electronics

MDPI

*Article*

# EARL—Embodied Agent-Based Robot Control Systems Modelling Language

**Tomasz Winiarski [1,*], Maciej Węgierek [1], Dawid Seredyński [1], Wojciech Dudek [1], Konrad Banachowicz [2] and Cezary Zieliński [1]**

1 Institute of Control and Computation Engineering, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland; M.Wegierek@elka.pw.edu.pl (M.W.); d.seredynski@elka.pw.edu.pl (D.S.); wojciech.dudek@pw.edu.pl (W.D.); C.Zielinski@ia.pw.edu.pl (C.Z.)
2 nomagic Sp. z o.o., ul. Rakowiecka 36, 02-532 Warsaw, Poland; konradb3@gmail.com
* Correspondence: T.Winiarski@ia.pw.edu.pl; Tel.: +48-2223-47397

Received: 22 January 2020; Accepted: 19 February 2020; Published: 24 February 2020

check for updates

**Abstract:** The paper presents the Embodied Agent-based Robot control system modelling Language (EARL). EARL follows a Model-Driven Software Development approach (MDSD), which facilitates robot control system development. It is based on a mathematical method of robot controller specification, employing the concept of an Embodied Agent, and a graphical modelling language: System Modelling Language (SysML). It combines the ease of use of SysML with the precision of mathematical specification of certain aspects of the designed system. It makes the whole system specification effective, from the point of view of the time needed to create it, conciseness of the specification and the possibility of its analysis. By using EARL it is possible to specify systems both with fixed and variable structure. This was achieved by introducing a generalised system model and presenting particular structures of the system in terms of modelling block configurations adapted by using instances. FABRIC framework was created to support the implementation of EARL-based controllers. EARL is compatible with component based robotic middlewares (e.g., ROS and Orocos).

**Keywords:** SysML; MDSD; robot system specification; cyber-physical system

## 1. Introduction

Robotic systems become more and more complex due to the incorporation of an increasing number of sensors and effectors as well as the complexity of the executed tasks. One of the main problems of modern robotics is how to design robot controllers effectively and correctly. At the core of this problem is the formulation of a robot controller specification method. Two of the requirements are of utmost importance. The specification should be platform-independent and, moreover, it should be simple to transform into an implementation compatible with a chosen technology. There is no single solution to those requirements. The literature regarding this subject focuses on best practices in this field [1,2]. The provided set of tips, principles and proposals for structuring and representing the operation of controllers is frequently presented with the help of Model-Driven Software Development (MDSDs) techniques. This representation method helps to effectively implement those principles when constructing a robot controller. In our work we propose EARL MDSD. To present the context of the creation of EARL, in the following, other MDSDs that preceded EARL are described. Moreover, SysML and the concept of Embodied Agent are introduced, as EARL is based on them.

### *1.1. MDSD in Robotics*

In 2016, Nordmann et al. [3] surveyed 137 publications dealing with domain-specific modelling approaches in robotics. Those publications took into account problems from nine subdomains of robotics, such as kinematics, dynamics or reasoning methods. Architectures and programming is the subdomain most relevant to our subject. This subdomain was further subdivided into 12 subsubdomains such as control and handling of events, error and exception handling, design patterns or architecture design decisions. The survey shows the level of complexity and multifacetedness of the field of specification of robot controllers. Moreover, the number of papers indicates the importance of the problem.

Ramaswamy et al. [4] discuss and compare four popular robotic MDSDs: RobotML, SmartSoft, BRICS Component Model and V3CMM. In the following, the main attributes of these MDSDs are presented. In Section 5, the selected MDSDs are systematically compared with EARL.

RobotML [5,6] is used to define a metamodel expressed in UML [7]. The model of the designed system is decomposed into components such as sensors, effectors, maps or planners. It defines the communication mechanisms between those components. RobotML enables the specification of the system architecture, inter-subsystem communication mechanism and the behaviour of the components. Code generation is based on an ontology. The domain metamodel includes architecture, communications, behaviour and deployment. Behaviours are defined in terms of Finite State Machines (FSMs) or algorithms. With each state and transition of the FSM activities are associated. Deployment specifies the middleware and simulators that are to be employed by the target system. RobotML significantly facilitates robotic system implementation, but it does not guide the designer as to what should be the roles of individual system components and what should be the rules of their composition.

BRICS Component Model (BCM) formulates guidelines and offers metamodels for the creation of individual components and their composition into entire systems [1,2]. It introduces a set of rules for building individual components and component-based architectures. The components are connected through ports (Component–Port–Component metamodel). Those connections are independent of the runtime platform. Specification based on those rules distinguishes five development aspects (5C): computation (processing, reading and writing data), communication (transfer of data between components), coordination (orchestrating the components), configuration (parametrisation of behaviour) and composition (coupling of reusable components producing predictable system behaviour once the behaviour of its components is known). The approach is based on meta-modelling.

Similar rules regarding the software structure of the system are defined in the SmartSoft [8] language. It highlights the description of the communication between components. Seven communication patterns are defined: *send*, *query*, *push newest*, *push time*, *event patterns*, *state* and *wiring* [4]. The language is related to the set of programming tools included in SmartMDSD Toolchain [9]. Those tools guide the system creator through subsequent phases of its development: design, implementation, integration and deployment.

The V3CMM [10] language differentiates three aspects of system architecture: (1) structure—a static structure of simple and complex components, (2) coordination—an event-based description of the components and (3) algorithms—a description of algorithms executed by a component in a given state.

The examples of MDSD tools mentioned above have many common features:

- Utilise well-known graphical specification languages—the MDSD itself, as well as the system described using this MDSD, are presented in the form of graphic diagrams derived from one of the well known graphical specification languages, e.g., UML [11] or SysML [12].
- Are based on formal models—rules and constraints defined by MDSD language are represented in the form of a formal model.
- Enable direct transformation of the specification into implementation—specific entities defined in the MDSD tools map to specific entities of the framework used for the creation of the robot software, e.g., components, communication channels and data types.
- Are compatible with open source frameworks—the specification can be easily transformed into executable code created by using open source frameworks.

- Are supplemented with controller code creation software—the MDSD tool provides or can cooperate with mechanisms that enable the transformation of specifications into the robot controller code.

### 1.2. SysML

Reliable methods of creating robot controllers are vital for the quality and certification of the produced software [13,14]. Thus, robotics MDSD tools usually include robot controller code generation facilities [15]. The frameworks utilised to implement lower layers of controllers are usually component oriented to make easier code reuse [16]. Additionally, components are the natural equivalent of blocks, classes, actions and other entities of UML and SysML [12].

Historically UML preceded SysML. Its first documentation, adopted as an Object Management Group (OMG) standard, was published in 1997. Many robotic MDSD tools such as BCM, RobotML, SmartSoft and V3CMM either use directly UML, or use UML profiles or at least adopt some UML concepts. In 2007, OMG published the first SysML specification, created as an extension/profile of the UML language. SysML enables the specification of general concepts, not only related to software, but also to the physical representation of cyber-physical systems. SysML graphical modelling language is widely used by engineers and specifically utilised in the robotics domain, e.g., [17–20]. SysML defines nine types of diagrams (Table 1), that enable multi-dimensional decomposition of the system into: Packages, Requirements, that the designed system should comply with, as well as Behaviours and system Structure.

**Table 1.** SysML diagrams and their abbreviations [12].

| Diagram Group | Diagram Kind | Abbreviation |
| --- | --- | --- |
| | Package | pkg |
| | Requirement | req |
| Behavioural | Activity | act |
| | Sequence | sd |
| | State Machine | stm |
| | Use Case | uc |
| | Parametric | par |
| Structural | Block Definition | bdd |
| | Internal Block | ibd |

As SysML has no formal semantics, a number of works has proposed its formalisation. By embedding SysML within a formal logic, formal methods can be used to maintain consistency as the design evolves [21]. In [22], the authors present TEPE, a graphical TEmporal Property Expression language based on SysML parametric diagrams. Properties are built upon temporal and logical relations between block attributes and signals. TEPE may be integrated with a SysML real-time profile. The paper [23] presents a method of formalizing the SysML Internal Block Diagram (IBD) semantics, by mapping it into the Hierarchical Colored Petri Net (HCPNs) semantics. The Description Logic, namely, SHIOQ(D), is used in [24] to describe the block diagrams. However, the informal semantics of SysML is often not completely captured or preserved when encoded in logic-based languages. Examples include the generation of a B-Specification from a UML class [25]. In [26], Chouali et al. use interface automata to formalize the semantics of SysML sequence and definition block diagrams. The work proposes verification of interoperability in component-based systems by combining interface automata and SysML models. The formalisation is presented by an algorithm and illustrated with an example. The approach is neither automated nor analysed.

