Best Practice in Robotics (BRICS)

Grant Agreement Number: 231940

01.03.2009 - 28.02.2013

Instrument: Collaborative Project (IP)

Harmonised interfaces and full integration of harmonisation guidelines in tool chain

Herman Bruyninckx

Deliverable: D8.2

Lead contractor for this deliverable:	Katholieke Universiteit Leuven
Due date of deliverable:	April 20, 2012
Actual submission date:	April 26, 2012
Dissemination level:	Public
Revision:	1.0

Executive Summary

Because robotic systems get more complex all the time, developers around the world have, during the last decade, created component-based software frameworks (Orocos, OpenRTM, ROS, OPRoS, SmartSoft) to support the development and reuse of "large grained" pieces of robotics software. This document describes the results of the project's *Component Model Task Force*, which created (under the driving force of the Harmonization Work Package 8) the *BRICS Component Model* (BCM) as the major paradigm to harmonize the efforts, designs and code of all developers contributing to robotic system software.

The BCM's harmonizing potential lies in it providing as much structure as possible during the development of, both, individual components and systems of components, irrespectively of which of the above-mentioned code frameworks are used (possibly more than one at the same time and in the same system!), and without introducing any framework- or application-specific details.

The BCM is built upon two complementary paradigms: the "**5Cs**" (separation of concerns between the development aspects of Computation, Communication, Coordination, Configuration and Composition) and the "**metamodelling**" approach from Model-Driven Engineering. These paradigms help developers who are producing code libraries in the (at least, in the robotics domain) traditional way (that is, with design and implementation done at the level of a concrete programming language), but the paradigms were also designed to form the core of a model-driven toolchain that can support developers at a higher level of abstraction, platform-independence, and, hence, reusability than what is common in robotics.

In addition, the BCM has proven to be also able to bring more structure in the *development process* itself, by identifying which aspects of the Component Model needs to be taken care of in the major phases of that development process: functional design, component design, system design, deployment, and runtime coordination. (In this way, the BCM forms also the core of the *Wikibook* that is provided as a Deliverable in the toolchain Work Package 4.)

Somewhat unexpectedly compared to what the writers of the project's *Description of Work* had foreseen some four years ago, the concept of concrete sets of *Application Programming Interfaces* (APIs) has lost the central role and importance it has in "traditional" programming, and that central role is now taken over by the **semantically much richer** concept of a *Component Model* (in which APIs are just one of the relevant system design instruments).

The BCM Task Force was active while having in mind as the ideal audience the senior robot software system engineer, with several years of experience in one or more of the mentioned robotic software frameworks, and with an interest in using *models* of components and systems *to generate code* instead of hand-crafting it.

The **impact** of the BRICS Component Model turns out to be much broader than the initially anticipated impact of the Harmonization WP, in the sense that it structures the whole development process with only a relative small set of semantic primitives. On the other hand, the non-foreseen application of the metamodelling paradigm resulted in a more abstract form of impact, which is definitely less intuitive to appreciate for junior developers. The harmonization impact has already been realised *within* the BRICS project (by influencing the way code is developed and refactored in the other Work Packages in the project), but also already *outside* of the project, via dissemination workshops both with targeted developers teams and on public fora, e.g., the European Robotics Forum), and more in particular in the Rosetta project (where the same metamodelling paradigm has been adopted to harmonize the knowledge representation aspects of complex *task specifications* for robot systems).

Contents

1 Introduction **1**

1.1 The BRICS Component Model 1

1.2 Related Work 2

1.3 The metamodelling paradigm from Model Driven Engineering 3

2 The BRICS Component Model as harmonizing metamodelling paradigm **5**

2.1 M3: The Component-Port-Connector metametamodel 6

2.2 From M3 to M2: the "5Cs" 6

2.3 The BCM and the Development Process 7

3 Best practices — Toolchain guidelines **10**

3.1 Coordination 10

3.2 Composition 10

3.3 Computation 11

3.4 Configuration 11

3.5 Communication 11

3.6 Miscellaneous 12

4 Conclusions **13**

5 Acknowledgements **14**

Bibliography **15**

1 Introduction

Because robotic systems get more complex all the time, requiring the integration of motion controller(s) on many motion degrees of freedom, multiple sensors distributed over the robot's body and embedded in its environment, planners and reasoners for ever more complex tasks, etc., developers around the world have, during the last decade, created *component-based* software frameworks [3, 28] (such as Orocos [5, 4], OpenRTM [19], ROS [30], OPRoS [16], Genom [11, 17, 12], SmartSoft [26, 25], Proteus [14]) to support the development and reuse of "large grained" pieces of robotics software. Component-based development complements the more traditional object-oriented programming in roughly the following ways: it focuses on *runtime composition* of software (and not *compile time* linking), it allows multiple programming languages, operating systems and communication middleware to be used in the same system, and it adds *events* and *data streams* as first-class interaction primitives in addition to method call APIs.

1.1 The BRICS Component Model

This document describes the *BRICS Component Model* (BCM) to provide developers with as much structure in their development process as is possible without going into any application-specific details. This BCM structure is, on the one hand, rich enough to make a real difference (as has been proven in the last two years by disseminating it to hundreds of robotics PhD students and engineers), and, on the other hand, generic enough to be applicable in all of the above-mentioned robotics frameworks.

