

# Design of (Robotics) Software Standards

Herman Bruyninckx  
Dept Mechanical Engineering  
K.U.Leuven, Belgium

Klas Nilsson  
Dept Computer Science  
Lund Institute of Technology, Sweden

(DRAFT Version 0.5, November 12th 2007)

## Abstract

This paper presents a strategy to design standards for *software libraries* in robotic systems (or rather, in large-scale engineering systems in general). The strategy involves the separation between *objects* (algorithmic functionality and single-activity behaviour) and *components* (multi-activity interaction architecture and behaviour), and the introduction of *four complementary levels* in software standardization, in order to develop standards that are *consistent* and *stable* enough to survive the never-ending evolution in the domain of robotics. The major motivation behind the standards-making strategy is *decoupling*: (i) to divide the domain to be standardised into a family of complementary sub-standards (either hierarchically ordered, or unconnected) that are each *as small as naturally possible*, and that can be combined in various ways according to the needs of specific application domains; and (ii) to separate the abstract programming interface of each library from its concrete implementation(s).

These standardization suggestions could inspire many (Open Source) robotics software projects, enhancing their interoperability.

## 1 Introduction

This text focuses on the *standardization* of software *libraries* for large-scale systems in robotics, machine tools and automation. Such large-scale systems need to close (realtime) feedback loops over multiple pieces of sensing and actuation hardware, to integrate software from different sources, to update parts of the software over the lifetime of the systems, and even to adapt the systems' architecture to evolutions in hardware and software as well as in the systems' goals. In addition, one will build various different systems in which one wants to reuse the maximum amount of software libraries. Such complex, evolving systems are best designed by using *stable* sub-systems [14]. In a narrow technical context, "stable" means: (ideally) able to be used in any possible system irrespective of the constraints imposed in that system, and consisting itself of only stable sub-systems. In a broader social context, "stable" also means: having a programming interface that is perceived by a large community of system builders to be "done right", such that they feel comfortable reusing the libraries over and over again. In a standards-making context, "stable" means: having a programming interface that remains the same "for ever", because it has been "done right". This text explains how to design standards for software interfaces in a systematic way, taking both the technical and the social aspects into account.

Standards making is often viewed as a boring activity of management committees full of people that are way past their creative years. However, even young innovative software projects should know what are the requirements for—and the benefits of—good technical standards, in order to give their software the highest possible chances for adoption by others. Indeed, robotic system architects are looking for the following features in software libraries they can integrate in their design:

1. **Consistency**: the library should offer one coherent set of functionalities that reflect properties of objects in the real world. It should not mix objects and functionalities that could be *decoupled* without loosing consistency.

2. **Abstractness:** a software library should strictly decouple its API (Application Programming Interface) from its implementation, such that a system architect can replace one implementation by a more appropriate one without changing the rest of the system. The API is what has to be standardized, not the implementations.
3. **Separation between mechanism and policy:** a library should represent the properties of “objects” in the real world, and *independently* of how the objects’ properties are used in a particular application. Taking this principle into account when designing libraries will increase the chances to reuse this same library in other contexts, i.e., with other usage policies of the same mechanism infrastructure of the library.

This first two features are supported rather well in modern *object-oriented* software design and the corresponding high-level programming languages, such as *Java*. (Which does not mean that they are achieved without efforts from the library designers!) The last feature, however, is less well understood, because it depends so much on the application domain. Many existing libraries (*and* existing standards such as even UML) violate this principle because (i) the separation between mechanism and policy is just very difficult to realise in practice, and (ii) these libraries and standards are developed in a context that gives too much focus to one particular application. The result is that these libraries are poor candidates for standardization. For example, it is better to have a separate standard for the kinematic and dynamic properties of *rigid bodies*, instead of making it part of the standardization of the kinematic and dynamic properties of *mechanical chains*, which is only one particular way of using rigid bodies. (Satellite control would be another application domain for the same rigid body dynamics library.) Similarly, it is better to have a separate standard library for an (application-independent) PID controller, instead of for only a *robot* control standard. Hence, this text emphasizes this separation between mechanism and policy in the domain of robotics and automation, and focuses only on the lower levels, close to the hardware. Of course, there is a need for standards at the *systems* level too, but these standards are much more involved, and much less generically useful than the “small scale” object-level standards that they rely on (and that this text is dealing with).

