BCM: A Minimal Robotic Component Model for Multitarget System and Component Generation

Markus Klotzbücher, Peter Soetens, Herman Bruyninckx
Department of Mechanical Engineering, Katholieke Universiteit Leuven
Celestijnenlaan 300B, 3001 Leuven (Heverlee), Belgium

Abstract— **Component based software engineering has become the paradigm of choice for constructing complex and reusable robotic systems. Unfortunately interoperability between frameworks is nonexistent in spite of all frameworks sharing many common semantical concepts. We present a minimal and extensible component model used to abstract the core concepts of today's robotics platforms and outline an approach for transforming platform independent models of components and systems to multiple robotic frameworks.**

I. Introduction

Component based software engineering has become the standard approach for building robotics systems, as can be seen by the multitude of component based frameworks [8], [7], [1], [3]. Although first standardisation efforts have been undertaken [5] components which are developed for one of these frameworks are unable to be used with an other. As a consequence developers are required to commit themselves to a particular platform at a development stage in which the needs of the final application can hardly be foreseen. Furthermore reuse of components is limited and requires tedious manual refactoring of low-level code.

We propose a Model Driven [4] approach to solve this problem by modelling components and systems in a platform independent manner. These models can then be transformed to various target frameworks as shown in figure 1. This way the same component can be reused for multiple targets and the user is freed from having to decide at an early development state which target platform to use.

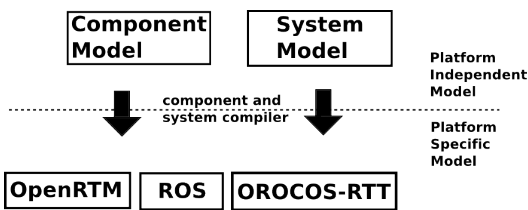

Fig. 1. Overview of component and system transformation

The rest of this paper is structured as follows. Chapter II describes the elements of our component model and their relationships among each other as ECore/UML diagrams. Chapter III describes how parts of the injected code are preprocessed in order to function correctly for the respective target platform. Chapter IV describes how data types are modelled and generated. We conclude and discuss future work in chapters V and VI.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).
markus.klotzbuecher@mech.kuleuven.be

II. Model Elements

This section describes the elements of our minimal component model. The model is minimal in the sense that it exclusively contains model elements which are used for transformation and code generation purposes and which are relevant to a majority of target frameworks. The model elements are modelled using the Eclipse ECore modelling language [2].

All model elements can be subdivided into two classes reflecting two major phases during development of a robot system. During the component design phase a single component is developed. This requires defining which interaction primitives will be used for communicating with other components, which platform independent computations are to be executed when and which configurable properties the component will offer. It is important that at this stage no assumptions are made about how the component will be used later.

The second phase is the system design phase. During this phase existing components are instantiated, configured and their interconnections are defined. The result is a model of a system which can, after transformation, be executed in order to establish a running system.

A. Component design phase

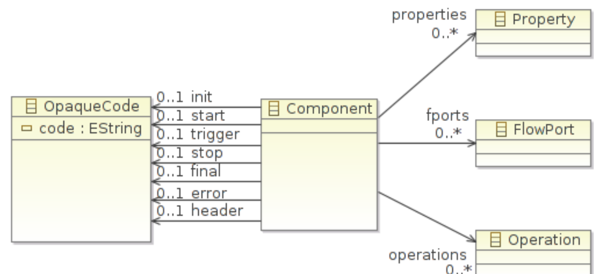

Fig. 2. Component

1) Component: A component is the central model element of a component model and the building block of software architectures. A component has containment relationships to its *primitives* which define its structure.

For the BCM component model these primitives consist of Properties, FlowPorts and Operations which are explained in more detail below.

As can be seen on the left hand of figure 2 a component is associated with a set of `OpaqueCode` entities. These are the actual functional computations of the component which are inserted into the generated code at the respective positions by the component compiler. When this opaque code is executed is defined by a generic component life-cycle state machine. This state machine is shown as a hierarchical UML state machine model in figure 3.

