UNIVERSIDAD POLITÉCNICA DE MADRID

Departamento de Automática, Ingeniería Electrónica
e Informática Industrial

E.T.S. INGENIEROS INDUSTRIALES

Model-based Self-awareness Patterns for Autonomy

PhD Dissertation

MSc. Carlos Hernández Corbato

Advisors: Dr. Ing. Ricardo Sanz Bravo
Dr. Ing. Ignacio López Paniagua

2013

Model-based Self-awareness Patterns for Autonomy

A mi abuelo Gregorio,
que siempre trabajó y construyó
para los demás

Abstract

Technical systems are becoming more complex, they incorporate more advanced functionalities, they are more integrated with other systems and they are deployed in less controlled environments. All this supposes a more demanding and uncertain scenario for control systems, which are also required to be more autonomous and dependable. Autonomous adaptivity is a current challenge for extant control technologies. The ASys research project proposes to address it by moving the responsibility for adaptivity from the engineers at design time to the system at run-time.

This thesis has intended to advance in the formulation and technical reification of ASys principles of model-based self-cognition and having systems self-handle at run-time for robust autonomy. For that it has focused on the biologically inspired capability of self-awareness, and explored the possibilities to embed it into the very architecture of control systems.

Besides self-awareness, other themes related to the envisioned solution have been explored: functional modeling, software modeling, patterns technology, components technology, fault tolerance. The state of the art in fields relevant for the issues of self-awareness and adaptivity has been analysed: cognitive architectures, fault-tolerant control, and software architectural reflection and autonomic computing. The extant and evolving ASys Theoretical Framework for cognitive autonomous systems has been adapted to provide a basement for this selfhood-centred analysis and to conceptually support the subsequent development of our solution.

The thesis proposes a general design solution for building self-aware autonomous systems. Its central idea is the integration of a metacontroller in the control architecture of the autonomous system, capable of perceiving the functional state of the control system and reconfiguring it if necessary at run-time.

This metacontrol solution has been formalised into four design patterns: i) the Metacontrol Pattern, which defines the integration of a metacontrol subsystem, controlling the domain control system through an interface provided by its implementation component platform, ii) the Epistemic Control Loop pattern, which defines a model-based cognitive control loop that can be applied to the design of such a metacontroller, iii) the Deep Model Reflection pattern proposes a solution to produce the online executable model used by the metacontroller by model-to-model transformation from the engineering model, and, finally, iv) the Functional Metacontrol pattern,

which proposes to structure the metacontroller in two loops, one for controlling the configuration of components of the controller, and another one on top of the former, controlling the functions being realised by that configuration; this way the functional and structural concerns become decoupled.

A reference architecture and an associated metamodel are the core pieces of the architectural framework developed to reify this patterned solution. The metamodel has been developed for representing the structure and its relation to the functional requirements of any autonomous system. The architecture is a blueprint for building a metacontroller according to the patterns. This metacontroller can be integrated on top of any component-based control architecture. At the core of its operation lies a model of the control system that conforms to the metamodel.

An engineering process and accompanying assets have been constructed to complete and exploit the architectural framework. The engineering process defines the methodology to follow to develop the metacontrol subsystem from the functional model of the controller of the autonomous system. The assets include software libraries that provide a domain and application-independent implementation of the metacontroller. They can be used in the implementation phase of the methodology.

Finally, the complete solution has been validated in the development of an autonomous mobile robot that incorporates an metacontroller. The functional self-awareness and adaptivity properties achieved thanks to the metacontrol system have been validated in different scenarios. In these scenarios the robot was able to overcome failures in the control system thanks to reconfigurations performed by the metacontroller.

Resumen

Los sistemas técnicos son cada vez más complejos, incorporan funciones más avanzadas, están más integrados con otros sistemas y trabajan en entornos menos controlados. Todo esto supone unas condiciones más exigentes y con mayor incertidumbre para los sistemas de control, a los que además se demanda un comportamiento más autónomo y fiable. La adaptabilidad de manera autónoma actualmente es un reto para tecnologías de control. El proyecto de investigación ASys propone abordarlo trasladando la responsabilidad de la capacidad de adaptación del sistema de los ingenieros en tiempo de diseño al propio sistema en operación.

Esta tesis pretende avanzar en la formulación y materialización técnica de los principios de ASys de cognición y autoconsciencia basadas en modelos y autogestión de los sistemas en tiempo de operación para una autonomía robusta. Para ello, el trabajo se ha centrado en la capacidad de autoconsciencia, inspirada en los sistemas biológicos, y se ha investigado la posibilidad de integrarla en la arquitectura de los sistemas de control.

Además de la autoconsciencia, se han explorado otros temas relevantes: modelado funcional, modelado de software, tecnología de los patrones, tecnología de componentes, tolerancia a fallos. Se ha analizado el estado de la técnica en los ámbitos pertinentes para las cuestiones de la autoconsciencia y la adaptabilidad en sistemas técnicos: arquitecturas cognitivas, control tolerante a fallos, y arquitecturas software dinámicas y computación autonómica. El marco teórico de ASys existente de sistemas autónomos cognitivos ha sido adaptado para servir de base para este análisis de autoconsciencia y adaptación y para dar sustento conceptual al posterior desarrollo de la solución.

La tesis propone una solución general de diseño para la construcción de sistemas autónomos autoconscientes. La idea central es la integración de un meta-controlador en la arquitectura de control del sistema autónomo, capaz de percibir el estado funcional del sistema de control y, si es necesario, reconfigurarlo en tiempo de operación.

Esta solución de metacontrol se ha formalizado en cuatro patrones de diseño: i) el Patrón Metacontrol, que define la integración de un subsistema de metacontrol, responsable de controlar al propio sistema de control a través de la interfaz proporcionada por su plataforma de componentes, ii) el patrón Bucle de Control Epistémico, que define un bucle de control cognitivo basado en el modelos y que se puede aplicar al

diseño del metacontrol, iii) el patrón de Reflexión basada en Modelo Profundo propone una solución para construir el modelo ejecutable utilizado por el meta-controlador mediante una transformación de modelo a modelo a partir del modelo de ingeniería del sistema, y, finalmente, iv) el Patrón Metacontrol Funcional, que estructura el meta-controlador en dos bucles, uno para el control de la configuración de los componentes del sistema de control, y otro sobre éste, controlando las funciones que realiza dicha configuración de componentes; de esta manera las consideraciones funcionales y estructurales se desacoplan.

Una arquitectura de referencia y un metamodelo son las piezas centrales del marco arquitectónico desarrollado para materializar la solución compuesta de los patrones anteriores. El metamodelo ha sido desarrollado para la representación de la estructura y su relación con los requisitos funcionales de cualquier sistema autónomo. La arquitectura es un patrón de referencia para la construcción de una meta-controlador integrando los patrones de diseño propuestos. Este meta-controlador se puede integrar en la arquitectura de cualquier sistema control basado en componentes. El elemento clave de su funcionamiento es un modelo del sistema de control, que el meta-controlador usa para monitorizarlo y calcular las acciones de reconfiguración necesarias para adaptarlo a las circunstancias en cada momento.

Un proceso de ingeniería, complementado con otros recursos, ha sido elaborado para guiar la aplicación del marco arquitectónico. Dicho proceso define la metodología a seguir para construir el subsistema de metacontrol para un sistema autónomo a partir del modelo funcional del mismo. Los recursos asociados incluyen librerías software que proporcionan una implementación del meta-controlador que se puede integrar en el control de cualquier sistema autónomo, independientemente del dominio de la aplicación o de su tecnología de implementación.

Para concluir, la solución completa ha sido validada con el desarrollo de un robot móvil autónomo que incorpora un meta-controlador. Las propiedades de autoconsciencia y adaptación proporcionadas por el meta-controlador han sido validadas en diferentes escenarios de operación del robot, en los que el sistema era capaz de sobreponerse a fallos en el sistema de control mediante reconfiguraciones orquestadas por el metacontrolador.

Agradecimientos

La sensación al escribir estas líneas es de liberación, de poner fin a un viaje que ha durado siete años. Ha sido un camino largo y difícil, con momentos de frustración, temporadas sin rumbo fijo en el océano inacabable del conocimiento; pero también con pequeñas victorias, esos instantes de lucidez, de *eureka*, en los que se añade un pequeño cabo al mapa y por un fugaz instante uno cree que ya puede cartografiar todo el mar. Son muchas las personas que me han acompañado en este viaje, y a las que tengo mucho que agradecer.

De entre todas esas personas, una ha sido especialmente importante, y forma parte de esta tesis tanto como yo mismo: Ricardo, mi director de tesis. Recuerdo como si fuera ayer el día que le conocí personalmente (ya me había deslumbrado antes en algunas charlas y como profesor de una asignatura). Conversamos de un nuevo proyecto que empezaba en el que iban a investigar el cerebro de las ratas y su aplicación a la construcción de máquinas inteligentes, emocionales, ¡incluso conscientes! Representaba todo aquello por lo que años atrás yo había decidido estudiar ingeniería. Lo que he recibido de él desde entonces ha sido mucho más de lo que podía pensar. Ha sido orientación y guía para realizar esta tesis, sí, pero también ha sido una introducción completa al mundo de la ciencia y la investigación, dándome oportunidades para viajar y formarme, asistir a congresos, participar en proyectos, etc. Siempre trabajando codo con codo, como compañeros.

Otro papel muy destacado ha sido también el de Ignacio, primero compañero y luego co-director de tesis, siempre buen consejero y guía. Su rigor en el método, espíritu práctico y claridad de ideas han sido fundamentales para sacarme del atolladero cuando encallaba. Son Ricardo e Ignacio dos personas extraordinarias. No son solamente dos de las personas más inteligentes que he conocido, auténticos hombres de ciencia y humanistas, cuya pasión y curiosidad científicas son contagiosas, sino que además son personas maravillosas, generosas, cuya amistad atesoro como el mejor resultado de esta tesis.

Tengo mucho que agradecer también a mis compañeros y amigos de doctorado, con quienes he compartido estos años de pasión investigadora: José Emilio, Miguel, Iñaki, Adolfo, Antonio, Alberto, Gonzalo, David, Marcos. Sin nuestras charlas de café, y nuestros partidos de fútbol y pádel el camino habría sido mucho menos divertido, y seguramente menos sano física y psíquicamente. Una mención especial se merecen Paloma y Lupe, que no contentas con aguantarme todos estos años, ¡se han

leído y revisado esta tesis en las últimas semanas! Ahora en serio, vuestra contribución ha sido muy importante para mi.

Este trabajo no habría sido el mismo sin mis compañeros investigadores de ASLab: Manuel, Jaime, José Luis y Juan. Nuestras discusiones sobre autonomía, función o cognición han sido fundamentales para ir moldeando las ideas presentadas en esta tesis. Julia, a ti te debo la luna, gracias por contar conmigo y por ayudarme siempre, salvando mares y océanos, eres un ejemplo de tesón y dedicación. Tampoco me quiero olvidar de los más jóvenes: José Alberto, Paco, Eusebio, Javi, Mikel y Alberto, con quienes tantas horas de cacharreo y compilación he compartido, sufriendo a veces, pero siempre disfrutando de su pasión por nuestras locuras.

Mi agradecimiento se extiende a toda la gente del laboratorio que me ha ayudado: a Rosa, Tere y Carlos, siempre facilitándome los trámites y gestiones, a Ángel, con quien tan buenos ratos de taller he pasado, y a todos los profesores. Y muy especialmente a Ramón, mentor y padre espiritual de todo alumno de doctorado que pasa por allí: gracias por no dejar de empujarnos hacia adelante.

Fuera de la universidad son muchas las personas que han sufrido mi tesis, amigos que han entendido y aguantado que siempre estuviese a tumbos con ella, poniendo trabas para quedar y hacer planes. Muchas gracias a todos vosotros, que aún así me habéis perseguido para salir y pasar un rato divertido con unas cañas, en el cine o en alguna escapada de fin de semana. ¡Qué habría sido de mi sin vosotros! Así que qué menos que citaros y agradecéroslo.

Santi, Jesús, Sergio, Alberto, Juan, Roberto, Pablo, cuántos exámenes y duras pruebas superamos juntos en la carrera. ¡Y qué gratos recuerdos! Aunque casi son incluso mejores los de nuestras tertulias alrededor de unas pizzas rememorando esas anécdotas. Puede que el doctorado sólo lo haya cursado yo, pero de alguna forma lo hemos seguido haciendo juntos.

Estos años los he compartido también con Lucía, Rober, Mercedes, Edu, Vaneza, Marcos, Elena, Laura, y recientemente Pedro. Nunca dejará de maravillarme vuestra altruista predisposición a aguantar de vez en cuando mis chapas sobre la autoconsciencia de las máquinas, ¡no se me ocurre un apoyo más incondicional! Y no me olvido de Laura, Lalo, Ernesto y Fran, que aunque ahora nos veamos menos, han sido parte importante de este maravilloso grupo.

Recuerdo también a Dani, Miguel e Iván, con quienes empecé la carrera allá en el mes de septiembre de 2001. Desde entonces siempre os he tenido ahí, aunque la tesis me recluyera. Y mis amigos más recientes, Alex y Maribel, gracias por aportarme otro punto de vista a las ciencias de la mente, el cine y las hamburguesas, y por tantas tardes de domingo entretenidas profundizando en éstos apasionantes y sabrosos temas.

Ahora recuerdo también todas esas veces en las que todos me habéis mostrado vuestra absoluta confianza en que sería capaz de culminar con éxito este trabajo, y lo importante que ha sido cuando me han asaltado las dudas.

No me quiero olvidar de mis nuevos compañeros en Global Incubator e Inner Virtuoso, que han aparecido en éste último año del viaje, pero han tenido su racioncita

de mi tesis y me han ayudado a compaginarla con nuestro “tranquilo" y “aburrido" trabajo sin que me volviera majareta.

Los últimos agradecimientos, pero los más importantes, son para las personas que han estado a mi lado desde siempre, dándome su cariño y ayuda: mi familia. Mis padres, mi hermano, mi abuela, mi tía, mis abuelos que ya no están, y el resto de la familia. Ellos me han inculcado los valores del trabajo, la honradez y la bondad. Ante todo el valor de ser una buena persona. Todo lo que he logrado se lo debo a ellos.
Gracias.

Decía al comenzar que tengo la sensación de poner fin a un viaje. En realidad sé que ha sido sólo una etapa. Hacer esta tesis me ha enseñado mucho, por ejemplo cuánto me queda por saber y descubrir, y me ha preparado bien para afrontar nuevos proyectos. Pero lo más importante que he aprendido es que con la gente de la que tengo la inmensa fortuna de estar rodeado no puedo estar mejor preparado para lo que me depare el futuro.

Madrid, domingo 29 de septiembre de 2013.

International acknowledgements

I would like to thank Antonio Chella and Haris Dindo, who kindly accepted to review this work and whose comments have been very helpful to improve its quality, and Radu Calinescu, Igor Aleksander, José Luis Fernández and Vicente Matellán, who accepted to be part of the defense committee and will probably be reviewing it by now.

During my PhD I have had the oportunity to meet and work with many other brilliant people. I am very grateful with the Spanish and European communities of cognitive systems: Retecog and Eucognition. They have given me the opportunity to meet most of these people and present and discuss my work with them. I am specially in debt with Toni Gomila, Xabier Barandiaran, Manuel Bedia and Raúl Arrabales, their passion for science and tireless initiative is an inspiration for the upcoming new generations of researchers.

There are some other scientists I have never met in person, but whose thinking is very familiar to me: James Albus, Bernard Baars, Stan Franklin, Mogens Blanke and George Klir amongst many others. They are the giants whose shoulders I have tried to climb.

I am also grateful to my colleagues in the HUMANOBS European project, with whom I’ve enjoyed some of my most exciting scientific and technical discussions in front of a whiteboard, and some funny and deeply philosophical conversations about life around our beers.

I would like to specially thank Kris, Eric and Helgi, who warmly welcomed me the two summers I spent at Reykjavik University. They are first-class researchers from whom I have learned a lot, and even better people.

Some of my most appreciated memories are of these two summers in Iceland. They have been an amazing experience, mainly because of the great friends I met: Pradipta, Matteo, Stefanía, Claudio, María, Jenni, Rolanda, Paulo, Ágnes, and many others that made me feel at home.

Madrid, September 2013.

Contents

List of Figures

List of Tables

Part I

The Context

Chapter 1

Autonomous Systems

Our society critically relies on technical systems, from electrical power grids to air traffic control systems, passing by chemical plants. And the pace on the demand on technology is doing but to augment. This technical growth is two-sided: technical systems are spreading everywhere in our lives, and at the same time we depend more on them. The result is also two-fold: technology is becoming more complex, and the requirements on it regarding dependability are becoming more strict. This situation presents important challenges to the engineering of autonomy in systems.

The work developed in this research project addresses the problem of providing technical systems with enhanced robustness in their autonomous behaviour, in this context of the infrastructure of modern society critically relying on them. The implications of this scenario are analysed in this introductory chapter, discussing the necessity for more robust autonomous systems and presenting the rationale behind the cognitive approach to build them.

1.1 Control Technology

Control systems are subsystems designed to improve the operation of most technical systems, or even to make their operation possible at all, constituting an integral part of them. In such a technified society as ours control systems or controllers permeate everywhere: they are in the heating of our homes, in the landing gear for planes, in our smart phones, heart pacemakers, or, at larger scales, keeping the electrical network stable. They drive the behaviour of systems to accomplish their function when a human operator is not desirable, for example because of safety or performance reasons, or because it is simply not possible. Control technology is the engineering solution to the automation of systems.

Departing from knowledge of the system and its processes, and the targeted behaviour, control engineers design a controller that makes it behave at runtime as re-

quired, even in the presence of some disturbances. However, conventional control technology has limits. It is not possible to design a controller for any system to obtain any specific behaviour.

And and even when it is theoretically feasible, there are unexpected situations at run-time that controllers cannot handle.

Example

Cruise control is a paradigmatic example of control engineering in everyday life. These controllers are capable of maintaining a desired speed of the vehicle. They use a very simple algorithm tuned according to the dynamical model of the car (not a very complex one is required, anyway). This was a problem solved long time ago by classic control. However, designing a control system to completely drive the car, that is, targeting the specification of the car's complete behaviour (not just its linear velocity), has but recently been solved in Google driverless Car [57], and with limitations.

1.1.1 Control Systems

This is the control engineering scenario: given a technical *system* (the electric network, a mobile robot) immersed in an *environment* (the whole European electric grid, an industrial facility), and some user needs that demand a certain functionality from the system (electricity available at any time without interruptions, the robot patrolling a certain area, etc.), also refered to as the system's *mission*, specified into a set of requirements, design the control system that makes the system fulfil the mission in the presence of disturbances.

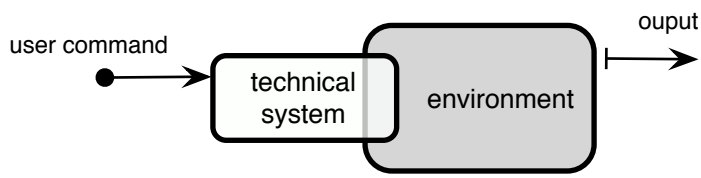


Figure 1.1: Technical systems are engineered to provide a certain functionality to their users: i.e. produce a certain output serviced according to the user command.

Let us take a general systems perspective. From this viewpoint, the user's requirements are but a certain desired behaviour for the system, that is a specific mapping between the user command to the system and the resulting output from it (see Fig. 1.1). For example, the drivers' command for the cruise control in a modern automobile is a desired speed, and the output is that the car maintains its velocity at that value. But it could be a more complex mapping, such as setting a destination city as

command and getting with the car there after several hours, in the case of Google's car.

In most cases it is not possible to directly implement the desired behaviour in a designed system's structure. Perfect and complete knowledge of the environment are never available, unmodelled dynamics and unexpected disturbances are always present. But even if we considered the world as perfectly and completely modelled, the formal resolution of the mapping function from the desired behaviour to a structure that renders it could not be physically realisable, that even in the case it were theoretically possible to find that solution, which could be not. As a result, the real behaviour at runtime of the engineered system would not fit the required one. Figure 1.2 shows a graphical interpretation.

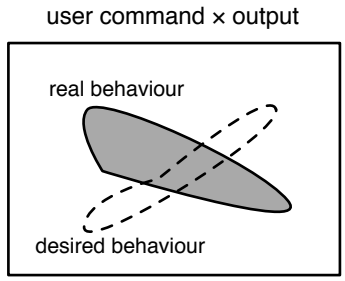


Figure 1.2: The desired and real behaviours are subsets of the set defined by all possible user commands and output results.

This is where control technology comes into play. To overcome the previous issues, a control subsystem is added to the system. Its objective is to drive the behaviour of the engineered system so that it achieves the demanded functionality. This way, the coupled behaviour of the control subsystem and the rest of the system plus its environment fulfils the desired behaviour (see figure 1.3). The control system is designed using the available knowledge about the technical system and its environment, e.g. in the form of a differential equations model.

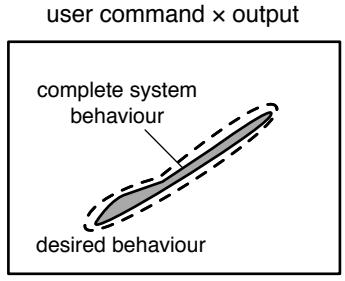
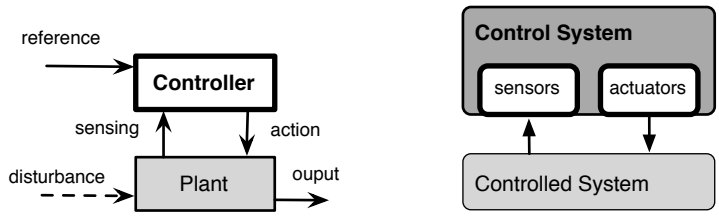


Figure 1.3: The resulting behaviour of the controlled system is to fall within the region specified by the requirements.

Figure 1.4a shows the control view of the issues presented. The *controller* receives sensing information from and actuates over the *plant*, to achieve a certain *reference* value as output. The concept of plant encompasses the technical system (without the controller) and its environment.

This control process is implemented by engineers in a *control system*, which senses and actuates through devoted I/O components: sensors and actuators. The right part

of figure 1.4b schematizes the physical or implementation view.



(a) The cybernetic view of control.

(b) The basic elements in the architecture of a control system.

Figure 1.4: The cybernetic view of control and the basic elements of the architecture of control systems.

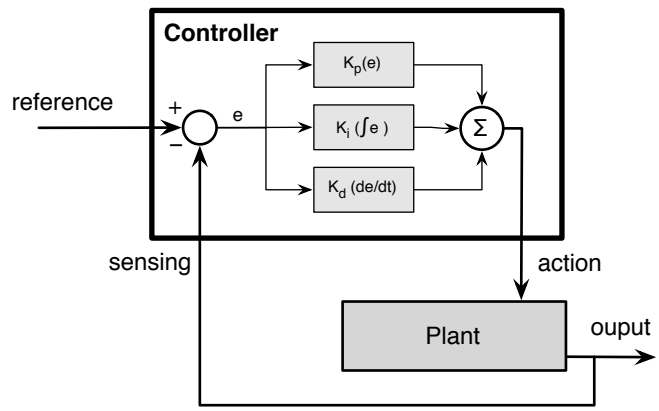


Figure 1.5: A PID controller.

For very elementary cases (e.g. the PID controller in figure 1.5), the design of the *control system* can be very simple, consisting of a basic component, i.e. a controller, reading information through a sensor and actuating through an actuator. However, most usually the *control system* design encompasses heterogeneous sensors and actuators, operator interfaces, etc., which results in a complex physical design comprising several components interconnected (e.g. state-space controller of figure 1.6).

Control engineers use knowledge about the mission requirements, the plant, but also about control technology and engineering, to design control systems (see figure 1.7). This is discussed in the following section.

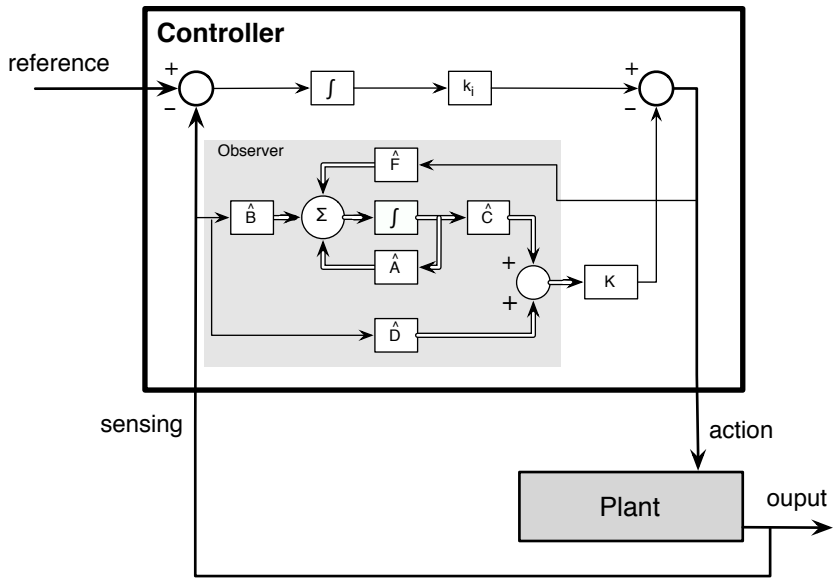


Figure 1.6: A servo-controller with a minimum order observer, based on the state space technique, with a matrix-based formulation (A,B,C,D) of the differential equations model of the system.

1.1.2 Control Engineering Life-cycle

The standard life-cycle of any engineered system consists of two basic stages:

1. **Engineering**: basically design and implementation.
2. **Runtime**: exploitation and maintenance.

Many systems undergo an interwoven sequence of stages that can be ascribed to either one of these two types; e.g. in maintenance, when normal runtime operation is halted to perform some change in the system's structure, can be considered as engineering when plant engineers perform actual process re-design on the fly. It can even be the case that the phases overlap in time, e.g. when the system is kept operating during maintenance. However, in the case of autonomous systems, where no human intervention is desirable or even possible for the previous purposes, there is a clear gap between engineering and runtime.

The engineering phase, which includes the design of the control system, is what determines and fixes the traits and functional capabilities that an autonomous system will exhibit during operation. Let us analyse why.

The engineering stage for the building a control system includes the following activities:

- mission specification
- plant and controller analysis

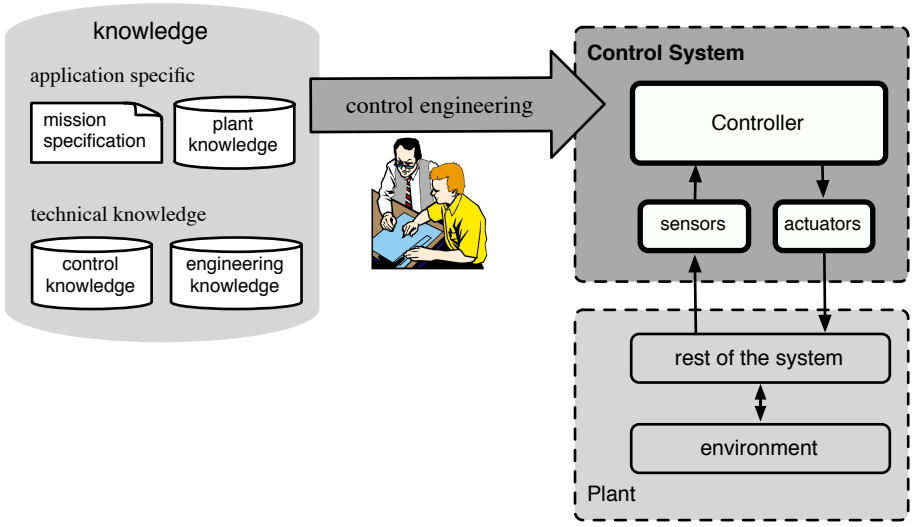


Figure 1.7: Knowledge about the system and its mission, and about engineering and control technology is used to build the system during the engineering phase.

- controller design
- controller implementation
- integration, validation and deployment

A priori knowledge is used intensively in the process (see figure 1.7), and embedded in the resultant control system. Plant knowledge in the form of mathematical models is normally used to design the controller following control theory methodologies (root-locus method, State Space techniques, Frequency response analysis, etc.). Evidently, these models are not perfect, and the real system's behaviour may deviate from the expected, due to inexactitudes, unmodelled dynamics, etc. In classical control the effect of this uncertainty is modelled as a *disturbance* signal that quantitatively affects the plant, deviating the evolution of the values of its magnitudes from the modelled. We shall call this uncertainty about the plant *intensive uncertainty* [90]. There are other *qualitative* uncertainties, whose effect cannot be quantified, that will be discussed later.

At the very core of control theory lies the feedback technique, which allows to compensate for the intensive uncertainty. Thanks to the feedback loop the actuation of the control over the plant is not fixed for each user command, but computed at runtime, using information about the instantaneous state of the plant, i.e. the sensory input. This allows the system to accommodate up to a certain point to the disturbances, and maintain the desired behaviour. The control policy implemented in the controller, however, is fixed at design time, based on the plant knowledge and the specific control technique selected, e.g. PID, state feedback, etc. (see figures 1.5, 1.6). The capability to adapt is therefore limited by it at run time.

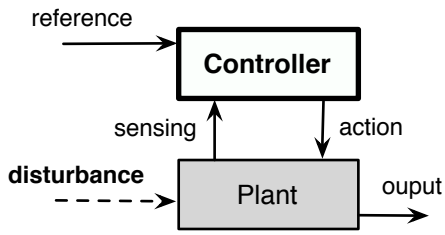


Figure 1.8: The disturbance signal in the control schema. The effect of the modelling errors and the unmodelled dynamics that affect the plant are modelled as a disturbance signal. This assumes that the contribution is quantitative, the rest of the model being valid, which sometimes is not the case.

Nevertheless, the previous is not the only assumption related to knowledge that fixes the properties of the system at engineering time. In order to implement the controller and integrate it with the rest of the system, engineering knowledge of the rest of the subsystems with which it will interact at runtime is also used. All this knowledge suppose assumptions on which control engineers rely for the developed system to behave at runtime as required. This way, uncertainty has been traditionally regarded as affecting the environment, since the engineered system was considered as perfectly defined in both static structure and dynamic operation by design. This was so even when referring to not so well defined systems, such as large production plants in which chemical processes were and remain not so well known. However, this is far from true, specially as systems grow in complexity. This is one of the focal points of this thesis.

1.1.3 Limits of Conventional Control

Control techniques allow to automate the desired functionality of technical systems to some extent, by accounting for quantitative uncertainty. The most basic feedback control techniques, such as the PID controller, can only account for a very limited quantitative uncertainty. As soon as the behaviour of the plant departs from the model used to design the controller, it fails. This is because the controller is completely fixed at run-time, and thus limited by the boundaries of validity of the model of the plant used to obtain it.

New intelligent control techniques allow to modify or change the controller at runtime to some extent. Model reference based adaptive control [86] (MRAC) allows to change the parameters of the controller during operation based on knowledge about the plant and observation of its actual behaviour, so that it increases its adequacy to it. Mode switching controllers follow a similar schema, but instead of tuning the parameters of an otherwise fixed control strategy, they switch between a battery of alternative controllers, so if the behaviour of the plant changes qualitatively, the control system can switch to another different control strategy. Note that we cannot speak of the controller accounting for qualitative uncertainty, since the existence of the predefined strategy in the battery implies previous knowledge about the “unexpected” situation.

Notwithstanding the previous solutions and others that will be described later, hu-

mans still do play a critical role during operation of many "automated systems". We can see that in process plants, where operators and plant engineers supervise and control the daily operation. Or even in the most cutting-edge control technology for autonomous systems: NASA Mars rovers are monitored 24 hours a day by a crew of highly specialised engineers, who perform operational reconfigurations and make decisions about the robot's course of action, considering the mission and system status, and also making intensive use of their scientific, technical and engineering knowledge.

The former techniques share the control "pattern" presented in figure 1.4a. From a requirements standpoint, we can observe that this means that for each functionality we want to automate in the system, e.g. maintain velocity, a control loop is designed. This way during run-time the system behaviour is automatically driven to achieve that requirement, even in the presence of disturbances, to the extent that the engineering assumptions assumed in design hold. Roughly speaking, what the control engineering process does is to convert functional requirements at design into target references for control loops at run-time.

There are many requirements and environmental conditions for which no specific knowledge suitable to design a closed loop controller is available. Systems are engineered to fulfil them, though, considering how they will behave at runtime. However, not every situation can be envisioned beforehand, and therefore human intervention is required in the operation of technical systems.

In the following we explore the specific issues at hand regarding autonomous systems and their control.

Example

Suppose we design a very basic controller for our robot to traverse corridor-like environments. To achieve this requirement we envision a behaviour for the robot that consists of advancing at a constant speed while maintaining a safe distance to the left wall, measured with a range sensor. A PID controller can be used for that, calculating the rotation velocity from the evolution of the distance to the wall. We have converted the mission to traverse the corridor into a design consisting of:

- a constant command for the linear velocity (a cruise control can be used to address this, using also a PID controller)
- a control architecture, i.e. PID feedback loop with a reference distance

Due to the closed loop induced dynamics, the robot will move oscillating towards and away from the wall. The performance of the movement can be tuned with the PID parameters; again fixed at design-time.

The system is thus only robust to uncertainty in the distance to the wall which is the reference of the control loop. Uncertainty concerning other design decisions can put the system's operation severely at risk: right-side obstacles and left-side doors would most probably invalidate the design solution to solve the mission by moving forward following a wall on the left, too much movement oscillations could lead to excessive battery consumption, etc.

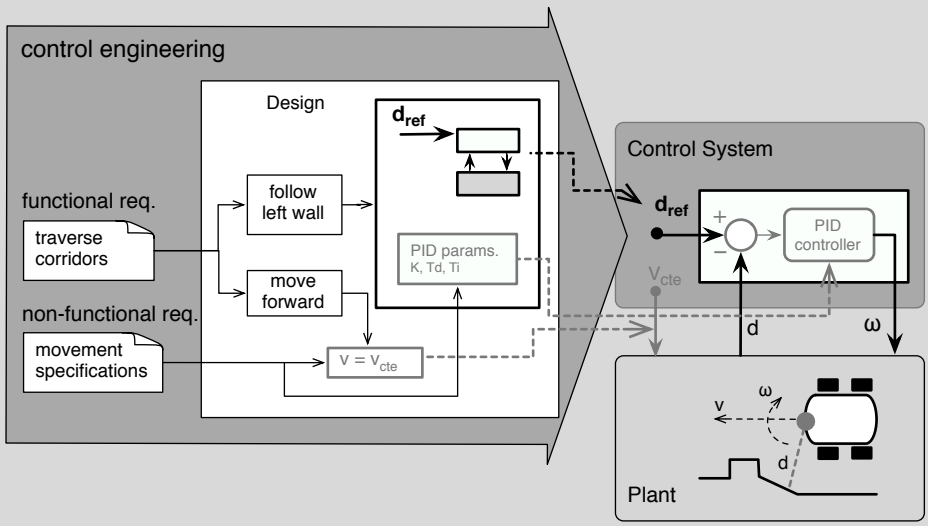


Figure 1.9: Example of how design decisions determine the run-time adaptivity of the system. The functional requirement to traverse the corridor is implemented as a PID feedback control loop at system design. This renders a system that is adaptable at run time to uncertainty in the distance d to the wall, by regulating the robot turning velocity ω. The other design decisions (PID parameters, constant forward speed v_{cte}) fix the rest of the robot behaviour at design-time, not being adaptable upon events unforeseen when they were taken, which could make the real robot's behaviour diverge from the requirements they addressed.

1.2 Autonomy

The term autonomous has a concrete meaning if we analyse its etymology: “having its own laws", from the Greek *autos* ‘self’ + *nomos* ‘law’. Thus an autonomous system is that which fixates its own laws. However, when applying the term to real systems several interpretations may arise:

- *A system is autonomous if it can fixate its own objectives.*
- *A system is autonomous if performs its function in absence of human intervention.*

These definitions separately do not capture well the concept of autonomy despite the feeling that both address a part of it. We may give an engineering definition for autonomy as:

> *The quality of a system of behaving independently while pursuing the objectives it was commanded to.*

There are still many open issues in the various fields of competence involved in the different technical processes that subserve complex system engineering. The core issue from the control engineering perspective, and which shall be considered transversal as it potentially affects many of the systems of tomorrow, could be summarised in:

> *design the control to make the system work alone.*

The search for autonomy has many reasons and implications but the concrete research target of this field is not clear at all as demonstrated by the fact that even the very term *autonomy* has many interpretations. But the search for autonomy is a major thrust in systems innovation. This is generally true for two main reasons: economical and technical.

Economical motivation is a major force because automated plants are less costly from an operational point of view (human personnel cost reduction, improved operating conditions implying less failures, *etc.*). But technical reasons are, in some cases, no less important: automated plants can be more productive, can operate fast processes beyond human control capabilities, can be made safer, more available, *etc.*

Levels of Autonomy

When confronting the challenge to build an autonomous system, engineers are not expected to build a system with full universal autonomy, that is, a system capable of achieving and/or maintaining any state of itself and the environment in the desired time without human intervention. That system would need unlimited resources and may not even be physically realisable. What is looked for is a system that would perform as autonomously as possible a certain task in a certain environment. This triad system-task-environment is what defines the problem of engineering an autonomous system [136].

The ALFUS working group [156] has proposed standardised metrics for autonomy according to three axis: mission complexity, environment difficulty and operator interaction or human interface (inversely proportional, the less the human intervention, the more the autonomy), as shown in figure 1.10. The autonomy level of a particular system is thus represented by the triangular surface determined by the values on the three axes.

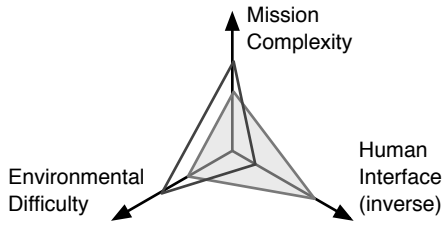


Figure 1.10: The three dimensions that determine the autonomy level for unmanned systems according to [70].

Example

Let us analyse how the ALFUS dimensions of autonomy apply to the previous exemplary case of an autonomous mobile robot, this time considering its full functionality of autonomous navigation:

mission – if the robot is capable of performing a security surveillance task we consider it as having a higher level of autonomy if it were only able to perform a point-to-point driving task.

environment – we consider the mobile robot more autonomous if it can navigate through a real-world dynamic office-like environment, where chairs may change position, there are passers-by... than if it can only move in a toy-like, fully controlled environment.

operator interaction – we could have a control system in the mobile robot so that the operator only needs to define an area for patrolling and then check every two hours if the robot is doing okay. Or it could be the case that the operator has to keep constantly an eye on the robot to approve the local navigation path selected for every corridor. In the later case we consider the robot to be less autonomous.

In this thesis we are focused on completely autonomous systems regarding operator interaction. Assuming this targeted ideal autonomy in that dimension, our work analyses and proposes an approach to maximise autonomy in the mission and the environment dimensions. The relevance of another dimension, the system itself, concretely its complexity, will be discussed later in section 1.4.

1.3 Intelligence for Autonomy

Control technology [81] has been addressing the problem of purveying systems with improved levels of autonomy, so as they can be operated by setting them reference target states instead of manually driving their actuators, and overcoming unforeseen perturbations/disturbances from their environment.

Control strategies have been successfully dealing with intensive uncertainty, from simple feedback controllers to complex robust control strategies, passing by state space and Kalman filter techniques. However, as discussed in page 9 they are still quite limited. These techniques rely on mathematical models of the plant that are often not available and in many cases not suitable enough. Knowledge about the environment is incomplete and generally not representable as a set of linear differential equations. The disturbance signal in such a mathematical model cannot account for that, and we could talk of a different kind of uncertainty: *qualitative uncertainty*.

> *qualitative uncertainty* refers to the occurrence of unexpected events that qualitatively change the behaviour of the plant. They cannot be accounted for in traditional control models as disturbances.

Qualitative uncertainty requires that the system interprets the unexpected situations and reacts to them appropriately, and it has to do so with incomplete and often innaccurate knowledge. This is what lies at the core of what we call *intelligent behaviour*.

Example

The PID controller we defined previously for our robot, based on maintaining a close distance to an hypothetic left wall, can make it navigate through corridor-like spaces, under very constrained conditions (no dynamic obstacles, no open doors at the left side...). However, this controller will not work in more open-ended environments, and does not allow for commanding the robot to go to a precise destination.

In addition, we demand that systems be capable of addressing higher level commands. That means that the control system must be able to manage information from the simplest level of plant sensed and actuated variables, up to a human level representation of the mission.

Example

If our robot were to be employed in a real surveillance application, operated by security staff, it would be most desirable to command it to "inspect sector A, C, D and E, avoiding passing by corridors 5, 6 and 7", so as not to interfere with some maintenance activities, rather than having to specify the complete route in a 2D map it shall follow.

Artificial Intelligence

From its earlier attempts, Artificial Intelligence has been concerned with *making intelligent machines*, in the words of John McCarthy, who coined the term [94]. The problem of designing systems capable of producing appropriate behaviour departing from incomplete, non-formal knowledge lies at its very core. Several approaches have been explored.

Soft computing

Several of these methods have proved useful in many domains in the resolution of specific problems, providing inexact but valuable solutions to computationally-hard tasks in computer science. They are included in the field of *soft computing*, and their advances have been nurturing the state of the art in software.

Symbolic approaches. Out there from the very beginning, *symbolic* approaches explored the representational nature of knowledge and intelligence as symbol manipulation. Several formalisms to encode knowledge have been developed: logic-based, semantic webs, rule-based for expert systems, or fuzzy logic. In relation and associated to them, different representations of actions and their results in the environment, such as graphs or octrees, have been explored in *planning*.

Connectionism. The connectionist approach, the main example being artificial neural networks, proposes a model of interconnected networks of simple and often uniform units processing information signals. Instead of using *a priori* models of the environment, biased by the engineers' beliefs of how it should be, empirical data is fed to the system to tune its configuration —i.e. learn— to produce the appropriate responses. Learning methods are thus the central issue.

Back to cognition

Since the 80s the seek for creating intelligent autonomous behaviour has been put back in the agenda, having been left to a side for the development of techniques for specific problems. The coupling of the agent —i.e. the controller— with the world in a loop through sensing and acting has moved the accent from information-processing intelligence to *cognitive* behaviour.

Cognitive architectures represent a systemic approach to intelligence and cognition. They try to define the basic processes and structures that provide for intelligent behaviour. Soar is a paradigmatic example [82]. However, in many cases they do not try to improve control technology but to model human intelligence for its study, such is the case of ACT-R [7].

Intelligent Control is the branch of control engineering, in the convergence point with AI, that has been applying all these advanced techniques to the construction of control systems. Expert systems and fuzzy controllers have been successfully applied in process plants. Bayesian methods, such as the Kalman and particle filters are an standard in controller for mobile robots, and neural networks.

Current control systems managing our high-tech technical infrastructures are complex systems integrating several of this technologies in a variety of heterogeneous software-intensive components.

1.4 Present Challenges to Autonomous Control Systems

In this quest for autonomy we demand more functionality and more robustness. Today automobiles do not provide us just with mechanical locomotion, now they also help us drive to avoid collisions, park, and even entertain the passengers with multimedia systems. Modern thermostats do not only control the temperature of the room, they are also able to learn our preferences through the day and the yearly seasons.

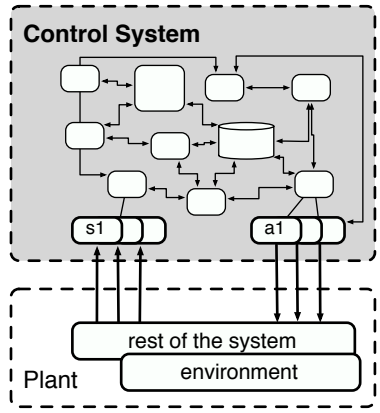


Figure 1.11: Control systems are now software-intensive, implemented in a variety of interconnected components.

This increased demand for more functionality has conveyed an increase in the control systems' complexity, in order to address it. Controllers now include more elements, with sophisticated functionality (for example, simulators, fault tolerance mechanisms, real-time problem solving), interacting in more complex ways. Incorporation of artificial intelligence into the control engineer's toolbox has also contributed to produce new waves and ripples of complexity, making controllers to be software-intensive systems where a great variety of heterogeneous components interact.Today's complex process control architecture is a mixture of all these things, running on technologically diverse computing platforms, with different implementation techniques and methodologies.

> The larger the number of components, the more things there are that could be faulty. (...) Similarly, the more complex a component, the more chance there are of it being faulty. [71]

This means more uncertainty, now on the control system itself, compromising the assumptions engineers do at design time. Unexpected undesirable interactions between the system's components threaten the runtime achievement of requirements. However, our dependency on these new complex technical systems is rising the requirements on availability, robustness and fault-tolerance.

1.4.1 Robust autonomy: focus on non-functional requirements

Dependability considerations have always been a matter of worries for real-world engineers. They encompass some of the non-functional[1] requirements that refer to the system's operation at runtime:

reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time,

availability can be simply defined as the proportion of time a system is in a functioning condition,

safety refers to the personal harm and equipment damage that can arise due to a system failure,

and others that refer to the evolution of the system:

maintainability refers to the ease with which the system may undergo repairs and changes during its life-time evolution,

scalability is the ability of a system to be enlarged, i.e. by incorporating new components, to accommodate growth either quantitatively or qualitatively in performance or in the functionality serviced.

Today, given the pervasiveness of technology and control systems in the very infrastructure of our world —telecoms, vetronics, energy production plants, distribution networks, etc.—, dependability has evolved from a necessary issue just in a handful of safety-critical systems to become an urgent priority.

Survivability [45] emerges as a critical property of autonomous systems. It is the aspect of system dependability that focuses on preserving system core services, even when systems are faulty or compromised. A key observation in survivability engineering —or in dependability in general— is that no amount of technology —clean process, replication, security, etc.— can guarantee that systems will survive (not fail, not be penetrated, not be compromised). This is so because the introduction of new assets into the system, while solving some problems, will add new failure modes both intrinsic to the new assets or emergent from the integration.

Of special relevance in the case of complex autonomous control systems is the issue of system-wide emerging dysfunctions, where the root cause of lack of dependability is not a design or run-time fault, but the very behavior of the collection of interacting subsystems. In this case we can even wonder to what degree an engineering-phase approach can provide any amount of increased survivability or we should revert

[1] Functional requirements refer to the services we demand of the engineered system; non-functional requirements are the desiderata about *how* the services will be provided, i.e. how well, how reliably, etc...

to the implementation of on-line survivability design patterns than could cope in operation time with the emerging dysfunctions.

1.4.2 Run-time adaptation

The augmented complexity of control systems is accompanied by increased risks of malfunction, compromise, and cascaded failures. Systems do not only fail due to their defects or their mismatches with reality, but also because their integration with other systems that fail, or that may cause emergent systemic failures.

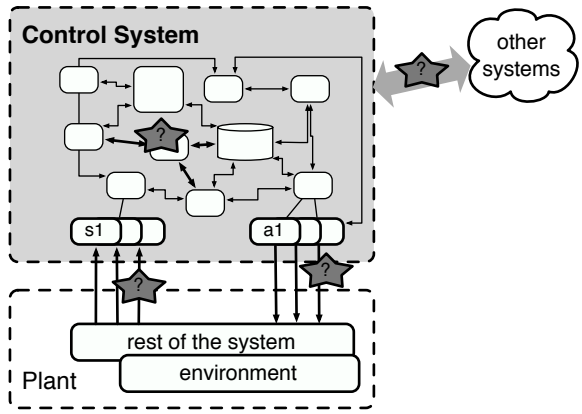


Figure 1.12: The complexity of control system increases the uncertainty levels in its internals and its interface to the environment. This is also aggravated because of the connections with other systems.

This situation results in a new functionality needed in control systems: they must take care of themselves.

We have already discussed how current control engineering provides run time adaptation for a limited environmental uncertainty, to address functional requirements, by converting them into the references for feedback loops. AI and intelligent control has helped engineers extend the capacity of control systems to adapt also to qualitative uncertainty at run-time, to some extent.

However, system's traits such as performance and dependability issues, linked to non-functional requirements, are only accounted for during development, embedded in the system's design. They are fixed at engineering time, and the system has no capability to adapt if unforeseen circumstances affect them.

> *More robust autonomous systems are needed. For that we need control systems capable of adapting at run-time to unexpected situations, either at the plant, the mission and the system (itself).*

This run-time adaptivity involves a series of self-x properties [64, p. 64]: self-monitoring, self-reflection, self-repairing, self-maintenance, self-reconfiguration. If intelligent capabilities provided by AI techniques have endowed artificial systems with the capacity to deal with qualitative uncertainty at both the mission and environment levels, metacognitive properties seem a reasonable direction to explore to cope with qualitative uncertainty at the self level and achieve any self-x property.

1.5 Structure of the Dissertation

The complex fabric of this thesis, touching a manifold of themes, together with the intricate and non-linear development process followed, makes it really difficult to find a way to present the research realised in a comprehensive, rigorous and complete written dissertation. Following we provide a guide to the document. The discourse is organised around the three phases of the methodological path followed —i.e. *analysis* of the problem and extant approaches, *development* of the solution and *validation*— and the basic elements produced as results.

Part I is the introduction, and presents the problem this thesis confronts:

- **Chapter 1** has described the context of technological demand for more autonomous control systems and engineering methods for them.
- **Chapter 2** introduces the departing motivation and ideas that have driven this work, detailing the pursued objectives to finally outline here how the resulting thesis is structured and reported in this document.

Part II is dedicated to the necessary foundation and background, together with a framing formalisation for the issues at hand:

- **Chapter 3** presents the scientific and engineering themes that are relevant for this thesis' work.
- **Chapter 4** develops a unified conceptual framework for autonomous systems, providing elements also to analyse their cognitive operation. This theoretical basement will serve two purposes: i) a foundation for this thesis developments, ii) an analysis tool to evaluate and compare other approaches to the problem as well as relevant state of the art work.
- **Chapter 5** offers an overview of the state of the art of extant approaches to build systems with similar features to those pursued in this thesis.

Part III presents the core contribution of this thesis, that is an architectural framework for self-aware autonomous systems:

- **Chapter 6** is dedicated the central ideas of this thesis, a set of postulates that contain autonomous systems design ideas.
- **Chapter 7** describes the design patterns developed that reify the thesis postulates.
- **Chapter 8** details the metamodel of components and functions developed to account for the ontology and teleology of autonomous systems.
- **Chapter 9** describes the reference architecture for self-aware autonomous systems that has been developed.

Part IV develops the validation of the developed framework for self-aware control systems:

Chapter 10 presents the engineering methodology developed to accompany the architecture framework for its application.

Chapter 11 describes the application of the framework to the development of the testbed application, and the results obtained.

Chapter 12 analyses the results reached in the research, and provides concluding remarks. Future lines of research are also discussed.

Part V includes reference material:

Bibliography that contains the list of references used in this dissertation and considered in the research.

Glossary of the concepts of the theoretical framework for autonomous systems and the specialised terms employed in this dissertation.

1.5.1 Notation

The discourse in this dissertation incorporates many terms common in engineering, but that are used here in a specific sense within the realm of autonomous systems. In addition, we have created new terms to refer to particular phenomena that was needed to dintinguish. All this wording refers to concepts and ideas from different levels of abstraction and prespectives, i.e. design, run-time operation, modelling, etc. To facilitate the reading, different font styles have been used to ease disambiguating the context of a term when it appears in the text:

- ***concepts in the theoretical framework*** for cognitive autonomous systems,
- `elements in the metamodel` for function and components structure,
- elements of the architecture for metacontrol of autonomous systems.

A consistent notation has also been applied to the figures used to illustrate the control ideas and designs developed. To represent conceptual elements square boxes have been used, whereas implementation elements have been depicted with round boxes. Dotted arrow lines have been used for informational flows, and unbroken arrow lines for actual connectivity between elements (see figure 1.13).

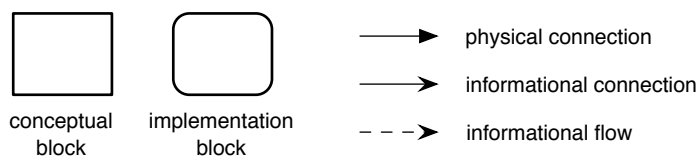


Figure 1.13: Conventions used in the figures of the thesis.

1.5.2 Examples

To aid in the explanation of the abstract ideas and generalisations presented in this dissertation, example scenarios are provided all along the text. For the sake of completeness and consistency, all of them will refer to the same exemplary system, which is a mobile robot.

The following text style in a gray box is used for all examples:

Example

Exemplary system: mobile robot

The exemplary system is a mobile robot with on-board computational resources and sensors, and a remote control application running in a desktop for supervision and operation. The robot consists of a four wheeled differential platform equipped with odometry and a range laser of 180°. The robot will move in an indoor **environment**, made of walls, doors, corridors and pieces of furniture like tables and chairs. The **mission** of the system is to go to the destination commanded from the supervision remote application, given by a point in the 2D Cartesian space. Upon initialisation the robot is provided with an occupancy map of the environment, as well as its initial position.

Although a variety of controllers have been used to illustrate control issues in this chapter, for the majority of the examples in this dissertation the same control architecture has been considered. This architecture consists of interconnected modules with different responsibilities. The localisation module uses the laser scans, the odometric information and the map to estimate the position of the robot with a Monte-Carlo method. The navigation module maintains an occupancy-grid map using the laser scans and the estimated position, plans the path to follow to reach the targeted destination, and computes the robot velocities using a path-following control algorithm that considers the robot inverse kinematics.

This control architecture is a simplified version of that of the testbed system that has been developed in the research. The testbed is presented in the next chapter, and fully detailed in chapter 11.

Example

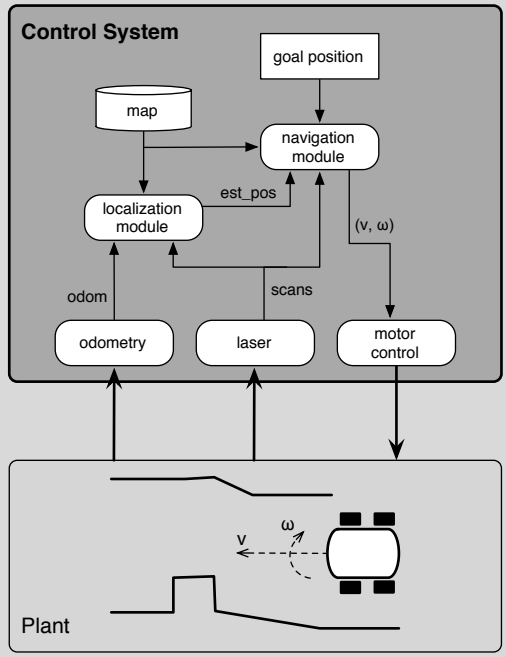


Figure 1.14: Schema of the control architecture in the exemplary mobile robot application.

Chapter 2

Approach and Objectives

Once the context of this work has been described in the previous chapter, now the concrete problem it addresses is discussed. The opening section of this chapter presents the engineering approach to autonomy in which the research is framed, the ASys project. Then the discussion about the scope of this work motivates the need for self-aware capabilities in control systems, in order to improve the robust autonomy of technical systems. Finally the goal of this thesis, which is the analysis and design of an engineering solution for this self-aware adaptive capabilities, is formulated into a concrete list of objectives.

2.1 Engineering Autonomy: the ASys Project

This work is situated in the research line of the Autonomous System Laboratory (ASLab) of the Universidad Politécnica de Madrid, which focuses on the development of universal control technology for autonomy. The thesis builds on top of previous research efforts in the group, from early attempts to apply intelligent control techniques for autonomy [128], to recent theoretical achievements in the conceptualisation the domain of autonomy and perception [90], or developments that formalise it into usable software assets, such as an ontology for the engineering of autonomous systems [18]. All these works, including the present, are framed in ASLab's ongoing long-term research project: *ASys* [139].

2.1.1 ASys Vision

The ASys[1] Project intends the creation of science and technology for *universal autonomy*. In this context, "universal" means that the technology shall be effective in augmenting the level of autonomy of *any kind of artifact in any kind of mission* [137].

[1] **A**utonomous **Sys**tems

This implies a wide range of (autonomous) systems to be considered in the research, from robot-based applications to industrial plants for continuous processes. Additionally, the ASys science may serve as a systems-centric substrate for theories of natural autonomy [36].

As described in the previous chapter, dealing with runtime uncertainty is the principal challenge in the pursuit of robust autonomy. Tools from the AI field have been extensively explored for intelligent control in the past few decades, improving the adaptivity and efficiency of systems. However, as it has been pointed out, they are usually valuable only for a restricted type of problems. Besides, a not negligible amount of tuning for their application to the specific system is usually required. This means that: i) an important part of the adaptivity is still determined at the design stage, and ii) the achieved solution is only partially reusable for other systems, and hardly transferable to other domains.

The underlying ideas of the ASys approach are: i) to overcome the previous issues by moving the responsibility for adaptation from design time to system's run time, and ii) provide the solution in an architectural form, rather than an algorithmic one, so as to meet the non-functional requirements presented on page 17.

2.1.2 Model-based Autonomy

The strategy proposed to address adaptation at run-time is that of exploiting cognitive control loops, which are control loops based on knowledge. This knowledge is realised in the form of different models: of the system, of the environment, and of its mission. This is the so called model-based view on cognition.

The pervasive model-based approach in ASys is two-folded: an ASys system will be built using models of it (see figure 2.1), but an ASys system will also exploit models of itself to drive its operation or behaviour. Model-based engineering and model-based behaviour then merge into a single driving concept: **model-based autonomy**. The main goal: the models the system will use to control its behaviour will eventually be those very same models used by engineers to build it.

Models constitute thus the core for our autonomous systems research programme. The type and use of models to be specifically developed for the autonomous system are considered. User and designer requirements, and constraints imposed by the system itself will guide the development of the models. ASys is deeply rooted in the Model-Driven Engineering approach, making models the cornerstone for the development of the autonomous systems, targeting the ultimate goal of automatic construction of the system from this models. The next stage is to extract from the built models a particular view of interest for the autonomous system. Unified functional and structural views are considered critical for our research as they provide knowledge about both the intended and the expectable behaviours of the system.

The obtained models will be exploited by means of commercial application engines or customised model execution modules. Models will be exercise during runtime to derive the most suitable actions to do. Note that models are the knowledge that

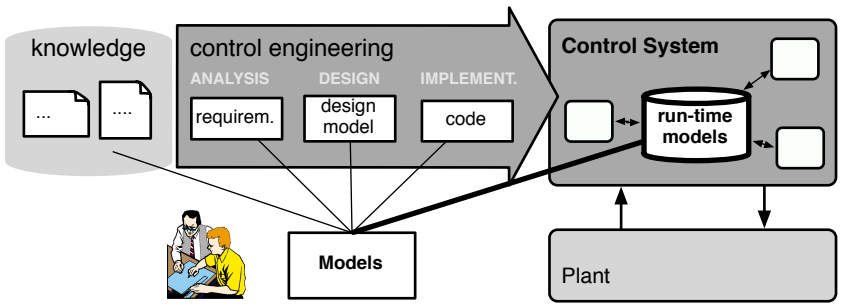


Figure 2.1: ASys model-based approach: the knowledge used to build the system is converted into engineering models during its development. The autonomous system also uses models for its operation.

the autonomous agent will use to act properly. This knowledge shall include knowledge about the cognitive processes of the agent itself. Considering the metacognitive needs of autonomous systems, metamodels are addressed in the research. Progressive domain focalisation will be used to address the different levels of abstraction between metamodels and models, following the ontological approach to metamodelling [11].

2.1.3 Architectural Approach

Focusing on the system's architecture is focusing on the structural properties that constitute the more pervasive and stable properties of the system [144]. Architectural aspects are what critically determine the final capabilities of any information processing technology, such as control systems.

ASys' seek for an architectural solution derives from its intention to address all the domain of autonomy, and therefore for the developed technology to be of general applicability, so it be usable to develop control systems for any plant, e.g. continuous process plants, air-traffic control or mobile robots; and to meet the critical demands for dependability and survivavility exposed on page 17, because they critically depend on the system's architecture.

In addition, architecture-based development offers the following advantages:

- Systems can be built in a rapid, cost-effective manner by importing (or generating) externally developed components.
- It is possible to predict global qualities of the final system by analysing the architecture.
- The development of product lines sharing the same architectural design is easier and cheaper.
- Restrictions on design variability make the design process more productive and less prone to faults.

ASys is thus interested in domain variations of architectural designs for their application to different autonomous systems. A reference architecture is an architecture where the structures and respective elements and relations provide templates to derive concrete architectures in a particular domain or in a family of software systems, and can be expressed in a reference architecture model [1, p. 9].

The ASys approach is strongly conceptual and architecture-centric. The suitability of extant control and cognitive control architectures and how they match the ASys research ideas and developed products (ontologies, models, views, engines) shall be determined and possible adaptations and extensions shall also be considered. The definition and the consolidation of architectural patterns that capture ways of organising components in functional subsystems is critical. As a generalisation strategy, the different elements considered in the ASys research programme will be assessed in different testbed systems: mobile robots, a chemical reactor, etc.

2.1.4 An integrated approach to engineering autonomy

The strategy to follow to materialise the ASys vision is the construction of a systems engineering framework [73] that can support the engineering processes associated with the construction of autonomous systems. Figure 2.2 summarily depicts the entities, tooling and activities involved in this process. It integrates the previously discussed ideas: an architecture-centric design approach, a methodology to engineer autonomous systems based on models, and an asset base of modular elements to fill in the roles specified in the architectural patterns.

The ASys engineering process covers from the initial capture of application requirements and domain knowledge, so as to define the initial system specification, to the implementation of the final product —i.e. the autonomous system.

The ASys vision suggests that the self-x competences [132] that are necessary to increase system autonomy and resilience, can be based on the models used in the engineering of the system. As depicted in figure 2.2, the autonomous system *Model* is transformed, by synthesis, in 1) the *Autonomous System* and 2) the run-time *Model* of itself that it uses during its operation.

The central part of figure 2.2 shows this process where the Model is transformed into the Autonomous System. The Integrated Control Environment (ICe) toolset shall be used to exploit an asset base to sustain this engineering process:

The asset base contains :

ASys Ontology: An ontology of the domain of general autonomous systems. This serves as the core ontological substrate for all ASys elements.

Domain Ontologies: Domain ontologies focused in the different application domains of ASys. The two domains that are the current focus of the work are process control systems and mobile robotics —the domain of this thesis.

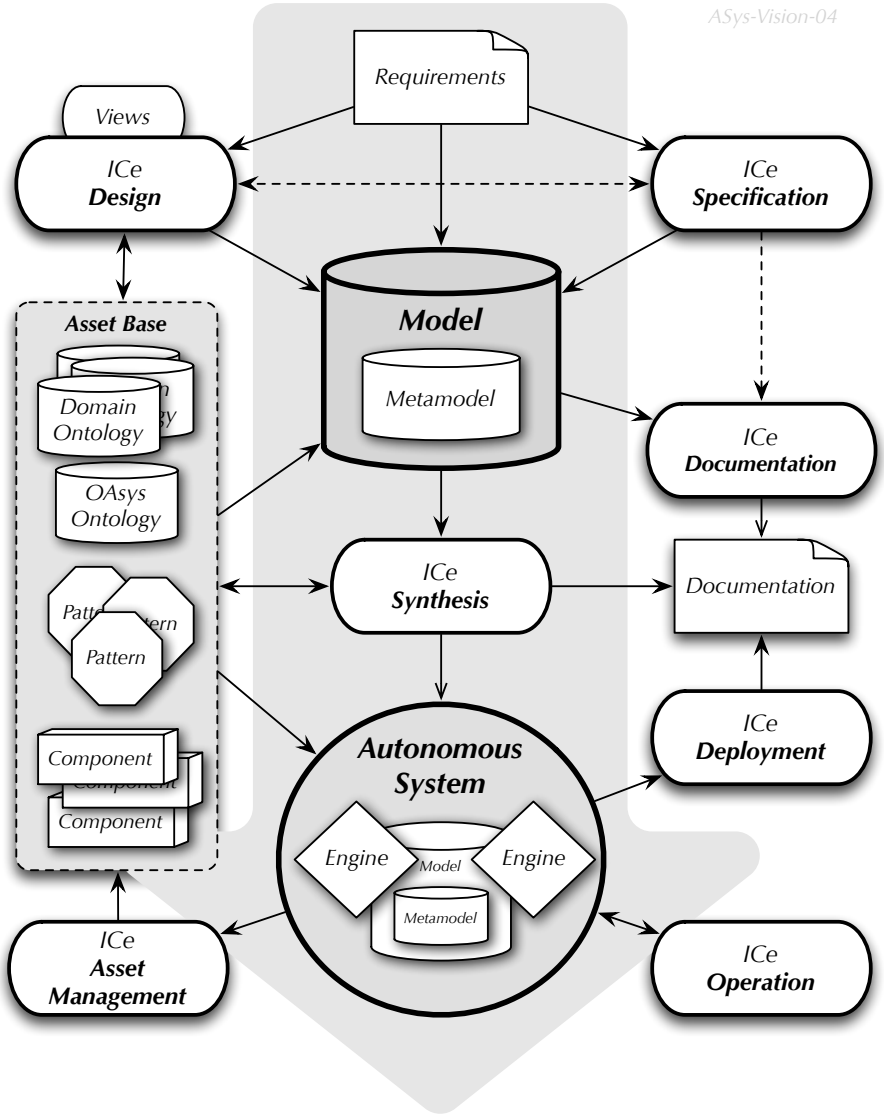


Figure 2.2: Tasks and products in the overall ASys vision (adapted from [139]).

Design Patterns: Reusable designs of subsystems of the autonomous agent. A big part of this thesis deals with some of the core patterns for ASys.

Components: Reusable software implementations that can be used to play the roles specified by the patterns.

The ICe tool supports several activities:

Specification: The specification of the ASys concerns the intended use and autonomy properties for the agent.

Design: The design specifies the architectural organization of the agent using the ASys pattern language as design vocabulary.

Synthesis: The synthesis produces both the agent implementation —obtaining components for the pattern roles— and the run-time model —by means of model transformation methods.

Documentation: The capture of all reference material that may remain somewhat hidden due to the automated nature of some of the processes.

Deployment: Deployment of the ASys and run-time models in real operational conditions.

Operation: Operation of the ASys through an RCP-based user interface.

Asset Management: Management of the asset base including incorporation of new assets specifically developed for concrete applications.

2.2 Scope of this thesis

The motivation for this PhD work is the search for a technology capable of improving current levels of autonomy in technical systems, addressing the demands on robustness and dependability as presented in 1.4, by leveraging their capacity for a special cognitive capability: functional self-awareness.

It addresses the ASys' objective to achieve, eventually, any level of autonomy by letting the control system handle itself [133], moving the responsibility of adaptation from engineers at design time to the control system itself at run time. This would eventually allow systems to cope with all kinds of uncertainty either at the environment, mission, or the system itself levels.

2.2.1 Self-awareness for Run-time Adaptivity

For this runtime adaptation we have explored the possibility of providing control systems with the key element that control engineers use to design the required system's behaviour: *knowledge*; and the capability to exploit it, as engineers do. Knowledge about the requirements (the mission), knowledge of the system itself, and the mechanisms to understand their relationship, as reified during design. To use it, the system

must be able to relate it to the run-time circumstances. This implies that the system must be able to observe its state in relation to its functionality, and from that also be capable of reconfiguring its structure on the fly if needed.

It is in the point of observability where self-awareness comes into play. The control system must be capable of deriving appropriate control actions (reconfiguration) from the observed state. So it is not just an issue of monitoring certain internal variables, the system needs an *understanding* of the functional implications of the observed situation. It has to analyse it in the view of the required behaviour, as stated in its mission.

There is a cognitive capability in some biological systems closely related to these issues of understanding, meaning and self-representation: consciousness or self-awareness. Recent research on this phenomena is showing the evolutionary value of self-awareness, which could be related with the leverage of other cognitive traits. This has been the motivation to explore the idea of introducing in the control systems self-aware mechanisms that exploit the models for enhanced run-time adaptivity.

Considering the context of ASys, this work has tried to advance in the realisation of the model-based cognition view, with special focus on extending it to integrate the self-awareness phenomena.

2.2.2 Architecture for Self-aware Control Systems

Exploiting knowledge about the mission and the system itself is not new. From the very beginning of AI the issue of how to represent the system's mission as *goals* has been a main research theme. Artificial agents [72] are usually designed to maintain representations of themselves and their capabilities. However, these are typically black-box representations: the agent knows nothing of the design intricacies of how its structure renders those capabilities or functions that realise its goals. The representations are used in open-loop, added to the agents' behaviour generation mechanisms, without a dedicated feedback control mechanism with a reference.

There are other control approaches, such as fault-tolerant control, that dedicate architectural mechanisms to use knowledge of the system, but it is usually limited to faults, and tending to ad-hoc algorithms and predefined configurations.

Adhering to ASys focus on architecture, this thesis aimed to explore the incorporation of self-awareness into the control architecture. That is, have a control loop, based on engineering knowledge of the control system, controlling the control system itself. This is what we have defined as *metacontrol*. In summary:

> *this thesis has intended to advance in the formulation and technical reification of ASys principles of model-based cognition and having systems self-handle at run time for robust autonomy. For that it has focused on the biologically inspired capability of self-awareness, and explored the possibilities to embed it into the very architecture of control systems.*

Example

Suppose we have developed a control system for our mobile robot that allows for autonomous navigation in an office-like environment, so we can command it to go to a certain destination. Such a controller typically consists of several components: I/O drivers, localization module, obstacle mapper, planner, local navigator, operator interface... adequately configured for the application, i.e. different cost functions for mapping obstacles are advisable depending on the cluttering of the environment.

During the system's operation many things can go wrong. Let us imagine two scenarios: that make the robot fail:

Scenario 1: the range sensor stops working due to an internal failure.

Scenario 2: the localization algorithm does not converge to a solution.

In both cases the result is a deviation from the desired navigation behaviour. The self-localisation of the robot fails, that typically resulting in the planning algorithm to execute a fail-safe action, such as stopping or turning around in the same spot to obtain new sensory readings if possible.

To robustly preserve autonomy, adequate response targeting the source of the problematic situation is needed. Fault-tolerant techniques could address scenario 1 to some extent. However, the more general solution, and the one is actually very commonly employed, resigns autonomy. It consist of engineers[2] take care of the problem at runtime:

Firstly by diagnosing the situation, i.e. identify the failure and evaluate its impact on the functions of the system, which are, in each scenario:

Scenario 1: laser internal failure, which compromises the functioning of localization and navigation.

Scenario 2: localization algorithm not achieving a solution as expected, so no estimation of the position is generate, which prevents global navigation.

