-10.

Minsky, M. (1986), *The Society of Mind*, Simon & Schuster, Inc.

Miller, W.T., R.S. Sutton, and P.J. Werbos (1995), *Neural Networks for Control*, The MIT Press Cambridge, Massachustts London, England, ISBN: 0-262-13261-3.

Miyamoto, H., M. Kawato, T. Setoyama, and R. Suzuki (1988), *Feedback-Error-Learning Neural Network for Trajectory Control of a Robotic Manipulator*, Neural Networks, vol. 1, pp. 251-265.

Molin, P. and L. Ohlsson (1996), *The Points and Deviations Pattern Language of Fire Alarm Systems*, in Pattern Languages of Program Design 3, R. Martin, D. Riehle, and F. Buschmann, eds., MA: Addison-Wesley, pp. 431-445.

Monostori, L., J. Vancza, and S.R.T. Kurama (2006), *Agent-Based Systems for Manufacturing*, Annals of the CIRP, vol. 55, nr. 2, pp. 697-720.

Moore, K.L., M. Dahleh, and S.P. Bhattacharyya (1992), *Iterative Learning Control: A Survey and New Results*, Journal of Robotic Systems, vol. 9, nr. 5, pp. 563-584.

Morinaga, S. and K. Kosuge (2003), *Collision Detection System for Manipulator Based on Adaptive Impedance Control Law*, in Proc. of fhe 2003 IEEE International Conference on Robotics and Automation, pp. 1080-1085.

Narendra, K.S., J. Balakrishnan, and M.K. Ciliz (1995), *Adaptation and Learning using Multiple Models, Switching and Tuning*, IEEE Control Systems, pp. 37-51, June 1995.

Narendra, K.S. and J. Balakrishnan (1997), *Adaptive Control Using Multiple Models*, IEEE Trans. on Automatic Control, vol. 42, nr. 2, pp. 171-187.

National Instruments Corporation (2009), http://www.ni.com/labview/.

Ning, K.J. and R.Q. Yang (2006), *MAS based Embedded Control System Design Method and a Robot Development Paradigm*, Journal of Mechatronics, vol. 16, pp. 309-321.

Nise, N.S. (2004), *Control Systems Engineering*, 4th edition, John Wiley and Sons Inc., ISBN 0-471-44577-0.

Nwana, H.S. (1996), *Software Agents: An Overview*, Knowledge Engineering Review, vol. 11, nr. 3, pp. 1-40.

Orocos (2009a), the OROCOS website: http://www.orocos.org/.

Orocos (2009b), the OROCOS Component Builder's Manual, Link: http://www.orocos.org/stable/documentation/rtt/current/doc-xml/orocos-components-manual.html.

Paynter, H.M. (1961), *Analysis and Design of Engineering Systems*, MIT Press, Cambridge.

Pechoucek, M., Thompson, S., Baxter, J., Horn, G., Kok, K., Warmer, C., Kamphuis, R., Marík, V., Vrba, P., Hall, K., Maturana, F., Dorer, K., and Calisti, M. (2006), *Agents in Industry: The Best from the AAMAS 2005 Industry Track*, in IEEE Intelligent Systems, vol. 21, nr. 2, pp.86-95, ISSN: 1541-1672.

Pechoucek, M. and V. Marik (2008), *Industrial Deployment of Multi-Agent Technologies: Review and Selected Case Studies*, International Journal on Autonomous Agents and Multi-Agent Systems, ISSN 1387-2532.

Pilz Automation Technology (2007), *Safety Advice for First-time Users of Industrial Robots*, Link: http://www.machinebuilding.net/ta/t0078.htm

Radestock, M. and S. Eisenbach (1996), *Coordination in Evolving Systems*, in TreDS'96: Proceedings of the International Workshop on Trends in Distributed Systems (London, UK), pp. 162-176, Springer-Verlag.

Rahimi, M. and W. Karwowski (1990), *A Research Paradigm in Human-Robot Interaction*, International Journal of Industrial Ergonomics, vol. 5, issue 1, January 1990, pp. 59-71.