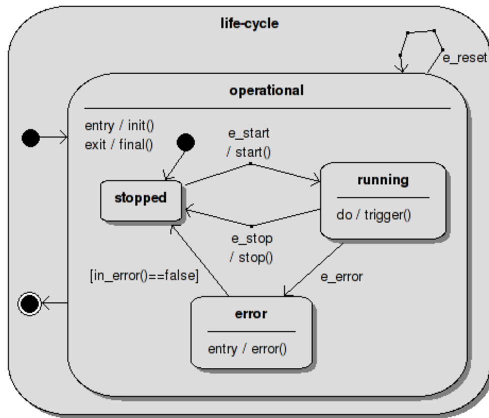

Fig. 3. Component life-cycle State Machine

Any modeled component begins its life by entering the operational mode composite state and thereby executing the OpaqueCode `init()` which typically is used to prepare the component for operation, allocate resources etc. After this the component enters the stopped state in which it resides until started by the user. In the figure 3 the transition to `running` is taken when the event `e_start` is received.

In the `running` state the OpaqueCode `trigger()` is executed periodically or aperiodically, depending on the configured mode of the component.

The `error` state is entered in case of an unhandled error of the component. The OpaqueCode `error()` can be defined in order to attempt to recover from this condition. In case of success the state machine transitions back to the `stopped` state. In case of failure the event `e_reset` will be raised which will cause an *external self-transition* to the `operational` state which will result in reinitalization of the component by first executing the OpaqueCode `final()` followed by `initial()`.

Obviously not all robotics frameworks use exactly this life-cycle state machine. Fortunately this does not pose a problem. If a framework provides more states these are simply left unused or mapped to states of the generic state machine. If less states or no life-cycle state machine is provided at all the component generator automatically generates this simple state machine for the given component.

Fig. 4. Property

2) Property: A Property (see figure 4) is a configurable parameter of a component which can be get and set through a well defined interface. Properties are commonly used during system design phase to configure or fine tune the behavior of components. A property is always of exactly one `Type` which defines the structure of the Properties data.

Properties can also be set as read-only for indicating that the value is not to be changed currently.

Not all framworks provide Properties as explicit primitives. For these it is straightforward for the code generator to simulate these by generating get and set Operations.

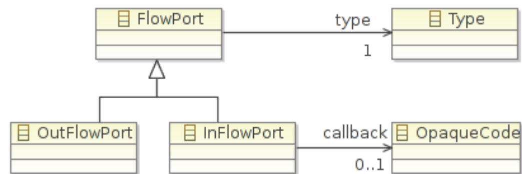

Fig. 5. FlowPort types

3) FlowPorts - InFlowPort and OutFlowPort: FlowPorts (see figure 5) are an interaction primitive used for communicating data anonymously between an OutFlowPort and an InFlowPort of two components. This primitive is named after the identically named SysML model element [6]. Communication through FlowPorts takes place without making any assumptions about the receiving side. This form of component interaction is very common in robotic applications and therefore such primitives can be encountered in almost all robotic frameworks. During the system design phase the connection between an InFlowPort and an OutFlowPort can be modelled by a FlowPortConnection.

An InFlowPort can additionally be associated with one OpaqueCode. If this association is defined the InFlowPort becomes event driven and the OpaqueCode `callback` is inserted into the event handling function.

Similar to Properties FlowPorts are always associated with a `Type` which models the transferred data.