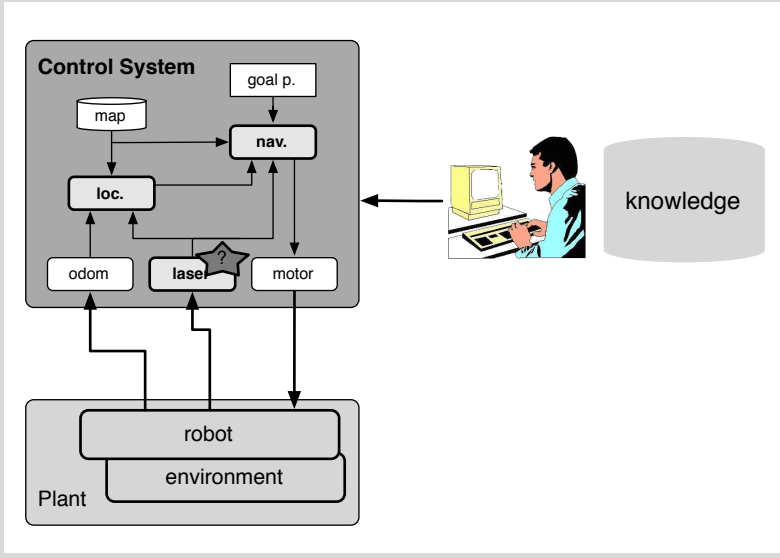


Figure 2.3: To recover from unexpected situations, sometimes engineers have to re-configure the system on the fly.

Then they would take appropriate reconfiguration actions, for example:

Scenario 1: halt movement, and reconfigure the localization and navigation to use other available sensory source if possible, such as infrared sensors.

Scenario 2: reconfigure the localization algorithm parameters, and slow down motor velocities.

In this thesis we have explored architectural design solutions so that these actions are not taken by engineers, but by the robot itself, being the result of a single architectural mechanism embedded in the control system of the robot, making it self-aware of its mission, itself and its environment, and eventually, the relationship between both at the very engineering level, which is the cornerstone here.

2.2.3 Dimensions of generality

This thesis intends *generality* in its developments. In this line, this thesis seeks the definition of patterns and the implementation of components to address system self-awareness needs of general applicability. As part of the ASys project, it stands in an universality track; we seek unified general solutions instead of collections of specific ones.

Take for example the case of achieving immunity against cyber-threats by self-x mechanisms. In a conventional approach, a classic virus scanner is a self-x mechanism that is automatically triggered to detect threats in the system. In a step towards

generality Musliner et al. [98] offer an implementation of a system that rather than scanning a computer all night to see if it has been compromised by an exploit, it will scan for vulnerable software and repair or shield it. It is not focused on detecting *specific* viruses but on identifying vulnerabilities that would be exploited *by any of such* viruses. The FUZZBUSTER system approaches generality by getting rid of the virus specificities and focusing on the inherent physical and operational structure of the attack (in strong relation with the structure of the "self" as system).

Generality means the system being not limited to one particular case. Generalisation can be thought-out as an inductive process, progressively addressing particulars using a fundamentally simple metamodel. This is what virus scanners do: collect information about hundreds of viruses using simple binary signature metamodel. However, true generality is attained when a general but deep description of the many cases —a deep model— is attained. Individuals match the features of the model, that are applicable to the whole as well as to every one of the individuals.

However, in the context of systems engineering, the term "generality" needs a conceptual clarification. This is needed because of the different aspects of generality that the problem and its solution may imply [100], especially in the case of software systems.

There are many directions to pursue, and generality can be sought in relation to:

- The different domain systems that are the final products of the ASys engineering process. A solution —a pattern, a component— is said to be general if it can be applied to different domain systems [62]. For example, a GPS-based localization component can be used in different mobile robots without any major change.
- The different subsystems in a specific domain system. A solution is said to be general when it can be applied transparently to different elements that form part of the domain system [148]. For example, a check-pointing pattern is said to be general because it can be applied to different subsystems; e.g. it can be applied both to to the mission constraints database and the localisation subsystem.
- The different platforms over which it can be deployed. A solution is said to be general when it can be realised in different implementation platforms straightforwardly [89]. For example, when a design pattern can easily be realised as a collection of distributed Java objects using RMI or as a DDS application.
- The different control levels of the ASys. A solution is said to be general when it can be applied to tackling similar problems at different layers of a control pyramid [109]. For example when a robust navigation algorithm can be used at the level of local object avoidance and global mission path planning.
- The different meta levels of the system. A solution is said to be general when it can be applied to a hierarchy of meta/domain pairs [112]. For example the same kind of model-based perception algorithms can be used to perceive the state of the world around the robot and the state of the components that constitute the robot controller.

The objective of generalisation is dual: a) on one side it contributes to a reduction of effort, as is the case of reusing software implementations of algorithms across control layers [78]; and b) on the other side it contributes to usability, quality and robustness in the sense of having designs that are cleaner, more understandable and applicable[40]. Object-oriented technology is specially suitable for this effort [117] and it is what has been used in this thesis in the form of a software framework [152].

2.3 Objectives

To condense the scope of this research, the question that it has tried to answer is:

> *How can we enhance control systems with self-aware capabilities so as to robustly improve their autonomy?*

Following the ASys' architectural approach, the theoretical results of this work should be developed in an architectural form. Concretely, a reference control architecture, providing an engineering guideline for the development of self-aware control systems for any application domain.

The aim of this thesis, despite the deep theoretical research it addresses, has been to be of practical engineering applicability. Software assets should be developed to reify the reference architecture. To guide their implementation and validate the approach, the architecture should be deployed in a real autonomous system application, simple enough to be achievable in a small-scale project, but requiring of robust autonomy in a context of challenging uncertainty.

All these intentions can be synthesized in the following list of specific objectives:

---

1. Analyse the functional value of the adaptive mechanisms related to biological self-awareness for its applicability in cognitive artificial systems. Extant technical mechanisms rendering a similar functionality will also be studied. *Analysis*
2. Explore the relation between self-awareness and models. Given a conceptualisation of cognition as the exploitation of explicit models to efficiently drive the behaviour of systems, self-awareness is to be explained under this theoretical framework.

---

3. Elaborate control design principles to guide the development of control systems with self-aware properties. These principles will be the result of the previous studies. *Development*
4. Develop an architecture for self-aware control systems from the previous principles. This architecture shall be of general applicability to the design of autonomous systems, independently of the domain, the particular application and the specific technical platform employed for its implementation.

---

5. Build reusable software assets for the application of the developed architecture to the building of control systems in different domains. *Validation*

6. Demonstrate the validity of the approach by developing a testbed application using the architecture and assets developed.

2.4 Research methodology

To address the previous objectives, a multidisciplinary work has been carried during the last 5 years. During that time, apparently disparate themes from different disciplines —control, cognitive science, general systems theory, software development, modelling, cognitive architectures, etc.— have been studied seeking for the appropriate conceptualisations and tools to tackle the proposed problem.

Given the multidisciplinary character in this work and the generality sought, while maintaining an ambition of immediate engineering applicability, a mixed methodology in between the scientific method and the engineering process has been used. Firstly, preliminary analysis phase was conducted to characterise the problem, identify the key issues and the research areas and technologies of potential relevance, studying extant approaches that tackle the problem, or variants of it. The results obtained from this phase were used at the inception of the solution proposed, which was developed from initial design principles into a complete architectural framework for the design of self-aware control systems. A concrete implementation of the architecture for a testbed application was finally realised to validate the approach and the work done.

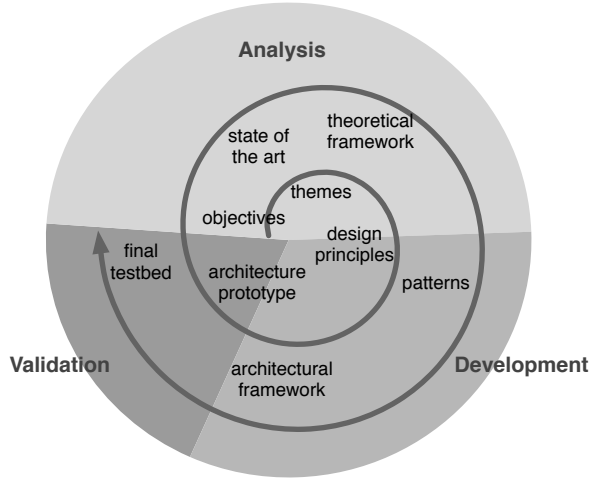


Figure 2.4: The phases of the iterative process followed in this thesis, and the milestones achieved.

Due to the novelty of the approach and the mixture of themes touched and aspects addressed (biological models, engineering methodologies, control theories…), there was no standard solid common ground to depart from, so, to guarantee a sound progress, the development process of this work was iterative, somehow following the

spiral model from software development [23]. Figure 2.4 shows the three discussed phases and the basic elements addressed and results.

2.4.1 Mobile Robot Testbed

A testbed system has been used to guide the work. The deeply theoretical and abstract nature of the basement of this thesis put the results of the work at risk of loosing connection with the current reality and practical applicability. The testbed has served to purvey a bottom-up perspective to help clarifying the issues at hand, and keep the deep theoretical developments deeply grounded in the needs of real systems.

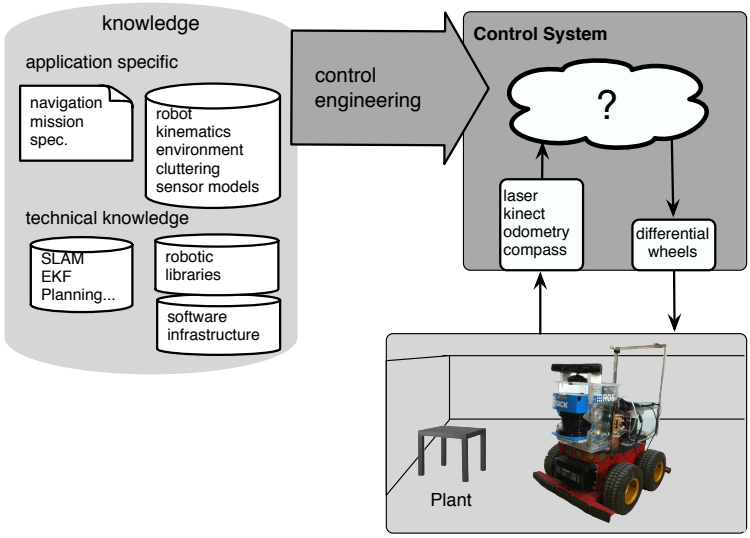


Figure 2.5: Relevant elements in the implementation view of the control for the testbed application. Well established control techniques for the robotic domain such as SLAM and Kalman filtering, as well as COTS software, such as driver libraries or middleware infrastructure, have been used, and so knowledge about them implicitly *and explicitly* used in the engineering of the control system.

To maintain the general validity of its outcomes, the selected testbed had to cover most of the issues addressed by this thesis. It should also be broadly representative for other systems, its nature and properties including as much a variety as possible so as to cover the widest range of possibilities.

For these reasons an autonomous mobile robot application was chosen. Mobile robot applications demand autonomous operation in open-ended and real-time environments, thus encompassing both quantitative and qualitative environment uncertainty. Their navigation-based missions require sophisticated functionality such as localisation, planning and obstacle avoidance, while reliability and safety are critical. Control systems for mobile robots are software-intensive, including heterogeneous components integrating different techniques and complex algorithms. Besides,

the system encompasses continuous and discrete variables, physical and informational components, etc.

The concrete testbed consists of developing the control system for patrolling an indoor office-like area. The system consists of a mobile robot platform equipped with range and odometry sensors and a depth camera, and on-board and remote computing resources. Figure 2.5 depicts the basic elements in the testbed application, which are described in detail in chapter 11.

2.4.2 Basic elements of this work

The result of the research done for this PhD thesis is a framework for the engineering of autonomous systems: the OM Architectural Framework. It encompasses theoretical conceptualisations, guidelines and principles, together with design and implementation assets. They have been validated by the development of the testbed control application.

Figure 2.6 sketches all these results into basic elements, showing their relations according to the development process followed. The theoretical framework contains the conceptualisations and vision of autonomous systems at the base of the research developed. The elements in bold have been completely developed in this research. The OM Architectural Framework, and its accompanying engineering process, are the reification of the results of the work done, demonstrated by their application to the development of the testbed mobile robotic system.

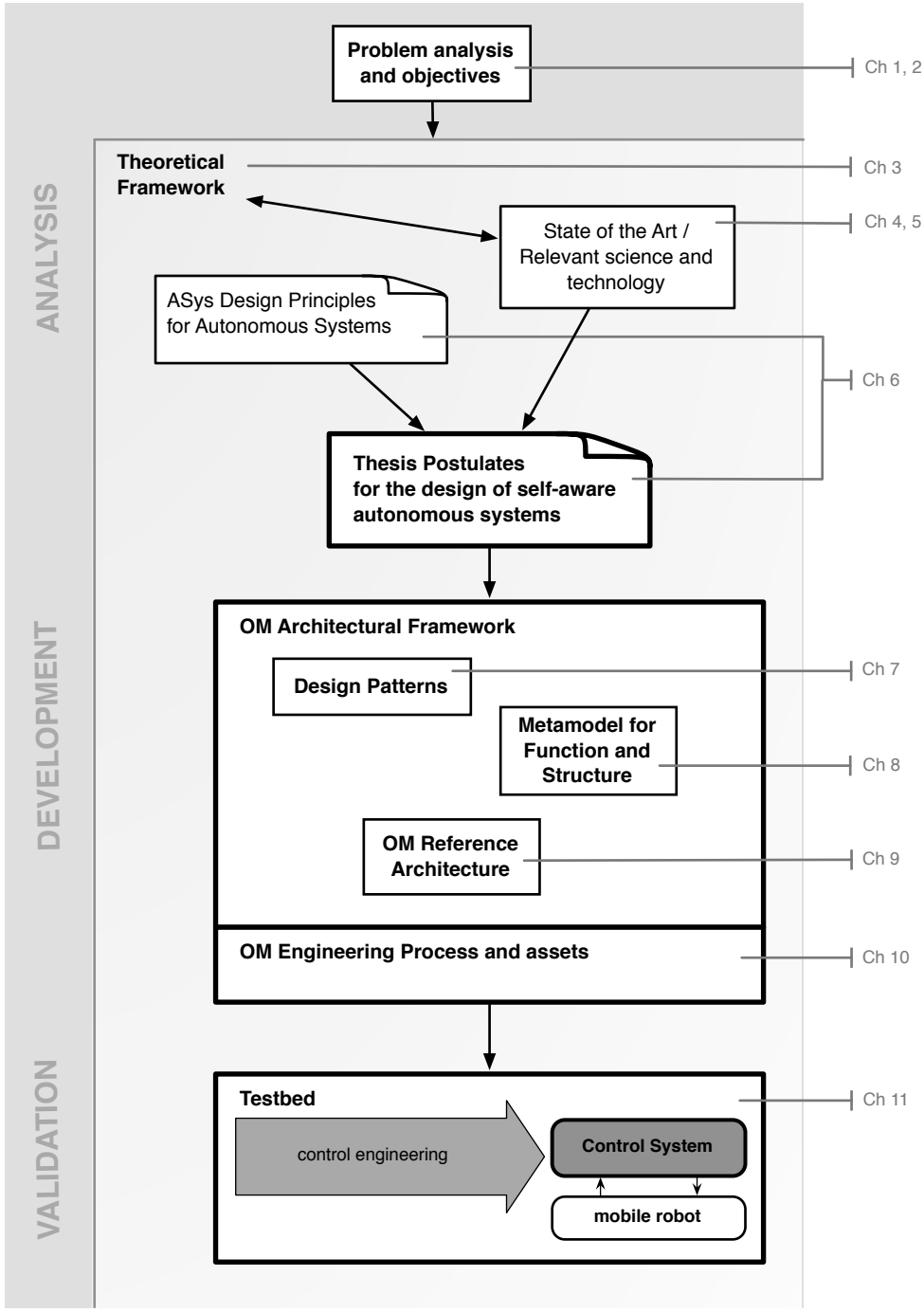


Figure 2.6: The basic elements this thesis is structured around, and the chapters that discuss them.

Part II

Foundations and State of the Art

Chapter 3

Core Themes

This chapter presents the core themes that have been addressed during the development of the research. They are from different disciplines, and may seem disconnected and far apart in the scientific domain. However they converge in our exploration of new engineering methods for autonomous systems. Biological self-awareness was the initial triggering motivation, given its claimed evolutionary value for adaptation. Modelling is a central and pervasive topic in our research, and it is discussed in section 3.3, devoted to functional modelling, and section 3.2, which treats software modelling and theoretical considerations about general modelling. Design patterns, to which section 3.4 is devoted, are a technique used to describe design solutions, which has been used to shape the one developed in this work. Fault-tolerance is an are of maximal relevance for our research regarding run-time adaptivity in technical systems. Finally, component platforms currently seems the more appropriate technology for realising complex control applications such as autonomous systems.

3.1 Biological Self-Awareness

Biological inspiration has usually guided research on autonomy and intelligence. The brain and the rest of the central nervous system is regarded as a very adaptable control system, providing intelligent behaviour. The study of the biological mind and its cognitive capabilities has thus inspired many useful AI techniques. Following this bio-inspiration we have looked at the most advanced cognitive trait in biological systems: consciousness or self-awareness, to investigate the possibilities that it may render in our quest in the artificial realm.

Our ability as human beings to think it over what we are doing, reflecting about our actions including the process of thought itself, goes beyond cognitive abilities that make use of internal representations of our environment in order to optimise behaviour. These are aspects involved in self-awareness, a phenomena associated to consciousness. It is an attribute we humans have reserved for millennia to those of our kind, as

a distinctive trait of our very nature, and that science has only started to demystify in the past century.

Recent research on consciousness[1] is showing its evolutionary value, which is probably shared with other mammals, at least our relatives the big apes. It is argued that it could provide for some core functionality enhancing the cognitive capabilities of these species [6] and resulting in improved adaptability [41]. This has led us to investigate on the possibility of developing a technology for control by exploiting similar traits.

3.1.1 The Conscious Phenomena

We can find a profusion of terms for referring to consciousness in different disciplines: vegetative state, coma, sleep or wakefulness in medicine, reportability, attention or voluntary control in psychology or phenomenology and qualia in philosophy are a few examples. Some refer to precise clinical states —i.e. medical terms—, others cognitive functions —in psychology—, and sometimes they are overused and become buzzwords —i.e. self, awareness—. A good part of the confusion around consciousness is precisely due to the relaxed use of these concepts, which pertain to different domains and levels of abstraction, with specific context-dependent meanings and connotations. Terms and concepts from different realms usually refer to similar parts of the problem of consciousness, but their mapping is usually less than perfect so their loose use results misleading. As Sloman puts it [146]:

> *"it (*consciousness*) is a cluster concept, in that it refers to a collection of loosely-related and ill defined phenomena."*

What follows is a reasonable list of the referred phenomena related to consciousness, synopsized from a cross-domain perspective:

Awareness of the world: It is usually explained as the access to some information that is used to control/generate behaviour [32]. A theoretical approach already formulated by Craik [39] considers that this information is actually an internalised model the agent has of the surrounding world.

Self(–awareness): As commonly understood, the problem of the self has two strands: one involving the differentiation one's own from the rest of the world with the related sense of agency, and the other one comprising the identity of oneself as a result of development, like the record of autobiographical memories which render personality in humans[41].

Introspection: Our ability to observe our own mental and emotional processes is one of the most puzzling aspects of consciousness, with many theories trying to account for mental states whose objects are other mental states while avoiding the *homunculus* trap. It is related to inner speech and imagery, and is also re-

[1] Consciousness and self-awareness will be used indistinctly for the moment; more detail on the different aspects of this phenomena will be given in the following

ferred to as reflection, which can be considered a special kind of access to some intellectual resources [22].

Attention: The term consciousness is often conflated with attention in the literature, thus promoting confusion [14]. However some authors neatly distinguish them while preserving their deep relation [150]: the psychological phenomenon of attention is generally regarded as the selective process responsible for deciding which contents in the mind become conscious. Other authors [147] consider that attention is also a selective process not for deciding which enters consciousness, but for focusing on the more relevant contents within those already conscious, the rest forming a background.

Voluntary control: This volitional aspect is closely coupled with the already referred sense of agency. There is a clear common sense distinction between involuntary actions, like kicking when hit in the knee or an spontaneous smile, and voluntary ones like rising an arm because of deciding so, the later "voluntary" control being equivalent to "conscious" control [14] in the line of James' *ideomotor theory* of controlling action as a result of bringing to consciousness the desired goal.

The reader familiar with the research on consciousness may miss phenomenological aspects from the previous list. We have left at a side the problem of phenomenal consciousness and qualia in this work, given that not only its value but its very nature is still a subject of intense and philosophical debate. However, some daring and interesting approaches have been done from the engineering realm [58, 9], including a recent interpretation in the context of this thesis [131].

3.1.2 Models of Biological Consciousness

There is a pervasive debate in the scientific study of consciousness regarding its nature. Some authors claim consciousness is a *property* of some mental contents. A living organism is thus considered conscious if its mental operation involves such contents. Other authors hypothesize that consciousness is a mental *process*. As it usually happens, there are models of consciousness covering the whole spectrum from process to property.

One of the more popular models of consciousness is Bernard Baars Global Workspace Theory (GWT), which considers a dual nature, consciousness being both a property of some mental contents and a process related to them. GWT is a theory of consciousness in the approach to cognitive psychology that considers a computational view to the mind, that is in terms of modules and processes that manipulate information. Its central idea is the Global Access Hypothesis [15]:

> *Consciousness enables global access to multiple brain capacities, which otherwise function separately.*

GW Theory can be explained using a theater metaphor: consciousness resembles a bright spot on the theater stage of working memory, directed by the spotlight of attention. The conscious processes are the actors competing for the spotlight, which

allows them to broadcast their content to the audience, the unconscious processes. Behind the scenes are contextual systems, such as intentions, expectations, which shape conscious contents.

Most models of consciousness distinguish different levels. For example Sommerhoff identifies three dimensions:

awareness of the surrounding world: from sensory input an internal representation or model of the surrounding is maintained,

awareness of the self: not only the body, e.g. posture and movement, but also a self-image including aspects of personality, ultimately related to self-awareness,

awareness of one's thoughts and feelings: this involves representations of the processes of the mind.

Baars and Damasio consider similar dimensions of the conscious phenomena.

In relation to the perspective of consciousness as a property of mental contents, the first scientific challenge is the definition of these mental contents. This is another philosophical debate in cognitive science [25]. In the cognitivist [154] stance this thesis takes, mental contents are *representations*, even if disputing about their nature —e.g. symbolic vs connectionism.

An interesting theory relating representations in the mind and consciousness is that of Sommerhoff [147], which considers mental representations as expectancies. Otherwise there would be symbols, and that leads to the symbol grounding problem [60]. This means that an object is represented in the mind in the form of the behaviour related to it, rather than a symbolic mapping of its sensed featured. The sensory input elicit or sustain these representations. Sommerhoff states that consciousness is precisely an Integral Global Representation (IGR), a functional unit that integrates representations of the fact that first-order representations of sensory inputs and stimuli are part of the state of the organism.

Regarding the view of consciousness as a process, some models suggest that consciousness provides an infrastructure of services to support high-level cognitive processes [67]. This is the case of the models of François Anceau [6] and Johnson-Laird [74], which pertain also to the computational and informational view of cognition.

Anceau hypothesizes that consciousness provides an environment or platform to support high-level cognitive processes. Its sequential character is its key property. According to him the function of consciousness seems to be but to sequentially trigger actions and thoughts, its sequentiality guaranteeing the temporal coherence of their operation.

Johnson-Laird's theory is that of the operating system metaphor. He proposes to consider consciousness as a central executive managing cognitive processes. This executive receives messages that represent the world from the processors in lower levels and would send messages to them to communicate its plans. That conscious process is, according to him, ontologically different from the unconscious processes:

> *The conscious process is the serial process of explicitly structured symbols, whereas the unconscious are parallel processing of distributed symbolic representations.*

[74]

3.1.3 Analysis of the functions of consciousness

Notwithstanding the previous discussion, the relations and couplings between the above aspects of consciousness, and the interdisciplinar use of the terms used to talk about them suggest that there are some common core principles underlying these phenomena. Being engineers in the pursue of methods for building more autonomous technology by applying useful principles underlying natural systems, and not committed to the mimicking of its material realisation in animals [129], we are interested in the functional concepts, rather than in the specific physical substrate of consciousness[2].

Therefore we shall now make a summary of the more relevant functions, according to our perspective, identified so far by the theories developed in the search for explanation of the previous aspects of consciousness. We will try to present them as a sound set of functional concepts as general as possible, by abstracting from the domain specific details, and as far as possible clearly differentiated, separating intermingled concepts, while preserving the terminology used in the literature.

Access: Many theories on consciousness assign it the role a blackboard has in the so named architectures in artificial intelligence, which is that of allowing the different processes running in the system—i.e. the mind—to put their content at disposal of the rest by means of a broadcasting mechanism. This is, for example, the main hypothesis underlying Baar's Global Workspace Theory [15].

Sequentiality: The serial and limited character of consciousness could be a mechanism for guaranteeing the *consistency* and unity in the mental contents [14]. The sequential character of conscious contents could also be involved in our sense of time, allowing for the temporal analysis of perceptions as Anceau [6] proposes.

Integration: An important function attributed to consciousness is that of integrating multiple sensory input into a single unified experience [16].

Meta-representation: another one of the more common ideas about what consciousness is (or how does it work) is that of structures in the mind/brain representing other structures in the mind/brain, that is second order representations [22]. That is Damasio's idea of conscious creatures constructing images of a part of themselves forming images of something else [41], Singer's meta-representations of the brain's own computational operations [145] or Sommerhoff's representations in the IGR [147].

[2]The physical substrate in the brain for consciousness is usually referred to as the neural correlate in the specialized literature.

Metareasoning: This feature of consciousness is strongly related with the previous one, and refers to the capacity of conscious brains to operate upon their own operations, for example monitor and reason about them or evaluate their performance, as Singer [145] proposes. Other authors separate high-level (meta-) cognitive processes, such as reasoning or long term memory from consciousness, but keep them related because, due to these processes being very resource demanding, only conscious contents have access to them [15]. Anceau goes further by proposing that the role consciousness is providing the underlying mechanisms —i.e. the previously enumerated functions— that subserve the functioning of those high level processes [6].

Evaluation: Some authors relate consciousness to value-assigning systems in the brain, e.g. the mentioned view of Singer associating it to a certain monitoring at a metalevel which provides the brain with the capability of comparing the performance of its operations [145], or the evaluation of plans by affective states, bringing up the close interconnection between emotion and consciousness [4].

Learning: while the learning process is itself unconscious, there is strong evidence for learning of conscious events and no robust one so far for long-term learning of unconscious input [16]. There seems to be a relation between events or entities entering consciousness and our capacity to learn them [15].

The previous list is not exhaustive, but it captures the functional hypothesis about consciousness put forward in the literature that we judge more relevant for our approach to self-awareness.

As a line of research in the pursue of building more autonomous and robust systems, machine consciousness must not be over-restricted by its biological counterpart, we do not want the self-aware control system of a chemical plant not allowing for the production of more than a product at a time because of its limited attentional capacity, for example. We propose that, despite providing a plausible explanation of human consciousness in terms of mechanisms evolved for synchronization and adaptation to a causal world [6], the sequential character of consciousness is not necessary for self-awareness in artificial systems. Other techniques for assuring consistency and cohesion can be used, together with alternative mechanisms for dealing with time, while maintaining a distributed and parallel processing paradigm.

3.2 Models

Models are central to this research, and they are at the core of the ASys vision of cognition. Models are important as they are the way that everybody (humans and systems) knows how to interact with the real world. A model is a construct to help in a better understanding of real world systems.

Rosen [121] provides a definition of model in terms of a modelling relation that fits the ASys perspective:

> A system A is in a modelling relation with another system B —i.e. is a model of it— if the entailments in model A can be mapped to entailments in model B.

In the case of cognitive systems, model A will be abstract and stored in the mind or the body of the cognitive agent and system B will be part of its surrounding reality [134].

Models may vary widely in terms of purpose, detail, completitude, implementation, etc. A model will represent only those object traits that are relevant for the purpose of the model and this representation may be not only not explicit, but fully fused with the model exploitation mechanism.

The focus of this work is the engineer of control systems for autonomous applications, which are software intensive. Therefore, we are specially interested in models in software. This is not just software models, but the use of models in the life-cycle of software systems. This section focuses on models from the software perspective.

3.2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software development methodology centered around the exploitation of models rather than algorithmic or computational issues. It is not that these later aspects are neglected, but rather than the driving mechanism in the progressive design of the solution by refinement are domain models: abstract representations of the knowledge and activities that govern the domain of the application.

MDE tries to improve productivity by maximising re-usability of models and simplifying the design process with the use of standard domain solutions captured in design patterns, also modelled.

Simplifying a bit, development consists of three phases:

1. Analysis: all aspects of the problem, that is from the client perspective, are captured in analysis models.
2. Design: produces an architectural design specification of the system, in the form of design models of the solution at different levels of detail.
3. Implementation: from the design models the final system is implemented.

According to MDE, models must be formally connected, so that it is possible to trace the refinement of every design element, down to its implementation and up to the more abstract analysis model and related requirements. Portions of the solution at any level can be thus re-used for other applications. moreover, the ultimate objective is to automate this model refinement through automatic transformations. Automatic code generation in C++ or Java from UML models is an example of this.

Model re-use through transformation, serialisation and exchange is key. For all this to be realisable, meta-modeling is required: the models must be formal, that is conform to a certain set of rules, defined in metamodels [11].

Model-Driven Architecture

MDE has gained popularity thanks to the OMG's effort to build a foundation for it, called Model-Driven Architecture (MDA). It is a software design approach that focuses on architecture, so that models incorporate progressively more detail in terms of the implementing platform. The platform can be the component model middleware, such as CORBA or DCOM, the programming language such as C++ or Java, or the libraries or frameworks used in the implementation.

The development process departs from a computation-independent model (CIM), which models the knowledge about the engineering problem, including a domain model and requirements. From here, the designer defines the system functionality in abstract terms, developing the Platform-independent Model (PIM), which describes the engineering solution abstracting from any platform the system may run on. To conclude, the Platform-specific Model (PSM) is produced by extending the PIM with information of the specific platforms used to implement the system. Eventually all this information is captured in models and incorporated through automatic transformations from the PIM to the final PSM.

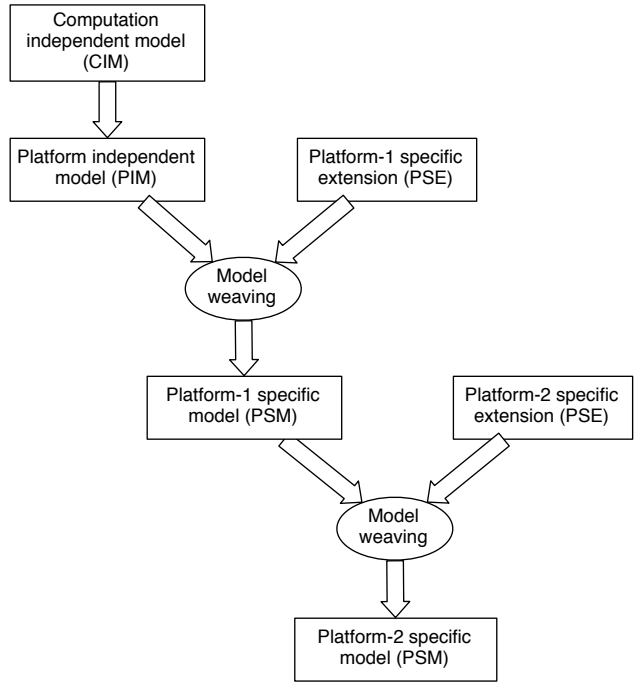


Figure 3.1: Model weaving process in MDA to go from PIM to PSM, adapted from [11].

MDA tools: the OMG has produces a set of standards and specifications, rather than implemented solutions, to support the MDA approach:

UML – Unified Modelling Language: the well known modelling language for software systems. According to OMG meta-modelling, UML is the metamodel for all models written in the UML language.

MOF – Meta-Object Facility: the metamodelling architecture for MDE, can be considered a standard for writing metamodels.

XMI – XML Metadata Interchange is the OMG's standard for interchanging metadata (e.g. models) through XML. It can be used for any model expressed in MOF, for example all UML models.

QVT – Query/View/Transformation is a standard set of languages for model transformation.

3.2.2 MDE and control applications

Model-Driven Engineering and MDA are relevant to this work because it leverages the use of explicit knowledge in the form of models in the development of systems, in this case software systems.

Hastbacka et al. [61] propose an approach to use MDE and domain-specific modeling to industrial process control applications. In these applications there are different disciplines involved, and modeling concepts and methods are needed for control patterns and algorithms, domain-specific concepts, or instrumentation (sensors, actuators). For that purpose they have developed the UML Automation Profile (AP) based on the first-class extension mechanism for UML, which constitutes a metamodel for the domain of process control applications.

3.2.3 Models and metamodelling

In software engineering a model is an artifact constructed according to a certain a modeling language, such as UML [105, 104], that describes the system, including usually a graphical representations of it using different types of diagrams. These diagrams allow an easier and faster understanding of the software system than the code level.

> A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made. [79]

Models adhere to the closed world assumption [11]: anything that is not specified by the model is implicitly disallowed (or allowed). Models in MDE specify the implementation of systems, they are thus prescriptive.

The OMG MDA approach is based on the utilization of a language to write metamodels called the Meta Object Facility (MOF). MOF is defined as a four-layer architecture for metamodelling, as depicted in figure 3.2. It provides a meta-meta model

at the top layer, called the M3 layer. The M3 model is the language used by MOF to build metamodels, called M2 models. The most prominent example of a model in layer 2 is the UML metamodel. M2 models describe M1 models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe real-world objects. MOF is a closed metamodelling architecture; it defines an M3 model, which conforms to itself.

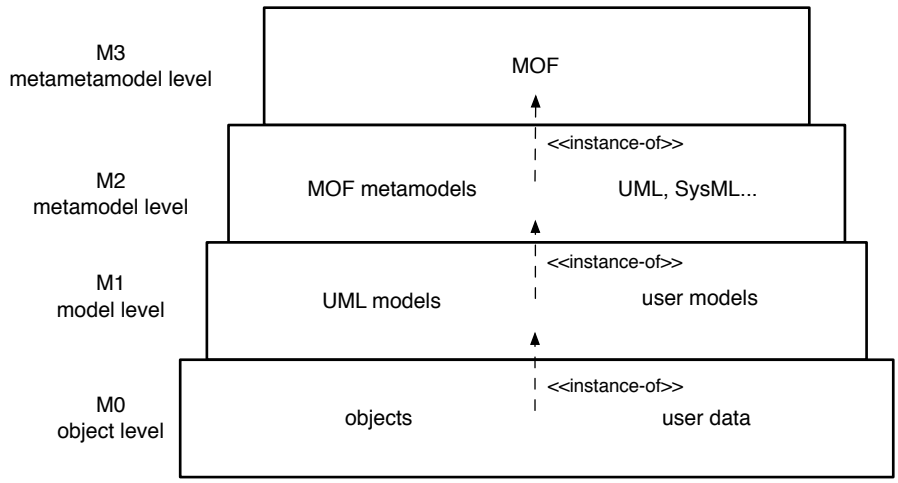


Figure 3.2: OMG's metamodelling architecture: MOF.

Metamodels

A metamodel is a model of models, [63] or a prescriptive model of a modelling language.

> "[A metamodel is a] model that defines the language for expressing a model."[104]

A model is said to *conform* to its metamodel when each element in the model maps to a definitional element in the metamodel [63]. Atkinson et al. discuss about this, since strict conformance does not always hold, and when it does it leads to some inconsistencies [13]:

Strict metamodeling: is based on the tenet that if a model A is an instance-of another model B (its metamodel) then every element of A is an instance-of some element in B.

Loose metamodeling: one model is an instance-of another model, but where the strict requirements on the instance-of relationship between individual model elements does not hold.

3.2.4 Ontologies

A special flavour of models, ontologies, became relevant to the Knowledge Engineering community some years ago, mainly promoted by the Semantic Web. The term ontology was borrowed by the software community from Philosophy as ontology was understood a systematic account of Existence.

This notion of ontology developed from Artificial Intelligence research on how to represent knowledge to support intelligent behaviour [12]. This is way they are typically intended for exploitation of knowledge at run time, and must be "formal" in the sense of being understandable by a computer —i.e. they support automated reasoning.

One of the most agreed definition is that of Guarino [56]:

> An ontology is an explicit specification of a conceptualization.

An ontology formally represents knowledge as a set of concepts within a domain, and the relationships between those concepts. It can be used to reason about the entities within that domain.

An ontology consists of classes, instances, functions, relationships and axioms. Classes correspond to entities in the domain. Instances are the actual objects which are in the domain. Functions and relationships relate entities in the domain. Axioms constrain the use of all the former elements [56].

Ontologies could be considered to be reusable building components when modeling system at a knowledge level. They have been regarded as an extension to knowledge–level modeling, by enabling knowledge sharing and reuse. Therefore, they are key techniques to build knowledge–intensive systems [18].

Many authors ([11, 63]) agree on the distinction of two main categories of ontologies:

Domain ontologies: most works on ontologies refer to domain ontologies, which express conceptualisations of a certain domain in the world, e.g. medicine, defining a shared vocabulary in between the stakeholders of that domain.

Foundational or upper-level ontologies: provide a conceptualisation for other ontologies, making no claims about the world.

3.2.5 Ontologies vs Models & Metamodels

We have presented two main approaches to modeling in software systems that are relevant for this thesis: the metamodelling approach in the MDE paradigm, and the ontology approach used for knowledge based systems. In the following we present their differences, but also their possible relations to complement and integrate in the MDE approach.

Differences between models and ontologies have been summarised in [11]:

- Sharedness: ontologies are meant to be reused and share among users, whereas models are less so.
- Open–world assumption: it states that anything not clearly defined is unknown. Ontologies rely on this assumption, as they regard the lack of knowledge as unknown. It does not imply falsity of former knowledge when adding new information. As opposed, models usually consider a closed–world assumption, i.e., anything not specified is considered to be false, to prevent unforeseen consequences when extending the system.
- Descriptive properties: ontologies are usually descriptive, i.e., they describe the reality (such as a domain), but reality (domain's objects) is not built upon it. Models, on the other hand, are prescriptive. They define the structure or behavior of reality (such as a system), which allows to construct reality (system's objects) as specified in the model. Therefore, prescriptive system models are usually known as system specifications.

Ontologies are used to model domains in knowledge-based systems. Models, on the other hand, are generally used for specification purposes for software systems. They usually describe the expected behavior of the system.

Ontologies and Metamodeling

Notwithstanding the previous differences from models, ontologies can be understood in the context of metamodeling. On the one hand, an ontology could be considered as a metamodel, as it describes the constructs (entities) and rules (relationships and axioms) of a domain of interest [18]. However, an ontology is *descriptive* describing the domain of the problem whereas a metamodel is *prescriptive* describing the domain of the solution. Ontologies do not describe systems, only domains. They should therefore play the role of an analysis model [11]. Pidcock [114] argues that:

> A valid metamodel is an ontology, but not all ontologies are modeled explicitly as metamodels

Assman et al. [11] propose an integration of ontologies in the OMG's standard meta-modelling hierarchy as depicted in figure 3.3. Henderson-Sellers [63] continues on this idea proposing to use foundation ontologies at the same abstraction level than metamodels, and domain ontologies at the same level than (design) models.

3.3 Functional Modelling

This thesis explores the possibility of moving the engineers' role to design/adapt a system into the system itself at run-time. This role relies in a functional understanding of the system (page 28). A merely ontological knowledge, capturing the structure and behaviour of the system, is not sufficient.

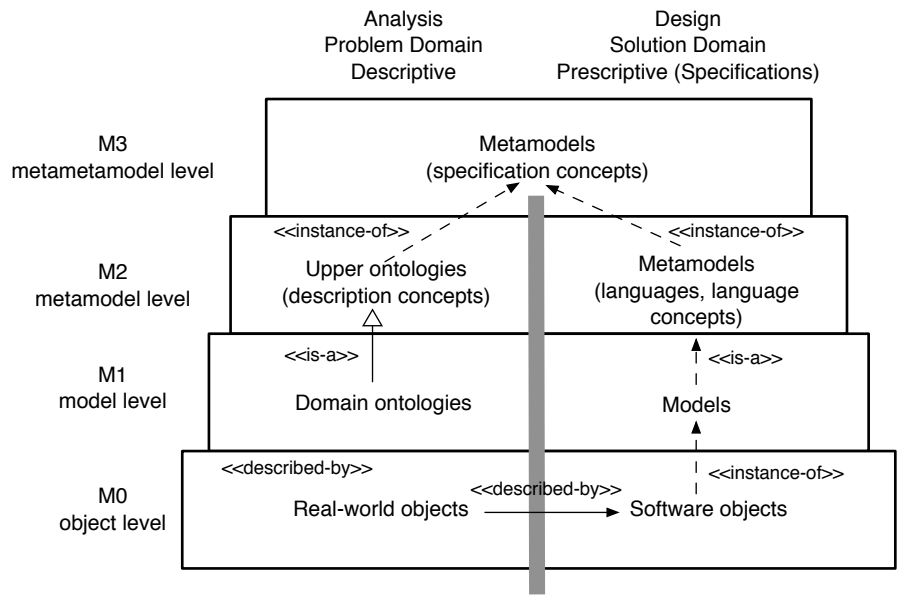


Figure 3.3: Integration of ontologies in the OMG metamodelling hierarchy proposed by Asmann et al. (adapted fom [11]).

For a deep understanding of the engineered system, knowledge of the mission and the relation mission-system is crucial. It is necessary to include the intention of the designer, the purpose for designing the system that way and not in a different manner. Why use those components organised that way? How does that system organisation or configuration addressed the requirements?

Functional modelling is an emerging discipline that focuses on representation formalisms and methods to incorporate teleological knowledge, about the intention and purpose of designers, into the models of systems that are used in engineering activities.

According to Modarres [96]:

> Functional Modeling is an approach used to model any man-made system by identifying the designer defined overall goal it must achieve and the designer/user defined functions it must perform.

Lind's definition makes clear the engineering context of functional modelling:

> Functional Modeling comprises concepts, methods and tools which for representing the purposes and functional organization of complex dynamic systems [88].

The aim of functional modelling is to relate the ontology of the system, i.e. its physical structure and behaviour, and its purpose or goals. Chittaro argues for this relational character of the concept of *function* in [35].

3.3.1 Functional concepts

As Chittaro [35] notes, the term *function* is usually associated to a component or part of a system, and understood in an operational sense as the relation (e.g. a mathematical function) between the input and output of the component, be them energy, mass or information.

However, in functional modelling, and also in this thesis, the term *function* is used in its teleological flavour, referring to the relation between the purpose of the component and the behaviour rendered by its structure. This is conceptualised by the *means-ends dimension* of functional modeling argued by Lind [87]. Lind elaborates this conceptualization to differentiate the two views of an engineered system: causality (relating structure with derived behaviour) and intentionality (desired behaviour, i.e. teleology).

Functional modelling is based on a simple set of fundamental causal and intentional concepts:

Structure: system's physical parts and their interconnections in the physical topology [87].

Behaviour: physical mechanisms or phenomena responsible for the interactions [87].

Intentionality[3] concepts:

Goal: states the outcome or objective toward which certain activities of a system or its parts are directed.

Function: roles the designer intended a subsystem to have in the achievement of the goals of the system of which it is a part [87].

3.3.2 Uses of functional modelling

In contrast with conventional methods for modelling systems in control systems engineering —differential equations, frequency analysis, etc.— which support quantitative reasoning, functional modelling is intended for qualitative reasoning.

The main areas of application of functional modelling are [87]:

Design: reasoning on the functional model of a system can help re-design it, and functional modelling can help in the conceptual analysis and synthesis of control systems, or in control re-configuration. [42].

Diagnosis: of disturbances and failures.

Operation: supervisory control of complex plants, planning of remedial actions.

[3] note that the term "intentionality" is not used here in the sense of of the philosophical analysis of consciousness referred to in previous section 3.1, but in Lind's sense, closer to the vernacular sense of our common language

Functional models can be used by design engineers off-line, or automated reasoning can be performed on them by tools on-line to provide support for operators [87].

3.3.3 Functional Modelling Techniques

There are different techniques in functional modelling, three of the most relevant are the Multilevel Flow Modeling by Lind et al.[87], the Goal Tree–Success Tree by Modarres and Cheon [97], D-Higraphs by De la Mata and Rodríguez [42] and Blanke et al. component model [21].

Multilevel Flow Modelling

Multilevel flow Modeling (MFM) is a modeling technique to produce graphical models of complex process plants. It is based on the means-ends concepts, to describe the system in terms of components, functions, and the physical components that realise them; and whole-part decomposition, to represent that functional structure at different levels of abstraction in a hierarchical way. Flows —of mass, energy or information— are the core representation elements for functions in MFM because they are responsible of the main interactions in a process plant.

Goal Tree – Success Tree

The Goal Tree – Success Tree (GTST) is a functional modeling technique for complex physical systems. A GTST is a functional hierarchy of the system organised in levels starting with "functional objective" at the top, which describes the main purpose of the system. The functional decomposition proceeds downwards so that sub-functions must explain *how* a function is achieved. At the bottom, the role or purpose of actual pieces of equipment must become explicit and unambiguously defined. The success tree (ST) in the GST describes the topology or structure of the system related to the physical, i.e. bottom level, functions of the GT part. Modarres proposes to use the Master Logic Diagram (MLD) to present the success logic of complex systems.

D-Higraphs

D-higraphs [42] is a formalism that includes in a single model the functional as well as the structural (ontological) components of any given system. It is a functional modeling technique for process plants based on Higraphs, which are an extension and combination of conventional graphs and Venn diagrams. However, it is also rooted in MFM and GTST.

In D-Higraphs, structural and functional views are merged in its basic element: the *blob*, which represents a function and the device that performs it. Blobs are connected by *edges*, which represents flows like in MFM. Blobs can be nested, the parent

function requiring its nested to perform. This captures the hierarchy of functions. But blobs can also be partitioned, each partition representing a different possible realisation of the function of the blob. Blobs can also intersect, that meaning that the actors/functions can be shared.

Component models in fault-tolerant control

While the previous modelling techniques have been developed in the process industry, there are other modeling techniques that can be considered functional modeling, such is the case of the use of component models in fault-tolerant control. Generic component models [21] describe the system architecture formally, as a set of components and their interconnections, in order to perform analysis for fault-diagnosis and fault-tolerant-control design. Components can be considered at any level in the system hierarchy.

The focus of the generic component model are the *services* that the component provides. A service is a transformation of some *consumed variables* into some *produced variables* according to a certain *procedure* and using some *resources*. Some components have built-in tolerance, so that is some resources needed to offer a service are faulty, they can still provide it through a different *version* of the service, that employs different resources and/or inputs. Moreover, the component usually renders different services during different operating modes of the system, e.g. initialisation, safe modes. The component model captures the structure of services in different *use-modes*, and the possible transitions between them.

3.4 Patterns

The plasticity of software makes it ideal for the construction of most information processing functions, the very essence of any controller [127]. It has thus come to play an increasingly important role in control systems, making them not only hardware but also software-intensive. This variety makes more complex their design and implementation, because the knowledge required spans several disciplines: electronics, software engineering, automatic control, and domain specific knowledge —such as the differential equation models used to capture system dynamics. There are no perfectly established methods to manage this heterogeneous knowledge and facilitate its use in the design of control systems. Adequate engineering techniques and procedures are needed bridging in-between software developers and control experts in the engineering teams.

A methodological tool of extreme importance is the possibility of capturing partial designs that can be composed to generate complete systems. This is well known in the domain of control engineering, where textbooks are full of controller structures that capture just some of the essential parts of control systems. This is also well known in software engineering, where design patterns have become the standard way of capturing design solutions to common problems.

As presented in [138] and Figure 2.2, the ASys approach to building autonomous systems is strongly based on the use of *design patterns* to document, transfer and exploit autonomous systems design knowledge.

3.4.1 Design Patterns

The pattern concept that we use in this work is the concept originally evolved in the domain of object-oriented software engineering [51]. A *design pattern* is a reusable solution to a recurring problem of general nature in a specific context of use. The software pattern idea was inspired by the work of an architect, Christopher Alexander, and his colleagues, who first described a pattern (in the architecture of buildings), as a three-part rule expressing a relation between a certain context, a problem, and a solution [5].

During the 1990s, the design pattern methodology was successfully adapted for use by the software community, as forms that describe recurring programming language idioms, architectural designs or how-to methods in software engineering. While patterns can address system-wide design issues, they are usually not complete designs for whole systems but descriptions of partial aspects of the whole design. In many cases they are immediately translatable into portions of code, offering a solution template of problem solving strategies that may be instantiated for concrete problems. In principle, patterns are used to capture good practices and the anti-patterns are used for capturing bad practices: things to do versus things to avoid when designing or implementing specific applications [113].

They have been amply used in the development of object-oriented software applications, constituting basic building blocks specially at the level of the detailed design of software mechanisms. The work by Gamma *et al.* has become such a reference in this area that both the patterns described in that work and the authors themselves have become nicknamed as the *Gang Of Four*. They have defined a new language to talk about object-oriented designs, where the names of the patterns become new domain terms to be used in the communication between systems design stakeholders. Well known examples are the Observer Pattern the Façade Pattern or the Singleton Pattern. Such a collection of new terms constitute what has been called a *pattern language*: it is used to talk about a system design. The ASys project intends such a feat: the development of a *generative* pattern language for the systematic design-driven construction of model-based autonomous systems.

3.4.2 Pattern Schemata

To make patterns truly reusable, standardised descriptions of them should be available. Standardization easies the communication of the content of the patterns among all stakeholders. It offers a scaffolding to organise pattern descriptions so they can be used systematically for their intended use. For that purpose authors capture and document patterns following a stereotyped form or template *i.e.* a *pattern schema*.

A pattern description using a pattern schema usually consists of plain text organised in sections according to the different aspects of the pattern, complemented by appropriate figures, such as block diagrams, state machines, *etc*.

A well-known schema was proposed by Dough Lea in the Patterns FAQ. This schema is based on Alexander's original work in architectural patterns [5]. The essence of the pattern description is the communication of what is the solution to a problem and what are the factors to take into account. Summarily it could be read as:

```
IF    you find yourself in CONTEXT
      for example in EXAMPLE
      with a PROBLEM
      entailing some FORCES
THEN  for some REASONS
      apply a DESIGN RULE
      to construct a SOLUTION
      leading you to a NEW CONTEX
      and OTHER PATTERNS
```

There are other schema alternatives, such as the one used by the Gang of Four, the schema used by Buschmann *et al.* in their architectural patterns book [30] or the pattern schema proposed by [125] in the domain of software-intensive controllers. The pattern examples showed along this section adheres to a simple template based on this, reducing it to the more relevant sections for the explanation in each case.

3.4.3 Patterns for Control Systems

Control engineering can greatly benefit form the use of design patterns [127]. Today's complex control systems involve sophisticated functionality and a myriad of elements with heterogeneous implementation technologies (fault-tolerance mechanisms, simulators using differential equations, expert systems using fuzzy-logic rules...). This is especially true in cognitive control systems, where sophisticated system architectures and multidisciplinarity are common trade.

In this context, pattern technology is crucial because patterns are capable of expressing design knowledge across disciplines and design layers. This way they facilitate the share of knowledge between the different stakeholders, because they provide a common, abstract vocabulary to talk about design and implementation.

Patterns can be very useful at the architectural design level of such systems, because they can be used to express details of the controller architecture that are not amenable to more formal and technology-specific descriptors (like the ordinary differential equation approach or classic control engineering or the Petri net approach of the discrete systems communities).

In its most common form, an architectural design pattern specifies *roles* and interactions between the entities performing those roles. For example, in the more basic

control pattern, a *controller*[4] receives sensing information from the plants thanks to the *sensors*, and actuates over it by commanding the *actuators*, to have a certain goal value y_{ref} in a variable y of the plant.

Patterns for Architecture Generation

To properly manage patterns, they must be organised. Common practice is to gather them into collections or frameworks, according to a certain classification criteria: design level, application domain, life-cycle phase, etc. For example, Gamma *et al.* [51] categorize software patterns in three groups according to the function they perform in a system: creational patterns, structural patterns, and behavioural patterns.

When patterns in such a collection are closely interrelated so that they can be used to describe or design a whole system, they form a *generative pattern language* [38].

Patterns capture the very essence of each design, and the central piece in the design of a system is its architecture. Patterns simplify the process of designing the architecture of a new system: they abstract from the application-specific details the fundamental traits of the architecture that will define the most general system properties —e.g. performance or resilience.

Good patterns are composable so that they can be used to define new architectures for other domains and applications —other requirement collections— reusing the principles captured in them.

Pattern Collections for Control and ASys

There are many catalogs of patterns for different application domains. For example, [140] offers a collection of patterns for the construction of concurrent and distributed systems using object-oriented software technology.

The same can be said for the control systems domain. Apart of the classic approach used in control engineering, there is also documented experience of the use of software patterns in control systems. One of these examples is the Patterns for Time-Triggered Systems (PTTES) Collection [116]. The PPTES collection addresses the implementation level, capturing detailed design aspects for embedded control systems: programming language constructs, real-time kernels, etc.

At a higher level, [95] offers a collection of design patterns for constructing cognitive systems systems (from autonomous robots to intelligent information retrieval agents). These patterns have been obtained by detailed examination of over twenty research-oriented cognitive systems. As Miller says "Rather than reinventing the wheel, these design patterns can be reused for architecting future cognitive systems".

Given the architecture-centric basis of ASys, it has focused on architectural patterns. Part of the ongoing work directly targets the use of pattern technology to provide

[4] role names will be written in italics

an extensible framework to express control application designs, mostly architecture in the domain of complex controllers. The ultimate goal within the ASys context is the creation of a generative pattern language to support the construction of intelligent integrated controllers for any class of autonomous system in any class of mission.

3.4.4 Pattern Examples

Lots of pattern examples can be found in any of the pattern catalogs mentioned so far (e.g. Gamma et al. or Buschman et al.). Some more specific patterns in the domain of control systems can be found in the work of Sanz et al. [127, 125].

Feedback

The feedback pattern lies at the very core of control theory.

Pattern: **Feedback**

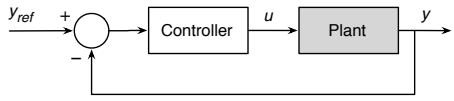
Related Patterns: Feedforward, proportional control, PI control, PID

Context
The system has a measureable output and a controllable input. A system model may not be available. A desired reference signal exists.

Problem
Make the output follow the reference. Unmodelled plant dynamics and the disturbances make open-loop control impossible. The system, output does not behave as desired: the response may be too slow, to oscillatory, unstable...

Forces: a) plant behaviour is affected by disturbances, b) knowledge about the plant is not always accurate

Solution
The plant input (u) is determined from the difference between the reference signal (y_{ref}) and the plant output (y) by a feedback controller.



Variants
State feedback

References
Pattern adapted from [127]

TMR

Pattern: **Triple Modular Redundancy**

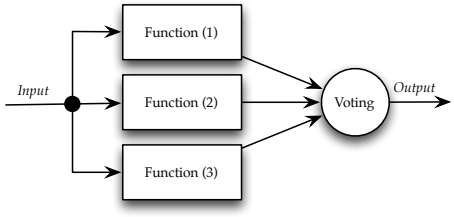
Related Patterns: Hot replica, Redundancy.

Context
The system has a modularised function.

Problem
The module that implements the function can fail and this is not tolerable for the mission of the system.

Forces: a) redundancy is costly, b) safety is mandatory, c) not all faults can be tolerated.

Solution
The system output —the function— is provided by a voting system that uses three realization of the module that implements the function. The value that is output is the result of the majority voting among the three modules.



Variants
5-modular redundancy.

References
Invensys' Tricon is a state-of -the-art fault tolerant controller based on a Triple-Modular Redundant (TMR) architecture. It uses two-out-of-three voting to provide high integrity, error-free, uninterrupted process operation.

Fuzzy control

Pattern: **Fuzzy control**

Related Patterns: Stochastic control, expert control.

Context
We have some control knowledge extracted from humans in vague terms.

Problem
The plant is so ill defined and unidentifiable that there is no formal mathematical model of it to be used in the design of the controller.

Forces: a) precision is desirable, b) robustness is necessary, c) humans cannot introspect easily.

Solution
Extract the human control knowledge in the form of fuzzy IF-THEN rules. The controller is implemented as a rule-based fuzzy inference engine with a fuzzyfication and a defuzzyfication processes before and after the fuzzy engine.

Variants
Bayesian control.

References
[135]

3.5 Fault-tolerant systems

Dependability issues have become a crucial challenge for control systems (section 1.4), specially in autonomous applications, where there are no operators included in the run-time loop to detect and help recover from failures. A fault is something that changes the behaviour of a system such that the system no longer satisfies its purpose [21]. Fault-tolerance is an engineering field that develops methodologies to prevent, mitigate the effect and recover from faults.

According to [71] we shall distinguish three concepts in relation with reliability: a failure is a deviation of the system behavior from the specification. An error is the part of the system which leads to that failure. Finally, a fault is the cause of an error. A fault can be an internal event in the system, a change in the environmental conditions or it can even be a wrong control action given by a human operator or an error in the design of the system.

We can distinguish two main approaches to improve the reliability of a system: fault *prevention* and fault *tolerance*. It can be assumed that prevention technique will not solve the problem, since in real-time systems new or unforeseen faults are prone

to happen that engineering could not have accounted for specifically at design time. Fault tolerance, on the other hand, faces the problem once a fault occurs in the running system.

A system is said to be fault tolerant if it maintains its functionality, or and allowed degraded version of it, in the presence of faults. When the system suffers a failure, nothing can be done, its behaviour does not comply with the specifications. But when a component of the system fails something can be done so as the overall system does not do so: this is the goal of fault tolerance.

Different approaches to fault-tolerance have been developed in different disciplines: software, control, process industry, etc. In the following we present the two of them more relevant for this thesis.

3.5.1 Fault-tolerant software systems

Fault-tolerance has been deeply investigated in computer-based systems. Jalote [71] identifies the following phases in the operation of a fault-tolerant system in the presence of a failure:

1. Error detection: the presence of a fault is deduced by detecting an error in the state of the subsystem.
2. Damage confinement and assessment: the damage caused by a fault is evaluated and delimited (affected parts are identified and effect on objectives estimated).
3. Error recovery: correction of the error to avoid its propagation.
4. Fault treatment and continued service: faulty parts of the system are deactivated or reconfigured and the system continues operation.

Fault tolerance can be, and is applied at various levels in a computer system. Therefore, it distinguishes between hardware and software. Hardware fault tolerance is based on fault and error models, which permit identifying faults by the appearance of their effects at higher layers in the system (software layers). Hardware fault tolerance can be implemented by several techniques, being the most known:

TMR (Triple Modular Redundancy): three hardware clones operate in parallel and vote for a solution.

Dynamic redundancy: spare, redundant components to be used if the normal one fails.

Coding: addition of check bits to the information bits such that errors in some bits can be detected and, if possible, corrected.

Research on fault-tolerance in computing systems has been very important to the establishment of a general conceptualisation and theory about faults. However, its specific techniques are not as relevant to this thesis as those developed in the control field, which we present next.

3.5.2 Fault-tolerant control

There is a control-based approach to the design of fault-tolerant systems: fault-tolerant control. It concerns with the interaction between a given system (the plant), that may eventually be subject to some fault, and its control system, understood in its broadest sense presented in chapter 1, i.e. not limited to the feedback control law, but including the complete system, from low level I/O to higher levels involving decision making.

In classical control the controller is designed considering a faultless plant, so that the closed loop achieves the desired objective or function. Notwithstanding, there are well known techniques that allow the accommodation of faults to some extent:

Robust control: in this control technique [44] the controller is designed to tolerate changes in the plant dynamics, while it keeps fixed, so it can be considered as passive fault tolerance, with no run-time reconfiguration. However, robust controllers exist only for a restricted kind of changes, and they works sub-optimaly for the nominal conditions, since their parameters are determined from a trade-off between performance and robustness.

Adaptive control: the controller parameters are effectively modified according to changes in the plant model parameters. In the case that the plant changes are due to some fault, this technique may provide *adaptive fault tolerance*. Unfortunately, adaptive control is efficient only for linear systems whose parameters have slow variations, conditions not usually met by faulty systems, a fault typically causing non-linear behaviours with abrupt changes.

A fault-tolerant controller, on the contrary, is capable to react to the occurrence of a fault of any magnitude in principle, changing the control law so as to maintain the behaviour of the closed-loop system in a region of acceptable performance.

Faults are usually classified into:

- Plant faults, that change the dynamical I/O properties of the system.
- Sensor faults, when there are errors in the sensor readings, probably rendering the system unobservable.
- Actuator faults, that causes the controller not be able to properly actuate on the plant, rendering the system uncontrolable.

There are generally two main phases in the behavior of a fault-tolerant controller[21]:

1. Fault diagnosis: The existence of faults is detected and they are identified. This corresponds to the error detection and fault assessment in software systems.
2. Control re-design: the controller is adapted to the faulty situation so the overall system achieves its objectives. This corresponds to the error recovery and fault treatment in software. It is the system reconfiguration or adaptation.

The basic elements in the architecture of a fault-tolerant control system are shown in 3.4. The *Supervisor* level gathers all the elements that are built on top of the traditional control system to provide for fault-tolerance as schematised in figure 3.4. They

can be differentiated in two main blocks addressing the principal phases previously described (3.5.2):

Diagnosis block analyses the consistency of the actual I/O from the plant with the model, so as to identify the existence of any possible fault.

Controller re-design uses fault information to adapt the controller to the current faulty situation.

The traditional control system forms the *execution level*, assuring set-point following. The supervisor forms an additional loop on top of that, actuating when a fault occurs by changing the controller.

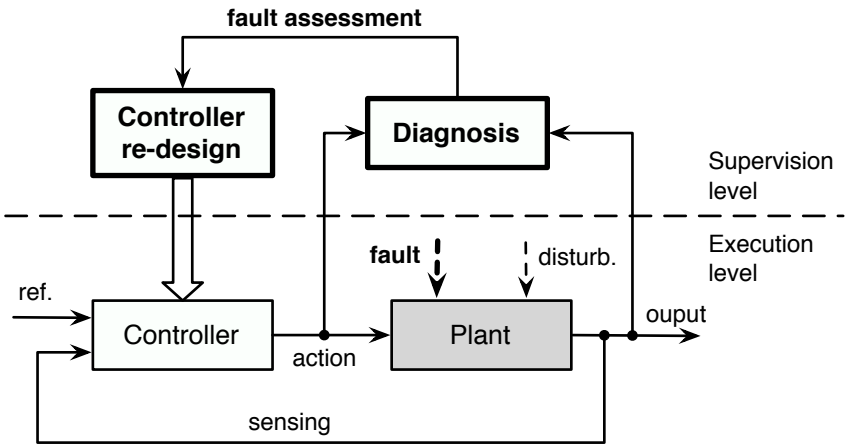


Figure 3.4: Architecture of fault-tolerant control, adapted from [21].

To accomplish this behaviour we can distinguish two main approaches:

- The traditional methods for fault tolerance include limit-checking and spectral analysis of certain signals for fault diagnosis, and when specific faults are detected the controller re-design consist of the switching to a redundant component. This is the case of software dynamic redundancy. This approach present some disadvantages: it requires ad-hoc engineering based on experience and process knowledge, for example to identify the possible faults with measurable signals in the system, and it relies on **physical redundancy**, which is expensive for requiring duplication of components —TMR is another example—. This is thus only affordable for safety-critical systems.

- Fault tolerant control follows the principle of **analytical redundancy**. An explicit mathematical model is used for both phases. The diagnosis of the fault is performed using information from the model and from measurement signals. Then the model is adapted to the new scenario and the controller re-designed accordingly, so the system with the faulty plant continues to achieve the objectives. Redundancies are still needed for reconfiguration, but that does not imply

duplication of many components: one extra sensor can provide enough analytical redundancy, and re-configuration can be achieved by controller changes rather than substitution for another one.

For all the reasons above explained, fault-tolerant control is a most promising field for providing for more realiable systems. It can address a broader span of faults than classical control techniques, while being more efficient, and is cheaper than ad-hoc traditional methods for fault-tolerance. In this respect, it presents the great advantage of providing a systematic methodology with an architectural approach that will be described in the following section.

3.6 Components for Control Systems

Sophisticated controllers —esp. for high autonomy applications— are software-intensive systems that benefit from the use of component-based approaches. Component technology [149] is a well-established engineering approach to the construction of complex software systems. Apart of the inherent modularity that it provides there are several other reasons that justify this approach and that will be described later (see Section 3.6.2).

Component-based construction emphasizes reuse by composition of pre-built elements that are organized in software frameworks [28, 111]. The construction of software systems using reusable components shifts the effort from design + programming to component selection + composition. This reduces the development effort and time, potentially increasing quality and somewhat decreasing the flexibility of the design. In the context of this thesis, the component frameworks can be classified into two categories:

General component frameworks: These are software platforms that enable the construction and reuse of componentized software. While they are specified and built with some requirements in mind —requirements coming from some domains of use— they are not application-finalistic.

Domain specific frameworks: These are frameworks that offer concrete functional components that offer specific services for an application domain.

In the case of the development of control systems, both kinds of frameworks are common trade [27, 26]. General component frameworks like RT-CORBA/CCM or Ptolemy offer the possibility of building any kind of autonomous control application. on the other side, examples of domain-specific component frameworks for control abound. Apart of the well established ROS [52] —the one selected for this work— there are many others that have been built as part of different research programmes that recognized the possibility that software reusability gives to control architecture experimentation [120]. In our opinion, the most intelligent strategy is to build a domain-specific, control framework atop a general component frameworks (e.g. in the robot control ORCA [142], BRICS [77] or OpenRTM [8]). Recent attempts at standardization are also worth considering [102].

3.6.1 Rationale for Components

The component-based approach to software development is a pervasive strategy that is now fully mainstream in the domain of embedded control systems. This is somewhat surprising given that the traditional strategy for embedded systems has been driven by the primary goals of maximizing performance whilst using minimum computational resources —esp. memory footprint and power consumption. Assembly language programming still remains as a development practice in those systems. However, even for the most stringent class of embedded applications —those deployed over Digital Signal Processors (DSP)— the development tools now provide component models for software that are tailored to such devices. The component approach is nowadays the standard development strategy in mobile appliances —e.g. on Android or iOS devices— but in the case of control systems the road is still being paved. In the case of software-intensive controllers, the principal themes are the treatment of real-time and resilience requirements for embedded applications (RT/E) through the use of a standardized component model.

The evolution towards component-based control middleware is the need of software reusability. Reuse allows fast development of systems by composing previously developed building blocks that provide specific functionalities. Components are building blocks that provide application-level services to other components or external systems through clearly defined component interfaces. Components typically execute inside a component container provided by the framework, that is tailored to a specific target platform, and provides system-level and management services to the components it hosts.

Robot control research needs the flexibility offered by these systems to explore alternative designs and efficiently synthesize the controller. From the functional design standpoint, the controller is composed of interconnected components; from a configuration and deployment (non-functional) standpoint, the platform containers manage components' life-cycles. The component-container model allows the use of software modules that are defined and managed at a higher level of abstraction than allowed by conventional imperative or object-oriented programming models. The separation of interface, implementation, and life-cycle is very well adapted to autonomous systems research circumstances where an application requires software units to be distributed across multiple devices and be dynamic in their deployment and interaction.

3.6.2 Advantages of Component Technology

There are many advantages of using component-based technology in the robotics domain, fully rooted in the core architecture of the component-container model.

Portability: Control applications are developed for a wide diversity of operating systems and computing platforms. The core abstractions in the component model enable the development of specialized (fit-for-purpose) containers, that replace the hardware abstraction layers of conventional software portability methods.

This allows the re-deployment of components from one RT/E device to another without the need for source-code modification.

Reusability: In a component platform there are formal definitions that express the component dependencies and therefore enhance visibility and enforcement of rules for application assembly. This has the overall effect of minimizing ambiguity when choosing components for re-use, reducing development costs, and reducing time-to-operation that is the objective in robotic research software construction.

Separation of Concerns: In conventional software control systems, there is often an unavoidable need of understanding system-wide issues —e.g. non-functional properties, enabling technologies, software integration, global system architecture etc. The separation of concerns evident in the component approach provides the opportunity for domain experts to focus exclusively on the design development of functional elements, resulting in a more cost-effective use of development resources.

Visibility: In a standards-based component model (e.g. CCM), organization coding styles/patterns can be formally specified and even offered in the public domain. This fosters more open business models, dynamic practices, cleaner integration and encourages closer partnerships between collaborating parties.

Quality in Development: Conventional programming leads to monolithic systems that hampers testing and validation efforts. The re-composition using component-based architectures, allows software units to be naturally isolated for testing and validation. Component-level testing becomes the central functional assurance process. Component quality creates a true opportunity of using an evolutionary approach to control system development and test.

Quality in Production: Strategies for self-management that ensure high system availability —a core objective of this thesis— can be applied more systematically using components deployed over standardized containers than with other more conventional systems.

Chapter 4

Theoretical Framework

In this chapter a theoretical framework for autonomous cognitive systems is presented. This conceptualisation has been elaborated to provide the scientific basement for the work developed. It also allows to analyse in a common frame the different approaches in the state of the art that address the problem at hand of self-awareness and runtime adaptivity. To address the goal of generality, the conceptualisation is rooted in the general systems theory and a previous theoretical foundation by López [90] that applied it to autonomous systems.

4.1 Introduction

Provided the aim of this thesis to seek for a universal —or universalizable— solution for autonomy engineering, applicable to any kind system, a general theoretical basement to analyse the problem was required.

ASys modelling approach to autonomy claims that the system shall exploit its engineering models to optimise and adapt its behaviour at runtime. Therefore, this basement conceptualisation should be able to support the analysis of the system from the engineers perspective, and also to account for the representations the system will use about itself to drive its action. It is the exploitation of these very explicit models that we identify with cognition. Therefore the framework should also account for this view of the cognitive phenomenon.

In his *Foundation for Perception in Autonomous Systems* [90] Lopez develops fundamental concepts of cognitive autonomous systems. This conceptualisation provides a completely general analysis of autonomous systems, since it is based on general systems theory, and provides a well integrated and solid account for cognition. These are the reasons why it was selected as the appropriate foundation for this research, which could be considered as an engineering reification of it.

In the following sections of the chapter the concepts that form this theoretical backbone are discussed, from the more general systems theory concepts to our particular formulation of the functional/structural perspectives of engineered systems, passing by the ideas about autonomy and cognitive systems.

4.2 General Systems Theory

Lopez's theoretical framework for autonomous systems is based on the general systems theory, specifically on the formulation by George J. Klir [76], which is a precise one that was found found desirable for its application in an engineering domain.

General Systems Theory (GST) is an interdisciplinary field of science and the study of the nature of complex systems in nature, society, and science. More specifically, it is a framework by which a system is understood as a set of elements, each with its own properties, and a set of relations between them that causes the system to present properties that can not be inferred by only analysing its elements separately. The system could be a single organism, any organisation or society, or any electro-mechanical or informational artifact. Ideas in this direction have been pointed out back to personalities such as Leonardo or Descartes. However, Ludwing Von Bertalanffy with his works on General Systems Theory [155] in the middle of the 20th century is regarded as the pioneer formulating the concept as it is understood nowadays.

In the following we present the basic concepts of Klir's GST that are relevant for our theoretical framework of cognitive autonomous systems.

4.2.1 Fundamental concepts

Let us think about what we understand by system, by considering it in relation to what surrounds it. If all possible entities form the ***universe***[1], a ***system*** can be regarded as a part of it, which is considered isolated from the rest for its investigation. All which is not system is called ***environment***. The different disciplines of science share this general understanding in particular ways, usually differentiated from each other in the criteria for separating the system from the universe.