Rao, A.S. (1996), *AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language*, in Proc. of Modelling Autonomous Agents in a Multi-Agent World, number 1038 in LNAI, pp. 42-55, Springer-Verlag.

Ren, Z. and C.J. Anumba (2004), *Multi-Agent Systems in Construction - State of the Art and Prospects*, Automation in Construction, vol. 13, nr. 3, pp. 421-434.

Rezola, E.I. (2009), *Learning Multi-Agent Control with OROCOS*, Master thesis, report no. 001CE2009, Control Laboratory, University of Twente, Enschede, The Netherlands.

Rubel, B. (1995), *Patterns for Generating a Layered Architecture*, in Pattern Languages of Program Design, D.C. Schmidt and J.O. Coplien, eds., MA: Addison-Wesley, pp. 119-128.

Russel, S. and P. Norvig (1995), *Artificial Intelligence: A Modern Approach*, Prentice Hall.

Saffiotti, A. (1997), *Fuzzy Logic in Autonomous Robotics: Behavior Coordination*, in Procs. of the 6th Int. Conf. on Fuzzy Systems, Barcelona, Spain, pp. 573-578.

Sanz, R. and J. Zalewski (2003), *Pattern-Based Control Systems Engineering*, IEEE Control Systems, vol. 23, nr. 3, pp. 43-60.

Schoop, R., R. Neubert, and A. Colombo (2001), *A Multiagent-based Distributed Control Platform for Industrial Flexible Production Systems*, 27th Annual Conference of the IEEE Industrial Electronics Society, pp. 279-284.

Schuster, G. and M. Winrich (2009), *Robotics Safety*, Rockwell Automation, Inc.

Seghrouchni, A. El Fallah, and A. Suna (2004), *CLAIM: A computational language for autonomous, intelligent and mobile agents*, in M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, Programming Multiagent Systems, first international workshop (Pro-MAS'03), vol. 3067 of LNCS, pp. 90-110. Springer Verlag.

Selic, B. (1996), *An Architectural Pattern for Real-Time Control Software*, in Proc. PLoP'96 3rd Annual, Pattern Languages of Programming Conf., Monticello, IL.

Selic, B. (2003), *Architectural Patterns for Real-Time Systems*, in UML For Real: Design of Embedded Real-Time Systems, L. Lavagno, G. Martin, and B. Selic, eds., Kluwer Academic Publishers, Norwell, MA, pp. 171-188.

Slotine, J.E. and W. Li (1988), *Adaptive Manipulator Control: A Case Study*, IEEE Transaction on Automatic Control, vol. 33, nr. 11, pp. 995-1003.

Smits, R., T.D. Laet, K. Claes, P. Soetens, J.D. Schutter, and H. Bruyninckx (2008), *Orocos: A Software Framework for Complex Sensor-Driven Robot Tasks*, IEEE Robotics and Automation Magazine.

Soetens, P. and H. Bruyninckx (2005), *Realtime Hybrid Task-based Control for Robots and Machine Tools*, in Proceedings of the IEEE International Conference on Robotics and Automation (April 18-22, 2005), Barcelona, Spain, pp. 259-264.

Soetens, P. (2006), *A Software Framework for Real-Time and Distributed Robot and Machine Control*, PhD thesis, Katholieke Universiteit Leuven, Belgium, ISBN: 90-5682-687-5.

Starrenburg, J.G., W.T.C. Van Luenen, W. Oelen, and J. Van Amerongen (1996), *Learning Feedforward Controller for a Mobile Robot*, Control Engineering Practice, vol. 14, nr. 9, pp. 1221-1230, ISSN: 0967-0661.

Tadele, T.S. (2009), *Development of OROMACS Browser*, Individual Design Project, report no. 021CE2009, Control Laboratory, University of Twente, Enschede, The Netherlands.

Takagi, T. and M. Sugeno (1985), *Fuzzy Identification of Systems and Its Applications to Modeling and Control*, IEEE Trans. Syst. Man Cybern, SMC-15, pp. 116-132.

The MathWorks Inc. (2009), http://www.mathworks.com/.