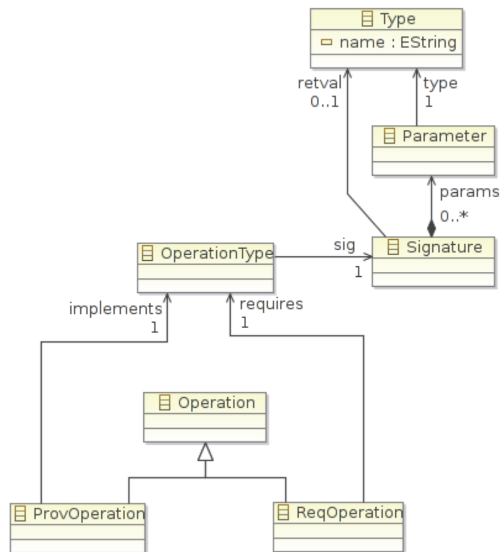

Fig. 6. Operation

4) Operation - OperationType, ProvidedOperation and RequiredOperation: Operations (see figure 6) are RPC like request-reply calls which can have zero or more arguments and a return type. Components contain zero or more provided operations (`ProvOperation`) or required operations (`ReqOperation`). The first is a functionality which can be called by other components while the latter is a functionality which is required by components to function correctly. ProvOperation and ReqOperation have an 'implement' or 'require' association to an OperationType respectively which has a function signature and attaches semantics to this signature by giving it a name. As an example a signature could be defined by `KDL::Frame(void)`. An OperationType then adds semantics to this signature by defining a name such as `KDL::Frame get_end_effector_frame(void)`.

A `ProvOperation` can be connected to a `ReqOperation` in the system design phase by means of a OperationConnection which is described in section II-B. It is important to note that in order to make such a connection it is not necessary that ProvidedOperation and RequiredOperation are of the same OperationType. This would be an impractical restriction[1] considering that components can be distributed binary only and can come from different vendors. Instead the weaker form of typing called duck typing is applied for determining the identity of an OperationType. Two OperationType models are considered identical if their name and signature are identical. Two `Signature` models are considered identical if return value, number of parameters and the respective types are equal. This weak typing guarantees much higher reusability in unanticipated circumstances.

5) Interface: An Interface is a model element which groups a set of OperationType elements into a logical unit as shown in figure 7. Components can require or provide an interface which means that the respective collection of OperationTypes are offered to other components or are required by a component in order to function correctly. Interfaces serve for two major purposes:

- as an abstraction mechanism. For example a component which implements a LaserScanner interface can be exchanged by any other component implementing this interface.
- for allowing the system designer to conveniently connect multiple ProvidedOperations with RequiredOperations.

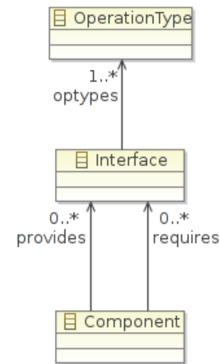

Fig. 7. Interface

The coupling between a provided Interface and ProvidedOperation elements is loose in the sense that there exists an unmodeled constraint which the model by itself can not enforce: each OperationType of an Interface which is provided by a component must also be implemented by a ProvidedOperation of the same component. The toolchain must validate this constraint before code generation and issue appropriate warnings if unsatisfied.

B. System design phase

During system design the availabe components are instantiated, configured and interconnected. This information is captured in a system model shown in figure 8.

The elements of this system model are described below.

1) ComponentInstance: A ComponentInstance is a model of a concrete instance of a Component such as the components modeled during the Component design phase. A ComponentInstance has a type association to the component of which it is an instance.

2) Connections: Connections interconnect the primitives `InFlowPort` and `OutFlowPort`, `ProvOperation` and `ReqOperation` and provided and required `Interface`. Connections are quite complex because they introduce constraints both on the ComponentInstance and the type of the instance.

For modeling the characteristics common to all connections a `Connection` superclass is used. These are associations to the source and the target ComponentInstance.

FlowPortConnection

[1]However this is often reality when using object oriented middlewares such as CORBA.