The observer selects a system according to a set of main features which we shall call ***trait***. They will be characterised by the observer through the values of a set of ***quantities***. Sometimes, these values may be measured, and we talk of *physical* quantities, such as length or mass. Other times quantities are *abstract*, and they cannot be measured. The instants of time and the locations in space where quantities are observed constitute the ***space-time resolution level***. The values of the quantities over a period of time constitutes the ***activity*** of the system.

[1] concepts that will be explicitly incorporated to the theoretical framework will be identified by typesetting them this ***way***

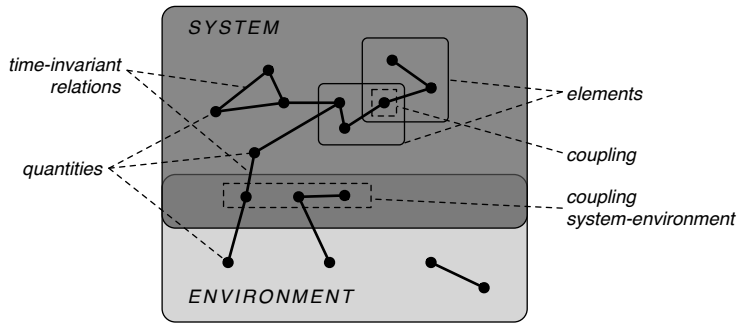


Figure 4.1: Basic notions of General Systems Theory.

The main task of the observer is to explain the activity of a system. This will be accomplished by identifying patterns in the activity of the system. The quantities of the system may satisfy ***time–invariant relations***, by which the values of some quantities may be expressed as function of others. The set of all time-invariant relations is the formal notion of ***behaviour*** of the system.

We may realise that the behaviour is due to the ***properties*** of the system. In other words, a system with different properties would exhibit a different behaviour. The set of all properties is be called the ***organisation*** of the system.

The study of a whole system can be really complex. To address it the quantities of the system can be divided into groups, and then each of them analysed as a system on its own. This groups of quantities are the ***subsystems*** or ***elements*** of the system. Elements may share a set of quantities, which are thus the ***coupling*** of the elements. Since we are interested in systems that interact with the world, there would always be a coupling system–environment too. The system's elements and their coupling form the ***universe of discourse and couplings*** or UC-structure of the system, which define the structural aspects of the system.

The dynamics of the system are given by its ***state-transition structure*** or ST-structure, which defines all the possible states of the system according to the values of its quantities, and the possible transitions between those states.

Example

GST analysis of the mobile robot

We could analyse our mobile robot from a GST perspective. The system object of interest is the control system, the rest, that is the world and the robot body and accessories not directly involved in the control functions, correspond to the environment. The elements of the system are the different components of the controller: the localisation and navigation components, the sensors, etc.

The coupling system-environment consists of the sensed and actuated variables, or quantities: the measured distances to obstacles, the robot position and velocity,

etc. The position of an obstacle is quantity of the environment. However, it maintains a time-invariant input relation with the readings produced by the robot sensors, which are quantities of the coupling system-environment. The other way around occurs for the velocity command issued by the control system, which is a system output quantity in a time-invariant relation with the robot actual velocity.

All this corresponds to a simple analysis of the system with a low space-time resolution level. For example, the quantities in the coupling could be further decomposed to consider the motor voltage and current, and their time-invariant relations to the robot velocity and torque at the wheels.

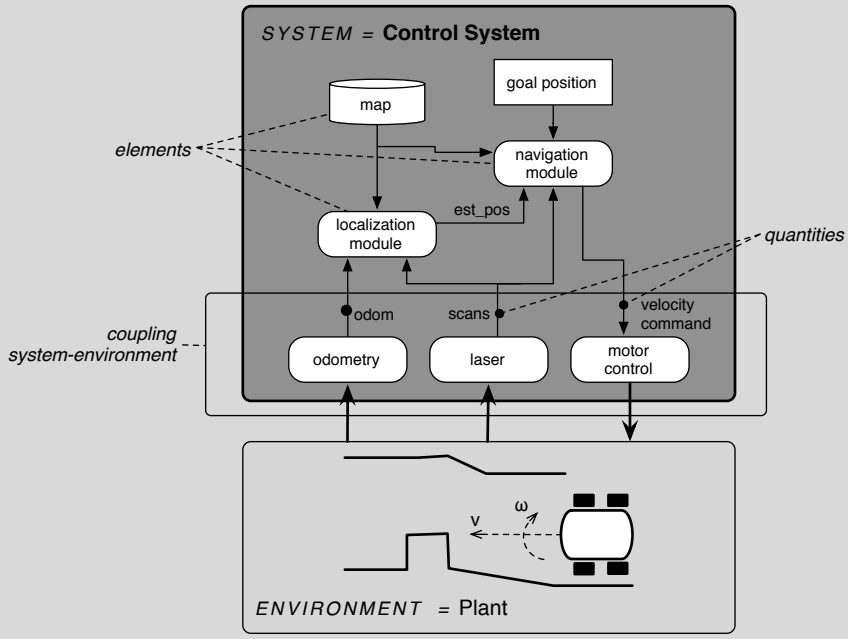


Figure 4.2: GST concepts applied to the autonomous mobile robot.

4.2.2 System Behaviour and Organisation

If we consider a particular system during a particular activity, we may observe that some of the time-invariant relations between its quantities may hold for a certain interval but eventually change. We shall say that these relations correspond to the *local* scope. Observing the same system during a different activity, we may observe that some of the time-invariant relations hold. If we again observe the system during a third activity, we could find that some of these relations would have changed. We would say they are of *relatively permanent*, for they hold for only some of the activities of the system. If we were to observe the system during an infinitely large number

of activities, we would find that a particular set of relations would always hold between its quantities. They would be *permanent*. Accordingly, we can distinguish three kinds of behaviour [76]:

- Permanent behaviour.
- Relatively permanent behaviour.
- Temporary behaviour.

The first may also be called *real behaviour*. The second, *known behaviour*. Temporary behaviour refers to the local scope, for it holds only for brief intervals within a particular activity.

We may observe that permanent and relatively permanent behaviour may not be clearly distinguished from each other when analysing systems. This is due to the impossibility to test the temporal persistence of relations beyond a restricted range of activities.

Let us return to the organisation of the system. We may realise that the different behaviours derive from different kinds of properties. We may distinguish two main kinds, which we shall call ***program*** and ***structure***. The temporary behaviour of a system derives from its program, which is the set of properties of local scope. Permanent and relatively permanent behaviours derive from the structure of the system, which we may in turn classify in ***real structure*** and ***hypothetic structure***, [76], so that the causal relations are as follows:

$$\begin{aligned} \text{organisation} &\longrightarrow \text{behaviour} \\ \\ \text{real structure} &\longrightarrow \text{permanent behaviour} \\ \text{hypothetic structure} &\longrightarrow \text{relatively permanent behaviour} \\ \text{program} &\longrightarrow \text{temporary behaviour} \end{aligned}$$

Example

Mobile robot organisation

Let us analyse the organisation of the state of the art controller for the mobile robot we have presented in 21. The components and their invariant algorithms and parameters form the real structure of the system. Those mechanisms that allow the activation and deactivation of routines during operation, such as fail-safe turning to remap obstacles for re-planning when no route can be calculated, correspond to the hypothetic structure. The instantaneous state of the memory containing the running computer program that instantiates the control components corresponds to the system's program.

4.3 Autonomous Systems

Once we have presented the basics concepts about systems that we will be using, we shall focus on the particular systems that are the focus of this work: autonomous systems. While systems are primarily defined in the universe as a pair <*system, environment*>, autonomous systems require a triplet for their definition:

<*system, environment, mission*>

Intuitively a system in a given environment is autonomous if it capable of achieving its objective(s) —the mission— there.

Our theoretical backbone thus will center around the concept of objective and its relationship to system's structure and behaviour in order to address the domain of autonomy, making use of the ideas developed in by López [90, pp. 103-150].

4.3.1 Directiveness

The behaviour of an efficient autonomous system is directed towards its objective, this being an specification of the state of the system, its environment or both. This ***directiveness*** or *finality* is the distinct trait of this kind of systems. According to [90] we can distinguish two kind of directiveness:

Structural directiveness stands for the fact that a particular organisation of the system results in a behaviour convergent to its current objective. This convergence may hold for small variations in the conditions of the environments, in the case the system presents some robustness to perturbations, but it will generally not hold for greater variations, for which the system must alter its organization to compensate. Structural directiveness depends on the system and its environment, the objective being implicit in the system (organization).

Purposive directiveness corresponds of the reconfiguration of the system organization that are dependent on the objective, so that the resulting behaviour shall be convergent to it. Since this reconfiguration is dependent on the particular objective, the processes in the system that produce it must to so operating with explicit representations of the objective of the system.

This a formulation of the sense of autonomy related to pursuing its own objectives commented in 1.2. Reformulating the challenges about coping with uncertainty from the environment and the system itself, and guaranteeing survivavility and dependability presented in 1.4, we can summarise them in three fundamental aspects of autonomy [90]:

1. System directiveness.
2. Minimum dependence of the system from its environment.
3. System cohesion.

4.3.2 Objectives

As previously mentioned, objectives are the central concept for characterising autonomous systems. Following [90], in close agreement to most of the literature on the subject, we may understand an ***objective*** as a

> *state of the system, of the environment or of both, to which the system tends as a result of its behaviour.*

It can be *complete* if it specifies all the aspects of the system and the environment, or *partial* if it refers only to some aspects, leaving the rest unspecified. A partial objective thus refers to a region of the state space rather than to a point.

According to our framework, the state of the system in a certain instant of time is given by the values of all its quantities at that time. The objective is always relative to the system[2], so if it refers to the environment it must do so according to the properties that are observable from the system, which in general corresponds to the quantities of the coupling.[3]

We shall distinguish two classes of objectives according to their morphology [90, pp. 117-119]:

explicit objective: there is typically a variable in the system representing that objective. It is a variable, so it can change. It is addressed by the program of the system.

implicit objective: there is no representation of it in the system, but the system directiveness makes it put pursue that objective. It is addressed usually by the system real structure, but we are building our system so that its hypothetical structure addresses it.

Example

Mobile robot objectives
In the control architecture of our mobile robot we can easily find explicit and implicit objectives. For example, the goal position which is the target reference for the navigation is an explicit objective. It can change depending on the goal commanded to the robot. Implicit in the architecture of the control system is the objective of obtaining scan reading with distance to obstacles in the environment. This is a permanent objective that is required for the operation of the system.

It is important to note here that in the case of an artificial system, all the objectives are explicit (or at least shall be) from its designer's perspective, being her choice to

[2] We could also consider, in the case of artificial systems, the designer objective, but for the moment we consider the objective from the perspective of the operating system.

[3] There is a particular case in which the objective can include specifications on the environment outside of the coupling, which is in the case that system can build conceptual representations of it, but we will discuss that later in section 4.4.

implement them in the system as explicit or implicit. This later decision is very related to the possibility of adaptivity regarding the achievement of the objectives.

Objectives can differ in other aspects, which can be classified in two main categories: time-scope, regarding the time necessary for the system to realise them, and level of abstraction. Both categories are related: high-level of abstraction objectives are usually associated to longer time-scopes.

Systems of a certain degree of complexity may operate concurrently at different levels of abstraction, showing a collection of objectives at each level. Usually, abstract objectives cannot be realized directly, and must be decomposed into a collection of more particular ones, and these into new ones in turn. This decomposition gives rise to a ***hierarchy of objectives***. The objectives in the lower levels of the hierarchy have a lower level of abstraction than those upwards the hierarchy. The objectives that are at the top of the hierarchy are the ***root objective***. They do not contribute to realise higher objectives. They represent the reification during operation of the top level functional-requirements for the autonomous system.

Objectives and Organisation

Objectives drive the composition of the system's properties, which leads to a corresponding behaviour. It can therefore be established the following relation of causality for autonomous systems:

$$\text{objective} \rightarrow \text{organisation} \rightarrow \text{behaviour}$$

We may realize that root objectives constitute a part of the definition of the system itself. In artificial systems they stand for the primary objectives of the designer. They underlie the longest time-scope of operation in the system and they establish the highest level of abstraction. They are a constitutional part of the system, as other fundamental properties, all of which form its *real structure*:

$$\text{root objectives} \rightarrow \text{real structure} \rightarrow \text{permanent behaviour}$$

As the root objectives, real structure and permanent behaviour are constant in time by definition; we may deduce that the adaptivity of the system relies on the rest of objectives, the hypothetic structure, the program, and correspondingly, the relatively permanent and temporary behaviours. We shall call these objectives *intermediate objectives*. *Local objectives* are the intermediate objectives of shortest scope. *Intermediate* and *local objectives* correspond to the *hypothetic structure* and to the *program* of the system respectively, as the *root objectives* correspond to the *real structure*:

$$\text{intermediate objectives} \rightarrow \text{hypothetic structure} \rightarrow \text{relatively p. behaviour}$$
$$\text{local objectives} \rightarrow \text{program} \rightarrow \text{temporary behaviour}$$

It must be remarked that, whereas Lopez's ontology of objectives in [90] accounts for the structure and dynamics of objectives, we restrain ourselves here to the static objectives of the autonomous system.

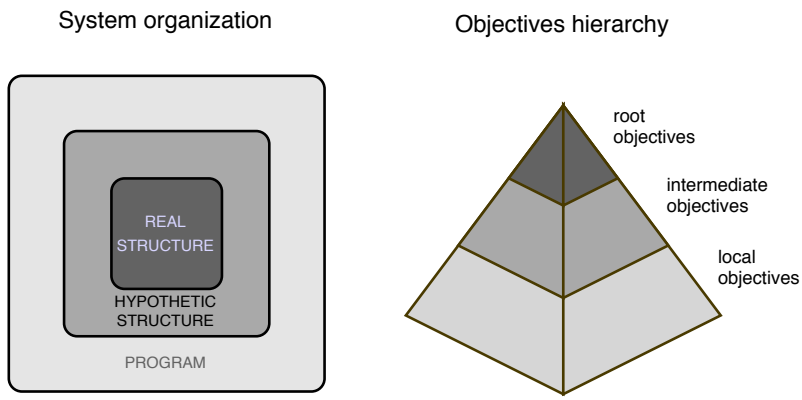


Figure 4.3: Correspondence between the hierarchy of objectives and system organisation.

In the next section we will address the problem of how system's directiveness is is realised in the system's organisation, by presenting our foundational concept of function.

4.3.3 Functions

The relation between an engineered system's purpose and its designed structure has been deeply discussed in functional modelling, as it has been presented in section 3.3. In this section we conceptualise that relationship according to our framework, relating it to the autonomous system's run-time operation.

We have already presented the concept of objective to refer to the purpose(s) of a system. In our framework, the concept of ***function*** can be attributed to the conceptualisation of how the system's directiveness is implemented in the system's organisation.

Lopez [90] argues that a first notion of function is a succession of states associated to a particular objective. A set of states and their transitions is a subprogram, and following that subprogram will move the system to the objective. This view considers the function from a behavioural standpoint. Let us analyse how the notion of function fits in the organisation of a system.

It can be the case, the subprogram corresponding to a function may not be followed at a certain time, but stored in the system knowledge. The function is then in *conceptual form*, or as an algorithm. There can be several algorithm to address the same objective. The process of assigning a function/algorithm to an objective is ***functional decomposition***. The result, the instantiation of the algorithm in the real quantities of the system, is the ***grounded function***. According to [90]:

> *Functional decomposition involves three aspects: objectives, resources and algorithms. An algorithm in the system knowledge is selected in order to realize a particular objective with specific system resources.*

When taking an operational/implementational view of functions, Lopez considers a ***function definition*** a conceptual entity that represents a complete specification of a functional decomposition in the current scenario of the running system.

The previous functional decomposition corresponds to the grounding of a single function capable of addressing an objective. However, it is usually the case that the function defined by the algorithm relies on other objectives to be also fulfiled, which are usually referred to as sub-objectives. Different function decompositions can require the same sub-objective. This allows for the reuse of the resources associated to the function that address this sub-objective.

This is how the system's operation can be decomposed from the root objectives into a hierarchy of objectives-functions, since it is the functions decompositions what accounts for the relationships between the objectives in the hierarchy.

Example

Functions and objectives in the mobile robot

We can consider the hierarchy of objectives in figure 4.4 for our mobile robot. The localisation function, whichis grounded in the localisation module, requires the sub-objective to obtain scan readings of obstacles. This sub-objective is also required by the trajectory planning function that addresses the objective to plan a trajectory.

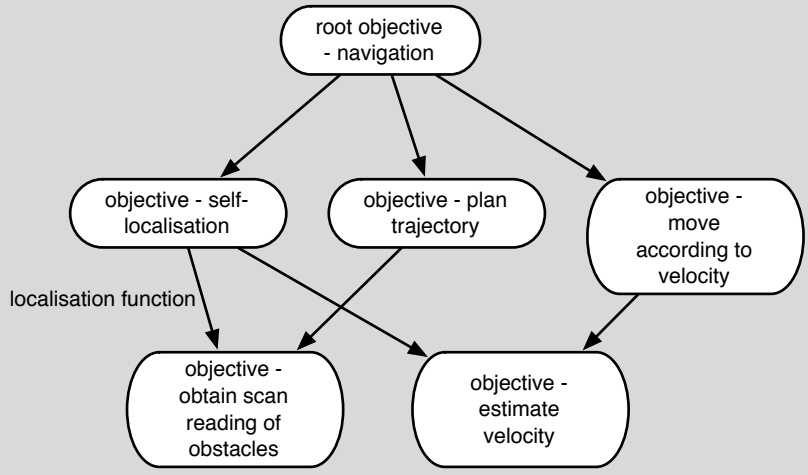


Figure 4.4: Example: mobile robot's hierarchy of objectives.

Abstract Functions and Function Versions

We could have different triplets <objective, algorithm, configuration> that share a common objective: we shall call these ***function versions*** of an ***abstract function*** defined by the objective. These versions can differ between themselves either in the algorithm, in the components configurations, or in both. This conceptualisation is similar to that proposed by Blanke et al. with the concept of *versions of services*[21, p. 77]. Two function decompositions that differ only in the resources correspond to physical redundancy: there are two different instances of the same function instantiated for the same objective. Two function decomposition that differ in the algorithm correspond to analytical redundancy: it means that there are two alternative function designs capable of addressing the same objective.

4.4 Cognitive Autonomous Systems

Once presented the concepts that explain from the GST perspective the issues at hand in autonomy, now it is time to move further to account for the cognitive phenomena in such systems. In this section we describe the relevant concepts of the theoretical framework that analyse cognitive aspects. They were gathered by Lopez in the Cognitive-Grounded System Model (CGSM) from [90], which is an ontology that explain global aspects of the cognitive operation in autonomous systems. These ideas have been further developed in the inception of the present research, and the results presented as the *General Cognitive System* conceptualisation in [64]. The basic idea of the CGS Model is to consider the system as a duality of a *grounded system* (GS), and a *cognitive system* (CS). Let us explain this.

We may assume that, in the most general case, a cognitive autonomous system operation can be analysed at two levels. The first one, which we may call *physical*, answers to physical laws: gravity, magnetism, etc. Indeed, an important part of the system's operation is its physical action on the environment; for example a robot picking up objects, or a mobile robot exploring new territory. This kind of operation can be observed by measuring a certain amount of *quantities*, representing speed, temperature, force, *etc.* These are the *physical quantities* we referred in section 4.2.1. The *grounded system* is formed by the physical quantities and their dynamics.

The other kind of operation in a general autonomous system is *conceptual*. A *conceptual quantity* is a specific resource of the system whose state represents the state of a different part of the universe [76]. For example, the area of memory used for an integer may represent the speed of a robotic mobile system, *encoded* in the state of its own bits. The part of the system that performs conceptual operation is called the *cognitive system*. The cognitive subsystem has the capacity to operate with conceptual quantities, using them for representing objects in their environment, for simulating the effect of its own action over them, or for inferring new objects among other examples.

We shall differentiate the abstract quantities from other conceptual quantities. Abstract quantities are not measurable and cannot relate to actual physical quantities.

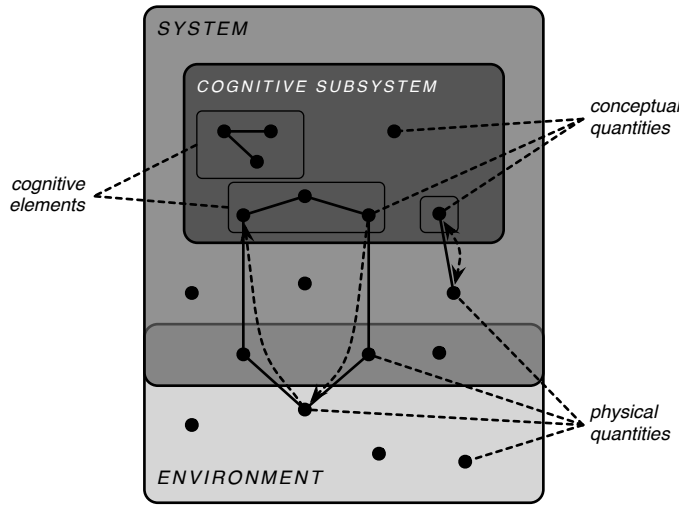


Figure 4.5: Grounded and cognitive systems and their quantities.

Between the rest of conceptual quantities there will be some that relate to real current physical quantities, we shall say they are *instantiated quantities*, and those that are not, but could eventually be: they are *potentially instantiated quantities*.

A model can be conceived as a set of conceptual quantities closely related, that is in a representation relationship to an entity (i.e. an element) in the world. Therefore, models or representations are elements in the cognitive subsystem. Those models that represent actual quantities in the environment or the grounded system are composed of instanted quantities, whereas those that represent hypothetic, predicted or simulated entities, consist of potentially instantiable quantities.

4.4.1 Conceptual Operation

Once we have introduced the organizational particularities of cognitive systems, we will now explain how are they involved in the operation of CS and GS. The processes of purposive directiveness are intrinsically part of CS operation, as they involve symbolic representations of system, environment, objectives, and conceptual processes. The result is a conceptual representation, which is then grounded into GS.

The relation between a conceptual quantity and its physical counterpart directly relates to the symbol grounding problem as analysed by [60]. Leaving out of the discussion the hard problem of meaning, we shall define the relation between the conceptual quantity (the virtual landscape) and the physical one it refers to (the actual landscape in the world) as the *grounding*. A conceptual quantity may refer to a physical quantity of the environment or a physical quantity of the system.

The bidirectional nature of the relation is represented by the *sensing-perception*

and *grounding-action* cycle. Sensing relates a physical quantity in the environment with a physical quantity in the system. Perception relates the physical quantity with a conceptual quantity in the cognitive subsystem. The physical quantity may be in the environment, in which case we shall talk about exteroception, or in the own system, then we shall talk about proprioception. We represent perception as a link between a physical quantity in the physical subsystem and a quantity in the cognitive subsystem because in any case the initial quantity must be mapped to one in the same substrate –embodiment– that the cognitive subsystem, that is the physical part of the system. This mapping is the sensing.

Grounding*[4] is the process of making physical quantities correspond to their conceptual counterparts. The other way round grounding* relates a conceptual quantity in the cognitive subsystem with a physical quantity in the system, while action relates the quantity in the system with the sensed physical quantity in the environment.

We call *embodiment* of a conceptual quantity to its physicalisation, that is to say the relation between the conceptual quantity and the physical quantity that supports it [84], in which it is *embodied*, i.e. in our example the relation between the robot speed and the memory bits used to represent it.

Example

Cognitive Operation related to velocity control
Let's take the example from a mobile robotics application. The speed of the robot (physical quantity of the coupling system-environment) is sensed in the signal from the encoder (physical quantity in the system) and through perception it is conceptualised in the conceptual quantity speed of the cognitive subsystem. This conceptual quantity may be manipulated in cognitive processes, such as planning, that result in an increase of the value of the quantity. The conceptual quantity is grounded through the quantity voltage applied to the motors, whose action finally results in an increase of the initial physical quantity speed. Only instantiated conceptual quantities can be updated through perception and/or grounded.

The conceptual quantity is embodied in the state of the RAM memory position that stores the variable of the control program in the robots on-board computer.

[4]This grounding is intimately related to the previous grounding, but we use them with a slight difference. Grounding refers to the whole relation between conceptual and physical quantity, whereas grounding* refers to it in the direction from conceptual to physical

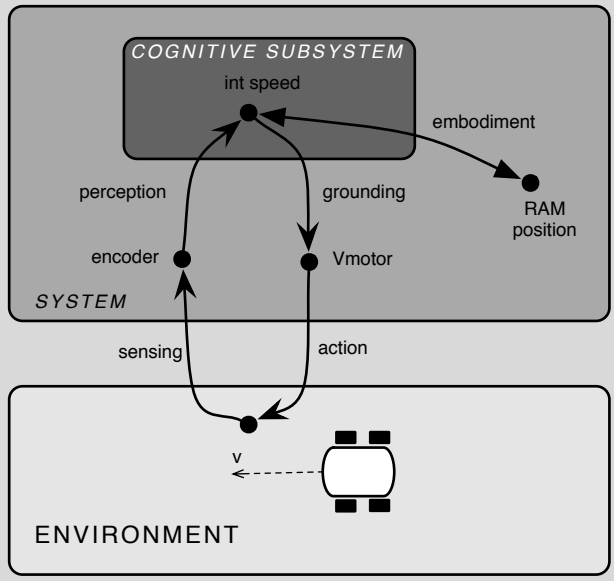


Figure 4.6: Schematic representation of the quantities involved in the cognitive operation related to the robot's speed.

As we described in 4.3.2, systems that present purposive directiveness operate with an explicit representation of their objectives. Hence they present conceptual operation that can be described with the aforementioned concepts. However, we will keep the adjective *cognitive* to define those systems in which conceptual operation involves representations not just of objectives, but of the two other elements in the triplet: the environment and the system itself.

The organisation of the knowledge of the system is key to determine the properties of its cognitive operation. The instantiated quantities correspond to the program of the cognitive subsystem: they account for the instantaneous representation of the current situation. The potentially instantiated quantities correspond to the hypothetic structure: when the activity in the cognitive system changes, new quantities can be instantiated, to represent the new situation. Finally, the abstract quantities correspond to the real structure: they determine which models the cognitive system operates with, and the algorithms it uses.

4.5 Analysing Cognitive Autonomous Systems

This theoretical conceptualisation is aimed at providing a common analysis framework for autonomous systems. In this last section we provide guidelines for the application

of the presented concepts to the analysis of autonomous systems, considering its organisational properties and its cognitive operation. These guidelines were compiled in a preliminary work to this research [64].

4.5.1 Autonomous operation: performance and adaptivity

Let us analyse how uncertainty affects the operation of the system in relation to its cohesion and directiveness. We have modelled uncertainty as a certain disturbance to the system, which, coming from the environment or from the system itself, can propagate in the organisation of the autonomous system, as displayed in figure 4.7

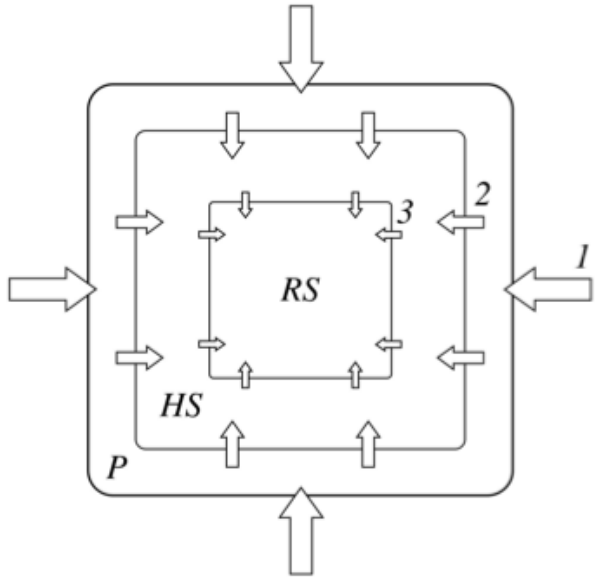


Figure 4.7: Propagation of disturbances in the organisation of an autonomous system (from [90]).

The system's program has a certain capacity to compensate these disturbances, mainly if they are intensive. The concept of ***performance*** has a very specific meaning in this context by referring to these capacities. Performance is therefore the effectiveness of the temporary behaviour of the system. However, performance may be not sufficient to cope with certain disturbances, typically the qualitative ones. In this case a program failure happens.

The consequences of a program failure may affect the hypothetic structure of the system (type 2 arrows in the figure). At this level, mechanisms of purposive directiveness may activate to try reconfiguring the system to correct its behaviour. This may consist of modifying algorithms or reconfigure a certain part of the structure of objectives. We shall call this capacity of the system ***adaptivity***. System's adaptivity can be

structural, in the case it is a function of the current functional structure, or purposive, in the case it develops dynamically. In the second case it implies conceptual operation. It may happens that system's adaptivity could not compensate the program's failure. We shall call this situation structural failure. Structural failure can propagate to the real structure of the system (type 3 arrows), breaking partially or totally system's cohesion, and affecting its directiveness towards the objectives.

Example

For example, in a varying parameters PID, while the plant remains in a certain region the controller parameters do not change, but the control signal does, according to the error. That corresponds to the program and performance of the system. By contrast, when the plant enters a different region of operation the PID parameters change accordingly, this stands for the hypothetic structure of the system and its adaptivity.

4.5.2 Principles of Autonomy

From the analysis of the system organization, Lopez's has enunciated some design guidelines, the *principles for autonomy*, which are a series of factors related to the provision of increased levels of adaptivity to a system. The most central for this work is the *principle of minimal structure*[92]

Principle of minimal structure: the structure of the system must be minimized for higher autonomy.

This stands, within the whole organisation for maximizing the system's program, which equals to maximize system performance. This is because more program provides better accuracy and resources more adapted to system resources. Secondly, within the structure, the principle stands for minimizing the real and maximizing the hypothetical structure. This equals to providing maximum adaptivity, because it means more reconfigurability.

In our extension to Lopez's foundation, maximizing hypothetic structure can be done by increasing the number of alternative function designs.

There are other principles for the design of autonomous systems derived and/or related to the basic principle of minimal structure. Lopez details them as follows:

Encapsulation: has two main aspects: i) minimization of the couplings between elements, ii) the construction of interfaces, in order to encapsulate heterogeneous elements. Minimization of couplings contributes to minimization of structure. Second, encapsulation favours reconfigurability. This is so because, by definition, *isolating a function within a module and providing a precise specification for the interface to the module* [29, p. 73].

Regarding modelling (including self-modelling) well-defined interfaces and isolated functionality allows traceability of interactions between modules.

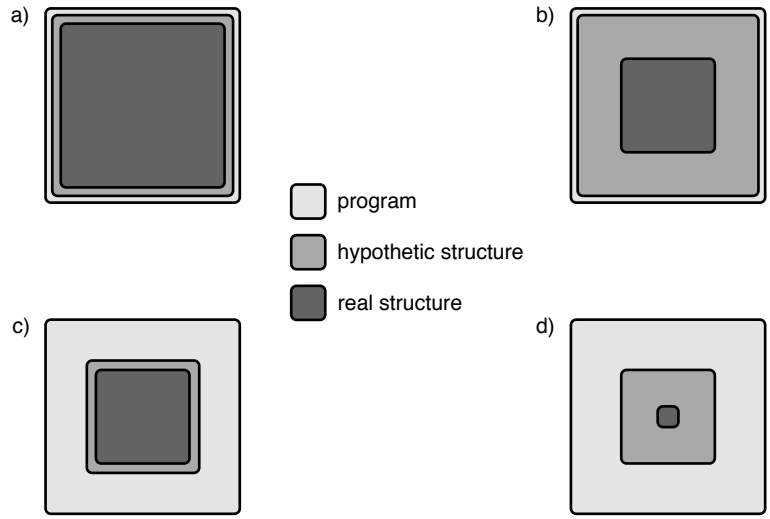


Figure 4.8: Evaluation of system organisation: system a) has almost only real structure, thus it would perform badly and could not adapt at all, any disturbance would affect the real structure, therefore leading to structural failure if it were not robust enough. System b) has great proportions of structure, both real and hypothetic, but no program at all and hence, despite it may be able to adapt and reconfigure itself for different situations, it has no means to address small disturbances and would perform poorly. On the other hand, system c) has no hypothetic structure: the system could not adapt to disturbances overwhelming the compensating capacity of program, so if they happened it lead to structural failure. The best organisation is that of system d), in which real structure is minimal and program and hypothetic structure maximal, resulting in a system with a priory good performance and adaptivity.

Therefore, the models of encapsulated systems are more easily integrated in mechanisms of directiveness such as functional decomposition or objective reconfiguration.

Homogeneity: Considering the elements and couplings of the system, similarity between is a factor for interchangeability. For heterogeneous elements and couplings, homogeneity can be achieved with additional elements enabling the indirect coupling between them: the interfaces.

Homogeneity represents increasing system efficiency, in the sense of optimizing the use of its resources. Similarly, homogeneity of knowledge maximizes its potential scope of use and power of representation.

Isotropy of knowledge: refers to the reusability of knowledge between different situations or contexts. Function definitions and objective decompositions are design knowledge that is generated for a particular scenario, we call this knowledge *biasing*. Perfect knowledge isotropy means that the knowledge and models can be applied independently of biasing, that is it is general knowledge that can be applied to many scenarios.

Scalability: is a factor that refers to the capacity of the system to grow. The principle of scale stands for maximizing the degrees of minimal structure, encapsulation, homogeneity and isotropy of knowledge by system growth.

4.5.3 Cognitive operation

It can be intuitively considered that quantitative uncertainty is accounted for by the systems performance, and qualitative uncertainty by adaptivity [90, p. 150]. Increasing knowledge mainly contributes to adaptivity, whereas increased physical resources contributes to performance. Since we are concerned with adaptivity in this work, our main focus is the maximization of the system's knowledge. If we enhance the system's knowledge by making its models more representative, then the system will be able to better account for qualitative uncertainty, increasing its adaptivity.

The organization of the cognitive system is an important factor regarding the capabilities of the autonomous system. Abstract quantities allow the system for generalisation, thus representation power, and inference and other deliberative processes that allow better knowledge exploitation. Potentially instantiated variables are necessary for planning and reflection over past situations. Instantiated quantities stands for models of current situation. More instantiated quantities means more modelling refinement of current state, whereas larger quantities of potentially instantiated quantities stands for greater capability of the architecture to represent different situations or the same situations with different models; we could talk about larger "model repository" (see figure 4.9).

Example

Knowledge in the mobile robot

Let us consider the knowledge embedded in the control system of our mobile robot. We could consider two different implementations of the navigation module and the map it exploits. In implementation 1, the control system uses only very detailed 2D Cartesian representations of the environment. The organisation of the knowledge thus involves a big portion of instantiated quantities, to account for the instantaneous representations of obstacles. The different maps stored in the map server for the different areas the robot can traverse correspond to the potentially instantiable conceptual quantities. The definition of the 2D coordinates and their occupancy state by obstacles correspond to the abstract quantities that for the real structure of the system's knowledge. The organisation of the cognitive subsystem for implementation 1 is depicted in the upper part of figure 4.9.

We could consider a different implementation 2, with more diverse modelling capacity in the system, using integrated representation consisting of occupancy grip maps optimised into quadtrees or potential maps for local mapping, and graphs to capture the relations between mapped areas. Using these more simple representations, the amount of conceptual quantities instantiated to account for the instantaneous modelling of the environment is smaller in this implementation 2. The same applies to potentially instantiable quantities, given the simplified information stored in quadtrees, when compared to 2D Cartesian maps. However, the cognitive subsystem has now a greater amount of abstract quantities, given the many different conceptualisation of space it handles.

Implementation 1 provides a better performance: more precise maps mean more efficient and accurate navigation. However, this navigation can consume too much computer memory, and be too slow in cluttered environments, where there are many obstacles. Implementation 2 allows the control system to adapt to this circumstances by changing from one representation and corresponding algorithms for navigation to another more suitable for the environment.

Example

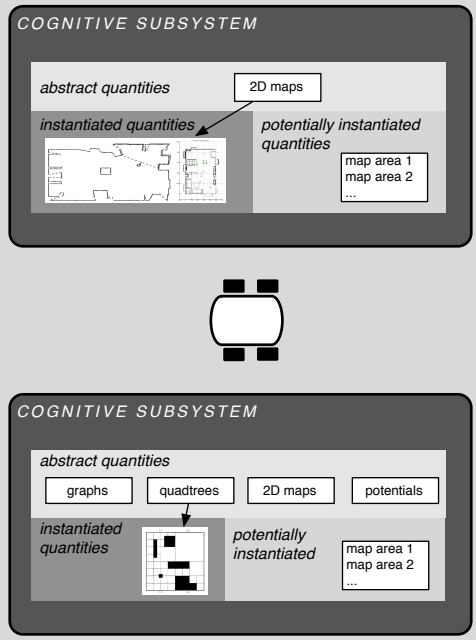


Figure 4.9: In a cognitive system, different proportions of conceptual quantities stand for different modelling properties.

Chapter 5

State of the Art of Self-Aware Systems

As it has been discussed in the introduction, a diversity of techniques have been explored to tackle the problem of uncertainty when building robust autonomous systems. In the following sections we present the state of the art in different approaches that target system's adaptivity to overcome the problem. They come from different subdisciplines, but they share a common objective with the present work. The current research is highly inspired by these approaches.

5.1 Autonomous Supervisor for fault-tolerant control

As it has been discussed in section 3.5.2, fault-tolerant control addresses the problem of faults (i.e. internal uncertainty) in control systems, and using methods rooted in control theory. Blanke et al. [21, pp. 612–618] have proposed guidelines for the design of fault-tolerant control. It involves the implementation of an Autonomous Supervisor architecture in the control system (figure 5.1).

The architecture comprises four levels:

1. The lower level consists of the traditional control loop, I/O and references. Fault-tolerance includes an additional block that allows to validate and filter if required the input signals to the controller, i.e. references or setpoints and sensory input.
2. The second level includes algorithms for fault diagnosis in the Detectors, and Effectors to perform the remedial actions for controller re-design.
3. The third level is the supervisor logic, which determines the most appropriate remedial action from the diagnosed state.

4. The fourth level includes plant-wide control and coordination.

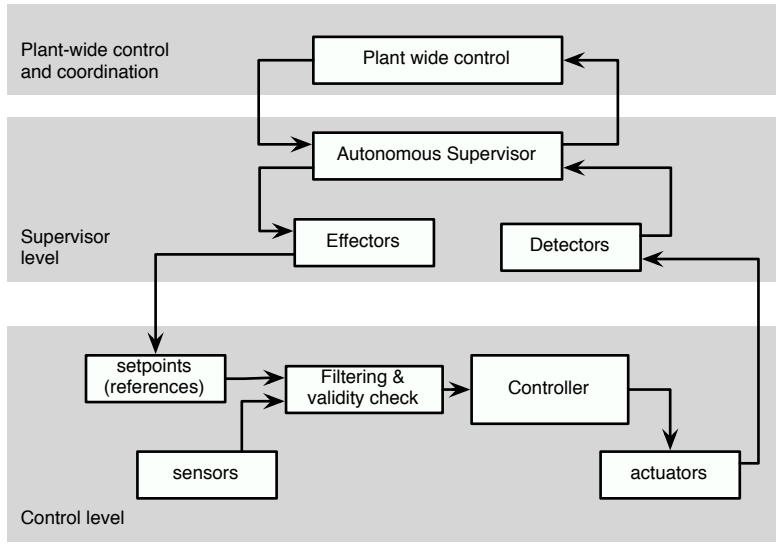


Figure 5.1: The Autonomous Supervisor architecture, adapted from [21].

The design procedure uses the analysis of fault propagation and structure to develop the fault diagnosis and controller re-design actions performed at the supervision level.

The component based analysis relies on the component model of the system presented in 56. It results in a list of component related faults and their effects. The structural analysis provides information on the redundancy available in the system for diagnosis and reconfiguration.

The supervisor logic consists of inference rules developed using the previous information about the known faults and their effects. These rules encode the knowledge of which remedial action must be taken according to the system fault state diagnosed.

5.1.1 Fault Diagnosis

The information obtained by the diagnosis should be used for the controller re-design, so it cannot only indicate when a fault occurs, but also identify its location and magnitude. The successive steps are:

1. Fault detection, decide if a fault occurred and determine the time at which happened.
2. Fault isolation finds the location of the fault, i.e. in which component it occurred.
3. Fault identification and estimation of its magnitude.

Different techniques can be used for fault diagnosis, depending on the nature of the system (continuous-variable, discrete, ...), as well as different architectural options: embedded, distributed, decentralised, coordinated. An integrated approach for fault analysis and system reconfigurability has been proposed in [54]. It consists of a self-updating model based on functional analysis that captures the availability of services provided by the systems components in a state transition graph.

5.1.2 Controller re-design

The objective of the controller re-design is to change the controller when a fault occurs so it continues to satisfy the requirements. Two ways of doing so can be distinguished:

Fault accommodation implies changing the controller parameters to comply with the new dynamical model of the faulty plant. This can be achieved by switching to pre-designed controller configurations selected off-line for each fault.

Control reconfiguration if the previous is not a solution, the whole control loop, including its structure, has to be changed. Alternative input and output is used for the controller and a new control law has to be designed on-line. This is typically the case with sensor and actuator failures, in which alternative components have to be found to replace the faulty ones.

There can be different possible remedial actions for each of the listed faults, making use of physical or analytical redundancy depending on the structural analysis. Some of them may not necessarily involve control re-design.

- Switch to another version of service with full performance, if redundant hardware is available.
- Change to a version that those not require the faulty component, even if with degraded performance. This is the case of replacing a sensor by an observer that does not used its faulty measurement, or considering controllability without a faulty actuator, using only the remaining.
- Control re-design. An online control re-design may be required when the remedial actions depend on the state of the system. Autonomous controller re-design is a challenging solution equivalent in complexity to adaptive control.
- Change objective, when other possibilities are exhausted, relax the performance objectives and design an appropriate controller for the faulty system.
- When appropriate fault handling cannot be achieved, the supervisor should command the system to a safe state.

5.1.3 Analysis

The Supervisor architecture for fault-tolerant control improves the adaptivity over a standard control architecture, because it converts the control level from ***real structure***

into ***hypothetic structure***. The amount of hypothetic structure in the system, and thus the scope of the system's adaptivity, depends on the re-design alternatives available, that is the analytical redundancy of the system. The amount analytical redundancy depends not only on the number of alternative designs for a certain part of the control system, but also on the granularity at which alternative does exist. If the supervisor is limited to switch between predetermined alternative designs for the whole control system, the hypothetic structure is limited. However, if these alternatives are not at the system level, but at the component/service level, the possibilities grow exponentially, as does the hypothetic structure.

Considering the Supervisor level from a cognitive perspective, the knowledge is embedded in the supervisor logic. If it is based on rules, thus not being an explicit model of the control level structure of components, that means small amount of abstract quantities and isotropy of knowledge. This reduces the potential model repository and limits the scenarios and circumstances to which the supervisor can adapt the system for.

5.2 Self-adaptive software

In software systems a lot of research has been done for dynamic adaptation in relation to fault-tolerance. Traditional self-repair has been conducted at the code level [53]. These mechanisms, such as exception handling or timeouts, have become standard coding techniques, well supported by modern programming languages (e.g. Java exceptions) and programming libraries. However, they are pre-defined at development time, so they cannot account for unpredicted phenomena, neither can they take into consideration run-time information to determine the source of the failure and take the most appropriate corrective measure.

During the last decades different approaches have been explored to empower computing systems with adaptive mechanisms at run-time, moving the solution from the code-level to the module-level, be it a component or a service. Multi-agent systems from AI are an example of this. More industry and business approaches following this line are dynamic architectures in service-based software systems and, more recently, the autonomic computing approach. Following we describe exemplary research works of both lines that are relevent to this work.

5.2.1 Dynamic architectures

Garlan et al. [53] proposed an approach in which an stylized architectural design model of the system is maintained at run time and used to detect when the system's behaviour departs from the desired range and thus decide high-level repair strategies.

The cornerstone in their approach is the architecture model, for which they adopted a simple metamodel (shared by the core of most architectural description languages) that represent the system architecture as an annotated, hierarchical graph. Nodes are

components and arcs are *connectors*. Annotations are lists of *properties* that account for the semantic properties (e.g. bandwidth of a connector, load in a component for computing)

A key in their approach is the concept of architectural style, which:

> defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed.

Typical styles are *client-server* or *publish-subscribe*, and they have the advantage of mapping well to many component integration infrastructures or platforms, such as CORBA or EJB.

Their innovation is to use the style not only at design time, but to support run time adaptation by incorporating it to the knowledge in the form of a *stilised architectural model* used by the repair mechanisms.

The style is thus the knowledge that is used to determine: a) what properties of the system to monitor, b) what constraints to evaluate, c) what to do when constraints are violated, d) how to repair.

The complete Adaptation Framework they have proposed works as illustrated in figure 5.2.

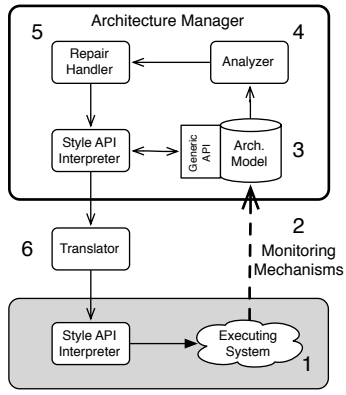


Figure 5.2: Garlan's et al. Adaptation Framework (adapted from [53]).

> an executing system (1) is monitored to observe its run time behavior; (2) monitored values are abstracted and related to architectural properties of an architectural model; (3) changing properties of the architectural model trigger architectural analysis to determine whether the system is operating within an envelope of acceptable ranges; (4) unacceptable operation causes repairs, which (5) adapt the architecture; (6) architectural changes are propagated to the running system.

Requirements for applying this: the system (its impplementation platform) must be reconfigurable (e.g. possibility to redirect communications, to load dynamic libraries, to do dynamic resource management).

Application of the framework:

- Define an architectural style - this is re-usable for other systems with the same style.
  - Define the adaptation operators (actions to change the architecture)
  - Define the repair strategies.
- Define the architecture model, including the initial configuration of the system.
- Connect the framework to the system:
  - Style-based monitoring: probes (implementation dependent) and gauges, model-specific (reusable in other system with the same style)
  - Map adaptation operators to implementation operators, which are offered by the Runtime Manager.

Another and more recent example of dynamic software architectures is that of Fiadeiro & Lopes [48], who presented a model for dynamic reconfiguration in service-based software architectures. Their model can be considered a metamodel to describe the architecture of service oriented systems. The architecture of these systems has the particularity of not being determined at design time.

5.2.2 Autonomic Computing

In the recent years an initiative by IBM was launched to address some of the issues discussed in 1.4 in the realm of software and computing systems.

The overarching goal of autonomic computing is to realize computer and software systems and applications that can manage themselves in accordance with high-level guidance from humans.

It is based on the example of the human autonomic nervous system, which together with the endocrine system is constantly regulating and maintaining homeostasis [110]. This natural system is constantly monitoring changes and disturbances in our body and generates correcting action of corresponding size to keep our internal balance, by specific and dedicated but interdependent regulation mechanisms.

The essential issue in autonomic computing is **self-management**[75], by which system's administrators are made free of most details of the system's operation and maintenance, labours now addressed by the system itself, thus implementing a high level of autonomy. There are several aspects considered in self-management:

- **Self-configuration**: autonomic systems will be able to configure themselves according with high-level policies. That will permit new components to integrate effortless in extant systems provided both incorporate autonomic technology:

the new component will configure itself for the system, which in turn will adapt to it.

- **Self-optimization**: an autonomic application will be continuously detecting opportunities to tune itself for a more efficient operation, i.e. they will automatically seek for updates to improve its execution.
- **Self-healing**: autonomic systems will have unseen fault tolerant capabilities, being able to analyse the root of eventual failures and smoothly recover from them.
- **Self-protection**: an autonomic system will be able not only to defend from detected external attack, but to proactively anticipate problems based on early monitoring information.

Autonomic applications and systems will exhibit these characteristic at many levels of granularity. An autonomic system will consist of a myriad of *autonomic elements*, each one providing for some services to users or other elements. The self-management will arise both from the interaction between the autonomic elements, supported by a distributed service-oriented infrastructure, as well as from the internal autonomic capabilities of each one.

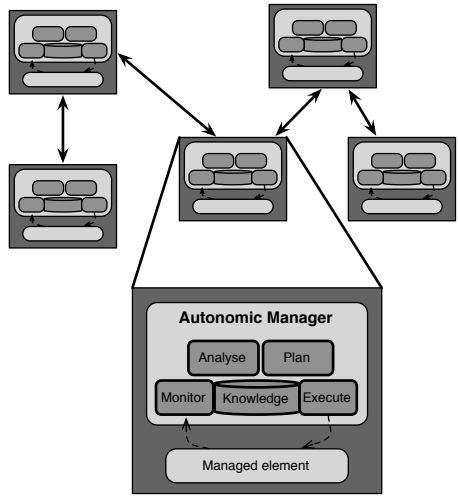


Figure 5.3: An autonomic system consists of a set of autonomic elements interacting. Each element is formed by an Autonomic Manager and a managed element (from [75]).

As shown in 5.3 an autonomic element consists of an autonomic manager that controls one or more managed elements. This later maybe any regular element in a computing system, either hardware: a hard disk, a CPU, or software: a database. At higher levels the managed element could be a whole service application. It is expected that eventually the autonomic manager will be completely fused with the managed

element in implementation, being the distinction merely conceptual.

At the lower levels, autonomic elements will have a limited range of internal behaviours, mainly hard-coded and fixed, to provide for adaptation. At higher levels more flexibility is needed, and this behaviour will be more dynamic and undefined at design-time, defined in goal-oriented terms, being the details resolved on the fly by the autonomic element in its interactions with others and the user.

Reviewing the aspects listed in 5.2.2, we may observe that all they need for two underlying capabilities:

self-awareness: since the autonomic manager relies on knowledge on the managed element, its behaviour and state.

context-awareness: it has to be continuously monitoring its environment to be able to react to changes

5.2.3 OMACS and adaptive multi-agents organisations

Oyenan, DeLoach et al. [43, 108] have proposed a multiagent framework to develop autonomic systems. They argue that agents are autonomous and map naturally with autonomic computing principles. Their framework allows to build multiagent systems that are capable to adapt to unpredicted situations in their environment by reconfiguring its organisation at runtime. They claim that their approach provides for the demanded systematicity and repeatability, opposite to other ad-hoc approaches, because it is based on a formal framework (OMACS) and is supported by a multiagent development process.

The core of the approach is the Organization Model for Adaptive Computational Systems (OMACS), which is a metamodel for artificial organizations, i.e. multiagent or autonomic systems. It defines the knowledge required for the system to be able to self-organize at runtime. OMACS defines an **organisation** as [108]:

> a set of goals that the system is attempting to accomplish, a set of roles that must be played to achieve those goals, a set of capabilities required to play those roles, and a set of agents who are assigned to roles in order to achieve organization goals.

Basically, an OMACS system or organization is an instance of the OMACS metamodel presented in Figure 5.4 that fulfils all the constraints defined in the metamodel.

Using the previous information about its organisation, an OMACS system is able to reason about its state in terms of the achievement of goals and self-configure according to its capabilities to improve it if needed.

A model-driven development process for agent-based autonomic systems supports the approach. The process involves a set of tasks to produce, departing from the system's requirements, the different models (Goal, Roles, Agents, etc.) compliant with the OMACS metamodel that specify the design of the system.

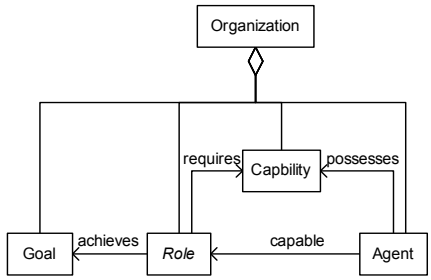


Figure 5.4: Simplified version of the OMACS metamodel, from [108].

The framework is complemented with the Organization-based Agent Architecture (see fig. 5.5). This agent architecture defines two parts of the agents: the Execution Component, which would correspond to the traditional agent performing application tasks, and the Control Component (CC), which is the autonomic part. The CC uses the model or specification of the organisation according to OMACS to reason about the state of the system and take appropriate reconfiguration measures. Figure 5.5 depicts the typical implementation of an OMACS system, with a centralised design of the autonomic part in the Organization Master, to which all the CCs of the agents report. The OM then decides the next appropriate configuration (agent-role-goal assignments) and

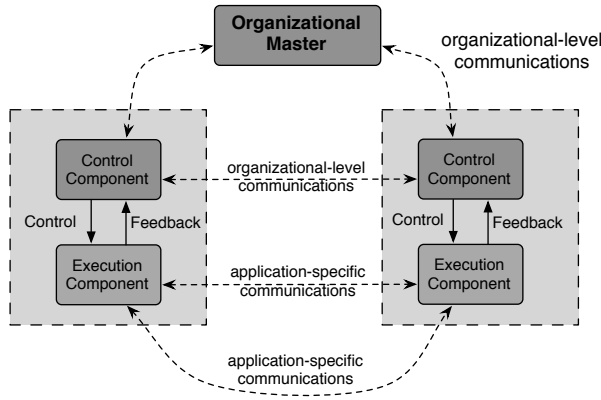


Figure 5.5: The Organization-based Agent Architecture used in most OMACS exemplary systems (adapted from [108]).

Analysis

The previous approaches for software systems address adaptivity at the architectural level rather than at the code level. If we apply the principle of minimal structure, we can argue that this increases the amount of hypothetic structure of the system in relation to its program and real structure.

For example, in OMACS, the possibility of runtime re-organization of the agents, i.e. control re-configuration, means that the organization of agents corresponds to the hypothetic structure, instead of being the real structure if it were a classical fixed organization of agents. This provides the system with improved adaptivity, which is limited by all the capabilities of the agents (the real structure in this case) as a set, and not to the more limited number of capabilities of a certain organization.

From the cognitive perspective of autonomous systems, the OMACS approach is totally knowledge-based, with the definition of a metamodel for agent organizations. The more rich the OMACS model of a sytem is, regarding information about agents capabilities and heuristics to assign agents to goals, i.e. potentially instantiable quantities, the better the capability to adapt of the system will be.

5.3 Cognitive Architectures

Cognitive architectures is an interdisciplinary research area in which converge the fields of artificial intelligence, cognitive psychology/cognitive science, neuroscience and philosophy of mind. A cognitive architecture is a blueprint for intelligent agents. It proposes (artificial) computational processes that act like certain cognitive systems, most often, like a person, or that act intelligent under some definition. Cognitive architectures form a subset of general agent architectures. The term architecture implies an approach that attempts to model not only behavior, but also structural properties of the modelled system.

We shall distinguish three main categories of cognitive architectures according to their purpose:

- **Architectures that model human cognition.** One of the mainstreams in cognitive science is producing a complete theory of human mind integrating all the partial models, for example about memory, vision or learning, that have been produced. These architectures are based upon data and experiments from psychology or neurophysiology, and tested upon new breakthroughs. Examples of this type of cognitive architectures are ACT-R [7] and Atlantis. These architectures do not limit themselves to be theoretical models, and have also practical application, i.e. ACT-R is applied in software based learning systems: the Cognitive Tutors for Mathematics, that are used in thousands of schools across the United States.
- **Architectures that model general intelligence**. These are related to the first ones but, despite of also being based upon the human mind (as the only agreed intelligent system up to date), do not constraint to explain the human mind in its actual physiological implementation. They address the subject of general intelligent agents, mainly from a problem-solving based perspective. Example of these architectures are Soar [82] and BB1.
- **Architectures to develop intelligent control systems**. These architectures take a more engineering perspective, and although they also address the general in-

telligence problem, they focus on applying it the building of technical systems. They are intended as more powerful controllers for systems in real environments, and are mainly applied in robotics and UAV's and UGV's [1]. Some examples of these architectures are 4D/RCS [1] and Subsumption [25], despite some debate on the last one about if it can be considered "cognitive".

This research falls mainly in the third category of cognitive architectures, of using them to build controllers with higher degrees of autonomy. Notwithstanding, an analysis of the other two classes was conveyed, to explore and study artificial implementations of cognitive traits, specially those related to self-awareness.

5.3.1 Classification of cognitive architectures

As discussed in the introduction (page 15) for AI, cognitive architectures can be classified according to different criteria:

Connectionist vs Symbolic

Cognitive architectures can be divided between the two main paradigms that exists in these fields:

Connectionist approach: The central connectionist principle is that mental phenomena can be described by **interconnected networks of simple units**. The form of the connections and the units can vary from model to model. For example, units in the network could represent neurons and the connections could represent synapses. Another model might make each unit in the network a word, and each connection an indication of semantic similarity.

Computationalism or symbolism: the computational theory of mind is the view that the human mind is best conceived as an **information processing system** very similar to or identical with a digital computer. In other words, thought is a kind of computation performed by a self-reconfigurable hardware (the brain).

There are of course *hybrid* architectures that have a part of each paradigm, such as ACT-R, with its symbolic and subsymbolic levels. Recently, Soar has included in its latest version (Soar 9) connectionist mechanisms for learning.

Deliberative vs reactive

Another classification related to the field of intelligent agents distinguish between deliberative and reactive architectures.

Deliberative architectures. These architectures come from the GOFAI (Good Old-Fashioned Artificial Intelligence) paradigm. The working of a deliberative agent

[1] UAV: Unmanned Aerial Vehicle
UGV: Unmanned Ground Vehicle

can be described in terms of a sense→model→plan→act cycle. The sensors sense the environment and produce sensor-data that is used to update the world model. The world model is then used by the planner to decide which actions to take. These decisions serve as input to the plan executor which commands the effectors to actually carry out the actions.

Reactive architectures. Reactive architectures appeared in the 80's in opposition to GOFAI, claiming that there is no need of representation of the world for an intelligent agent having the own world at disposal [24]. Reactive architectures are designed to make systems act in response to their environment. So instead of doing world-modeling and planning, the agents should just have a collection of simple behavioral schemes which react to changes in the environment in a stimulus-response fashion. The reference for reactive architectures is Brooks' Subsumption architecture [25].

There are also in this case hybrid architectures that combine both reflective and reactive capabilities, like RCS or ATLANTIS. In fact, for a cognitive architecture to be useful for real-time control systems the hybrid approach seems not only appropriate but necessary.

It is also remarkable that the symbolic paradigm is strongly related to deliberative architectures and the connectionist with the reactive approach, despite there is a full gradation between the extremes and in practice most architectures used in real systems, and not only simulated environments, are hybrids to some extent.

5.3.2 RCS

RCS is a cognitive architecture that reifies Albus theory of cognition [2]. However, it is also a Reference Model Architecture, suitable for many software-intensive, real-time control problem domains.

RCS defines a control model based on a hierarchy of *nodes*. All the control nodes at all levels share a generic node model. The different levels of the hierarchy of a RCS architecture represent *different levels of resolution*. This means that going up in the hierarchy implies loss of detail of representation ad broader scopes both in space and time together with a higher level of abstraction 5.6. The lower level in the RCS hierarchy is connected to the sensors and actuators of the system. The nodes are interconnected both vertically through the levels and horizontally within the same level via a *communication system*.

RCS Node

The RCS node is an organisational unit of a RCS system that processes sensory information, computes values, maintains a world model, generates predictions, formulates plans, and executes tasks. The RCS node is composed of the following modules: a *sensory processing* module (SP), a *world modelling* module (WM) together a *behaviour*

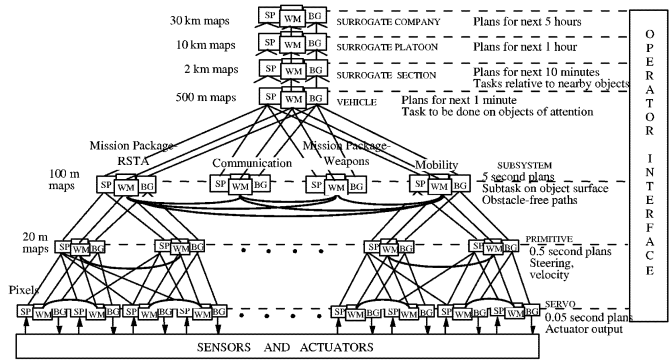


Figure 5.6: Example of a RCS hierarchy from [1], in which the different resolution levels can be appreciated.

generation module (BG) and a *value judgement* module (VJ). Associated with each node there is also a *knowledge database* (KD). Figure 5.7 illustrates the elements and their relations within the node.

Queries and task status are communicated from BG modules to WM modules. Retrievals of information are communicated from WM modules back to the BG modules making the queries. Predicted sensory data is communicated from WM modules to SP modules. Updates to the world model are communicated from SP to WM modules. Observed entities, events, and situations are communicated from SP to VJ modules. Values assigned to the world model representations of these entities, events, and situations are communicated from VJ to WM modules. Hypothesised plans are communicated from BG to WM modules. Results are communicated from WM to VJ modules. Evaluations are communicated from VJ modules back to the BG modules that hypothesised the plans.

5.3.3 Soar

Soar (which stands for State, Operator And Result) is a general cognitive architecture for developing systems that exhibit intelligent behavior. It is defined as *general* by its creators so as to emphasize that Soar does not only intend to address the problem of human cognition, but that of intelligence as a universal phenomenon. The Soar project was started at Carnegie Mellon University by Newell, Laird and Rosenbloom as a testbed for Newell's theories of cognition.

Soar is designed based on the hypothesis that all deliberate *goal-oriented* behavior can be cast as the selection and application of *operators* to a *state*. A *state* is a representation of the current problem-solving situation; an *operator* transforms a state (makes changes to the representation); and a *goal* is a desired outcome of the problem-solving activity.

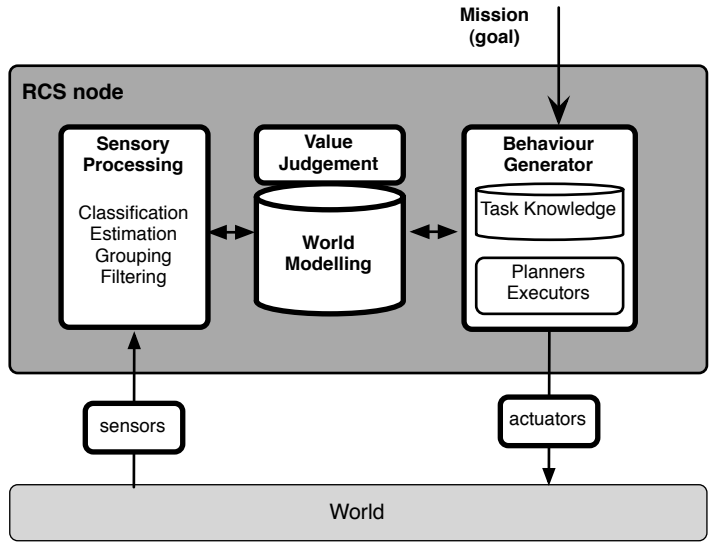


Figure 5.7: Functional structure of a RCS node, adapted from [3].

The functioning of Soar is based on a sequence of actions which is called the *Execution Cycle* and is running continuously. Soar's memory is a production system, and consist of three differentiated modules: *working memory*, which represents current state, results of intermediate inferences, active goals and active operators, *production memory*, where Soar stores long-term knowledge mainly procedural, and *preference memory*.

5.3.4 Machine Consciousness Architectures

In the recent decade a new field is gaining momentum, at the convergence of cognitive architectures, scientific research of consciousness and self-awareness, and cognitive robotics. This new research topic is *machine consciousness* [69].

Even though there are old arguments against the possibility of machine consciousness [99], several attempts at realizations of machine consciousness have been made recently. In some cases, these systems propose a concrete theory of consciousness explicitly addressing artificial agents [34, 58]. In other cases, the implementations follow psychological or neural models of human consciousness. This is true, for example, in the case of the many implementations of Baars' Global Workspace Theory (GWT) of consciousness [50, 143, 10].

However, the multifarious character of consciousness is an obvious problem [22], which most of the approaches circumvent by focusing on just one aspect of it. Access consciousness seems to be the main target, leaving phenomenality to further clarifications of the hard problem [32] by philosophers and cognitive scientists.

Chella et al. [34] have developed a robot cognitive architecture in which self-consciousness is based on higher order perception of the robot, in the sense that first-order robot perception is the immediate perception of the outer world, while higher order perception is the perception of the inner world of the robot.

Arrabales' CERA-CRANIUM cognitive architecture is a reification of Baars' Global Workspace theory of consciousness for software agents. CERA architecture is a software architecture that allows the integration of different cognitive components into a single autonomous system. CRANIUM (Cognitive Robotics Architecture Neurologically Inspired Underlying Manager) is a tool for the creation and management of large amounts of parallel processes in shared workspaces; it can be considered an implementation of a Dennet's pandemonium. Arrabales has used it to implement Baars Global Workspace.

LIDA Architecture

Franklin et al. LIDA[2] architecture [50] is a paradigmatic case of cognitive architecture for machine consciousness.

LIDA provides a conceptual (and computational) model of cognition implemented as a software agent. The IDA model implements and fleshes out Global Workspace theory. The LIDA implementation of GW theory yields a fine grained functional account of the steps involved in perception, several kinds of memory, consciousness, context setting, and action selection.

The LIDA codelets are specialized programs that monitor the occurrence of a particular event that may require a conscious intervention. When such an event occurs, these codelets form coalition with other codelets that contain information about the situation. The coalition then competes to be placed under the focus of attention (spotlight). If the coalition wins, its contents spread to other codelets coming.

Cognitive processing in IDA consists of continually repeated traversals through the steps of a cognitive cycle. Incoming sensory stimuli is filtered through preconscious perception, where meaning is added and a percept produced, and then enhanced with information from un-decayed percepts from previous cycles. The current structure from working memory cues transient episodic memory and declarative memory, producing local associations, which are stored in long-term working memory. The conscious broadcast (ala Gobal Workspace Theory) occurs, enabling various forms of learning and the recruitment of internal resources. Finally Procedural memory responds to the contents of the conscious broadcast instantiating action schemes that matches them, and the action selection mechanism chooses an action from one of them for this cognitive cycle. Different learning mechanisms take place using a variety of activation nets models at the perceptual, episodic, procedural and attentional phases of the cycle.

[2]The initially named IDA architecture is, since the inclusion of learning mechanisms, the LIDA architecture.

5.3.5 Analysis

Cognitive architectures that model intelligence, such as Soar and ACT-R, provide design patterns for some intelligent capabilities, such as inference and learning, but they do not address the complete description of patterns and functional decomposition required in an architecture for building complex control systems, such as appropriate I/O interfaces, distribution a encapsulation in computational separate components, etc. For example, dedicated modules or interfaces for sensing and acting must be implemented ad-hoc in Soar.

On the other hand, architectures for intelligent control such as RCS provide a good methodology to address the construction of a whole control system. Arrabales CERA-CRANIUM architecture also provides architectural guidelines to integrate sensors and actuators in the lower layer of CERA.

This is closely related to the grounding of cognitive operation discussed in page 80. Soar does not provide a solution for grounding conceptual operation. On the other hand, RCS provides specific design patterns for perception and grounding in a hierarchical manner.

Architectures for general intelligence such as Soar and ACT-R provide good tools for the cognitive operation, specially decision-making and planning.

Cognitive architectures organisation

Let us analyse the properties of cognitive architectures from the standpoint of the organisation of the cognitive autonomous systems built with them, using the criteria presented in section 4.5.

Organisation. In a RCS system the number of nodes and their layering are established at design time, so they correspond to the system's real structure and do not change during operation. However, the connections between nodes for a mission, that is, the command tree structure, is determined by the system itself at runtime. Since the command tree hold for long periods of time —a mission or a task— it corresponds to the hypothetic structure, allowing the system to adapt to different missions.

Encapsulation. RCS encapsulates control according to the abstraction level in nodes. The architecture comes with an engineering methodology to systematically ad- dress the encapsulation of control and cognitive functions. Interfaces between nodes are clearly defined: bottom-up through SP modules and top-down through BG modules.

Self is the key aspect that RCS fails to address, in an otherwise suitable architecture to develop cognitive controllers. The absence of explicit representation of lifetime objectives and the lack of self-modelling prevent a system built with RCS from being able to monitor its own operation and reconfigure it at the functional level.

Part III

The OM Architectural Framework

Chapter 6

Model-based Self-Aware Cognitive Control

If we view the development of this thesis as an engineering process, so far we have presented the results corresponding to the analysis phase. We have studied the extant solutions related to the problem of building autonomous self-aware control systems, and we have characterised the problem with the theoretical framework we have compiled. Now we shall introduce the solution designed for the building of self-aware autonomous systems. This chapter presents the main thesis of this work. The first section discusses the basic ideas and principles from which we have departed, before discussing our solution in the central section. It is described at an abstract level here, stated as a set of first principles. To conclude this core chapter, a global overview of the methodological path we have followed for the engineering realisation of the thesis is outlined in the final section.

6.1 Guidelines for Developing Autonomous Systems

This work is deeply rooted in the ASys vision for the development of technology for building autonomous systems. In this section we will present the basic ideas that have been developed into the solution that this thesis proposes for the construction of self-aware autonomous systems.

The underlying ideas of the ASys vision have already been presented in the motivation of this work (section 2.1.2), and can be synthetised in the following statement:

> *It is possible to engineer any-level autonomy systems using cognitive control loops.*

Let us now discuss in more detail the core aspects of ASys concerning the design of the autonomous system that this thesis addresses. Those other ASys' aspects that

relate to the engineering methodology of this work will be treated in section 6.3 of this chapter.

6.1.1 Self-engineering for autonomy

Considering the adaptivity challenge for the control engineering of autonomous systems presented in chapter 1, there is a need to design the autonomous system (plant plus controller) so that it is capable of directly addressing the mission objectives —the desires in the mind of the customer or the functionality the control engineer designs for. The ASys system, in order to do this may be in the need of re-designing, re-building itself, now at runtime, to adapt to the new circumstances. In ASys, this may be not just self-configuration or self-tuning; this may be indeed self-engineering.

The claim of the ASys approach is thus that to achieve robust autonomy it is necessary to:

Break the engineering/runtime gap.

The scenario that ASys addresses goes beyond the current standard engineering life-cycle, and is stated as *autonomy engineering*. It can be defined in crosscutting terms: given the user needs (electricity availability, etc.) design the overall system (plant plus control) that keeps fulfiling the needs embracing unexpected change.

What this necessarily implies is that what is kept as target setpoint for the autonomy loop (see 6.1 later) is not the system as structure but the system as set of processes that render a function. So the objective of artificial autonomy is not self-sustainment of systems but self-sustainment of functions. The vision of a seamless engineering process for autonomous systems captures this capability by embedding into the system the very engineering mechanisms used in its construction.

Artificial autonomous systems are self-engineered systems.

In the standard control engineering scenario, the engineering/runtime gap in a control system implies that unforeseen environmental conditions —i.e. out of design range disturbances— can render the control law, which was fixed at design-time, no longer applicable. Disturbances from within the system itself may also critically hamper its operation. Broadening these adverse conditions is the well-known domain of self-tuning, robust, adaptive or fault-tolerant control. In complex, technical, multi-authority systems-of-systems conditions are even harder.