The BCM is a *design paradigm*, in that it introduces a methodology without being able *to prove* that that methodology is "better" than other approaches. But the last Chapter of this document presents a number of *best practices* for robot system design that are based on the BCM paradigm, and with which we have helped developers find better solutions to real problems in their complex robotics applications.

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.

Examples of such scientific paradigms are: the theories of Newtonian and Einsteinian dynamics, 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 certain extent, a subjective basis. But all of them are also *scientific* paradigms 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. This is the reason why the BCM creators put much emphasis on the complementary roles of (i) advocating the importance of introducing formal *models* into the robotics development process (instead of the mostly *code-only* frameworks that are now popular), and (ii) well-documented "best practices" that solve particular use cases. The medium-term ambition of the BCM developments is that, both together, will suffice to allow developers to first *model* their components and systems (hopefully starting from a rich model repository with "best practice" sub-systems and components), and then *generate code* from those models, using the best available implementations in available software frameworks.

The good news of having paradigms is their harmonization impact: practitioners within the *same* paradigm need very few words to communicate or discuss their ideas and findings, because they share the paradigm's large amount of "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 and results. And to realise that their difficulties are caused in the first place by their thinking inside different paradigms.

Hence, this document should be read with a mindset that already accepts the paradigm of model-driven engineering "metamodelling" as meriting its legitimate place in robotics. This is not at all obvious in the state of the practice of most current robotic software developments, since the community is currently still exploring what is the best way to realize large-scale robotics software systems. This document's ambition is to explain in what ways the BRICS Component Model contributes to this exploration.

1.2 Related Work

At the time that the robotics community became aware that it makes sense to spend time developing reusable software frameworks,[1] the *CORBA Component Model* (CCM) was a large source of inspiration. However, the advantages of its component model rather quickly lost the battle against the huge learning curve and massiveness of the CORBA standard: it was considered to be way too heavy and all-encompassing for the needs of the robotics community. So, the CCM did not survive too well in mainstream robotics, except for the (at least in the Western world) less popular OpenRTM, OPRoS, and SmartSoft, and, to a smaller extent, Orocos which has had industrial users since its inception. Nevertheless, the most recent "Lightweight" version of the standard [23] has a focus on realtime and embedded systems, and would fit a lot better to the current robotics needs and mindset in the community. However, the ROS framework has in the meantime attracted most of the community's attention, via its low entry threshold (in combination with a tireless and very active support from Willow Garage). But of all the robotics software frameworks mentioned in this document, it is farthest away from the CCM. The BCM introduced in this document *shares* (only) its *structural* model with that of the CCM (components, interacting via methods, data streams and/or events, and *composed* by *ports* together). But the BCM *adds* to the CCM four complementary *types* of components, with clear semantic meaning in the context of robotics, so that it can guide developers in separating the functionality of their applications in *reusable* parts, that can be *composed* in larger systems within various concrete application contexts.

Robotics software development shares its context and goals with many other engineering domains that require lots of online data processing. For example, some large application domains outside of robotics also have seen the need for component-based software frameworks [8]: *automotive* (with AUTOSAR [6] as primary "component model" standard, and AADL [27] as pioneer modelling language for "computational resources"), *aerospace* (driving UML-based evolutions such as MARTE [21]), *embedded systems* [18, 15], and *service component architectures* in web-based systems [20]. However, none of these component models has seen the explicit need to introduce (all of) the BCM's "5Cs" (Chap. 2.2) as its formal set of component types, but only go as far as identifying and supporting three or four of them. (Running ahead a bit on the material introduced in Chap. 2.2, the missing component types are most often *Configuration* and *Coordination*.)

Of the most popular robotics software frameworks in the "Western" robotics community, ROS nor Orocos have an *explicit* and *formal* component model, in contrast to OpenRTM ("Japan") and OPRoS ("Korea"), which both use Eclipse [10] as a *programming* tool (but only to a limited

[1] This was around 2000, with GenoM [11], Player-Stage [13], Miro [29], Orocos [5] and SmartSoft [26] as pioneers.

extend as a *model-driven engineering* tool). However, also the component model in OpenRTM and OPRoS is less semantically rich and explicit than the BCM's "5Cs".

1.3 The metamodelling paradigm from Model Driven Engineering

As major short-term ambition, the BCM wants to introduce into the robotics community the *mindset* that it is worthwhile to provide fully formal models for *all* of a systems' constituents (that is, the "5Cs"), as well as an explicit model-driven engineering *development process*, with support of models and development workflows in large-scale toolchain ecosystems such as Eclipse. The motivation for this approach is the success that *model-driven engineering* [1, 2] has seen in domains where *industry* (and not academics) is driving the large-scale software development: the paradigmatic belief is that complex systems can only be developed in a maintainable and deterministic way if they are first modelled, analysed and verified abstractly, and only then code in programming languages is generated. The robotics domain is not that far yet, mostly because of the attitude of software developers that they can produce code faster and better in their favourite programming language than via the "detour" of formal models. This observation is, in practice and in the short term, very often a valid one, because model-driven engineering requires a lot more toolchain support than a good programming language compiler.[2] But as soon as the critical developments of (i) model definitions, (ii) model-to-text code generation, and (iii) MDE toolchain support for all phases of the development process (including the currently poorly supported phases of deployment and runtime coordination!), will be realised, the overall development process is expected to speed up with an order of magnitude.