The literature review presented above indicates that so far no single dominating standard of formalization and use of SysML has emerged. Usually, the method of SysML formalisation and how it is used depends on a specific application.

### *1.3. Embodied Agent*

A robotic system can be composed of one or more agents. The discussion whether a symbolic representation of the environment is necessary in control of intelligent robots lead to the reformulation of the concepts of embodiment and situatedness within robotics, which subsequently lead to the formulation of the concept of an embodied agent [27–35]. The classification presented in, e.g., [35] or Section 2.4, points out that structurally an embodied agent is the most complete type of an agent. It gathers the information about the state of the environment using its receptors and influences the environment using its effectors. Its control system is aware of the task that it has to execute. Using that knowledge combined with the information produced by receptors it commands the effectors in such a way as to fulfil the task. A monolithic control system would be too complex to specify and implement, thus its decomposition into subsystems is required. A natural way of decomposing complex systems is to partition them into hardware drivers and the task dependent part. Thus, the control system of an embodied agent is decomposed into its control subsystem, virtual effectors and receptors [32–35]. Virtual effectors transform commands obtained from the control subsystem into commands acceptable by the real (hardware) effectors. Virtual receptors aggregate the information acquired by the real (hardware) receptors. The control subsystem, being aware of the task it has to accomplish, uses the aggregated receptor data to produce effector commands. The transformative abilities of the virtual effectors and receptors enable the control system to express the task in terms of concepts more appropriate for that purpose than if it would have to process raw sensor readings and produce hardware control commands. Thus, the structure of an embodied agent produces the natural control loop: from the environment, through the real and virtual receptors, further to the control subsystem and finally through the virtual and real effectors back to the environment. As receptors sometimes have to be configured and the control subsystem might need proprioceptive data, a reverse path also exists.

The subsystems of an embodied agent communicate through data buffers. Subsystems also have internal memory. The contents of subsystem input buffers and internal memory form the arguments of transition functions producing the contents of the output buffers and internal memory. Transition functions are responsible for performing computations only. However, data has to be also propagated between subsystems. Iterative acquisition of new data, computation of the transition function and dispatch of the results is called a behaviour. The iterations cease when either a terminal or error condition is fulfilled. Both conditions take as arguments the contents of input buffers and internal memory and produce a Boolean value, and are therefore predicates. Multiplicity of subsystem behaviours leads to the necessity of choosing the next one, once the previous one terminates its activities. This is done by the subsystem finite state machine (FSM). The directed arcs of the state graph of the FSM are labeled by predicates called initial conditions. The true initial condition directs the FSM to its next state, in which a successive behaviour is invoked.

Embodied agents were used to specify research-oriented controllers for the investigation of control laws [36,37], fuzzy logic based controllers [38] and object-oriented ontology [39]. They utilised Finite State Machines, Hierarchical Finite State Machines and Petri Nets [40] to describe the system activities. This approach was used to develop the controllers of many different types of robots:

- Industrial robots, e.g., modified IRb-6 manipulator [34], whose control software was an inspiration for the example included in this article.
- Service robots, e.g., Velma robot [37].
- Mobile robots, e.g., Lynx [41], with selectable modes of locomotion, either horizontal or vertical.
- Skid steering platform Rex [42].
- General model of the wheeled robot [43].
- Social robots [44].

### 1.4. EARL

Although Robot Operating System (ROS) is the most commonly used robot control system implementation tool, the authors of [45] indicate that no hints guiding the creation of robot controllers are provided with it. The authors state that ROS is very well suited for creating various control systems, but it lacks support for the reuse of once created architectural solutions. They opted for SciROS, dedicated to the implementation of hybrid behaviour-deliberative systems. It purposefully constrains the developer creating a specific set of functionalities. On the one hand, this approach makes it impossible to create a controller with an architecture that does not meet the requirements, and on the other hand, it allows the reuse of its fragments. The problems indicated in this publication do not appear only in the popular frameworks, used for the creation of modern robot controllers. Those problems have a very general nature. In our work, we propose another solution, guiding robot controller developers creating systems composed both of real-time (RT) and non-RT components. This approach, in particular, can be applied to systems implemented using the ROS and Orocos frameworks.

EARL proposes a standardized approach to the control system specification of cyber-physical systems. The Embodied Agent (Section 1.3) is its foundation. EARL maps the concepts associated with Embodied Agents into SysML (Section 1.2) blocks with theirs properties, i.e., parts, references, values and operations. EARL fulfils all of the requirements formulated at the end of Section 1.1. It extends the set of best practices, by answering the following questions.

- How to organize a specification into SysML packages?
- For what purposes should the graphical tools be used and where the mathematical notation should be applied directly?
- How to map the specification into component systems?
- How to describe systems with a time-varying structure?

Figure 1 presents the dependencies of EARL packages. The model utilised by EARL is defined in the Model package (Section 2). The system instances that «realize» EARL model constraints are defined in the System Instance package (Section 3). This package «uses» independently defined computational structures from the Calculation Components package and data types from the DataTypes package.

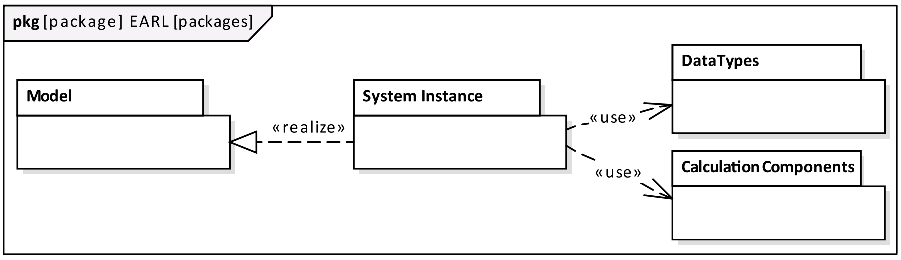



**Figure 1.** EARL package dependencies.

A robot controller is created by first producing an EARL based system specification, and then it is implemented with the support of FABRIC framework (Section 4). Section 5 provides a comparison of EARL with other MDSD tools. Section 6 provides final remarks and conclusions.

## 2. Model Formulation

The model of a system specified in EARL is composed of concepts describing its structure and behaviour. The structure of the model is specified with SysML Block Definition Diagrams (bdd) and Internal Block Diagrams (ibd) [46]. For clarity of presentation, the various aspects of the structure are presented by separate diagrams. The model is composed of a set of diagrams. Each of the diagrams presents only a part of the structure, however the whole set has to be consistent. Some of the model constraints are defined by mathematical equations.

### 2.1. System and Its Parts

System and Robot are the most general EARL concepts. They are structurally defined as in Figure 2. A System must contain at least one Robot $r$. A Robot is composed of at least one Agent $a$. A system may contain agents that are not elements of robots, e.g., an Agent coordinating the work of a group of Robots [35]. Agents are connected with *aa* inter-agent communication Links. Each *aa* Link can be referred by a Robot. In general, the Links parts names are created by combining the source block part name at the beginning of the Link part name and destination block part name at the end of the Link part name.