In ASys, a self-engineered control system is a controller augmented with a *meta-control loop*: it not only maintains a reference set-point for a variable in the environment —the basic control loop—, but it also maintains the very control function that achieves that, even in the presence of the unexpected.

Self-engineered systems realise a teleological process towards a specified function so, in a sense, function is the setpoint or reference injected in an autonomous controller. These systems will embed the capabilities of design-time engineers for their

runtime operation. Obviously the engineering competence requires high-level cognitive performance.

Many efforts have focused on the automation of systems analysis and design tasks. Some of them have been discussed in this dissertation (e.g. in sections 3.3.2 and 3.2.1). There even exist those advancing in the ASys' line to break the design/runtime gap, for example the fault-tolerant control techniques discussed in section 3.5.

Notwithstanding, these extant solutions typically comprise algorithms that rely on a model of the system that is: i) domain dependent —e.g. MFM for the control of processes—, therefore it is not of general applicability, and/or ii) partial —e.g. failure model in fault-tolerant control, which is limited to fault information—, and thus the scope of applicability is usually limited.

ASys proposes to generalise the control strategy so as to address any domain, in the original sense of cybernetics and the early years of AI. This cross-domain applicability intended, together with the high competences required for self-engineering, are the reasons for seeking solutions amongst the techniques of AI and cognitive systems to implement an intelligent controller, capable of embedding the required high-level cognitive competences.

6.1.2 Model-based Cognitive Control

From the original objective of building systems capable of general intelligence [151], Artificial Intelligence has spread into a pletora of subfields, depending on the formalism chosen to encode knowledge. This has rendered many of the technologies developed of limited applicability, since these encoding formalisms are better suited for some domains rather than others.

ASys' seek for generality and robustness has determined the decision to look for an architectural approach to intelligence, rather than an algorithmic one. This more hollistic and general strategy is also the target of the field of cognitive architectures, whose more relevant advances have been discussed in section 5.3. Much in line with many of these research efforts, but even closer to cybernetics, ASys explores a bio-inspired approach studying the principles of natural cognition.

From its control engineering perspective, ASys considers cognition as the exploitation of knowledge —i.e. models— to realize control. We claim that we can equate knowledge and models. Departing from the basic principle: *a system is said to be cognitive if it exploits models of other systems in their interaction with them*, we have started building up the ASys principled approach to cognition and consciousness, published in [134]. This way, these principles prescribe a set of architectural traits for cognitive systems.

The first principles frame the behaviour of a cognitive system in a modelling perspective:

Principle 1: ***Model-based cognition*** *— A cognitive system exploits models of other systems in their interaction with them.*

Principle 2: ***Model isomorphism*** *— An embodied, situated, cognitive system is as good as its internalized models are.*

Principle 3: ***Anticipatory behavior*** *— Except in degenerate cases, maximal timely performance is achieved using predictive models.*

Principle 4: ***Unified cognitive action generation*** *— Generating action based on an unified model of task, environment and self is the way for performance maximization.*

Principle 5: ***Model-driven perception*** *— Perception is the continuous update of the integrated models used by the agent in a model-based cognitive control architecture by means of real-time sensory information.*

Autonomous systems have directiveness, as explained in section 4.3: their behaviour is oriented to the prosecution of some objectives, i.e. the task. In a cognitive system, action steams from the evaluation of the state of affairs, as estimated in the updated model, in terms of the objectives. This value is thus the *meaning* that for the system has its knowledge (the models), rendering it *aware*:

Principle 6: ***System awareness*** *— A system is aware if it is continuously perceiving and generating meaning from the continuously updated models.*

Awareness can be used to direct the system processes so as to be more efficient. This puts *attention*, as understood in natural and bio-inspired systems, in our modelling perspective [68]:

Principle 7[1]: ***System attention*** *— Attentional mechanisms allocate both physical and cognitive resources for system perceptive and modelling processes so as to maximise performance.*

Self-awareness

In the above conceptualisation of cognition, we claim from our model-based position that a system is self-aware or conscious when the model that is being continuously updated and producing meaning includes a sub-model of the cognitive system itself.

Principle 8: ***System Self-awareness/consciousness*** *— A system is conscious if it is continuously generating meaning from continuously updated self-models.*

The self-model allows conscious systems to introspect and reflect on how their own actions produce value, being able to give meaning to them. In section 3.1 we already discussed the basic properties attributed to self-aware systems.

As can easily be inferred from the model-based approach to cognition in ASys, the models a system exploits for operating are what critically determines its possibilities. When coming to artificial systems, there is a big difference between the (design) models that engineers use to build a technical artifact and the (run-time) models that some extant systems capable to some extent of reflection may use during their operation. The use of the design models, developed at engineering time, as run-time self-models will break the design-time/run-time divide and leverage the full potential of model-driven development [17].

6.1.3 Baseline principles for the engineering of autonomous systems

So far we have presented the ASys vision that is the basement of this work: section 6.1.1 regarding the requirements for the engineering of more autonomous technical systems, and section 6.1.2 presenting a principled roadmap for building autonomy by giving cognitive competences to the system. Let us now assemble these ideas into the following design principles to address the development of autonomous systems:

- **Metacontrol:** Teleological robustness —the stubborn prosecution of mission goals— is achieved by control loops to reject disturbances. When these can happen in the control systems themselves we need metacontrollers to reject these disturbances [83] by modifying the controllers and this way maintain their function.
- **Model-based cognitive control:** Cognition is our core competence to develop into autonomous systems; in the ASys perspective a cognitive control loop is based on the exploitation of explicit models of the system under control [37].
- **Break the run-time divide:** Using design models from the engineering of the system as run-time self-models will break the design-time/run-time divide, and would allow for self-engineering systems.

This thesis proposes and realises a solution to reify these principles. The next section presents the central ideas of this solution. Its development into a complete design solution will be addressed in the following chapters of this part of the dissertation. The final section of this chapter provides a global perspective of the methodology followed in this engineering development of the thesis.

6.2 Thesis

Once we have already presented the principles to guide the development of autonomous systems to which this work adheres, from the model-based approach to cognition to the relation between self-awareness and the exploitation of explicit self-models, we are in position to propose an answer to the question addressed by this thesis and raised in section 2.3:

> *How can we enhance systems with self-aware capabilities so as to robustly improve their autonomy?*

In a controlled plant, a human engineer or operator is responsible for "controlling" the control system, adapting and reconfiguring it if demanded by the circumstances (remember section 1.1.3). We propose to add another controller, a *meta*controller, to the control schema, devoted to adapt the conventional controller to disturbances falling beyond its limits. A conventional controller operates according to a model of the plant based on quantities of the plant. The metacontroller will use, instead of these quantities, a model that captures the architecture of the system in terms of its

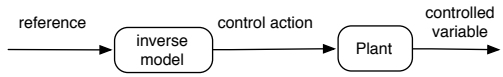
components and functions, to adapt this functional architecture in order to achieve the desired behaviour.

In the terminology presented in chapter 4, the meta-control reconfigures the organisation of the control system in order to adapt it to unforeseen disturbances affecting its directiveness, and does so by exploiting a functional model of the system. We can then say that the complete control system possesses *self-aware* capabilities, since it uses an internal representation of itself to guide its behaviour for adaptation.

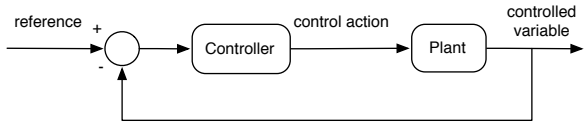
We have structured and detailed this answer in the following three postulates:

I. Functional Metacontroller: A robust autonomous system shall control its function. If we want a system to robustly realise a certain goal or set point in the presence of disturbances, control theory tells us to close a loop on the variable the goal refers to, by sensing those accessible variables from which the value of the desired one can be inferred, and acting on those that influence that value. In the case of an autonomous system, our objective as designers is that the system robustly provides the functionality required —i.e. a certain behaviour—, even in the presence of disturbances such as unforeseen environmental conditions or internal faults. To realise that meta-control purpose, stated in ASys principle I, we propose to close a functional loop between the control system and the metacontroller. This can be achieved by a control loop that takes as reference the functionality required, and senses and actuates on the system's structure, which is what determines its behaviour. Therefore the proposed meta-control operates in the domain of the function and structure of autonomous systems. Putting it in terms of our theoretical framework: to engineer a system with maximal directiveness we shall close a control loop in the system at the functional level, by having it operating on its own structure at run-time. This "functional" meta-controller is the proposal of this thesis regarding the metacontrol principle.

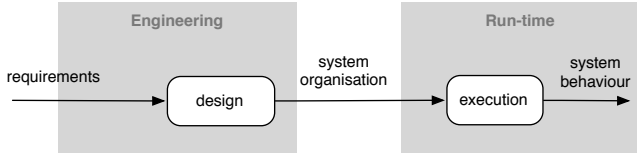
II. Transforming system engineering models into run-time knowledge. If we want to close a control loop on the system's functionality, we need a model of it. In the case of the design of a regular control loop, the model of the plant is obtained from the real plant in a modelling effort by the control designer. In the case of designing the meta-control loop on the functionality of an engineered system, on the contrary, the engineering models that capture the functional architecture of the system are usually developed before the system itself, they are prescriptive: the system is built according to them. The "functional meta-controller" is to be designed from these models. Given the difficulty of translating them into a mathematical formulation, such as that of linear systems, suitable for directly obtaining the controller by control design techniques, we propose to follow the ASys model-based cognitive control principle II and convert these models into another explicit formulation, suitable to be used by the "functional meta-controller" at run-time. This would prove a solution to breaking the design/runtime gap (ASys principle III). The prescriptive nature of the engineering models of the system represents also another advantage for the meta-control designer: modelling errors are inexistent, or at least can be minimized, and the functional meta-control has to deal "only" with the disturbances at run-time.



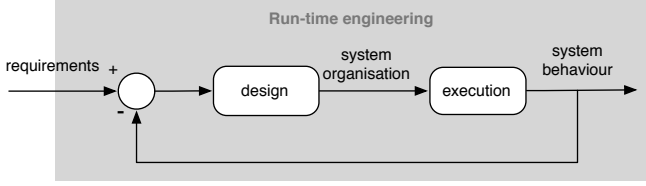
(a) Open-loop control.



(b) Closed-loop control.



(c) Open-loop engineering.



(d) Closed-loop autonomy engineering.

Figure 6.1: Analogy between control strategies and the engineering of autonomous systems. The classical engineering process, with a gap in between engineering and run-time phases, makes it impossible for the system to cope with situations not considered during engineering, in the same way as any disturbance invalidates open-loop control 6.1a. Our metacontrol proposal is a solution for the ASys autonomy-loop shown in 6.1d: a control loop is closed on the functional requirements of the system, so a re-design of the organisation of the system is performed continuously for adaptation, merging the engineering and run-time phases.

III. Control the function by controlling its realisation. Formalising it according to our theoretical framework: for controlling the system organisation to maximise its directiveness we propose to decouple the functional and realisation views, assigning a control loop to each. This way we propose a meta-controller design consisting of two nested loops. The outer one shall maintain the system functionality by performing a regulation control. It will sense the organisational state of the system and it will estimate the functional status of the system, evaluating it according to the required functionality. This way the systems becomes *self-aware* in our sense.

This loop will produce as its control action the organisational specification that grounds the functions required to maintain the system's functionality. The inner loop would receive that specified organisation as the reference, and will perform a servo-control to configure the system according to it.

6.3 Engineering Roadmap

In order to realise the previous postulates for the building of self-aware autonomous systems, we have followed the ASys engineering strategy, centered around the exploitation of reusable assets over architectures defined by means of *design patterns* [127]. This has enforced the generality sought for the solution designed, and also the re-usability of the assets produced for its application to the engineering of autonomous systems.

6.3.1 A Pattern-based Strategy

The postulates stated in this chapter propose a path towards an eventual design solution for self-aware autonomous systems, still to be fleshed out. Following the ASys vision, we have elaborated this solution in the form of design patterns. As discussed in 3.4, patterns provide a formal way of capturing design knowledge, by abstracting the general schema of a solution for a recurring problem in a given domain.

The next chapter will describe the patterns that develop the postulates presented in this chapter and the ASys' principles into fully usable design knowledge for autonomous systems. These patterns furnish for abstract schemas we can apply to shape a design so that it presents the desired traits. They are design chunks or tools that we have used to hammer a concrete design solution in the form of an architecture model.

6.3.2 Architectural Solution: a Reference Architecture

We motivated in sections 2.1.3 and 2.2.2 our orientation to develop a solution for self-awareness at the architectural level. This architectural approach has led us to develop our solution in the form of a reference architecture. As it has been explained in page 26, a reference architecture defines a template for the architectures of systems in a domain.

Let us further justify the architectural approach we have chosen to solve the problem. If we are to follow the previous thesis' statements in order to design systems with improved robust autonomy, we have to be careful to maintain generality and not to develop a specific algorithmic-based solution applicable only to a certain kind of systems, for example those for which a mathematical linear model can be obtained. The architectural approach helps us in that respect.

6.3.3 Engineering Solution

To summarize the roadmap we have followed in the realisation of this work: we have elaborated a framework for the development of self-aware autonomous systems, by applying a progressive refinement from abstract principles to a concrete cross-domain architecture.

The conceptual and methodological path to this framework, which we have called the OM Architecture Framework, is depicted in figure 6.2. From left to right: the principled approach to model-based cognition and self-awareness for autonomy, discussed in this chapter, has been captured as a set of design patterns; they have been applied to synthesize the OM Architecture, composed of a reference architecture, a metamodel and an engineering methodology, which includes a Java implementation of the control architecture; finally, the framework has been validated in the implementation the control architecture of a mobile robot.

This part III of the dissertation deals with the core assets developed in this work: the OM Architectural Framework. In this chapter 6.2 the postulates of the thesis have been presented, refining the ASys principles for the design of controllers for self-aware autonomous systems. Chapter 7 is devoted to the design pattern that reify our principles, while chapter 9 describes the OM Architecture we have developed with the aid of these patterns. Part IV of the dissertation discusses the engineering application of the framework, defining an engineering methodology and detailing its application in the development of the robotic testbed with it.

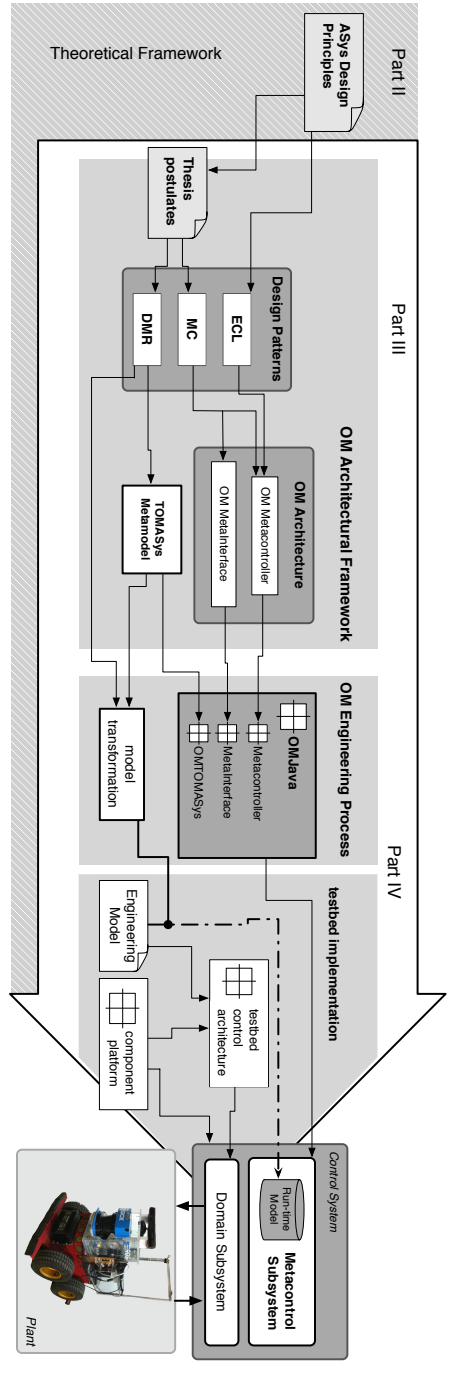


Figure 6.2: The methodological path followed in the development of the OM Architectural Framework, with the elements and core assets developed.

Chapter 7

Design Patterns for Self-Aware Autonomous Systems

In this work we have followed a pattern-based design strategy. This chapter describes the core patterns that formalise the design solutions developed to address the engineering principles stated in the thesis. The opening introduction discusses the common rationale and context behind these patterns. In section 7.2 we present the Epistemic Control Loop pattern, which reifies the ASys principle for Model-based cognitive controllers. Section 7.3 describes the MetaControl pattern for the design of control systems with self-aware capabilities, while section 7.4 discusses the Deep Model Reflection pattern, which allows to engineer systems that use engineering design knowledge during their operation. Each of these sections follows the schema proposed in section 3.4.2 to describe its corresponding pattern.

7.1 Design Patterns for Self-Aware Autonomous Systems

As it has been already explained in 6.3, this thesis follows a principled, pattern-based architectural approach to provide a solution for building self-aware autonomous systems. Therefore, the postulates stated in this thesis, together with the ASys principles they advance, have been developed as a set of patterns. These patterns encompass not only the design of autonomous systems at the architecture level, but some also provide a solution affecting the development process of these systems.

Table 7.1 summarizes this set of four core patterns that will be discussed throughout the chapter.

Table 7.1: Four Design Patterns for Self-Aware Autonomous Systems.

Acronym	Name	Content
ECL	*Epistemic Control Loop*	To exploit explicit models to perform a control loop.
MC	*MetaControl*	A controller that has another controller as its controlled plant.
DMR	*Deep Model Reflection*	To use the system engineering model as self-representation.
FSM	*Functional/Structural Metacontrol*	To control the function by re-configuring the structure.

7.1.1 Pattern Schema

The patterns that have been developed will be described using a pattern schema (see page 57) based on that proposed by Sanz *et al.* in [126] in the context of ASys for the capture and documentation of design patterns in the domain of complex intelligent controllers. We have modified and adapted its sections to better suit the objective here of describing novel design ideas.

Below we briefly describe the different sections of this schema we will be using. Not all sections of them are equally applicable to all the patterns.

Introduction presents an introductory rationale for the pattern.

Name The name of the pattern. Names are important if things named are going to be reused, inter-changed and discussed. Their name should be simple but with a clear relation with the pattern itself.

Aliases Patterns are usually not new; most of them were employed for a long time using different names for them.

Context Contextual information regarding the circumstances of application of the pattern: a design situation giving rise to a design problem.

Related with Other patterns related with this one, by structure, by way of use or —very important— because they are applied at the same time to a system. Some typical related patterns are: **inbound patterns** —higher-level patterns that can benefit from this one— and **outbound patterns** —lower-level patterns that can help implementing this one.

Example The example provides a sample framework of application of the pattern. This section identifies a possible use of the pattern in a real situation.

Core is the main part of the pattern.

Problem The problem the pattern tries to solve.

Problem Statement Essential problem statement.

Forces The competing factors that make the decision of the system architect difficult: requirements, desired properties, constraints...

Solution What the pattern does. What type of solution it provides. The core of the solution.

Solution Principle stated as an instruction emphasizing the general solution principle.

Description A stepwise description of the structure of the solution and an outline of its behaviour, which can be organised in the following subsections:

Structure Architectural description of the pattern. Roles and relations between roles. It is usually a diagram. It includes a description of all the roles that appear in the pattern.

Dynamics How system activity happens. Sequences of role activation. It is important to note that adequate documentation tools are critical for this section. Message sequence charts or finite state machines are common tools for this work but there are many other alternatives like for example use case maps that are very suitable to high level dynamics description.

Appropriate diagrams should accompany both types of description of the solution

Detailed considerations of the solution All subsequent paragraphs explain the solution and its implications in more depth.

Consequences Implications derived from the use of the pattern in an application. Both unavoidable consequences (desirable or not) and those that should be stressed by the system designer.

Implementation Practical issues regarding the put on practice of the pattern to a real situation.

Example application Issues regarding the application of the pattern to the framework presented in the example section at the introduction.

Variants Common modifications to the basic pattern structure. Similar patterns with deviations from the solution provided by the present one.

Similar patterns extant patterns or solution templates that tackle the same problem in a similar vein.

References Bibliographic references for the pattern.

7.1.2 Context

In relation to this schema, the proposed patterns share a common context, which has already been thoroughly discussed in former chapters of the dissertation. Let us resume

it here to adhere to the pattern schema and not repeat it for every pattern, although the context section will still be present to comment specific aspects of each one of them.

The context for the previous design patterns is the development of robust control architectures for autonomous systems, in the current scenario of increasing complexity and interconnection of technical systems, with need of augmented dependability. The design strategy for these systems has to address not only the problem of the uncertainty in the environment, but also of the uncertainty arising from the systems themselves, due to faults, unforeseen, emergent behaviour resulting from the interplay of their components, or their connection with other systems [85]. More traditional approaches such as fault-tolerance based on redundancy are too expensive and not efficient. The universality of the problem demands a general approach rather than specific solutions for certain applications, which are difficult to transfer to other domains.

7.2 Epistemic Control Loop (ECL)

This section describes the design pattern we have developed in this work to reify the ASys principle of model-based cognitive control.

Related patterns The ECL pattern is rooted in well established control schemas: feedback control [127], already presented in section 3.4.4, model-based predictive control [115] or model-based control [153].

Inbound patterns The ECL pattern is intended for the design of complex control systems, for example it can be applied to design the computer-based controller as defined in the Computer Control pattern discussed in [157].

Outbound patterns The ECL Pattern Framework comprises a set of patterns for the design of the elements in the ECL, conforming to a pattern language. It is a work under development in the context of ASys, some of its results will be presented in chapter 9.

7.2.1 Introduction

Context/Rationale

As discussed in the overall context for the patterns presented in 7.1.2, current control systems face increasing demands on functionality and autonomy. Therefore controllers are becoming increasingly complex. Control engineers face the challenge of incorporating more cognitive capabilities in the design of controllers, in order to address these demands, while maintaining a high level of dependability. General solutions befitting any application domain are sought, rather than specific approaches of

limited applicability that are hardly scalable. Solutions at the architectural level of the control system seem to be most desirable in this context.

Example

Consider the design of the control system for an unmanned ground vehicle application. A standard mobile robot platform —i.e. the plant—, equipped with a range laser sensor and odometry, is to be controlled so that it navigates to any location it is commanded in an indoor office-like environment.

Several alternatives to implement the control system are available, including well established techniques for SLAM, and planning algorithms and 2D movement control. These techniques rely on their specific model of the problem: occupancy grid maps, differential equations, etc.

7.2.2 Core

Problem

The ECL Pattern addresses the reification of the ASys' principle for Model-based cognitive control (109).

Problem statement – Design a control loop that uses an explicit model of the plant, so that all the knowledge exploited by the controller to generate action is centralised in such model. The explicitness of the knowledge in the model shall therefore allow for the decoupling of the different operations performed around it. This way, different algorithms and techniques could be used for each operation, and the resulting controller would be scalable so that new ones can be incorporated without modifying the rest of the operations.

Forces – a) Control algorithms determine the representation of the knowledge they use —i.e. the model—, encompassing all the activities in the control cycle: sensing, evaluation with the reference setpoint or action generation. b) These basic activities in the control process are sometimes not clearly separated in control algorithms, making more difficult the scalability of the controller for incorporating new algorithms for each of these activities.

Solution

Solution principle – The Epistemic Control Loop pattern prescribes the basics of the structure and behaviour of a control loop that exploits explicit plant knowledge

—i.e. the model of the plant— in the performance of situated action. The ECL establishes an explicit separation of the operations in the loop, centered around the exploitation of the information contained in the model. This way the Epistemic Control Loop pattern structures the loop into elements according to the activities developed in a cognitive control system [130]: perception, evaluation, thinking and action generation or control. Figure 7.1 shows this separation of concerns in the basic architectural elements or roles in an ECL.

The control loop operates at a certain level of resolution given by the model that constitutes the heart of its operation.

This loop is a more abstract version of that defined by the *Feedback* pattern in classical control [127]. Here signals conveying continuous or discrete variables are generalised to informational flows.

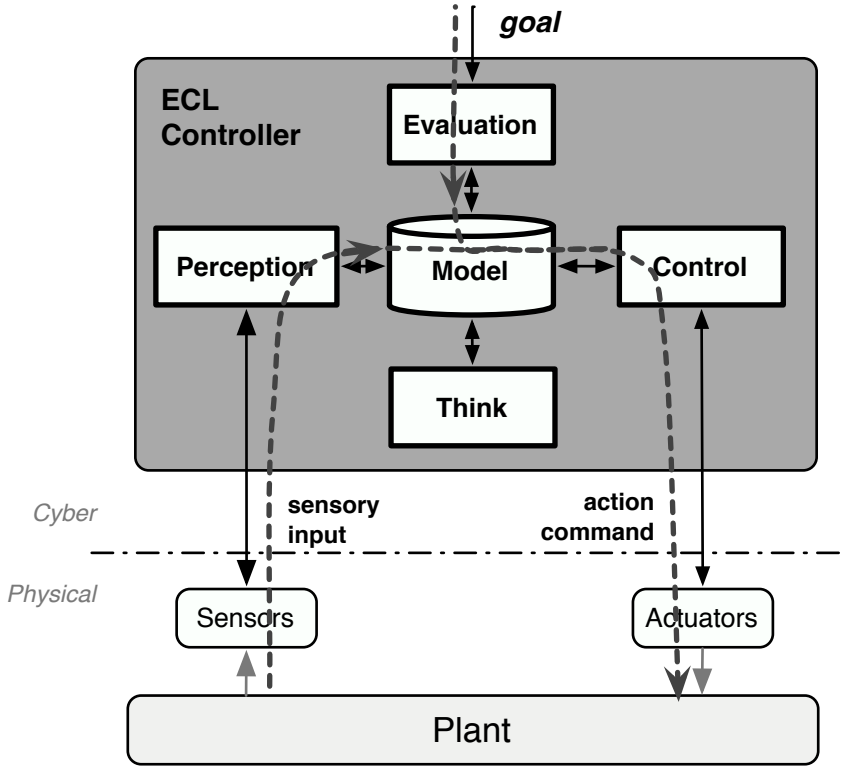


Figure 7.1: The Epistemic Control Loop Pattern structure. Arrows show structural connections between roles, with the arrow head indicating the direction of the data-flow, whereas dashed line arrows show the basic flow of information that leads to action generation.

Structure – The ECL pattern prescribes a general architecture for the control loop, from now on named the *control* or *ECL unit*, defining both its external connections to other elements of the system and its internal organization.

External view An ECL control unit has three basic informational flows to other units or elements in the system: sensory flow, goal, action command.

The *sensory input* flow consists of signals from sensors conveying information about the sensed variables. Sensors do not usually directly measure the state of the plant as the ECL model represents it, but rather quantities related to that state. Sensors convert those quantities into conceptual. The sensory flow consists of those conceptual quantities, from which a representation of the plant state can be obtained through a perceptive process.

The *goal* is an input informational flow consisting of the reference or setpoint for the controller, its *objective* according to our theoretical framework.

The *action command* is the control action that is outputted to the actuators. It may consist of several commands.

Internal organization. The ECL pattern proposes an structural separation of activities which are otherwise not very different from those that could be found in other control patterns, although maybe implicit, whereas the ECL explicits them in the structure of the control unit. This way, four basic activities are performed around the *Model*, which is the core element of the control unit: *Perception*, *Evaluation*, *Control* and *Thinking*.

All application and domain specific knowledge an ECL controller uses is contained in the *Model*. This means that information about the concrete application is included in the model: regarding the physical plant and environment —f.g. differential equation model of the plant, noise models of the sensors, available commands in the actuators and their expected consequences in the plant— and also concerning the mission —reference setpoints, constraints—. But moreover, domain knowledge is also included in the model; this comprises algorithms for obtaining the state of the plant from sensory observations, control action from that state, etc. The *Model* must be accessed and manipulated explicitly, so it can be designed using for example the Database Management System pattern from the computer science domain.

The *Perception* element in ECL encompasses the processing of the sensory input from the *Sensors* to update the estimation of the plant state in the *Model*. This process may also use information of the previous state, the sensor model, etc., contained in the *Model*. It can also drive the sensors.

The *Evaluation* process evaluates the estimated state in relation with the current *goal*. An example of the result of this evaluation is the error signal of classical closed-loop control.

The *Control* is responsible for generating proper actions by using the evaluation result, the information about possible actions contained in the *Model*, and any available knowledge, for example about planning issues. This action is sent to the *Actuators* as the action command.

The *Think* process includes additional reasoning activities operating on the *Model*, e.g. to improve the state estimation, together with any operations that involve the manipulation of knowledge, such as prediction, in which potential future states are produced and evaluated for planning purposes.

Dynamics – The ECL defines a cyclic operation for the control unit in which each cycle follows the perceive-reason-act sequence. However the *Model* serves as a decoupling element that prevents the blocking of the operation caused by a failure in any of the steps. The *Perception* process in ECL corresponds to the first step and may include other reasoning operations as described previously. The *Evaluation* process then generates value from the current estimated state in the *Model*. In the last phase of the loop the *Controller* produces the most appropriate action based on the information available from the evaluation.

7.2.3 Detailed Considerations

Consequences

The *Model* contains both the instantaneous state of the plant and more permanent general knowledge about it. It is the explicitness of this last static knowledge that differentiates the ECL from other control patterns. In these other cases, the static knowledge —i.e. the plant model—, which is application dependant, is supposed to be fixed and embedded in the controller together with the control algorithm, so it is not possible to change or incorporate an element to the control schema without entirely re-implementing it. With an explicit model bearing all the information used in the different elements of the control, the ECL design allows for changing the algorithm of any element of the control, or incorporating a new one, without modifying the rest.

Implementation

Having the application and domain knowledge explicit means that a model for the domain must be developed, whose metamodel must be the model of the implemented architecture for the ECL unit.

References

RCS The ECL pattern is heavily inspired by the RCS node presented in section 5.3.2. The RCS (Real-time Control System) node [3] defines similar functions that are re-

quired in a real-time intelligent control unit. However, the ECL pattern does not include hierarchical organisation, as specified in RCS. Perception and action hierarchies will result from the application of other patterns to the inter connection of ECL units.

PEIS The PEIS (Physically Embedded Intelligent System) loop [123] also considers the aggregation of distributed control components with different functionalities.

OODA Boyd's OODA Loop (Observe, Orient, Decide, and Act) [59, 107] is a concept originally applied to the combat operations process, often at the strategic level in the military operations[1]. It is now also often applied to understand commercial operations and learning processes.

7.3 MetaControl (MC)

Aliases – Meta Architecture (HUMANOBS project), Reflection

7.3.1 Introduction

Context/Rationale[2]

Nowadays many control systems are not anymore simple, isolated PID-like controllers. They are complex software-intensive systems, composed of many interacting elements. These controllers implement sophisticated control functionalities in their application domains: from navigation capabilities in unmanned vehicles, to optimization of production in chemical plants. Additionally, increased levels of dependability and autonomy are demanded. Unforeseen environmental conditions and internal uncertainty are disturbances that threaten the autonomous and reliable operation of these control systems.

7.3.2 Core

Problem

The MetaControl pattern addresses the Metacontrol postulate of this thesis stated on page 112.

Problem statement The MetaControl pattern addresses the problem of designing control systems that are capable of self-adaptation to maintain their functionality, even in the presence of disturbances both external —changes in environmental conditions— and internal —faults, unexpected behaviour.

[1] The concept was developed by military strategist and USAF Colonel John Boyd. More information can be found at `http://en.wikipedia.org/wiki/OODA_Loop`

[2] specific to this pattern, in addition to the general context and rationale introduced in page 119

Forces a) Unforeseen environmental conditions can made a control system misbehave. b) Internal faults cause malfunction of the control system

Solution

Solution principle – MC proposes a separation of concerns in the control system in between providing the control functionality in the domain, and providing the capability to adapt that control. For this second concern the control approach will also be used, therefore having a control loop controlling the controller.

This separation will be translated to the structure of the control system, which will therefore be divided into two subsystems: the *Domain Subsystem*, which consists of the traditional control subsystem responsible for sensing and acting on the plant so as to achieve the domain goal given to the system —e.g. move the mobile robot to a certain location, grab a certain object with a robotic hand...—; and the *Metacontrol Subsystem*, which is a control system whose plant is in turn the *Domain Subsystem*, and whose goal is the system's requirements.

Structure – According to the MetaControl pattern the two subsystems in which the control system is to be divided operate in different domains: the Domain Subsystem and the Metacontrol Subsystem.

The Domain Subsystem (DS) operates in the application domain. The DS can be patterned after any usual reference architecture for control: it achieves the application goals taking them as its control setpoint, and *sensing* and executing appropriate *action* over the plant. The Domain Subsystem could be from a simple PID controller to the navigation architecture proposed in [93] for a mobile robot.

The pattern imposes some requirements on the architecture of the Domain Subsystem:

In relation to its internal structure, the Domain Subsystem's architecture must be modular. That is, it has to be made of components connected through well defined interfaces. This modular structure shall have some redundancy, not necessarily physical but maybe analytical [21], in order to provide for reconfiguration opportunities.

Concerning the externally exposed interface:

- the implementation of the DS has to provide a *monitoring* infrastructure, providing data at run-time about the processes and elements realising its control function.
- the implementation platform shall yield mechanisms for *reconfiguration*, including the possibility to modify the internals of the components —e.g. changing the value of their parameters— and their externals: deactivating or eliminating components, instantiating new instances, or changing their connectivity.

The Metacontroller Subsystem (MS) can be patterned after any control architecture, with no additional constraints on its internals. Its reference setpoint will be the

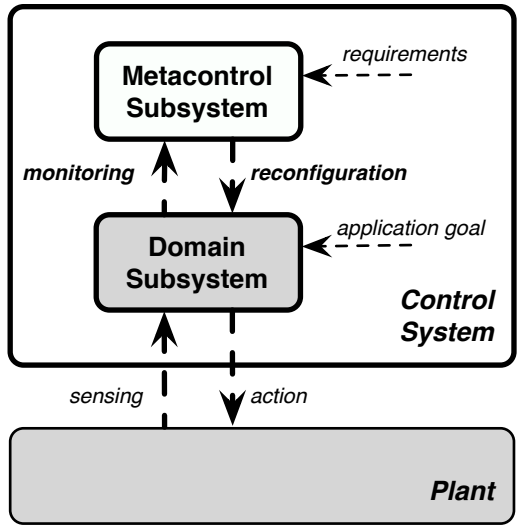


Figure 7.2: The structure proposed by the MetaControl Pattern.

functional *requirements* imposed at design to the Domain Subsystem. The MS will connect to the DS through an interface including:

- the monitoring about the DS, as the sensing input to the MS,
- the reconfiguration mechanisms, serviced by the DS the control action made available to the MS.

Dynamics – The Metacontroller controls the Domain subsystem by changing its structure, and hence its dynamic behaviour. An action from the MS can completely change the dynamics of the DS. Therefore, to maintain the stability of the controlled system composed of DS plus Plant, the MS shall operate at a much slower frequency than the system.

7.3.3 Detailed Considerations

Consequences

Some important issues must be considered regarding the dynamical coupling of MS and DS.

The MS must filter with special care any high frequency noise in the monitoring data: it is not desirable that a rare, transient non-critical fault in a component of the DS may cause an action from the MS that completely reconfigures the Domain Subsystem.

The transitory dynamics of the DS whenever the MS performs an action over it are also important. Attention has to be paid so that this transient does not also destabilise the DS + Plant system. Atomicity in the reconfiguration actions, during which the actions of the DS over the Plant may be halted, could be used.

Variants

Several control schemas proposing an adaptation mechanism for the controller have been put forward; they can be considered variants of the MetaControl pattern to some extent. Some of the more relevant in the control literature are: Model-based Adaptive Control, Reference Model Adaptive Control. A more recent variant in the field of Fault-tolerant systems is the schema proposed by Blanke et al. in [21] for Fault-tolerant control.

References

The separation of the domain control and the meta-control is at the core of AERA, the architecture for autonomous agents developed in the HUMANOBS project[3].

The issue of metacontrol is also discussed in relation to the reconfiguration of control systems in [42].

7.4 Deep Model Reflection (DMR)

7.4.1 Introduction

Context/Rationale

The context for this design pattern is the gap between the engineering and runtime phases of an autonomous system, discussed in section 6.1.1. Summarising it here, the standard engineering of a control system fixes as frozen its capabilities to deal with uncertainties. The resulting system cannot therefore cope at runtime with circumstances not considered during design. However, there was engineering knowledge available at that former phase that could be useful to give a solution for those unexpected circumstances. Autonomous systems could then greatly benefit if they were designed so that the engineering knowledge could be applied at run-time.

[3]HUMANOBS (Humanoids the Learn Socio-Communicative Skills by Imitation) is an European project funded by the FP7 program. More info at `www.humanobs.org`

7.4.2 Core

Problem

The Deep Model Reflection pattern addresses the second postulate of this thesis:

> Transforming system engineering models into run-time knowledge.

Problem statement – This pattern addresses the problem of how to use the engineering models of a control system as a self-representation, so that the system can exploit it at run-time to adapt its configuration and thus maintain its operation converging to its requirements.

Forces – a) The engineering models of a system contain valuable information that could be exploited at run-time. b) Engineering models can be interpreted only by domain engineers.

Solution

The Deep Model Reflection pattern proposes a solution not at the level of the system's design, but at the level of the engineering process and accompanying framework for it.

Solution principle – The Deep Model Reflection pattern proposes to develop a metamodel capable of explicitly capturing both the static engineering knowledge about the system's architecture and functional design, and the instantaneous state of realisation of that design. This *Functional Metamodel* has to be machine readable to be usable by a model-based controller.

Besides, a *transformation model* from the design languages, used to model the system during engineering to this metamodel, shall be feasible, in order to generate the *Run-time Model* of the system to be exploited during operation from the *Engineering Model* of it.

7.4.3 Detailed Considerations

In order to economize efforts, a single engineering model is desirable. Otherwise, several transformations should be developed for every relevant model of the system —functional, electrical, mechanical—, and a careful integration to unite the results into a consistent and coherent run-time model would be needed.

The metamodel (FM) to which the Run-time model has to adhere to be exploitable for online system self-reconfiguration shall comply with more complex and detailed

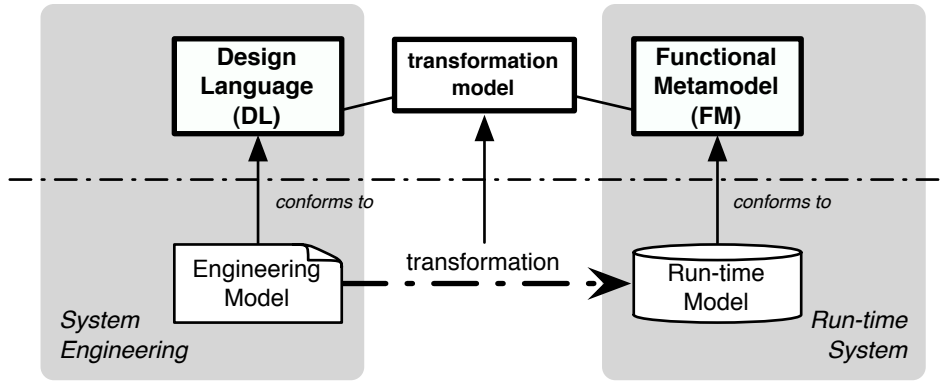


Figure 7.3: The roles involved in the Deep Model Reflection Pattern: automatic generation of the Run-time Model departing from the Engineering Model is possible if each one conforms to a formal metamodel, and a transformation model between both metamodels is provided.

requirements than just being *formal*. A complete discussion of them, together with a solution for the Functional Metamodel is provided in chapter 8 of this dissertation.

Consequences

The Deep Model Reflection design pattern mainly affects the engineering process of autonomous systems. Apparently it may increase the effort in the design of such a system: the engineering models of the system have to be converted into the runtime model, and, moreover, an extra component in the system must be designed so as to exploit the said model during operation (this component is out of the scope of the present design pattern, but it is addressed by the MetaControl pattern in 7.3 and the Epistemic Control Loop pattern in 7.2). However, if the appropriate methodology and toolset are available, the extra effort can be minimal. Specifically, a Model-Driven Engineering approach can be used to develop the system, thus naturally producing a single, unified engineering model of the system as a result of the design phase. SysML is the most suitable option for the language of that model, bringing also the possibility of an automatic transformation into the runtime model. This is so because SysML has a formal metamodel (MOF).

Implementation

The implementation of the DMR requires the following actions:

- Provision of a Functional Metamodel, which could be one already available.
- Application of a strict Model-Driven Engineering methodology to design the system, using a single design language to model the system (e.g. UML or SysML).

- Provision of a transformation model from the design language to the Functional Metamodel.

Once the former elements have been incorporated to the engineering methodology and toolset, the DMR pattern will display in the design phase of a concrete autonomous system:

- the modelling of the system during design will produce a complete Engineering model as a result of the strict application of MDE.
- the Run-time model is automatically generated by transformation of the Engineering Model.

Example Application

The DMR has been applied in the Reference Architecture and accompanying framework developed in this thesis work. As a core result, the **Functions and Components Metamodel** has been developed as a core asset (see chapter 8), to provide for the DMR *Functional Metamodel*. Additionally, **UML** was selected as the *Design Language* for the development of autonomous systems, and a transformation model from UML to the Functions and Components metamodel has been specified as a set of rules. Chapter 10 about engineering autonomous systems with the OM Framework shows how the Deep Model Reflection pattern has shaped the engineering framework developed in this work.

Regarding the application to the development of a specific system, chapter 11 about the robotic testbed for this thesis describes how the engineering model for the mobile robot was produced and transformed into the runtime model.

Variants

In the discipline of functional modelling, closely related to the process industry, several methodologies have been proposed to obtain functional models of the plants from the engineering models. The purpose of these functional models is to be used for diagnosis, analysis of abnormal situations, etc., as discussed in the state of the art about functional modelling in section 3.3. Such approaches can be considered variants of this Deep Model Reflection pattern. The difference is that in DMR the finally produced model is intended to be exploited by the system in order to re-engineer itself.

References

Metamodelling is a core topic in the domain of software modelling [79, 55, 13], and functional modelling has been addressed in many disciplines, for example in the control of industrial processes [87]. A more detailed analysis of all these references was already provided in section 3.2.

7.5 Functional/Structural Metacontrol (FSM)

Related patterns

Inbound patterns The Functional/Structural Metacontrol pattern provides a design solution to the Metacontrol Subsystem defined in the MetaControl pattern.

Outbound patterns The OM Architecture, which will be discussed in chapter 9, defines a set of patterns and guidelines for designing a metacontroller according to the FSM pattern.

7.5.1 Introduction

The Functional/Structural Metacontrol pattern addresses the problem of building a self-aware metacontroller for autonomous systems. It does so by following the design guideline of the third postulate of the thesis (page 114).

7.5.2 Core

Problem

Problem statement – Design a Metacontrol Subsystem to couple with the Domain (control) Subsystem so that the behaviour of the complete system (Plant + Control) fulfils the system's requirements.

Forces – a) Environment disturbances out of the scope of the DS. b) Internal Faults in the DS. c) changes in the system's mission

Solution

Solution principle – Divide the Metacontroller, in two layers. The lowest one a control loop on the structure of the Domain Subsystem, in terms of its components' configuration, and the topmost realising a control loop on the system functions.

Structure – The lowest layer, called the Components Layer, receives as sensing the monitoring data from the DS, a certain desired configuration of the components forming the DS as its goal commanded from the upper layer, and produces the appropriate reconfiguration commands over the DS.

The uppermost layer, called Functions Layer has as goal input the requirements of the system. It receives as sensing from the lower Components Layer the estimated state

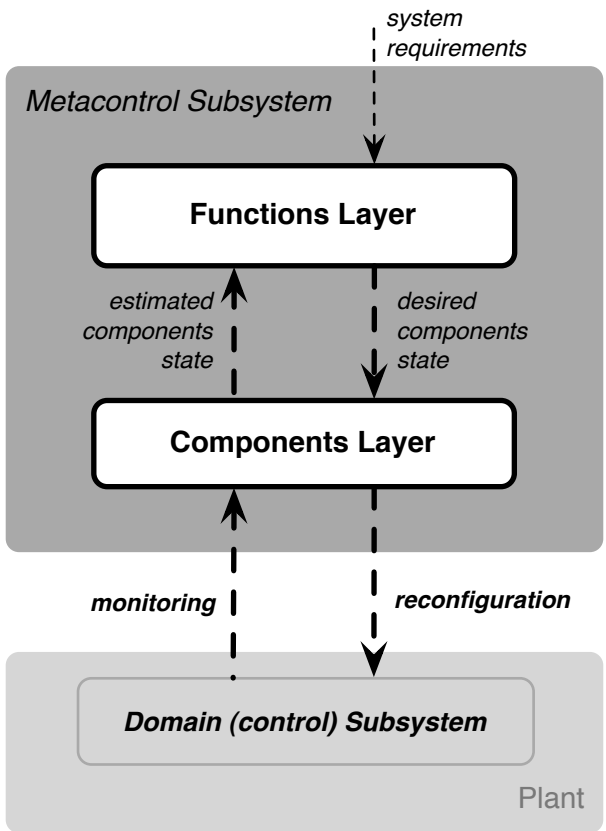


Figure 7.4: The two-layered feedback loop structure of a metacontroller in the Functional Metacontrol Pattern.

of the components of the DS, from which it has to observe the current functionality rendered by the system. It then generates the DS components' configuration that best fulfils the system requirements in the observed circumstances, and commands it to the lower layer.

7.5.3 Detailed Considerations

The Functional/Structural Metacontrol could be further generalised into a more general pattern. This can be done by considering that the domain subsystem does not need to be a controller, it can be any kind of system that can be modelled in terms of functions.

The FSM pattern can then become the FSC: Functional/Structural Control. This pattern describes how to divide the organisation of a function/structure controller operating over a certain (domain) system. Essentially, the FSC pattern specifies how to develop a system that can exploit the available knowledge about its functional structure to improve run-time operation by acting over this very function and structure. Using the controller the system is able to reorganise itself to keep operation.

There are plenty of realizations of this pattern whenever structural/functional knowledge is used at runtime. For example, we could consider some fault-tolerant controllers as a realization of this pattern: the domain system performs a certain valuable activity by means of some organisation -the structure- of a collection of components -that serve the functions. The FT mechanism is able to use knowledge about this organization to re-organise the system in case of detecting a faulty component so as to keep the service.

Example Application

The FSM pattern has been applied for the metacontrol architecture developed in this thesis, and it is described in chapter 9.

Chapter 8

TOMASys Functional Metamodel

In this chapter we present TOMASys, the metamodel we have developed to support the model of itself that an OM system will exploit for self-awareness. It presents a functional conceptualisation of an artificial system based upon that described in the theoretical framework of chapter 4. Along this chapter, the rationale behind the main elements and concepts of the metamodel will be presented, and the metamodel's design decisions will be discussed.

8.1 Rationale

The solution presented in the core of this thesis (section 6.2) intends to endow technical autonomous systems with self-awareness. It is deeply rooted in the idea of the systematic run-time exploitation of explicit models for control purposes, presented in section 6.1.2 and reified by the Epistemic Control Loop pattern (section 7.2).

The run-time model used by the metacontrol subsystem is therefore a key element in the OM Architecture (figure 8.1). The Deep Model Reflection pattern discussed in section 7.4 provides a metamodelling solution to build this model. Taking a metamodelling approach provides to the solution independence from both the application domain and the specific component platform used for system's implementation.

In this chapter we present the metamodel we have developed to apply the DMR pattern. It is a metamodel that must be capable to model any autonomous system – the *domain subsystem*–, and formal enough so that the resulting model be machine readable and executable, that is exploitable by the *metacontrol subsystem* of the OM Architecture.

In order to adhere to postulate II, so that the model shall be obtained from the

engineering models of the system, the DRM pattern prescribes that there must exist a model transformation from the engineering languages into the metamodel. This has driven the formalisation chosen for the metamodel. The transformation will be discussed in section 10.3.

According to the thesis postulate III for separating organisational and functional concerns (page 114), the metamodel must include concepts for the explicit representation of the autonomous system's structure *and* also of its function.

The aim for developing the metamodel was to elicit the minimal set of concepts that would allow the metacontrol subsystem to reason on the impact of the system's current state upon its mission or objectives, in an analogous way to what a plant engineer would do.

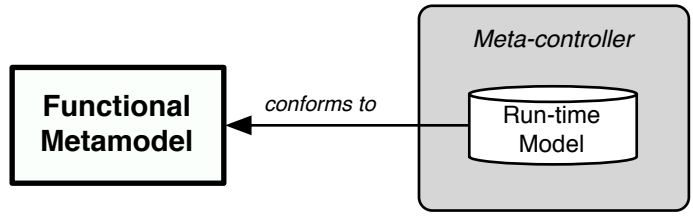


Figure 8.1: The metamodel to which the run-time model conforms is central to the solution developed in this work.

8.1.1 Requirements and Scope

Considering the rationale discussed above, we can synthetize the following list of mandatory properties and requirements for our metamodel for the function and organization of an autonomous system:

- the metamodel must explicitly capture the complete knowledge about the organisation and directiveness realisation of the autonomous system, separating the ontology of the system from its teleology,
- the metamodel must be formal enough to have machine readable implementations, and
- it must be feasible to develop a transformation from the engineering description languages used to specify the design and functionality of autonomous systems, such as UML, to our metamodel.

In relation to the scope for the envisioned metamodel, we have limited ourselves in the different dimensions of the problem of its construction. Regarding the core traits of an autonomous system, we have limited our solution, to the structural aspects of the system, given the difficulty of including dynamical aspects. Therefore the metamodel only covers the former.

In relation to the engineering knowledge encoded in the model of the system and the cognitive process it supports, the metamodel provides infrastructure only for the perception of the dynamic state of the components and the functional designs. The knowledge about the types of components in the system and their properties, that is the ***structure*** of the system, is supposed static ad hard-coded at development time. The same applies to the functional designs. However, the metamodel is envisioned to support the development of learning mechanisms that would allow the metasystem to perceive new components of previously unknown types, and create new design solutions from them.

8.1.2 Relation to other functional models and specifications

The metamodel for the function and organization of autonomous systems we have developed is based on previous efforts to model the function of artificial systems. Those that have had more influence on it have been introduced in section 3.3. MFM [87], GTST [96], Di-Higraphs [42] have furnished conceptualisations for the functional perspective. The component model of Blanke [21] and the metamodels coming from software: OMACS[108], Fiadeiro *et al.* [48] and Garlan *et al.*[53], have supplied organisational notions, together with concepts to capture the relation between function and organisation. Additionally, these metamodels have provided solid ideas and methodologies for the formalisation and operationalisation of the metamodel developed in this work.

Considering the most basic and general issues of modelling itself, as discussed in section 3.2, the MDE approach to modelling has provided useful concepts. For example, concepts from UML, SysML and MOF, such as Component, Port, or Class, Instance and Property, have been used to formalise the metamodel, constituting its meta-metamodel.

Component-based software specifications

There is a series of technical specifications from the OMG, in the context of component-based software systems and MDE, that define conceptualisations of systems. They are in the line of UML, but more specifically oriented to component-based systems. These specifications have been a source of relevant concepts for the metamodel we have developed too. But, in addition to their content, their descriptions have also furnished useful formats, notations and conventions to express our metamodel.

The core model, which underlies the rest, is the UML (2) Components and Composite Structure packages. These packages from the UML modeling language define a *component model* through a series of elements: component, ports, connectors, properties, internal structure diagrams, etc. They conceptualise a software system in terms of components:

From UML:

> A component is a self contained unit that encapsulates the state and behavior of a number of classifiers.[...] A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality[...]
>
> Ports represent interaction points between a classifier and its environment. The interfaces associated with a port specify the nature of the interactions that may occur over a port.[...]

Robotic Technology Component Specification [103] defines a component model and associated infrastructure services for the development of software for robotic systems. It is based on the Robotic technology Component (RTC), which is defined as a "logical representation of a hardware and/or software entity that provides well known functionality and services". RTC are intended as powerful building blocks (RTCs) so that they can be easily integrated and reused in different applications. The specification includes an introspection section that describes a data model for querying and administering RTCs at runtime. This corresponds to the monitoring and reconfiguration capabilities demanded for our MetaControl pattern.

The Dynamic Deployment and Configuration for Robotic Technology (DDC4RTC) [106] is a specification of data models and service interfaces that departs from RTC Specification to define the application of that component model to the activities of dynamic deployment and reconfiguration in robotic applications. These activities are closely related to the goal of this thesis to provide adaptivity to autonomous systems, since deployment and reconfiguration of components can be used as basic adaptivity mechanisms in component-based systems.

8.2 Teleological and Ontological Model of an Autonomous System

The metamodel we have developed to fulfil the presented requirements is based on the theoretical framework for artificial autonomous systems that has been presented in chapter 4.

The concepts of a system's elements and couplings in the framework have been developed into those of *components* and *connectors*, in order to capture the organisational/structural aspects of an autonomous system. On the other hand, to account for the system's directiveness, ideas from other functional modelling approaches, such as the ones proposed by Blanke en at. [21] and de la Mata and Rodríguez [42], have been adapted to operationalise the concepts of *objective*, *function* and *grounded function*. The concept of *role*, well known in the literature [105, p. 169], has been added. The design of a system defines a series of subsystems (FunctionDesign), each one rendering a certain Function within the system. This definition consists of a specification of components (Roles) whose joint behaviour renders the desired Function.

With all these elements we have defined the metamodel for function and organisation, called Teleological and Ontological Metamodel for Autonomous Systems (TOMASys).

An overview *a la* UML of the principal concepts that form TOMASys is displayed in figure 8.2.

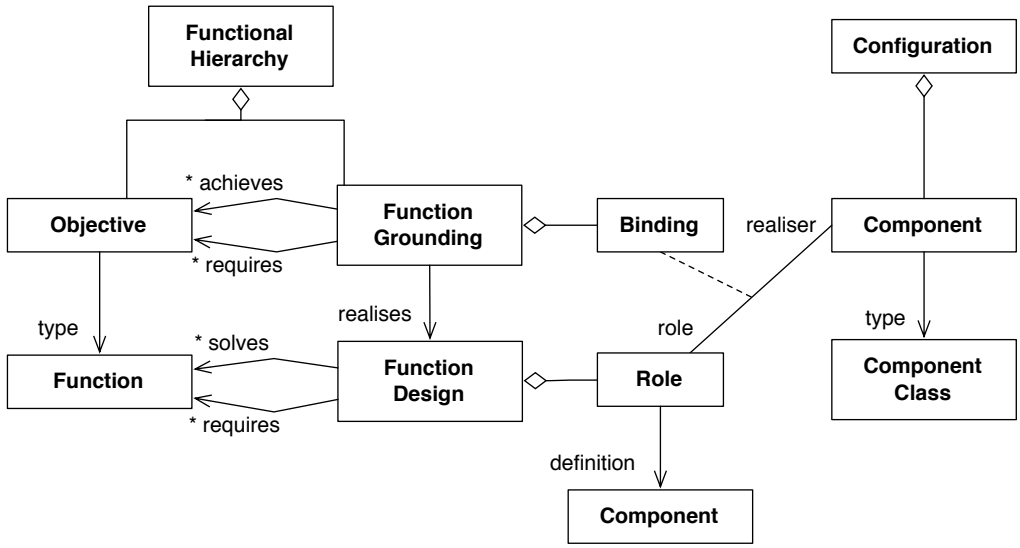


Figure 8.2: Core elements in the TOMASys metamodel.

8.2.1 Model of an autonomous system with TOMASys

TOMASys captures the control subsystem (CS) of an autonomous application in terms of the components that compose it and the functions they are realising. This way, the model consists of a tuple $CS = \langle O, C, FG, F, CC, FD, R, AD \rangle$, that contains two kind of elements:

Elements that capture the instantaneous state of the control system:

- O: the hierarchy of objectives for the system.
- C: the set of components, connected by connectors, which form the system.
- FG: the set of function groundings, which capture the assignments:
  - objective/function design
  - component/role

Elements that represent static knowledge about the design of the control system and the properties of its components:

- F: the set of functions
- CC: the set of component classes
- FD: the set of function designs, each of them defining a set of roles R

Additionally:

- *ADM* the application domain model defines the domain knowledge that constraints and specifies the entity types (components, quantities) that exist in the particular engineering domain of the autonomous system application, e.g. mobile robot. This knowledge is distributed in the concrete value of some of the properties of the previous TOMASys elements.

The elements listed above and their relations and properties will be described in the following sections.

Example

TOMASys model of the mobile robot exemplary system

We could define a TOMASys model of our mobile robot exemplary system presented in 1.5.2. It would consist of the following basic entities:

Components, representing the instantaneous state of core modules of the control system at runtime:
C= { higgs_motors, higgs_odometry, higgs_laser, higgs_map_server, higgs_localisation_module, higgs_navigation_module }
considering that our robot is named Higgs, and every running instance of the components is named by prefixing that name to the type of element. Observe also that single components for the sensors and actuators are considered, which encompass both the physical device and the software driver.

Component Classes, capturing the information about properties of these modules: CC= {motors, laser, odometry, map_server, localisation_module, navigation_module}

Functions, representing abstract objectives:
F= {navigation, localisation, range_scan}
navigation represents the explicit objective to go to the target destination.; whereas localisation and range_scan refer to the implicit functionalities to obtain estimations of the robot self-pose and reading from the laser sensor, respectively.

Function Designs, capturing functional designs that achieve the functionality defined by the Functions:
FD= {grid_cell_navigation, acml_localisation, laser_sensing}

Objectives, that is the state of the run-time instances of the Functions, which represents the instantaneous state of the system's hierarchy of objectives: O= {o2:navigation, o0:localization, o1:range_scan}

Function Groundings: realisations of the Function Designs:
FG= {fg1:grid_cell_navigation, fg2:acml_localisation, fg3:laser_sensing}

Note that to name the Objectives and Function Groundings, the UML notation for instances is used, the instances being the elements of O and FG, and the corresponding classes the elements in F and FD. This corresponds to ontological instantiation [79], and has been used here for conciseness. In the rest of the chapter this notation is used for linguistic instantiation [79], which relates elements in a model to the language (TOMASys) element it is an instance of (see page 141). Simple names are used for these instances, e.g. fg1, o1, with letters to identify their class and a number to identify them unequivocally.

The properties of these entities will be detailed in the examples of the following sections. For the shake of conciseness we will discuss only a portion of the control system: the localisation subsystem, which involves all the elements required to realise the functionality of estimating the robot self-pose.

8.2.2 Organisation of the Metamodel

The elements of the metamodel listed above can be divided into two categories, each of them accounting for one of the two referred views of an autonomous system:

Organisation Elements (C, CC), represent the ontology of the system as it is implemented, i.e. its organisation, captured in terms of components and their connections (C), and their instantaneous state, i.e. accounting for the ***program***. It also includes static knowledge about their properties (CC), i.e. accounting for the ***real structure***.

Function Elements ($O, FG, F, FD, R,$) capture the teleology of the system, in the sense of [87] of representing the roles the designer intended for the components to achieve the objectives of the system. These representations involve the design solutions provided at engineering time for the required functionality (F, FD, R), i.e. explicitly capturing the system's ***structural directiveness***. They also include instantaneous information of how they are being realised in the run-time system (O, FG), i.e. capturing the ***objectives hierarchy***.

The system is thus modeled in TOMASys as a certain `Configuration` of components at the ontological level, and as a `Functional Hierarchy` of instantaneous objectives and the groundings of functions that realise them at the teleological level, as depicted in figure 8.2.

Formalization of TOMASys

For the specification of the TOMASys metamodel we have used a UML-based notation. Each element is captured as a *class* with a set of *properties*. To differentiate when referring to an element of TOMASys in the text, instead to a general concept, this `font` is used. UML Class diagrams have been used to show graphically the properties of the metamodel elements and their relations, as illustrated in figure 8.3.

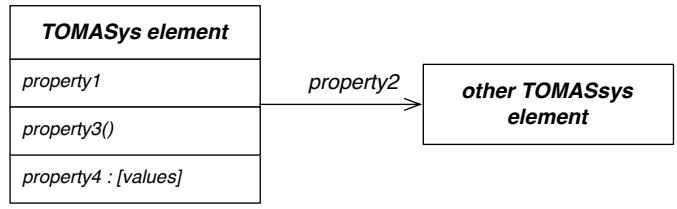


Figure 8.3: Example of the graphical representation used for TOMASys elements. The elements on the left has 4 properties: property 2 is an instance of the metamodel element at the arrow end, property 3 is a function of other properties (that is, it could be considered as a UML method), and property 4 can have only a discrete set of values.

The diagrams in the examples are based in the UML Object diagram. They depict instances of the metamodel elements as Objects. The header section of the box contains the name of the instance, separated from the type of metamodel element it is an instance of by a colon. In the content section of the box their properties and the values they have are expressed as $property = value$. In some cases, the instance that corresponds to the value is also graphically depicted (this is not the case in strict UML diagramming, in which this only occurs in composite diagrams).

8.3 Organisation Elements

In order to represent the ontological view or organisation of an autonomous system, i.e. what the system is, we have developed a minimal set of concepts that could account for most design descriptions of the construction of autonomous systems. Most of current systems are nowadays mechatronical and increasingly software-based, but as for most engineered systems the standard design descriptions, independently of the concrete methodology followed, are centered around the idea of decomposing them into a set of minor subsystems or *components* that interact with each other through defined couplings or *connectors*[1].

Following we describe the elements of TOMASys that account for this view of engineered autonomous systems.

8.3.1 Components and connectors

`Component:` The concept of `Component` is the core element for describing the organisation of a system. A `Component` is a logical representation of either a physical or software element of a system, which performs defined operations producing an output that in the most general case depends on the input and the internal state

[1] for the moment we have not consider the hierarchical aggregation of components

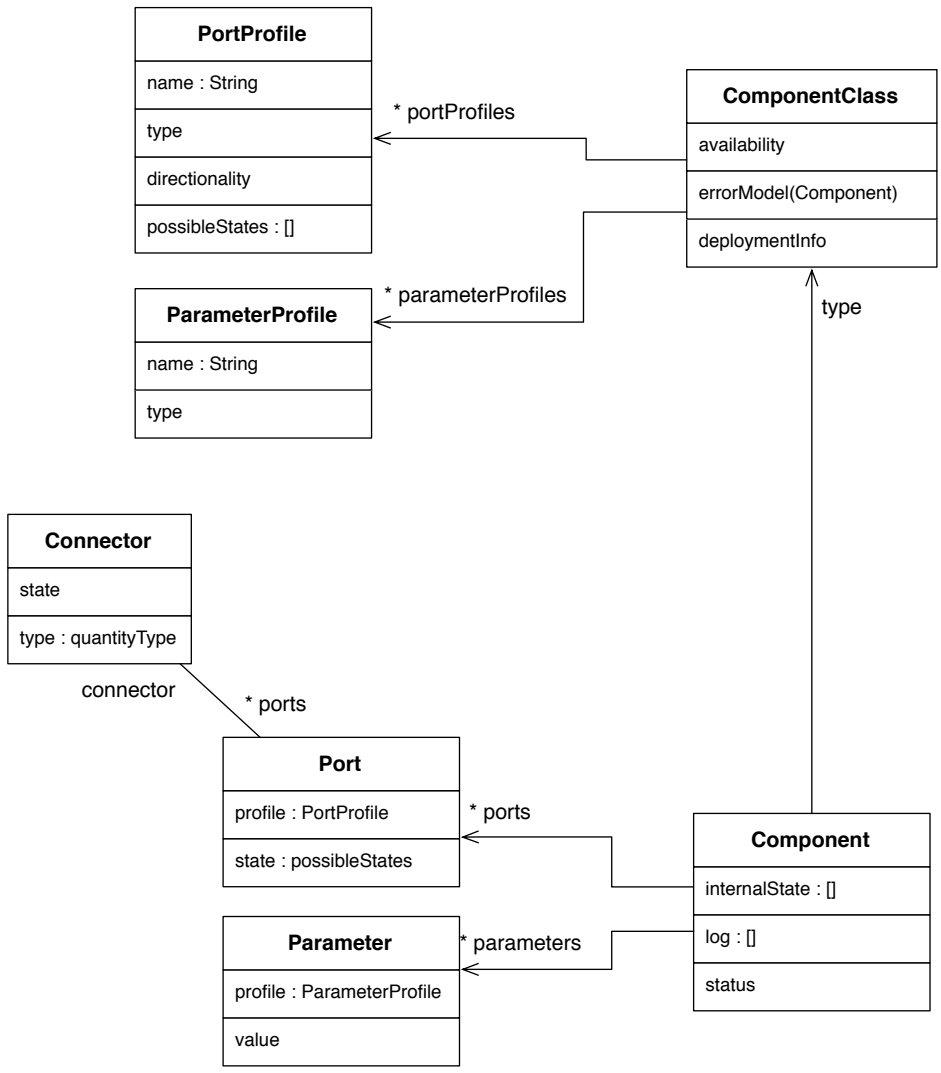


Figure 8.4: TOMASys elements for describing the organisation of an autonomous system.

of the component. The concept of Component in TOMASys operationalises that of ***element***, defined in section 4.2.1.

`Connector`**:** Components interact with other components through input and output quantities —*couplings*, according to our general systems framework—, their directionality with respect to each component depending on whether the component produces (output) or consumes (input) them. These quantities constitute the `connectors` between components. A `Connector` represents one or a group of these quantities in the coupling of two or more `components`[2]. A connector is thus the TOMASys representation of a ***coupling***.

`Port`**:** On the Component's side, the relation between a component and a connector is modelled by a `Port` that the Component owns. Ports thus represent the input and output of a Component for its designed operation. A port is linked to one connector, and only one, whereas a connector can be linked to any number of ports, each of them thus pertaining to a different component. In relation to fault analysis, a port is in a certain `state`, depending on the state of signal that flows through it. TOMASys defines two possible states: {OK, ERROR}, whose meaning is straightforward, but additional states can be defined in the application domain model of the system, such as LOW, FLUCTUATING, etc.

Example

Components of the localisation subsystem

The function of the localisation subsystem is to periodically provide an estimation of the position of the robot. To achieve that, run-time data from the odometry and laser readings are combined and compared to previous knowledge about obstacles, which is received from the map server, using probabilistic techniques in the localisation module. A more detailed description of this subsystem is provided in the discussion of the testbed in section 11.2.

Let us detail the TOMASys Components of the localisation subsystem. The localisation subsystem consists of: two sensory sources: the odometry and the laser sensor, a map server and the localisation module that implements the localisation algorithm.

[2]TOMASys only considers components as the building block of a system from the structural point of view. Connectors are limited to represent the connections between components. Other component models consider connectors as elements with similar properties than components. In TOMASys, such a complex connector can be modelled as a `component`, connected through TOMASys `connectors` with the components it interconnects.

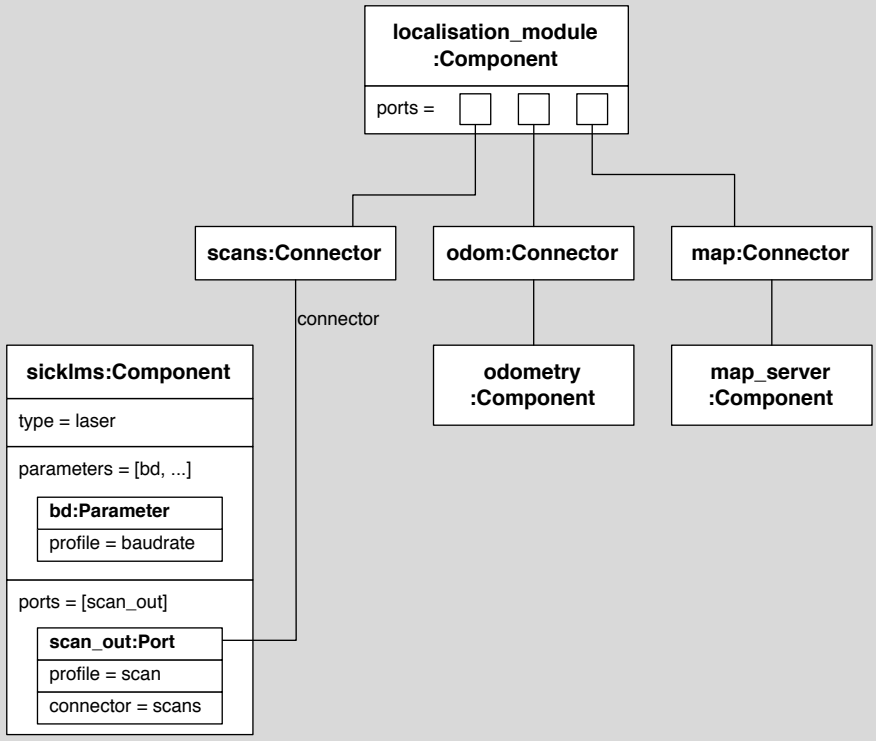


Figure 8.5: Graphical representation of the main organisation elements in the model of the localisation subsystem. For simplicity only the *higgs_laser* is completely detailed. The notation follows basic UML syntax for the definition of instances: the header of each element contains the name of the instance and its class. Inner boxes contain the attributes and their values. The *bd* parameter and *scan_out* port are explicitly displayed. The connector attribute of the port is depicted as a line to its value, which is the *scans* model element, of class `Connector`.

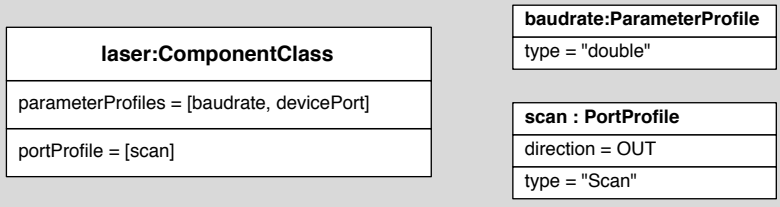


Figure 8.6: Graphical representation of the TOMASys elements that capture the real structure of the *laser* sensor.

8.3.2 Internal Structure of Components

The elements enumerated so far allow for a black-box representation of components. Now we present the elements in TOMASys that capture the internals of a component in relation with the operation it is designed to perform.

The behaviour or operation of a component –i.e. the relationship between input and output quantities– depends on two kind of quantities:

`Parameters`**:** A parameter is a quantity whose value determines the relatively permanent behaviour of the component. An explicit access to modify this value has been designed so as to control the operation of the component through it. A component may have any number of parameters. Example: the baudrate at which a laser range sensor publishes its readings.

`Internal State`**:** in the more general case, the instantaneous output of a component not only depends on its instantaneous input, but also of its history of inputs, represented in the notion of *state* in systems theory terminology. In a designed component, this state is captured in a set of quantities we shall refer to as its `Internal State`

In technical systems, some components are usually monitored because of its importance for the system correct operation, so information about it's internal state is available at runtime, e.g. in software logs. However, this is not the internal state, but designed conceptual quantities that capture information about the operation of the component internals. We have considered valuable to include an element in TOMASys to incorporate this information in the dynamic knowledge of the component.

`Log`**:** captures all the monitoring information that is available at each instant about the operation of a component.

Finally, we have defined the component status to synthetize the information about how the operation of the component is going, adopted from the literature of fault-tolerant systems. The fault-analysis knowledge about the component is discussed in the next section.