Figure 1.1 gives an overview of the paradigm of *metamodelling*, which has its origin in the *Model Driven Engineering* (MDE) domain of software engineering, aiming primarily at improving the process of *generating code* from abstract models that describe a domain. The MDE terminology for going from a higher to a lower level of specificity or detail in the knowledge of a domain is: from *platform-independent* to *platform-specific*, by adding the *platform knowledge*.

The Object Management Group [22] is the main driver of standardization in this context of Model Driven Engineering, for which it uses the trademarked name *Model Driven Architecture*. A huge software effort is being realized in the Eclipse project ecosystem[3] to support all aspects of MDE.

The four **levels of platform abstraction** of Figure 1.1 have, in this document's context, the following meaning:

M3: the highest level of abstraction, that is, the model that represents all the *constraints*, or *restrictions*, that an model has to satisfy without encoding any *specific* knowledge in a domain. The Eclipse Modelling Framework consortium[4] has standardized the M3 level via its *ECore* metametamodel language.

M2: the level of the so-called (in MDE speak) *platform-independent* but already *domain-specific* representations, introducing modelling primitives with concrete names,[5] concrete composition relationships, and concrete constraints, and with semantics that conform to (the much smaller) set of relationships and constraints in the M3-level model.

In the MDE paradigm, an M2 model can be represented in a so-called *Domain Specific Language*, which represents the knowledge of the properties and relationships in the

[2] Hence, the BRICS project is working on an Eclipse-based toolchain, BRIDE, [7]. Deeper discussions about this toolchain are beyond the scope of this document.

[3] http://www.eclipse.org

[4] www.eclipse.org/emf

[5] Those names are "natural" for the practitioners of the specific domain in which the M2 metamodel is introduced.

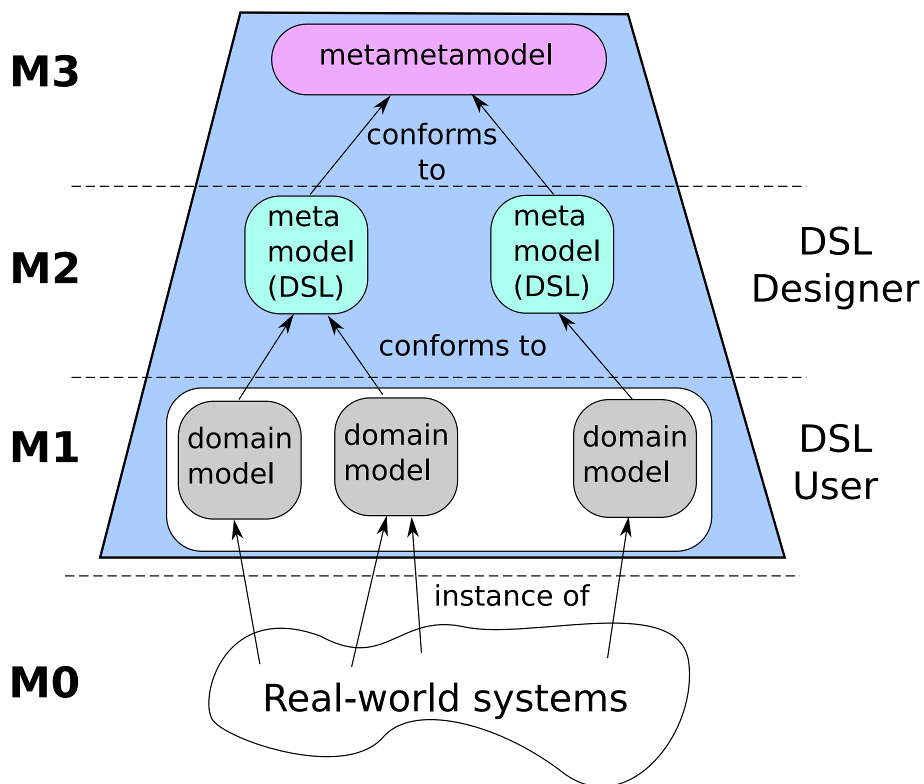

Figure 1.1: The four *levels of platform abstraction* in OMG's metamodelling standard for Model Driven Engineering, illustrating the role of Domain Specific Languages (DSL).

model with a terminology and syntactical constructs that practitioners in the domain are familiar with.

M1: the level of a so-called *platform-specific model*. That is, a concrete *model* of a concrete robotic system, but without using a specific programming language.

M0: the level of an *implementation* of a concrete robotic system, using software frameworks and libraries in particular programming languages.

Note that Figure 1.1 has *"conforms to"* relationships between the *modelling* levels, and an *"instance of"* relationship between the concrete system *model* and its concrete *implementation*. The fundamental differences between both relationships are explained well in [2].