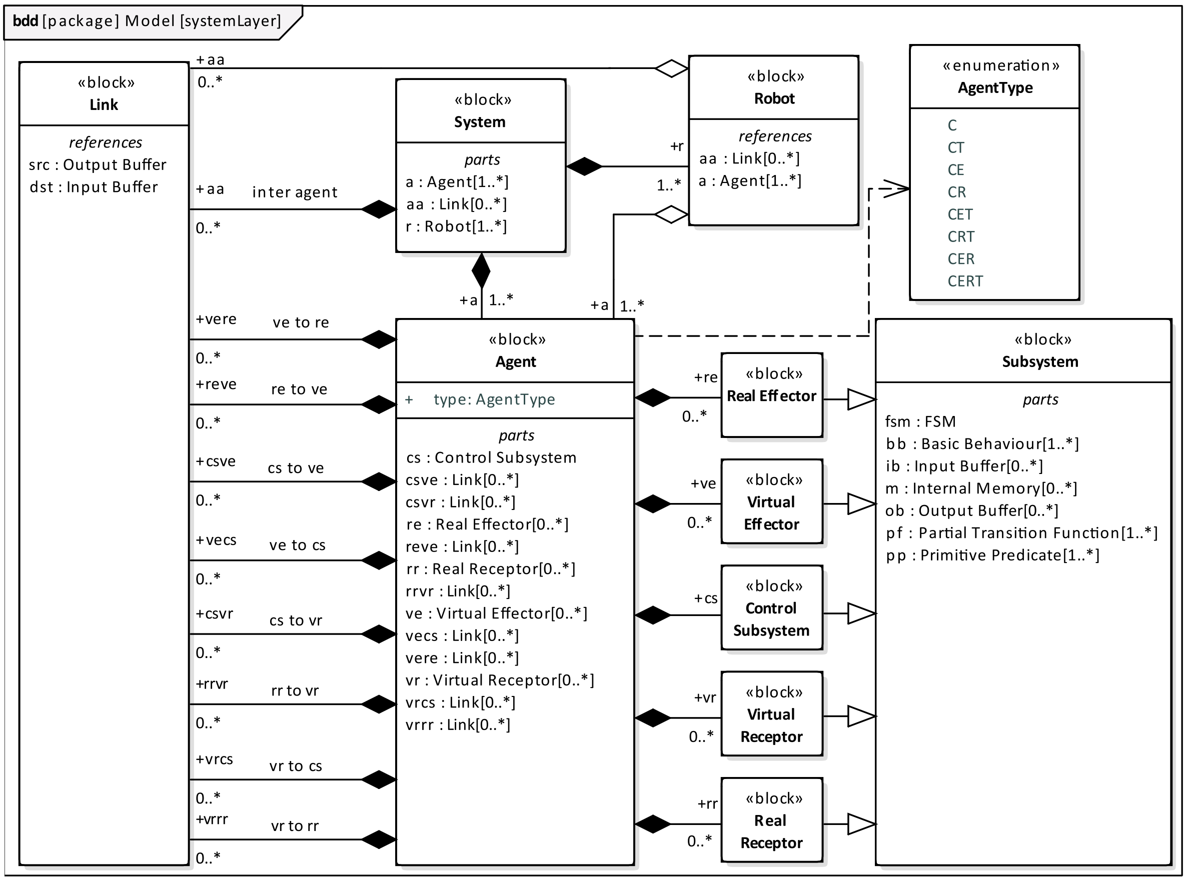



**Figure 2.** System and its parts.

In cyber-physical systems an Agent usually has a physical body, thus it is an Embodied Agent. It represents either a whole or a part of a robot [47]. The structure of an Agent is defined in Figure 2. The specific features of robotics, where an Agent can take on various roles, from real-time control, through sensor data processing, to execution of computationally demanding tasks [48], require its decomposition into various types of Subsystems and specialized Links between them. The variety of link names was introduced to distinguish the types of Subsystems or Agents that communicate with each other and the direction of data transmission. The blocks cardinality presented in Figure 2 is general, but particular system structure may introduce more strict constraints according to the extra rules presented further.

There are five different specialisations of Subsystems (right side of Figure 2). The main one (indispensable for an Agent) is a Control Subsystem *cs*, which coordinates the Agent's Subsystems and communicates with other Agents. Real Effectors *re* are Subsystems which affect the environment, whereas Real Receptors *rr* (exteroceptors) gather information from the environment. Virtual Subsystems (Virtual Receptors *vr* and Virtual Effectors *ve*) supervise the work of Real Subsystems. Therefore, the Real Subsystems of a particular type, cannot exist without virtual ones and vice versa, see Equation (1).

$$|vr| \geqslant 1 \iff |rr| \geqslant 1, \quad |ve| \geqslant 1 \iff |re| \geqslant 1. \tag{1}$$

Inequalities Equation (1) represents the necessary conditions ensuring the preservation of system integrity. Additional constraints have to be imposed on the number of Subsystems due to the specificity of inter-subsystem communication Links (Section 2.3).

## 2.2. Subsystem and Its Parts

The structure of a Subsystem is defined in Figures 3 and 4a. It contains Input Buffers *ib* and Output Buffers *ob*, Internal Memory *m* and other entities that are used to model both structural and behavioural aspects of a Subsystem, i.e., FSM *fsm* (Finite State Machine), Primitive Predicates *pp*, Basic Behaviours *bb* and Partial Transition Functions *pf*.

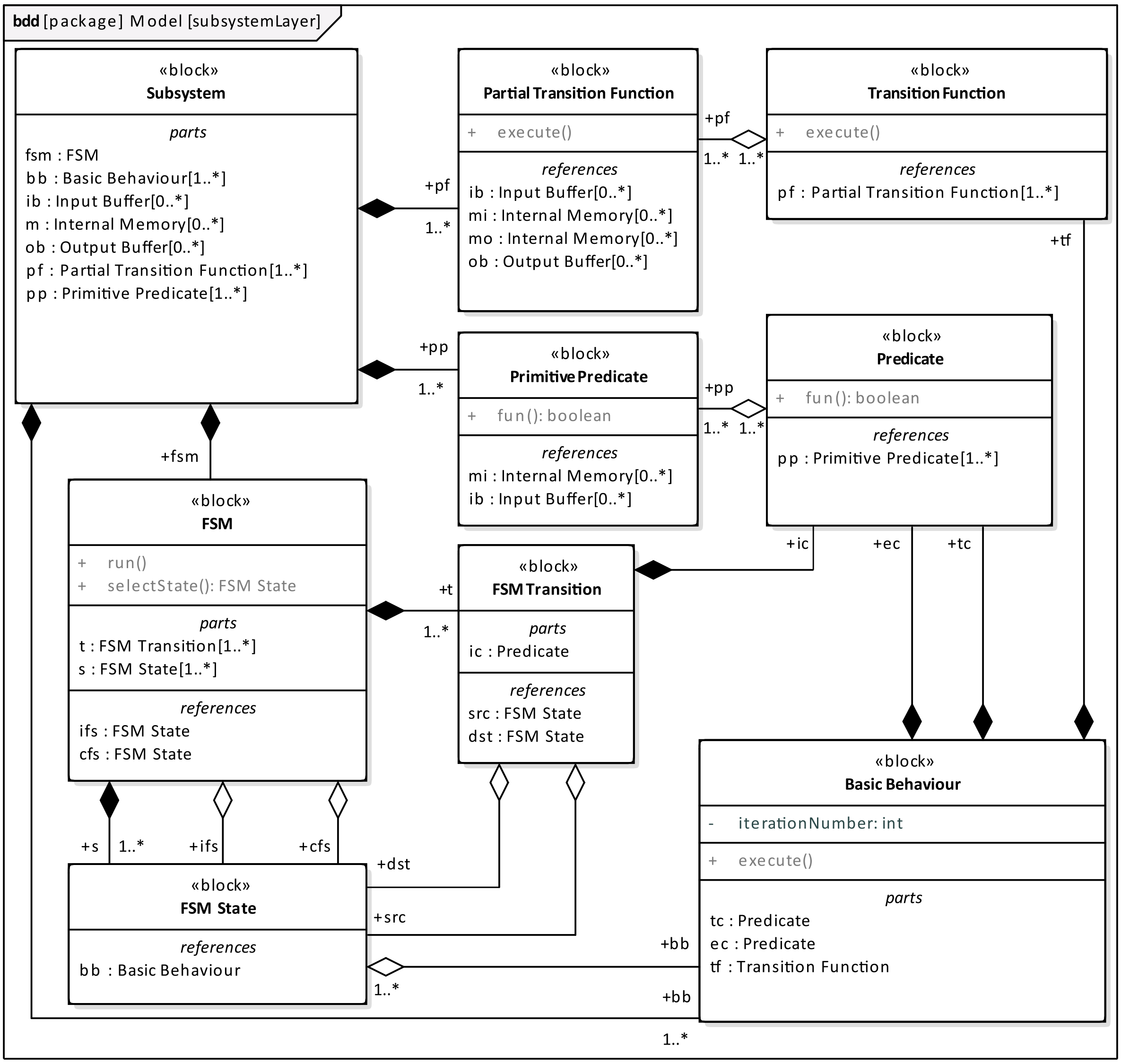
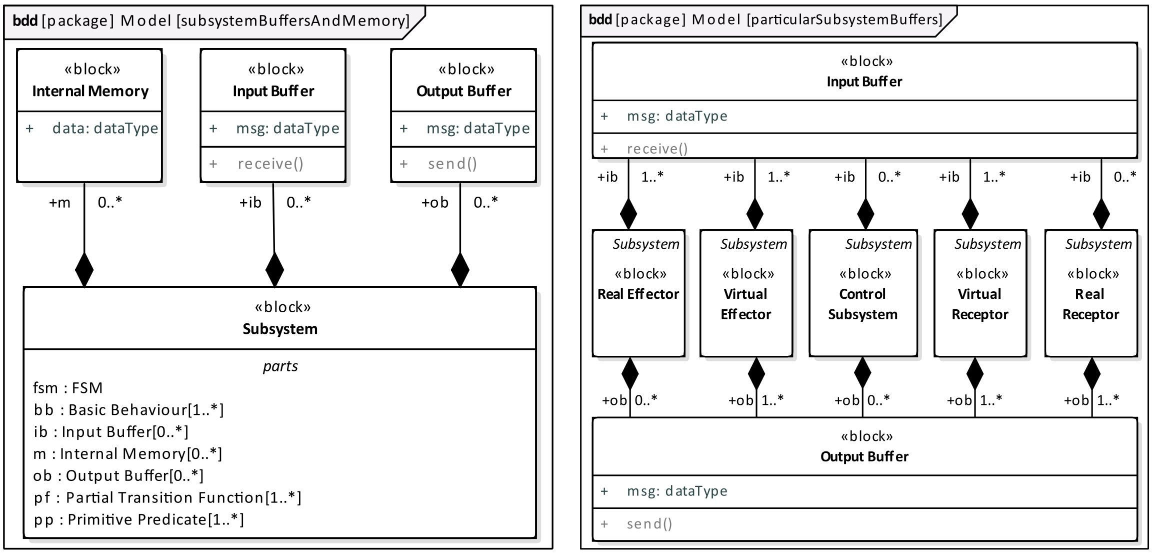