`Component Status`**:** represents the operational state of a component: how compliant its behaviour is to that expected from the component. We have defined a discrete and finite number of status a component can be in: *ONLINE* if the component is operating normally, *ERROR* if it is not properly operating, but the reason is known —could be internal, an input out of bounds, a parameter not correctly configured...—, *FAILURE* if it is not properly operating due to an unkown cause.

The following properties correspond to what is usually referred as the configuration of a component: ports, parameters. The specific behaviour of a component of a certain type during a certain activity depends on how it is configured, in terms of the values of its parameters and its connections through ports to the rest of the system. Therefore, the configuration of ports and parameters, together with the type of the property, define the component's hypothetic structure. On the other hand, the value

of the rest of its TOMASys properties can, and will, vary instantaneously during an activity. They represent the instantaneous state of the component, so this TOMASys elements represent the program of the component.

8.3.3 Component Classes

The elements of the previous subsection capture the instantaneous state of the components within a system. There is other important information to model the system organisation: the *properties* of its components and connectors. This information can be considered static knowledge, since the properties are constant during longer spans of time. A group of these properties can also be shared by a set of components, which is thus considered a *type* of components. This corresponds to the metamodelling concept of *types* or *classes* [80]. In this subsections the TOMASys elements to represent the different types of components and their properties are described.

`Component Class:` describes the knowledge about a set of components that share some of its properties. For example, the properties of all the laser range sensors could be described by a *laser range* Component Class as illustrated in figure 8.6.

A `Component Class` aggregates the following elements that encode the properties common to all `Components` that are of that class.

`Port Profile:` describes a port that all Components of the class have: the name of the port, to identify it amongst the rest of the ports of the component, its directionality, that is whether it is an input or output port, the type of the quantity of the I/O connection the port represents –i.e. power, a certain data type. . . –, and the set of possible states of the port (remember that they are part of the ADM). A Component Class may contain any number of Port Profiles, but a given Port shall have only one Port Profile as its profile.

`Parameter Profile:` similarly to a Port profile, a Parameter Profile encodes the knowledge about a parameter common to all the components of a certain class. For example the *baudrate* at which all laser sensors publish their readings, shown in figure 8.6. A Parameter Profile thus has a name, to identify the parameter amongst those of the Component Class, a quantity type, and maybe additional knowledge of allowable values for the quantity corresponding to the parameter.

Both Port and Parameter profiles correspond to *properties* that define the internal structure of elements that have it, such as *Classes* and *Components*, in the UML metamodel. They provide information about the real structure of components.

However, at engineering time we usually also have information about how the state of the component affects its behaviour or operation. Very important fields such as risk analysis, reliability engineering or fault diagnosis deal with the assessment of the effects of faults in the operation of dependable systems. A common technique accepted as a standard in everyday industrial use is the failure-modes and effect analysis

(FMEA) [21, p. 86], which can be integrated in a component model [21, pp. 83–107]. Failure modes are listings of the ways in which a component may fail. There are databases with this information available for many industrial components. Analysis of fault propagation in the FMEA technique uses a Boolean mapping based on matrix schemes. The Fault propagation matrix determines which faults cause which effects in the component.

We have included two different entities in TOMASys to integrate the FMEA information: the `Internal Failure Model` and the `Complete Failure Model`. This is because they are used in different scopes: the Internal is used to quickly obtain the status of a component at the organisational level, whereas the Complete is used to assess the effects of fault propagation between components in the system's functionality (this will be discussed later in page 152 in relation to `roles`).

`Internal Failure Model:` encodes the engineering information about the possible internal failures of a component. It defines which identifiable faults, as captured in the `log` information, cause a complete failure of the component, rendering all component outputs unavailable. The `Internal Failure Model` has the form of a boolean function[3] ifm:

$ifm(log) : \mathscr{L} \rightarrow \mathscr{C}_{status}$ with $\mathscr{L}$ the set of log messages for that component

`ifm` outputs the `Component Status` of a `Component` as a function of its `log` information. The ifm function in TOMASys thus conveys the information encoded in a row in the FPA matrix corresponding to the effect of a complete component failure.

`External Failure Model:` captures the complete information of the fault propagation analysis, generalised for any system the component may participate in. The `External Failure Model` of a component has the form of a function fm:

$fm(log, port_i^{IN}) : \mathscr{L} \bigcup \mathscr{P}^{IN} \rightarrow \mathscr{P}^{OUT}$ `fm` outputs a vector with the states of the output ports of the component as a function of the log information and the states of the input ports.

The fault analysis information embodied in the specific ifm and efm functions of a component is part of the `ADM` of the system.

[3] beware that the term function is used here in the mathematical sense. Along the chapter the term *function*, is used to refer to different concepts: i) the broad concept of function in systems engineering (in regular style), ii) the concepts defined in the theoretical framework for autonomous systems, in *emphasized* style iii) in the name of a TOMASys element, in `tomasys` style, iv) in the mathematical sense, also in regular style, but easily identifiable by the context.

Example

TOMASys failure models for the laser sensor

The FPA scheme for the laser sensor is illustrated in the following table:

Effect	scan low rate	laser scan unavailable
Fault	laser driver glitch	laser not properly connected
Log message	[WARN: A scan was. probably missed]	[ERROR: Device disconnected] [ERROR: Initialize failed! are you using the correct device path?]

The FPA matrix is defined:

$$\begin{pmatrix} e_{l.rate} \\ e_{unav.} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} f_{glitch} \\ f_{notconn.} \end{pmatrix}$$

Modelling this knowledge with TOMASys, we can define the following `ifm` and `fm` functions as Boolean mappings:

For `ifm`:

$\mathscr{M}_{ifm} : \mathscr{L} \times \mathscr{S} \rightarrow \{0,1\}$

$m_{i,j}^{ifm} = 1$ if $log_j \Rightarrow status = ERROR$

$m_{i,j}^{ifm} = 0$ otherwise

For mf:

$\mathscr{M}fm : \mathscr{F} = \mathscr{L} \bigcup \mathscr{P}_{IN} \times \mathscr{S} \rightarrow \{0,1\}$

$m_{i,j}^{fm} = 1$ if $f_j \Rightarrow port_i^{OUT} = ERROR$

$m_{i,j}^{fm} = 0$ otherwise

Engineering knowledge of a type of component can also include information on how to deploy that component.

Finally, there is engineering information about a Component Class that may dynamically vary during system operation, and that is the `availability` $[0,1]$ of that type of component. For example, a sensor can be restarted/re-deployed to fix a wrong initial configuration of one of its parameters. However, no matter what reconfiguration we may carry, if there is only one physical sensor in the system, and it is broken, the component instance will not work: it is *unavailable*, i.e. the `availability` of that

sensor `Component Class` is 0.

In TOMASys, the different levels of the organisation of a component are captured by different sets of elements:

- real structure: it is defined by the types of the components, i.e. their Component Classes, which define their permanent traits.
- hypothetical structure: it is captured by the values of the parameters and the connections of the ports.
- program: the internal states and logs of the component capture information of the program.

Correspondingly, the real structure of the whole system is represented in TOMASys by the set of Component Classes CC, the hypothetical structure by the set of possible configurations of components, given by the set of possible values of the set of parameters P and connections between components.

8.4 Function Elements

The elements of TOMASys shown so far model the organisation of an autonomous system from an ontological point of view. The elements presented in this section model that organisation from a teleological perspective. They capture the objectives that the system pursues during operation, and the functions that it instantaneously performs to achieve them.

This part of the metamodel operationalises the concepts of *objective* and *function* from [90] that have been presented in the theoretical framework of this work (pages 75 to 77). The functional elements of TOMASys thus model the realisation of the directiveness of an artificial system into its organisation.

On one hand the concepts of `Objective` and `Function` model the knowledge about the requirements of the system, as obtained in the analysis phase of the system engineering, and their instantiation at runtime. On the other hand `Function Design` and `Function Grounding` model the knowledge of the design solutions, defined in the engineering phase of the system, and how they are grounded in a set of resources that realise them.

8.4.1 Objectives and Functions

Autonomous systems are designed with a purpose, described at engineering time by a set of requirements, and instantiated in the run-time system as a set of *objectives* as defined in 4.3.2. They are modelled in TOMASys by the concepts of `Objective` and `Function`.

`Objective`**:** a certain subspace of the state space of the system plus its environment towards which the system drives its behaviour.

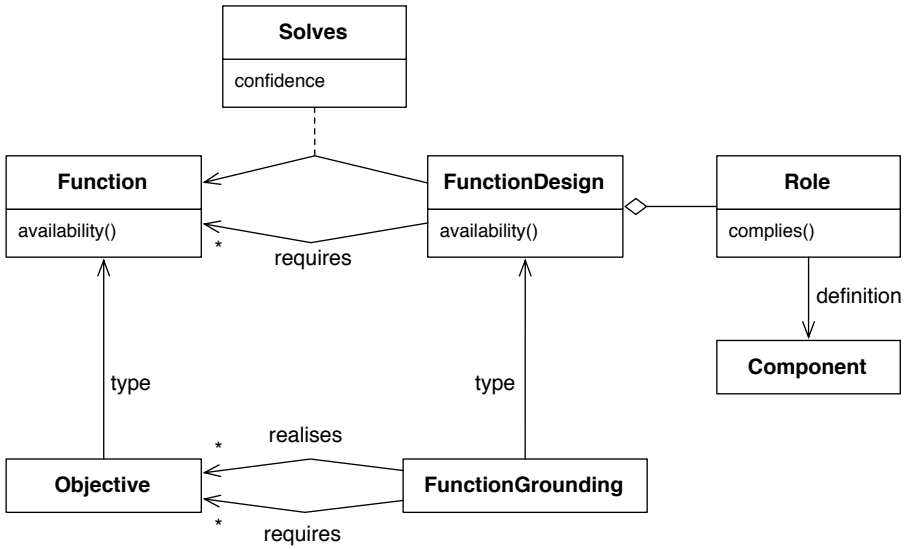


Figure 8.7: TOMASys elements for representing the directivenes in an autonomous system.

TOMASys defines a discrete set of states for the objectives of the system, according to whether the behaviour of the system is directed toward it (*CONVERGENT*), diverges from it (*ERROR*), or its equipontentiality region is non-existent, i.e. no matter what actions the system may take, even reconfiguring it will never achieve the objective (*UNACHIEVABLE*).

There can be different *types* of objectives, defined by parameterisations of the target subspaces that define them. For example, the instantaneous objective for a mobile robot to go to a certain destination (x_d, y_d, z_d), can be generalised by considering as parameters the Cartesian coordinates of the destination point. We then have an abstract objective or abstract function (see 79) f to go to destination point (x, y, z) to be determined at runtime by assigning concrete values to the parameters, thus resulting in a particular instance of the function, which is the runtime Objective. This is modeled in TOMASys with the objective being of a certain `Function type`.

`Function`**:** A TOMASys `Function` represents the concept of an *abstract function* (see 79), which is defined by a parameterized target subspace. This TOMASys element corresponds to a demanded functionality, as defined by the *end* concept in [87].

Engineers design solutions to realise the functions required in the system, by specifying configurations of the system components and algorithms to prescribe their behaviour. This is what Lopez refers to as ***function definition*** (see page 77). It is modeled in TOMASys with a `Function Design`. According to TOMASys, an autonomous system has a set of Function Designs it can use to `solve` its demanded functions.

`Function Design:` defines a design solution to address or `solve` a demanded `Function` with a certain `confidence` [0..1]. It corresponds to the concept of ***function definition*** presented in 4.3.3.

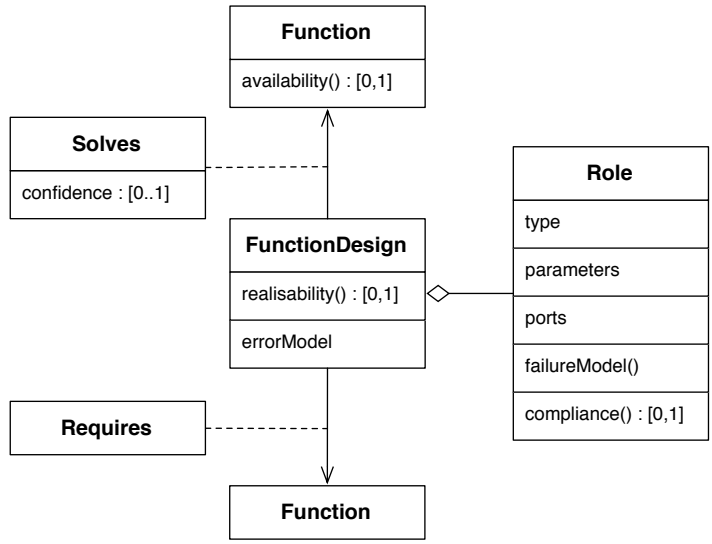


Figure 8.8: he solve and require associations between Functions and Function Design shape the runtime `Functional Hierarchy`.

The `confidence` of a Function Design is a user defined attribute that encodes the knowledge of how well the design is expected to accomplish the `Function` it `solves`.

A `Function Design` contains a set of `roles` that define how certain components in the system must be configured so that their operation will realise the `Function`.

`Role:` encodes the specification of a `Component`, by defining its structural properties both internal (parameters) and external (ports). It also defines additional constraints about the relations with other components. This concept is based on the software engineering concept of role [105, p. 169].

A `Role` thus has some properties as Component: it has a type, that prescribes the Component Class of the Component, and a set of `parameters` and a set of `ports`. The meaning of these properties is different, though. Whereas in a `Component` they are descriptive, capturing the actual values for a real component, in the Role they are prescriptive, specifying what these values should be for a component to play the Role successfully to realize the Function Design.

The `error model`, is another property of the `Role`. The error model of a Role computes if its realiser Component is playing the role properly, or a fault has the effect of preventing it. It is formalised as an em function that returns OK when the realiser component is properly performing the `role`, and ERROR otherwise. The error model conveys fault analysis knowledge, but not context-generic for a type of

component, as does the `ifm` and `fm` functions of a Component Class. It encodes fault-analysis specific for the Role, that is, for the context of an instance of a `Component Class` with a certain configuration, interacting with a specific configuration of other components. However, a default error model can be given by the `Failure Model` of the `Component Class` that is the type of the Role. This defines a minimum fault analysis, since any particular instance of the component will be prone to the faults included in the `Failure Model` knowledge. A more strict `em` function can be defined also as part of the ADM, in a similar fashion to the `fm`, but including a larger and more specific set of faults.

TOMASys `Role` includes also a `compliance` function $C \rightarrow [0,1]$ to determine if a component can play the role given the definition. It can be the case that a role requires a component of a class that is not available anymore in the system. In that case the `Function Design` cannot be grounded. The `realisability` [0,1] defines if a Function Design can be realised in a Function Grounding given the availability of components to play the roles it defines.

Non-critical systems usually contain just one `Function Design` for each function. In fault-tolerant systems with physical redundancy, this one-to-one correspondence between functions and `function designs` still holds, but `Function Designs` for critical functions are or can be grounded several times using different resources, because there are multiple instances of those required. However, in fault-tolerant systems with conceptual redundancy[21], there are more than one `Function Design` that solve a critical function.

When none of the `function designs` that solve a certain `function` is realisable, that Function is not available in the system (`availability` $= 0$), and all `objectives` that are particular instances of it are in `UNACHIEVABLE` status.

A `Function Design` may also `require` the realisation of sub-objectives —other `Functions`— by other parts of the system. The `requires` associations of Function Designs and Functions in the TOMASys model of a system determine the runtime hierarchy of dependencies between Objectives and Function Groundings (see figure 8.8).

Problems or errors in the components defined by the roles, or in the objectives required by a Function Design can have different impacts on the realisation or grounding of that function at runtime. The effect may vary from a transitory lower performance, to a global failure of the function. This is described in the `errorModel` of the `Function Design`, which is a function e:

$e : CS \times SO \rightarrow \{OK, ERROR, FAILURE, PERMANENT\_FAILURE\}$ with CS being the set of status of the components playing one of the roles of the Function Design, and OS the set of statuses of the objectives required (see later on page 8.4.2).

TOMASys expects the behaviour of the subsystem defined by the roles of a Function Design to be convergent to its objective. So the roles and additional constraints imply some desired and expected behaviour. This is defined in the Function solved. Although an understanding of that behaviour is important for the engineering of the

system, the elements and their attributes and associations presented so far are sufficient for the purpose of the metamodel, so it is not further specified in TOMASys.

Example

OM metacontrol of the localisation subsystem

Let us exemplify the concept of Function Design with the localisation subsystem of our mobile robot. Suppose we could deploy two alternative designs to achieve the self localisation objective, because there is another sensor in the robot that can input scan readings, such as a Kinect camera.

This can be modeled in TOMASys with two `function designs`, depicted in figure 8.9: fd_loc1 uses the laser, and solves the self localisation function with a high certainty, fd_loc2 uses a Kinect sensor, and solves the function with a lower certainty, since the scan readings of the Kinect are not as many and as accurate and those from the laser. The roles and associated constraints are graphically shown, similar to an UML collaboration diagram. Some of the parameters of the localisation_module component have different values in each case, since they depend on the characteristics of the sensory sources (this is represented by the gray boxes inside the amcl components in figure 8.9). Note that we are here considering `Function Designs` for localisation that embed all the required functionality in the set of roles. We could have externalised some of the requirements as a required `function`, for example receiving scan readings, and thus eliminating the Role for the scan sensor.

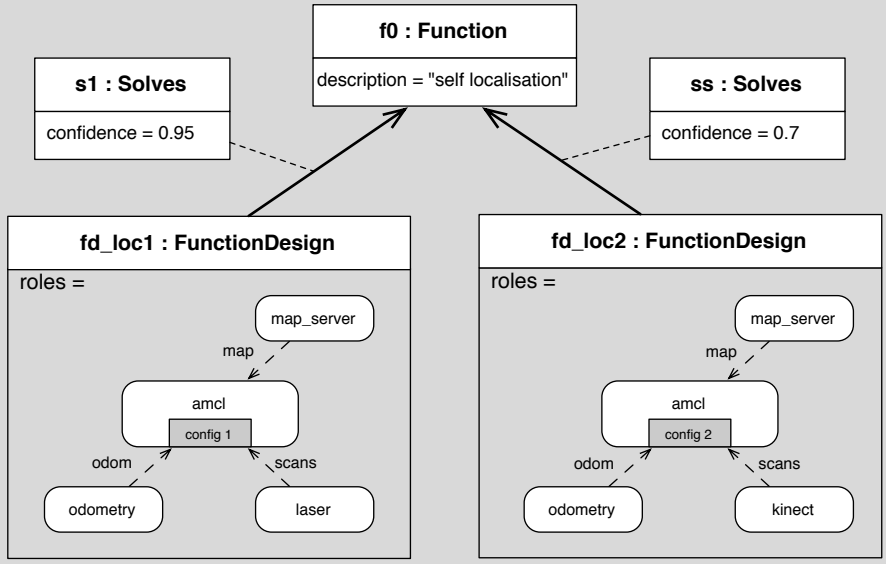


Figure 8.9: Two alternative `function designs` that solve the localisation `Function`. The figure sketches for each of them the different configuration of `components` that are defined by their set of `Roles` and `constraints`.

8.4.2 Functional Hierarchy: instantaneous state of the system's directiveness

At runtime specific components are assigned to fill the roles of the `Function Designs` in the system, and concrete `Objective` instances of its required Functions are required to ground it. This runtime information of the grounding of the functional design to achieve the objectives is modeled in TOMASys as a set of `Function Groundings`:

`Function Grounding`**:** represents the instantiation of a `Function Design` to address a particular `objective`. It thus contains the instantaneous state of grounding of the `Function Design`, as a set of assignments or `Bindings` linking system `Components` and the `Roles` they play to realise the design.

The set of Function Grounding thus models in TOMASys the instantaneous state of the system's directiveness.

A `Function Grounding` models how a part of the system configuration is devoted at runtime to achieve an objective of the `Functional Hierarchy`, and how that depends on other objectives downstream that hierarchy.

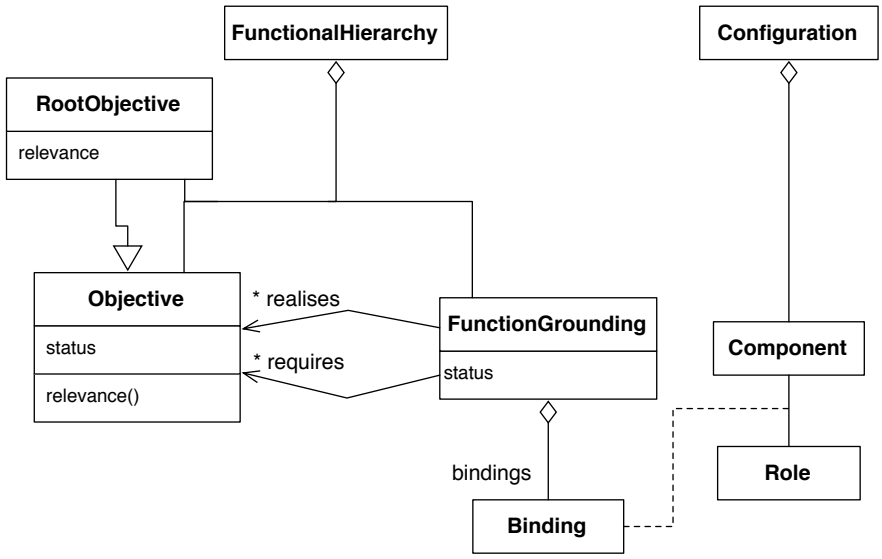


Figure 8.10: The elements in TOMASys that represent the instantaneous state of the system's functions as a Functional Hierarchy of objectives and grounded functions.

The `status` of a `Function Grounding` synthetizes its working state. It can have the following discrete set of values :

OK: when it is working properly the components in the bindings comply with their roles and the required objectives are in the course of being achieved. In this case the status of the Objective it solves is `ACHIEVED`.

FAILURE: when the function grounding is not behaving as expected due to a failure in one or more of its required objectives, and thus not achieving its objective, because one of its required objectives is not being achieved.

ERROR: when the function grounding is not working due to an internal error.

PERMANENT_FAILURE: when it is not working due to a not recoverable internal error, i.e. it is impossible to instantiate a component compliant with one or more of its roles. Its `function` is unavailable.

We can now define the following default `errorModel` for an `Objective`, defining its `status` in terms of the state of the TOMASys elements it is related to:

CONVERGENT: when there is a Function Grounding instantiated to achieve the objective and its status is *OK*.

ERROR: when there is no Function Grounding addressing it, or if there is, then it is not in *OK* status.

UNACHIEVABLE: when there is no Function Design available for instantiation that would realise the `function` of the Objective.

Objective relevance

At runtime, the system Objectives have different `relevances` depending on how much their failure will impact the system overall performance. This is determined by the instantaneous state of the `Functional Hierarchy`, but for those in the head of it: the `root objectives`. For this a fixed `relevance` $\in [0..1]$ is defined by engineers to account for how important is that the system behaviour complies with it. Root Objectives are the more abstract ones and are static during system operation. The rest of the objectives downstream are determined instantaneously depending on the Function Designs that are grounded to realise for the objectives upstream. The same occurs to their relevances. The function that determines their relevance is part of the domain model and is defined by engineers.

Example

Functional grounding of the localisation subsystem

Let us analyse the complete TOMASys model at the functional level for the running localisation subsystem. Here the required input from a range sensor is externalised as a required Function, differing from the function designs illustrated in 154

At the top of the Functional Hierarchy for the localisation subsystem is the root objective $o0$, instance of the function $f0$ "self localisation". There is a realisation named "localisation" that solves $f0$. It defines a configuration of components through the roles $[r01, r02, r03]$ and $fd\_loc.constraints$. This configuration of roles

for fd_loc is depicted in the upper part of figure 8.12: an amcl component, which implements a Montecarlo-based algorithm for localisation estimation, receives information from the odometry of the robot and a map server component and input range information through a *scans* connector. It then outputs a estimation of the robot's position to a *pose* Connector[4]. fd_loc requires the function $f1$, i.e. that range scan data be continuously published in a connector. There is a realisation for $f1$: the F_laser Function Design specifies in the role r1 a configuration for a laser component to get range scans.

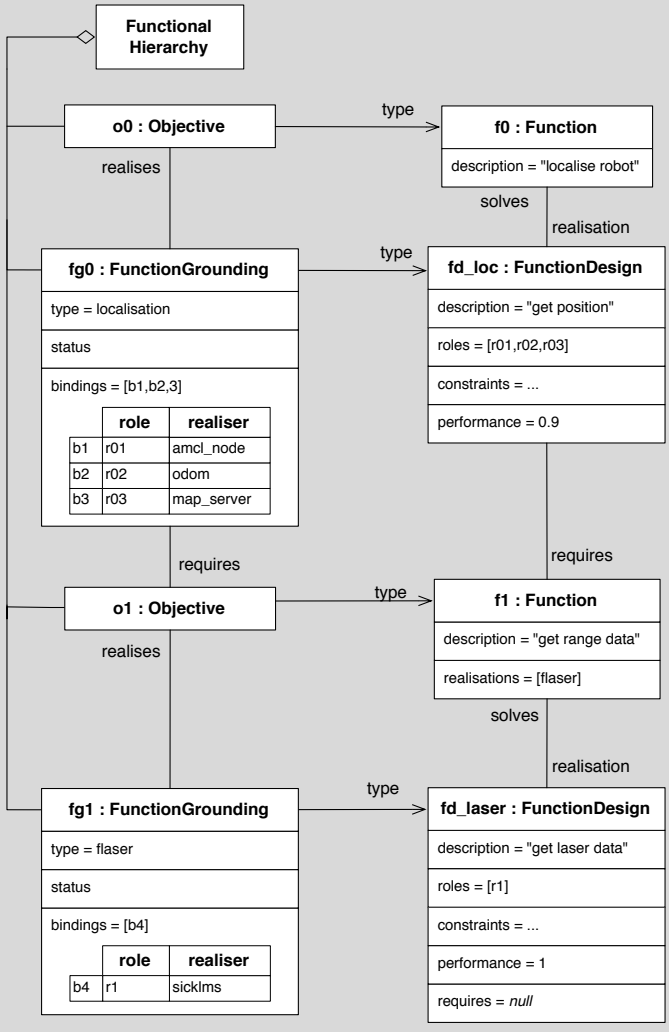


Figure 8.11: The `Functional Hierarchy` of the localisation system.

The Functional Hierarchy is thus composed of $o0$, a Function Grounding $fg0$

of fd_loc, which contains the bindings of the roles to the run-time components, described in 8.5, that realise them, the objective $o1$ required and the grounding $fg1$ with the binding of the *sicklms* Component to the $r1$ laser role.

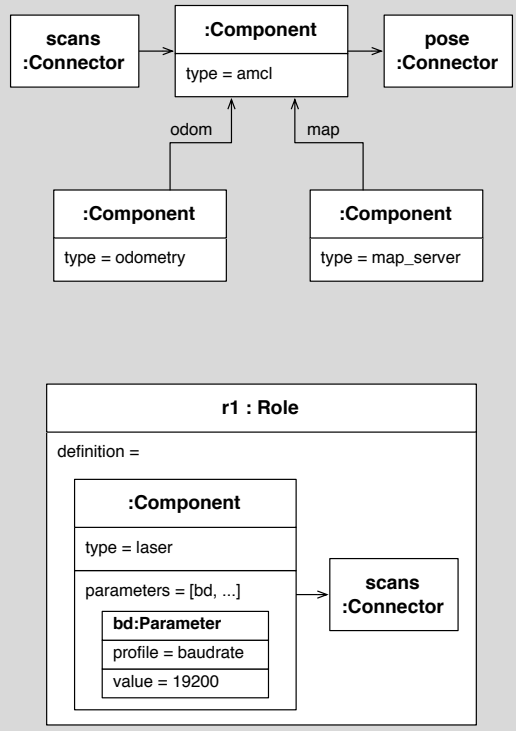


Figure 8.12: Example of the TOMASys definition of a `function design`. The lower box shows the definition of the `role` r1 of the flaser `function design`. The upper part shows a simplified graphical representation of the specifications of `components` and `connectors` for the localisation `function design`, as captured by its defined `roles` and associated constraints.

8.5 Overall analysis of TOMASys

TOMASys constitutes a sound conceptualisation of autonomous systems in terms of components and functions. It can be considered as an upper ontology in the vision of Asmann et al. [11] described in section 3.2.5, and a metamodel, because it provides the language to build models of autonomous systems.

The metamodel approach to knowledge reified by TOMASys provides the autonomous system with an enhanced knowledge organisation (see section 4.5.3). The metamodel corresponds to the real structure of the cognitive subsystem. Having a metamodel means moving information from specific models that were real structure,

to general, abstract knowledge at the metamodel level, therefore maximizing the hypothetic structure, with a larger amount of potentially instantiable models. It thus contributes to system adaptivity. Regarding the resolution level of the TOMASys model of a system, the more detailed the model is —decomposing the control system structure in many components, specifying every parameter, defining all the functions and their—, the larger the program of the cognitive subsystem, and subsequently the better the performance of the metacontrol.

The TOMASys functional decomposition by the required relation between functions and objectives provides isotropy for the design knowledge. This decomposition allows the functional hierarchy to be determined instantaneously, the general knowledge captured in the relations between functions and function designs. This knowledge is thus reusable in different possible hierarchies for different scenarios.

8.5.1 TOMASys and other functional metamodels

Although TOMASys constitutes a *component model* in the sense presented in page 56, taking ideas from the software specifications introduced in section 8.1.2, it is intended to address a functional modelling of the system in the line of the approaches presented in section 3.3.

The OMACS approach analysed in section 5.2.3 has many characteristics in common with TOMASys, addressing the modelling of dynamic software architectures (agents in their case, control components in ours) from a goals/functions perspective. However, we have tried to improve the limitations that we consider the OMACS model has. For example, it merges the structural and functional levels, as other functional models do, e.g. the Di-Higraphs of De La Mata et al. presented in page 55. Another difference is that OMACS does not explicitly models the differences and relations between instances and types of components. We have considered that key not only for the scalability and reusability of model elements, but for the representation of physical redundancy and the applicability of the approach to large systems. In addition, TOMASys considers objectives at different levels, forming a hierarchy, as in the case of the Goal-Tree – Success Tree model (page 55).

TOMASys is envisioned to integrate in OASys (Ontology for Autonomous Systems) [18], as the subontology to provide the concepts for self-modelling of ASys systems. TOMASys elements have been developed from the same autonomous systems conceptualisation than those in OASys different packages.

Chapter 9

The Operative Mind Architecture

Now that the basic architectural patterns for cognitive control and the proposed metamodel for self-awareness in autonomous systems have already been presented in the previous chapters, this chapter presents the *reference architecture* that we have developed making use of them. This architecture provides a blueprint for universal metacontrol systems that realises the solution for self-aware robust autonomy enunciated in the thesis postulates (section 6.2).

The chapter is organised as follows. The first section presents a global overview of the architecture, describing its characteristics, and discussing how it addresses the thesis statements concerning the architecture of an autonomous system. It also sketches its basic operation and how the architecture reifies the design patterns for self-awareness. The next section discusses the organisation of the metacontroller, the core technological design of this work. Two more sections describe in detail the operation of the two control loops that lay at its core. Finally, in the last section the functioning of the metacontroller is discussed for the target scenarios of robust autonomy envisioned.

9.1 An Architecture for Metacontrol

To realise the metacontrol solution proposed in this thesis we have taken an architecture-based approach. This way, the Operative Mind (OM)[1] architecture operationalises the metacontrol solution for self-aware autonomus systems that have been postulated in

[1]The name of the architecture comes from the Operative Mind, which is the conceptual framework that has resulted from analysing the supporting value that biological consciousness renders to the mind when considered as a controller [67]. Those preliminary notions have eventually resulted in this reference architecture, and thus we have named it after them.

chapter 6. It defines the structure and internal processes of the metacontroller that provide the autonomous system with enhanced adaptivity and robustness.

9.1.1 A Reference Architecture

The OM Architecture is a reference architecture as introduced in page 26: it defines the functional processes, elements, relationships and data flows in a metacontroller. A key characteristic of the this architecture is that it can be applied to provide metacontrol capabilities to *any* controller fulfiling a minimal set of requirements, in order to improve its autonomy. The OM Architecture is intended to be universal: it has been developed in line with the aim of this work to be of general applicability to any system. The OM Architecture is independent both of the domain of the autonomous system (e.g. a mobile robot, an automated process plant, etc.), and the component platform in which the control system is implemented.

9.1.2 Scope of the OM architecture

The OM Architecture tackles the problem of building a metacontrol subsystem to be included in the control architecture of an autonomous system, as proposed by the MetaControl pattern (section 7.3), in order to improve the system's capability for self-adaptation. It is a blueprint to guide the building of said metacontrol subsystem and its integration with the rest of the control system.

There are two requirements that the application of the OM Architecture imposes on the (domain) control of the autonomous system. Firstly, its is required that the controller be implemented using componentised technology (see section 3.6), e.g. [119]. This is because at the center of OM is the exploitation of a functional model of the system based on TOMASys, as will be discussed in section 9.3.2, and so it assumes the system can be represented in that component model. Secondly, the control system must also offer mechanisms for online monitoring and reconfiguration of the components, or at least the possibility to implement them. This is what is known in software as *introspection* mechanisms.This implies that this technology for self-awareness —as realized— is still not universal —as was the initial objective of ASys— because it can only be applied to certain classes of software-based controllers but not to any kind of system. For example, it is not easy to use this realization to add self-awareness-based functional metacontrol to neural network controllers, because these are typically non componentable and the introspection mechanisms are too low level. The same can be said about some classes of physically-realised (as opposed to software-realised) controllers. In these, while strongly componentised, we usually lack the necessary introspection mechanisms.

However, while the realization in OM is only applicable to certain classes of software controllers, the theoretical model is still widely applicable beyond these constraints (which are, in fact, not too hard considering the current trends in real-world controller implementations).

The metacontrol scenarios

As discussed in section 3.6, the control system of an autonomous system consists of several software components interconnected in a certain configuration. Unforeseen events, i.e. uncertainties, can alter the behaviour of that configuration and make it deviate from its objectives: from slight deviations degrading system's performance to lost functionalities or, in the worst scenario, a general system failure.

For example, these uncertainties can be individual component failures in the control system, or unexpected data flow incoming from the plant, due to unpredicted events, plant hardware damage or out-of-range circumstances.

The goal of the metacontroller defined in the MetaControl Pattern is to manage the runtime control system so that it keeps system's directiveness towards its objectives, even in the presence of referred disturbances. Within this context, the OM Architecture has been designed to address the following scenarios regarding the operation of the system organisation of components:

1. One or several components of the control system undergo a transient failure. The metacontroller is expected to recover those components from the failures, maintaining the configuration of components.
2. The system actual configuration is incorrect (for example due to an erroneous initialisation). The metacontroller fixes the configuration.
3. One or several components undergo a permanent failure and no components that could perform identical roles are available, that is: the desired configuration of the system is not feasible. The metacontroller re-designs the system, i.e. by instantiating a new functional hierarchy, and reconfigures it accordingly.
4. The control system fails to achieve its functional requirements due to an unknown cause. The metacontroller reconfigures the system following the best alternative design —i.e. functional hierarchy— available to maintain directiveness.

These four scenarios can be reduced to only two, considering how the failure affects the system from a design standpoint:

- Component recovery, corresponding to scenarios 1 and 2, which fall into the typical fault-tolerance scenario.
- Function recovery, in the case of scenarios 3 and 4, which requires an actual re-design of the control architecture.

Example

Reconfiguration of the localisation system

Let us suppose that the control system initially consists of the grounding of the localisation function fd_loc1 of the example on page 154 which uses the laser sen-

sor, so that the domain control system consists of the components shown in figure 9.1(a). In the case of a transient failure of the laser, e.g. due to a synchronisation problem, a re-initialisation of the laser driver could solve the problem. This corresponds to the first scenario described above. However, in the case of a permanent failure, e.g. because the physical sensor is disconnected, a simple recovery at the component level is not possible. This is the third scenario. The metacontroller can only achieve the localisation objective by configuring the system according to the function design fd_loc2, which uses the available Kinect component (see figure 9.1(b)), and reconfigures the system according to it.

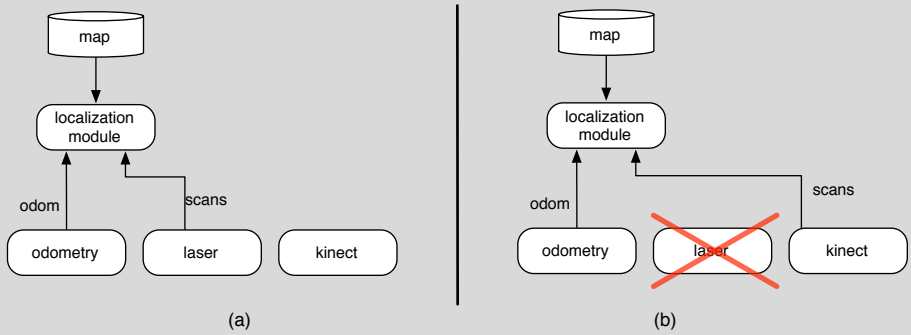


Figure 9.1: Reconfiguration of the localisation subsystem in the event of a permanent laser failure. The figure depicts the initial configuration (a) and the configuration after the failure and subsequent reconfiguration (b).

Operational needs for metacontrol

Note that the previous operational scenarios demand the following functionality from the metacontrol system:

- maintain a representation of the functional design of the control system that is running at any instant.
- maintain an updated state of the functions for the currently instantiated functional design.
- produce a new instance of a functional design accordingly to the available components for the system.
- memory of past reconfiguration actions.

This requirements shall be addressed by the OM Architecture. They can be further decomposed into the capabilities tree of figure 9.2.

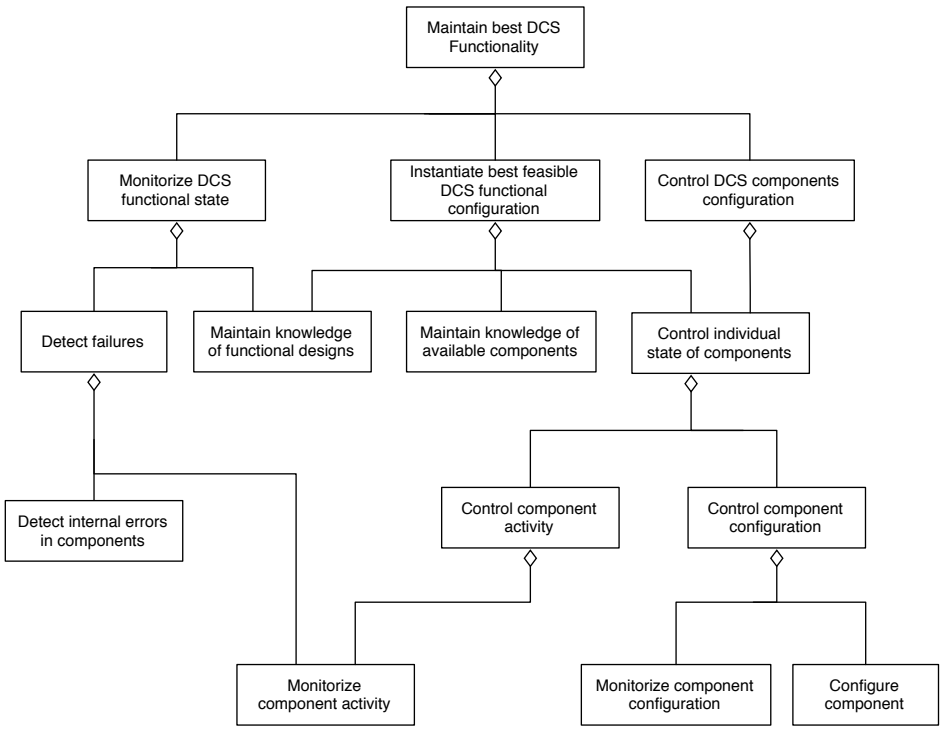


Figure 9.2: The functional requirements of a metacontroller expressed as a tree of capabilities.

9.1.3 OM-based metacontrol overview

The OM Architecture consists of a core element, the Metacontroller, responsible for controlling the control system, connected to the Meta I/O Module, which grounds its conceptual operation. The MetaInterface specifies the data flows between both elements. They consists of a sensory flow, for obtaining information about the domain control, i.e. monitoring, and an action flow, to adapt the control system to the current circumstances, i.e. reconfiguration.

Let us put in a nutshell how these elements of the architecture, schematised in figure 9.3, interact. While the control system is running, the Meta I/O module uses the platform mechanisms to observe and gather monitoring information about it. This signal is used by the OM Metacontroller to update an integrated model of the components and functions of the control system. The current functional state of the system is evaluated against the required functionality, and corrective reconfiguration actions are computed if needed. These reconfiguration actions are finally propagated back to the Meta I/O module, which uses the available platform mechanisms to make the corresponding changes to the running control system. That is, it activates required components, deactivates those unwanted, re-connects and changes the parameters of them as appropriate. All actions are done over components and their connections.

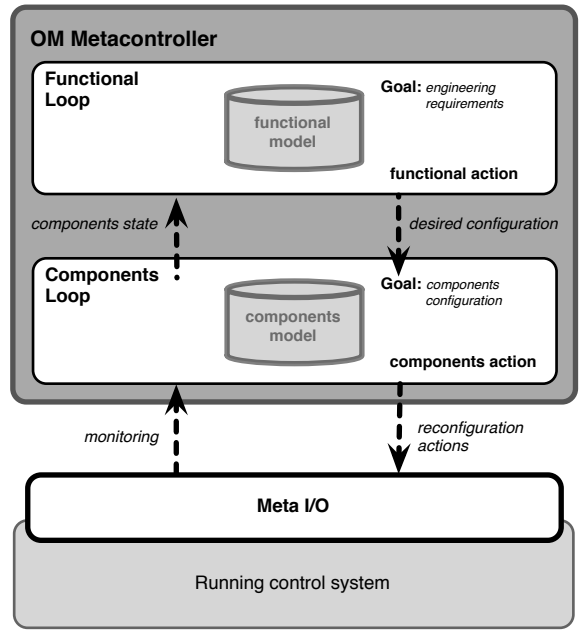


Figure 9.3: General view of the OM architecture.

Looking inside the OM Metacontroller, it is organised as a two-layered controller following the Functional Metacontrol Pattern (page 132): one layer controls the con-

trol system's organisation, and the other controls its function[2]. The activity of both layers is driven by an integrated model of the function and structure of the control system, according the Epistemic Control Loop Pattern (page 120).

At the Components Loop, the monitoring information from the Meta I/O is used to update a model of the control system in terms of its structural organisation. The resulting estimation of the current state of the components is evaluated against a desired configuration. Reconfiguration actions are determined from that evaluation and commanded if necessary to the Meta I/O module. A reconfiguration action can consist of the instantiation of new components, elimination of others, or re-wiring or modification of the parameters of those already existing.

At the upper layer, the Functional Loop takes as sensory input the evaluation of the component's organisation, and from it updates the functional state of the system captured by the `Functional Hierarchy` (described in 155). The hierarchy is evaluated in terms of the state of achievement of the root objectives. If any of them is not achieved, system design knowledge is used to compute a new configuration for the control system, which is finally sent to the Components Loop. The Functional Loop thus performs a very basic "design" activity, using the minimal designs of functions stored in the TOMASys model to instantiate a feasible design for the complete system. The *design problem* is a hard one; in the OM framework we are tackling a very simplified version according to the limited scope of the TOMASys metamodel.

9.1.4 Integration of patterns for self-aware autonomous systems

The OM Architecture has resulted from a synthesis combining the four design patterns presented in Ch. 7: MetaControl, Functional Metacontrol, Epistemic Control Loop and Deep Model Reflection. Figure 9.4 shows how the progressive application of the patterns modifies the standard engineering process of building an autonomous system that was discussed in 7. Each of them contributes to the architecture of the controller and to its engineering process. Steps 1, 2 and 3 reflect how the application of the patterns affects the runtime architecture of the control system, whereas the 4th pattern application affects the development process of the control system:

1. The MetaControl pattern prescribes the division of the control system into two subsystems, for the different concerns of producing appropriate action on the application domain, and adaptation action over the control itself.

2. The Functional/Structural Metacontrol pattern structures the metacontrol subsystem as two layered loops, one of them devoted to the control of the controller's structure and the other one to its functionality.

3. The Epistemic Control Loop pattern organises the operation of both loops around a model that contains the knowledge about the structure of the system in terms of components and how they realise the functions. The application of this pattern

[2]This is indeed a manifestation of the old philosophical duality of structure/function; note than in OM we are able to decouple both to improve system autonomy.

renders a model-based metacontroller in which the metamodel determines its capabilities, in line with approaches in self-adapting software such as [43, 53, 48], already presented in 5.2.

4. The Deep Model Reflection pattern prescribes the form of the run-time model used by the metacontroller and how it is obtained. The metacontroller model must conform to the TOMASys metamodel. This way, if a transformation model is defined from the engineering modelling language of the controller (the domain control subsystem) to the TOMASys metamodel, the run-time model could be obtained by direct transformation from that engineering model, as explained in the description of the DMR pattern (page 128).

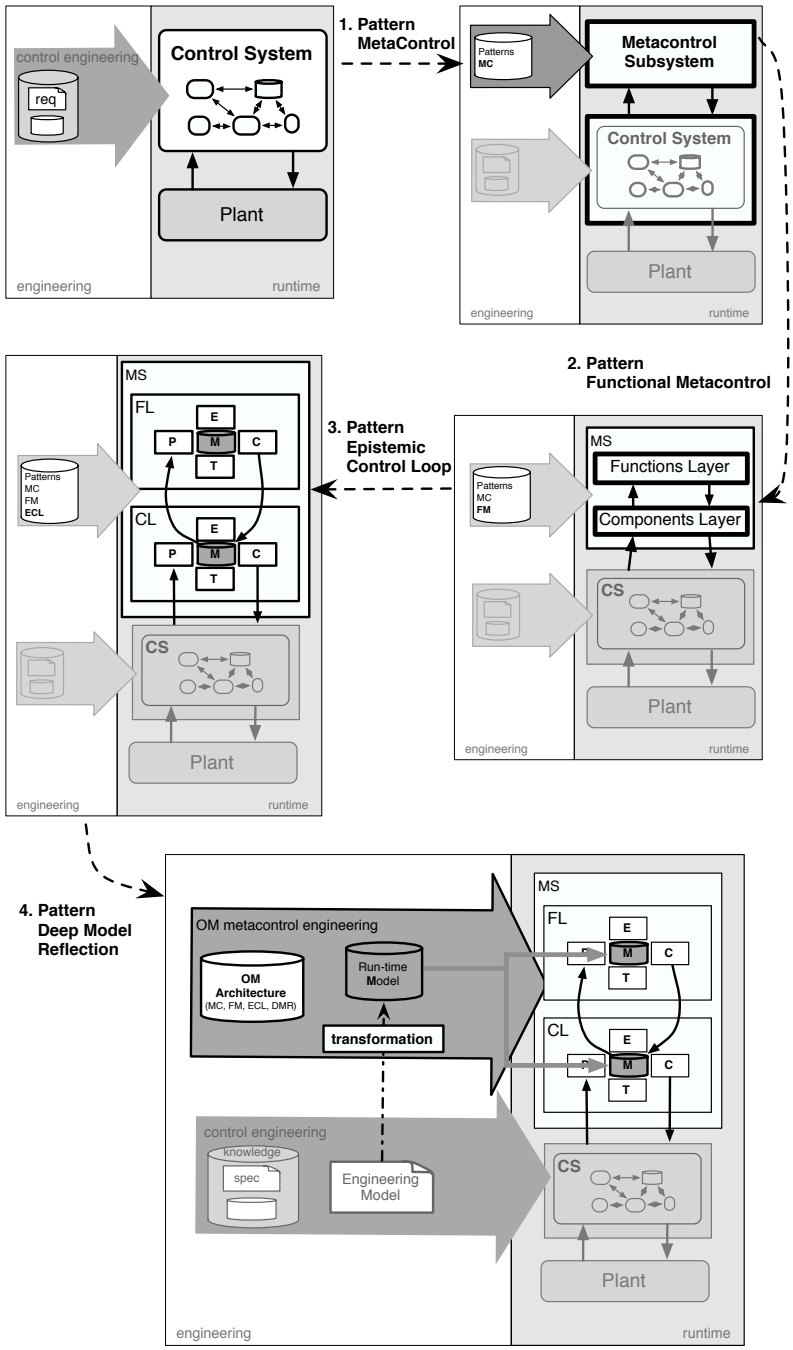


Figure 9.4: Progressive application of the four patterns for self-awareness to the engineering of a control system. The top left box depicts the traditional engineering of a control system, whereas the bottom box shows the engineering with the OM framework. Dotted arrows in between boxes represent the stepwise application of the design patterns.

9.2 Instrumenting the Domain Controller

The OM Metacontroller interacts with the domain control system by monitoring (sensory input) and reconfiguring it (action output). The raw monitoring information available, as well as the reconfiguration actions that are possible, depend on instrumenting the implementation technology of the control system, i.e. its component platform [119]. For that reason, in order to maintain the OM Metacontroller independent of the domain platform, an interface that defines the monitoring and reconfiguration signals, the MetaInterface, has been specified. An adapter module is thus necessary to adapt the available domain infrastructure to the MetaInterface: the Meta I/O module. It plays the roles of the sensors and the actuators for the OM Metacontroller, as defined by the ECL pattern (page 120).

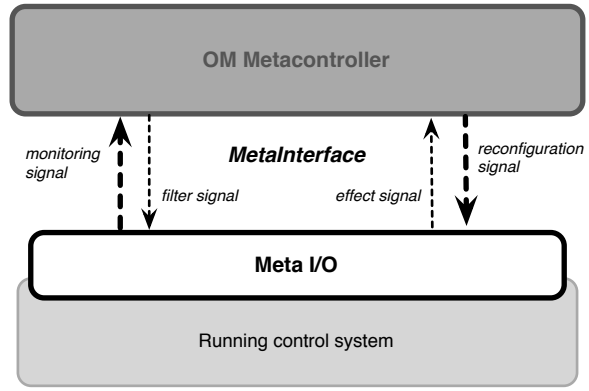


Figure 9.5: The signals between the OM Metacontroller and the Meta I/O module that are specified by the MetaInterface.

9.2.1 Meta I/O Operation

The Meta I/O module is strongly tied to the infrastructure provided by the platform of the control system, and so the OM reference architecture only prescribes which behaviour it must exhibit as a black box component. A concrete design for the internals of this module is presented in 11.4.2, in which the ROS implementation developed for the testbed of this work is described[3].

Regarding its monitoring function, the Meta I/O gathers all instantaneously available information about the relevant components of the system and sends it periodically to the Metacontroller. Which components are relevant for monitoring is decided by the OM Metacontroller based on the knowledge of the components involved in the

[3]ROS (Robot Operating System) is a state of the art component platform for robotic applications. Note that OM is general and not necessarily tied to ROS-based control systems. The ROS platform is used in this thesis as a case study.

functions defined by the model of the control system[4]. It can use the filtering to command the Meta I/O to filter-out information about certain components, e.g. to save bandwidth, or to actively gather information about the absence of other important components. More complex attentional mechanisms could be incorporated.

With respect to its reconfiguration function, the Meta I/O receives from the OM Metacontroller asynchronous commands to reconfigure the control system. These commands are only issued by the metacontroller when they are required. The command consists of a set of actions over components and their connections. The Meta I/O reports back about the execution of each of those actions through the effect signal, whether they have been successfully completed, they are still under processing, or they could not be completed.

9.2.2 MetaInterface

The signals exchanged between the Meta I/O and the Metacontroller modules are specified by the MetaInterface, which defines the interface that the infrastructure of the control system must expose to the OM Metacontroller. This interface can be divided in two, according the monitoring and the reconfiguration services it provides.

Monitoring interface

There are two data flows involved in the monitoring of the control system:

Monitoring signal: contains the monitoring information with the state of the components in the system. It consists of a vector of Component Observations:

monitoring signal:= $\{comp\_obs_1, comp\_obs_2, \ldots\}$[5]

A Component Observation is a data structure that contains information about the instantaneous state of a component of the control system. It can be considered as a "normalised" reading of monitoring data, since it only includes directly observed quantities, without any additional knowledge about the control system, and it is independent of the technology of that component, contrary to the reading directly provided by the platform-specific signal from the platform instrumentation.

A Component Observation contains of the following elements:

- *name* of the component, which identifies it in the system.
- *parameters*: it is a vector containing pairs $(param\_name, value)$, with the information of the values of the parameters of the component.

[4] note that some components and functions may be left out of the model, because they are not to be controlled by the metacontroller.

[5] EBNF-like syntax is used to define signal and data-structures in OM

- *ports*: a vector containing information about the ports of the component. Each entry contains the port name, its directionality, the connector it is connected to and its type.
- *log*: it is a string containing the latest log entry produced about the component, i.e. any information not fitting in the previous fields and admitting to be stored as a string.

Monitoring information about the connectors between the components is implicit, embedded in the *ports* section of each Component Observation.

Filtering signal: it contains two vectors: one with the names of the components about which information is required, the other with the names of the components about which information shall be omitted, not being of interest at that moment for the OM Metacontroller.

$$\textit{filtering signal} := \begin{bmatrix} \textit{observe} := \{comp\_name_1, comp\_name_2 \ldots\} \\ \textit{omit} := \{comp\_name_a, comp\_name_b \ldots\} \end{bmatrix}$$

Reconfiguration Interface

The Reconfiguration Interface defines the signals that are interchanged between the OM Metacontroller and the Meta I/O for the execution of reconfiguration actions.

Reconfiguration signal: it is sent from the Metacontroller to the Meta I/O module. It is a command consisting of a set of reconfiguration actions to be executed on a corresponding set of components. Each action is defined by a unique identifier, a type that identifies its nature, and a set of arguments.

$\textit{reconfiguration signal} := \{action_1, action_2, \ldots, action_n\}$
$action := (action\_id, type, args\{(arg_1\_name, value), \ldots (arg_n\_name, value)\})$

The nature of the actions is described later in the component action vocabulary.

Effect signal: the reconfiguration interface includes a feedback signal from the Meta I/O module to the OM Metacontroller, to inform it about the state of execution of the commanded actions.

$\textit{effect signal} := \{(action\_id, action\_status), \ldots\}$
$\textit{action status} \in \{PROCESSING, SUCCESS, FAILURE\}$

9.2.3 Component Action Vocabulary

The action vocabulary is the set of primitive actions the OM Architecture defines for the reconfiguration of components. They are:

LAUNCH: initialises and starts the execution of a component.

RESUME: resumes the execution of a component which is initialised.

STOP: halts the execution of a component, which maintains its configuration but loses its internal state.

PAUSE: suspends the execution of a component, which maintains its configuration and internal state, but produces no output.

KILL: stops and removes a component from the system.

RESET: puts the component in the default configuration (the factory one, not necessarily the one we want it to operate with).

CONFIG: changes the configuration of a component: parameter values and port connections.

Each of these types of actions takes a different number of arguments. The action types RESUME, STOP, PAUSE, KILL and RESET take a single argument, which is the name of the component to act upon.

The LAUNCH action takes as arguments:

- a vector of strings containing the deployment information for the component that is to be launched in each element:
  - the name of the Component Class to be deployed,
  - the place where to deploy it, typically a computation node.
- a Component Specification data structure, which contains the configuration information:
  - a vector of $(param\_name/value)$ pairs with the values for the component's parameters,
  - a vector of $(port\_name/connector)$ pairs with the connections to be set for the component.

The CONFIG action takes two arguments:

- the name of the component to be reconfigured,
- a Component Specification data structure with the new configuration for the component. It only contains $(name, value)$ pairs for those parameters and ports that need to be modified.

This vocabulary could be extended with other possible commands, such as:

SELF_TEST: to initiate a testing process in a component so that it sends current information to the monitoring infrastructure. A pro-active sensing.

CHECKPOINT: Save a copy of the state to recover it sometime in the future. This can be used for a pro-active perception of the `internal state` of the component too.

TRACE: Use logger component to trace the behavior of the component. A parameter can be the trace level. This is another "attentional" mechanism affecting the bandwidth of the input for the perception of the `log` of the component.

MOVE: to re-deploy a component somewhere different without losing its state.

REPLACE: Ask to negotiate with other component the exchange of roles in a system.

9.3 OM Metacontroller

The OM metacontroller is a model-based cognitive controller, as they have been proposed in our principled approach presented in section 6.1.2. Its goal is to maintain the functionality of the control system. For that it keeps a representation (the OM Model) of the functional state of the control system and analyses it to decide if a reconfiguration action is required. The model is constantly updated with the monitoring information coming from the Meta I/O module. The reconfiguration action is generated using the design knowledge about the control system that is contained in the model, and implemented by commanding the Meta I/O module. It may involve the activation of new components, the elimination of extant ones, and/or, most commonly, the re-configuration of the components, by changing the values of their parameters and/or their connections to others.

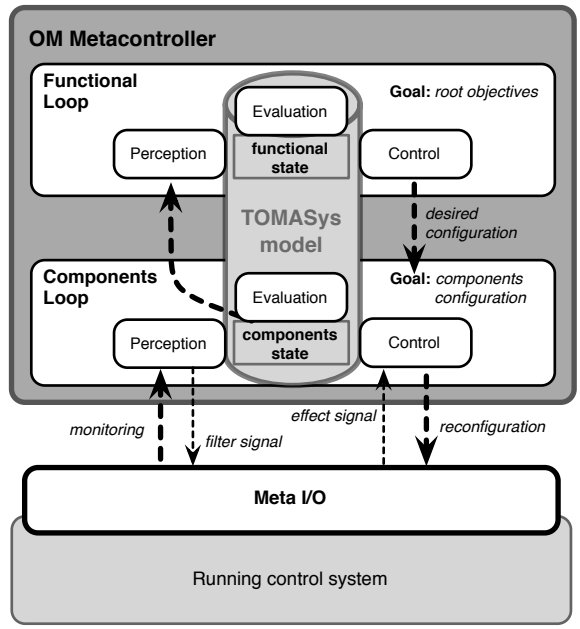


Figure 9.6: The main elements of the internal structure of the OM Metacontroller. Both the Components and Functional loops follow the ECL pattern.

9.3.1 Epistemic Control Loops for metacontrol

The OM Metacontroller is organised according to the ECL pattern: the perception, evaluation and control activities are integratd and coorditaned through the central OM Model. However, it does not consists of a simple ECL unit. The OM Metacontroller consists of two layered control loops, following the Functional Metacontrol pattern.

The upper loop, the Functional Loop, has the target goal of achieving the root objectives of the autonomous system. The lower layer, the Components Loop, functions as a servo-controller, trying to maintain the configuration of the system compliant with that commanded from above. Each loop is an ECL unit that exploits a shared TOMASys model of the control system. Their operation is detailed in sections 9.4 and 9.5.2, which discuss the details of their perception, evaluation and control activities.

Note that the Components Loop is a feedback loop —i.e. it is always late in its response—, reacting to components' failures or undesired states. However, the Functional Loop can be feedback, reacting to current functions' failures, but also anticipatory, correcting in advance functions' failures that are not present but predicted from the causal relations in the functional hierarchy.

9.3.2 OM Model

The cornerstone of the operation of the OM Metacontroller is the Model of the control system: it is the knowledge exploited according to the Epistemic Control Loop pattern. The OM Model is an integrated model of the functions and components of the controller, and is shared by the Functions and Components loops. The OM Model consists of data structures that contain static and dynamic information about the operation of the control system in terms of its functional and components structures.

On one hand, it is a symbolic representation that conforms to the TOMASys metamodel, which provides the domain ontology knowledge for the OM Metacontroller. It defines the entities that exist in the plant —i.e. functions and components— their properties and relations —i.e. parameters, ports, connections, solve and require relations— and the basic semantics for them.

On the other hand, the OM Model is organised according to the ECL Knowledge Repository design pattern for cognitive controllers compliant with the ECL pattern. This structural pattern organises the elements in a repository for their use in an ECL-based controller. In addition, the KR pattern defines semantics for the model elements for the ECL controller (in this case the OM Metacontroller) to operate with them in the control loop. These semantics are specified by the ECL Metamodel.

We can represent these considerations about the OM Model in a metamodelling relation in which any OM model conforms to the OM Metamodel, whose elements inherit properties from both the TOMASys metamodel and the ECL Metamodel (see figure 9.7). This way the Model in an OM Metacontroller integrates the operational semantics and organisation specified by the ECL Metamodel and the Knowledge Repos-

itory pattern, with the domain ontology and semantics about autonomous system organisation and directiveness defined by the TOMASys metamodel.

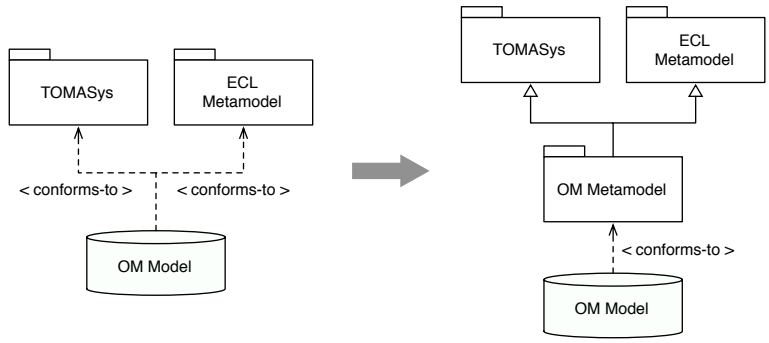


Figure 9.7: The metamodelling approach enforced by the ECL KR pattern as it has been applied in OM to produce the OM Metamodel.

ECL Knowledge Repository and Metamodel semantics

Let us start by briefly discussing the ECL KR pattern. The ECL *Knowledge Repository* is a pattern that structures the Model of an ECL unit as a repository (*KR*) of data structures that contain symbolic representations of the plant.

The elements of the Knowledge Repository are *knowledge atoms* or simply *atoms*, and other conceptual quantities corresponding to simpler pieces of information.

knowledge atom: atomic piece of data in the *KR*. It is a chunk of information or knowledge characterised by having semantics for its perception and evaluation.

There are two types of of atoms in the *Knowledge Repository*:

state atom: is an atom of knowledge that describes the state of an instance in the plant, be it actual or desired, e.g. a goal. This instance can be as simple as a single quantity or variable, such as the temperature, or as complex as a complete entity, e.g. an obstacle, in which case the state atom may include several fields, e.g. geometric position, size, colour, etc., for representing the relevant quantities of the entity. These fields containing the instantaneous state of the atom are domain dependent and they are specified by the *type* of the state atom if known, which is a concept atom.

concept atom: contains static, long-term knowledge about the entities in the world and their properties, or procedural knowledge about actions. This way, concept atoms contain the implementation of the semantics for the types of entities in the plant, for example by defining methods like the following:

- *recognise(atom)* gives a value related to the membership of the atom element passed as parameter to the type of entity defined by the knowledge

atom.

- *measure(atom1, atom2)* contains the metrics for comparing two entities. If they pertain to the same type of entity, and their state is measurable, the difference between them is given, in addition to a measurement of that difference in the real interval $[0, 1]$ with 0 representing the case the two atoms are identical.

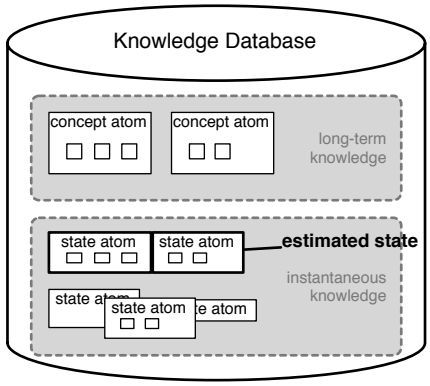


Figure 9.8: The entries in an ECL Knowledge Repository

According to these two basic types of entries, the knowledge in the KR is divided in two parts:

instantaneous information: it includes information about instances in the plant. It consists of state atoms and associated entries. They may refer to the current state of the plant, a past one or predictions of future or hypothetical states and entities. It encompasses the two packages of knowledge usually named in cognitive architectures as the immediate experience and short term memory. Information about the current course of actions is also part of the instantaneous information stored in the KR.

long-term knowledge: it is made of the concept atoms. It contains more permanent information about the types of entities that exist in the plant, their properties and their behaviour. The long-term knowledge also includes procedural knowledge: types of actions, inverse models about actions' effects, and algorithms and domain heuristics related to action decision and planning.

The knowledge repository contains two static elements that are fundamental in the operation of the ECL unit:

Estimated State: it is the set of state atoms that account for the representation of the current state of the plant.

Goal: it is a set of state atoms that represent the desired state of the plant that the ECL unit is designed to achieve or maintain.

Domain-specific semantics for evaluation need to be provided so as to measure the difference between the *estimated state* and the *goal*, or *error signal*.

Following we present the definition of the *KR* elements and their semantics that we have developed for the domain-specific case of the OM Model.

OM Model elements

Considering the OM Model a particular case of an ECL KR repository, its elements are particular instances of the TOMASys elements, extended with the operational semantics discussed before.

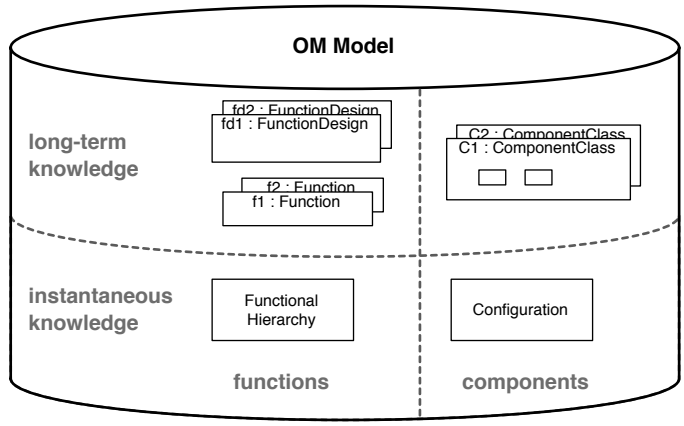


Figure 9.9: The elements in the OM Model.

This way, in the OM Model the *state atoms* in the *instantaneous knowledge* are instances of the following OM Metamodel elements: OMComponent, OMPort, OMParameter, OMObjective and OMFGrounding, which inherit their properties from the corresponding TOMASys elements. They represent the current `configuration` and `functional hierarchy` of the autonomous system.

Correspondingly, the *long-term knowledge* contains instances of the following *conceptual atoms* : OMFunction, OMFunction Design, OMRole, OMComponentClass, OMParameter and OMPortProfile. The autonomous application-specific knowledge is captured in the TOMASys `Application Domain Model` discussed in page 140. These metamodelling relations are identified in figure 9.10. The OM Model contains not only information about the elements that conform the control system, but also semantics for them. Part of these semantics are embedded in the OM Metamodel, but another part is defined for the concrete system. These semantics include the TOMASys Application Domain Model, but also rules for perception and action related to the ECL-based operation of the OM Metacontroller. For example, the rule that two components are of the same `ComponentClass` if they have the same profiles of `ports` and `parameters` reifies a perceptive function for categorization. The rule to decide if

two OMComponent atoms correspond to the same component in the control system, for example if the values for their parameters and ports are the same, corresponds to a possible perception function of identification. Using the mismatches between two components can be used as an *evaluation* function to check if an OMComponent atom satisfies an OMComSpec.

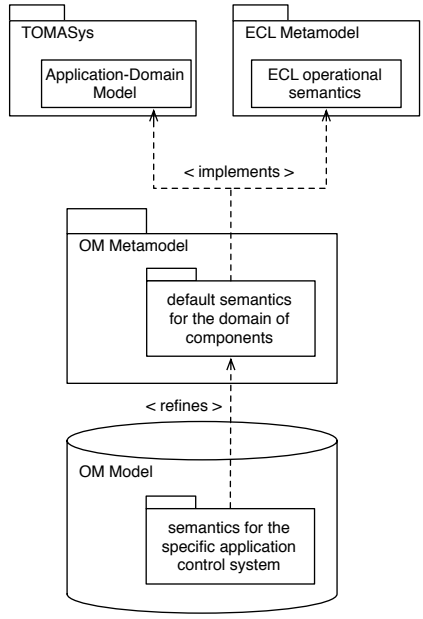


Figure 9.10: Metamodeling relations of the OM Model. The OM Metamodel implements default semantics for ECL operations, and also for the TOMASys ADM. These semantics can be further refined in the concrete OM Model for the autonomous system.

The knowledge in the OM Model is divided in terms of elements that represent the functional design of the system, used by the Functional Loop, and elements that capture its grounding in terms of components, which is the responsibility of the Components Loop. A more detailed explanation of how each one of the loops uses its corresponding part of the Model is given in the following sections.

OMComponentSpecification: linking Functional and Components loops. A special element, with no corresponding TOMASys counterpart, has been added to the OM Metamodel: the OMComponentSpecification (or OMCompSpec for brevity). This element encodes the specification of a configuration for a component. It encapsulates a specification of the properties of the component that corresponds to its configuration, i.e. port connections and parameter values, and not to its internal state. It includes two additional properties: a compliance(spec, s) function, that defines the semantics to compute for a given OMCompSpec atom spec, if it is satisfied by another atom s, which can be either another specification, or an OMComponent representing an actual

component in the system. Let us remember that a TOMASys `role` defines a template specification of a component that participates in a `Function Design`. In the binding when the Function Design is grounded, that templatized specification corresponding to the role is given concrete values and becomes an actual specification. This is what is represented in the OM Metamodel by the OMComponentSpecification modeling element.

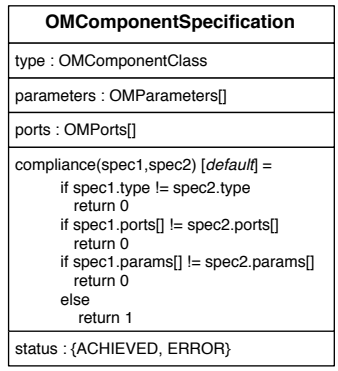


Figure 9.11: Properties of the OMComponentSpecification metamodel element.

The OMCompSpec allows to decouple the Components and the Functional loops (figure 9.12). On one hand, the goal of the Components Loop is defined by a set of atoms instances of OMCompSpec, which represent a desired reference configuration of the system. At runtime, the OM Metacontroller updates the status of these specifications by evaluating if the *estimated state* of the configuration of components in the system satisfies it (arrow lines from the atoms in the Components Configuration Estimated State to the specifications in the Components Goal). On the other hand, at the Functional Loop, within the bindings defined by the function groundings in the Functional Hierarchy, the roles have as realisers the OMCompSpec instances that compose the Components Loop goal.

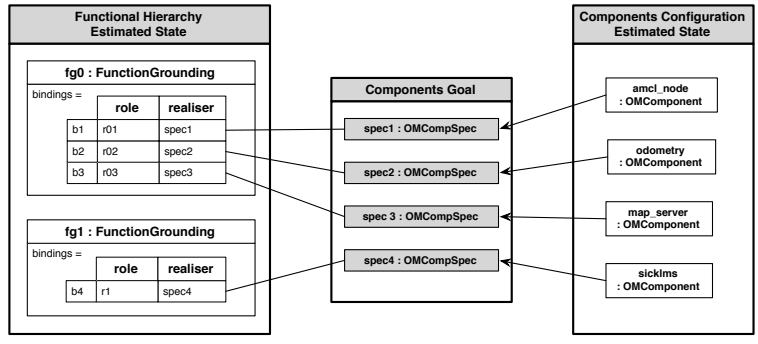


Figure 9.12: The specifications in the Components Goal relate the estimations of the actual components in the system with the roles they fulfil in the functional hierarchy.