2 The BRICS Component Model as harmonizing metamodelling paradigm

The major contribution of the BRICS project to its *harmonization ambition* is that it applied the general metamodelling ideas of Fig. 1.1 to the domain of robotics, in the specific and innovative way illustrated in Fig. 2.1:

- the M3 level is the *Component-Port-Connector* model (Fig. 2.2). This model is universally present (but most often only implicitly!) in all robotics software designs, not just for modelling component-based systems but also in many functionality libraries (control systems, Bayesian networks, Bond Graphs, etc.). This metametamodel is not unique to robotics (hence, it is really a *meta*metamodel) but it is relevant in many engineering systems contexts where sub-systems interact with each other through well-defined interaction points ("ports").
- the M2 level brings in robotics-specific domain knowledge, in the sense of the BCM (BRICS Component Model). This BCM is a formal representation of many of the implicit ideas and concepts that are already in use in the popular robotics software frameworks (hence, its *harmonizing* role!), so it is rather straightforward (but still most often tedious. . .) to provide *formal* models to these frameworks.
- The M1 level represents concrete models of concrete robotic systems. The important difference with M3 and M2 is that, at this M1 level, toolchain support is very important to support developers in the complex tasks of concrete system design or component development. A major design issue to improve "user friendliness' is the ability of such a toolchain to provide the above-mentioned models with terminology that is familiar to the robotics developer community.

The BRICS project supports M1 via its BRIDE toolchain (Work Package 4), but this is not the focus of this Deliverable. The following sections give more details about the core contributions of this document, namely the level M3 and (especially) M2.

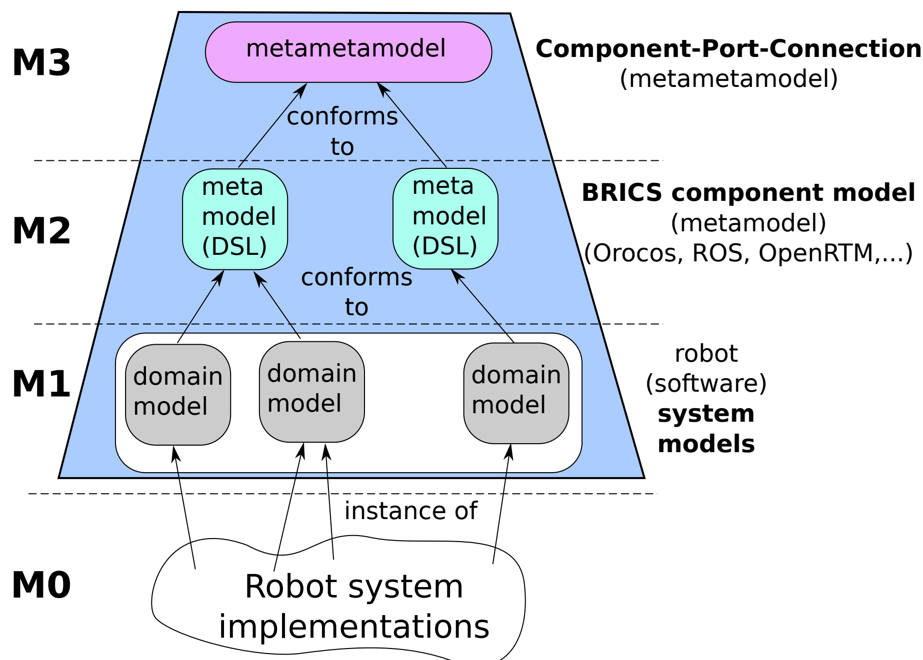

Figure 2.1: The BRICS metamodel.

2.1 M3: The Component-Port-Connector metametamodel

This is the simplest, most generic model introduced in this document. (It is very similar to the Lightweight CORBA Component Model, which we consider as a "best practice" in this context.) The Component-Port-Connector metametamodel (CPC) has the following parts:

- *Modelling primitives:* The Components (`C_A, C_B` in Fig. 2.2) provide the *containers* of functionality and behaviour, while the Ports (`pa, pb`) give localized and protected access to the Components' internals.
- *Composition rules*: The Connections (`I_1`) represent the interactions between the functionalities and behaviours in two Components, as far as accessible through the Components' Ports.
- *Constraints*: not all compositions make sense, e.g., connecting three Ports to each other directly. Here is a (not yet exhaustive) list of composition constraints:
  - Connections form a *graph*, with Ports *always* inbetween Components and Connections.
  - a Component contains zero or more Components.
  - a Component can be contained in only one Component (different from itself).
  - a Component contains zero or more Ports.
  - a Port belongs to one and only one Component.
  - a Connection is always between two Ports within the same composite Component.

At the time of writing, no final choice has already been made about (i) *the* constraint set that should be standardized, or (ii) the formal language in which the constraints will be expressed.

