Robotics software framework harmonization by means of component composability benchmarks. "The manifolds of four"

Herman Bruyninckx
Katholieke Universiteit Leuven, Belgium

30 September, 2010

Abstract

Robotics has passed a milestone tipping point: after 50 years of mostly *single-sourced* system development, current (professional and domestic) robots are built from hardware and software components provided by *multiple vendors*. This evolution has been triggered by necessity (because a single organisation can no longer design and develop all the required functionality and infrastructure on its own), and confronts the robotics community with new design and integration challenges, driven by the need to facilitate the composition of large robot software systems out of components provided by independent suppliers. Since the robotics domain is not yet mature enough to solve this problem by *standardisation*, a strong case can be made to motivate developers for the second-best approach, i.e., to *harmonize* the "interfaces" of their components. That is, to let them use the same component design primitives, code packaging granularities, interaction decoupling policies, coordination semantics, etc., without requiring them to use exactly the same data representations and programming interfaces. The more harmonized the component designs of different providers, the easier the *reuse* and *integration* of their components. This paper contributes to that harmonization goal, by (i) a set of complementary benchmark criteria (each with *four* discrete levels) to guide component developers *and* system builders in their evaluation of the reusability of individual components (and, by extension, component *frameworks*), and (ii) motivations as for why these benchmarks can indeed measure *best practice* in the domain of compositionality and composability. The basis of the contributions are three strong *"axioms"* that form a *paradigm* of complex systems design, with special attention to the particularities of *robotics* systems.

Keywords: reusability, composability, compositionality, component-based software systems, separation of concerns, design of complex systems

1 Introduction

Roughly since the beginning of the 21st century, *common off the shelf* hardware (actuators, sensors, data acquisition modules, fieldbusses,...) and general-purpose software infrastructure (operating systems, development tools, communication middleware,...) have become cheap and powerful enough, to allow (larger) robotics groups worldwide to start experimenting with the construction and control of *"complex robot systems"*; e.g., the German *Desire*, *Justin* and *Care-O-Bot* robots, the American PR2 robot, the Korean and Asian humanoids, etc. This paper does not focus on the kind of complexity stemming from the *individual functionalities* being put in the robot (e.g., planning and control *algorithms*), but about the complexity of *combining* more and more components together in a system. In this latter case, the "non-functional" aspects require a more-than-linear increase of attention from the *system* designer: the communication between components, the appropriate configuration of each component and all of its communications, and, in particular, the coordination between the "activities" that run inside interconnected components, all require *much* more effort when dealing with 100 components than with only a handful of components, because the number of ways that components can interact scales exponentially with the number of interacting components. Hence, the predictability of the system behaviour and performance

risks to go down very fast with the scale of the system. *Unless* the component developers foresee this integration scaling problem, and design the components they offer in such a way that later integration by the system designer becomes easier. And this *integration facilitation* is the main topic of this paper, i.e., how to make the job of, both, component providers and component-based system builders, easier, by providing a set of insights, guidelines and benchmarks that can make components more *composable* and systems more *compositional*.

1.1 Reusability, composability, compositionality

The following definitions are being used in this document:

- *Composability*: the property of a *component* to be integrate-able into a larger system.
- *Compositionality*: the property of a *system* to have predictable performance and behaviour if the performance and behaviour of the components are known.

Of course, both properties are not independent, and achieving full compositionality and composability is an ideal that is probably impossible to achieve. This paper makes contributions at exactly this boundary between components and systems, to give developers a structured context for *evaluating* ("benchmarking") how their software deals with the design force of *information hiding* that naturally drives component developers and system builders in opposite directions: component developers want to show as little as possible to the outside world about how their component works internally, while system builders want to optimize their system's robustness by selecting components whose internal design is such that it can cope with as many system variations as possible.