**Figure 3.** Subsystem and its parts (Input, Output Buffers and Internal Memory are excluded).

Figure 4b depicts relations between a particular Subsystem and its communication buffers. The communication constraints depicted in Section 2.3 cause that each Virtual Receptor or Virtual Effector must have at least one Input Buffer and one Output Buffer. A Real Effector needs at least one Input Buffer to receive commands, and a Real Receptor needs at least one Output Buffer to send sensory data.




(**a**) (**b**)

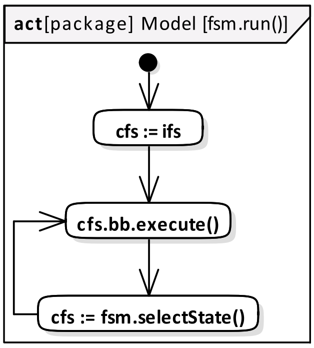
**Figure 4.** Subsystems and Buffers. (**a**) Subsystem Buffers and Internal Memory; (**b**) Relation of particular Subsystems to Communication buffers.

Input Buffer, Output Buffer and Internal Memory are defined analogically as in [49]. Each Buffer contains a data structure *msg*, which stores data of type dataType. The dataType can be defined either as a primitive type or a composite and nested structure. Input Buffer possesses an operation $receive()$, which enables communication with Output Buffers, and stores the received data in the Input Buffer. Analogically, Output Buffer has a $send()$ operation, which dispatches the data stored in the Output Buffer to the connected Input Buffers. Internal Memory stores *data*, which is a value of type dataType. Input and Output Buffers are graphically represented by squares connected by an arrow showing the direction of data transfer. Internal Memory is represented by a square with a bidirectional arrow. Various forms of communication between Subsystems have been described in the paper [33].

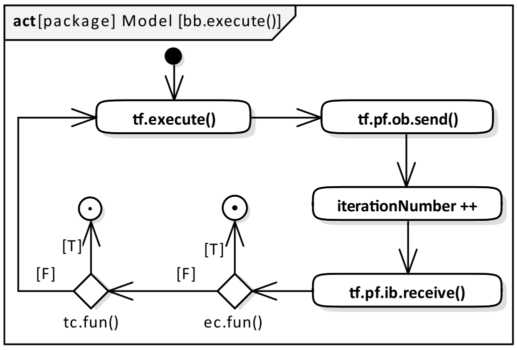
Similarly to [5,32], the EARL Subsystem structural model contains a Finite State Machine (FSM) that determines its activities (Figure 3). To define the FSM, the set $s$ of FSM States and the set $t$ of FSM Transitions are distinguished. With each of the states a behaviour $bb$ is associated. Figure 5a defines how the $run()$ operation works. The FSM starts in the initial FSM State $ifs$. Then, while the Subsystem is running, the $bb.execute()$ operation executes a behaviour associated with the current state, which is represented by $cfs$. The $fsm.selectState()$ operation evaluates the predicates associated with the FSM Transitions emerging from $cfs$ to select the next FSM State. FSM Transition (Figure 3) is defined by the source and destination FSM States as well as the associated Initial Condition, i.e., predicate $ic$.

In the following part of the article a SysML dot "." notation [46] is used to depict the nesting of the part instances as well as other block properties. The dot "." can be treated as an extraction operator. It is assumed that if a specific instance of a part is not indicated, the set of all instances of the part is taken into account. In particular, if there is only one instance, there is no need to name it explicitly, only the part name is needed. The same rule applies to references. As the particular parts compose objects of the same type, they can be interpreted as sets in mathematical formulas.

The structure of a Basic Behaviour is defined in Figure 3. It should be noted that in our previous papers using the concept of an Embodied Agent, e.g., [32–35], the "Basic Behaviour" was called shortly a "Behaviour". The name has been extended as "Behaviour" is a very general term in UML. The Basic Behaviour includes a Transition Function $tf$; a Terminal Condition $tc$ , which is a Predicate determining when the execution of the Basic Behaviour should terminate; and an error condition $ec$, which is a predicate indicating that an error has been detected in the execution of the Basic Behaviour. Basic Behaviour also posses an $execute()$ operation (Figure 5b). That operation, first executes a Transition Function $tf.execute()$, then all calculated Output Buffers values are sent out by $tf.pf.ob.send()$. Next, $iterationNumber$ is incremented, and $tf.pf.ib.receive()$ gets new values into Input Buffers. Finally, Error Condition $ec.fun$ and Terminal Condition $tc.fun$ are tested. If both values are false starts a new iteration of operations composing the Basic Behaviour; otherwise, the $fsm.run()$ operation designates the next FSM State (Figure 5a).




(**a**) FSM.$run()$.




(**b**) Basic Behaviour.$execute()$.

**Figure 5.** FSM and Basic Behaviour operations.

The structure of a Transition Function is defined in Figure 3. A Transition Function is decomposed into Partial Transition Functions. This sometimes reduces the redundancy of the specification, making it more comprehensible. Moreover, if the implementation of the specified system is based on components, a Partial Transition Function can be identified with a separate component or a set of components [50,51]. In this case, a Partial Transition Function can be reused in more than one Transition Function similarly as a component can be reused in more than one of the separate groups of components, where one group implements one specific behaviour of a system. Partial Transition Functions composing a Transition Function can be executed in diverse orders, see, e.g., in [47]. To define the execution of Partial Transition Functions within a Transition Function, the operation $execute()$ was introduced. The operation may vary between particular instances of Subsystems.

The structure of a Partial Transition Function is defined in Figure 3. It refers to Input Buffers, Output Buffers as well as Subsystem Internal Memory (Figure 6). A Partial Transition Function can read from the Internal Memory (using the $mi$ reference) or write to it (using the $mo$ reference). It can be defined as a composition of components from the Calculation Components Package (Figure 1). The composition is defined by a $tf.execute()$ operation. The Partial Transition Function algorithm is executed by an $pf.execute()$ operation. The concept of the Embodied Agent as presented in this paper introduces no restrictions on how to implement both of these operations.

Terminal Conditions used by a Basic Behaviour and Initial Conditions utilised within an FSM Transition can be decomposed into Primitive Predicates. A Primitive Predicate takes its arguments from Subsystem Buffers, see Figures 3 and 6. Both Predicate and Primitive Predicate execute an operation called $fun$ producing a Boolean output.