Example

OM Model of the localisation subsystem

The OM Model of the localisation subsystem contains elements that represent the configuration of components and the functional hierarchy as described in the examples of chapter 8. They are shown in figure 9.13

The ECL semantics are provided by the information corresponding to the TOMASys `Application Domain Model`.

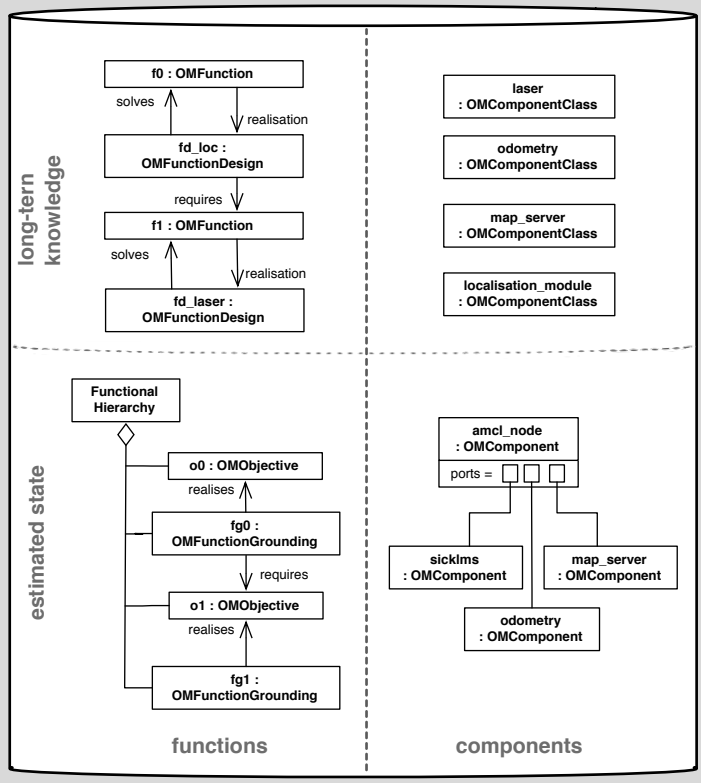


Figure 9.13: Example: OM Model of the localisation subsystem.

9.4 Components Loop

The Components Loop is an ECL unit that targets as goal a certain desired configuration of the control system. To achieve it, the Components Loop senses the current system configuration using the Monitoring interface of the Meta I/O module, and manipulates the current configuration of components using the Reconfiguration Interface. The Components Loop follows a feedback schema and its operation consists of a periodic executive cycle of sequential ECL activities: perceive → evaluate → control. In this section we describe the different ECL processes involved in the Component Loop operation, which are sketched in figure 9.14.

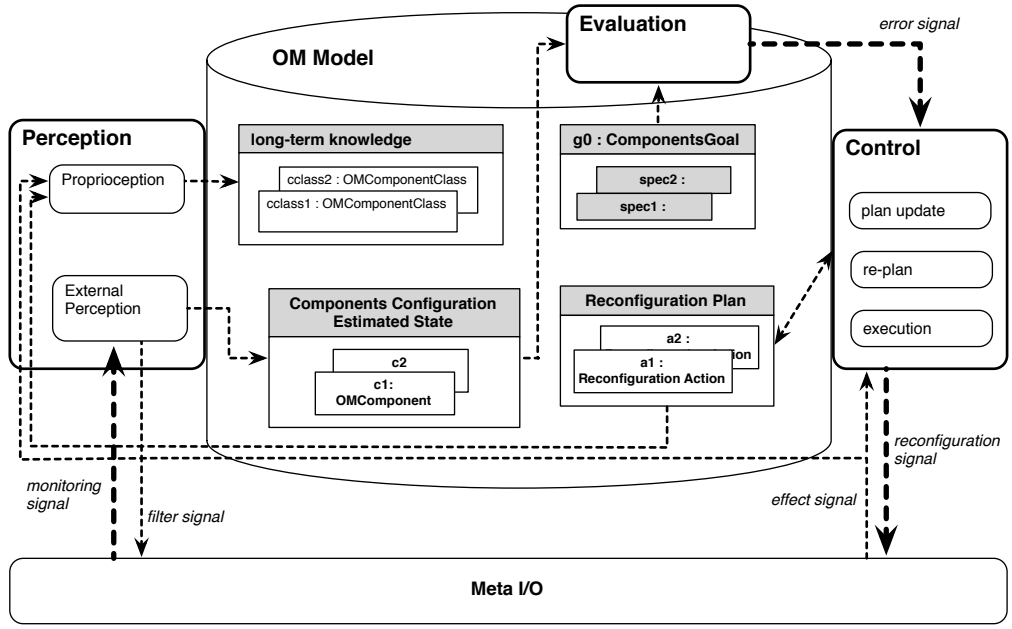


Figure 9.14: Basic model elements, processes and flows of information in the Components Loop.

9.4.1 Components Model

The knowledge exploited by the Components Loop are instances of elements in the OM Metamodel that inherit from TOMASys organisational elements: Components, Connectors, Component Classes, etc.

Components Goal: the goal of the components loop is a set of component specifications, which represent the desired configuration of the control system $< spec_1, spec_2, \ldots, spec_n >$

Each specification is an atom instance of the OMComponentSpecification

The Components loop works as a servo-controller acting to make the estimated configuration of components coincide with the one specified in its goal.

Estimated State: the estimated state in the Components Loop is a TOMASys `configuration` of `components`: $< component_1, component_2, \ldots, component_n >$, representing the current state of the components of the control system.

Long-term knowledge consists of concept atoms that are instances of the OMComponentClass, and other atoms that define their properties, e.g. instances of OMPortProfile and OMParameterProfile. These atoms define the properties of the components in the autonomous systems: their parameters, the ports they expose, and their error models.

The OMComponentClass provides a default `failure model` for components. It specifies that if any of the connectors of the input ports has no components connected with an OK status (i.e. the component is not receiving any of its required inputs), or an internal failure has arisen, then the component is in FAILURE. This general failure model may be overridden if a particular one is defined in the OMComponentClass atom of the component.

In the OM Metamodel, TOMASys elements have been augmented with semantics for perception and evaluation. These will be discussed later when presenting the perception and evaluation activities performed in the Components Loop.

Example

Representation of a laser sensor in the OM Model

Let us detail the representation of one of the components of the localisation subsystem, for example the laser sensor.

The state atom sicklms encapsulates the information about the instantaneous state of the laser sensor:

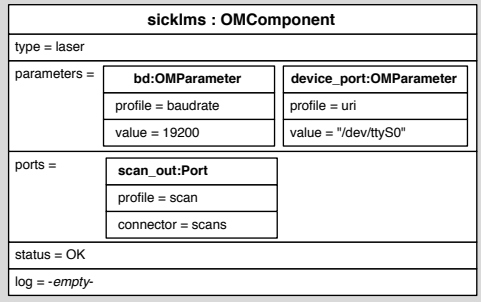


Figure 9.15: Example: the estimated state of the laser sensor.

The knowledge about the characteristics of the sensor, e.g. the types of its parameters and ports, its failure models, is coded in the concept atom laser:

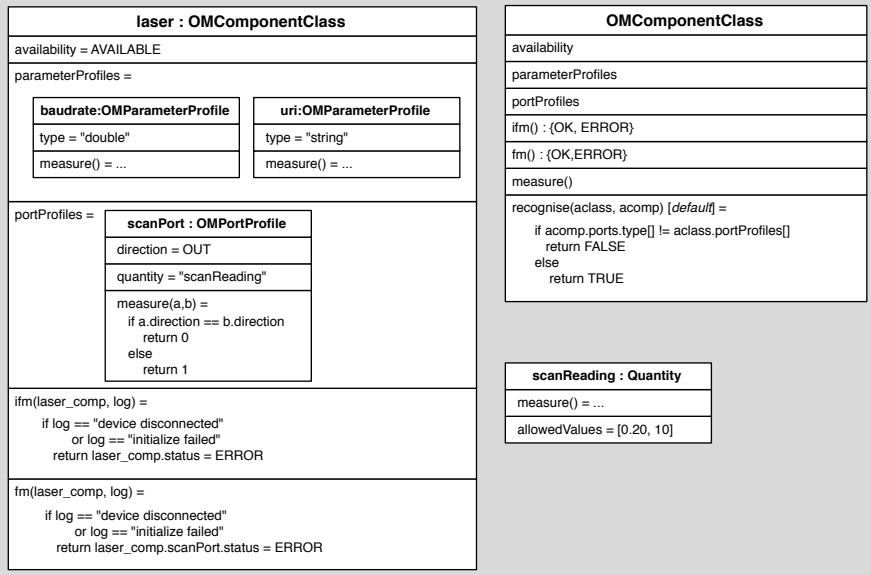


Figure 9.16: UML-like representation of the conceptual atom that stores the knowledge about the robot's laser sensor.

The `ifm` and `fm` failure models that the laser OMComponentClass implements are those described in the example about failures models in page 149.

An example of application-domain metrics is shown in the scanPort *concept atom*. This element encodes the knowledge about the output interface of scan range sensors.

For the moment, the `Application Domain Model` is basically limited to information about the organisation elements (concept atoms of type OMComponentClass).

9.4.2 Components Perception

At the beginning of each operation cycle, the Components Loop updates the estimated state and the availability of components in the system. The first objective is achieved using the Meta I/O monitoring, whereas for the second one the effect signal is also used. From a cognitive control point of view we could talk of two different perceptual activities: external perception in the first case, and proprioception in the second.

External perception

The external perception process updates the estimated state in the Components Loop at each cycle, using the monitoring signal incoming from the Meta I/O. This process follows the perception model of López [90], depicted in figure 9.17.

According to López's model, the perceptive process of an autonomous system analyses the values of environment magnitudes arriving to the sensors, which is called *proximal stimulation*, and generates a representation in the system, i.e. a percept.

The perceptive process identifies and characterizes the presence of certain entities in the environment. These entities concepts that are relevant for the objectives of the system, and are called *referents* of the perceptive process. When an instantiation of a referent is identified in the environment and characterized, it is conveyed to the system through the representation system as a *perceived object* or *percept*.

Referents usually cannot be sensed directly, but inferred from a series of patterns in the proximal stimulation input which are singular to them. These are called the *singularities*.

A generic perceptive process will then present two phases. First, identifying singularities in the proximal stimulation, and then associating them to the existence of instantiations of the referents in the environment. The first phase shall be called proximal information processing. The second, cognitive information processing.

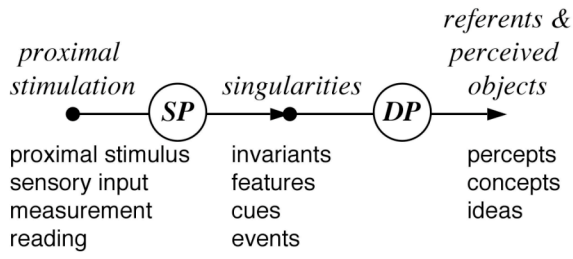


Figure 9.17: Phases and main elements of a perceptive process, from [90]. Over the arrow López's terms are indicated. Below, alternative terms from the literature.

In an OM Metacontroller, the *sensory processing* is performed at the Meta I/O, and results in the component singularities that conform the monitoring sugnal. The external perception thus consists of the *cognitive perception*, in which the monitoring signal is processed to identify in it the components of interest in the system, which are the *referents* of the perception process. These referents include the OMComponent atoms in the currently estimated state, the OMComponentSpecification instances in

the Components Goal, and the OMComponentClass concept atoms corresponding to both sets.

For each component observation in the monitoring signal the process results either in an update of the information of one of the components already in the estimated state, or in the generation of a new OMComponent entry in the estimated state. The external perception follows the following activity flow:

1. Update referents with the components in the estimated state and in the current Components Goal. Additionally, a filtering signal is produced from the referents, containing the names of the referents, and it is sent to the Meta I/O.
2. For each Component Observation (CO) in the monitoring signal, the following actions are performed:
   (a) Identification: identify if the singularity corresponds to one of the referent instances. For this, the identification model for Components is used.
   (b) if the Component Observation is identified as corresponding to a Component in the estimated state, that Component is updated with the information in the singularity.
   (c) if not, then:
   the observation is matched against the OMComponent lass atoms corresponding to the classes of the referents using their recognition model. If the singularity is recognised as corresponding to a component instance of one of the classes, then a new OMComponent instance of that class is added to the estimated sate, and filled with the information of the singularity.

Proprioception

The effect signal containing the feedback about the execution of reconfiguration actions is asynchronously received from the Meta I/O. These feedbacks are buffered and processed in each cycle of the Components Loop after the monitoring signal. Two different perception activities are performed over the different feedbacks contained in the effect signal:

- The status of the reconfiguration actions issued is updated according to the feedback, e.g. if an action has been successfully completed, or if its execution has failed; this process is described in detail in page 189.
- The feedback information about the failure of an action usually conveys information about the component over which the action was performed. For example, if the execution of a LAUNCH action over a component could indicate that its corresponding Component Class is no longer available. The OMComponentClass extends the `ComponentClass`, adding an availability() function to it, which adds more knowledge about the class of component, in addition to the `error models` already defined by TOMASys. This function takes as input

the entries in the effect signal corresponding to components that are instances of the OMComponentClass, and associated log information in the monitoring infrastructure, and updates its status of availability accordingly.

9.4.3 Component Evaluation

After an estimation of the current state of the system configuration is obtained in the perception process, the state of achievement of the Component Goal is assessed accordingly. Let us remember that the Goal of the Component Loop consists of a set of OMComponentSpecification atoms, which encodes a desired configuration of components in the control system. The complies() functions of the specifications are used as evaluation policies to determine, for each OMCompSpec, when an OMComponent atom in the estimated state fulfils it.

The set of evaluation policies is computed at the functional level from the compliance functions of the roles that define the desired configuration. This is explained later in section 9.5.3.

The evaluation process can be decomposed in two steps:

1. Firstly, an error signal is computed as the difference between the estimated state and the Component Goal. For this measurement metrics for components are defined in the domain application model, included as measure(c1, c2) functions in each OMComponentClass, which output the differences between two OMComponent atoms.
2. Assess the fulfilment of the Goal according to the difference, by using the evaluation policies for each specification in the goal.

Error Signal

For the computation of the error signal, specific metrics for the domain of components are provided in the OM Metamodel as part of the TOMASys Domain Application Model. Besides, the Difference metamodel element has been added to account for distances between instances of the Organisational elements of TOMASys. A Difference knowledge atom represents the relation between two atoms of the same type, for example a parameter, being the one corresponding to the Component in the goal the *origin*, and the one in the estimated Component the *end point*. The Difference also has the attribute n_diff, which is a numerical value of the difference in the range [0..1]. The n_diff is determined by metrics for that element defined in the `application domain model`.

The error signal is a vector that contains one element per each entry that specifies a component in the component goal. Each element can be either null if there is a component in the estimated state that fulfils the goal, or a Component Difference otherwise.

$error\_signal = < ComponentDifference_1, \ldots, ComponentDifference_n >$

if there are n component specifications in the goal.

A Component Difference is a data structure consisting of a vector of differences between the specification and the component in the estimated state that is considered to correspond to that specification.

Supposing that there are m differences in between the ith component specification and its corresponding component in the estimated state:

$ComponentDifference_i = < difference_1, \ldots, difference_m >$

Evaluation

For each specification in the current goal, the associated Component Difference with the estimated state is evaluated according to the evaluation policies provided with the specification. The evaluation policies are functions:

$eval : C \rightarrow [0,1]$
that define for each component if it fulfils the specification (C is the set of all components).

A very simple example of evaluation policy could define that if there is any difference, as obtained in the error signal using the domain metrics, then the component does not comply with the specification ($eval = 0$).

A more complex example of evaluation policy could use the sum of the numerical differences obtained between the component and the specification, and compare it with a certain threshold value, below which the specification would be considered as achieved ($eval = 1$). This evaluation policy can be formalised with the following equation: $\Delta_i = \sum_k \delta_k \cdot w_k$

if $\Delta_i \geq \sigma \rightarrow$ the specification is not achieved

where:
Δ_i is the numerical value computed for the ComponentDifference of the *ith* component specification
δ_k is a numeric value for each Difference entry in the ComponentDifference (provided by the Component Metamodel)
w_i is a weight defined by the evaluation policies of the specification
σ is the threshold defined for component specifications

The result of the evaluations processes is a set of evaluations that contains, for each specification in the components goal, a status value $\in \{OK, ADDRESSED, ERROR\}$ that defines if the specification is achieved, and an error signal, which is used in the control process to compute corrective reconfiguration actions.

The status of an specification can be OK if it is achieved, ADDRESSED if it is not but there is an ongoing action to achieve it, or ERROR if it is not achieved.

Example

Here is an example of the evaluation of an entry in the specifications of the Components Loop goal that specifies a laser component in the desired configuration. It is depicted in the specification on the right, whereas the corresponding entry in the estimated state is shown on the left.

c1 : Component	
name	'sicklms'
type	laser
ports	scan
parameters	baudrate = 19200

cs1 : ComponentSpecification	
name	'sicklms'
type	laser
ports	scan
parameters	baudrate = 19000

Let us consider that for the `Parameter Profile` baudrate we define domain metrics such that the numerical difference is 0 only if the two values of the baudrate parameter are equal, and it is 1 in any other case. The corresponding entry in the error signal vector would be:

ComponentDifference	
param. - baudrate	n_diff = 1

The evaluation of *c1* according to the specification *cs1* —following policies like those previously described— would be:

$\Delta_i = \sum_k \delta_k \cdot w_k = 1 \cdot 0.1 = 0.1 \rightarrow$ specification not achieved

where:

$\delta_k = 1$ the difference corresponding to the baudrate parameter

$w_i = 0.1$ given than a different baudrate may not be acceptable

$\sigma = 0.1$ is the default threshold for component specifications

9.4.4 Components Control

The Component Loop computes, at each cycle, a control action that is the reconfiguration signal sent to the Meta I/O module. The reconfiguration required may involve different actions over different components. These actions may not be independent one from another; e.g. the reconfiguration of a component may require that those components using its outputs shall be restarted afterwards. That is, reconfiguration actions shall be executed following a plan.

For this reason the Component Loop uses an action-planning pattern to compute and execute its control action.

At each cycle, the control process takes as input the current error signal, and computes a reconfiguration plan. Then it sends the reconfiguration command to the Meta I/O module.

The reconfiguration plan stored in the Components Loop consists of a set of Reconfiguration Actions. Each action is computed for one of the specifications and may

require other actions.

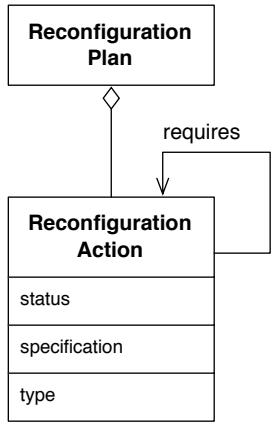


Figure 9.18: UML class diagram showing that the Reconfiguration Plan is composed of Reconfiguration Action atoms.

The control process then involves three activities: **reconfiguration plan update**, **re-plan** and **action execution**. Following we describe step by step the whole process.

1. First, the action feedback is processed, and the status of the actions in the **reconfiguration plan is updated** accordingly (see figure 9.19):

- if a success is received for an action a1:
  - the action is eliminated from the current plan.
  - the action is eliminated from the set of required actions of those that require it.
- if a failure is received for an action:
  - the action is eliminated, and the associated specification status becomes ERROR.
  - all the actions required by a1 and requiring a1 are canceled and eliminated from the reconfiguration plan.

2. **Re-plan**: the error signal is processed to compute new actions and include them in the current plan if needed:

- if the specification is achieved, all waiting actions related to it in the reconfiguration plan are cancelled. Any processing actions are allowed to finish.
- if the specification is not achieved and it has not been addressed, a suitable action plan is computed. An action plan contains a set of actions that achieves the specification. To obtain the plan, the inverse model of the Component Class of the component the specification refers to is used. This inverse model is defined in the `application domain model` and specifies the action commands that can be applied to reduce or eliminate a difference.

- if the specification is not achieved but it has been addressed, and thus there are processing or waiting actions in the current plan to achieve it, nothing is done.

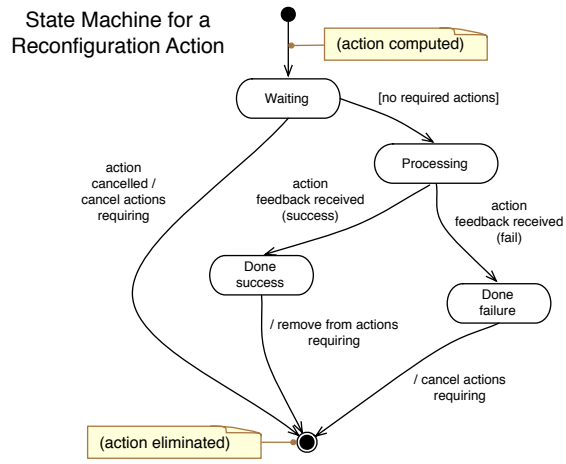


Figure 9.19: The different states of a Reconfiguration Action computed, and the transitions between them.

3. Finally, all those actions in the reconfiguration plan that have left the waiting state because they do not require any previous actions are added to the reconfiguration signal and sent to the Meta I/O module.

9.5 Functions Loop

The Functions Loop is another ECL unit on top of the Components Loop, and is responsible of maintaining the grounding of functions that best fulfils the control system's requirements, i.e. its `root objectives`.

Functional Goal: the goal of the Functions Loop is to guarantee that the root objectives can be achieved.

To achieve this goal the Functions Loop commands the Components Loop an action consisting of a certain desired configuration of components that best fulfils the root objectives. This desired configuration is a set of `Component Specifications` that define the desired *configuration* for the components of the domain control system. These specifications define the minimal set of components that grounds all the functions that address the system's root objectives. The Functions Loop receives as sensory input from the Components Loop the evaluation of the specifications with the current estimated state of components. From it the Functions Loop updates the state of the Functional Hierarchy of the system, assesses how well it achieves the root objectives, and computes any required design reconfiguration.

9.5.1 Functions Knowledge

The knowledge exploited by the Functions Loop contains instances of the functional elements of TOMASys: Objectives, Functions, Functions Designs, Functions Groundings, etc.

Estimated state: the estimated state at the Functions Loop consists of state atoms representing the instantaneous `Functional Hierarchy` of the system, and thus are instances of `Objective` and `Function Grounding` metamodel types.

The long-term knowledge at the Functions Loop consists of the functions that can be grounded in the system to realise its functionality. It thus consists of a set of `Functions Designs`, the `Functions` they realise, and the `Roles` they define.

The TOMASys model defines some of the semantics for these elements, which define the perception and evaluation processes at this functional level. These semantics are complemented with application-specific ones, defined in the domain application model. They are encoded in the compliance functions of Roles, and the error models of Function Designs. They are described in the following sections.

9.5.2 Functions Perception

The perception process in the Functional Loop updates the functional knowledge in the OM Model, both instantaneous and long-termed. It involves two activities: updating the currently estimated state of the system at the functional level, which drives the behaviour of the Functions Loop, and updating the functional knowledge, which constrains that behaviour. The estimated functional state drives the behaviour of the Functions Loop, because it generates control actions to achieve the root objectives. The functional knowledge constrains that behaviour because the functional groundings decided by the Functions Loop must be realisable, for example because only available Functions Designs can be grounded.

These perceptual activities take place at the beginning of each cycle in the Functions Loop. Following we describe both.

Estimating the functional state: update the Functional Hierarchy

As previously commented, at the Functional Loop the estimation of the state, obtained by the perceptual activity, consists of the upstream propagation of the status of the elements of the Functional Hierarchy of the system. This eventually determines the state of the root objectives, which is the goal of the loop.

The Functional Hierarchy is a set of Objectives and Function Groundings related by `requires` and `realises` associations, going downstream from the root objectives. The perception process of the hierarchy thus consists of the partial update of the status of the elements in the hierarchy considering their internal failure models, and their integration, as depicted in the activity diagram shown in figure 9.20. The sensory

input used in this perceptive process is the evaluation produced at the Components Loop: the status of the component specifications and the associated estimated state of components.

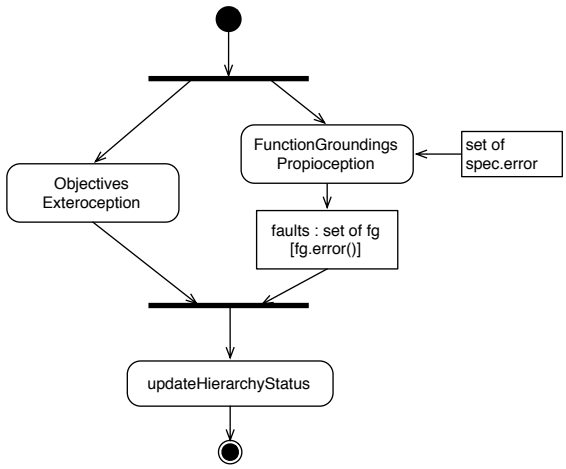


Figure 9.20: Activity diagram of the update of the Functional Hierarchy.

Update of the status of the Function Groundings: The status of the `ffunction groundings` is updated using the compliance functions of their roles and their internal failure models (defined in their respective Function Designs), the input being the updated state of the Components Specifications in the Component Goal, which is produced by the evaluation at the Components ECL. A default internal failure model can be used, which consists of the following rules:

- a binding is in error if the component that fulfils the specification, which in turn plays the `Role`, is in INTERNAL_FAILURE. This is a default compliance function for Roles.
- a function grounding is in internal error if any of its bindings is in error.
- additionally, a `Function Grounding` becomes in PERMANENT_FAILURE if the `Functions Design` it realises is UNREALISABLE.

More complex compliance functions and internal failure models could be considered so, that, for example, transitory error states are ignored, and a Function Grounding does not become in FAILURE unless one of its bindings has been in ERROR for more than a certain time.

Update the status of Objectives: Some of the objectives in the functional hierarchy could be observable. For example, in the case of the sample localisation subsystem, a software instrument can be implemented to detect whether an estimation of the robot position is computed with the desired frequency. This is the case in which the objective directly refers to a conceptual quantity in the control system, that is a TOMASys `connector`. For an objective to be observable a perception model must be provided

for the objective function, and the sensing information it requires must be available. These requirements are application specific.

The case of updating the status of an objective from internal signals that refer to connectors can be considered as a case of *direct observation*. There are cases in which log information of the components that realise the design grounded to achieve the objective can be used to detect if the objective is being achieved or not. This saves the implementation effort of building probes. In the case of our localisation subsystem, the log of the localisation component provides error messages that inform whether an estimation is being produced or not.

Direct observation: the perception model uses readings of the quantities the objective refers to.

Indirect observation: the perception model uses other information, for example log information in the components that interact with the objective's referred quantities.

Integration: The integration of the information of the status of Objectives and Functions Groundings is done through their error models, which define their status as a function of the status of their realiser Function Grounding or their required Objectives, respectively. It thus consists of an upstream propagation of errors in the Functional Hierarchy.

The following default error models have been defined in the OM Metamodel, implemented as a set of logic rules:

OMFunctionGrounding: a function grounding is in error if any of the objectives it requires is in FAILURE state.

OMObjective: an objective is in FAILURE if its realising function grounding is in FAILURE or ERROR status. This error model for objectives is actually implicit in TOMASys.

Example

Functional perception of the laser failure

Let us explain how the perception of the state of the functional hierarchy proceeds, taking the localisation subsystem and a permanent failure in the laser sensor.

The failure of the sicklms laser component scales up to the function level by causing its associated OMCompSpec in the Goal to fail. In the Functional Loop this failure implies the internal failure of the fg1 functional grounding (the only one in which the laser component realises a role). Then the failure propagates upwards the functional hierarchy according to the realises associations from OMFunctionGrounding atoms to OMObjective atoms, and the requires associations from OMObjectives to OMFunctionGrounding.

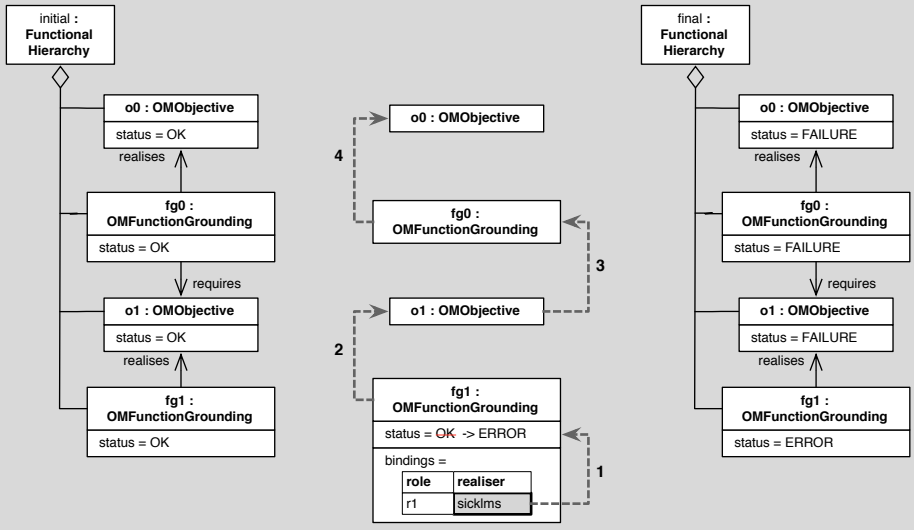


Figure 9.21: The functional hierarchy before the failure of the laser sensor on the left side figure. The propagation of the failure in the functional hierarchy in the center. The final state of the functional hierarchy on the right.

Update Functional Knowledge

The realisability of Function Designs and the availability of Functions are updated in each perception cycle from the availability of Component Classes. This way the knowledge about available design alternatives is thus updated and available for the Functional Loop to decide how to reconfigure the system.

This update is computed using the realisability functions of the Function Designs in the OM Model, and the availability functions of the Functions, and taking as input the status of availability of the Component Classes, as currently updated by the Components Loop.

The OM Metamodel defines the following default `realisability` and `availability` functions for `function designs` and `Functions`, respectively:

Realisability model of a OMFunctionDesign

- if any of the `component classes` in the roles is UNAVAILABLE → UNREALISABLE
- if any of the required `functions` is UNAVAILABLE → UNREALISABLE
- otherwise the `function design` is REALISABLE

Availability Model of a Function

- if any design for this `function` is REALISABLE $\rightarrow$ AVAILABLE
- else it is UNAVAILABLE

9.5.3 Evaluation and Reconfiguration of the Functional Hierarchy

The goal of the Functions Loop is to compute a configuration system that optimally achieves the root objectives. This optimality premise can be formulated as a maximization and a minimization:

- the maximization of a certain *global functionality* of the system, function of the root objectives of the system (RO). This gf function can be formulated as a weighted average of the achievement of the objectives, considering a numerical valuation of the status of a root objective: $status(o) = 1$ if CONVERGENT, and 0 otherwise:

  $gf : RO \rightarrow [0..1]$

  $gf = \frac{\sum_{\forall o \in OR} status(o) \cdot relevance(o)}{\sum_{\forall o \in RO} relevance(o)}$

- the minimization of the resources required by the system configuration. Remember that this configuration is determined by the bindings specified by the Functional Hierarchy.

To compute the system configuration, the Functions Loop has a set of design alternatives, i.e. the Function Designs, and resources, i.e. Component Classes available. These define a subspace of realisable Functional Hierarchies, each one specifying a possible system configuration. The problem of computing a desired system configuration is thus that of moving in the subspace of Functional Hierarchies, with the drivers of the optimality defined above.

At each cycle the Functions Loop departs from a certain extant configuration, and a state of the Functional Hierarchy that is updated accordingly in the perception process described in the previous section. The maximization premise dictates that if a root objective is not CONVERGENT, the part of the hierarchy related to that objective must be modified so as to make it CONVERGENT. Since any reconfiguration, in the general case, involves a cost in terms of system resources and performance, at least transitorily, for the shake of the minimization premise we consider as a reconfiguration policy to change only the part related to that objective. It could be the case that there are other Function Designs better realisable in terms of system resources for other parts of the hierarchy, but as a simplification we consider that that does not compensate the cost of reconfiguration. The reconfiguration policies determine this selection. Let us discuss a bit about reconfiguration policies.

A possible reconfiguration policy can be to try to ground a realisable function for all the objectives that are in error. However, there is no point in grounding a new function for those that are in error because their current realiser Function Grounding are

in external error. Simply providing a new Function Grounding for those Objectives whose current realiser is in failure would suffice. However, even this produces unnecessary grounding of functions, since there is no point in providing new realisers for objectives that are only required for function groundings that are in internal error.

The reconfiguration policies defined in OM are intended to avoid the previous issues and address the optimality premise. To implement its reconfiguration policies, the OM metamodel defines evaluation semantics for the system objectives by defining a relevance function for Objectives, and implements a set of reconfiguration rules. These are discussed following.

Functions Evaluation: objectives' relevance

In the Functional Loop, the aim of the evaluation process is to assess the objectives in the Functional Hierarchy in terms of how relevant they are to the fulfilment of the root objectives.

Example

Suppose the estimated state for the hierarchy of functions and objectives showed in figure 9.22. Two function groundings, $fg0$ and $fg1$, are in internal error, and the perception process at the functional level has updated the state of the functional hierarchy accordingly, so that the root objective ro and the intermediate objective $o1$ are in error. A new realiser for fg0 is needed. However, a reconfiguration including also a new realiser for $o1$ would be pointless, since the new realiser for ro may not require o1. We say that $o1$ is no longer relevant for ro, since the Function Grounding that relates them, which is $fg0$, no longer holds.

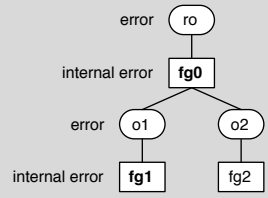


Figure 9.22: Hypothetic functional hierarchy in error. Objectives are represented as blobs and function groundings as boxes.

The ECL evaluation process in the Functions Loop consists of the downstream propagation of the relevance of objectives, from the topmost root objectives —whose relevance is fixed— downwards to the bottom objectives in the hierarchy, whose Functions Grounding do not have required Objectives.

The propagation of relevances is specified in the semantics for the TOMASys relevance function for Objectives, which have been included in the OM Metamodel:

- The relevance of a function grounding is that of the objective it is realising that has the largest relevance.
- All the objectives required by a function grounding that is in internal error have $relevance = 0$.

The result of the functional evaluation process is the set of Objectives in the Functional Hierarchy with $relevance > 0$. These are the objectives that require functions grounded to address them, because either there are no already grounded functions realising them, or these functions are in error.

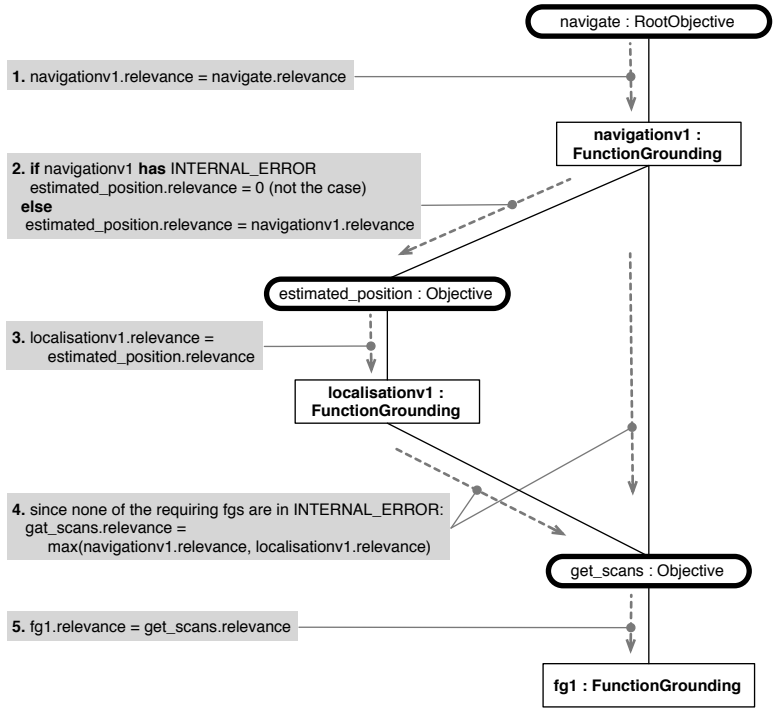


Figure 9.23: Example of the evaluation process in the Functional ECL in the previous case of a failure of the laser component.

Functions Control: reconfiguration

The Control activity at the Functions Loop produces at each cycle, if required, an action command consisting of a reconfiguration of components in the domain control system. This reconfiguration is a set of specifications of components that would ground the functions required to better fulfil the root objectives, as discussed formerly in the evaluation process.

To determine the reconfiguration, the following inputs are considered:

- the evaluation of the hierarchy of functions and objectives, that dictates the set of objectives that require new function groundings to realise them,
- the function designs' realisabilities, that establish what design solutions are implementable given the current state of the system.

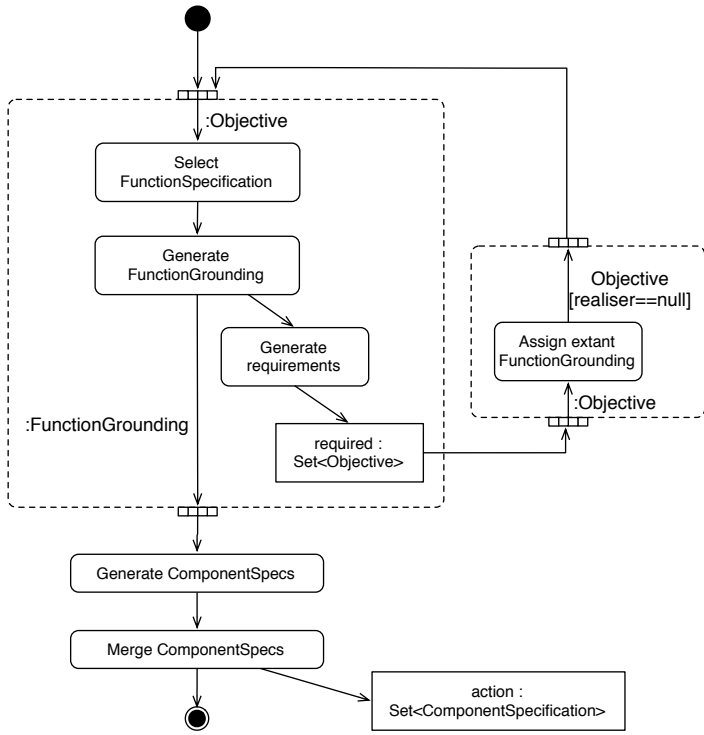


Figure 9.24: UML Activity diagram of the Control process in the Functional Loop. The computation of the function groundings is a recursive process that is executed over the set of objectives (symbolised by the array of small squares).

The computation of the reconfiguration action is performed following the reconfiguration rules defined in the OM Metamodel, as part of its implementation of TOMASys ADM. Different rules could be envisioned. The ones we have designed define the following procedure to compute the reconfiguration action (shown in figure 9.24):

1. First, for every objective in ERROR status and relevance greater than zero, an OMFunctionGrounding that achieves it is produced. This is done by:
   (a) selecting the Function Design that solves the function type of the objective (it must be in REALISABLE status) with the highest confidence, and then
   (b) generate a Function Grounding with the parameters resolved to the values given by the objective.

   If there was a previous Function Grounding realising the objective in error, it

is deleted from the Functional Hierarchy. The bindings of the new Function Grounding are created later, at this stage the objectives required by the new function groundings are instantiated. This process of instantiation of Functions Groundings for those objectives with no valid realiser goes on recursively for the newly created objectives, but for those cases in which there already exists a Function Grounding that realises the new objective.

2. The new function groundings and required objectives are added to the functional hierarchy.
3. Now, for each Role in the bindings of the newly created FGs a default OMCompSpec is created from its definition. This results in a new set of OMComponentSpecifications.
4. This new set is added to the extant set of OMCompSpec atoms. The specifications corresponding to Function Groundings that have been deleted are also eliminated from this set.
5. The complete set of OMComponentSpecifications is simplified, following the minimization premise, reducing it to the minimal subset of those specifications that complies with the complete specifications defined by the initial set.
6. Finally, the resulting specification of components, which represents the control action produced by the Functional Loop, is sent to the Components Loop to become its new Components Goal, as was described on page 180.

9.6 Operation summary of the OM Metacontroller

So far we have detailed each process and activity defined by the OM Architecture to metacontrol a control system. Let us put all the pieces back together to show how the operation of an OM-based metacontroller addresses the two basic scenarios presented at the beginning of the chapter (section 9.1.2):

- S1 – Recoverable component failure.
- S2 – Non-recoverable component failure.

The scenarios are applied to the metacontrol of the localisation subsystem of our mobile robot. Let us consider the design analysed in the example of page 154, where there were two different designs for the localisation subsystem, differing on the range sensor used. We shall start from an initial configuration of the localisation subsystem corresponding to the laser design, because is provides a greater `confidence` to achieve the functionality of self-localisation.

Once defined the domain control system, the metacontrol system is defined by the OM Architecture and the concrete OM model of the former that it exploits. Let us discuss the contents of the model. The concept atoms that represent the long-term knowledge in the model are depicted in Figure 9.25.

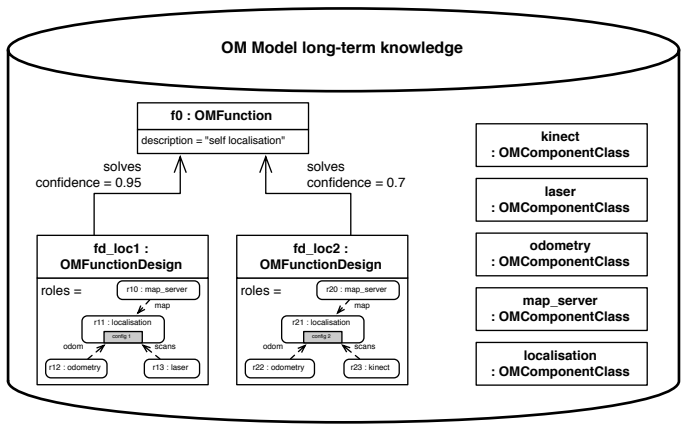


Figure 9.25: The long-term knowledge in the metacontroller contains two design alternatives and the properties of the components that can be part of the control system.

The ultimate goal of the metacontroller is the goal reference for the Functions Loop, and consists of solving the `root objective` o0 "self-localisation".

To achieve its goal, the Functions Loop commands a functional hierarchy that realises the function design fd_loc1 that uses the laser as the scan sensor, because it is the one with the highest confidence. The grounding of this function requires the fulfilment of the goal g1, consisting of a set of specifications of components that fulfil all the roles in fd_loc1.

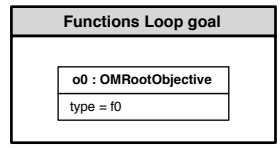


Figure 9.26: The goal of the Functions Loop is the achievement of the `root objective` o0, instance of the `function` f0 "self-localise".

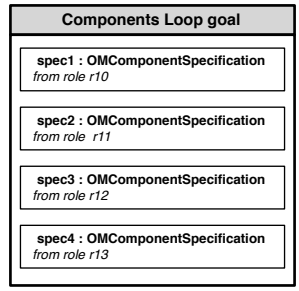


Figure 9.27: The specifications that form the goal for the Components Loop correspond to the definitions of the roles of the function design fd_loc1.

Considering the departing situation of the localisation subsystem performing well its function using the laser configuration, the estimated state perceived by the Functions and the Components loops consists of the `functional hierarchy` and the components `configuration` depicted in figure 9.28:

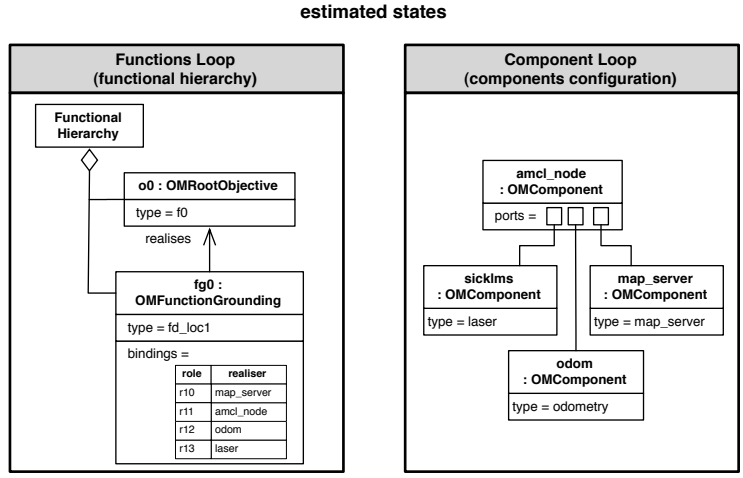


Figure 9.28: Estimated state of the localisation subsystem using the laser configuration, as modeled by each loop of the the metacontrol.

In the following, the basic actions and the resulting information signals occurring in the OM metacontrol system are schematized for each scenario, using an UML Activity diagram. These behavioural diagrams depict the sequences of actions that are representative of each scenario. We can consider the diagrams presented as white-boxes, since they show in different swimlanes which module of the whole system is

responsible for the actions: the running control system, which can be considered an UML actor from the standpoint of the metacontroller, and the Meta I/O module, the Components Loop and the Functions Loop, which are subsystems of the metacontrol system.

9.6.1 S1: Recoverable component failure

In the first scenario the localisation subsystem is working with the configuration that uses the laser sensor. The metacontroller is monitoring the system, and no actions neither at the Component Loop nor the Functions Loop are issued, the estimated states at both loops fulfiling the respective goals o1 and g1. The contents of the run-time model are those described in figures 9.25, 9.26 and 9.27.

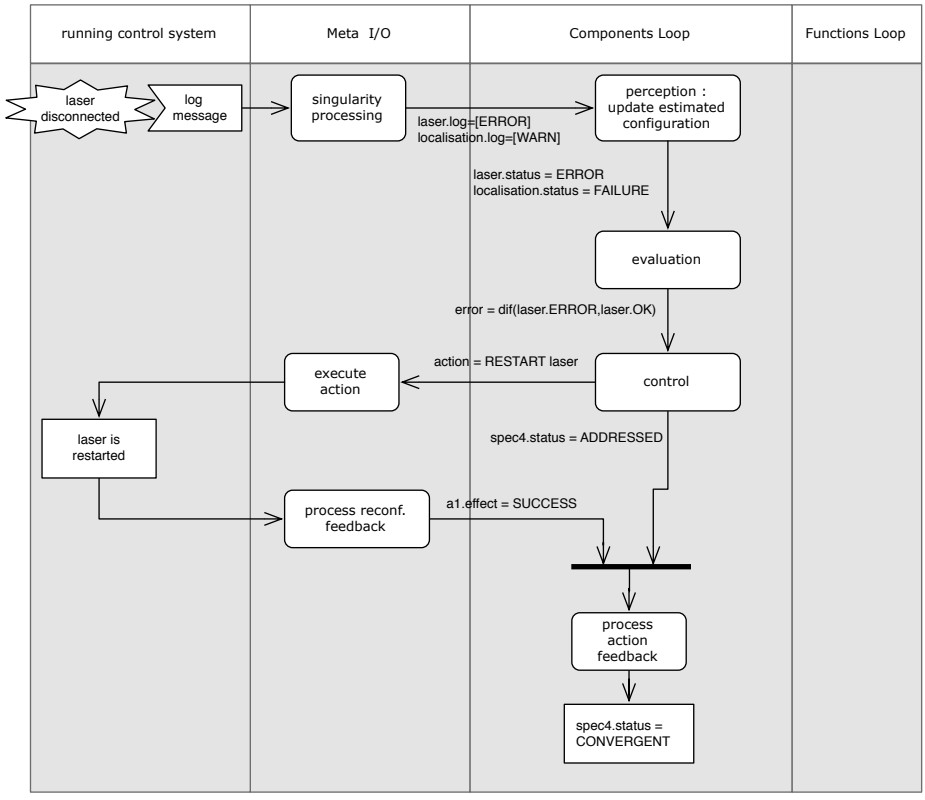


Figure 9.29: Recoverable failure of the laser sensor. The diagram schematises the operational flow at the different layers of the OM metacontroller. Arrows represent events with their associated information signal, and rounded boxes the processes and operations.

9.6.2 S2: Non-recoverable component failure

In this second scenario, the laser suffers a permanent failure, for example because it is disconnected from the robot's on-board control computer, due to whatever reason. The scenario departs from the same state of the localisation system and the metacontroller as scenario 1. The same process takes place upon laser failure: the situation is perceived at the Components loop, where reconfiguration action to restart the laser sensor is determined and commanded to the Meta I/O module.

However, the flow of events diverges here from that of the first scenario. Because of the permanent nature of the laser failure, the reconfiguration action issued to restart the sensor fails. This is detected by the Meta I/O module, which compiles the information of the infrastructure about it and sends it to the Components loop in an effect signal.

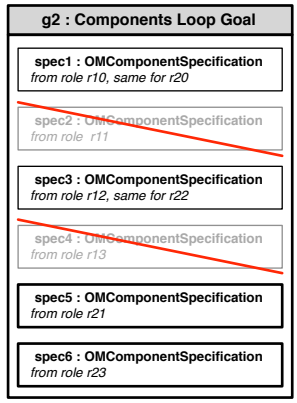


Figure 9.30: The new goal g2 for the Components Loop represents the reconfiguration as a new set of components specifications.

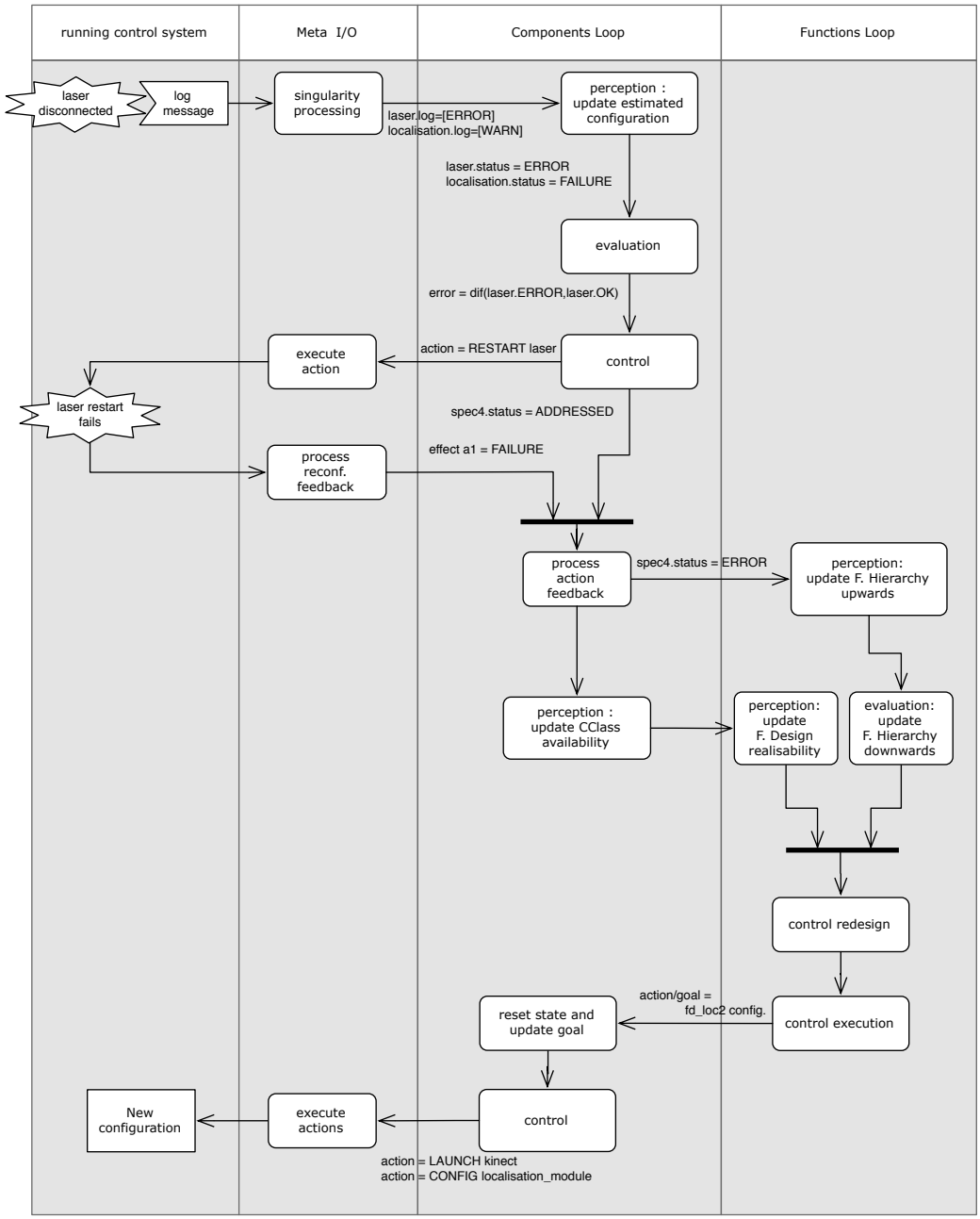


Figure 9.31: Non-recoverable failure of the laser sensor: the failure scales up to the Functional Loop, where a reconfiguration of the Functional Hierarchy is generated, which is then grounded and executed.

9.7 OM Architecture overall assessment

The OM Architecture provides a blueprint for building metacontrol systems following the patterned approach proposed in this thesis for self-aware autonomous systems. Its reification of the Epistemic Control Loop pattern makes OM a reference architecture close to RCS in its underlying conceptualisation of the cognitive phenomena.

When compared with cognitive architectures, the OM Architecture aim is not to be an architecture for general intelligence, as RCS or Soar are, but rather to apply a cognitive structural pattern (ECL) to provide for a specific cognitive trait: metacognition; and to do it in a control engineering framework.

The OM Architecture could be regarded as a blueprint for the Supervisor level in the control architecture for fault-tolerance proposed by Blanke et al. discussed in section 5.1. OM cognitive approach, with an estimation of the state based on perceived feedback rather than open-loop proprioceptive information, allows the metacontrol to deal with higher levels of uncertainty within the system. Incorrect configurations of components, caused by either human supervisors or other automatic supervisory systems, can be detected and corrected by the OM metacontroller. This is specially interesting for systems of systems, were federation leads to uncertain interactions between systems:

> Orchestrating a... system requires supporting interdependencies and controlling the consequences of local actions with respect to their effect on the emergent whole, even though each part of a system might be actingto maximize its utility. [101]

The architectural framework presented, encompassing the OM Architecture and the TOMASys model, presents some similarities with autonomic computing approaches such as those discussed in section 5.2. However, OMACS solution focuses on the metamodel, the architecture that exploits it being more application contingent. OMACS provides minimal blueprint for an agent architecture, which is a particular design possible for the control of autonomous systems. On the other hand, OM provides a complete architecture for the metacontroller, and it is possible to adapt it to any component-based system, not only multi-agent systems.

9.7.1 Self-awareness and the OM Architecture

Let us now analyse the self-awareness properties of the OM Architecture, under the prism of the valuable functions related to conscious phenomena discussed in section 3.1.3.

We can argue that the OM Architecture involves meta-representation. These meta-representations are the OM Model that lies at the core of the OM Metacontroller operation, which can be regarded as a representation of the Self in the pursuit of its objectives. The OM Model is a (functional) representation of the processes that conform the (domain) control system, that is the "mind" of the autonomous system, taking

a cognitive perspective of control.

This model integrates all the monitoring information in a unified vision of the organisation (self-image) and directiveness (volition) of the control system (the unified Self [15, pp. 142–153]). The metacontroller performs reasoning and evaluation activities over this model to assess the performance of the control system and take appropriate reconfiguration actions if required.

The reconfiguration of the components of the domain control system performed by the OM Metacontrol can be regarded as a resource allocation function of cognitive resources. Attentional mechanisms associated to self-awareness [16, p. 50] are attributed a similar role [68], although in relation to access to conscious contents. Notwithstanding the former, the OM Architecture only configures these resources to improve the system's ***adaptivity***, and not to maximise its ***performance***, which is the fine grained allocation that attentional mechanisms related to self-awareness are claimed to contribute to.

To organise the operation of the metacontroller, the OM Architecture proposed makes also use of sequentiality, in the form of an executive cycle that guarantees the consistency of operation in the two loops that form the metacontroller.

Part IV

Implementation and Validation

Chapter 10

OM Engineering

This chapter describes the engineering methodology and related assets we have developed for the application of the OM Architecture. The first section details the engineering methodology envisioned to apply the OM architectural framework to the development of a self-aware autonomous system. Then the second section discusses the application of the MDA progressive software refinement in the OM architectural framework, and the OMJava library developed that provides a platform specific model of the OM Architecture in Java. The next section explores the possibility of an automated transformation to obtain the TOMASys model of the autonomous system to completely realise the Deep Model Reflection Pattern in the engineering of an OM metacontroller. Finally, the last section discussed how the elements of the OM architectural framework fit in the ASys vision for the construction of autonomous systems.

10.1 OM Engineering Process

We refer to the complete solution for self-awareness through metacontrol presented in part III as the *OM Architecture Framework*, which includes all the elements already discussed, from the design principles, to the final OM Architecture and its required TOMASys metamodel, including the Design Patterns for Self-awareness in between. For the OM Architectural Framework to be of engineering applicability a process or methodology to apply its design solution is required. A methodology is a set of related processes and techniques: the process defines *what* is to be done, the techniques specify *how* it is to be done [46]; tools can be provided or suggested to support the methodology.

The OM Engineering Process (OMEP) described in this chapter defines a methodology that can be followed to produce a control system (including both the domain and the metacontrol subsystems) using the OM Architecture Framework.

OMEP distinguishes two sub-processes: one that addresses the "standard" devel-

opment of the (domain) control system for the autonomous application, and a "meta" one to develop the metacontrol subsystem:

1. OMEP Control Development: it is the development of the domain control system. OMEP defines constraints on two of the phases:
   - Design – the core workproducts are the models of the plant and the control system, including a functional model.
   - Implementation – it takes as input the models, which following a MDE guided process are converted into the software that implements the control system.
2. OMEP Meta Development: it is the development of the metacontrol system. It takes as input the functional model of the control system, and uses the OM architectural framework to build from them and the OM Architecture the metracontroller for the autonomous system. This process defines also the design and implementation stages:
   - Design – the design process for the metacontroller has the particularity of including an initial step to convert the functional model of the control system into a TOMASys model.
   - Implementation – some tools have been developed in the form of software packages to support the metacontroller implementation. They will be presented later in sections 10.2.1 and 11.4.

These two sub-processes of OMEP are schematized in figure 10.1, and will be detailed in the following sections.

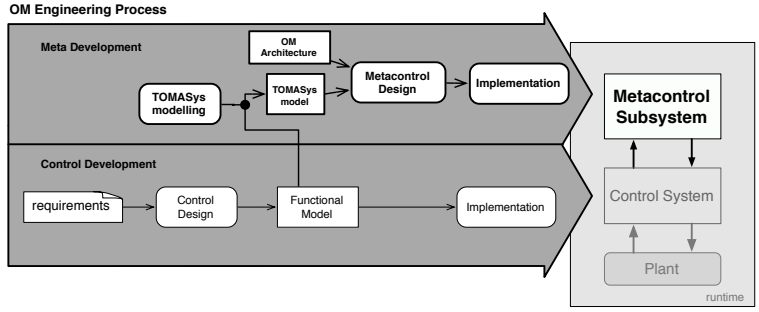


Figure 10.1: The OM Engineering Process: the Meta Development process is incorporated to the control engineering to build the Metacontroller for the autonomous system.

10.1.1 OMEP Control Development

OMEP does not provides a complete specification for the engineering process of the domain control subsystem, but rather defines what it must fulfil for the OM Architec-

tural approach to be of applicability. This way, the OMEP methodology imposes two requirements to the development process of the control system:

- A component-based technology shall be employed to build the control system. This is a requirement inherited from the OM architectural approach.
- A functional model of the control system must be produced in the process. This requirement results from the thesis postulates (page 111).

Metacontrol requirement: There is a pervasive pre-requisite, concerning the design and implementation of the control system, which is the existence of alternative designs and implementation artifacts, i.e. analytical and/or physical redundancy. The control system implementation must have the possibility of having organisational alternatives to support the advantageous exploitation of the multiple configurations by the metacontrol. Redundancies may be necessary. If there is only one possible controller there is no controller design space to be explored by the metacontroller.

OMEP follows ASys model-based engineering principles, and thus enforces a strict MDE-based approach for the development of the control system. This guarantees the production of the required functional model of the system. The OMEP subprocess for the development of the control system consists of two phases:

- Control Design, which involves requirement specification, domain analysis and functional design. It has a final product: the functional model of the controller.
- Control Implementation, when the component platform is used to finally produce the (domain) control system.

For further guidelines on how to realise this process, OMEP recommends following the OASys-based Engineering Methodology [20]. However, other MDE-based methodologies would also be possible, for example the ISE&PPOOA methodology proposed by Fernández-Sánchez *et al.* [47], which include a model of the functional architecture of the system, or consider also the methodology proposed in [141]. These methods guarantee the generation of models that conform to formal modeling languages, and thus the applicability of the Deep Model Reflection Pattern.

The **OASys-Based Engineering Methodology** proposed by Bermejo-Alonso [18], aims to provide guidelines to the engineering process of an autonomous system in the context of ASys. The scope of this methodology is limited to the Analysis phase in the generic engineering process discussed in page 7. The methodology describes how to carry out the generic ASys process in terms of phases, tasks and work products . It uses as guideline the ontological elements in the System Engineering and ASys Engineering subontologies of OASys [19]. The two main phases are ASys Requirement to identify the requirements of the autonomous application, and ASys Analysis (see table 10.1). This latter is the most relevant for OMEP, since its purpose is to describe the autonomous system from the structural, functional and behavioural perspectives. Analysis is divided into:

Structural Analysis: consists of different modelling tasks to analyse the system de-

composition into subsystems, elements, quantities, etc [1]. That is, it characterises the `Components` of the system.

Functional Analysis: identifies the Actions and Operations, according to OASys. That is, it identifies the `Roles` and the `Function Designs` in the system.

Behavioural Analysis identifies the dynamical behaviours of the components. It has not been included in OMEP yet, since TOMASys does not cover dynamical concerns.

Phase	**Tasks**	**Work Products**
ASys Requirement	UseCase modelling and Requirement Characterisation	UseCase models and Requirement specifications
ASys Analysis	Structural Analysis	Structure and Topology models
	Behavioural Analysis	Behaviour Model
	Functional Analysis	Functional Model

Table 10.1: The main phases of the OASys-Based Methodology, adapted from [18].

10.1.2 OMEP Meta Development

The building of the metacontrol subsystem is the core of this work; the OM Architectural framework has been developed to provide a complete set of assets to support it. The different phases defined in OMEP for the building of the metacontrol subsystem are:

1. Build a TOMASys model of the domain control system. The Deep Model Reflection Pattern can be applied so that the TOMASys model would be automatically generated from the functional model of the system produced in the design phase of the domain control system, using a model-to-model transformation.

2. Build the metacontroller that exploits the previous model following the OM reference architecture. The OM Architecture defines a universal functional design for the metacontrol system, so there is only the detailed design (for generating a platform specific model) remaining for the developer of the concrete autonomous system application.

[1] note that OASys is rooted in Lopez's [90] autonomous systems' conceptualisation, as also is the theoretical framework of this work.

10.2 MDA in the OMEP methodology

The OM Architecture provides a solution to the design phase of the metacontroller in OMEP. To realise the implementation phase, the OM Engineering Process relies on the MDA process of progressive refinement of models, according to the MDA *model weaving* pattern [11] discussed on page 48.

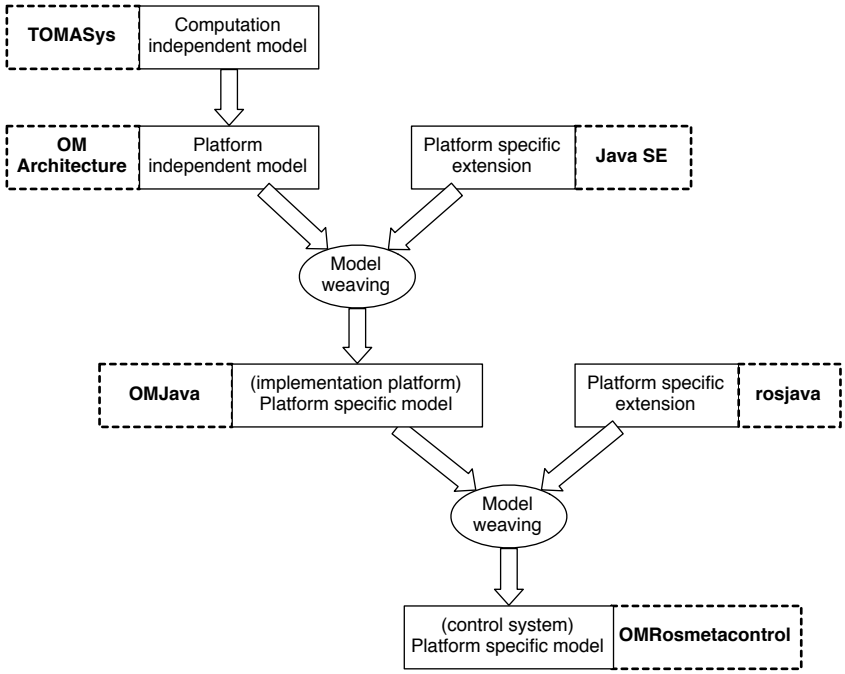


Figure 10.2: The MDA pattern in OMEP: application of model weaving to the assets of the OM Architectural Framework.

We can consider the OM assets from an MDA viewpoint (see Figure 10.2). From this perspective (section 3.2.1), the TOMASys metamodel can be regarded as a computation-independent model. It contains a domain model, describing the concepts of functions and components and their interrelations. The OM Architecture could then be considered as a platform-independent model for the metacontrol system.

Departing from the previous considerations, we have defined for the implementation phase of the metacontroller a process of progressive platform-specific refinement as shown in figure 10.2. Several libraries and components have been developed for the platform-specific models of the OM metacontrol. The Java library (OMJava, discussed below), specifies the (metacontrol) system in the dimension of the implementing platform technology. A ROS stack (omros, described in the following chapter), specifies the OM metacontrol for a specific type of control systems, according to their

component-platform, which is the ROS platform[2].

10.2.1 OMJava library

We have developed a Java library that implements the OM Architecture Framework: the TOMASys metamodel and the OM Architecture. This library constitutes a Platform Specific Model of both TOMASys and OM Architecture in Java.

We have selected Java because it suits our aim of generality. Java is multiplatform: its vitual machine execution paradigm guarantees that computer programs written in the Java language must run similarly on any hardware/operating-system platform. Besides, it is most suitable for an MDA approach such as ASys' and this work; given that Java is an object-oriented programming language, and many MDA frameworks provide tools for it.

Following we present a brief rationale about the main packages in the OMJava library.

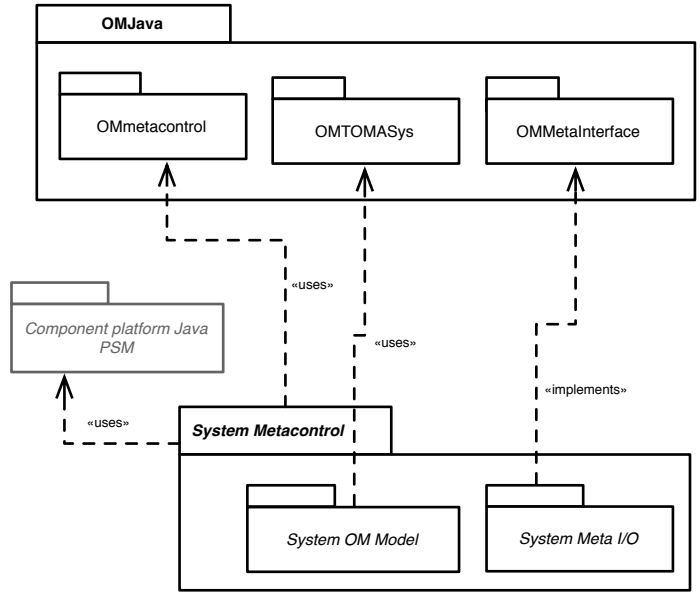


Figure 10.3: The relation between the elements in the implementation in Java of the metacontroller of a system and OMJava and its sub-packages.

[2]ROS (Robot Operating System) [52] is a component platform for robotics

OMTOMASys package

The OMTOMASys package contains a Java PSM of the TOMASys metamodel. TOMASys has been specified in UML; this model constitutes the platform independent model (PIM) for TOMASys. From that PIM, the TOMASys metamodel has been implemented in one of the packages of OMJava (which is therefore the Java PSM for TOMASys), in two steps. Firstly, a set of interfaces has been developed from the PIM to define the elements of TOMASys and the operations that can be performed over them. However this does not provide a complete specification of the metamodel, since no semantics are captured. Then, the Java PSM is completed with a set of Java classes that implement the interfaces. The package containing all these classes is actually a complete implementation of the OM Metamodel in Java (PSM). It provides default semantics for the elements in the Application Domain Model (e.g. error models). The classes also implement the ECL semantics for perception, evaluation and control operations.

Using inheritance from these Java classes, more refined PSMs can be obtained for the specific component platform employed in the construction of the autonomous application. This is demonstrated in section 11.4 of the next chapter for the Robot Operating System (ROS) platform.

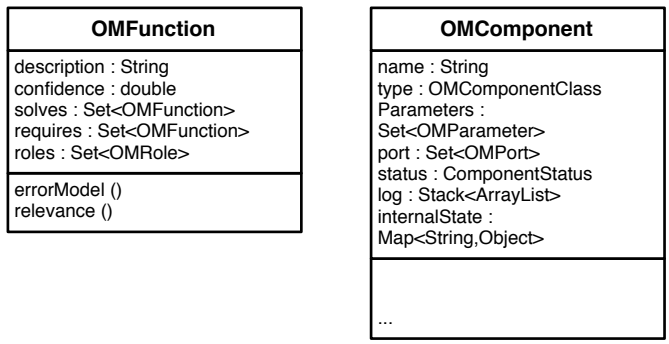


Figure 10.4: Examples of the classes in the OMJava library.

OMmetacontrol package

The OM Architecture has been implemented in a set of Java classes and interfaces. They implement a metacontroller, and define its connection with the Meta I/O module through a Java interface (OMMetaInterface). A generic Java implementation of the Meta I/O module is not provided with the library, since it is platform specific. However, an example has been developed for the testbed mobile robot application in the ROS platform (section 11.4.2).

MetaInterface. The MetaInterface has been implemented as a Java interface and a set of classes for the datatypes to represent in the object-oriented paradigm the

signals involved in the interface between the metacontroller and the platform of the running control system.

OMmetacontroller. The OMmetacontroller class implements the OM Metacontroller, containing instances for the two loops , an initialisation method that instantiates both with a given model, and implementing methods to control the operation of the metacontroller: start(), pause(), terminate().

The ComponentsLoop and the FunctionsLoop classes implement the two nested control loops. Both loops follow the periodic and sequential execution cycle of $perceive \rightarrow evaluate \rightarrow control$ that are specified in the OM Architecture. Each one is implemented in its own dedicated thread.