This paper claims contributions in improving *reusability* and *composability* of components: it explains to component builders what to take into account during their developments to make the resulting *component* as reusable as possible, at the level of a "binary blob" of code that can be integrated in as many different system architectures as possible. The focus is not on the reusability and the performance of the *source code* from which the component is being built; this source code's "flexibility" ambition is very worthwhile in itself, and fully complementary to the ambitions of this paper, but both ambitions are best kept separate. So, this paper does not discuss *continuous benchmarks* to measure the *"performance"* of *one particular implementation* of a component or of a system design; but it presents discrete benchmarks that are comparable in nature and purpose to other already established discrete evaluation measures (in other domains than robotics software!) such as *Technology Readiness Levels* (TRL) [13], or *Safety Integrity Levels* (SIL, described in the IEC 61508 standard). The similarity between all these discrete benchmarks is that they all measure the advancement of the (sub)systems that an application or technology domain has already made towards an ambitious, "final" target situation. In the context of this paper, this final target would be the situation in which every component can be reused in any possible system architecture without any redesign or even recoding.

1.2 Standardisation vs harmonization

One way of measuring the *maturity* of a technology domain is by the availability and practical acceptance of *standards*. According to this metric, robotics is not a mature domain at all. But, it is also not the ambition of this paper to present material that is ready to be transformed into a standard, let alone to pretend to become *the* standard, with which to measure how good a particular complex robot software system is. However, the ambition *is* to set a first step in this direction, by making concrete and motivated suggestions *to harmonize* the way different robotics software frameworks develop, present, document, evaluate, defend and integrate software components (from different "vendors"!) into one single robotic system, or into a "system-of-system" application (e.g., a RoboCup footbal *team*). The approach of the paper is, to the best of the author's knowledge, unique and original.

1.3 Robotics frameworks: Orocos, ROS, OpenRTM, OPRoS

The paper makes its rather abstract harmonization discussion more concrete, by illustrating it by four of the worldwide most applied and featureful open source robotics software framework: Orocos [4, 5, 20, 19], OpenRTM [14], ROS [22], and OPRoS [8]. GenoM [7, 12], Miro [11, 1, 21] and Orca [3] were "early movers" in the same

domain, but they (seem to) have failed to gain the critical mass to attract the attention of modern service robot builders. Most of these frameworks score only a "1" or a "2" on the scales of four, connected to each of the presented benchmarks in Sec. 3.

1.4 Lessons learned

This Section summarizes a number of often-experienced causes for failure in creating and running complex systems, in the field of robotics, but also in many other domains. The motivation for having this list is to use it as a check-list in the later Sections of the paper, to help make sure that the suggested contributions can indeed lead to improvements in the state of the practice in robotics. This Section has no references to the literature (being only based on the personal experience of the author), for the simple reason that system builders tend not to publish much about their failures... "Failure" can be as clear as "simply not working at all", but most often it is invisible behind the screens of a working system, in the form of, for example, too high development efforts, far from optimal performance, cumbersome maintenance, missed opportunities for reuse of previous development efforts or of "off the shelf" components, race conditions between components striving for the scarce resources in the system, etc. In addition to being invisible, these failure causes are also extremely difficult to benchmark, or to quantify in a measurable way. four-level scale of evaluation for each of them. The author's subjective *top three list* of causes for *system-level* failure is as follows:

- the system design is too much driven by *one single* goal (function, hardware,...). This quickly leads to *too much and too early optimization* in the components for use within that particular system. And hence, makes it more difficult for system builders to adapt the system to evolving requirements (in functionality, in hardware platform, in user interfaces,...), or to reuse their efforts in other systems.
- the system is designed with a *too centralized coordination* design: all decisions in the system are taken by one single component, and its developers (most often, implicitly!) assume that that component has instantaneous access to all information from all over the system, whenever it needs that information and with the required quality. This assumption is quickly violated in modern robotic systems, and certainly so in multi-robot systems.
- the system and its components come with *too many hidden assumptions built in*, about the hardware platform, the performance requirements to be optimized, the interaction *policies* between components, the *real meaning* of the data and services exchanged between components, the dynamics in the environment and in the robot's tasks, etc. Each particular assumption about these issues is prone to lead to design decisions that are difficult to change afterwards. This problem escalates in the typical context where (i) the assumptions have never been made explicit, and (ii) it's not only the original developers who want/need to update the components' code and design.