**Objects and components.** This text makes the following distinction between *objects* and *components*:

- *Objects* provide *data processing* functionalities on *data structures* that are all executed inside of one single “activity” or “agent”. Typical functionalities are: calculation of continuous input/output behaviour, configuration of the parameters of the behaviour, reading the “state” of the object, making the state of an object “persistent” (i.e., how can it be read from, or written to, file or database). The *Finite State Machine* is the appropriate “decision making” concept to encode the changes in object behaviours.
- *Components* provide *multi-activity services* (that is, offering the functionalities of the objects that “run” inside the component to various “clients”, and executing these object functionalities in one or more processes on the operating system) and *interprocess communication* (IPC, i.e., the interconnection “architecture” between those processes as well as the synchronization and data exchange behaviour used on these interconnections). Components typically have (discrete) *service* behaviour: a client sends a request to the component to perform a certain service; several clients can send requests “at the same time”, without having to know about each other, and without specifying how the component has to process the service. In general, component services require dynamic reconfigurability, deployment and discovery: that is, in the overall system of clients and service components, new clients or services can appear or old one can disappear, without having to stop the system in order to make the clients aware of the new service situation. Decision making at the service component level is typically of the *work flow* type: when to execute activities and how to synchronize them, with Petri nets or (Harel) statecharts as the appropriate multi-activity representing mathematical concept. Component “states” are complementary to those of objects: they store the information about how far a component has progressed in the servicing of each of its clients, and what services it still has to deliver.

Objects and components provide different, complementary semantic primitives to construct complex software systems, so the standardization of objects and components should take these inherently different aspects into account, and mixing them leads to (technically and socially) “unstable” standards. In addition, the system architects’ flexibility in making components should not be hampered by libraries that offer too many and too loosely coupled objects. For example, a designer of a specific mobile robot software system will want to use the rigid body and PID control libraries, but seldom the kinematical chain and robot control libraries (if this latter was developed in the context of serial arm manipulators). Indeed, having a *robot* object in a software library is a big indication that this library will not be useful outside of the application scope foreseen by the users of the specific robot(s) the library was developed for.

**Relevant standards.** To this day, few application-level software standards exist already in the domains of robotics and automation. Mostly the lower software levels, on top of the bare hardware, have seen standardization, primarily via (open and proprietary) standards for field busses (CAN, Profibus, etc.) and computer interfaces (PCI, USB, etc.). A notable exception is the standardization of programming languages for *Programmable Logic Controllers*, IEC 61131, also called “function blocks”. On the other hand, the ISO and other standardization bodies have developed standards for non-software aspects in robotics, automation and manufacturing: for example, standards for terminology (ISO 8373), definition of reference frames (ISO 9787), static and dynamic motion performance (ANSI/RIA R15.05-1-1990, ISO 9283), safety (ISO 10218), etc. (The RoSta (Robot Standards) website has a list of standards in robotics, [4]).

The computer science level “below” robotics and automation has many more standards, some of which have succeeded in becoming widely accepted in practice and education. A set of interconnected standards which are very relevant for all engineering systems are:

- **UML** (Unified Modeling Language). This supports the generic, language-independent description of object and component properties, with integrated support for state machines and statecharts.
- **SysML** (System Modeling Language). This standard extends UML to engineering sciences, in that it offers more specific primitives such as resource constraints.
- **MARTE** (Modeling Analysis of Real-time and Embedded Systems) is another UML extension, specifically for real-time and embedded systems.

In the context of this paper, these standards are not directly useful in themselves, but will be used to develop new robotics-specific standards, that is to define objects and components that can be used directly in robotics and automation systems (and in the general purpose tools to develop such systems, e.g., Eclipse).

**Software patterns.** Standardization of the programming interfaces of software libraries is important to reuse middleware, but it is in itself not sufficient: there must be *implementations* available (because most system builders will not be interested in writing their own implementations), and these implementations should be of the highest possible quality. The latter requirement does not only relate to the computer science aspects of the implementation, but also to the extent in which they support domain-specific *patterns*. A (software) pattern is a generic solution to a problem that is important in the practice of the domain and for which so much experience has been gained that the optimal solution can be defined, as a trade off between several “forces” that pull the solution in different directions. By definition, these patterns will underly most of the libraries of the large-scale systems, so “standardizing” these patterns is a very relevant complement to the API standardization.