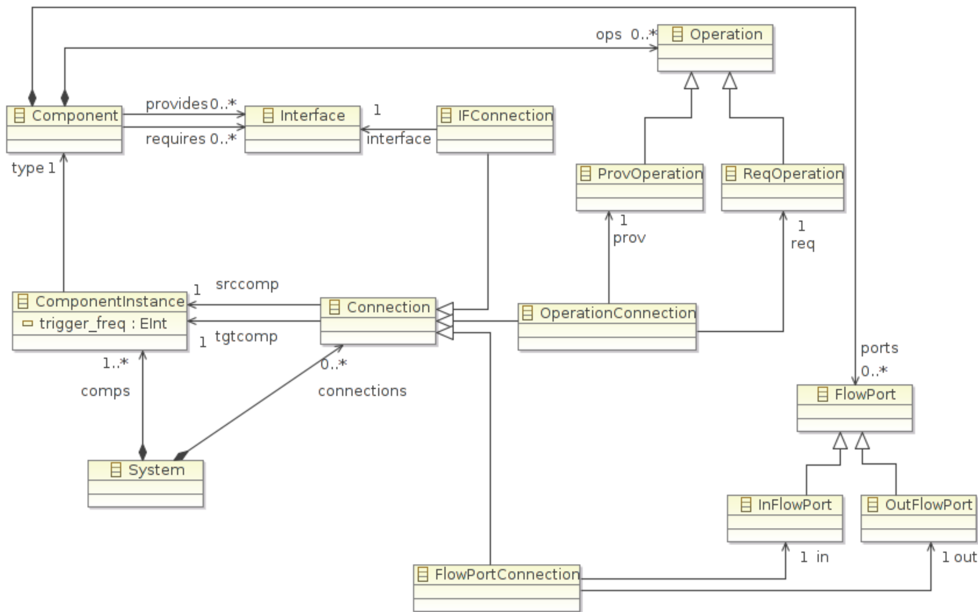

Fig. 8. System

A FlowPortConnection models the connection between an OutFlowPort and an InFlowPort. The associations `in` and `out` identify the source and target FlowPorts of the Component type (not the ComponentInstance!). The relation to the specific instance is achieved by the associations `srccomp` and `tgtcomp` of the `Connection` superclass.

An unmodeled constraint which needs to be enforced by the toolchain is that the types of the Output and InFlowPort must be identical.

OperationConnection

An OperationConnection is defined analogous to the FlowPortConnection. `prov` and `req` identify the Operations of the type which need to be connected and the `Connection` superclass provides the association with the `ComponentInstance`. Similarily to FlowPortConnections the following unmodeled constraint must be enforced by the toolchain: `OperationType` of the `ProvOperation` and the `ReqOperation` must be identical.

IFConnection

An interface connection connects a provided and a required Interface. The semantics of `IFconnection` entities are simple: the toolchain will create an OperationConnection between each corresponding ProvOperation of the source component and each ReqOperation of the target component which are part of the IFConnections interface. As the provided and required Interface is the same model element no unmodeled constraints exists here.

Obviously this type of interface is very lightweight and only distantly resembles the interfaces known from traditional programming languages such as Java. This approach has the advantage of being easily added during code generation to any robotics framework without adding dependencies on object oriented middlewares such as Corba or ICE. Secondly it gives the developer the freedom to efficiently connect Operations by IFConnections or if more control is required individually by OperationConnections.

C. Attributes and Annotations

ECore attributes are used throughout the model for various properties common to all model elements. For example a ComponentInstance defines an attribute `trigger_freq` which models the frequency at which the periodic trigger OpaqueCode shall be executed. Attributes are essential for a component to function and might be subject to further constraints.

Annotations are key-value pairs which are applicable to all model entities as a consequence of being sub-classes of the `Object` element. In contrast to Attributes Annotations are non-essential and are commonly used to enable framework specific options. It is crucial however that these options are of non-functional nature and do not impair the functioning of a component on the other target systems. This is achieved by requiring that any component must still function correctly after removing all annotations (although possibly at lower performance). An example of a annotation could be the instruction to use lock-free connections as offered by the OROCOS-RTT. Although this annotation will affect the real-time behaviour the component will still work correctly on on frameworks which do not support lock-free connections.

III. Opaque Code Processing

Unfortunately the strings of opaque code which are inserted into the component body can not be entirely opaque. This is because a component developer must use function or method invocations for using FlowPorts, Operations or Properties. For example in the opaque code one might have to write a value to an OutFlowPort or read from an InFlowPort. Of course these calls can not be of any particular platform as this would break the cross platform portability. We solved this problem by introducing a generic BCM API which must be used when interacting with component primitives in opaque code. Before model transformation takes place these primitives are then parsed and extracted into model elements of Primitive and (now truly) opaque code. During code generation these model elements can then easily be transformed to the appropriate API of the framework. As the opaque code is not completely opaque we refer to it as *semi-opaque code*.