Overview of the paper. This paper makes an effort to remedy the problems outlined in the previous paragraphs, by introducing (Section 3) robotics software developers to ten complementary aspects of complex robotics systems for which a simple but effective four-level scoring scale is given and explained. But first, Section sec-paradigm makes explicit in which context ("paradigm"), and for what reasons, the suggested evaluations scales are presented.

2 The harmonization paradigm: 4G, 4C and 4S

This Section explains the *context* in which the contributions of this paper have to be interpreted and evaluated. The ambition is to describe this context by the following *minimal amount of structure*: (i) the four levels of *granularity* ("4G") with which software entities are designed and provided, (ii) the four design *concerns* ("4C") that need to be kept separate for optimal composability, and (iii) the four levels of *system complexity* ("4S") that must be supported. In contrast to the set of four-valued *ordinal*[1] benchmarks of Sec. 3, this set of four "levels" should not be interpreted as "benchmarks", but only as a *categorical*[2] structure.

[1] Providing a *strict order* that is interpreted as a *quality value* assessment.
[2] Providing a *classification* to which no quality value assessment should be attached.

2.1 The role of paradigms

The (refutable) "research hypothesis" of this paper is that the contextual structure of 4G–4C–4S is complete and simple enough to guide all discussions about (what is important in) the design of reusable components for building complex robotic systems. In itself, the introduction of those context and structure to the robotics community is considered a major contribution of the paper, since their role is to serve as a *paradigm*. In general, a paradigm is the set of all models, thought patterns, techniques, practices, beliefs, mathematical representations, systematic procedures, terminology, notations, symbols, implicit assumptions and contexts, values, performance criteria,..., shared by a community of scientists, engineers and users in their modelling and analysis of the world, and their design and application of systems. So, a paradigm is a *subjective, collective, cognitive but often unconscious* view shared by a group of humans, about how the world works, or about how an engineering system *should* work. Examples of such scientific paradigms are: the dynamics of Newton and Einstein, the astronomical theory of Copernicus, Darwin's evolution theory, meteorological and climatological theories, quantum mechanics, etc. Some of those are universally accepted, others are less so. And all of them have, to a high extent, a *subjective* basis of concepts that are deemed to be essential and accepted without proof. But all of them are also *scientific*, in the sense that all interpretations and conclusions based on experimental data, and made on top of the paradigm's subjective basis, are derived in a systematically documented, refutable, and reproducable way. The ambition of this paper is to provide an explicit and motivated structure for *robotics system* design aspects, facilitating a documented and refutable discussion about how to best create and evaluate such complex robot systems. This includes the discussion about whether or not the presented paradigm itself is an appropriate choice.

The good news of having paradigms is that practitioners within the *same* paradigm need very few words to communicate or discuss their ideas and findings (or to document their software), because they share the paradigm's large amount of (implicit) background knowledge and terminology. The bad news is that practitioners from *different* paradigms often find it difficult to understand each other's reasoning and to appreciate each other's procedures, "results" and values. The largest difficulty of all is caused by their inability to realise that they are thinking inside different paradigms in the first place. So, a major goal of this paper is to try to explicitize some paradigmatic assumptions which form the "value system" with which the robotics community can discuss and evaluate the design quality of robot software systems.

In the context of this paper, the most probable "paradigm clashes" are those between (i) "object-oriented programming in the small" and "component-based programming in the large", and (ii) centralised and decentralised coordination. The former is explained in Sections 2.2 and 2.3, on, respectively, the granularity of software entities, and on separation of concerns; the latter is the subject of Section 2.4, on the categories of (de)centralised system coordination.

Whether the material in the following Sections will turn out to be indeed good enough to form a paradigm, depends on whether the community thinks that it is (i) sufficiently *abstract* to be easy to explain, to discuss about between developers, and to put into software documentation, while (ii) still having sufficiently *strong and innovative structure* to make a noticeable impact on the design and analysis practice for robotic software systems and components. While much of the discussion is (on purpose) applicable to all sorts of complex software artifacts, it is particularly suited for robotics use cases: the *task* and the *environment* of the robot often impose a level of complexity (functional, interaction wise) that no software design can make go away, and hence must deal with.