Often, such patterns are called “architectures” in robotics and automation, and they are special cases of more generic patterns for whose mechanisms they have (implicitly) filled in many policies that are relevant in the specific domain.

## 2 Requirements for object standards

(Software) object standards are *agreements about how to represent information*, that is, the meaning of, both, the *data* and the *algorithms* that process these data. For example, the displacement of a robot with respect to a world reference (“data”), and the composition of two such displacements (“algorithm”). A crucial point where standards tend to fail is that they impose—mostly implicitly—too many *implementation decisions* onto the users, because the standard makers ignored the variety in the use cases in the domain; for example:

- *Resolution and range*: a standardization of *position* in robotics should allow users to represent positions from nanometers (robotic manipulation of molecules) up to millions of kilometers (satellites).
- *Type variation*: the same kind of motion operations should be implemented for various data representations (complex, float, double, . . .), and for various motion spaces (2D, 3D, . . .).
- *Quality of service*: users can be interested in having the results of computations available in hard realtime (e.g., for motion control of machine tools and robots), or with a specified numerical accuracy (e.g., not better than the accuracy that the robot can achieve), or from a “stop anytime” algorithm (i.e., the implementation can give a *feasible* solution at any time the user wants it).

In most cases, no single information representation, and no single implementation, can achieve all these requirements. Hence, standardization makers should adapt to this reality, and prepare “pyramidal families” of complementary standards that share as many as possible of the four generic levels discussed below.

### 2.1 Generic representation levels

One of the two core contributions of this text (the other one is presented in Sec. 3.1) is that any standardization effort should (but most often does not!) define a “family” of the following four generic levels of representations:

1. *Ontology representation*, defining *in words* the terminology and the meaning (“semantics”) of objects that are relevant in the scope of the standard, and of the natural operations on those objects. This part of the standard serves the *human*.
2. *Mathematical representation*, providing formal data structures (“coordinate” representations) for the above-mentioned objects, as well as the API (Application Programming Interface) for the natural operations on the physical objects. This part of the standard is a *symbolical* translation of the ontological representation, preferably (but not necessarily) in *computer-understandable* form.
3. *Computer representation*, defining how coordinates and algorithms are represented in computer-readable form. This part of the standard is the computer-understandable representation as implemented in a *computer programming language*.
4. *Native hardware representation*, defining the mapping from computer-readable variables into bit representations in the *computer hardware*.

The mappings between the different levels of this family are “pyramidal”: for each level, there exist various choices for standardizing the level below. *If* the computer-understandable forms of the ontological and mathematical representations are available, implementations of two or more standards at the same level could co-exist in a large-scale system, since the code to “glue” two implementations together could, in principle, be automatically generated.

Three of the four levels (*ontological*, *mathematical* and *native hardware* representations) are most often not formalised into a standard, and, at best, exist only implicitly in the domain. However, the trend in modern robotic systems is towards larger-scale and multi-platform systems supporting a variety of human users, so the lack of these levels introduces more and more semantic and implementation ambiguities, and is a sign of the poor *state of the practice* in software middleware for robotics and automation.

The following sections explain and motivate the above-mentioned four levels with somewhat more detail.

### 2.1.1 Ontology representation

In order to represent a certain domain, the *meaning* (“semantics”) of all concepts, objects and the operations on those objects should be made unambiguously clear.<sup>1</sup> Most standards cover this level only *implicitly*, via some form of textual documentation, and seldom use *explicit* computer support—for example, via “semantic web” standards such as OWL, [17], and tools such as Protégé, [3]—to formalize the ontology representation into computer-readable form. Hence, opportunities are missed (i) to make the semantics unambiguous, and (ii) to bridge automatically the gap between (the implementations of) two or more standards that have to be used in the same large-scale software system. For example, manual intervention is currently still needed in order to use, in a robot control application, a library for robot kinematics or a physics engine that were developed in the computer animation domain, because the terminology is so different: a “kinematic chain” becomes an “armature”, a “via point” becomes a “key frame”, etc. Standards can solve this problem by ontological *domain wrappers*: all object names have various *aliases* in different domains, because domain-dependent names are more effective in conveying the meaning of the classes to the practitioners in those domains.

