Abstract—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 paper introduces the BRICS Component Model (BCM) to provide developers with putting as much structure as possible in their development of, both, individual components and systems of components, using one or more of those frameworks at the same time, 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. The material in this paper is presented with as ideal reader in mind a 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.

I. 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 [4, 5], OpenRTM [19], ROS [30], OPRoS [16], Genom [12, 17, 11], 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 calls.

This paper introduces the BRICS Component Model1 (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 Section of this paper 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 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 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 reproducible way. This is the reason why the BCM creators put much emphasis on the complementary roles

1Named after the European research project BRICS, Best Practice in Robotics, (Grant number FP7-231940) in which it is developed.
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 that 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 (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 and results. And to realise that their difficulties are caused by their thinking inside different paradigms in the first place. Hence, this paper should be read with a mindset that already accepts the paradigm of model-driven engineering as meriting its legitimate place in robotics (which is not at all obvious in the state of the practice of most current robotic software developments!), but that the community is currently still exploring what is the best way to realize it. This paper’s ambition is to explain its readers in what ways the BRICS Component Model contributes to this exploration.

II. RELATED WORK

At the time that the robotics community became aware that it makes sense to spend time developing reusable software frameworks, 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, ROS has conquered most of the community 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 paper, it is farthest away from the CCM. The BCM introduced in this paper shares its structural model with that of the CCM (components, interacting via methods, data streams and/or events, and composed by ports together), but adds to the CCM four complementary types of components, with clear semantic meaning in the context of robotics to guide developers in separating the functionality of their applications in reusable parts.

It shares this 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 [22]), 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 the BCM’s “5Cs” (Sec. IV) 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 Sec. IV, 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 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”.

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, analyses 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. 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 reconfiguration!), will be realised, the

---

2This was around 2000, with GenoM [12], Player-Stage [13], Miro [29], Orocos [4] and SmartSoft [26] as pioneers.
overall development process is expected to speed up with an order of magnitude.

Figure 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 [21] 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\(^4\) to support all aspects of MDE.

![Fig. 1. The four levels of platform abstraction in OMG's metamodelling standard for Model Driven Engineering, illustrating the role of Domain Specific Languages.](image)

The four levels of platform abstraction of Figure 1 have, in this paper’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\(^5\) has standardized the M3 level via its ECore metametamodel language.

**M2**: the level of the so-called (in MDE speak) platform-independent representation of a domain, introducing models with concrete names and concrete relationships (conforming to the M3-level constraints).

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 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.

**Overview of the paper.** Section III explains how metamodelling can be introduced in robotics, with the envisaged balance between concrete formal structure and simplicity. Then, Section IV introduces the core of the BCM, the so-called “5Cs” that help developers with a concrete formal component model. Some “best practices” of using the BCM are briefly introduced in Section V, and the paper concludes with a discussion of its core contributions in Section VI.

III. THE BRICS METAMODEL

The general metamodelling ideas of Fig. 1 are applied to the domain of robotics as illustrated in Fig. 2:

- the M3 level is the Component-Port-Connector model (Fig. 3). 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 metametamodel) but holds 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, 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, but this is not the focus of this paper. The following subsections give more details about the core contributions of this paper, namely the level M3 and (especially) M2.

A. **M3: The Component-Port-Connector metametamodel**

This is the simplest, most generic model introduced in this paper. (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. 3) provide the containers of functionality and be-
How are computations, communications, configurations, and coordinations interacting?

- 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.

By lack of space, this paper only focuses on the 5C model, and not on the models of Services, Connections and Ports.

- 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 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 configured to be active in each of the components. In other words, Coordination provides the discrete behaviour of a component or a system.

IV. FROM M3 TO M2: THE “5CS”

The M2 level adds specific semantics that are relevant in a particular domain. For robotics, Figure 4 gives an overview of the “5Cs” underling the separation of concerns [9] that motivates our BRICS Component Model (BCM):

- four types of Components (more details follow later): Computation (functionality), Coordination (discrete behaviour), Communication, Configuration.

- Services on top of Connections and Ports:
  - uni-directional Communication between two Ports.
  - with various handshake protocols, buffering, . . . , and

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.

From Fig. 2, the BRICS metamodel.

Fig. 2. The BRICS metamodel.

haviour, while the Ports (p_a, p_b) 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 in between 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.

Fig. 3. The Component-Port-Connector metamodel.

Fig. 3. The Component-Port-Connector metamodel.

Fig. 4. Overview of the metamodeling paradigm.
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 more fine-grained “Composition” and “Configuration” aspects of the BCM. The following Section illustrates the still rather theoretical descriptions of the “5Cs” with a concrete set of design guidelines.

V. BEST PRACTICES

This Section provides a list of what the authors 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, due to the lack of space. The best practices are presented in the “5C” structure, complemented with more general “best practice” design guidelines.

**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 lifecycle 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).

**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.

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.

**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.

**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.

**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 communicative middleware” projects that have been created by robotics developers in the last decade.

**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.

### VI. 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; or targeted dissemination workshops with selected developers), 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.

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.

This paper’s focus is on only the component development and system composition phases in the overall development process of complex robotic systems, hence it did not talk about the other important phases, such as packaging, deployment and runtime reconfiguration.

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.

### ACKNOWLEDGMENTS

The authors acknowledge the support from the KU Leuven Geconcerteerde Onderzoeks-Actie Global real-time optimal control of autonomous robots and mechatronic systems, and from the European project BRICS (FP7-231940, Best Practice in Robotics).

### REFERENCES