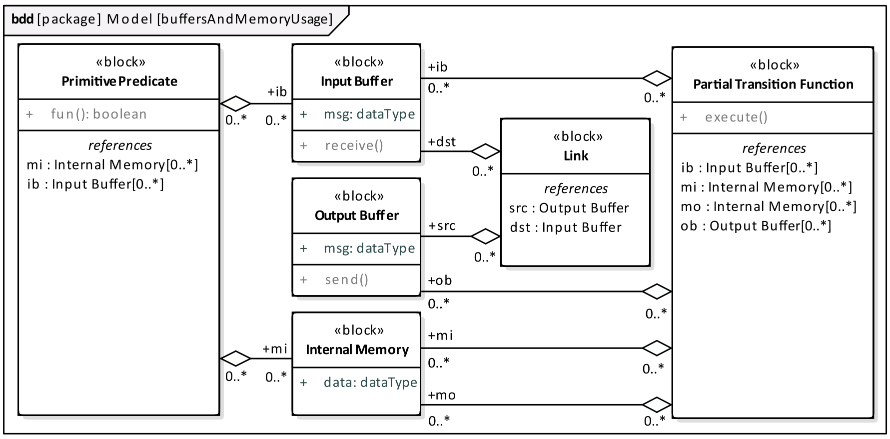



**Figure 6.** The utilisation of Buffers and Internal Memory by: Partial Transition Function, Primitive Predicate and Links.

### 2.3. Embodied Agent Communication

The general system architecture is defined by the Agents and their Subsystems, being the building blocks forming the system structure, and the communication links between those entities. In a way, the architecture is defined by the constraints that are imposed on permissible connections. If no constraints are imposed on the communication links, then the system designer has an excessive freedom of choice, which in the case of his limited experience might lead to an obscure structure. Therefore, architectures limiting this choice are preferred, thus leading to freedom from choice [9]. This provides guidance to the designers, which results in a clear system structure.

In the case of EARL, inter-agent and inter-subsystem communication [47] is defined by unidirectional communication Links (see Figures 2 and 6). The communication takes place between Input Buffers and Output Buffers of Subsystems. Figure 7 presents acceptable communication links between pairs of Subsystems. Note that the inter-agent communication is realized between the Control Subsystems of the communicating agents. Additionally, Figure 7 shows that for each Real Effector present in the system at least one transmission chain should exist. The commands produced by the Control Subsystem, transformed by the Virtual Effector, must reach the Real Effector. Analogically, for each Real Receptor, there is one compulsory communication chain that transmits and processes sensory data. The Real Receptor provides data to the Virtual Receptor which in an agregated form passes it to the Control Subsystem. The other communication Links appearing in Figure 7 are not obligatory. To define bidirectional communication, a pair of unidirectional communication Links is used. Detailed discussion of communication in Embodied Agent systems is presented in [33].

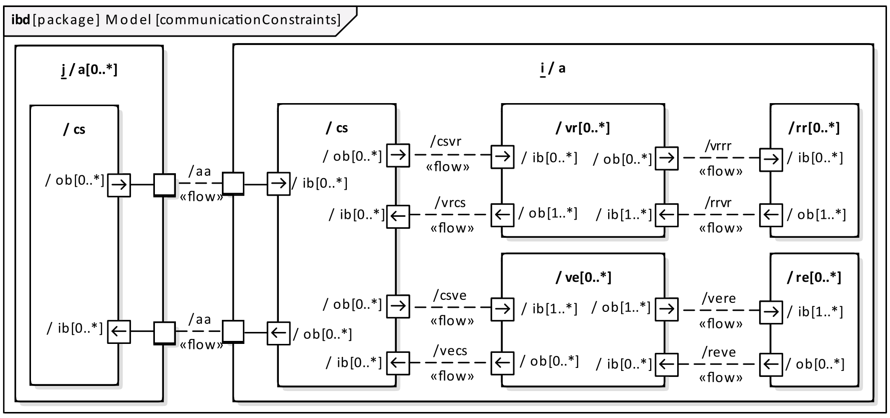



**Figure 7.** Communication constraints, where $i \neq j$.

### 2.4. Types of Agents

Four general activities of an Agent can be distinguished [35]:

**C** – overall control of the agent,
**E** – exerting influence over the environment by using effectors,
**R** – gathering the information from the environment by using receptors, and
**T** – inter-agent communication (transmission).

The first activity is indispensable, but the other three are optional, thus eight types of Agents result (Table 2), depending on their capabilities. However, only seven are of utility, as an agent without any of the optional capabilities is useless.

**Table 2.** Type of Agent, number of its Subsystems ($|ve|$, $|re|$, $|vr|$, $|rr|$) and number of inter agent communication Links ($|aa|$) expressed with respect to the number of Buffers of the considered Agent.

| | $\lvert cs \rvert$ | $\lvert ve \rvert$ | $\lvert re \rvert$ | $\lvert vr \rvert$ | $\lvert rr \rvert$ | $\lvert aa \rvert$ | **Description** |
|---|---|---|---|---|---|---|---|
| **C** | 1 | 0 | 0 | 0 | 0 | 0 | zombie (useless) |
| **CT** | 1 | 0 | 0 | 0 | 0 | 1..* | purely computational agent |
| **CE** | 1 | 1..* | 1..* | 0 | 0 | 0 | blind agent |
| **CR** | 1 | 0 | 0 | 1..* | 1..* | 0 | monitoring agent |
| **CET** | 1 | 1..* | 1..* | 0 | 0 | 1..* | teleoperated agent |
| **CRT** | 1 | 0 | 0 | 1..* | 1..* | 1..* | remote sensor |
| **CER** | 1 | 1..* | 1..* | 1..* | 1..* | 0 | autonomous agent |
| **CERT** | 1 | 1..* | 1..* | 1..* | 1..* | 1..* | full capabilities |

### 2.5. Specification of a Particular Robot Control System

The particular structure of a system is specified by application of instances of specializations of blocks [12] constituting the general model presented above. The names of instances should be long enough to be descriptive and intuitive to interpret, thus reducing the need for additional glossaries. In our approach, each instance can set the number of parts and references (e.g., associated Buffers), however within the limits imposed by the general model. Similarly, each instance can redefine the particular operations of parent blocks present in the general model (e.g., each instance of Partial Transition Function redefines $pf.execute$ operation).

In general, a system instance is defined as a graph. Its nodes represent Agents $a$ and the directed arcs represent the communication Links $aa$ between them. It is a good practice to name Links by using the names of communicating Agents: first the source Agent name, then a comma, and finally the destination Agent name. Input Buffers and Output Buffers of the Control Subsystems are depicted as sources and destinations of dataTypes being transmitted through the Links. The Buffer names reflect the content of dataType being transmitted. The Subsystems are defined analogically.

Specification refers to a system with a static structure and invariable behaviour, or a system with a variable structure at a certain time instant that both can be efficiently solved by using advanced optimization techniques proposed in [52,53]. To specify a particular system, instances of the relevant concepts appearing in the general system model should be concretised. The SysML diagrams [54] are a part of the EARL-based system (Table 3). Some of the EARL concepts are specified mathematically:

- model and system instance constraints that can not be practically formulated in diagrams,
- $fun$ operations of Predicates and Primitive Predicates, and
- some calculations performed inside actions of Activity Diagrams of Partial Transition Functions, e.g., control laws.

In addition, mathematical notation is used to express formal conditions ascertaining the correctness of the composition of Partial Transition Functions.

**Table 3.** SysML diagrams describing system parts in EARL.

| System Part and Function | SysML Diagrams |
|---|---|
| System and its parts, initial analysis | req, uc |
| System and Agent internal structure, Links, Input Buffer, Output Buffer | ibd |
| FSM, FSM State | stm |
| Operations of blocks | act |

## 3. Example of a System Specified Using EARL

This section is devoted to the illustration of how to use the EARL language to specify a robot control system. The example presents a single robot multi-agent system containing **CT** and **CET** agents. For the obvious reason of briefness, this description is not a complete specification, but contains only examples of important aspects of the general model and its use:

- Structure of the whole System with Buffers, Internal Memories, inter Agent communication Links, and dataTypes used by them.
- Structure of the particular Agent with Buffers, Internal Memories, inter Subsystem communication Links, and dataTypes used by them.
- Specification of a particular Subsystem, its structure and behaviour, i.e., Buffers, Internal Memories, dataTypes, FSM, Basic Behaviours and their Terminal Conditions and Error Conditions; Primitive Predicates, FSM Transitions and their Initial Conditions; method of both composition and execution of Partial Transition Functions and control law utilised in the activity diagram of Partial Transition Function.