2.2 4G: granularity of software primitives

A first pillar of the paradigm has to do with the amount of code that is designed to be "reused" as a whole, and about the "primitives" that are provided at different levels. Packaging code into modules of the "appropriate size" is important, since humans can only deal with a maximum amount of complexity before they loose oversight (and hence their ability to design). The following four levels of granularity are deemed to provide that "appropriate" complementarity of levels of abstraction:

G1 ***Data structure***: finding the "best" representation for *one single piece of reality* requires full attention of the designer, but at a short time scale, and with a very narrow focus.

G2 ***Class***: this level is about finding the "right" selection of *highly related objects* (their data structure and methods) to provide as *one inseparable service* to be published and reused.

G3 ***Packages***[3]: this level's ingenuity lies in finding the right collection of Classes *to pack together* in one single, relevant namespace, and such that all Classes are allowed to access all other Classes in the same package.

Up to this level, developers typically assume that they can have access to all *source code*.

G4 ***Module*** (component, agent,...): the level on which developers are not focusing on source code anymore, but on how to best *export/import services* that can be used as stand-alone "software products". This level must also pay attention to attach the right meta-information about how a component becomes visible (and behaves itself), such that the system in which it is reused can "optimally" configure it.

2.3 4C: separation of concerns

This Section is based on the not so well known work of [17], whose relevance to robotics was only recently explained in the White Paper [16]. It claims that the following four aspects of separation of concerns [6] are *indispensable* for the analysis and synthesis of complex software systems:

C1 ***Computation***: the code that provides the useful application-specific *functionality* that is being computed inside the components. From the users' perspective, the added value of the whole system is only coming from this part of the system; the three other aspects below are "just overhead", that users ideally don't want to spend resources on (time, CPU, communication bandwidth,...).

C2 ***Communication***: the code that supports exchange of data between Computations in different components. Communication costs typically scale much worse than linearly with the number of interacting components in a system.

C3 ***Configuration***: the software that allows each component's Computations to get their most appropriate (application-dependent) parameter settings from the users, or the system designers, whenever needed and from wherever possible. This can be as simple as setting the values of some control gain parameters, but also as complex as choosing a threading model, or a buffering policy in Inter-Process Communication.

C4 ***Coordination***: the software that supports the Computation inside components to switch their *behaviour* in a coordinated way over all components involved in a particular interaction. Coordination complexity also rises more than linearly with a growing number of components in a system.

The desire to separate Computation from Communication was a major reason behind the creation of all large-scale robotics software frameworks. However, none of those frameworks currently has reached a status in which all 4Cs are already taken into account in the decoupling design: especially Configuration and Coordination typically don't get a *first class citizen* treatment, and tacitly sneak in in many of the Computations and Communications components, without developers being aware of the reusability consequences of not separating them explicitly.

2.4 4S: system coordination complexity

Most code, literature and developers discussions are (tacitly) making a number of assumptions about the complexity they have to deal with. Hence, their designs only work well in systems with a matching complexity level, hence reducing robustness and reuse opportunities in systems with higher complexity. In robotics, use cases exist for all of the following four levels, with a growing importance for the higher complexity classes:

S1 ***All 4Cs centralised***. This is the case in most of the *older* industrial robots, whose software was directly reused from the hard automation context.

S2 ***Centralised coordination***. Most modern industrial, humanoid, and mobile robots have already rather complex hardware designs, involving distribution over field busses, and multi-CPU computing. Hence, computations and communications are rather well separated. But there is still one single task that coordinates (and configures) everything that is going on in the whole system.

[3]The Java nomenclature is used in this Section; other programming language ecosystems might give other names to the same levels of code abstraction.

S3 ***Decentralised coordination, with shared task***. A RoboCup team of soccer robots is a typical use case: all robots do most of the decision making on their own, but they have been designed to always cooperate as a team, with *one single, system-level task*.

S4 ***Decentralised coordination, with individual tasks***. A typical example of this complexity class is a fleet of *robotised* wheelchairs in *"ambient intelligence"* environment such as a hospital: each wheelchair will have its own tasks (driving to, and then transporting, one single patient), which it has to realise in an environment *whose resources it has to coordinate* with other devices that do not share the same task.