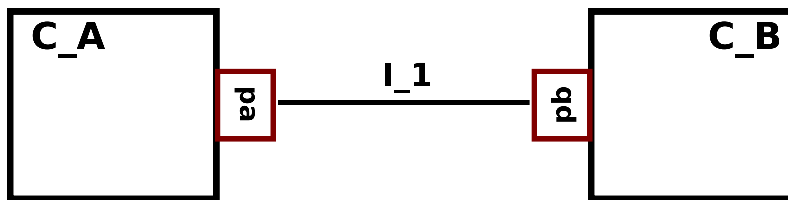

Figure 2.2: The Component-Port-Connector metametamodel.

2.2 From M3 to M2: the "5Cs"

In general, the M2 level adds specific semantics that are relevant in a particular domain; for robotics, Figure 2.3 gives an overview of the "5Cs" underlying the *separation of concerns* [9] that motivates the relevance of our BRICS Component Model (BCM) for the harmonization of all robotics systems designs:

- four types of Components (more details follow later): *Computation* (encapsulating the *useful functionality* within the system), *Coordination* (encapsulating the *discrete behaviour* of the system), *Communication*, and *Configuration.*
- *Services* on top of Connections and Ports:
  - uni-directional Communication between *two* Ports,
  - with various handshake protocols, buffering,. . . , and
  - with an *IDL* (Interface Description Language) to model the service.

  This model is a formal representation of the semantics and syntax of the data that is "sent" over a Connection and through Ports; as in the Lightweight CCM, three types of Communication are identified as indispensable: method calls, data flow, and events.

This document only focuses on the semantics of the 5C model (and less on the models of Services, Connections and Ports), because that is exactly the place where the BCM is scoring best in its *harmonization* ambition:

- Computation: this is the core of a system's functionality, and it implements (in hardware and/or software) the domain knowledge that provides the real added value of the overall

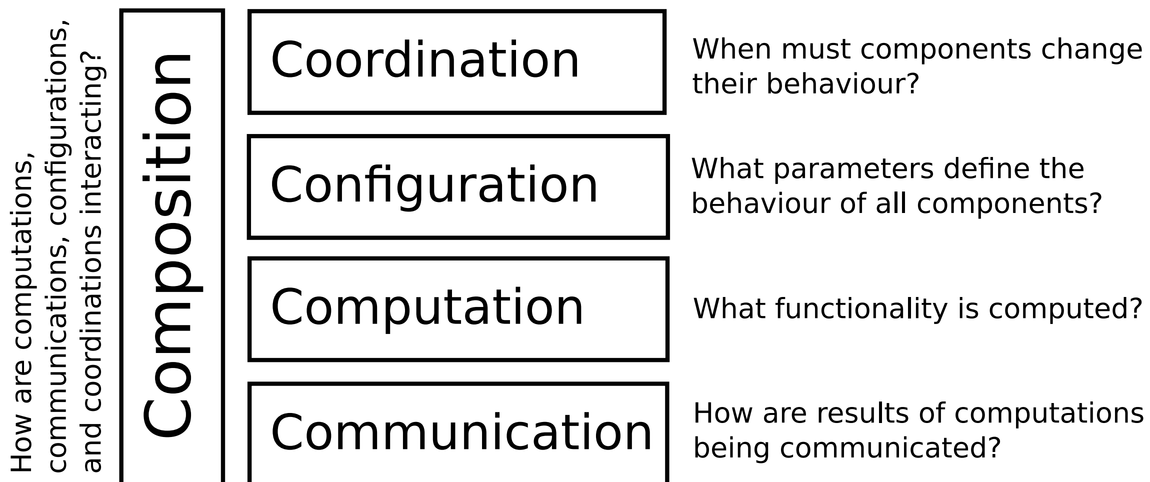

Figure 2.3: Overview of the "5C" *metamodelling* paradigm.

system. This typically requires "read" and "write" access to data from sources outside of the component, as well as some form of synchronization between the computational activities in multiple components.
- Communication: this brings data towards the computational components that require it, with the right quality of service, i.e., time, bandwidth, latency, accuracy, priority, etc.
- Coordination: this functionality determines how all components in the system should work together, that is, in each state of the coordinating *finite state machine*, one particular behaviour is selected to be active in each of the components. In other words, Coordination provides the *discrete behaviour* of a component or a system.
- Configuration: this functionality allows users of the Computation and Communication functionalities to influence the latters' behaviour and performance, by giving concrete values to the provided configuration parameters; e.g., tuning control or estimation gains, determining Communication channels and their intercomponent interaction policies, providing hardware and software resources and taking care of their appropriate allocation, etc.
- Composition: while the four "Cs" explained above are all motivated by the desire *to decouple* the design concerns as much as possible (while avoiding to introduce too many separate concerns!), the design concern of Composition models the *coupling* that is always required between components in a particular *system*. Roughly speaking, a good Composition design provides a motivated trade-off between (i) *composability*, and (ii) *compositionality*. The former is the property of individual *components* of being optimally ready *to be reused* under composition; the latter is the property of a *system* to have predictable behaviour as soon as the behaviours of its constituent components are known.

This 5C model is an extension of the "4Cs" of the seminal work [24], by separating their "Configuration" idea in the semantically more fine-grained and specialised "Composition" and "Configuration" aspects of the BCM. The following Chapter illustrates the hitherto probably still rather theoretical descriptions of the "5Cs" with a concrete set of design guidelines, but first we explain where in the overall development process developers should focus on each of the different "Cs".