### 2.1.2 Mathematical representation

Engineering systems must represent the physical concepts in the system in a mathematical form. Typically, multiple mathematical representations are possible for the same physical concept. For example, the position and orientation of a robot can be represented mathematically by a *homogeneous transformation matrix*, or by the combination of a *position vector* and a set of *Euler angles*. The practical problem is that even these “standardised” representations are less standardised than they appear: Which one in the set of twelve possible Euler angles is used? Is the homogeneous transformation matrix representing a right-handed or left-handed reference frame? What *physical units* are used to represent the various concepts. Meters or millimeters or inches? Radians or degrees? Seconds or hours? Etc. Therefore, a standard needs *extra ontological information* to represent the chosen convention(s).

A standard should also separate the following aspects of the object properties in complementary *sub-standards*, in order to allow users to rely on the smallest possible consistent set of object properties:

- *Object state representations* of the objects, that allow information about a robot system to be represented, saved in a file, or transmitted between different control systems.
- *Functional operations* on the objects, such as the transformations between different coordinate representations.
- *Support operations* for the objects, such as support for the software configuration of the objects, or the reading from, or writing to, persistent storage (file, XML protocol, ...); ...

An example of the advantage of having these sub-standards is that the functionality of storing and transmitting object states is often needed in itself, without having to know about the operations on the objects, and without having to specify which medium is used for the storage or the transmission.

### 2.1.3 Computer representation

Even after having selected (and semantically documented!) any particular mathematical representation, the problems for the standards makers are not yet over. They now have to choose how the mathematical primitives are to be encoded in computer languages:

- Is a position coordinate represented by a “float”, a “double” or a “signed integer”?
- Does the standard need both a “high resolution timer” representation for fast realtime control purposes, or is the “system clock resolution” offered by the operating system sufficient?
- Is a matrix to be stored in “column major” or in “row major” format?

---

<sup>1</sup>The *symbol grounding problem* [9, 16] is neglected in this document.

- Are arguments in function calls accessed “by value” or “by reference”?
- ...

A standard that neglects these computer representation aspects has a high risk of causing *very subtle* incompatibilities between various implementations, especially on differing computer languages, compilers, or middlewares. A systematic solution to this problem is to define only *abstract interfaces* (available under different forms in Java, CORBA and C++) as standardized mathematical representations, and to provide *explicitly* different (documented and cross-referenced!) *implementations* to cover the different computer representation cases. Each implementation can have its own *extra properties* (that are not standardized), with which the users can tune or configure the implementation to their application needs; for example, the value of “zero” in a numerical algorithms such as a pseudo-inverse based inverse kinematics.

#### 2.1.4 Native hardware representation

Standard makers must also take into account the variations in hardware platforms to the extent that they can influence the practical value of a standard:

- Are the meanings and implementation of, for example, “float” and “double” the same on all possible architectures?
- Can “enumeration types” flow over when porting an implementation from a 32 bit CPU to a deeply embedded application using 16 or 8 bit CPUs?
- Is there an influence of the byte order of the hardware architecture (“big endian” or “little endian”)?
- ...

The errors generated by these aspects are even more subtle than in the case of computer representation. They are certainly to be taken into account when the standardization process involves the definition of *binary* file or message formats for storing and exchanging information between different components in a large-scale software system, which is an extra motivation for the subdivision in sub-standards of Sec. 2.1.2. Two examples of mature binary file formats are netCDF [11] and HDF5 [1].

### 3 Requirements for component standards

The previous Section discussed the aspects of *software objects* (i.e., data structures and their data processing algorithms) to be taken into account for standardization. This Section does the same for *software components*, i.e., *activities*<sup>2</sup> and their *Inter-Process Communication* (IPC). In robotics and automation, the state of the art for software components is less mature than that for software objects, hence it is more difficult to reach consensus about standards. The good news, however, is that the lower-level aspects of *computer representation* and *native hardware representation* are most often already taken care of by *middleware*, *frameworks* or *operating systems*<sup>3</sup> on top of which robotics and automation components are typically built. At those lower levels, some relevant standards exist already, such as CORBA [12], and several open source projects provide quite usable (but incompatible) implementations, e.g., [2, 6, 13]. In addition, some major *mathematical representations* that are relevant in the component standardization context are quite well developed, (i.e., they have well-defined semantics, mathematically verifiable implementations, and design and code generation tools):

- *Finite State Machines* (FSM) and their cousins “Hybrid Event Systems”, for the *sequencing* of activities *within* components. Since components can be used hierarchically, a FSM in one component could control the sequencing of the activities *between* components in its interior.