A manipulation robot with N degrees of freedom and a gripper is considered, capable to perform e.g. pick and place task. The specification process starts with the definition of the System structure. Tips on the specification of requirements and use cases using SysML can be found in [55,56].

### 3.1. Structure of the System Composed of Agents

There are three Agents in the System (Figure 8). The Agent *task/a* supervises the task execution, i.e., picking and placing objects; the Agent *manip/a* controls the N-DOF manipulator; and the Agent *grip/a* controls the gripper. The gripper controller is separate from the manipulator controller, because different grippers can be attached to the manipulator, thus separate Agents facilitate system modification.

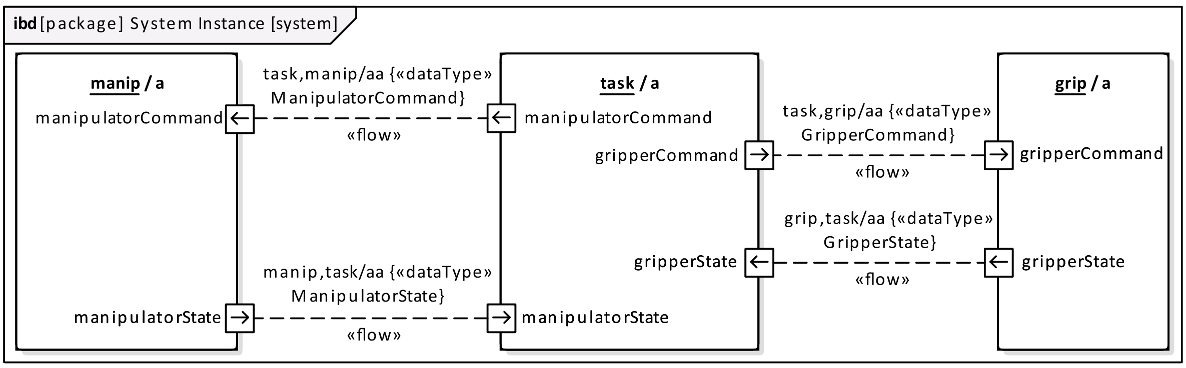



**Figure 8.** Structure of the considered exemplary System.

Figure 9 presents the dataTypes transmitted between the Agents. The Task Agent *task/a* sends ManipulatorCommands to the Manipulator Agent *manip/a*. The commands contain parameters, e.g., operational or joint position setpoints and a command to perform emergency stop. In return *task/a* gets a ManipulatorState dataType containing: the current operational or joint position, status of the manipulator movement and information whether an emergency stop occurred. The Task Agent *task*/*a* sends GripperCommand messages to the Gripper Agent *grip*/*a* and receives GripperStatus in return. Similarly to messages exchanged between *manip*/*a* and *task*/*a* Agents, the GripperCommand and GripperStatus messages contain parameters describing the desired and current gripper finger positions.

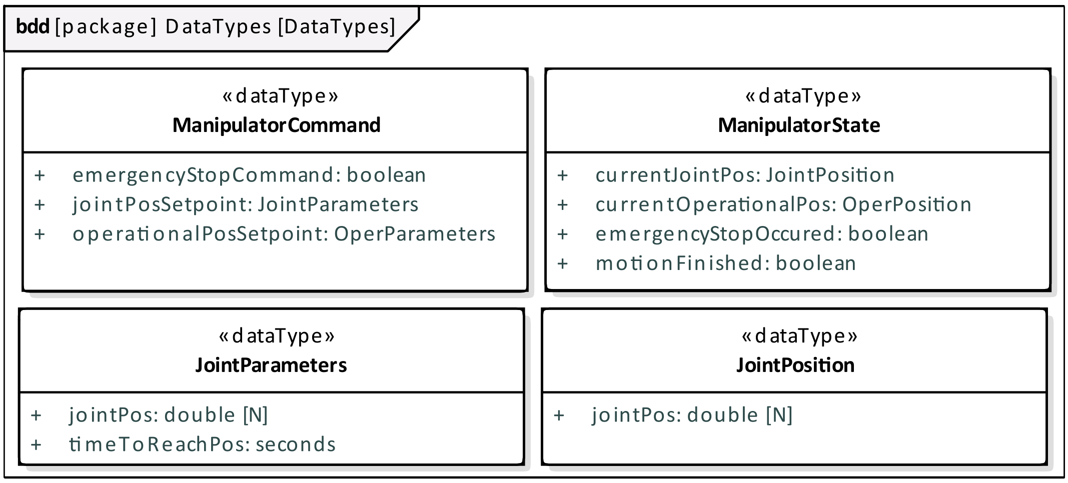



**Figure 9.** dataTypes transmitted within the System.

### 3.2. *Manipulator Agent manip*/*a*

The structure of the Manipulator Agent *manip*/*a* is presented in Figure 10. Each Real Effector *re* represents one of the $N$ drives of manipulator joints. Each drive is controlled by a Virtual Effector that, e.g., implements a motor position regulator. All $N$ Virtual Effectors *ve* are controlled by a single Control Subsystem *cs*, which causes the manipulator to move either in joint space, where it interpolates between joint positions, or in operational space, where it interpolates between Cartesian poses of a frame affixed to a chosen link of the kinematic chain.

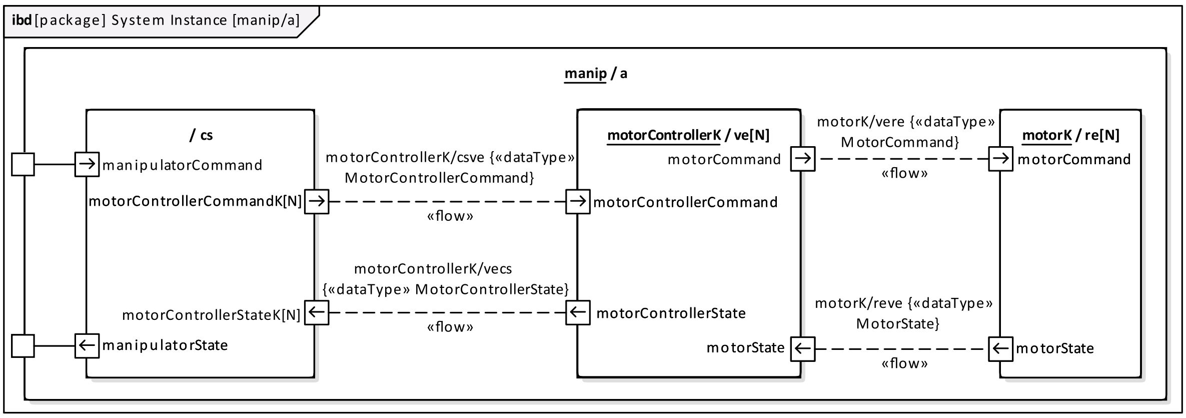



**Figure 10.** Structure of the Agent *manip*/*a*; letter K placed at the end of the instance name should be substituted by a number, i.e. $K \in \{1, ..., N\}$.

The dataTypes transmitted inside the Manipulator Agent *manip*/*a* are presented in Figure 11. The Control Subsystem *cs* sends MotorControllerCommand to each Virtual Effector *motorControllerK*/*ve*. The dataType contains the desired winding current value or a command to switch the hardware driver to the emergency stop state. Each Virtual Effector *motorControllerK*/*ve* sends to the Control Subsystem *cs* information about the current motor position and whether the hardware driver is in an emergency stop state. Each Virtual Effector *motorControllerK*/*ve* sends the desired motor winding current to its respective Real Effector $motorK/re$, and in return receives the encoder readings. Table 4 describes types of data stored in the $manip/a.cs$.