2.5 Paradigm of this paper

To summarize the previous Sections: the material presented in Section 3 must be interpreted, discussed and evaluated in a context in which the following are the most important design aspects:

- *G4*: the robots are built from "large-scale" software combinations, typically provided in binary packages only.
- *C2, C3, C4*: the efforts spent on making the robot's software are not only spent on the *computational functionality* of the components, but more on more on all the "invisible" infrastructure that is needed to make those computations work together, system wide.
- *S3–S4*: the robots are so-called "system-of-systems", in which distribution over networked computing infrastructure is inevitable.

The material is also *useful* but not *necessary* for components and systems at the lower levels than the one just discussed.

3 Harmonization benchmarks

This Section introduces ten complementary aspects of component-based robotics system design. (The order in which they are presented is arbitrary, and should not be interpreted as a reflection of "importance".) Each aspect is scored on a discrete scale from "1" to "4", with the latter representing the best composability/reusability.

3.1 4D: dynamism

D1 ***Static***. The component always provides the same functionality and behaviour, since the system in which it works does never change.

D2 ***Dynamic***. The component is robust against the system in which it works being extended with new components, but the component itself does not change its functionality or behaviour.

D3 ***Adaptive***. The component does change its functionality or behaviour, in order to improve its "performance" in a dynamically changing system.

D4 ***Self-organising***. The component provides an adaptation that always results in a *stable* new system, irrespective of the magnitude of the change.

No component in the four robotics frameworks scores better than "D2".

3.2 4F: Finite State Machine composability

Finite State Machines are, in one form or another, often used as the instrument of choice to implement *Coordination*. Their reusability can be measured as follows:

F1 ***Single state chart semantics***. The FSM implements one particular FSM specification, such as those in UML, or Matlab/Simulink. In practice, these different specifications have so many "semantic deviation points" that the Coordination behaviour can not be predicted when making interconnections between two or more components that implement different FSM specifications.

F2 ***Composable subset of state charts***. Component developers who are aware of the above-mentioned problem should restrain themselves from implementing more FSM features than just parallel and hierarchical states, with basic transition guards. These are the only features that all FSM implementations provide, so composability is improved.

F3 ***Robust against semantic policies***. On step further than *avoiding* the semantic deviation problem is: being robust against them. That means that a Coordination component should provide the same behaviour to its robotic system, irrespective of, for example, in which order the FSM implementations in the Computational components in the system do condition checking, or transition event handling.

F4 ***Robust against interaction unpredictabilities***. The highest composability a Coordination component can achieve is to make sure that its Coordination *does not depend on the assumptions* that other components (*and* the real world!) make about their dynamic behaviour.

No robotics frameworks scores better than "D1"; or rather, most of them do not even have a decent FSM implementation.

3.3 4IC: component interaction constraints

Components are, by definition, required to interact with other components. Each such interaction puts constraints on the working of each of the interacting components. The composability of a component is scored by to what extent it deals with these inter-component interactions:

I1 ***Set-point***. In the simplest case, the *output* of one component is an *immutable input* for another component.

Block diagrams for control are typical examples of I1 interaction.

I2 ***Bi-directional, hard constraint***. Similar to I1, with the difference that the component allows *bi-directional interaction* with other components. That means that no *a priori* input/output causality is imposed on the interaction.

Bond Graphs [10, 15] for control are typical examples of I2 interaction.

I3 ***Soft constraint***. The components interact *bi-directionally* and the nominal interaction constraint can be "*violated*" (at a certain cost) by all interacting components *independently*. The softness of the interaction constraint offers all interacting components more opportunities to keep the interaction satisfied under various disturbances.

The *Fast Research Interface* of the KUKA Light-Weight Arm, [18] provides a soft constraint interaction in various ways: it allows the interacting component to vary the frequency of the Communication between 100Hz and 1000Hz, *and* it offers an *impedance control* [2, 9] which is a physically soft constraint.

I4 ***Soft constraint with constraint monitoring***. Like I3, plus the component is configurable with various *magnitudes* of violation of the interaction constraint, which each give rise to various *constraint violation events* to all interacting components (*and* to the *system Coordination*). This allows a higher level of *adaptation* in the components.

No robotics software framework scores better than I1.