10.2.2 OMJava in OM Engineering Process

The OMJava library supports the implementation phase of the metacontroller in OMEP (see figure 10.5). Firstly, it provides a Java PSM for an application-independent metacontroller. Secondly, the OMJava MetaInterface can be used to develop the Meta I/O module specific for the component-platform of the control system. Finally, the OMTOMASys package supports the creation of the executable model for the concrete application.

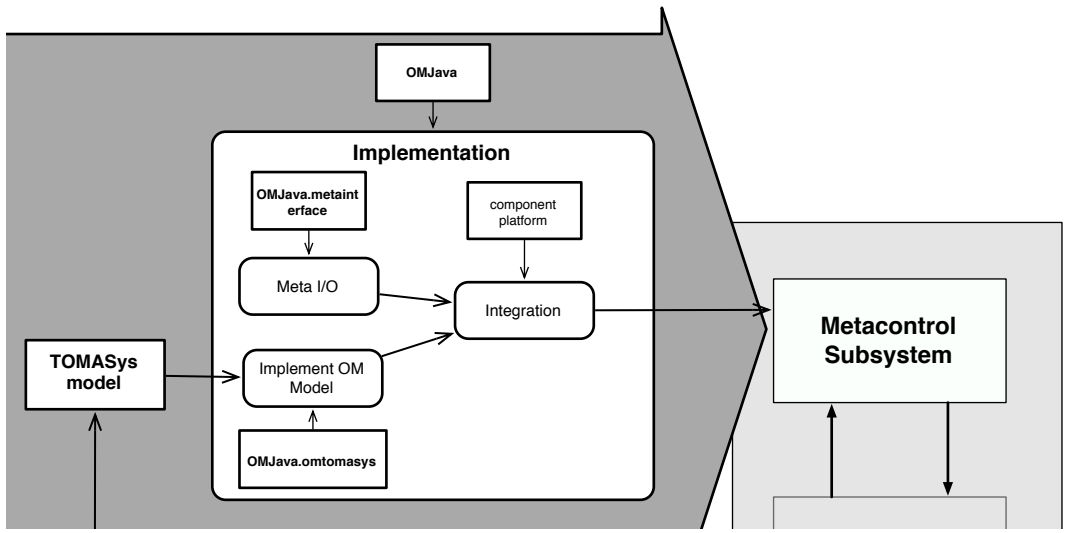


Figure 10.5: OMJava and the OMEP methodology. The OMEP Meta process is simplified with the use of the OMJava library, bacause it allows to directly implement the metacontrol subsystem from the TOMASys model.

10.3 OM model transformation

To fully implement the Deep Model Reflection pattern into the OMEP methodology, the specification of a model-to-model transformation from the functional model of the control system into the TOMASys model of it is required. Such a possibility requires the formalisation of TOMASys and the selection of a particular modeling language and additional semantics for the functional modeling. This task has been left to future works continuing the development of the OM Architecture approach. However, in the following we briefly discuss a possible path to follow.

Following the MDA approach, a MOF-based language could be used for the functional model. Reasonable alternatives are UML or SysML. This later seems more appropriate given its orientation to systems engineering, whereas UML is very focused on software engineering. However, UML is a well established standard in the industry, with plenty of toolsets and frameworks developed around it, whereas SysML is still in an exploratory stage regarding its application in industry. Let us assume we selected UML as the referent metamodel to develop the transformation, most considerations are translatable to SysML.

TOMASys could then be formalised using MOF as its meta-meta-model. This would guarantee a feasible transformation, TOMASys and UML sharing their meta-meta-model.

The model-to-model transformation could be implemented by defining a UML profile for TOMASys. Stereotypes will be defined for TOMASys elements, so that the application of such a stereotype to an element in the UML functional model, that element would be converted in the transformation in a TOMASys element with the type and properties determined by the stereotype.

10.4 OM Architectural Framework in the ASys vision

The elements of the OM Architecture Framework developed in this work can be easily integrated in the ASys vision for the engineering of autonomous systems (figure 10.6).

The four design patterns for model-based self-aware autonomous systems are part of the ASys generative pattern [65]. The OM Architechture is an asset that actually constitutes the *metamodel* for the design model of the metacontrol subsystem, in the sense that the design of the control architecture of a particular system is an instance of the reference architecture (if it exists). TOMASys is an asset base in the category of the ontologies, such as OASys, that allows to produce one of the models for the autonomous system: the self-model.

OMJava library is a package of components in the ASys base that allows the synthesis or implementation of the Autonomous system. The library for implementing an OM metacontroller in a ROS-based control system, which will be presented in the next chapter, is also a package of components.

The OMEP methodology shall be integrated with other ASys processes required for the design and synthesis of ASys systems.

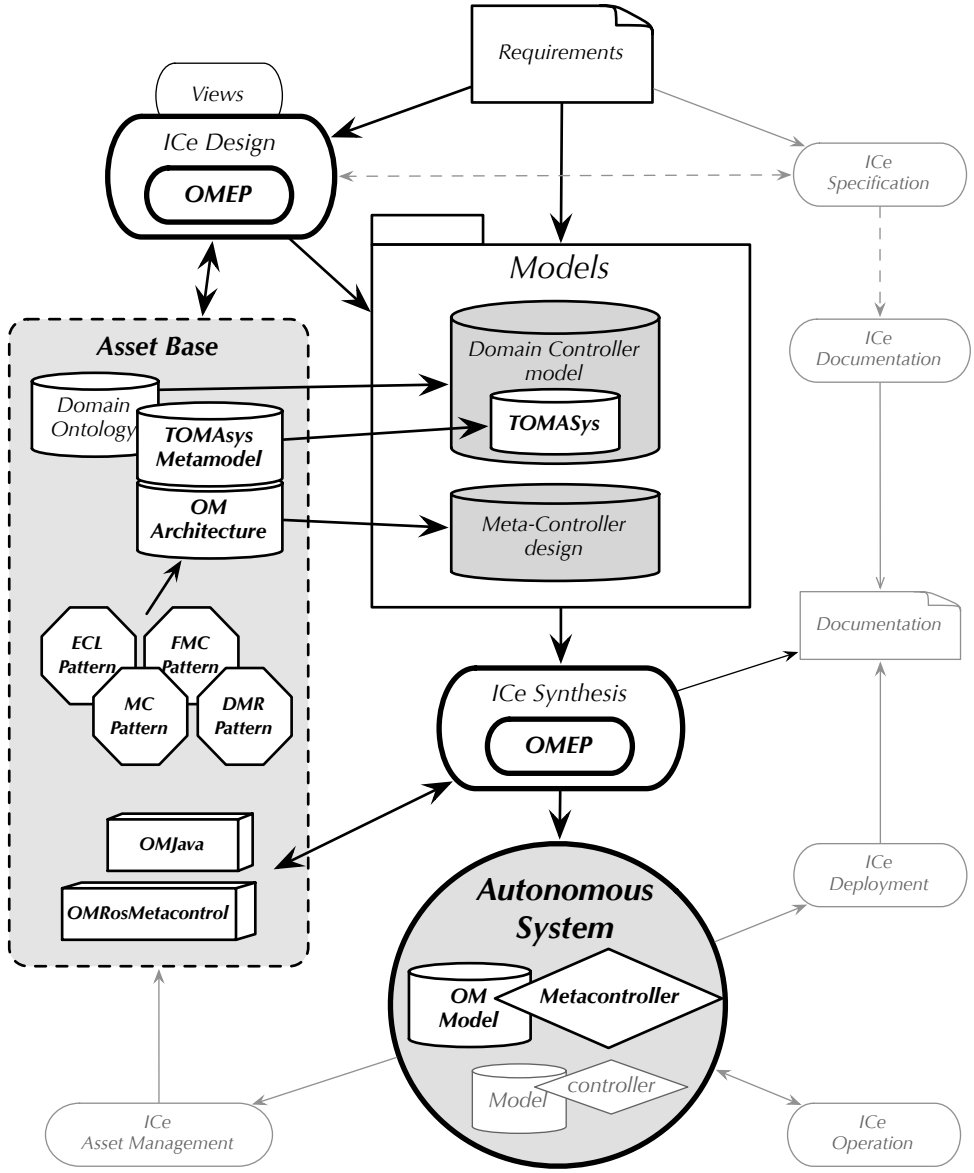


Figure 10.6: The framework and assets developed in this work in the overall picture of the ASys engineering of autonomous systems.

Chapter 11

Testbed System

For the demonstration of the theoretical and engineering frameworks presented in this work we have developed a state-of-the-art mobile robotic application, with improved adaptivity based on self-awareness. For that, a metacontroller has been developed with the OM Architectural Framework discussed in part III of the dissertation. It has been implemented and integrated in an autonomous robot following the OM Engineering Process and using the OMJava library presented in the previous chapter.

11.1 The Autonomous Mobile Robot

To validate the OM architectural framework a testbed autonomous application has been developed, as advanced in the presentation of the approach of this work (page 35). It is based on an autonomous mobile robot for navigation in office-like environments. The mission of the robot is to navigate through an office-like environment passing by a user-defined set of locations or waypoints, without bumping into obstacles.

This application was selected for a series of reasons. Firstly, it is a challenging autonomous application: the robot operates in an open-ended uncertain environment. Although it is assumed that the environment is static, and an initial map of the area to navigate is provided, deviations of the actual environment from the provided map are possible, for example due to changes in the location of furniture and appliances (chairs, packages, etc). In addition, the system involves heterogeneous components, both hardware and software, and the required control system has the sufficient level of complexity to be representative of the challenges addressed. The robot autonomous operation faces different uncertainties: sensors are subject to failure, and algorithms for navigation can also fail due to scarce and/or noisy input information.

The developed system consists of a mobile platform equipped with different sensors and a control system to perform a navigation task, with a remote station providing for off-board computation capacity and the operator interface. The control

system includes: i) a state of the art navigation architecture for mobile robots (domain controller), and ii) the metacontrol subsystem we have developed with the OM Architecture. The metacontroller purpose is to sense the state of the navigation system and reconfigure it if required in order to preserve its functionality (see figure 11.1).

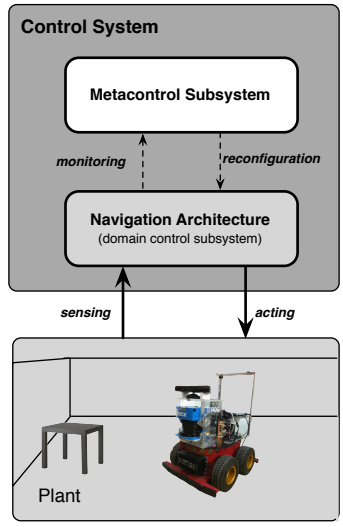


Figure 11.1: Overview of the autonomous mobile robot testbed: the Navigation System is controlled by the Metacontrol System, which provides for self-awareness and reconfiguration, providing the complete control system of the autonomous robot with enhanced adaptivity.

11.1.1 Mission and requirements

The mission of the mobile robot is to patrol a certain area. For that a human operator provides through the OI an occupancy map of the area and the waypoints of the route to patrol. The robot navigates autonomously following the route, without bumping into obstacles. The operator can pause and resume the operation at any moment, as well as manually teleoperate the robot, or command it to go to a destination in the map different from the waypoints in the provided route. This application can be specified in the following set of requirements or capabilities using the methodology presented by Fernández et al. [47]:

1. Teleoperation: at any moment a human operator must be capable of manually teleoperating the robot by direct command of its speed and direction.
2. Commanded navigation: the human operator can command the robot to go to a certain location in the area. The robot shall be able to navigate to it without bumping into obstacles, if possible, or report when it is not feasible.
3. Autonomous patrolling: the robot travels periodically a route with a given set of

way points, navigating autonomously.

4. Operation mode switch: at any moment the operator must be able to switch between the different modes of operation by: i) pausing and resuming the patrolling mode, ii) pausing navigation when a target destination is commanded by the operator, iii) at any moment teleoperation overrides the other operation modes.

5. Localisation: at any moment a reasonable estimation of the robot position must be available for display in a map of the patrolled area, so a human supervisor can make informed decisions concerning robot operation.

Robust-autonomy and adaptivity requirements

Continuing with Fernández's methodology for capturing the system's requirements, we shall analyse now the non-functional capabilities, related to quality of service and resilience. In this work we are addressing robustness and adaptivity capabilities and so the robot testbed focuses on them, in relation to fault-tolerance aspects:

- In the event of a transient failure of one of its components, the system shall put itself in a safe state, i.e. stop moving but by teleoperation, and recover autonomously from the failure resuming its operation.
- In the event of a permanent failure in a component, the system shall safely reconfigure itself to maintain its navigation capabilities, from higher to lower priority: 1) teleoperation, 2) commanded navigation, 3) autonomous patrolling.

These requirements are the objective of the metacontrol subsystem, and they will be further refined in section 11.3.1.

11.1.2 The mobile robotic platform

Let us first provide a more detailed description of the *plant* in our autonomous application. It consists of a mobile robotic platform equipped with sensors —a Kinect, a laser scan, a compass and wheel encoders— and actuators —differential wheel drive— connected through appropriate I/O mechanisms to an on-board computer; and a remote computer for the human operator.

Here is a summary description of all the hardware elements of the system[1]:

- Pioneer Robot: the mobile robot is a Pioneer 2AT8, which is a standard platform for mobile robotics research. It provides the following capacities:
  - Differential wheel drive (actuator) for the robot's motion, with embedded PID control loop, accepting as command the setpoints for the linear and angular velocities.

[1] Only the hardware elements relevant for this thesis are described.

  - Odometry sensors (wheel encoders), that provide information of both the robot's pose (ded-reckoning) and velocity (linear and angular).

- Laser: it is a SICK LMS200 range sensor providing readings in a range of 180°, one per degree. Therefore this sensor provides accurate detection of obstacles.
- Kinect: the Kinect camera that provides a matrix of RGB (image) and depth information in an horizontal and vertical range of 60°. In this application only the depth information is used to detect obstacles. The information is much more imprecise and noisy than the laser's.
- Compass: provides orientation with respect to the magnetic north. Connected to the on-board computer through an Arduino[2] based I/O board. The reading of the compass is used to obtain a better ded-reckoning localisation estimation, by integrating this information with the odometry in an extended Kalman filter.

The system also involves a series of computing resources and communication infrastructure. A computer on-board the robot provides access to the mobile platform services and the sensors and actuator devices. The drivers for all the hardware are thus deployed in this computer. The on-board computer is connected by WiFi to a remote computer. The remote computer is used for the operator interface with the system and to execute those components in the navigation architecture that are too resource demanding to run in the on-board computer. An exhaustive description of the complete mobile robotic platform can be found in [66].

11.2 Control Development

To develop the domain control system of our autonomous application, which we shall refer to as the control or navigation system for simplicity, we have selected the ROS [118] platform because it fulfils the requirements for the OM Architecture, described in section 9.1.2. Most important of all, it is a component-based platform that can be modelled with TOMASys. Its computation model consists of nodes that publish and subscribe to message channels or topics. This type of loosely coupled components fulfils the requirement of the OM Architecture. For the connection between the metacontrol system and the control system, the interface defined by the `MetaControl` pattern, the ROS infrastructure provides an API that possibilitates to implement mechanisms for *monitoring* and *reconfiguration*. Additionally, there are open-source ROS implementations of components for navigation of mobile robots available, which has facilitated the task of developing the testbed.

A ROS-based control system consists of a set of ROS nodes running in parallel and interacting through message publication and subscription. They are active components executing with a certain frequency to publish the data they produce. Their reception of data, in the form of messages, is asynchronous. They all connect through the ROS middleware infrastructure. This allows them to be seamless distributed amongst the on-board and remote computers.

[2] open-source I/O board with microcontroller `http://www.arduino.cc`

Instead of developing the navigation system from scratch following a MDE approach, an extant state-of-the art implementation has been adapted to build it. Concretely we have used the ROS navigation stack that implements the navigation system developed by Marder-Eppstein et al. [93] at Willow Garage[3]. This way we have also validated the applicability of the OM Architecture to enhance extant autonomous systems by adding self-awareness and adaptivity capabilities.

The ROS navigation stack provides components to build a system that navigates to a commanded location given an initial occupancy map of the environment and an initial position of the robot. For that, odometry information is used, as well as information about obstacles from different types of sensors, including laser range scanners and Kinect devices.

11.2.1 Overview of the Navigation System architecture

To develop our patrolling robot we have adapted these ROS components to work with our mobile robotic platform, and we have built additional ones for the mission control level. The basic schema of the control architecture is depicted in figure 11.2. Note that alternative designs have also been developed to provide for the redundancy requirement pointed out on page 213. They will be presented in the following section.

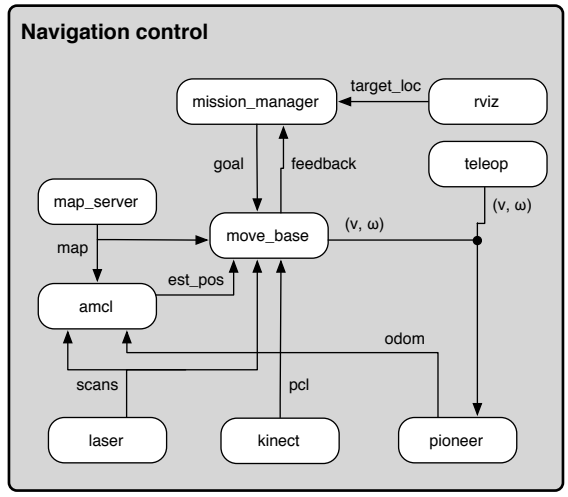


Figure 11.2: The component architecture of the testbed's control system. Only the basic components and signals between them are depicted.

Following we provide a brief description of the main components involved and their role in the application.

[3] `www.willowgarage.com`

Drivers and I/O processing

A set of components perform as sensory drivers: they access the system sensors to convert their readings into messages containing sensory information for those other components that use it.

Pioneer: This package contains the implementation of a ROS node (called `pioneer`) that is a ROS wrapper for ARIA, which is the C++ library for the robot's operation provided by the manufacturer, MobileRobots. The `pioneer` node produces odometry messages and accepts command messages for the robot linear and rotational velocities.

Laser: The ROS package `sicktoolbox_wrapper` was used to implement the laser driver. It uses the SICK LIDAR Matlab/C++ toolbox library[4] from the laser manufacturer. The `laser` node implemented with this package publishes in a topic the laser readings as a LaserScan message type.

Kinect: For the connection with the Microsoft's Kinect sensor, the ROS package `openni_camera` was used to implement the `kinect` component. This package provides a ROS node that publishes PointCloud (PCL) messages[5] containing the image depth readings. There are other ROS nodes associated to the Kinect deployed in the testbed, for example those that provide geometric information on the source of the kinect readings (`kinect_base_link...` nodes).

It is also possible to obtain range scan readings from the Kinect's depth image. This has been used to have analytical redundancy in the system, as will be explained in the following section. A set of ROS nodes and nodelets from the ROS package `pointcloud_to_laserscan` have been used to implement the `pcl2scan` component, which is actually composed of several ROS nodes. A more detailed description of this component can be found in [31, pp. 68–69].

Compass: The ROS `compass` node has been developed for publishing the orientation data coming from the compass sensor as IMU (Inertial Measurement Unit) messages [6]. The `compass` node is a wrapper that connects internally to the CORBA service for the compass. These CORBA services have been developed for the I/O interface with the sensors connected to an Arduino board in the mobile robot, also part of other ASys projects [124].

[4] Available at http://sicktoolbox.sourceforge.net/

[5] Point Cloud is a data type for 3D sensors [122].

[6] The IMU message type is defined in the ROS package `sensor_msgs` and holds orientation data, as well as angular and linear velocities, with the respective covariance matrices. This message type is used in ROS to encapsulate data read from digital compasses, accelerometers and gyroscopes.

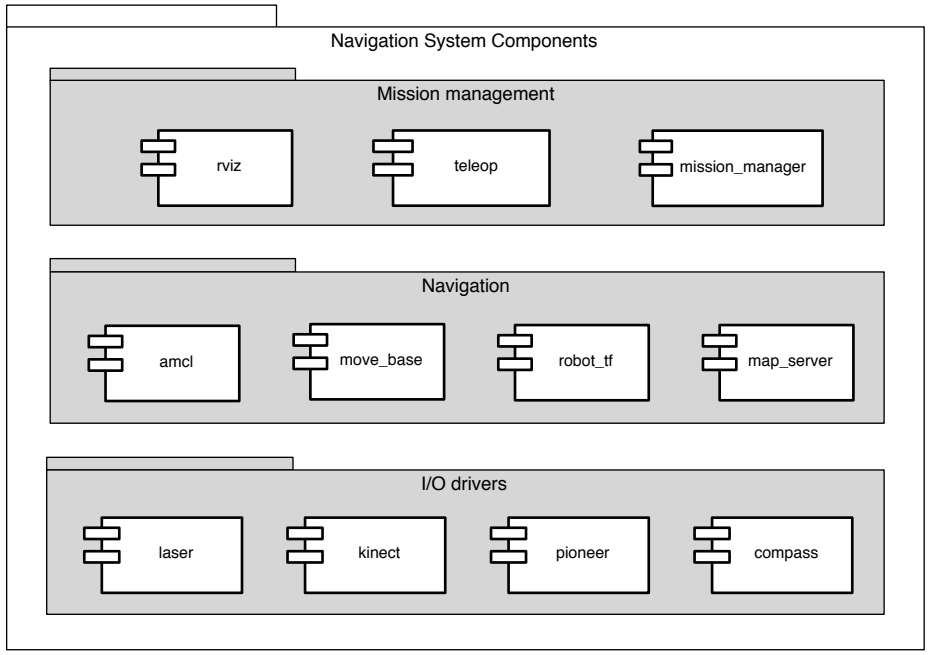


Figure 11.3: Main logical components of the software of the testbed control system.

World and Robot model

The navigation system uses information about the environment, in the form of an initial occupancy map, and the robot, in the form of geometric information that situates its sensory readings in a common frame.

Map server: The `map_server` ROS node loads the map data of the patrolling area from a file and publishes it as an occupancy grid map. The map is essential for the operation of the localisation and the navigation components as will be explained below.

Robot tf model: We have developed the `robot_tf` ROS node for publishing the static geometric information of the robot and its sensors, using the ROS `tf` package for geometric transformations. The information consists of a series of transformations between the different coordinate frames in the robot. Each sensory reading in the robot is referred to its sensor's coordinate frame —the laser, the kinect, the odometry, etc.—. Publishing the transformations between the different frames in a transformation tree allows the use of the readings by any other ROS node, allowing the developer not to need worrying about which coordinate frame the sensory reading is stored in[7].

[7] More information about the ROS tf package can be found at `http://www.ros.org/wiki/tf`

Navigation

To implement the navigation subsystem of the patrolling application we have used the ROS navigation Stack developed by Marder-Eppstein et al. [93] at Willow Garage. It provides ROS nodes connected in a subsystem that takes in information from odometry, sensor streams, and a goal pose —*position* + *orientation*— and outputs safe velocity commands that are sent to a mobile base (`pioneer`). As a result the robot moves to the goal pose without bumping into obstacles.

We have configured the Navigation Stack for the shape and dynamics of our robot platform. The core elements of the navigation system are the localisation and the navigation control components.

Localisation: This package implements the adaptive Monte Carlo localisation approach (or KLD-sampling) as described by Dieter Fox in [49]. The localisation component, named the `acml` node, consumes laser scan and odometry readings (in the form of a `tf` message provided by the `pioneer` node), together with a given occupancy map of the environment, and produces an estimation of the robot pose[8], which it publishes as a ROS message.

The `amcl` ROS node has many parameters that need to be adjusted for each robot, environment and sensors. They can be divided in three categories:

- Overall filter parameters: These parameters specify the particle filter characteristics (min/max number of particles), maximum error of the distributions, update frequencies, the initial pose and covariance of the particle filter, among others.
- Sensor model parameters: These parameters define the model of the range scan sensory source used. These include the laser maximum and minimum range, the number of beams and the measure's covariance.
- Odometry model parameters: These parameters are set accordingly to the robot's odometry characteristics. These include the odometry expected noise in the rotation and translation, as well as the frame's names of the odometry, `base_link` and the coordinate frames published by the localisation system.

Navigation control: the central navigation component is the ROS `move_base` node, and is the node responsible for the path planning for a given destination point, and also the immediate movement of the robot to follow that path. It uses a planning approach based on cost maps: a "cost" value is assigned to each cell in the map depending on its estimated occupancy state, as observed from the sensors readings. To fulfil its role the `move_base` node performs four activities:

- Maintain a local map of the immediate robot surroundings. For this it integrates the information about obstacles given by sensory sources, which in the testbed

[8] The *pose* includes the position and the orientation.

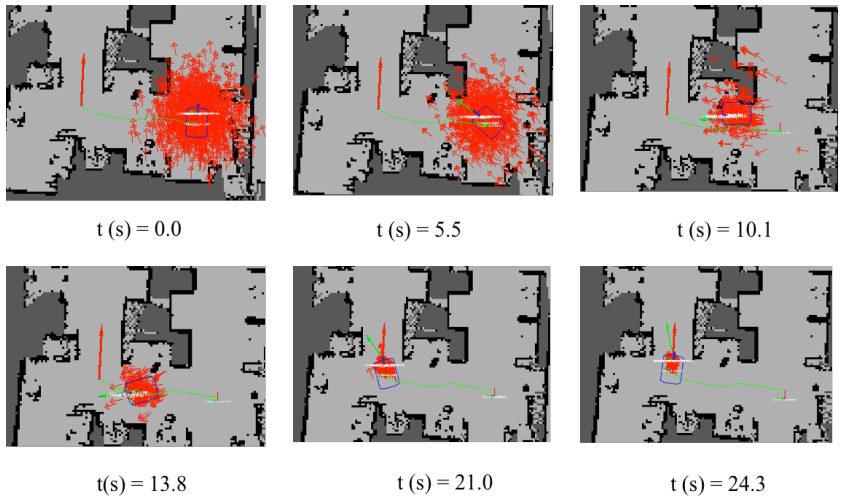


Figure 11.4: Performance of the navigation subsystem integrated by the ROS `amcl` and the `move_base` nodes in a simple navigation task. The figure shows the path computed and followed by the `move_base` and the evolution of the particles for the estimation of the robot pose in the `amcl`, which converges to the actual pose as the robot moves and more obstacle references are sensed.

are the laser scan readings and the Kinect PointCloud data, and considers the robot's dimensions.

- Maintain a global map integrating the information from the initially given occupancy map with that received from sensors.
- Upon reception of the position objective (as a goal message) it produces a trajectory to reach that position according to the global map. An A* algorithm is used for the search.
- Determine the next velocity commands required to follow the computed trajectory, considering the local costmap, and command the immediate one to the robot. A breadth-first search is used to calculate the next velocities (local plan).

Mission Control

The control at the mission level and the Operator Interface have been developed implementing a specific component, the `mission_manager`, and integrating it with the node `rviz` from the ROS package for data visualisation and a node for robot teleoperation.

Mission manager: The `mission_manager` node is responsible for the mission-level control of the patrolling function. It provides the operator with a simple computer

interface to start, pause and resume the patrolling mission, as well as loading the patrolling waypoint from a file.

RViz: The `rviz` node is a graphical interface provided by the ROS visualization packages that displays geometric information of the running system: positions, maps, sensor readings, etc. It also allows to publish messages using that interface, for example selecting a point in the map to provide the robot's initial position for the localisation subsystem, or marking another location as the next goal for autonomous navigation. We have integrated these functions with the `mission_manager`, so that manually commanding a new destination goal in `rviz` pauses the autonomous patrolling, which can be resumed once the manual goal is achieved, or at any moment after cancelling that goal.

Teleoperation: The `teleop` node allows to teleoperate the mobile robot using the remote computer's keyboard. It publishes messages in the `cmd_vel` topic, which is the `pioneer` setpoint for velocities, at a higher rate than the `move_base` node, so that when the operator uses the keyboard to control the robot, she overrides the commands from the autonomous navigation subsystem.

11.2.2 Functional analysis of the mobile robot

The path we have followed to develop the control system in our testbed is slightly different from the standard OMEP Control Development process. The reason is that we have started from the extant ROS navigation stack and other packages. These components already define a certain control architecture, which we have adapted to our requirements. This means than instead of beginning by defining the functional model of the system in design, and then implementing it, we had to realise a functional analysis from the ROS navigation implementation, and then adapt it to our robot. The result of this functional analysis and design adaptation is the functional model of the navigation control of the robot. This model will be the departing point for the development of the metacontrol system.

The functional decomposition of a system is a design decision. Engineers decide how to group components into subsystems that perform specific functions, sometimes depending on the functions realised by other subsystems. Several options are usually possible, all compliant with the system requirements. However, one is chosen because it groups and encapsulates the components in a more suitable way according to the system's requirements of scalability, maintenance, etc. The same thing applies to the functional decomposition we have done of the patrolling mobile robot testbed. Figure 11.5 depicts the functional breakdown into subsystems of the control architecture for the testbed discussed in the previous sections.

MOTION subsystem: contains the `pioneer` node, which allows control and monitoring of the robot's movement, by accepting the set points for the robot's velocities, and publishing ded-reckoning information from the robot's odometry.

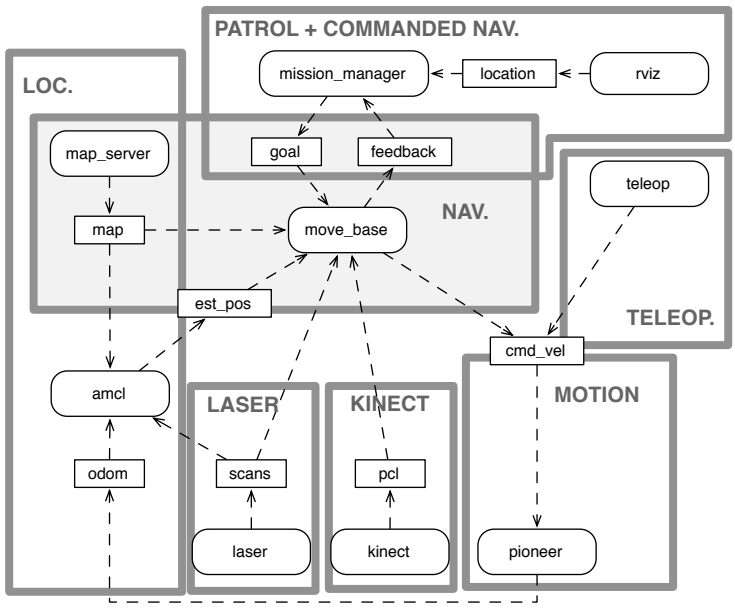


Figure 11.5: Functional decomposition of the mobile robot testbed.

LASER subsystem: is composed of the `laser` node. Its function is to provide range scan readings of distances to obstacles in the environment.

KINECT subsystem: includes the `kinect` node, and its function is to provide a Point Cloud with 3D information of the environment.

LOCALISATION subsystem: is composed of the `map_server` and the `amcl` nodes. Its function is to provide an estimation of the robot's position. To realise this function, it depends on the LASER subsystem, for sensor readings about reference obstacles, and the MOTION subsystem for odometric information.

NAVIGATION subsystem: consists of the `move_base` and `map_server`, and provides the function of autonomous navigation. For that it relies on the LOCALISATION subsystem, that provides periodically an estimation of the robot's position, and the LASER and KINECT subsystems, which provide readings about obstacles in the environment.

MISSION CONTROL subsystem: it is composed of the `rviz` and `mission_manager` nodes. This subsystem allows the operator to perform mission-level control, pausing, resuming and cancelling the patrol mission, as well as commanding navigation to a specific location. It depends on the NAVIGATION subsystem for the robot to navigate autonomously to the destinations commanded.

TELEOPERATION subsystem: integrated by the `teleoperation` node for manually driving the robot.

The previous decomposition of the control architecture results in the functional

dependencies between the different subsystems depicted in figure 11.6.

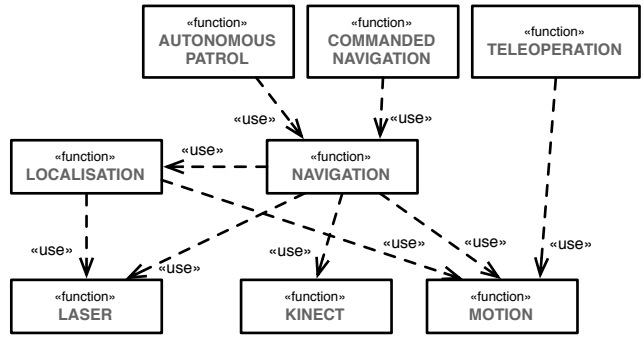


Figure 11.6: Dependencies between the subsystems in the mobile robot.

Note that we could have decided a different set of subsystems, considering different grouping of the components. For example, the `map_server` could have constituted a subsystem on its own, the NAVIGATION and LOCALISATION functionally depending on it, instead of being a role in each of those subsystems. However, having no alternative component or design to provide the map functionality, there is no need for functionally encapsulating it to have a point for variance in the functional architecture. The case of the sensors is just the opposite, we need to encapsulate their functionality to have that function as a point of variance, since the system is to be designed so as to adapt to faults in its sensors, as defined in the scenarios.

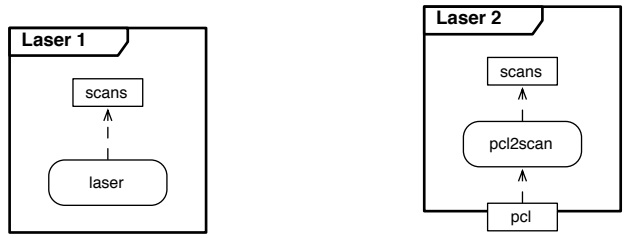
11.2.3 Design alternatives

The functional architecture of the domain control system is not unique and static. There are design alternatives for some of the functions or subsystems identified in 11.6. This is necessary to have redundancy in the system, be it physical or analytical. Each alternative design for a function (a TOMASYs `FunctionDesign`) is a variance point in the functional architecture of the system, so that the different alternatives for the complete control system is the product of the numbers of alternatives in these variance points. Following we describe the alternative designs we have developed to address the requirements of robust autonomy and adaptivity presented in page 223.

LASER subsystem

The `laser` node, implemented using the `sicklms_toolbox` package, can be deployed in the system to obtain scan readings from the SICK laser sensor of the robot. This is a straight-forward design for the LASER subsystem.

There is an alternative to obtain laser-like scan readings when there is no laser sensor available (figure 11.7a). The `pcl` node, implemented using the `pcl` ROS package,



(a) The `laser` node directly provides scan readings.

(b) Laser-like scan readings can be obtained also using the `kinect` and `pcl` nodes.

Figure 11.7: Two different configurations of ROS nodes that provide laser-like scan readings of obstacles in the environment.

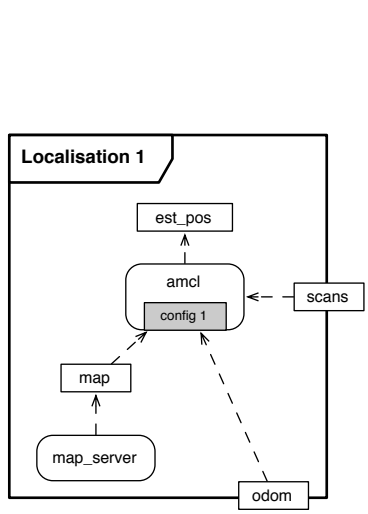
allows to obtain LaserScan (scan) messages from the PointCloud (pcl) data published by the `kinect` (figure 11.7b). However, the scan readings obtained with this alternative design are of a narrower span angle (60° of horizontal span in the Kinect sensor, compared to the 180° of the laser), and lower quality (they are noisier and less accurate).

LOCALISATION subsystem

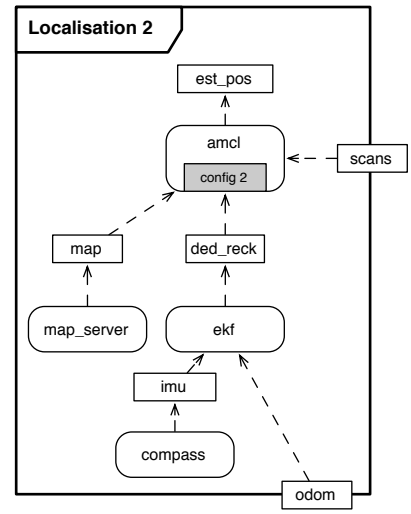
The `amcl` node uses scan readings to match reference obstacles as explained in the previous section. As shown above, these can be easily obtained the SICK LMS laser sensor using the `laser` node. This is the default *laser* design for the LOCALISATION subsystem in figure 11.8a.

However, when these scan readings are obtained from the Kinect's PoinCloud, using the LASER 2 subsystem, they have different characteristics (noise) as discussed above. The algorithm's parameters regarding the sensor model must be adjusted accordingly. However, this is not enough to achieve a LOCALISATION subsystem reliable enough, due to the poor quality of the sensory readings available. This can be overcome by improving the quality of the other sensory source: the ded-reckoning estimation obtained from the robot's odometry. The `ekf` node from the ROS packages, which implements an Extended Kalman Filter, can be used to obtain a better ded-reckoning estimation of the robot position by integrating the odometry information from the `pioneer` with the orientation information provided by the `compass`. This *kinect* design of the LOCALISATION subsystem is shown in figure 11.8b.

Although the robot can operate with this later design for the localisation, as has been previously commented it is less reliable than the design that uses the laser readings for the mapping of reference obstacles. The obstacles mapped with the scans obtained from the Point Cloud information are fewer and with a greater covariance, given the already commented much narrower span angle of the Kinect sensor and the noise of its readings. This results in a poorer performance of the localisation algorithm



(a) The standard configuration of the LOCALISATION subsystem using laser scan readings and odometry information from the robot.



(b) The alternative design of the localisation subsystem that uses an Extended Kalman Filter to improve the ded-reckoning input to the Monte Carlo algorithm.

Figure 11.8: Two alternatives for the LOCALISATION function, depending on the availability of the laser sensor.

than when the laser scans, which include a larger quantity of obstacles detected, are used.

NAVIGATION subsystem

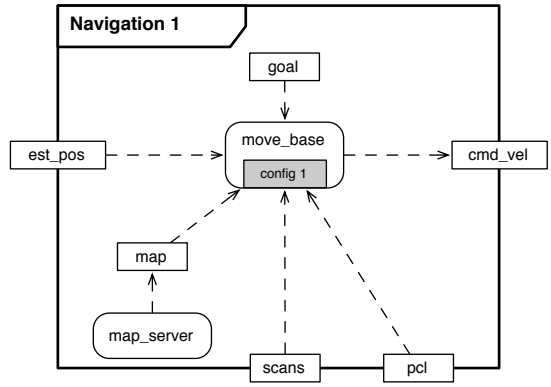
The `move_base` node can use different sensory sources to map obstacles to the robot's navigation. There are different configurations for the NAVIGATION subsystem depending on the sensory sources used. In our mobile robot testbed there are two sources available: the laser scan readings and the kinect's PointCloud. This way we have designed three alternative NAVIGATION systems, depending on which source is used, or if it is both, as explained in figure 11.9.

The *main navigation* design (fig. 11.9a) uses both the laser scan readings and the Kinet's Point Cloud information to update the information about obstacles, the *laser navigation* design (fig. 11.9b) uses only the laser scan readings, and finally, the *kinect navigation* design (fig. 11.9b) uses only the Point Cloud information to map obstacles. In addition to the previous differences between the three configuration for the navigation subsystem, the parameters for the control law applied by the `move_base` that constrain the robot's movement are tuned differently in each of the alternatives. For example, given the much more limited span angle covered by the Kinect sensor (60° against 180° for the laser readings), and the lower update frequency, the `move_base` commands lower velocities for a more safe and smooth movement of the robot.

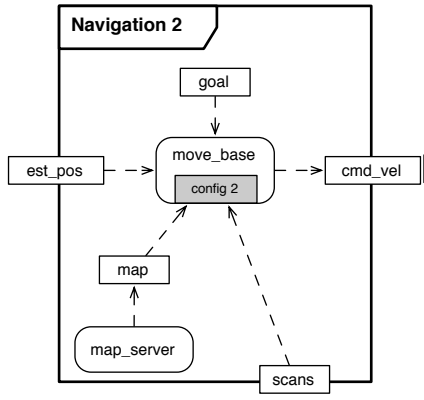
Overall alternative functional designs

Considering the previous alternative designs for the Localisation, Navigation, and Laser functions, we have two alternative control architectures for the complete testbed system, as depicted in figure 11.10, in addition to the main design of figure 11.5 that makes use of the laser and the Kinect sensors. The *laser* design (fig. 11.10a) maintains the good performance of the *main* design, although it is less safe due to the more complete 3D information provided by the Kinect that is missing, whereas the *kinect* design (fig. 11.10b) has a poorer performance.

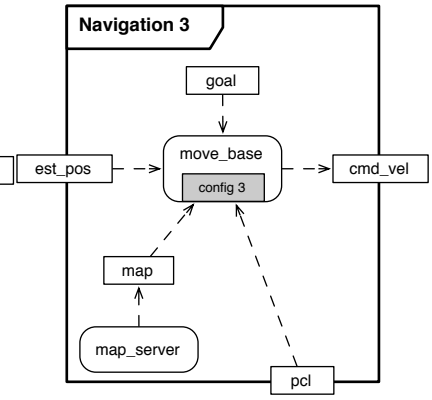
The design alternatives have different performances. The *laser* configuration provides a better and more reliable performance than the *kinect* design, for the reasons already explained. Some quantitative measurements of this has been obtained in experimental works [31] conducted in relation to this research (see table 11.1).



(a) The navigation subsystem using both scan readings from a laser sensor and PointCloud information from a Kinect to map obstacles.



(b) The navigation subsystem using only the laser readings for obstacle mapping.

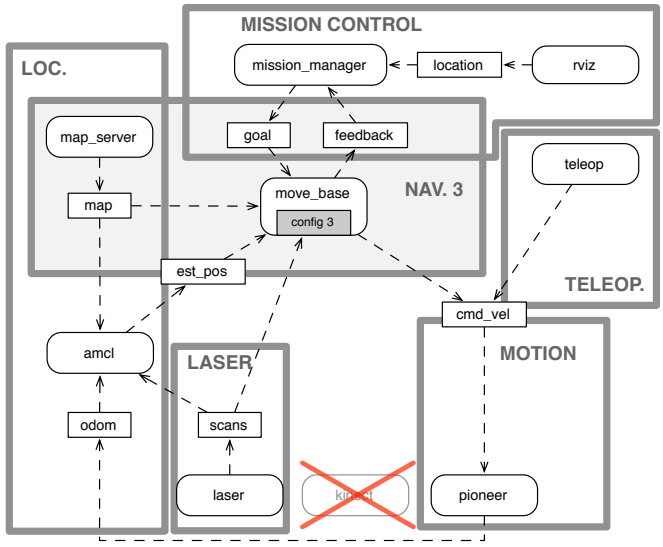


(c) The navigation subsystem using only the Kinect's PoinCloud information for obstacle mapping.

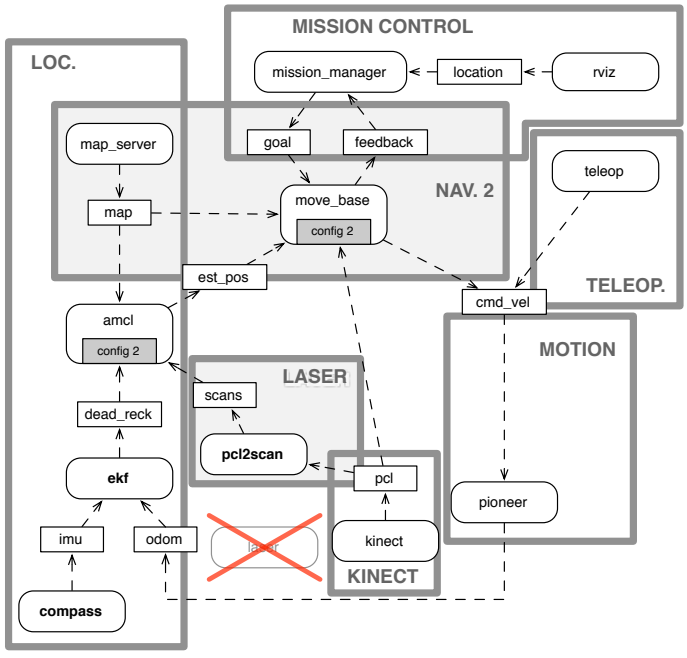
Figure 11.9: Three alternative designs for the NAVIGATION subsystem, depending on the availability of sensors for obstacle perception.

	Lap 1	Lap 2	Lap 3	Lap 4	Lap 5	Lap 6	Lap 7	mean $\pm$ std. dev.
Laser	180.5	198.5	158.9	202.0	153.0	211.6	135.9	177.2 ± 26.4
Kinect	224.9	302.0	346.2	536.4	489.5	331.7	422.2	379.0 ± 101.3

Table 11.1: Time measures, in seconds, of the robot navigation through the rectangular corridors of the patrolling area using the different design configurations. The mean time of each lap when using the laser is less than half the time when the robot uses the Kinect. Besides, the system is much more reliable with the *laser* design: the standard deviation obtained is 26.4 seconds, versus 101.3 seconds of deviation for the *kinect* configuration.



(a) Design of the testbed control architecture that does not use the Kinect sensor.



(b) Design of the testbed control architecture that does not use the laser sensor.

Figure 11.10: Two alternative designs for the testbed control architecture that realise the same functionality to address the system's requirements.

11.3 Metacontrol System Development

For the development of the metacontrol system in the mobile robot the OMEP Meta Development process has been followed. This way, the metacontroller has been developed by first obtaining the TOMASys model of the testbed control system, and then implementing a metacontroller that uses it to adapt the patrolling system if required by the circumstances. Following the MDA approach for OMEP described in section 10.2, the OM Java library has been used for the implementation of the metacontroller, as described on page 218. According to this process, a ROS Java package has been developed, in order to deploy the OM Architecture in a ROS system such as the testbed. This package has been developed to be general for any ROS-based control system, and not just for our mobile robot testbed.

For the building of the metacontrol system, the specific resilience and adaptivity requirements for this autonomous application, which have been introduced on page 223, have been considered. Following they are further analysed.

11.3.1 Metacontrol System Requirements Analysis

The requirements for the Metacontrol System derive from those of the complete testbed that refer to fault-tolerance and adaptability issues, which were presented in page 223. In this section we analyse them in detail, using the method proposed by Fernández et al. [47], in order to check the validity of the OM Framework to address them.

Following Fernández at al. method, we have performed the following tasks in order to elaborate the requirements for the Metacontrol System:

1. envisioned the operational scenarios or use cases,
2. identified operational needs that can be extracted from the previous scenarios,
3. gathered the functional requirements in a tree of capabilities and listed the non-functional requirements.

Operational Scenarios (Use Cases) and needs

These are the operational scenarios envisioned for the Metacontrol System in the mobile robot testbed:

1. S0 – No failure in the testbed system: the Metacontrol receives monitoring information from the patrolling system and maintains a representation of its state. The Metacontrol does not take any action because the patrolling system is working as desired.
2. S1 – Laser transient failure: during regular operation, the robot's laser driver fails and no laser information is received by the patrolling system. The Metacontrol observes this and re-starts the laser's driver.

3. S2 – Laser permanent failure: same incident with the laser's driver than above, but this time re-starting the driver does not solve the problem. The Metacontrol reconfigures the patrolling system so as to use the Kinect sensor input data instead of the laser's for navigation (kinect configuration).

Tables 11.2, 11.3, 11.4 describe the scenarios S0 to S2 in terms of the expected behaviour of the complete system, considering its initial and final states.

The previous scenarios have been proposed to demonstrate the adaptivity capabilities that the OM architecture gives to the control system of the robot. Let us consider them in relation with the metacontrol scenarios envisioned for the OM Architecture in section 9.1.2:

- Scenario 1 requires the usual functionality in active fault-tolerance, that is component recovery. The system detects the failure of one of its components and fixes it. In this case a structural mismatch between the specified (designed) system and the running system is detected and fixed.
- Scenario 2 demands a reconfiguration, since the initial design is no longer realisable; it corresponds to a function recovery. The metacontrol reconfigures the system according to the best alternative functional design that is available.

Operational needs Note that the previous operational scenarios demand the following functionality from the metacontrol system:

- maintain a representation of the functional design for the patrolling system that is running at any instant.
- maintain an updated state of the functions for the currently instantiated functional design.
- produce a new instance of a functional design according to the available components for the patrolling system.
- memory of past reconfiguration actions.

These requirements can be addressed by the OM Architecture (See 9.1.2)

Metacontrol System Requirements

We can apply for the metacontrol system requirements the same analysis that we have performed for the control system:

Functional requirements: they are a particular instance of those specified for any autonomous system in the OM Architecture. We have also added the following requirement regarding the operator interface for the metacontroller:

- the metacontroller is user-controllable, independently of the patrolling system, for example to switch in between operational modes: start, stop, sensing-only mode (in which the Metacontrol System monitors the patrolling system, but does not actuate upon it).

Table 11.2: Regular operation of the patrolling system with no failures.

Name	**S0: No failure in patrolling system**
Description	The metacontroller receives monitoring information from the patrolling system and maintains a representation of its state. The metacontroller does not take any action because the patrolling system is working as desired.
Preconditions	The patrolling system is up and running, the metacontroller system is also running.
Periodic	The patrolling system works properly with the configuration desired.
Basic Flow	The metacontroller monitors the state of the patrolling system, by periodically updating the model it maintains of its components and functions. Since that state complies with the desired design, no action is needed.
Postconditions	The patrolling system works as desired without the metacontroller's intervention.

Table 11.3: Laser transient failure scenario.

Name	**S1: Laser transient failure**
Description	During regular operation, the robot's laser sensor fails and no laser information is received by the patrolling system. The metacontroller detects this abnormal situation and actuates to recover from it by re-starting the laser driver.
Preconditions	The patrolling system is up and running with the *main* configuration, the metacontroller system is also in operation.
Triggering event	The laser driver fails and stops providing laser data to the patrolling system.
Basic Flow	1. Detect errors in the components of the patrolling system. 2. Update the state of components observed: the laser driver is in error. 3. Re-start the laser driver.
Postconditions	The patrolling system resumes its mission and continues moving to the waypoint it targeted when the scenario was triggered, operating with the *main* configuration, as it was before the triggering event. The metacontroller continues operating as previously, monitoring the state of the patrolling system.

Table 11.4: Laser permanent failure scenario.

Name	**S2: Laser permanent failure**
Description	The laser driver stops working and re-starting it does not solve the problem. The Metacontrol reconfigures the patrolling system so as to use the Kinect sensor input data instead of the laser's for navigation (kinect configuration).
Preconditions	The patrolling system is in laser configuration, but not running because of a laser driver failure, which has caused the system to undergo the laser transient failure scenario, not having returned to the normal operation scenario.
Triggering event	The laser component class ceases to be available.
Basic Flow	1. Detect errors in the components of the patrolling system. 2. Update the state of components observed: several functions are in failure, but the origin is an internal failure of the function to provide laser readings. 3. Update the functional status of the system: the function to obtain laser readings is no longer available. 4. The patrolling system is reconfigured to the *kinect* configuration.
Postconditions	The patrolling system resumes its mission by navigating to the waypoint it targeted when this scenario triggered, but now in the *kinect* configuration. The metacontroller continues operating as previously, monitoring the state of the patrolling system.
Includes	The laser transient failure scenario up to its postconditions, which do not hold in this case.

Non-functional requirements: are those defined for the OM Architecture:

- minimal intrusion in the performance of the patrolling system for monitoring,
- minimal reconfiguration time,
- safe transition between design configurations,
- minimal impact on the patrolling system's performance between functional configurations.
- failure in the metacontrol must not affect operation of the patrolling system.
- economy – performance-resilience trade-off [9].

11.3.2 Metacontrol Design for the Testbed

An OM-based metacontroller has been developed for the testbed patrolling system that addresses the previous metacontrol requirements. Note that this metacontroller could have been developed from scratch, departing just from the OM reference architecture provided in chapter 9. However, the objective of the testbed was to validate not only the thesis abstract principles, but also the framework and assets produced for their reification and application in the practical engineering of autonomous systems. It is for this reason that we have used the OMJava library to develop the metacontroller for the testbed robot.

Using the OMJava library simplifies the design and implementation of the metacontroller, since the library already contains an implementation for a generic OM-based metacontroller, requiring only:

a) the integration of the OMJava implementation of the OM metacontroller in the ROS component platform of the testbed,
b) the implemenattion of a Meta I/O module that complies with the MetaInterface be provided, in order to connect the metacontroller to the running control system,
c) a OM-TOMASys model of the testbed control system, embedding it in the metacontroller for its run-time exploitation.

Instead of developing a testbed specific implementation in order to address the previous requirements for the testbed, we have developed a general ROS stack[10] that addresses them for any ROS-based application. This is possible thanks to the MDA

[9] We could have the metacontrol system apply two design policies: i) have several function groundings realised for a single objective (both the laser and the kinect to obtain range scan reading), which is more expensive but improves resilience and performance (no need to reconfigure when one of them fails), or ii) have only one function grounding instantiated, and reconfigure the system to a different one if it fails. This second option is more economic in terms of resources, but hampers system performance because of the reconfiguration needed. Despite we have configured the OM Architecture functional reconfiguration according to the second option, it would be interesting to explore the possibility of a more sophisticated decision mechanism capable of optimizing a joint function of performance, economy and resilience

[10] A ROS *stack* is a set of related libraries, or *packages* in ROS terminology.

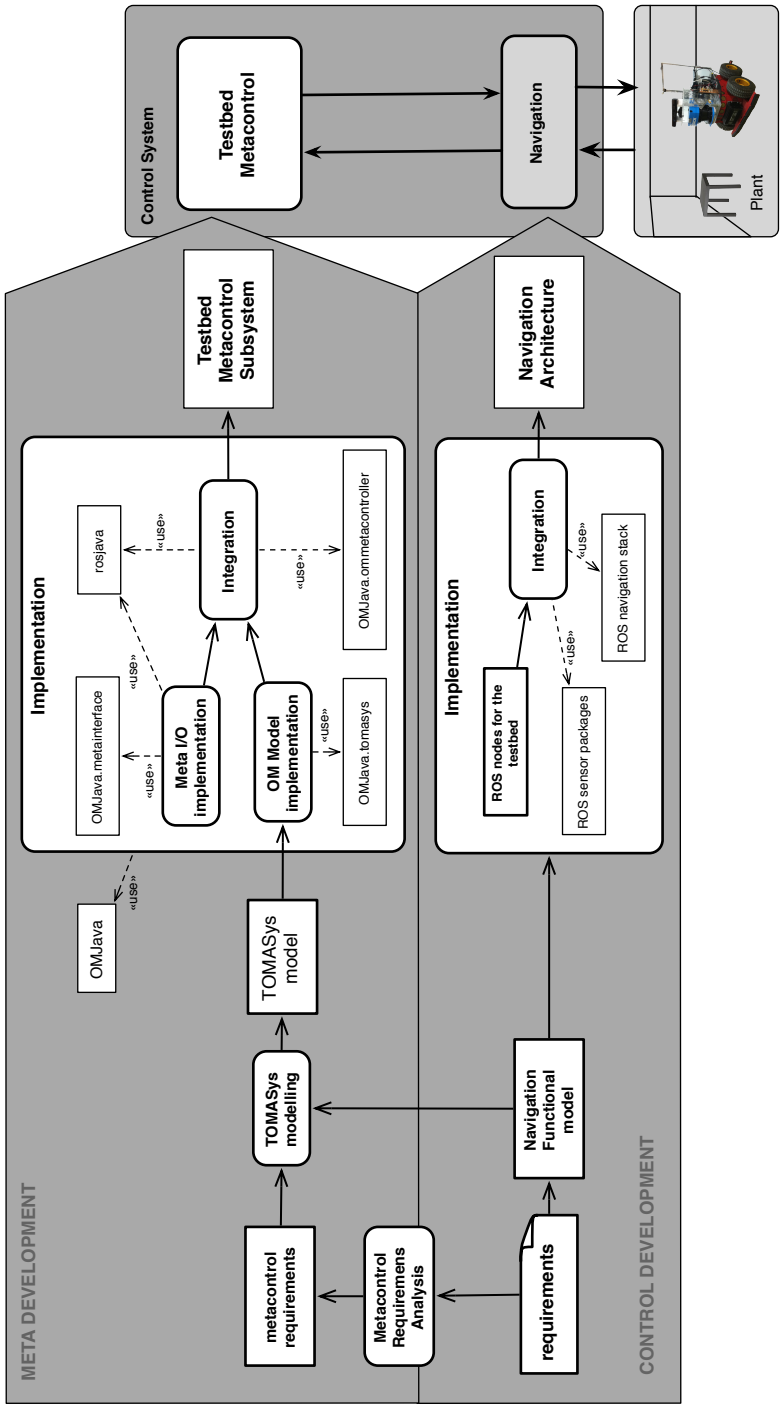


Figure 11.11: The OM Engineering Process applied to the mobile robot.

approach taken in the definition of the OMEP methodology and the extensibility characteristics of the OMJava library that support it. Figure 11.12 schematizes this 2-step *weaving* process. In the first step, the model of the OM Architecture is OMJava, which is specific for the Java implementation platform. This model is woven with `rosjava`, which is the extension of the ROS component platform for Java, resulting in the `omros` , which is a model of the OM Architecture specific for the Java implementation platform and the ROS component platform.

11.4 ROS implementation of the OM Architecture

Following we describe the ROS stack **omros**, which is the set of software libraries that have been developed for the application of the OM Architecture Framework in ROS-based systems. It implements the OMEP methodology and allows the integration of OMJava metacontrollers in control systems implemented with the ROS component framework. The omros stack was initially developed for the *Electric* version of ROS, but it has recently been updated to work in ROS *Fuerte* too.

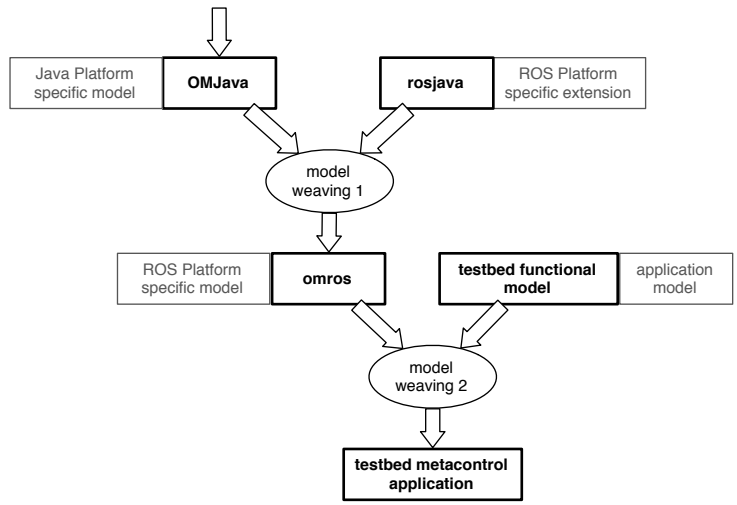


Figure 11.12: The MDA model weaving applied to the testbed. In the first step "model weaving 1", the OMJava library is woven with the `rosjava` ROS library to obtain omros, the implementation of OM for the ROS component platform. The final step"model weaving 2" consists in obtaining, from omros and the functional model of the control application, the metacontroller for the testbed application.

The omros stack provides a ROS node that includes an instance of the OMMetacontroller, to allow its deployment in a ROS system, and implements a Meta I/O module to connect it with the rest of the system for metacontrol purposes. Additionally, omros includes a library of Java classes that implements a TOMASys model of the

generic ROS node. This way, we have separated the generation of the OM Model for the testbed application in two steps, the first one, involving the building of this library, specific for the ROS platform, which can be reused for other ROS systems, and a second one, application-specific, implemented in this case for our concrete testbed robot. This reifies the MDA model weaving strategy for the ROS platform (see figure 11.12), as defined for the OMEP methodology in the previous chapter (page 215).

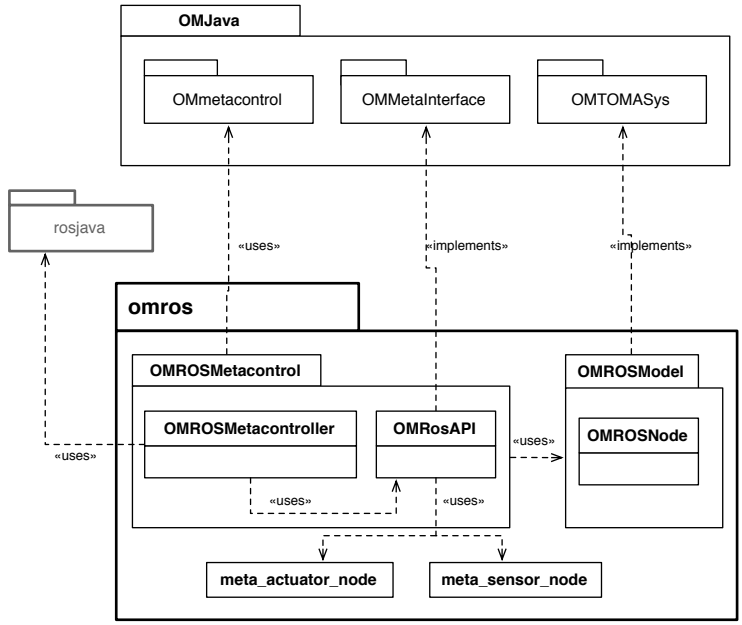


Figure 11.13: The elements of the omros stack, and their relations to the OMJava library.

11.4.1 OM-based ROS Metacontroller

The ROS package omrosmetacontrol contains all the classes needed to instantiate an OM metacontroller in a ROS system. It is basically a wrapper for the OMJava library in ROS.

The OMROSMetacontroller class in the package implements the `meta_controller` ROS node, which includes an instance of the OMMetacontroller class of OMJava, i.e. the metacontroller. OMROSMetacontroller uses the OMRosAPI class to connect to the ROS system for monitoring and reconfiguration. This class is an implementation of the OM Meta I/O described below. The OM Model for the metacontroller is dynamically loaded upon initialisation. Its implementation will be described later. Additionally, OMROSMetacontroller includes a basic console interface for the human operator to supervise and manage the operation of the metacontroller.

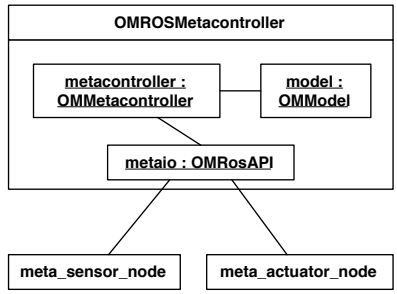


Figure 11.14: The metacontroller connects to the metacontrol infrastructure developed for ROS through OMRosAPI.

11.4.2 ROS Meta I/O module

The ROS component platform does not provide a standard monitoring and management infrastructure that could directly serve for the OM Architecture. However, we have developed a simple one making use of the ROS system libraries. It consists of two ROS nodes, to which the OMROSMetacontroller connects through the OMRosAPI class, which implements the Meta I/O module (see figure 11.14).

Monitoring and reconfiguration infrastructure

ROS offers a Python API for internal development that allows access to the running ROS system. We have used it to implement a basic infrastructure for metacontrol purposes. In order to integrate it with our Java-based implementation of the OM metacontroller, we have developed two ROS nodes that encapsulate the access to the ROS system with this Python API:

The `meta_sensor_node` implements the monitoring mechanisms. It periodically accesses the ROS Master and the Parameter Server[11] to gather information about the running nodes that form the control application, and publishes messages with that information (`meta_singularities` topic/signal in fig. 11.15).

The `meta_actuator_node` implements a limited set of the reconfiguration actions defined by the OM Architecture, because there is no infrastructure for online reconfiguration of individual components in ROS at the time of this research. This way, the node can only perform atomic operations to KILL a node or LAUNCH another with a given configuration. These commands are received through the `meta_action` topic.

ROS message types[12] have been defined for each of the topics involved in the operation of these two nodes.

[11] The ROS Master and the Parameter Server are at the core of the ROS infrastructure, and contain the information of the nodes and their properties in the running system. More information can be found at `http://www.ros.org/wiki/Master`

[12] ROS message types are the data structures for the information flowing through the ROS topics.

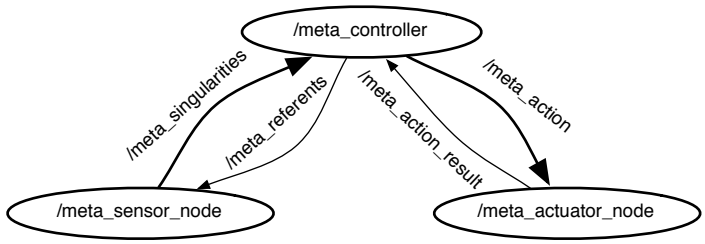


Figure 11.15: The ROS nodes that compose the metacontrol system for the testbed.

OMRosAPI

The OMRosAPI class implements the Meta I/O module. It connects to the `meta_sensor_node` for obtaining monitoring information about the ROS node in the system by subscribing to its `meta_singularities`. OMRosAPI performs a mapping from the information in the message ROS type `NodeState`, which is ROS-dependent, into the platform independent ComponentObservation data type defined by the OM Architecture. Additionally, OMRosAPI converts the filtering signal into the ROS message type `FilterNodes` to command to the `meta_sensor_node` the nodes from which obtain information.

Regarding the configuration of components, OMRosAPI provides a mapping from the action vocabulary defined in OM (page 172). For each action in the reconfiguration signal it receives from the metacontroller, a ROS Task Executor is dynamically instantiated in its own thread. The executor decomposes the component action into the available ROS reconfiguration actions and commands them by publishing `ROSaction` messages in the `meta_actions` topic. The commands for a component action form an action plan: they must be executed orderly because of dependencies. For example, to change the configuration of a node, its state has to be saved, then it can be killed and finally re-launched with the given configuration and the saved initial state. The executors monitor the execution status of the commands through the `meta_actions_results` topic. Upon termination, they send the result of the execution of their corresponding component actions in the effect signal to the metacontroller.

11.4.3 OM-TOMASys model of a ROS system

In order to easily build the OM Model of any ROS-based system, a mapping between the ROS component model and the OM-TOMASys metamodel has been defined. The omrosmodel Java library has been developed to implement this mapping. It can be considered the result of the model weaving of the Java specific model of TOMASys given by the OMTOMASys Java package and the ROS component model.

TOMASys representation of the ROS component model

There does not exist a formal specification of the ROS component model. This complicates the application of our MDA modelling approach in the construction of automatic transformations. However, we have defined the mapping between the ROS and TOMASys metamodels as follows:

ROS nodes are modeled as TOMASys `components`. Their ROS parameters can be represented as TOMASys `parameters`, the definition corresponding to the TOMASys concept of a `Parameter Profile`. The subscriber and publisher elements of a node, each one publishing or subscribing to a type of ROS message, can be represented as `port profile`. The instantaneous connection of a ROS node to a topic is thus represented by a TOMASys `port`.

ROS does not include a functional modelling methodology or any special considerations in that respect, so the functional part of the TOMASys model of a ROS system can be obtained from the functional model obtained in the design of the system, as will be illustrated for the testbed in section 11.5.

omrosmodel library

The omrosmodel library has been developed to facilitate the building of the OM-TOMASys model of any ROS-based control system. The library provides default constructors for the elements in the OM Metamodel, so they can be directly used in the implementation of the OM Model of the concrete application, for example the testbed system described in this chapter.

The OMROSnode class, which inherits from the OMComponentClass in OM-Java, implements the OM-TOMASys representation of a standard ROS node, and is the main class in the library. To represent the components of a particular ROS system, classes inheriting from OMROSNode provide with default properties that can be overridden with those particular to the component if needed. This is described in the following section.

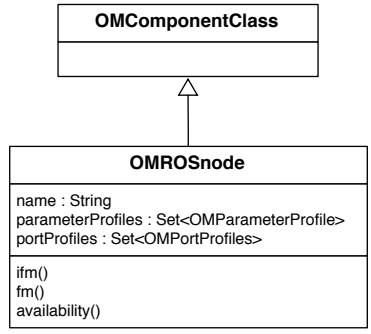


Figure 11.16: The OMROSnode class developed to implement the OM-TOMASys representation of ROS components.

11.5 OM-TOMASys model of the testbed

The provision of the previously described omros stack simplifies the OMEP implementation of the metacontroller for a ROS control system in the generation of the OM-TOMASys Model of it. In this section the model implemented for the testbed robotic system is described.

The OM-TOMASys model of the testbed must include all the knowledge required to address the metacontrol requirements. This includes the long-term knowledge atoms necessary to represent the functional model of the testbed control system and the available alternatives, as described in previous section 11.2.2, as well as the knowledge about the system's components that play the necessary roles in those functions, described in section 11.2.1.

To build the OM Model of the Testbed, the omrosmodel library has been used, in addition to the OMJava.tomasys classes. Java classes have been implemented extending the classes in these libraries to represent the knowledge atoms required. Following we describe the implementation of all these atoms that compose the OM-TOMASys model initially provided to the metacontroller for its operation. This model is maintained and updated by the metacontroller according to the state of the control system observed.

11.5.1 Components

The TOMASys `component` classes that represent the knowledge about components in the testbed system have been obtained almost directly from the ROS node types involved in the application. The mapping defined in section 11.4.3 has been used. Some of the nodes, however, have been grouped into a single `component class`, since they work as a single unit and no finer level of granularity was required; this is the case for example of the nodes involved in the operation of the Kinect sensor.

In order to produce the knowledge atoms about components the OMROSnode class of the library has been extended to model each of the components in the testbed. For example, specific `internal error models` have been obtained for the components and implemented in the corresponding classes by overriding the default `ifm` defined in the OMROSnode class. Additionally, specific factory methods have been implemented in these classes. For example the Laser class includes a method to produce atoms that represent components of type laser, with the `parameters` and `port profiles` characteristics of the `laser` node in the testbed control system.

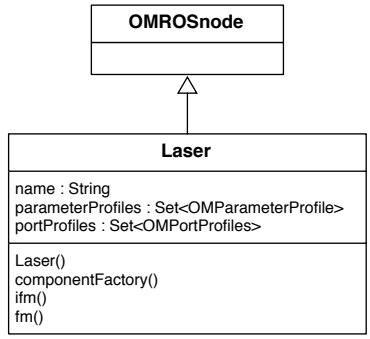


Figure 11.17: The Laser class implements the OM-TOMASys model of the laser sensor of the testbed.

11.5.2 Metacontrol goal for the testbed

The goal of the metacontroller, as defined in the OM Architecture, is a set of TOMASys `root objectives` that represent the requirements about robustness and adaptivity concerning the high level functions of the system. In the case of our mobile robot testbed, it is the realisation of the navigation function[13].

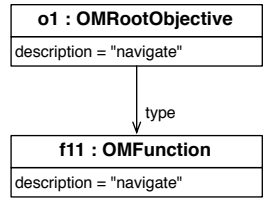


Figure 11.18: The goal of the testbed's metacontroller.

11.5.3 Functions

Figures 11.19, 11.20 and 11.21 depict the long-term knowledge of functions in the OM Model of the testbed, considering only the navigation function[14]. It consists of the TOMASys representation of the functional analysis and the design alternatives described in section 11.2.2.

[13] Note that we could have considered a metacontrol of all the high level functions of the testbed identified in section 11.2.2.