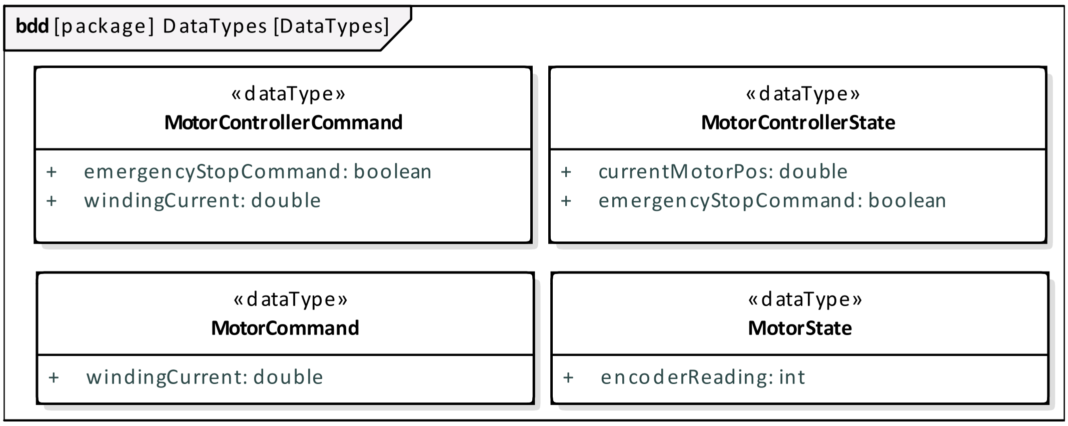



**Figure 11.** Definition of $manip/a$ dataTypes.

**Table 4.** $manip/a.cs.\boldsymbol{m}$ dataTypes.

| $\boldsymbol{m}$ | **dataType** |
|---|---|
| $motionFinished/m$ | boolean |
| $currentOperationalPos/m$ | OperPosition |
| $currentJointPos/m$ | JointPosition |
| $emergencyStopCommandK/m$ | boolean |
| $windingCurrentK/m$ | double |

Table 5 describes Primitive Predicates $pp$ used in the Control Subsystem $manip/a.cs$. They take as arguments the contents of the buffers and memory. The $newData(InputBuffer.msg)$ function producing Boolean values, returns TRUE if there is new data in the Input Buffer, and FALSE if the data is obsolete.

**Table 5.** Definitions of $manip/a.cs.\boldsymbol{pp}.\boldsymbol{fun}$.

| $\boldsymbol{pp}$ | $\boldsymbol{fun}$ |
|---|---|
| $emergencyStop/pp$ | $manipulatorCommand/ib.msg.emergencyStopCommand \lor$ $motorControllerState1/ib.msg.emergencyStopCommand \lor \ldots \lor$ $motorControllerStateN/ib.msg.emergencyStopCommand$ |
| $motionFinished/pp$ | $motionFinished/mi.msg$ |
| $newJointPos/pp$ | $newData(manipulatorCommand/ib.msg.jointPosSetpoint)$ |
| $newOperationalPos/pp$ | $newData(manipulatorCommand/ib.msg.operationalPosSetpoint)$ |
| $false/pp$ | FALSE |

Table 6 describes the Predicates utilised by $manip/a.cs$. Figure 12 shows possible transitions between the FSM States of the Control Subsystem $manip/a.cs$ as well as the association of Basic Behaviours to particular FSM States.

**Table 6.** Initial conditions labelling *manip/a.cs.fsm* transitions and terminal conditions of *manip/a.cs.bb*. It is assumed that *task/a* can not set simultaneously a new joint position and an operational space pose.

| **Labels of transitions between FSM States** | |
|---|---|
| $cs.fsm.\boldsymbol{t}.ic.fun \triangleq$ **PREDICATE** | |
| ***t*** | **PREDICATE** |
| $idle, jointMove/t$ | $newJointPos/pp.fun \wedge \neg emergencyStop/pp.fun$ |
| $jointMove, jointMove/t$ | $newJointPos/pp.fun \wedge \neg emergencyStop/pp.fun$ |
| $idle, operationalMove/t$ | $newOperationalPos/pp.fun \wedge \neg emergencyStop/pp.fun$ |
| $operationalMove, operationalMove/t$ | $newOperationalPos/pp.fun \wedge \neg emergencyStop/pp.fun$ |
| $jointMove, idle/t$ | $\neg emergencyStop/pp.fun$ |
| $operationalMove, idle/t$ | $\neg emergencyStop/pp.fun$ |
| $i, emergencyStop/t$; where $i \neq emergencyStop$ | $emergencyStop/pp.fun$ |
| **Definitions of Terminal Conditions** | |
| $cs.\boldsymbol{bb}.tc.fun \triangleq$ **PREDICATE** | |
| ***bb*** | **PREDICATE** |
| $idle/bb$ | $newJointPos/pp.fun \vee newOperationalPos/pp.fun \vee emergencyStop/pp.fun$ |
| $jointMove/bb$ | $motionFinished/pp.fun \vee emergencyStop/pp.fun$ |
| $operationalMove/bb$ | $motionFinished/pp.fun \vee emergencyStop/pp.fun$ |
| $emergencyStop/bb$ | $false/pp.fun$ |

The Control Subsystem *manip/a.cs* uses the following Partial Transition Functions.

- *calculatePosition/pf*—calculates manipulator joint positions and end-effector operational space pose.
- *jointMove/pf* / *operationalMove/pf*—generates the joint/operational space trajectory and calculates the winding current needed to realize the motion (Figure 13).
- *passiveRegulation/pf*—calculates the winding current needed to keep the manipulator in a stationary position.
- *emergencyStop/pf*—copies the information about the occurrence of an emergency stop to Output Buffers that are linked to the associated Subsystem Input Buffers.
- *outputManipState/pf*—composes *ManipulatorState/ob* (Figure 14a).
- *outputMotorCon/pf*—composes *DriveControllerCommandK/ob* (Figure 14b).