3.4 4IPC: Inter Process Communication

Communication is an indispensable case of inter-component "interaction", limited to the exchange of digital information. The following levels impose an increasing complexity (but also an increasing flexibility) to the communicating components:

IPC1 ***Synchronous method calls***. The component *controls* the state of all (CPU, RAM) resources within its own address space.

IPC2 ***Asynchronous method calls***. The component must, in addition, be *robust* against concurrent allocation requests to *RAM* resources within its own address space.

IPC3 ***Synchronous message passing***. The component must, in addition, be *robust* against *timing* inpredictabilities of its (CPU) resource.

IPC4 ***Asynchronous message passing***. The component must, in addition, be *robust* against inpredictabilities of the *other* programs it interacts with.

The robotics frameworks support all four levels, but they focus on IPC1, because that's easiest for *performance optimization*, while IPC4 is best for *composability* but most difficult to coordinate predictably.

3.5 4K: knowledge representation

Components can not be composed into a system if their Computations inside do not speak "the same language." There are four levels to be distinguished:

K1 ***Paradigm***. The components (or rather, their developers) agree *implicitly* about a lot of hidden assumptions and "values". It's here that *religous wars* are fought, and *subtle* system-incompatibilities sneak in, unnoticeably.

K2 ***Glossary***. The components share a *list* of terms and concepts, each defined separately.

K3 ***Taxonomy***. The components share a *tree* of relationships between concepts. A tree allows for more complex knowledge representation, hence for more complex component interactions.

K4 ***Ontology***. The components share a *graph* of relationships between concepts, again increasing the "semantic richness" of their possible interactions.

ROS and Orocos have achieved a first integration of their Communication at the K2 level, but there does not yet exist any real integration between all robotics frameworks. There *is* a growing awareness of the importance of sharing the same knowledge representations amongst all frameworks' developers, but this has not yet resulted in implementations that can profit from it.

3.6 4M: middleware awareness

All robotics software frameworks use various forms of Communication middleware, to facilitate distribution. However, too often, their components are still too much aware of the presence and the properties of that middleware. Hence, they seldom score better than M2:

M1 ***Client sees protocols of middleware***. That means that one finds, in the component's Computation, code that does things such as: *piggy-backing* events on data flow, implementing its own data flow buffering policy, etc.

M2 ***Client sees status of middleware***. The components' Computation contains code to check whether it is possible to send or receive data.

M3 ***Client sees middleware brokerage*** The components Computations still contains code to find the middleware registry, or to cope with dynamic changes in the interconnections, etc.

M4 ***No requirement on client to interpret messages***. The components only see the messages, and not the middleware, and they can even receive messages from others without failing, when they do not "understand" the messages they receive.

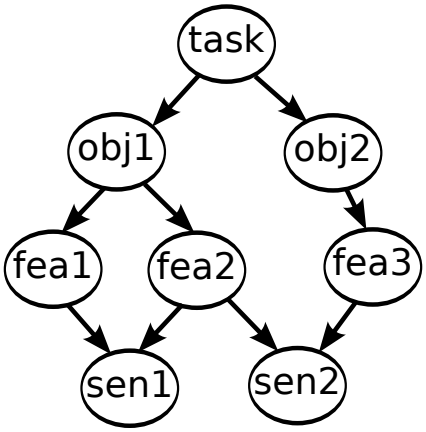

Figure 1: The highest level of perception components have explicit representations and Computations to deal with the (bi-directional!) exchanges of information between all four depicted levels.

3.7 4O: object representation

Since components must interact, they must have means to make sure that the data they exchange are correctly interpreted. So, it is not enough to know that a message consists of four bytes, representing a float, but the meaning of that number should be defined at higher levels than the computer hardware.

O1 ***Object file representation***. The executable forms of a Computation agree on the hardware types of data (float, integer,...).

O2 ***Programming language representation***. The Computations in two components can reuse each others' source code.

O3 ***Mathematical representation***. The different components use the same *formal* representation of data, so they can automatically generate source code from that model, in the programming language of their choice.

O4 ***Ontological representation***. Different components also share the *exact same meaning* of all operations and data structures with which they interact.

All robotics frameworks get no higher than O2, but the first inter-framework efforts to share the same mathematical models are expected to occur in the near future.