The currently used API is shown in table I

TABLE I

The BCM opaque code API

BCM API	description
`bcm_prop_get(name)`	get value of property 'name'
`bcm_prop_set(name)`	set value of property 'name'
`bcm_fport_read(name)`	read from a flow port 'name'
`bcm_fport_write(name)`	write to a flow port 'name'
`bcm_call(name, param1, ...)`	call required operation 'name'

IV. Type generation

For successful generation of components for multiple target platforms it is crucial to model data types. Only this way it is possible to validate the connection between FlowPorts, Operations and Interfaces. Furthermore all data needs to be accessed in an uniform way in the opaque code. This is because these data accesses can not easily be preprocessed as it is done for the BCM API due to the large variety of possible access methods. These requirements can be satisfied if data structures of different target languages are generated from a model. The BCM type model which is used for this purpose is shown in figure 9.

A `Type` consists of zero to many subtypes which can be either primitive or complex. A primitive type can be one of the typical available basic C types. A `ComplexType` has a association with `Type` which allows recursive nesting of types.

This type model is heavily influenced by the concept of message description files used in the ROS framework [7].

V. Conclusions

We have presented a minimal component model which captures the core concepts of most current robotic component based frameworks. Modelled components and systems can be transformed to specific robotic frameworks without manual intervention. Two classes of model elements for two different development roles can be distinguished. During component design the structure and behaviour of an individual component is defined, while during system design existing components are connected and configured.

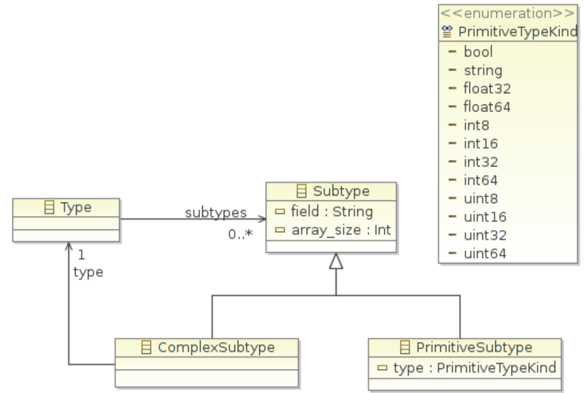

Fig. 9. BCM Type Model

An initial component compiler has been developed which can generate working ROS and OROCOS-RTT components from BCM models.

A paradoxical situation arises from the fact that a component developer modelling a component with the BCM has a much smaller set of options available than when traditionally developing for a specific target framework. For example OROCOS-RTT offers a `Command` primitive which represents an ongoing interaction between two components. This feature is not available in our component model as it is unique to this particular framework. On the other hand many target specific options can be regained by introducing appropriate annotations provided that these are of non-functional nature and do not break the component on different targets. Nevertheless it is a justified question if gaining the ability to transform a component for multiple targets outweighs the disadvantage of having to give up specific features only available on a certain target framework. In our opinion the answer is yes for a large amount of standard components and no for a small set of highly optimised, target framework specific components.

A second goal of a unifying component model is to stimulate harmonisation among different robotics frameworks by highlighting the large amount of semantic similarities. The BCM semi-opaque code API suggest a path towards a common set of interfaces, thus interoperability.

VI. Future Work

Future work will include adding further transformation targets, graphical tooling and modelling of opaque code. Currently the latter is mostly written manually during the component design process. However this code could itself be generated from a behavioural model such as Finite State Machines or an UML Activity Diagrams, thereby further reducing manual programming efforts.

An other promising aspect which will be explored is if the component compiler can be used to generate *heterogeneous*

bridging components for inter-component integration. For instance a heterogeneous component could be generated from the same model which reads a value from a ROS InFlowPort (ROS-topic) and writes these to a RTT InFlowPort.

REFERENCES

[1] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Orca: A component model and repository. In *Software Engineering for Experimental Robotics*, Springer Tracts in Advanced Robotics, pages 231–251, 2008.

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

[3] Anthony Mallet, Sarah Fleury, and Herman Bruyninckx. A specification of generic robotics software components and future evolutions of GenoM in the Orocos context. pages 2292–2297.

[4] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[5] Object Management Group. Robotics domain special interest group. http://robotics.omg.org/.

[6] Object Management Group. Systems Modelling Language (SysML). http://www.sysml.org/.

[7] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[8] Peter Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, May 2006. http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.