2.3 The BCM and the Development Process

Neither the BRICS Component Model, nor the Development Process were anticipated in the Description of Work of the BRICS project, but they were created gradually during the first years of the project. That creation was inspired by the hard-felt need to provide more and more

structure to developers, supporting them during the long journey from initial ideas about a system's functionality, to the design of the components needed in the system, and eventually to a working and deployed system. The Development Process was explained in a lot more detail in a previous Deliverable,[1], from which we here extract the most relevant *phases*:

- *functional design*: the functional architecture is created, using functional components with interactions via ports and connectors, but without much detail about the software design of those components and their interactions.

  Only the *structural part* (that is, the CPC metametamodel) of the BCM is relevant in this phase; but the BCM does not really bring such a lot of added value with respect to existing traditions in this context.
- *component design*: in this phase, developers focus on the design and implementation of each of the functional components identified and documented in the functional design phase.

  The BCM becomes highly relevant in this phase, for multiple reasons: (i) it makes developers aware of the different types of components,[2] (ii) it provides "best practices" for each of these types (see Chap. 3), and (iii) it harmonizes their design such that integration into a toolchain becomes easier.
- *system design*: in this phase, *Composition* is the major focus of the system developer, and, again, the BCM provides harmonization, structure, and best practice information.
- *deployment*: in most robotics projects, developers most often (implicitly) assume that their development work is over after the system design phase, that is, as soon as all the code compiles, links and can be executed. However, experience with modern robotic systems has already made clear that it is not a trivial job to start up large amounts of components in a deterministic way, and without a lot of manual tuning and ad hoc scripting.

  The BCM is very relevant in the deployment phase, mainly because it motivates developers to separate the *Configuration* and *Coordination* aspects of components from their *Computational* aspects. For example, the BCM foresees a *life cycle state machine* for each component, with *initialization*, *configuration* and *operational* behavioural states, which allows developers to start components one by one, configure them one by one, and only make them provide their functional behaviour when all the other components are also ready to do so.
- *runtime coordination*: modern and future robotic systems will have to run "for ever", at least, compared to the rather short demonstration life times of current service robotics "applications". So, system and component developers will have to make their components and systems a lot more ready to be re-configured, re-connected, re-placed,..., multiple times during the lifetime of the system, and preferably without having to shut down the system. This feature is not at all present in the existing frameworks, but will be essential for providers of service robotics products and services to keep their maintenance and updating costs under control: it does not make economical sense to have to send a service engineer to every robot system deployed in the field, just to make a software update.

The impact of the BCM in these different phases in the Development Process is not limited to the *design of the components* itself, but it also brings a lot of structure to the development of the *toolchain*. More in particular, it has become very clear during the runtime of the BRICS project that none of the existing toolchains (in robotics and elsewhere) provides a granularity of tools that fits perfectly to the above-mentioned phases of the development process. This insight, in combination with the documented structure of the BCM, has allowed the BRICS project to

[1] G. K. Kraetzschmar, A. Shakhimardanov, J. Paulus, N. Hochgeschwender, M. Reckhaus. *Specifications of Architectures, Modules, Modularity, and Interfaces for the BROCRE Software Platform and Robot Control Architecture Workbench.* BRICS Deliverable D2.2, August 2010.

[2] Computation, Coordination, Communication, Configuration.

have an *harmonizing impact* on that part of the community that is developing tools for the major robotics frameworks. This has become very clear in community-wide workshops around sofware engineering for robotics, such as, for example, on the *European Robotics Forum.* And it has, obviously, provided very concrete inputs to the activities in Toolchain Work Package 4.

3 Best practices — Toolchain guidelines

This Chapter provides a list of what the BRICS project partners have learned as "best practices" in applying the "5Cs" to real robotics software designs. The list is not exhaustive, nor is it discussed in deep detail. The best practices are presented in the "5C" structure, complemented with some more general "best practice" design guidelines. Since the BRICS Component Model is the structured foundation behind the BRIDE toolchain developed in Work Package 4, these guidelines can be integrated into that toolchain.[1]

The fact that such a condensed and structured description can be given for a lot of design and development guidelines is another proof of the harmonization power of the BCM. More in particular, it refers to the "good news" of having a modelling paradigm available (Sec. 1.1), in the sense that a shared set of semantic primitives is extremely efficient in conveying ideas and insights between practitioners.

3.1 Coordination

For typical software developers in the robotics community, Coordination is often the least familiar design aspect. Because Coordination models the discrete behaviour of a system, it is typically implemented as a Finite State Machine (FSM), and in general one should foresee one at *each* level of composition. Each state of a Coordination FSM corresponds to a specific Configuration of the behaviour of, and Connection between, the other Components in the system. Each Component should always have a *life-cycle FSM*, which allows the creation, (re)configuration, and finalization of a component. Reusability of a Coordination component is optimized when its content is restricted to *pure event processing*, that is, first order logic, probabilistic state machine, etc. One important observation to understand why robotic systems design is typically more complex than in other engineering domains, is that such systems typically have not only one *nominal* behaviour, but one needs to give them *robust behaviour* against lots of "disturbances" from the real world, and from interactions with other components. Such robust behaviour typically requires *a lot* more Coordination (in number of events as well as in behavioural system states) than the nominal behaviour. This also means that a lot of added value in a robot system is created by the quality of its Coordination component(s).