3.8 4P: perception

All robotics systems must have some way to interpret how the world looks like in their environment. Full perception (P4) means that all four of the following levels of sensor information interpretation are integrated (Fig. 1):

P1 ***Sensors***. This is typically not more trivial than providing a "device driver" to get the information from the physical sensor into the components that want to use it.

P2 ***Features***. This involves (the integration of) multiple raw sensor data into non-directly observed features. Being able to provide information about features leads to both data reduction (less bytes needed) and a first level of abstract interpretation of the world.

P3 ***Objects***. This involves the next step in abstract interpretation of the world, by appropriately combining the information about features into information about real objects that are relevant to the robots' task.

P4 ***Task/Affordance***. At the highest level of abstractness, the robot is able to interpret the progress in the task it is performing, and/or to identify what affordances the objects in its environment provide to the robot.

Robotics frameworks begin to mature at the P3 level, and to explore P4.

3.9 4R: reactivity

Robot systems are expected to react appropriately in more and more complex environments, and this ability is scored by the following benchmark levels:

R1 ***Feedback***. This is traditional and mainstream in robotics.

R2 ***Feedforward***. Also traditional, but already a bit less mainstream in most robotics components.

R3 ***Proactive/intention feedforward***. This allows peer components *to negotiate* about their interaction, and not just interact with feedback and feedforward.

R4 ***Undo-able reactivity***. Components that can undo what they previously did (in an interaction with one or more of their peers), are *more robust* against the unpredictability of their *peers* (and their environment).

None of the robotics frameworks offer components that can score R3 or better.

3.10 4RA: resource awareness

Every component consumes some resources when providing its functionality: memory, CPU utilization, energy, communication bandwidth, hard disk space, etc. Higher awareness of its resource needs, and a higher ability to share resources with other components (in a coordinated way!), make a component more composable, since it will be able to interact with the "platform" on which it is deployed to work with the available resources, at the moment these resources are made available to it. Resource sharing is still a difficult to compose part in modern robotic systems, and hence a frequent cause of *system* failure. Hence, the domain of robotics should still work hard to score better on the following scale of resource awareness:

RA1 ***Hidden by programming language***. This is traditional and mainstream in robotics: programmers just assume that all the resources they would like to have are available when needed. In the best case, they will check whether they indeed get the expected resources, and go into a failure mode when that is not the case.

RA2 ***Mutual exclusion***. This is also traditional in software engineering, but not yet mainstream enough in all current robotics frameworks. It means that components get access to their resources, in a cooperative, exclusion-aware, but not actively coordinated way.

RA3 ***Active object***. The robotics system explicitly provides one or more components that do nothing else but coordinating the access between (competing, exclusion-unaware) clients.

RA4 ***Active resource sharing***, not via dedicated components, but by the peer components themselves, since they are aware of the resource, and know how to follow access coordination policies. This can only work if all components *communicate* their resource *allocation intentions*, and are also able to monitor resource *violation trends*.

4 Conclusions

The aim of this paper is to provide a structured foundation for discussing the design qualities of complex robot systems, at the boundary between the two complementary aspects of *component design* on the one hand, and *system design* on the other hand.

The first contribution towards this goal is in making explicit the "paradigm" within whose context such discussions can take place, including the discussion about whether or not the presented paradigm is an appropriate one.

The second contribution of the paper consists of a range of discrete benchmarks to help component and system designers to make motivated choices during their development process. These benchmarks can also help

the robotics community as a whole, in evaluating how well it is progressing towards (the software support for) the ultimately intelligent robot systems of the future.

All benchmarks provide a discrete scale from "1" to "4", with a higher score indicating a better reusability/composability *potential* of a component in a system with higher demands on inter-component interactions. The paper focuses on the abstract *design* level, and its benchmarks do not provide quantitative scores to assess the quality of the *implementation* of components or systems.

The ambition of this paper is to provide component developers and system builders with a pragmatic set of evaluation criteria, with which they can benchmark their own designs with respect to the "holy grail" in robotic systems design and implementation. (This holy grail would consists of scoring "4" on all the presented benchmarks.) As witnessed by the low scores of the four major software component frameworks in robotics (Orocos, ROS, OpenRTM, and OPRoS), the robotics community has still a tremendous amount of work to perform. The good news is that, in a component-based context, such progress can be made component by component.