Thielscher, M. (2005), *FLUX: A Logic Programming Method for Reasoning Agents*, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules.

Tomizuka, M (2002), *Mechatronics: from the 20th to 21st century*, Control Engineering Practice, vol. 10, issue 8, August 2002, pp. 877-886.

Van Amerongen, J. (1981), *MRAS: Model Reference Adaptive Systems*, Journal A, vol. 22, nr. 4, pp.192-198.

Van Amerongen, J., H.J. Coelingh, and T.J.A. De Vries (2000), *Computer Support for Mechatronic Control System Design*, Robotics and Autonomous Systems, vol. 30, nr. 3, pp. 249-260, PII: SO921-8890(99)00090-1.

Van Amerongen, J. (2003), *Mechatronic Design*, Journal of Mechatronics, vol. 13, issue 10, December 2003, Elsevier, pp. 1045-1066, ISSN: 0957-4158.

Van Amerongen, J. (2006), *A MRAS-based Learning Feed-forward Controller*, 4th IFAC-Symposium on Mechatronic Systems, 12-14 Sep. 2006, Heidelberg, Germany, pp. 1-6. Elsevier. ISBN 978-3-902661-17-3.

Van Amerongen, J. (2007), *Mechatronic Design - A Port-Based Approach*, keynote speech in the Fourth International Symposium on Mechatronics and its Applications (ISMA07), Sharjah, United Arab Emirates, March 26-29, pp. 1-8.

Van Breemen, A.J.N. and T.J.A. De Vries (2000), *An Agent-Based Framework for Designing Multi-Controller Systems*, Fifth Int. Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology, Manchester, U.K., pp. 219-235.

Van Breemen, A.J.N. (2001), *An Agent-Based Multi-Controller Systems: A Design Framework for Complex Control Problems*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-365-1595-5.

Van Breemen, A.J.N. and T.J.A. De Vries (2001), *Design and Implementation of a Room Thermostat using an Agent-based Approach*, Control Engineering Practice, vol. 9, nr. 3, pp. 233-248, ISSN 0967-0661.

Van Breemen, A.J.N., K. Crucq, B.J.A. Krose, M. Nuttin, J.M. Porta, and E. Demeester (2003), *User-Interface Robot for Ambient Intelligent Environments*, in Proc. of ASER'03, Bardolino, Italy, pp. 132-139, March 13-15, 2003.

Van Brussel, H.M.J. (1996), *Mechatronics - A Powerful Concurrent Engineering Framework*, IEEE/ASME Trans. Mechatronics, vol. 1, nr. 2, pp. 127-136.

Velthuis, W.J.R. (2000), *Learning Feed-Forward Control - Theory, Design and Applications*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-365-1412-6.

Verwoerd, M. (2005), *Iterative Learning Control: A Critical Review*, PhD thesis, University of Twente, Enschede, The Netherlands, ISBN 90-365-2133-5.

Visioli, A. (2006), *Advances in Industrial Control - Practical PID Control*, 1st edition, Springer, ISBN: 1846285852.

Vrba, P. (2003), *MAST: Manufacturing Agent Simulation Tool*, in Proc. the IEEE Conference on Emerging Technologies and Factory Automation, vol. 1, pp. 282-287, Lisbon, Portugal.

Vrba, P. and V. Marik (2005), *Simulation in Agent-based Manufacturing Control Systems*, in Proc. the IEEE Int. Conference on Systems, Man and Cybernetics, pp. 1718-1723.

Vrba, P. (2006), *Simulation in Agent-based Control Systems: MAST case study*, Int. Journal of Manufacturing Technology and Management, vol. 8, no.1/2/3, pp. 175-187.

Waarsing, B.J.W., M. Nuttin, and H. Van Brussel (2003a), *A Software Framework for Control of Multi-sensor, Multi-actuator Systems*, in Proc. of the 11th International Conference on Advanced Robotics (ICAR), Coimbra, Portugal, pp. 41-46.

Waarsing, B.J.W., M. Nuttin, and H. Van Brussel (2003b), *Behaviour-based Mobile Manipulation: The opening of a door*, in Proc. of ASER'03, Bardolino, Italy, pp. 168-175, March 13-15, 2003.