[14] The model of the rest of the high level functions can be found in annex V in page 291.

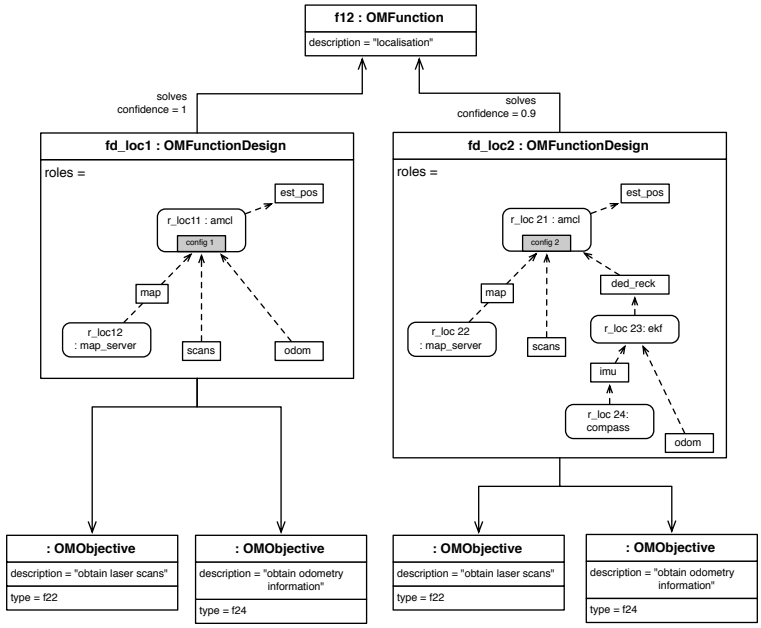


Figure 11.19: TOMASys representation of the alternative designs for the localisation function.

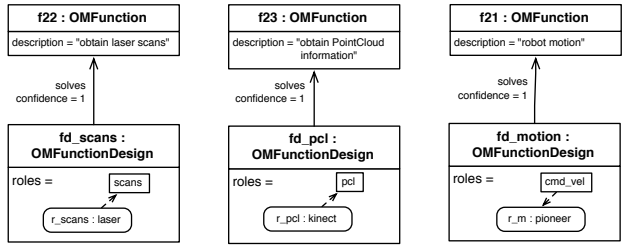


Figure 11.20: TOMASys model of the low level functions in the patrolling robot.

Figure 11.21: TOMASys representation of the alternative designs for navigation.

11.6 Testbed metacontrol operation and results

The metacontrol system described in the previous section has been validated in the failure scenarios defined in section 11.3.1. In order to do that the complete mobile robotic system has been deployed to execute a pre-defined patrolling mission in autonomous navigation mode (see figure 11.22) in several trials. During each trial one of the envisioned failures was simulated.

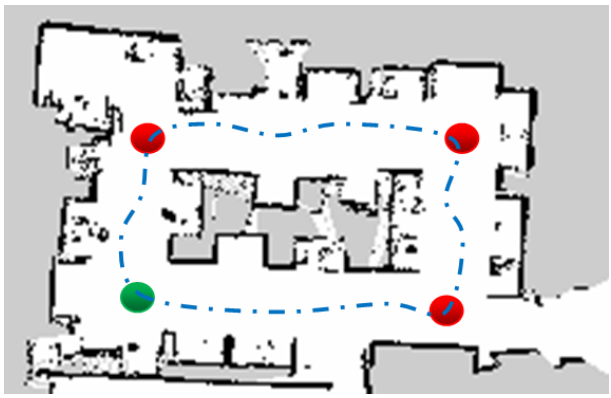


Figure 11.22: The robot's path and waypoints for its patrolling mission.

To integrate and deploy the metacontroller with the rest of the components of the ROS navigation system one has simply to execute the ROS node defined by the OMROSMetacontroller class, passing it as a parameter the Java class that contains the initial OM-TOMASys model of the system, defined in section 11.5.

An interesting property of the OM metacontroller is that, in order to deploy and start the whole system, only the metacontroller needs to be launched, apart from the physical infrastructure of the robotic system. The rest of the software components of the control system will be automatically launched by the metacontroller, because it will detect the absence of components to fulfil the roles in the functional hierarchy that is its goal. The metacontroller will bootstrap the whole system.

The initial conditions are the same for all the scenarios: the robot's navigation architecture is already up and running as specified, as well as the metacontrol system, and the OM-TOMASys estimated state contains the `component` and `function grounding` atoms that represent the instantiation of the main design, schematized in figures 11.23 and 11.24.

In the following we describe the operation of the metacontroller in each of the scenarios, discussing how it results in different reconfigurations of the control system to adapt it to the circumstances.

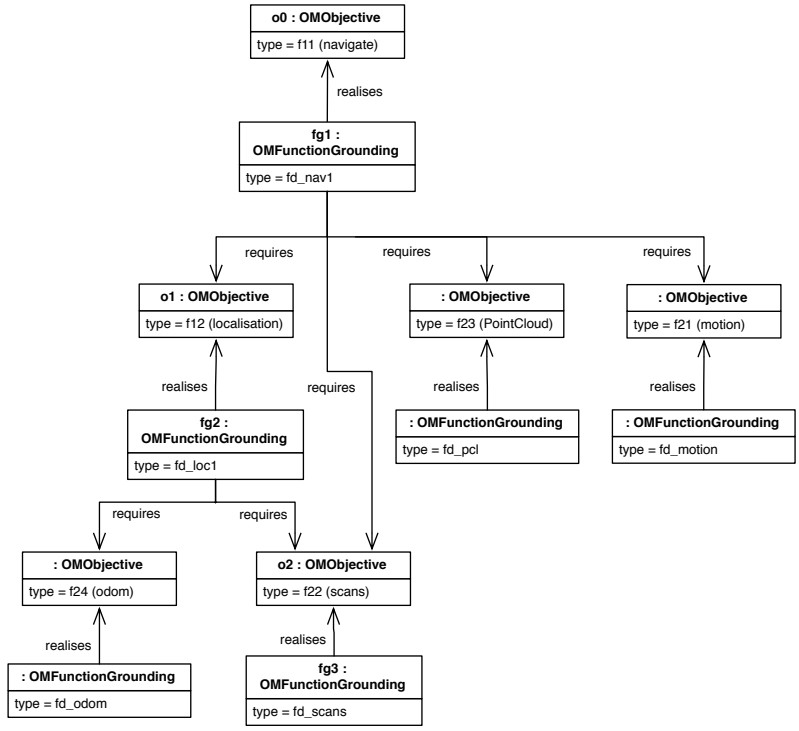


Figure 11.23: The state atoms of the `functional hierarchy` in the initial state of the run-time OM-TOMASys model of the testbed.

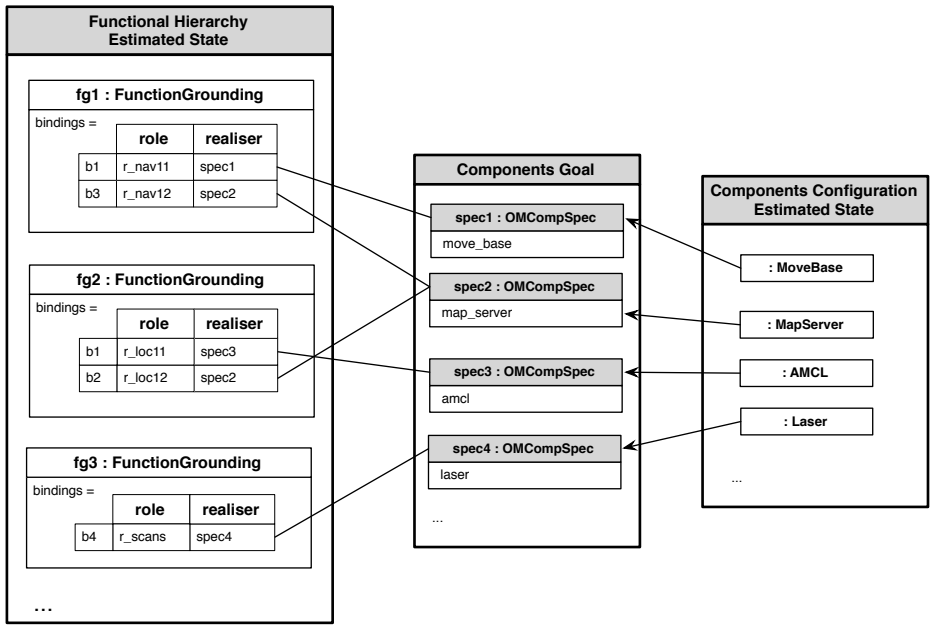


Figure 11.24: Some of the roles, component specifications and components state atoms in the initial state.

11.6.1 Scenario 1: Laser temporary failure

In this scenario, only the Components Loop of the metacontroller actuates to recover the system from a temporary failure in the laser component. The sequence of activity in the system proceeds as follows. While the robot is navigating, the laser sensor is disconnected. This produces an error in the `laser` ROS node, which stops publishing scan readings. The error is logged in the ROS system, and the `meta_sensor_node` of the metacontrol system gathers that information, so the Meta I/O sends it to the metacontroller in the Component Observation monitoring signal about the laser component.

At the Components Loop, during the update of the estimated state of the components, the error in the log of the laser component is mapped to an ERROR status of the `component` by the TOMASys `ifm`. The evaluation phase does not detect any other deviations from the desired goal of component specifications that this ERROR status of the laser component. A RESTART action (A1) over the `laser` node is therefore computed and executed through the Meta I/O, where an executor is instantiated that produces the ROS commands to first KILL the previous `laser` node, and then, upon success of the execution of the previous, LAUNCH a new `laser` node with the same configuration.

The final state of the robot's control system, of the metacontroller and of the runtime OM-TOMASys model of the system, is exactly the same than those at the begin-

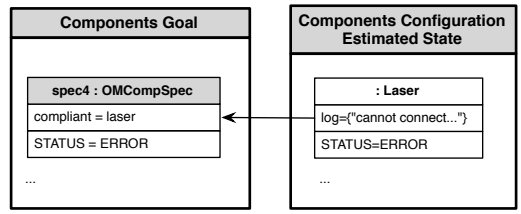


Figure 11.25: State atoms representing the laser failure in the estimated state and the goal of the Components Loop.

ning of the scenario.

It is interesting to note that the internal failure of a component is a particular case of the general scenario of a component non complying with its specification and thus not fulfilling its role. Other such scenarios are those in which the component is in a different configuration from the specification, e.g. because of a wrong parameter value or an undesired connection to other component. Some trials were also successfully performed for this scenarios, in which the metacontroller successfully reconfigured the robot's control system when it was incorrectly initialised.

11.6.2 Scenario 2: Laser permanent failure

The flow of activity in the metacontroller is the same as in the first scenario, from the laser failure to the execution of the metacontrol RESTART action A1 to reconfigure the system by re-starting the `laser` node. Then it diverges, when that action fails.

This failure is identified by `meta_actuator_node` when the command to LAUNCH the `laser` node produces an initialisation error. This information is sent to the OM-RosAPI instance, where the executor in charge of action A1 interprets the command failure and sends a Meta I/O effect signal to the metacontroller with the FAILURE of A1.

At the metacontroller the previous information is interpreted by the model of the laser `component class`, which establishes that a failure in a configuration action over a component of laser type implies that the `component class` is no longer available.

a. The component subgoal (specification atom) of having the `laser` becomes in ERROR.
b. The `component class` representing the `laser` becomes UNAVAILABLE.

The failure then scales up to the metacontroller's Functional Loop. The perception process updates the state of the functional hierarchy. Firstly, the UNACHIEVED status of *spec4* (the laser component specification) triggers the error of *fg3*, the grounding of the scan function.

The TOMASys failure models for the objectives and functions update the `objectives` and `function groundings` that require downwards *fg3*, as depicted in figure 11.26.

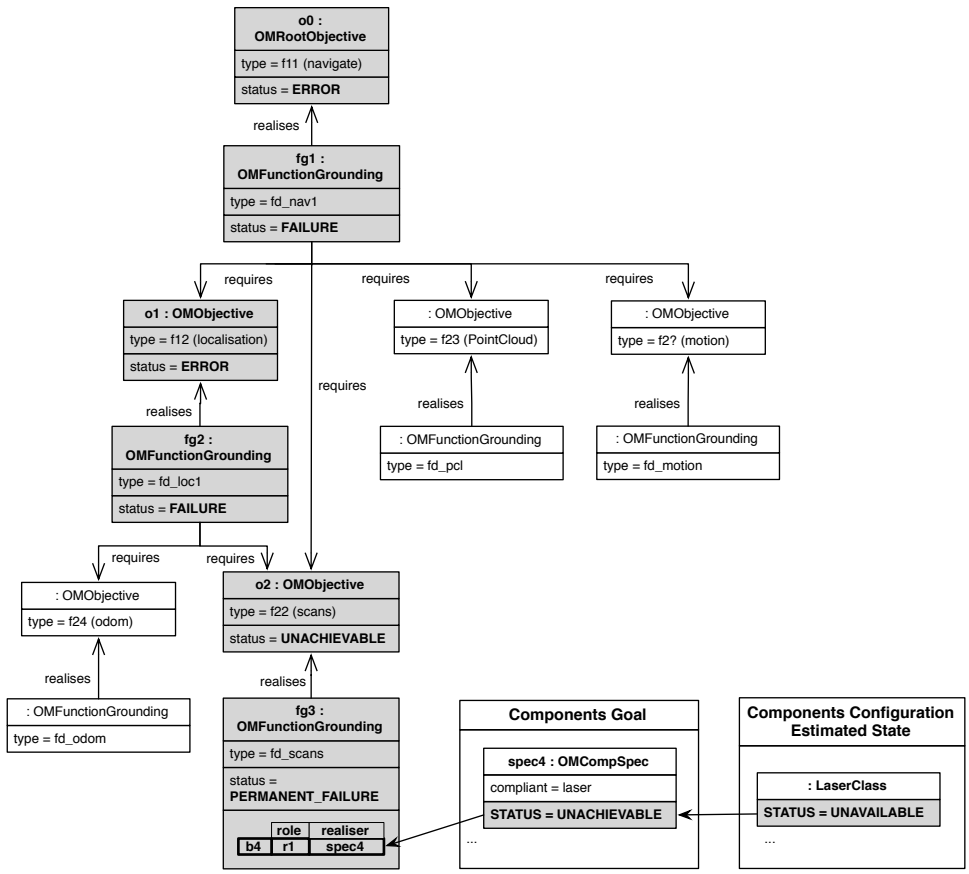


Figure 11.26: State atoms representing the laser permanent failure in the estimated state and the goal of the Components Loop, and its effect on the functional hierarchy perceived at the Functions Loop.

Additionally, the UNAVAILABLE status of the Laser `component class` makes the `function design` that uses a role of that class also unavailable (`availability`= 0). This is the case of the functions *fd_nav1* and *fd_loc1* currently instantiated, and *fd_nav2* in the knowledge atoms.

Given that the functional perception results in the ERROR of the *o0* navigation `root objective`, a functional reconfiguration of the system is required. It is computed by the Functional Loop of the metacontroller as follows.

Firstly the evaluation process updates the relevance of the currently instantiated `objectives`, following the procedure described in page 197. This results in all the objectives in the hierarchy but the single root objective being updated to `relevance`= 0, since all the hierarchy is devoted to comply with the requirements of the function *fg1*, which is in FAILURE and no longer available. Then, a new design that realises *o1* is computed using the available alternatives and following the algorithm described in page 198. This results in the generation of functional state atoms to represent the desired *kinect* design alternative for the testbed navigation system.

The functional hierarchy is updated with all the generated groundings and required `objectives`, whereas the `component specifications` corresponding to the required `roles` are sent as the new Goal of the Component Loop (see figure 11.27).

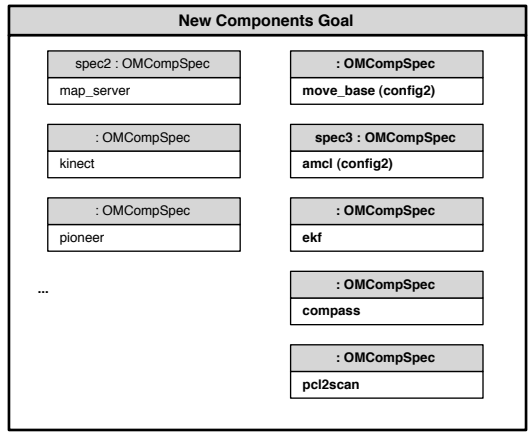


Figure 11.27: The new goal of the Components Loop. Some of the specifications (on the right), correspond to the new roles required.

Finally, the Components Loop, following its regular operation launches all the components that are required for the new specifications, deactivating those no longer needed.

(a) (b)

Figure 11.28: The navigation of the robot before (a) and after (b) the laser permanent failure (an X marks the moment in the robot's path when it occurred) and the reconfiguration from the main design to the kinect. The red dots correspond to the range scan reading of obstacles. Note the lack of lateral references due to the narrower angle span when in the kinect configuration.

11.7 Analysis

The testbed developed and tested in experimental scenarios, which has been described along this chapter, has demonstrated the validity of the approach for the engineering of self-awareness and adaptivity properties in autonomous systems. The OM-based metacontroller developed for the autonomous mobile robot has provided it with the capacity to detect a malfunctioning of the operation of its control system at both the component and the functional levels, and overcome it by adapting through reconfiguration. However, some improvements for the OM Architectural Framework have been identified that can be ilustrated with the previous scenarios. Following we discuss the results achieved and these further considerations.

In relation to the engineering of the metacontrol system, the OMEP model-weaving methodology has demonstrated its validity not only to build the final metacontroller for an autonomous application, but also to produce reusable intermediate assets. The OMJava library, applicable to the development of OM-based metacontrollers for any component-based control application suitable to embed a Java component, and the omros ROS stack, directly applicable to any ROS system with no more than defining its OM-TOMASys model, are proofs of this.

The OM model-based and architectural approach has a clear positive impact on the properties of the autonomous system, such as its scalability and maintainability. The Meta I/O is an interface that decouples the metacontroller and the control system,

so that each one can be developed independently. This way, the internal architecture (OM) of the metacontroller and its implementation can be optimized, extended or improved, without affecting the implementation of the control system. But more importantly, the ECL pattern model-centric design of the OM metacontroller decouples its operation from the information about the (meta-)controlled system. This way, the control system of the autonomous operation can be completely modified without affecting the metacontroller. To maintain the metacontrol system it is only needed to update the OM-TOMASys model of the control system.

Regarding the online operation of the metacontrol system, the success of the trials performed according to the scenario of the transient laser failure demonstrates that the developed OM metacontroller provides the system with fault-tolerance capabilities based on component-recovery. But furthermore, the second scenario of a permanent laser failure, in which a simple yet actual on the fly system re-design is required in order to maintain the robot's functionality, proofs that the OM Architecture also purveys this capability.

Notwithstanding, the OM Architecture can be improved in many aspects. For example, in the first scenario, the final state of the robot's metacontroller and, more importantly, the run-time OM-TOMASys model of the system are exactly the same as at the initial time. This is one point where the TOMASys metamodel and the OM Architecture could be extended and improved: the model of a component class could be further extended to include reliability information. The metacontroller could then learn from an episode such as that of scenario 1 and update the model of the laser, by modifying the failure model of the `Laser component class`, or a new numeric attribute representing its reliability.

Chapter 12

Conclusions and Future Work

This work is of a deep exploratory nature: a diversity of fields have been analysed, which deal with a variety of themes ranging from engineering frameworks, to philosophical concepts, and including scientific theories and models; all this in order to investigate the possibility of building general self-awareness capabilities into machines, so that they become more robustly autonomous.

This concluding chapter analyses the accomplishment of the objectives initially addressed in this thesis. The benefits and novelty of the solution proposed, the OM Architectural Framework, are discussed. Shortcomings and difficulties are also pointed out. Finally, possible extensions and improvements of the work are identified, and future research lines to address them are proposed.

12.1 A universal framework for self-awareness in autonomous systems

There is a need for building more autonomous technical systems, with a special focus on dependability issues, as was discussed in the first part of the dissertation. Intelligence and cognition have been thoroughly researched for solutions to improve autonomy in artificial systems. In the recent decades, the value of self-awareness capabilities is also being explored for technical systems, for example in the fields of autonomic computing and machine consciousness. The main purpose of this work was to explore self-awareness in the context of developing control systems for autonomous applications, concretely the ASys' model-based approach to engineering autonomy, in the line of López [90]. The ultimate objective is to move the responsibility of system's adaptivity from engineers at design time to the self-aware system at run-time. Formulating this goal as a question: (page 2.3):

> *How can we enhance control systems with self-aware capabilities so as to robustly improve their autonomy?*

This thesis argues that we can do that by incorporating a *meta*controller to the control architecture of the autonomous system, capable of *understanding* the functional state and reconfiguring its organisation at run-time if required. To do that it exploits the engineering knowledge of the system, captured in an explicit functional model of the system.

The aim of this work has been to provide a universal, yet of practical engineering applicability, solution. For that purpose, we have considered and developed elements at different levels of abstraction and engineering resolution:

Theoretical foundation. The extension and adaptation of Lopez's theoretical framework for autonomous systems came forth naturally during the analysis phase of the research, because the need for a formal basement to develop a consistent and coherent framework became clear.

Engineering postulates. The three thesis postulates extend the ASys principled approach for the building of cognitive autonomous systems. They represent a path to self-awareness through our metacontrol vision based on functional modelling.

OM Architectural Framework. To reify the previous design principles, a complete architectural framework was developed: the OM Architectural Framework. Relevant elements include:

- **Patterns for Self-awareness.** Four design patterns develop the thesis postulates and the model-based cognitive control principle into reusable design guidelines for the engineering of control systems. The Epistemic Control Loop pattern provides guidelines to develop a controller that exploits an explicit model of the plant. The Metacontrol and Functional/Structural Metacontrol patterns define the integration of a metacontroller in the control architecture of the system, and the organisation of the internal of the metacontroller, respectively. The Deep Model Reflection pattern specifies the modelling process to produce a functional model that is usable at run-time for metacontrol.

- **TOMASys.** The TOMASys metamodel was developed to define the functional self-model that an autonomous system could exploit for self-awareness. It provides a conceptualisation of the organisation in components and the functions of an autonomous system. The elements of the metamodel are based in a theoretical framework rooted in the general systems theory.

- **Reference Architecture.** The OM Architecture is an engineering blueprint for the metacontrol. It defines the model-centric processes involved in the *meta*-control loops that allow the observation of the functional state of a control system and its eventual reconfiguration.

- **Engineering methodology.** The OM Engineering Process suggests a methodology to apply the OM Architectural framework in the construction of the metacontroller for an autonomous system.

Software assets validated in a testbed. In order to demonstrate the validity of the proposed framework, a software library was developed and applied to the con-

struction of a self-aware autonomous system. The library implements an OM-based metacontroller, and interfaces to integrate it in component-based control systems.

12.1.1 Review of the Objectives of the Work

Let us review how the work developed has contributed to achieve the different objectives in which we had decomposed the purpose of this thesis:

1. **Analyse mechanisms for adaptivity** both from biological cognition centered on self-awareness, and extant technical approaches. Biologically inspired approaches, such as cognitive architectures, are more focused on high-level cognitive capabilities, and validating scientific models of intelligence, rather than structural mechanisms for robust autonomy. On the other hand, engineering approaches closer to industry, such as fault-tolerant control and autonomic-computing, are still missing to leverage the model-based vision of cognition.

   The following paper was contributed in the preliminary stages of the research, related to the analysis of intelligence as an adaptivity mechanism, in terms of the framework for autonomous systems:
   [91]

2. **Explore the relation between self-awareness and models.** The subject of modelling has been thoroughly explored, the results were presented in sections 3.3 and 3.2. The representation of *function*, in its relational sense between the organisation of a system and its directiveness, seems to play a vital role in the engineering of self-awareness in technical systems.

   The following journal article presented the results of studying the value of self-awareness/consciousness:
   [68].
   It contains some preliminary ideas for the OM Architecture, although focusing on the preliminary idea of the metacontrol in the operating system metaphor: providing the infrastructure and orchestration for the cognitive processes in an autonomous system to perform.

3. **Elaborate control design principles.** The three postulates of this thesis (section 6.2) extend the ASys model-based-cognition principled approach to engineering autonomy. They provide guidelines for implementing self-awareness in autonomous systems as a functional meta-control that exploits at runtime the engineering models of the system.

4. **Develop an architecture for self-aware control systems.** A complete Architectural framework has been developed. At its core is the OM Architecture and the TOMASys metamodel. But the framework also integrates the four design patterns for self-aware autonomous systems, the OM Engineering Process and accompanying assets.

5. **Build reusable software assets.** The OMJava library implements the OM Architecture and can be used to develop domain-independent metacontrol systems in Java. In addition, a ROS stack has been developed for the Robot Operating System [52] component framework that allows to integrate OM-based metacontrollers in ROS systems.

6. **Validate the research with a testbed.** The OMJava library has been applied to the development of a mobile robot application, where the value of the self-awareness provided by the OM metacontroller has been validated.

A paper presenting the initial version of the principles developed as reusable design patterns, and introducing the OM Architecture and the OMJava library, with its application to the mobile robot testbed, has been recently published [65].

12.1.2 The OM Architectural framework and the engineering of autonomous systems

The OM Architectural framework has much in common with the other approaches for the construction of autonomous systems that have been analysed. It has to be so, given the fact that it was inspired by them. This work could be viewed as an attempt to integrate the adequate principles and techniques from them into the ASys foundation, to develop the envisioned self-awareness for the adaptivity goal.

The TOMASys metamodel was initially conceived to allow the representation of the functionality of the system, capturing the relation between its structure and its objectives or functions, as the functional modelling approaches analysed in section 3.3 do. However, TOMASys defines models that are to be exploited by the system itself at runtime for reconfiguration, whereas the methodologies in the literature analysed are aimed at their use by engineers for design analysis, diagnosis or supervisory control.

The solution provided by the OM Architecture, with a metacontrol system exploiting a functional model conforming to the TOMASys metamodel is more similar to the fault-tolerant control approach with a supervision level including diagnosis and reconfiguration modules defined in [21, pp. 612-618]. While sharing a common pattern of decomposing the control system in a domain/control and metacontrol/supervision levels with the proposal by Blanke et al. proposal, the OM Architecture is not limited to fault-tolerance. The control architecture presented in this work addresses the achievement of the system's functionality as explicitly defined by its objectives. It does not distinguish whether the behaviour of the system diverges from them due to internal faults, external disturbances or unforeseen circumstances.

The architectural solution proposed in OM - i.e. to have a subsystem, the metacontroller, to control the correct performance of the rest of the control system- is similar to the autonomic manager in self-managed software systems. However, whereas the autonomic initiative considers a self-management of individual autonomous components, inspired in the regulatory homeostatic mechanisms in living organisms, our metacontrol approach considers a more centralised operation based on the exploitation

of explicit knowledge (models), inspired by cognitive architectures and the high-level functions of the brain.

12.1.3 Novelty and major Contributions of the Research

A major novelty of this work is its multidisciplinar character, integrating ideas and considerations from a great variety of fields, and yet arriving at a very specific solution of immediate practical applicability for the engineering of autonomous systems. This way, the thesis integrates philosophical and biological ideas about cognition and self-awareness, with industrial-oriented methods and technical standards, such as MDE or fault-tolerant techniques.

Other approaches to machine consciousness focus on the cognitive properties of the system that result from the application of a scientific model of consciousness, in comparison with natural systems. Our OM Architectural Framework is more focused on the engineering of the self-properties, as it is also the case of the work by Chella et al. [33].

We argue that, all in all, the OM Architectural Framework is ***universal*** and ***complete***:

Domain general. The work developed is rooted in the General Systems Theory to guarantee a solution applicable to any autonomous system, independently of its domain. This general character has been maintained in every further specification of the solution, from the more abstract concepts to the implementation of the platform specific models in the engineering framework.

Technology independent. The OM Architectural Framework is applicable to any control system that can be represented with a component model, no matter its particular implementation technology.

Implementation independent. The metacontrol solution has been developed in an architectural form, so that it can be applied at the design stage without constraints on the technology used for the implementation of the metacontroller.

Engineering applicability: from design principles to implementation. This work provides a *complete* solution to the building of self-aware autonomous systems because the OM Architecture Framework provides assets to guide their engineering covering all the development phases, and a methodology to follow. The basic principles proposed have been formalised into reusable design patterns, with which different control architectures can be developed. The OM Architecture is one of the possibilities, and provides a blueprint to design the control architecture of a particular system. The OMJava library is an implementation of the OM Architecture that can be applied in the final implementation of an autonomous system.

The patterns, as instantiated in the OM Architecture, were easily applicable thanks to the construction of a domain neutral implementation (OMJava). From it, the production of the ROS platform-specific model was straightforward, and only slightly

hampered by the lack of a formal Platform Definition Model for ROS. Considering strictly only the development of the testbed application, the addition of our reference architecture produced only a minor extra-effort when compared with a standard development of the control architecture for the mobile robot.

12.2 Future Work

The work developed in this thesis tried to address a huge quest covering a very wide scope, because it intended to address the engineering of autonomous systems considering all its aspects. It is for this reason that a deep and advanced solution for all of them has not been possible, and it was not the objective. Therefore, there are many ideas for future improvements or even complete research lines that have been identified.

The design patterns for self-aware systems could be further formalised, and a more complete analysis of the implications and considerations for each pattern is possible. For example, the MetaControl pattern is defined at the system level. It could be interesting to explore the possibility of a distributed metacontrol, the pattern being applied at the component level, as it is in the Autonomic Computing approach, so that each component has self-awareness and adaptivity capabilities.

An specially important ongoing work is the self-closure of the MC pattern: the application of the MetaControl pattern to the metacontrol subsystem itself, so it becomes also a subject of monitoring and reconfiguration for adaptation. This could face the problem of infinite regress, so characteristic of the self-consciousness phenomena, but could also open the door to the emergence of the Self in artificial systems.

A more advanced topic for future work is the federation of OM agents to generate societies of controllers. They could interchange or share models and collaborate, promoting mutual awareness.

Additionally, some ideas and open issues have appeared during the development of the TOMASys metamodel, which are relevant for the whole OM framework:

- Currently, the TOMASys metamodel only models basic structural aspects of a control system. It is necessary to incorporate behavioral aspects to our current metamodel so that the Metacontrol could be able to handle function-centric, dynamical issues in the domain control system.
- It would be interesting to weave the functional and ontological views in the decomposition of a system into components. As with UML components or SysML blocks, components can be considered as composed of other components interconnected. This nesting of components is not modeled in the current version of TOMASys, in which it is a `function design` the element that nests components performing roles. We could conceive a `function design` associated to a component, or rather to the `role` of the component. This can be conceived as an extension of the current TOMASys functional view. It would be a modelling of Functional Decomposition different from the present one, which consists of

the `function design` requiring an external `objective`. In this new model a Function Design would realise a role in a parent function design, so we can have the component composition of the system directly modeled as a composition of `function designs`. In the lowest/deeper level in the decomposition the role is performed by a component.

- Enhance components with generalisation, so we can have more general types of components, such as a range sensor. Describe implications with definition of roles, that we can now refer to (virtual) instances of the more general/abstract classes, so that it is possible to select an instance of the best concrete ComponentClass of that general type.
- Complete improvement of TOMASys extending with ECL concepts to account for cognitive autonomous systems, and integration with OASys. It is necessary to improve the metamodel by including the full ECL and OASys [18] concepts to further specify functionality.

Regarding the engineering process that accompanies the OM Architectural solution, The OMEP methodology should be completed to cover all the steps in the engineering process, and integrated with the OASys-based Methodology, formally defining the views of interest related to the metacontrol development. An important step towards MDA would be to fully develop the application of the Deep Model Reflection pattern, building a transformation model from UML or SysML (as engineering languages) to TOMASys. This would probably require the definition of a UML Profile with stereotypes to adapt this engineering languages to the functional representation required by our OM framework. The eventual result of this line of work would be the integration of all these tools in the ICe tooling for ASys.

12.3 Concluding remarks

The pattern-based, model-centric approach to the construction of self-aware autonomous controllers proposed by ASys can offer possibilities —both for engineering and run-time operation— that go well beyond current capabilities of intelligent autonomous robots. In this direction, the application of our OM Architecture, rooted on the four design patterns described in chapter 7, has provided the robot with deep run-time adaptivity based on a functional understanding, hence demonstrating enhanced robust autonomy through cognitive self-awareness.

The metamodelling specification of the model exploited by the metacontroller allows for its re-usability in different phases of the engineering process and by different mechanisms. The conformance of the OM Model to TOMASys allows to obtain it from the functional specification of the system. Tools can be defined according to TOMASys to produce, analyse or transform the model of the autonomous system for design activities. On the other hand, conformance to the ECL knowledge pattern offers a standard interface to exploit the model for online metacontrol. Algorithms and methods for cognitive control can be developed according to the ECL pattern and thus

be applicable also to metacontrol tasks, without needing to conform to the domain of components and functions defined by TOMASys, at the architecture level.

Every particular aspect of the work developed in this thesis may present some flaws, and surely they all can be improved, by consolidating their formalisation and improving their realisation with extant state of the art techniques. However, it is the holistic consideration of the complete theoretical and engineering frame developed that we consider provides an interesting novelty. Despite the long road ahead, we argue the work developed in this thesis is a small, yet substantial, step in the quest for a solution for the engineering of dependable autonomous systems.

Part V

Reference

Bibliography

[1] James Albus, Alexander Meystel, Anthony Barbera, Mark Del Giorno, and Robert Finkelstein. 4D/RCS: A Reference Model Architecture For Unmanned Vehicle Systems Version 2.0. Technical report, National Institute of Standards and Technology, Technology Administration U.S. Department of Commerce, 2002.

[2] James S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man and Cybernetics*, 21(3):473–509, 1991.

[3] James S. Albus and Anthony J. Barbera. Rcs: A cognitive architecture for intelligent multi-agent systems. *Annual Reviews in Control*, 29(1):87–99, 2005.

[4] Igor Aleksander and Barry Dunmall. Axioms and tests for the presence of minimal consciousness in agents. *Journal of Consciousness Studies*, 10(4-5):7–18, 2003.

[5] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[6] François Anceau. *Vers une étude objective de la conscience*. Hermes, 1999.

[7] J.R. Anderson, D. Bothell, M.D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060, 2004.

[8] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. A software platform for component based RT-system development: OpenRTM-Aist. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 87–98, 2008.

[9] Raúl Arrabales. *Evaluación y Desarrollo de la Conciencia en Sistemas Cognitivos Artificiales*. PhD thesis, Universidad Carlos III de Madrid, Leganés, February 2011.

[10] Raúl Arrabales Moreno and Araceli Sanchis de Miguel. Applying machine consciousness models in autonomous situated agents. *Pattern Recognition Letters*, 29(8):1033–1038, 2008.

[11] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. *Ontologies, Meta-Models, and the Model-Driven Paradigm*. Springer, 2006.

[12] Colin Atkinson, Matthias Gutheil, and Kilian Kiko. *On the Relationship of Ontologies and Models*, volume 96, pages 47–60. GI, 2006.

[13] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Sci. Comput. Program.*, 44(1):5–22, 2002.

[14] Bernard J. Baars. In the theatre of consciousness: Global workspace theory, a rigorous scientific theory of consciousness. *Journal of Consciousness Studies*, 4(4):292–309, 1997.

[15] Bernard J. Baars. *In the Theater of Consciousness: The Workspace of the Mind*. Oxford University Press, oct 2001.

[16] Bernard J. Baars. The conscious access hypothesis: origins and recent evidence. *Trends Cognitive Science*, 6(1):47–52, January 2002.

[17] Laurent Balmelli, David Brown, Murray Cantor, and Michael Mott. Model-driven systems development. *IBM Systems journal*, 45(3):569–585, 2006.

[18] Julita Bermejo. *OASys: Ontology for Autonomous Systems*. PhD thesis, Departamento de Automática, Universidad Politécnica de Madrid, October 2010.

[19] Julita Bermejo-Alonso, Ricardo Sanz, Manuel Rodríguez, and Carlos Hernández. An ontological framework for autonomous systems modelling. *International Journal on Advances in Intelligent Systems*, 3(3):211–225, 2010.

[20] Julita Bermejo-Alonso, Ricardo Sanz, Manuel Rodriguez, and Carlos Hernandez. Ontology-based engineering of autonomous systems. In M. Bauer, J. Lloret Mauri, and O. Dini, editors, *Proceedings of the The Sixth International Conference on Autonomic and Autonomous Systems*, volume 0, pages 47–51, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[21] Mogens Blanke, Michel Kinnaert, Jan Lunze, and Marcel Staroswiecki. *Diagnosis and Fault-Tolerant Control*. Springer-Verlag Berlin, 2006.

[22] Ned Block. *Consciousness, Function, and Representation*, volume Collected Papers, Volume 1. The MIT Press, 1 edition, may 2007.

[23] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, August 1986.

[24] R.A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[25] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[26] D. Brugali and P. Scandurra. Component-based robotic engineering (part i). *Robotics & Automation Magazine, IEEE*, 16(4):84–96, December 2009.

[27] Davide Brugali, editor. *Software Engineering for Experimental Robotics*. Springer, 2007.

[28] Davide Brugali and Katia Sycara. Frameworks and pattern languages. *ACM Computing Surveys*, March 2000.

[29] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[30] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. *A System of Patterns*. John Wiley & Sons, 1996.

[31] Arturo Bajuelos Castillo. Improving robustness in robotic navigation by using a self-reconfigurable control system. Master's thesis, FACULTAD DE INFORMÁTICA UNIVERSIDAD POLITÉCNICA DE MADRID, September 2011.

[32] David J. Chalmers. *The Conscious Mind: In Search of a Fundamental Theory*. Oxford University Press, USA, 1 edition, sep 1997.

[33] Antonio Chella, Massimo Cossentino, and Valeria Seidita. *Self-Conscious Robotic System Design Process - from Analysis to Implementation*, chapter Self-Conscious Robotic System Design Process - from Analysis to Implementation, pages 209–222. Springer, July 2010.

[34] Antonio Chella, Marcello Frixione, and Salvatore Gaglio. A cognitive architecture for robot self-consciousness. *Artificial Intelligence in Medicine*, 44(2):147–154, 2008.

[35] Luca Chittaro and Amruth N Kumar. Reasoning about function and its applications to engineering. *Artificial Intelligence in Engineering*, 12(4):331–336, 1998.

[36] John Collier. What is autonomy? *International Journal of Computing Anticipatory Systems*, 12:212–221, 2002.

[37] Roger C. Conant and W. Ross Ashby. Every good regulator of a system must be a model of that system. *International Journal of Systems Science*, 1(2):89–97, 1970.

[38] James O. Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley Professional, 1995.

[39] K. Craik. *The Nature of Explanation*. Cambridge University Press, 1943.

[40] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. Addison Wesley, Boston, 2000.

[41] Antonio R. Damasio. *The Feeling of What Happens*. Vintage, new ed edition, october 2000.

[42] Jose Luis De la Mata and Manuel Rodriguez. Abnormal situation diagnosis using d-higraphs. In *20th European Symposium on Computer Aided Process Engineering ESCAPE20*, pages 1477–1482. Elsevier B. V., 2010.

[43] Scott A. DeLoach, Walamitien H. Oyenan, and Eric T. Matson. A capabilities-based model for adaptive organizations. *Autonomous Agents and Multi-Agent Systems*, 16:13–56, February 2008.

[44] Geir E. Dullerud and Fernado Paganini. *A Course in Robust Control Theory: a Convex Approach*. Texts in Applied Mathematics. Springer, 1999.

[45] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff, and N. R. Mead. Survivable network systems: An emerging discipline. Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, 1997.

[46] J. A. Estefan. Survey of model-based system engineering (mbse) methodologies. Technical Report INCOSE-TD-2007-003-01, ModelBased Systems Engineering (MBSE) Initiative, INCOSE, 2008.

[47] José Luis Fernandez-Sánchez, Mario García-García, Jesús García-Muñoz, and José Patricio Gómez-Pérez. La ingeniería de sistemas y su aplicación a un vehículo aéreo no tripulado. *DYNA Ingeniería e Industria*, 87(4):456–466, August 2012.

[48] José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 70–85, Berlin, Heidelberg, 2010. Springer-Verlag.

[49] Dieter Fox. Adapting the sample size in particle filters through kld-sampling. *International Journal of Robotics Research*, 22:2003, 2003.

[50] S. Franklin, A. Kelemen, and L. McCauley. Ida: a cognitive agent architecture. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 3, pages 2646 –2651 vol.3, oct 1998.

[51] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[52] Willow Garage. Robot operating system, 2011.

[53] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In Rogério De Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting dependable systems*, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003.

[54] Anne-Lise Gehin, Hexuan Hu, and Mireille Bayart. A self-updating model for analysing system reconfigurability. *Engineering Applications of Artificial Intelligence*, 25(1):20 – 30, 2012.

[55] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. John Wiley & Sons, 2008.

[56] T. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[57] Erico Guizzo. How google's self-driving car works, 2011, 18 Oct.

[58] Pentti O. Haikonen. Qualia and conscious machines. *International Journal of Machine Consciousness*, 1(2):225–234, 2009.

[59] Grant Hammond. *The Mind of War: John Boyd and American Security*. Smithsonian Books, 2012.

[60] Stevan Harnad. Symbol-grounding problem, June 2003.

[61] David Hästbacka, Timo Vepsäläinen, and Seppo Kuikka. Model-driven development of industrial process control applications. *Journal of Systems and Software*, 84(7):1100–1113, July 2011.

[62] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. *Software Engineering, IEEE Transactions on*, 21(4):288–301, Apr 1995.

[63] B. Henderson-Sellers. Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software*, 84(2):301 – 313, 2011.

[64] Carlos Hernández. Adding consciousness to cognitive architectures. Master's thesis, Dpto. Automática, Ing. Electrónica e Informática Industrial, Universidad Politécnica de Madrid, March 2008.

[65] Carlos Hernández, Julita Bermejo-Alonso, Ignacio López, and Ricardo Sanz. Three patterns for autonomous robot control architecting. In Alfred Zimmermann, editor, *PATTERNS 2013, The Fifth International Conferences on Pervasive Patterns and Applications*, pages 44–51. IARIA, May 27 2013.

[66] Carlos Hernández, Adolfo Hernando, Ricardo Sanz, and Francisco Arjonilla. *Higgs Manual*. Autonomous Systems Laboratory, UPM - ETS Ingenieros Industriales José Gutierrez Abascal 2 28006 Madrid SPAIN, 2011.

[67] Carlos Hernandez, Ignacio Lopez, and Ricardo Sanz. The operative mind: a functional, computational and modeling approach to machine consciousness. *International Journal of Machine Consciousness*, 1(01):83–98, 2009.

[68] Carlos Hernández, Ricardo Sanz, and Ignacio López. Attention and consciousness in cognitive systems. In *ESF-JSPS Conference Series for Young Researchers: Cognitive Robotics*. ESF-JSPS, March 2008.

[69] Owen Holland, editor. *Machine Consciousness*. Imprint Academic, April 2003.

[70] H. Huang, E. Messina, Robert Wade, Ralph English, Brian Novak, and James Albus. Autonomy measures for robots. In *Proceedings of the 2004 ASME International Mechanical Engineering Congress & Exposition, Anaheim, California*, pages 1–7, 2004.

[71] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[72] Nicholas R. Jennings and Michael Wooldridge. Agent-oriented software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[73] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, October 1997.

[74] P.N. Johnson-Laird. *The Computer and the Mind: An Introduction to Cognitive Science*. Harvard University Press, 1988.

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

[76] George C. Klir. *An approach to General Systems Theory*. Litton Educational Publishing, Inc., 1969.

[77] Markus Klotzbücher, Nico Hochgeschwender, Luca Gherardi, Herman Bruyninckx, Gerhard Kraetzschmar, and Davide Brugali. The BRICS component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1758–1764, New York, NY, USA, 2013. ACM.

[78] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[79] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, July 2006.

[80] Thomas Kühne. Contrasting classification with generalisation. In *APCCM*, pages 71–78, 2009.

[81] Benjamin Kuo. *Automatic Control Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[82] John E. Laird. *The Soar Cognitive Architecture*. The MIT Press, May 2012.

[83] Christopher Landauer and Kirstie L. Bellman. Meta-analysis and reflection as system development strategies. In *Metainformatics. International Symposium MIS 2003*, number 3002 in LNCS, pages 178–196. Springer-Verlag, 2004.

[84] Rolf Landauer. Information is physical. In *Workshop on Physics and Computation, PhysComp '92.*, pages 1–4, 1992.

[85] Nancy G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. The MIT Press, 2012.

[86] Williams S. Levine, editor. *The Control Handbook*. CRC Press, 1996.

[87] Morten Lind. Modeling goals and functions of complex industrial plants. *Applied Artificial Intelligence: An International Journal*, 8(2):259–283, 1994.

[88] Morten Lind. The what, why and how of functional modelling. In *Proc. of the 1st International Symposium on Symbiotic Nuclear Power Systems for 21st Century (ISSNP)*, pages 174–179, 2007.

[89] Barbara Liskov. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[90] Ignacio López. *A Foundation for Perception in Autonomous Systems*. PhD thesis, Departamento de Automática, Universidad Politécnica de Madrid, May 2007.

[91] Ignacio López, Ricardo Sanz, and Carlos Hernández. Architectural factors for intelligence in autonomous systems. In Gal A. Kaminka and Catherina R. Burgart, editors, *Evaluating Architectures for Intelligence, Papers from the 2007 AAAI Workshop*, number WS-07-04, pages 48–52, Vancouver, British Columbia Canada, July 2007. AAAI, AAAI Press.

[92] Ignacio López, Ricardo Sanz, Carlos Hernández, and Adolfo Hernando. General autonomous systems: The principle of minimal structure. In Adam Grzech, editor, *Proceedings of the 16th International Conference on Systems Science*, volume 1, pages 198–203, 2007.

[93] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige. The office marathon: Robust navigation in an indoor office environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 300 –307, may 2010.

[94] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, 1955. *AI Magazine*, 27(4):12–14, 2006.

[95] M.S.P. Miller. Patterns for cognitive systems. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on*, pages 642–647, 2012.

[96] Mohammad Modarres. Functional modeling.

[97] Mohammad Modarres and Se Woo Cheon. Function-centered modeling of engineering systems using the goal tree–success tree technique and functional primitives. *Reliability Engineering & System Safety*, 64(2):181 – 200, 1999.

[98] David J. Musliner, Jeffrey M. Rye, Dan Thomsen, David D. McDonald, Mark H. Burstein, and Paul Robertson. Fuzzbuster: A system for self-adaptive immunity from cyber threats. In *Eighth International Conference on Autonomic and Autonomous Systems (ICAS-12)*, March 2012.

[99] T. Nagel. What is it like to be a bat? *The Philosophical Review*, 83(4):435–450, 1974.

[100] Pavol Návrat. Hierarchies of programming concepts: abstraction, generality, and beyond. *SIGCSE Bull.*, 26(3):17–21, September 1994.

[101] L. Northrop. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon Software Engineering Institute, 2006.

[102] OMG. Robotic technology component specification. OMG Adopted Specification formal/2008-04-04, Object Managemnt Group, 2008.

[103] OMG. Robotic technology component specification. Technical Report formal/2008-04-04, Object Management Group, April 2008.

[104] OMG. OMG unified modeling language (OMG UML), infrastructure. Technical report, OMG, February 2009.

[105] OMG. OMG Unified Modeling Language (OMG UML), Superstructure. Technical report, OMG, February 2009.

[106] OMG. Documents associated with dynamic deployment and configuration for rtc (ddc4rtc) 1.0 - beta 1. Spec. ptc/2012-08-33, OMG, August 2012.

[107] Frans P.B. Osinga. *Science, Strategy and War: The Strategic Theory of John Boyd*. Routledge, 2007.

[108] Walamitien H. Oyenan and Scott A. Deloach. Towards a systematic approach for designing autonomic systems. *Web Intelli. and Agent Sys.*, 8:79–97, January 2010.

[109] G.J. Pappas, G. Lafferriere, and S. Sastry. Hierarchically consistent control systems. *IEEE Transactions on Automatic Control*, 45(6):1144–1160, June 2000.

[110] M. Parashar and S. Hariri. Autonomic computing: An overview. *Unconventional Programming Paradigms*, pages 257–269, 2005.

[111] Alessandro Pasetti. *Software Frameworks and Embedded Control Systems*, volume 2231 of *Lecture Notes in Computer Science*. Springer, 2002.

[112] Donald Perlis and V. S. Subrahmanian. Meta-languages, reflection principles and self-reference. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 323–358. Oxford University Press, 1994.

[113] Jean-Marc Perronne, Laurent Thiry, and Bernard Thirion. Architectural concepts and design patterns for behavior modeling and integration. *Math. Comput. Simul.*, 70(5-6):314–329, February 2006.

[114] Woody Pidcock. What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?, January 2003.

[115] A.W. Pike, M.J. Grimble, A.W. Ordys M.A. Johnson, and S. Shakoor. *The Control Handbook*, chapter Predictive Control, pages 805–814. CRC Press, 1996.

[116] Michael J. Pont. *Patterns for Time-Triggered Embedded Systems*. Addison-Wesley, 2001.

[117] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press / Addison-Wesley, 1995.

[118] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[119] Ansgar Radermacher, S. Robert, C. Wigham, V. Seignole, and R. Sanz. State of the art in embedded commponent technology. IST COMPARE Project Deliverable 1.1, 2005.

[120] Laurent Rioux and Charles R. Robinson. A standards based architecture using JAUS and RTC. In *Proceedings of SiMPAR 2010 Workshop on Simulation, Modeling and Programming for Autonomous Robots*, Darmstadt (Germany), November 15-16 2010.

[121] Rosen Robert. On models and modeling. *Applied Mathematics and Computation*, 56(2-3):359–372, July 1993.

[122] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[123] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B.S. Seo, and Y.J. Cho. The PEIS-ecology project: vision and results. In *Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS)*, pages 2329–2335, Nice, France, 2008.

[124] Marcos Salom-García. Fusión sensorial en plataforma robótica móvil. Master's thesis, Escuela Técnica Superior de Ingenieros Industriales de Madrid (UPM), 2009.

[125] R. Sanz, A. Yela, and R. Chinchilla. A pattern schema for complex controllers. In *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, volume 2, pages 101–105. IEEE, Sept. 2003.

[126] R. Sanz, A. Yela, and R. Chinchilla. A pattern schema for complex controllers. In *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, volume 2, pages 101 – 105, sept. 2003.

[127] R. Sanz and J. Zalewski. Pattern-based control systems engineering. *Control Systems, IEEE*, 23(3):43 – 60, june 2003.

[128] Ricardo Sanz. *Arquitectura de Control Inteligente de Procesos*. PhD thesis, Universidad Politécnica de Madrid, 1990.

[129] Ricardo Sanz, Carlos Hernández, Jaime Gómez, and Manuel G. Bedia. Against animats. In *Proceedings of CogSys 2010 - 4th International Conference on Cognitive Systems*, Zurich, Switzerland, January 2010.

[130] Ricardo Sanz, Carlos Hernández, Adolfo Hernando, Jaime Gómez, and Julita Bermejo. Grounding robot autonomy in emotion and self-awareness. In *Advances in Robotics. FIRA RoboWorld Congress 2009, Incheon, Korea, August 16-20, 2009. Proceedings*, volume 5744/2009 of *Lecture Notes in Computer Science*, pages 23–43. Springer Berlin, 2009.

[131] Ricardo Sanz, Carlos Hernández, and M. G. Sánchez-Escribano. Consciousness, action selection, meaning and phenomenic anticipation. *International Journal of Machine Consciousness*, 04(02):383–399, 2012.

[132] Ricardo Sanz, Ignacio López, Julita Bermejo-Alonso, Rafael Chinchilla, and Raquel Conde. Self-x: The control within. In *Proceedings of IFAC World Congress 2005*, July 2005.

[133] Ricardo Sanz, Ignacio López, and Carlos Hernández. Self-awareness in real-time cognitive control architectures. In *Proc. AAAI Fall Symposium on Consciousness and Artificial Intelligence: Theoretical foundations and current approaches*, Washington DC, November 2007. AAAI, AAAI Press.

[134] Ricardo Sanz, Ignacio López, Manuel Rodríguez, and Carlos Hernández. Principles for consciousness in integrated cognitive control. *Neural Networks*, 20(9):938–946, November 2007.

[135] Ricardo Sanz, Fernando Matía, Ramón Galán, and Agustín Jiménez. Integration of fuzzy technology in complex process control systems. In *Proceedings of FLAMOC'96*, Sydney, Australia, 1996.

[136] Ricardo Sanz, Fernando Matía, and Santos Galán. Fridges, elephants and the meaning of autonomy and intelligence. In *IEEE International Symposium on Intelligent Control, ISIC'2000, Patras, Greece,*, 2000.

[137] Ricardo Sanz, Fernando Matía, and Santos Galán. Fridges, elephants and the meaning of autonomy and intelligence. In *IEEE International Symposium on Intelligent Control, ISIC'2000*, Patras, Greece, 2000.

[138] Ricardo Sanz and Manuel Rodríguez. The ASys vision. engineering Any-X autonomous systems. Technical Report R-2007-001, Universidad Politécnica de Madrid - Autonomus Systems Laboratory, 2007.

[139] Ricardo Sanz and Manuel Rodríguez. The asys vision. Technical report, Autonomous System Laboratory, February 2008.

[140] Douglas C. Schmidt. *Pattern Oriented Software Architecture: Patterns for Concurrent and Distributed Objects*. Wiley, Chichester, 2000.

[141] Miguel Segarra. *CORBA Control Systems*. PhD thesis, Escuela Técnica Superior de Ingenieros Industriales de Madrid (UPM), 2005.

[142] Azamat Shakhimardanov, Nico Hochgeschwender, and Gerhard K. Kraetzschmar. Component models in robotics software. In *Proceedings of the 10th Performance Metrics for Intelligent Systems Workshop*, PerMIS '10, pages 82–87, New York, NY, USA, 2010. ACM.

[143] M. Shanahan. A cognitive architecture that combines internal simulation with a global workspace. *Consciousness and Cognition*, 15(2):433–449, 2006.

[144] M. Shaw and D. Garlan. *Software Architecture. An Emerging Discipline*. Prentice-Hall, 1996.

[145] Wolf Singer. *Phenomenal awareness and consciousness from a neurobiological perspective*, pages 121–137. The MIT Press, 2000.

[146] Aaron Sloman and Ron Chrisley. *Machine Consciousness*, chapter Virtual Machines and Consciousness, pages 133–172. Imprint Academic, 2003.

[147] Gerd Sommerhoff. *Understanding Consciousness: Its Function and Brain Processes*. Sage Publications Ltd, 1 edition, nov 2000.

[148] Frank Svoboda. The three "r's" of mature system development: Reuse, reengineering, and architecture. In *Fifth Systems Reengineering Technology Workshop (SRTW5)*, Monterey, California, February 7-9 1995.

[149] Clemens Szypersky. *Component Software. Beyond Object-Oriented Programming*. ACM Press / Addison-Wesley, Reading, MA, 1998.

[150] John G. Taylor. Paying attention to consciousness. *TRENDS in Cognitive Sciences*, 6(5):206–210, May 2002.

[151] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[152] Andrea Valerio, Giancarlo Succi, and Massimo Fenaroli. Domain analysis and framework-based software development. *SIGAPP Appl. Comput. Rev.*, 5(2):4–15, September 1997.

[153] Paul M.J. van den Hof, Carsten Scherer, and Peter S.C. Heuberger. *Model-Based Control: Bridging Rigorous Theory and Advanced Technology*. Springer, 2009.

[154] D. Vernon, G. Metta, and G. Sandini. A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *Evolutionary Computation, IEEE Transactions on*, 11(2):151 –180, april 2007.

[155] L. Von Bertalanffy. *General System Theory: Foundations, Development, Applications*. The international library of systems theory and philosophy. Braziller, 2003 edition, 1968.

[156] ALFUS working group at NIST. Autonomy levels for unmanned systems. `http://www.nist.gov/el/isd/ks/autonomy_levels.cfm`, 2012.

[157] Janusz Zalewski. Real-time software architectures and design patterns: fundamental concepts and their consequences. *Annual Reviews in Control*, 25(0):133 – 146, 2001.

Acronyms

MDA Model-Driven Architecture. 48

MDE Model-Driven Engineering. 47

OM Operative Mind. 161

OMEP OM Engineering Process (see also OMEP in the *glossary*). 211, 212

PIM Platform-independent Model (see also platform-independent model in the *glossary*). 48

PSM Platform-specific Model (see also platform-specific model in the *glossary*). 48

TOMASys Teleological and Ontological Metamodel for Autonomous Systems (see also TOMASys in the *glossary*). 138

Glossary

A

activity — the activity of the system is the time series of values of the system quantities over a specific period of time. 70

adaptivity — adaptivity enables systems to change their own configuration and way of operating in order to compensate for disturbances and the effects of uncertainty in the environment, while preserving convergence to their objectives (adapted from [90]). 83, 223

application — a concrete realisation of a system, eps. software, to fulfil the needs of a set of specific stakeholders. 3

architecture — is the structure that identifies, defines, and organizes components, their relationships, and principles of design; the assignment of functions to subsystems and the specifications of the interfaces between subsystems. 25

- **reference architecture** — a reference software architecture is a software architecture where the structures and respective elements and relations provide templates for concrete architectures in a particular domain or in a family of software systems. 26, 114, 162
- **software architecture** — the term software architecture intuitively denotes the high level structures of a software system. It can be defined as the set of structures needed to understand the software system, defining the software elements, the relations between them, and the properties of both elements and relations. 94, 159

autonomy — the quality of a system of behaving independently while pursuing the objectives it was commanded to. 12, 13

availability — the proportion of time a system is in a functioning condition. 17, 286

B

behaviour — the behavior of the system is usually considered to be the series of values of the *observable system quantities* over a period of time. In more formal terms, the behaviour of a system is the set of all time-invariant relations of the system. 71

C

component — a modular, encapsulated part of a system, esp. reusable software implementations that can be used to play the roles specified by design patterns. 138

component model — a generic description of the system architecture, as a set of components and their interconnections. 162
configuration — the realized structural organization of the system or a subsystem, e.g. of a function version. 151
connector — a realized relation between two components. 144
consciousness — a cluster concept that refers to a collection of loosely-related and ill defined phenomena like awareness, self-awareness, introspection, attention or experience. 43
control system — a control system is a set of devices —sensor, controller, actuator— that manages, commands, directs or regulates the behavior of other device(s) or system(s). 3
controller — the part of the control system that determines the action to be generated and exerted into the plant. 5–8
coupling — a group of quantities that are shared by two or more elements of a system, or by the system and its environment (adapted from [76]). 71

D

dependability — in systems engineering, dependability is a measure of a system's availability, reliability, and its maintainability. 17, 18, 288
directiveness — (of an autonomous system) quality —usually behavioural— of the system, derived from a particular organisation, to behave in a convergent evolution towards its objectives (adapted from [90]). 74
  purposive directiveness — reconfiguration of parts of the organization of the system through processes which operate with an explicit representation of the objective of the system (adapted from [90]). 74
  structural directiveness — refers to the intentional behaviour of the system which derives from a particular organization or structure (adapted from [90]). 74

E

element — any of the entities listed in a *universe of discourse and couplings* model of a system. 71
environment — the rest of the universe which is not the system object of interest. 4, 70
error — the part of the system in wrong state which potentially leads to a failure. 62

F

failure — a deviation of the system behavior from the specification. 62
fault — the cause of an error, something that changes the behaviour of a system such that the system no longer satisfies its requirements (adapted from [21]). 62
function — a conceptualisation of how the system's directiveness is implemented in the system's organisation. Roles the designer intended a subsystem should have in the achievement of the goals of the system of which it is a part. The relation between the purpose of the component and the behaviour rendered by its structure.
  abstract function — a function as solely defined by the objective i.e. independent of the algorithms used or the components configurations. 79, 151

function definition — a conceptual entity that represents a complete specification of a functional decomposition in the current scenario of the running system. 78
function design — a triplet <objective, algorithm, configuration> that specifies a function. 79

G

General Systems Theory — the interdisciplinary study of all kinds of systems, with the goal of elucidating principles that can be applied to all types of them at all nesting levels in all fields of research. It may also refer specifically to a concrete approach as Klir's or Bertalanffy's. 70
goal — the outcome or objective toward which certain activities of a system or its parts are directed. 53

M

maintainability — refers to the ease with which the system may undergo repairs and changes during its life-time evolution. 17, 286
metacontrol — the control of the function of a control system. 29, 111
metacontroller — a controller that controls the function of another controller. 112, 162, 259
mission — (of a system) the functionality demanded from a system, which is technically specified into a series of requirements, plus the constraints it must satisfy. 4
model — a representation of a system. 48
meta-model — a representation of the entities used in creating a model of a system. 48

O

objective — a desired state of the system, of the environment, or both. 75
objectives hierarchy — a organised structure of objectives. A dependency graph, usually a tree, between the different objectives of a system. 141
root objective — a root or generative objective is an objective that does not contribute to realise any other objectives. In an autonomous system, they are typically the objectives with the longest scope and higher abstraction (adopted from [90]). 76
OM Architecture — reference architecture for the design of the metacontrol subsystem in an OM-based autonomous system. 115, 135, 162, 163, 166, 167, 201, 206, 211, 212, 216, 217, 219, 246, 260, 265
OM Architecture Framework — architectural framework for the design of autonomous systems with self-awareness and adaptivity capabilities. 115, 211, 216, 219, 244, 265, 287, 289
OMEP — the OM Engineering Process is a methodology to apply the OM Architectural Framework to the construction of autonomous systems (see also OM Architecture Framework in the *glossary*). 211, 212, 219, 220, 283
organisation — the set of all the properties —structural, relational— of a system (adapted from [76]). 71

P

performance — the effectiveness of the temporary behaviour of the system. Directly related to requirements (adapted from [90]). 3, 18, 64, 83, 156, 207
platform — the computing infrastructure that supports the deployment and execution of a control system. 48
  platform-independent model — a model of a software system or business system, that is independent of the specific technological platform used to implement it. 283
  platform-specific model — a model of a software system or business system, that is tailored to the specific technological platform used to implement it. 283
program — the set of system properties of local scope. 73
property — an attribute of a system. 71

Q

quantity — an observed attribute of a system (adopted from [76]). 70

R

reconfiguration — a change in the organisation of a system. 29, 31, 162
reliability — is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. 17, 286
requirement — a statement that identifies a necessary attribute, capability, characteristic, or quality of a system for it to have value and utility to a particular stakeholder. 10, 17, 24, 28, 47, 213, 222–224
robustness — the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions (from the IEEE Standard Glossary of Software Engineering). 3, 16, 64, 74, 223
role — a set of connected behaviours, rights, obligations, beliefs, and norms imposed to a component as conceptualised by patterns. 54

S

safety — refers to the personal harm and equipment damage that can arise due to a system failure. 17
scalability — is the ability of a system to be enlarged, i.e. by incorporating new components, to accommodate growth either quantitatively or qualitatively in performance or in the functionality serviced. 17
scenario — a scenario is a synoptical description of an event, situation or series of actions and events that is relevant for the behavioural analysis of a system. 232, 238
space-time resolution level — the instants of time and the locations in space where quantities are observed. 70
state-transition structure — a form of system representation that is based on the identification of system states and transitions between them. 71
structure — the system's physical parts and their interconnections in the physical topology. 73
  hypothetic structure — the part of the system structure responsible for the generation of the relatively permanent behaviour. 73
  real structure — the part of the system structure responsible for the generation of the permanent behaviour. 73

survivability — it is the aspect of system dependability that focuses on preserving system core services, even when systems are faulty or compromised. 17
system — part of all the possible entities from the universe, which is isolated from the rest for its analysis (adapted from [76]). 4, 70
  subsystem — in general systems theory, a subsystem or *element* is part of a system that is analysed separated from the rest as if it were a system of its own. 71

T

time–invariant relations — relations between the quantities of a system that do not change. 71
TOMASys — the Teleological and Ontological Metamodel for Autonomous Systems is a general model for representing an autonomous system's designed functions and structure (see also OM Architecture Framework in the *glossary*). 139, 141, 156, 158, 159, 162, 175, 211, 216, 217, 219, 247, 283
trait — features used by the observer in the selection of a system. 70

U

uncertainty — epistemic possibility of deviation from expectations. 8, 9
  intensive uncertainty — uncertainty in the values of the magnitudes of the system that is observed and/or modelled. 8, 14
  qualitative uncertainty — refers to uncertainties that cannot be quantified. Occurrence of unexpected events that qualitatively change the behaviour of the system (adapted from [90]). 8, 14
universe — the system plus its environment. The set of all entities of relevance for a systems problem. 70
  universe of discourse and couplings — a form of system representation that is based on the identification of *elements* and *relations* between them. 71

Mobile robot testbed additional figures

12.4 TOMASys model of the complete mobile robot testbed

Following the TOMASys model for the complete testbed system is presented in UML-like diagrams, including all the high-level functions in the system.

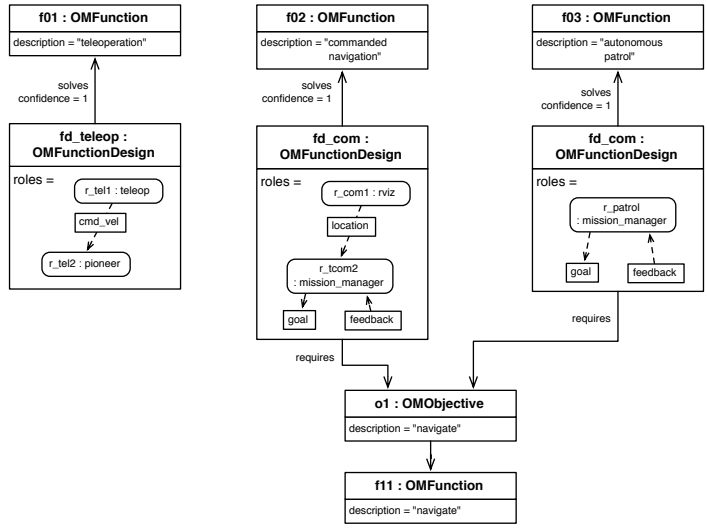


Figure 12.1: TOMASys model of all the high level functions in the patrolling robot. The localisation and autonomous navigation functions are shown in figures 12.2 and 12.4.

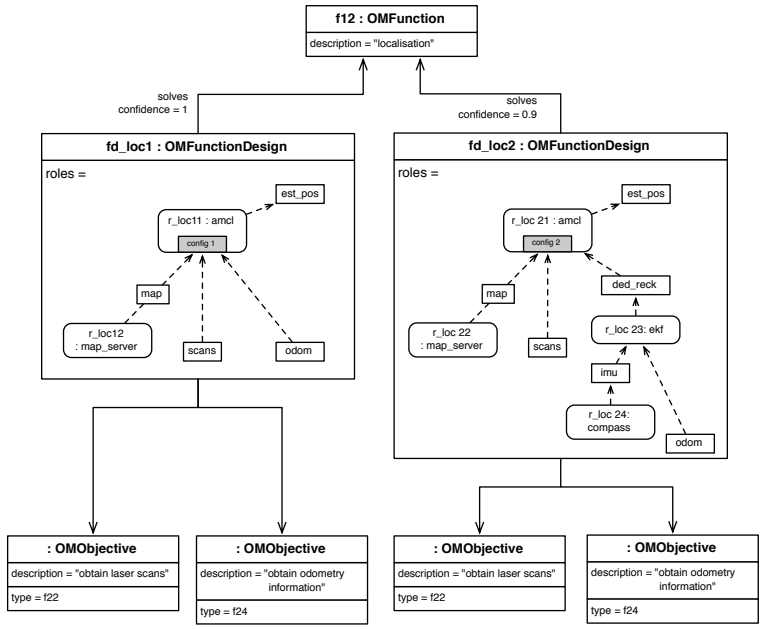


Figure 12.2: TOMASys model of the alternative designs for localisation.

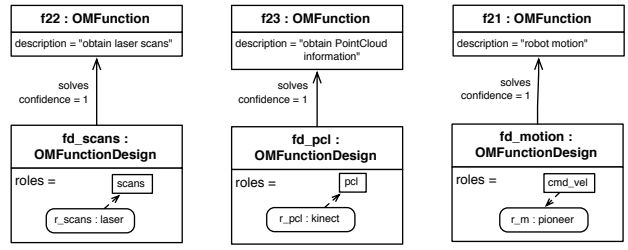


Figure 12.3: TOMASys model of the low level functions in the mobile robot.

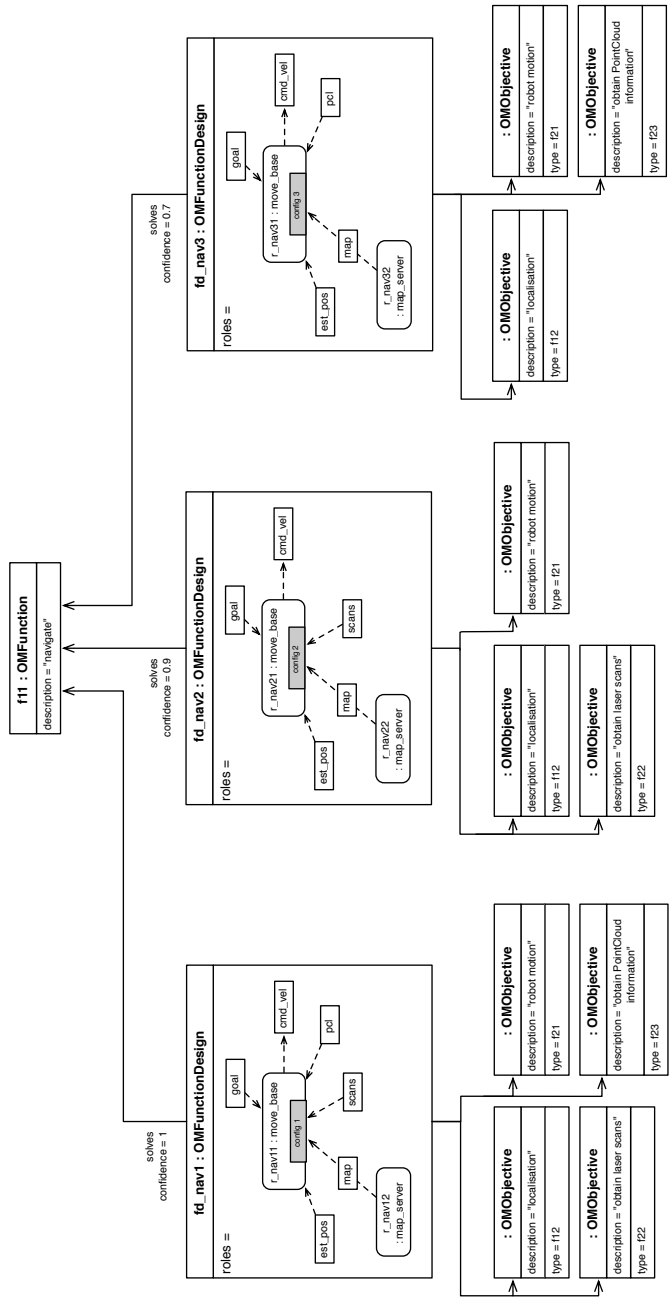


Figure 12.4: TOMASys model of the alternative designs for navigation.