<sup>2</sup>The terminology “activity” is used as a synonym for threads, tasks, processes,...

<sup>3</sup>No semantic distinction is made between these three terms.

- *Concurrent Sequential Processes* (CSP) [5, 10], *Petri Nets*, and *Harel Statecharts* [7, 8] (integrated into the UML standard) for the *(a)synchronous interaction* (IPC, both *data flow* and *synchronization*) of activities *between* components.

However, the “translation” from mathematical to computer representation of components still has a lot of practical challenges, the major ones being (i) the inherently *asynchronous* nature of the activities executing in concurrently running components, and (ii) the various *policies* with which IPC can be implemented and which are often not specified explicitly. More concretely:

1. IPC between activities takes finite time, and can happen at any moment in the execution of an activity. So, activity programmers must make their code robust against these unpredictabilities.
2. Many programming primitives offered by operating systems to support asynchronous activities can lead to execution inefficiencies, including deadlock. In addition, the efficiency of most of these primitives doesn’t scale well with the number of activities in the system.
3. When several of the IPC events take place “at the same time,” implementations most often make implicit choices about how to order the servicing of these events. So, an implementation that relies on that specific order has poor (technical) stability.

Examples of such practical problems for FSM and CSP implementations in a robotics and automation context are: state transitions do not take zero time in software, so the controller of the system will inevitably be in “no state” during parts of its operational period; the dynamic behaviour of the system in two different component states can be different and smooth transitions between these dynamics must be guaranteed, especially if state transitions happen “instantaneously”; simple CSP implementations use blocking *rendez-vous*, which are easier to implement and to formally verify, but which can reduce efficiency, even all the way to deadlocks; CSP loses much of its formal verifiability when it requires “buffering” of data while guaranteeing data integrity. These problems, however, are situated in the *implementations* (computer representation level, not the mathematical representation level), and, at that level, *message passing* and *event handling* (“interrupts”, “callbacks”) are two major supporting technologies with various mature (but again incompatible) implementations.

The rest of this text is limited to making a number of *ontological representation* suggestions, which are suited for standardization of components, because their semantics can be defined unambiguously, irrespectively of the above-mentioned problems that their implementations can suffer from. The core idea is to specify *the minimal set* of really needed programming primitives, such that:

- Designers of the standard need minimal effort to specify the precise semantics of the primitives. And therefore, the standard has higher chances to be accepted by many software projects.
- Implementors using the standard need less effort to learn the standard.
- Tool developers have less primitives to support, hence increasing the quality, and the code generation and correctness verification support offered in their tools.

### 3.1 Component programming semantics

Besides the four generic levels of object representations (Sec. 2.1), the second major contribution claimed by this paper is the definition of a *minimum set of programming primitives* that components need to have in the context of large-scale software systems in robotics and automation. It is also believed that this set of primitives is *sufficient* for all component use cases. If a wide community consensus can be reached around the following four primitives, they are a good candidate for standardization:

- *Method Calls*. This is the *synchronous* way for one component “to execute an object in another component”. “Synchronous” means two things: (i) in the head of the programmer, the method call happens *now* and *takes no time*; and (ii) from an operating system’s perspective, the process from which the call is made can be blocked (i.e., removed by the scheduler from the queue of running processes) until the execution of the Method Call is finished.

- *Events*. This is an *asynchronous* way for one component “to execute an object in another component”. “Asynchronous” means two things: (i) in the head of the programmer, the event is “launched” *now* and it will do “something” somewhere else, but the programmer knows that his event call might not have effect immediately and that he does not have to wait for its completion; and (ii) from an operating system’s perspective, the process from which the call is made need not be blocked and can continue its execution.

Whether or not events are allowed to carry “data”, or whether they are queued or not, are implementation details and policies.

- *Commands*. This is also an *asynchronous* call from one component to another component asking the latter to “to execute an object in its activity”. Commands differ from Events in that the executor of the Command must be ready to give the caller of the Command information about the “status” of the execution. This can semantically happen in two ways: (i) the caller can make a Method Call to a “status object” in the callee to “poll” the execution status; and/or (ii) the callee fires one or more Events that the caller can react to and from which it can deduce the execution status.
- *Data Flows*. Two or more components can interchange data in “continuous”, “streaming” mode, where each of the components can read or write data at any moment during its activity, without being blocked by the other component.