3.2 Composition

The robotics community is still not completely aware of the motivations to go from *hierarchical* compositions to *peer-to-peer* compositions. A guideline in this context is that one must avoid to let one "master" component kill multiple of its "slave" components, and instead let components limit themselves to sending out "termination of service" *events*, which are to be taken as serious advice (but nothing more!) in all other components that are *configured* to react to these events.

Another guideline in Composition is to realise the difference between "functional" and "component" architectures. The BCM is only dealing with the latter, and hierarchical (sub)architectures are only introduced when adding more specific *platforms constraints*; e.g., a specific kinematic model, controller, sensor, fieldbus, etc. At each level of composition, the developer should be aware that it often makes sense to introduce explicit *data structures* to support Coordination and Composition; in software frameworks that have no explicit 5C model, these data structures are often "hidden" in computational components, which makes those components (as well as the Coordination behaviour that they hide) more difficult to reuse.

[1] This integration has currently not yet been achieved.

Knowledge about components should only reside in the head of the Composition developers, and not in the Components themselves. But components should provide a *model* of their own behaviour and Ports in their configuration interfaces. This "introspection" of a running component allows other components *to configure* themselves *at runtime* to interact with that component, based on that semantic model. Indeed, developers should not forget that composition is not just a compile time responsibility, but that it also appears at deployment time and at runtime.

3.3 Computation

Various types of Computational models exist: *function blocks* (a.k.a. functional programming, or data flow programming); *Bond Graphs* (for the simulation and control of physical system that exchange energy); message passing over graphical models (for perception); and many others.

Current robotics developers seldom provide pure computational components, mainly they often mix them with application-specific Coordination, Configuration, Communication, and Composition. Because of its most limited support for a component model, ROS is a major "worst practice" in this context; or rather, many *developers* use the ROS component framework without much attention to the "5C" separation of concerns.

3.4 Configuration

One should provide one Configuration component to each Component, since configuration should take place *synchronously* with the configured Component, when the latter is in its Configuration state. Configuration should also be *atomic*, as observed by all other components. It is the Coordinator's responsibility to make sure that it triggers events (i) to bring components in their Configuration state, and (ii) to activate the Configuration components "to do the right thing" at the right moment. This guideline separates the actual configuration from the Coordinator component, since it does not have to know anymore *how* to configure the components. In other words, a Configuration component "shields" a Computation component from the Coordinator, in that the latter does not have to be aware of the computational state every component in the system under coordination is in at each moment in time.

3.5 Communication

Three Communication types (as in the Lightweight CORBA Component Model standard) are required:

- data: between components in a Computational composite.
- event: for Coordination.
- service requests: for Configuration and for data exchange between Components within the same composite.

Most component based software framework implicitly create a lock-in into the Communication policy supported by the framework, and this limits the reuse of its components in systems with other policies.

The biggest need, in each application domain, and in the context of Communication, is that of *standardization* of the *data structures* that components communicate between each other. The robotics domain scores extremely poorly in this respect; e.g., even the most fundamental geometric primitive of a *frame* has not yet been standardized. . .

The robotics domain is also achieving poorly when it comes to adopting Communication "middleware" frameworks developed in other domains, as illustrated by the dozens of "yet another commmunication middleware" projects that have been created by robotics developers in the last decade.

3.6 Miscellaneous

One component must not necessarily be one process. In other words, there is no compelling need *to deploy* one unit of computational component into one unit of computational resources.

Most developers in robotics make the error *to name* their components, variables, events, etc. after the *purpose* they serve in their current application, instead of after what they *are* or *do*. For example, try to avoid the name of "error state" in a Coordination FSM, since what is an error in the sub-system at the time of development of the FSM might be an expected state in a later larger-scale composition. So, the most appropriate name for a state is one that represents the *behaviour* that the system is providing when in that Coordination state.

Assigning a separate component to each *shared resource* (hardware, data, environment,. . .) is advantageous, since it allows to let that component, and only that component, coordinate what happens to the shared resource.

4 Conclusions

The ambition of the BRICS Component Model is to introduce models as *first-class citizens* in the robotics software development process. The major reasons are (i) providing explicit and formal models in a domain helps developers *to structure* their design efforts and outcomes, and (ii) code in concrete programming languages can then be *generated* from the models via a (semi)automatic toolchain ("model-to-text" transformations, in MDE speak), instead of hand-crafted. So, while code generation from models is essential in the ambition of the BCM, its current state of practice is still rather elementary in this context. However, the concepts of the BCM have already been tried on more than one hundred developers (PhD students, and academic and industrial robot software engineers) on various occasions (research camps organized by the BRICS project; targeted dissemination workshops with selected developers; public workshops on the European Robotics Forum), and the outcome is always positive: these developers quickly get concrete "best practices" from the BCM paradigm, even without full toolchain support or standardization of the "5C" models. This fast "transfer" of concepts and insights is a major results of the harmonization power of the BCM.