Wheeler, H.A. and D. Dettinger (1949), *Wheeler Monograph 9*, p. 7.

Winikoff, M. (2005), *JACK^{TM} Intelligent Agents: An Industrial Strength Platform*, chapter 7, pp. 175-193, in Bordini et al., editors, Multi-Agent Programming: Languages, Platforms and Applications, vol. 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer.

Wooldridge, M. (1999), *Intelligent agents*, Multiagent Systems, pp. 27-77, The MIT Press, Cambridge, Massachusetts.

Wooldridge, M. (2002), *An Introduction to Multiagent Systems*, John Wiley & Son Ltd., Chichester, England.

Yamada, Y., Y. Hirasawa, S. Huang, Y. Uematsu, and K. Suita (1997), *Human-Robot Contact in the Safeguarding Space*, IEEE/ASME Trans. on Mechatronics, vol. 2, nr. 4, pp. 230-236.

Acknowledgments

In May 2006, I started my Ph.D. research within the Control Engineering group at the University of Twente. More than four years passed and I am now writing the last lines of my Ph.D. thesis, which would not have accomplished without the help and support of numerous people.

First of all, it is a great pleasure for me to thank my promotor, prof.dr.ir. Job van Amerongen, and my assistant promotor, dr.ir. Theo J.A. de Vries, who guided me from the first steps to the ending stage of the research. Their continuous guidance and support helped me in all phases of my research. This thesis had been formulated from the discussions at regular meetings that we had on Thursday. Prof. Job and Theo suggested valuable and insightful comment that directed me to structure and to clarify my thoughts. In addition, they provided invaluable support through reading and correcting all versions of chapters. Their comment and correction greatly improved my thesis. Theo helped me to translate the summary of this thesis into Dutch. One time again, I would like to express my gratefulness to their guidance, support and help.

This research was financially supported by the Vietnamese Government through the 322 Project for four years (5/2006 - 4/2010). I am especially indebted to the Vietnamese Government for this financial support. Because my Ph.D. research was extended, Theo J.A. de Vries, on behalf of Imotec B.V., awarded me a scholarship for eight months (6/2010 - 1/2011) that enabled me to complete my research. I am acknowledged to this valuable financial support.

I would like to thank all members of the promotion committee for their careful reading and useful comments to my thesis. Many thanks to members of the Control Engineering group for a pleasant working atmosphere. I am thankful to Carla for her continuous helps with all kind of documents relating to my studying and living in the Netherlands. I would like to thank all colleagues and friends at the Hanoi University of Science and Technology for their supports.

A special thanks goes to all Vietnamese friends, who have studied and worked at the University of Twente, for what you shared and helped me during the past four years. I really enjoyed the cleaning job after all kinds of parties that you put on me, an underpromotion PhD student. But, from now on you know the truth that i am out

of service. I would like to thank all salseros and salseras in Enschede for what we enjoyed together in social salsa and zouk dance on Thursday and Sunday night at the Rico Latino. Thank the dancing and music because you helped me to relax after hard working days. Thank Enschede, the city where i lived and worked.