These semantics specify the *mechanism* of component interaction; each of them can have various *policies* (see Sec. 3.2). The consequences for standards designers are:

- The API of each component-level standard has calls that belong to one of the four semantic categories. And it would be strange not to have all four semantic categories present in the standard.
- Different standards can reuse the same above-mentioned semantic description of the four categories, leading to improved consistency and clarity. In other words, standard designers should first agree on standardizing the four semantic categories, at least on the ontological representation level, and, preferably, also on the mathematical representation level (e.g., what kind of FSM and CSP models to use).
- Standards should clearly separate the specification of policies (“configuration”) from the specification of mechanism.

## 3.2 Component policies

The previous section presents *mechanisms* for components, but each real application will also have to make a number of *policy* choices. This section suggests a small set of such policies whose representation *could* be standardized.

- *Data Flow buffering*: only keep the latest value written to the Data Flow; *First-In, First-Out* FIFO; *Last-In, First-Out* (LIFO); lockfree (i.e., without needing mutual exclusion locks [15]).
- Internal selection and scheduling of activities in a component: a FSM is used to select the behaviour of the component (i.e., how should the component react to its Method Calls, Events and Commands). This paper suggests the following minimal set of states for *all components*:
  - *Creating*. The component is busy with being created by the operating system, and does not accept any IPC but takes care of its own initialisation.
  - *Deleting*. The component is busy with being removed by the operating system, and does not accept any IPC but takes care of its own safe and deterministic deletion.
  - *Configuring*. The component has been initialised, and reacts only to “configuration IPC”.
  - *Running*. The component has been initialised and configured, and accepts all IPC.

- *Exception*. The component has reached a situation in which it cannot proceed with its nominal activity, and has brought itself to a “safe state” in which it reacts only to “error IPC”.
- The *priorities* with which different Method Calls or Events are handled, or with which activities are scheduled by the operating system.

## 4 Conclusions

This paper presents a systematic approach to the standardization of software objects and software components, in the context of large-scale robotics and automation systems. The major suggestions are:

- Make standards as small as naturally possible.
- Provide four levels of complementary standards: ontology, mathematical representation, computer representation, and native hardware representation.
- Separate abstract interfaces from concrete implementations.
- Separate object standards from component standards.
- Separate the specification of mechanism from the specification of policies.
- Use existing *standards-making standards*: XML, UML, SysML, etc.
- Identify the relevant software patterns and standardize their meaning and the domain-specific policies.

## References

- [1] Hierarchical Data Format 5. <http://hdf.ncsa.uiuc.edu/HDF5/>.
- [2] Miro robotics middleware. <http://www.informatik.uni-ulm.de/neuro/index.php?id=321>.
- [3] Protégé. <http://protege.stanford.edu>.
- [4] Robotics standards. <http://www.robot-standards.org>.
- [5] P. Brinch Hansen. Concurrent programming concepts. *ACM Computing Surveys*, 5(4):223–245, 1973.
- [6] A. Brooks, W. Kadous, T. Kaupp, A. Makarenko, and A. Oreback. Orca: Components for robotics. <http://orca-robotics.sourceforge.net/>.
- [7] D. Harel. State charts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [8] D. Harel, H. Lanchover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive system. *Trans. on Software Engineering*, 16:403–414, 1990.
- [9] S. Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.
- [10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [11] network Common Data Form. <http://www.unidata.ucar.edu/software/netcdf/>.
- [12] Open Management Group. CORBA: Common Object Request Broker Architecture. <http://www.corba.org/>.

- [13] D. C. Schmidt. ACE, The Adaptive Communication Environment. <http://www.cs.wustl.edu/~schmidt/ACE.html>.
- [14] H. Simon. *The Sciences of the Artificial*. MIT Press, 1969.
- [15] P. Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Dept. Mech. Eng., Katholieke Univ. Leuven, Belgium, May 2006.
- [16] M. Taddeo and L. Floridi. Solving the symbol grounding problem: a critical review of fifteen years of research. *Journal of Experimental and Theoretical Artificial Intelligence*, 17(4):419–445, 2005.
- [17] W3C. Owl. <http://www.w3.org/TR/owl-ref/>.