Currently, the BCM concepts can be applied successfully in the design and development of *new* components in existing "non-BCM-based" software frameworks such as ROS or Orocos, by developers that are sufficiently disciplined to map the BCM and its best practices onto the available component primitives in the frameworks. However, the other way around will most often fail; that is, it is typically impossible to make a 5C model out of an existing ROS or Orocos system, because those frameworks are not yet supporting their users to use the 5Cs in a systematic way. A major problem in ROS and Orocos is that they have no explicit primitive for Composition. And here the second harmonization power of the BCM is beginning to have impact, since developers in ROS and, especially, Orocos and OpenRTM have started to adopt many of the BCM ideas, suggestions and terminology.

Finally, it often turns out that, when a system is designed according to the BCM, most of the added value in each system is concentrated in the Coordination and Configuration components, while the Computations and Communications can most often be reused from existing software projects. (Often with some necessary but rather straitghtforward refactoring to cleanly separate the 5C concerns.)

5 Acknowledgements

The following people have contributed to the material presented in this Deliverable, via a very effective multi-Work Package *Task Force*: Herman Bruyninckx (KUL), Nico Hochgeschwender (BRSU), Markus Klotzbücher (KUL), Peter Soetens (KUL), Gerhard Kraetzschmar (BRSU), Davide Brugali (UBergamo), Hugo Garcia (KUL), Azamat Shakhimardanov (BRSU), Jan Paulus (BRSU), Michael Reckhaus (BRSU), Luca Gherardi (UBergamo), Davide Faconti (KUL).

Bibliography

[1] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[2] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[3] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Anders Orebäck, and Stefan Williams. Towards component based robotics. In *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 163–168, 2005.

[4] Herman Bruyninckx. Open RObot COntrol Software. http://www.orocos.org/, 2001.

[5] Herman Bruyninckx. Open robot control software: the OROCOS project. In *IEEE Int. Conf. Robotics and Automation*, pages 2523–2528, 2001.

[6] Autosar Consortium. Autosar—AUTomotive Open System ARchitecture. http://www.automationml.org, 2003.

[7] BRICS Consortium. BRIDE—the BRIcs Development Environment. http://www.best-of-robotics.org/bride/, 2012.

[8] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classifcation framework for software component models. *IEEE Trans. Software Engineering*, 37(5):593–615, 2011.

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

[10] Eclipse Foundation. The Eclipse Integrated Development Environment. http://www.eclipse.org.

[11] Sara Fleury, Mathieu Herrb, and Raja Chatila. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 842–848, 1997.

[12] Sara Fleury, Anthony Mallet, Matthieu Herrb, Séverin Lemaignan, Félix Ingrand, and Raja Chatila. GenoM: the Generator of Modules. www.openrobots.org/wiki/genom.

[13] Brian Gerkey, Richard Vaughan, Andrew Howard, and Nate Koenig. The Player/Stage project. http://playerstage.sourceforge.net/, 2001.

[14] Groupe de Recherche en Robotique. Proteus: Platform for RObotic modeling and Transformations for End-Users and Scientific communities. http://www.anr-proteus.fr/.

[15] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. Comparison of component frameworks for real-time embedded systems. In *13th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2010)*, pages 21–36, 2010.

[16] Korean Institute for Advanced Intelligent Systems. OPRoS. http://opros.or.kr/.

[17] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, and Félix Ingrand. GenoM3: Building middleware-independent robotic components. In *IEEE Int. Conf. Robotics and Automation*, pages 4627–4632, 2010.

[18] Raffaela Mirandola and František Plášil. CoCoTA—Common Component Task. In Raffaela Mirandola Andreas Rausch, Ralf Reussner and Plášil František, editors, *The Common Component Modeling Example. Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 4–15. Springer-Verlag, 2008.

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

[20] OASIS. Service Component Architecture. http://www.oasis-opencsa.org/sca.

[21] Object Management Group. Modeling and Analysis of Real-time and Embedded systems. http://www.omgmarte.org.

[22] Object Management Group. OMG. http://www.omg.org.

[23] Open Management Group. CORBA: DDS for Lightweight CCM. http://www.omg.org/spec/dds4ccm/.

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

[25] Christian Schlegel. SmartSoft: Components and toolchain for robotics. http://smart-robotics.sourceforge.net/.

[26] Christian Schlegel and Robert Wörz. The software framework SmartSoft for implementing sensorimotor systems. In *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 1610–1616, 1999.

[27] Society of Automotive Engineers. AADL: The SAE Architecture Analysis and Design Language. http://www.aadl.info.

[28] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[29] Hans Utz, Stefan Sablatnög, Stefan Enderle, and Gerhard Kraetzschmar. Miro—Middleware for mobile robot applications. *IEEE Trans. Robotics and Automation*, 18(4):493–497, 2002.

[30] Willow Garage. Robot Operating System (ROS). http://www.ros.org, 2008.