Component builders do not *have* to make their components score the maximum score of four on the presented benchmarks, in case they are developing for S1 or S2 systems such as "cheap" robots with deeply embedded control systems. However, end-users will, sooner or later, want to use even the simplest of such robots as components in a larger system. (This larger system need not necessarily be a *robotic* system, but can be other engineering systems such as warehouse IT.)

Acknowledgements

The author acknowledges the support from the K.U.Leuven Geconcerteerde Onderzoeks-Acties *Model based intelligent robot systems* and *Global real-time optimal control of autonomous robots and mechatronic systems*, and from the European project FP7-231940, *BRICS* (*Best Practice in Robotics*). Many thanks to all the BRICS partners that have critically discussed earlier versions of this document. Special thanks go to Klas Nilsson, for laying the foundations of this work in our cooperation within the FP7 *RoSta* (*Robot Standards*) project.

References

[1] Miro robotics middleware. http://www.informatik.uni-ulm.de/neuro/index.php?id=321.

[2] A. Albu-Schäffer, C. Ott, U. Frese, and G. Hirzinger. Cartesian impedance control of redundant robots. Recent results with the DLR Light-Weight-Arms. In *Int. Conf. Robotics and Automation*, pages 3704–3709, Taipeh, Taiwan, 2003.

[3] A. Brooks, W. Kadous, T. Kaupp, A. Makarenko, and A. Oreback. Orca: Components for robotics. http://orca-robotics.sourceforge.net/, 2004. Last visited March 2010.

[4] H. Bruyninckx. Open Robot COntrol Software. http://www.orocos.org/, 2001. Last visited 2010.

[5] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the Orocos project. In *Int. Conf. Robotics and Automation*, pages 2766–2771, Taipeh, Taiwan, 2003.

[6] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[7] S. Fleury, M. Herrb, and R. Chatila. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, pages 842–848, Grenoble, France, 1997.

[8] K. I. for Advanced Intelligent Systems. OPRoS. http://opros.or.kr/.

[9] N. Hogan. Impedance control: An approach to manipulation. Parts I-III. *Trans. ASME J. Dyn. Systems Meas. Control*, 107:1–24, 1985.

[10] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg. *System Dynamics: A Unified Approach*. Wiley, 2nd edition, 1990.

[11] G. Kraetzschmar, H. Utz, S. Sablatnög, and S. Enderle. Miro—Middleware for cooperative robotics. In *RoboCup 2001: Robot Soccer World Cup V*, volume 2377/2002 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2002.

[12] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building middleware-independent robotic components. In *Int. Conf. Robotics and Automation*, Anchorage, Alaska, USA, 2010.

[13] J. C. Mankins. Technology Readiness Levels. A white paper. Technical report, NASA, Office of Space Access and Technology, Advanced Concepts Office, 1995.

[14] National Institute of Advanced Industrial Science and Technology, Intelligent Systems Research Institute. OpenRTM-AIst. http://www.openrtm.org.

[15] H. M. Paynter. *Analysis and design of engineering systems*. MIT Press, 1961.

[16] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov. The use of reuse for designing and manufacturing robots. Technical report, Robot Standards project, 2009.

[17] M. Radestock and S. Eisenbach. Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond*, pages 162–176. Springer-Verlag, 1996.

[18] G. Schreiber, A. Stemmer, and R. Bischoff. The Fast Research Interface for the KUKA Lightweight Robot. In IEEE Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications How to Modify and Enhance Commercial Controllers (ICRA 2010), May 2010.

[19] 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. http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.

[20] P. Soetens and H. Bruyninckx. Realtime hybrid task-based control for robots and machine tools. In *Int. Conf. Robotics and Automation*, pages 260–265, Barcelona, Spain, 2005.

[21] H. Utz, S. Sablatnog, S. Enderle, and K. G. Miro—Middleware for mobile robot applications. *IEEE Trans. Rob. Automation*, pages 101–108, 2002.

[22] Willow Garage. Robot Operating System (ROS). http://www.ros.org, 2009.