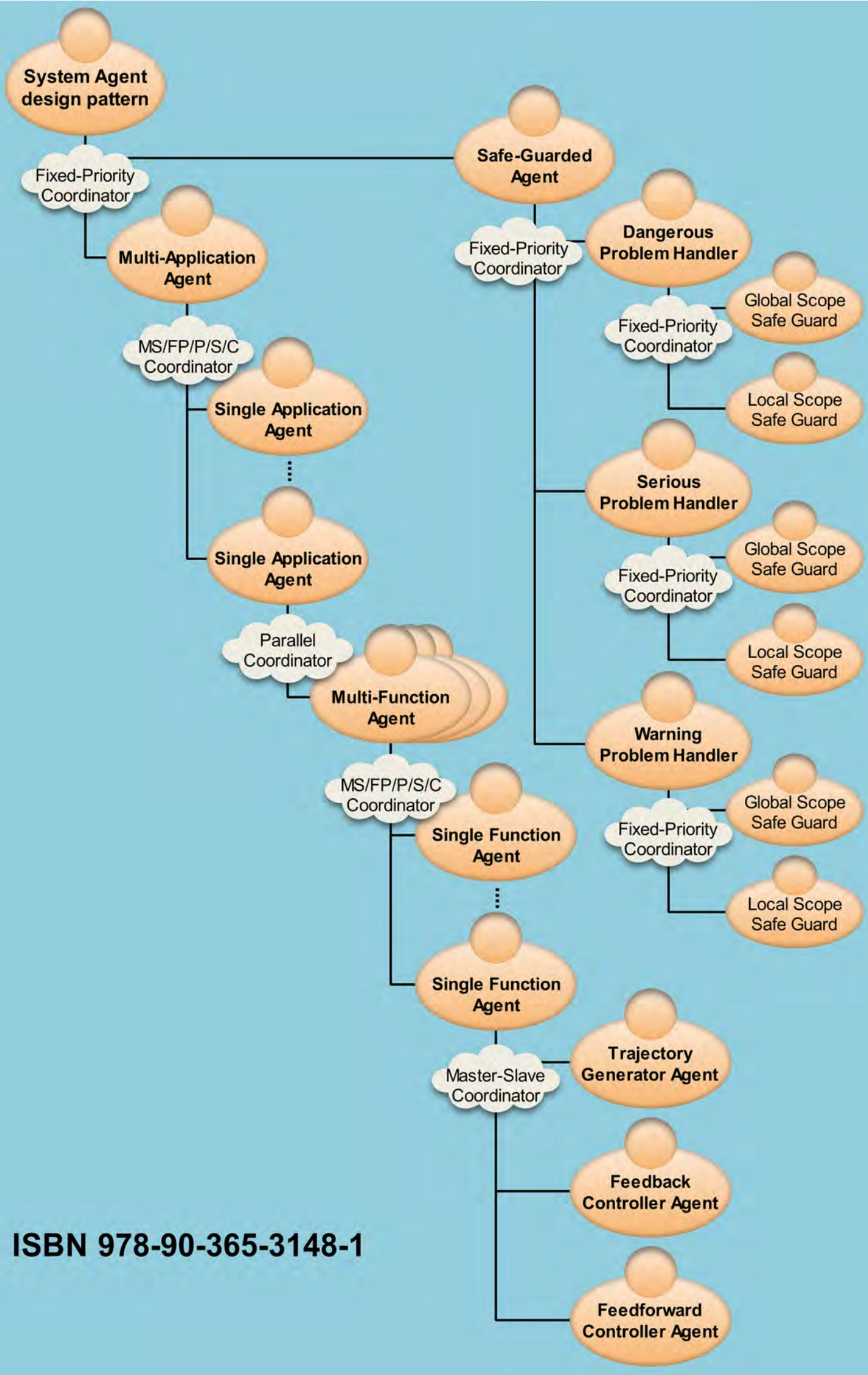
My most gratefulness is to all my family and especially to my parents. Without their love and understanding I would not be able to accomplish my Ph.D thesis. This thesis is dedicated to them.

Enschede, 11th of January, 2011.

Dao Ba Phong

About the Author

Dao Ba Phong was born on September 11th, 1978 in Haiduong, Vietnam. After completing his secondary education in 1996, he started his bachelor study in Electrical Engineering at the Hanoi University of Science and Technology. In 2001, he obtained his bachelor degree and started working as a lecturer at the Faculty of Mechanical Engineering of the same university. In 2003, he obtained his M.Sc. degree in Measurement and Control with a thesis on fuzzy controller design for motion systems. In 2006, he was granted a full scholarship from the Vietnamese Government through the 322 Project for a Ph.D. research program at the University of Twente in Enschede, under the supervision of prof.dr.ir. Job van Amerongen and dr.ir. Theo J.A. de Vries. He has successfully completed his Ph.D. research on Safe-Guarded Multi-Agent Control for Mechatronic Systems, with the focus on developing an implementation framework and design patterns.



ISBN 978-90-365-3148-1