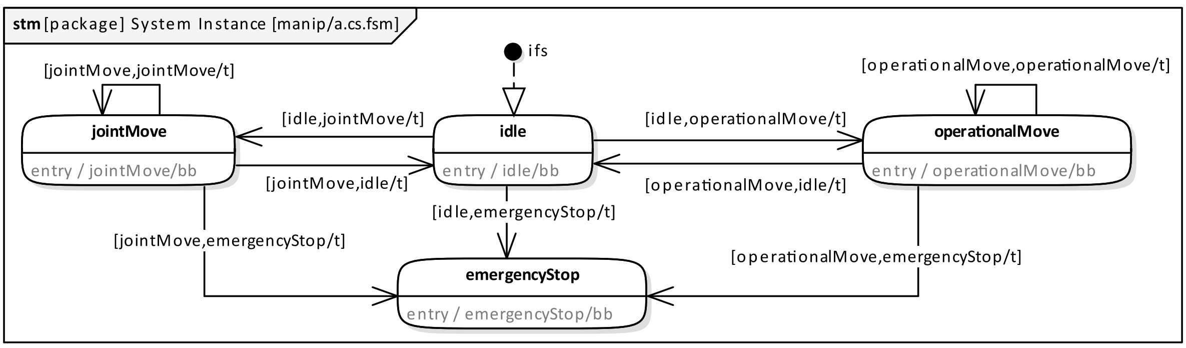



**Figure 12.** *manip/a.cs.fsm* definition. Conditions of transitions between FSM States are specified in Table 6.

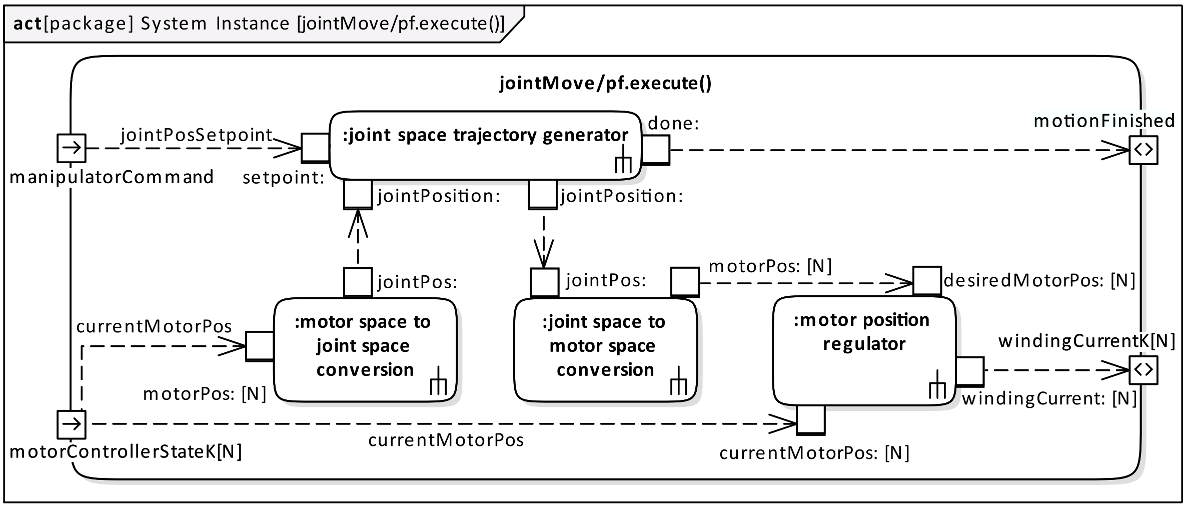
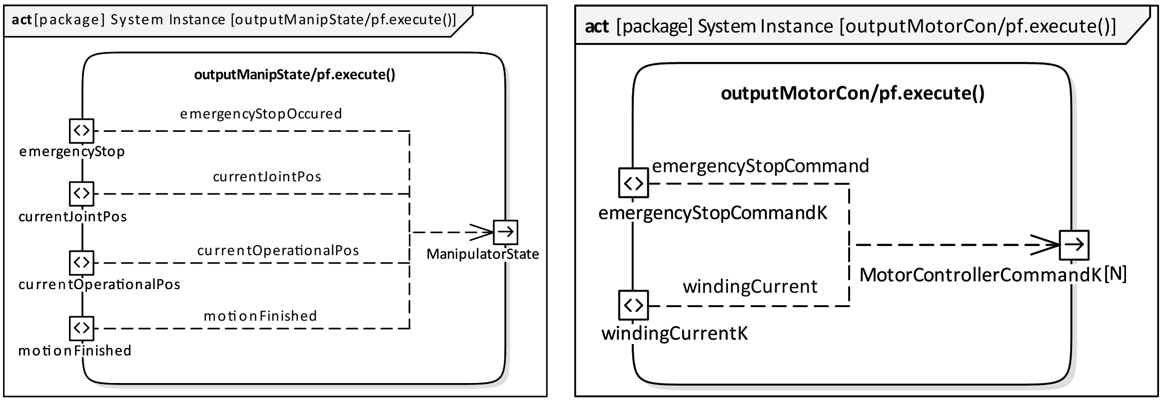



**Figure 13.** *manip/a.cs.jointMove/pf.execute()* – operation definition.




(**a**) *outputManipState/pf.execute()*. (**b**) *outputMotorCon/pf.execute()*.

**Figure 14.** *manip/a.cs.pf.execute()*—operations definition.

Figure 13 shows the execute operation of a *jointMove/pf* Partial Transition Function. This Partial Transition Function realises, e.g., the PI type *motor position regulator* for each joint Equation (2), Equation (3):

$$windingCurrent = K_p\, e(t) + K_i \int_0^t e(t')dt', \tag{2}$$

$$e = desiredMotorPos - currentMotorPos, \tag{3}$$

where $K_p$ and $K_i$ are, respectively, proportional and integral gain factors, $e$ is the position error, $t$ is time.

Table 7 shows which Partial Transition Functions constitute the definitions of Transition Function compositions used by Basic Behaviours of *manip/a.cs.bb*.

The Partial Transition Functions *manip/a.cs.pf* are subdivided into two disjoint sets: *pfc* and *pfo*. The *pfc* set Equation (4) contains Partial Transition Functions that take as arguments Input Buffers: *manipulatorCommand/ib* and *motorControllerCommandK/ib*$[N]$

$$\begin{aligned} pfc = \{&calculatePosition/pf, jointMove/pf, operationalMove/pf,\\ &passiveRegulation/pf, emergencyStop/pf\}. \end{aligned} \tag{4}$$

The values produced by them are inserted into the Internal Memory *mbo* Equation (5)

$$mbo = \{motionFinished/mo, currentOperationalPos/mo, currentJointPos/mo, emergencyStopCommandK/mo, windingCurrentK/mo\}. \tag{5}$$

Functions from the *pfo* set Equation (6) take arguments from the Internal Memory *mbo* Equation (5) and produce the Output Buffer values: *ManipulatorState/ob* and *MotorControllerCommandK/ob*[*N*], hence they produce output of the whole Subsystem

$$pfo = \{outputManipState/pf, outputMotorCon/pf\}. \tag{6}$$

It was assumed that any two Partial Transition Functions used by a particular Transition Function do not produce data to the same Output Buffers and Internal Memories, therefore the following conditions are formulated for the *pfc* set Equation (7) and the *pfo* set Equation (8), respectively,

$$(\forall x/bb)(\forall x/bb.i/pf, x/bb.j/pf \in pfc,\ i \neq j)(x/bb.i/pf.k/mo \nsim x/bb.j/pf.k/mo,\ k/mo \in mbo), \tag{7}$$

$$(\forall x/bb)(\forall x/bb.i/pf, x/bb.j/pf \in pfo,\ i \neq j)(x/bb.i/pf.k/ob \nsim x/bb.j/pf.k/ob), \tag{8}$$

where $\nsim$ stands for "is not the same entity".

**Table 7.** Compositions of Transition Functions *manip/a.cs.**bb**.tf.**pf***. The right part of the table presents what parts of output buffers and internal memory are produced by the specific Partial Transition Functions.

| | | /mo | | | | | /ob | |
|---|---|---|---|---|---|---|---|---|
| ***bb*** | ***pf*** | *motionFinished* | *currentJointPos* | *currentOperationalPos* | *emergencyStop* | *windingCurrentK* | *ManipulatorState* | *MotorControllerCommandK* |
| *idle/bb* | *outputManipState/pf* | | | | | | • | |
| | *outputMotorCon/pf* | | | | | | | • |
| | *calculatePosition/pf* | | • | • | | | | |
| | *passiveRegulation/pf* | | | | | • | | |
| *jointMove/bb* | *outputManipState/pf* | | | | | | • | |
| | *outputMotorCon/pf* | | | | | | | • |
| | *calculatePosition/pf* | | • | • | | | | |
| | *jointMove/pf* | • | | | | • | | |
| *operationalMove/bb* | *outputManipState/pf* | | | | | | • | |
| | *outputMotorCon/pf* | | | | | | | • |
| | *calculatePosition/pf* | | • | • | | | | |
| | *operationalMove/pf* | • | | | | • | | |
| *emergencyStop/bb* | *outputManipState/pf* | | | | | | • | |
| | *outputMotorCon/pf* | | | | | | | • |
| | *calculatePosition/pf* | | • | • | | | | |
| | *emergencyStop/pf* | | | | • | | | |

Transition Functions act in the following way. First, they compute the Partial Transition Functions from the *pfc* set, and then they compute the Partial Transition Functions from the *pfo* set. The fulfilment of Equations (7) and (8) makes it possible to run Partial Transition Functions being members of *pfc* in parallel in the first stage of the Transition Function execution, and then run the Partial Transition Functions being members of *pfo* set in parallel in the second stage of Transition Function execution. To illustrate the above considerations, Figure 15 shows the definition of $jointMove/bb.tf.execute()$ operation – practical realization of Partial Transition Functions execution for $jointMove$ Basic Behaviour.

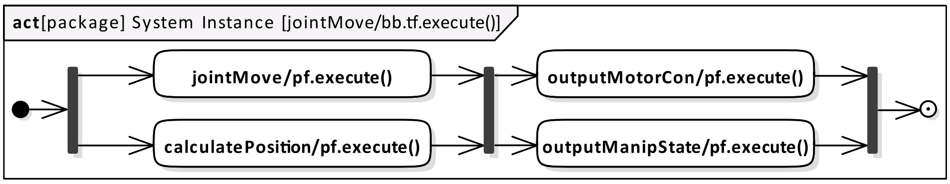



**Figure 15.** $jointMove/bb.tf.execute()$ operation definition.

## 4. FABRIC Framework

Four roles can be assigned to people developing and using robotics software: *end users*, *application builders*, *component builders* and *framework builders* [57]. The following considerations pertaining to code generation take into account the mentioned roles. The code of a Subsystem is generated out of its specification expressed using EARL and the source code written manually in C++. The specification items are referred to using the numbers and letters, e.g., 1.a—*deployer*. Figure 16 presents the relationship between: specification, patterns, parameters and code.

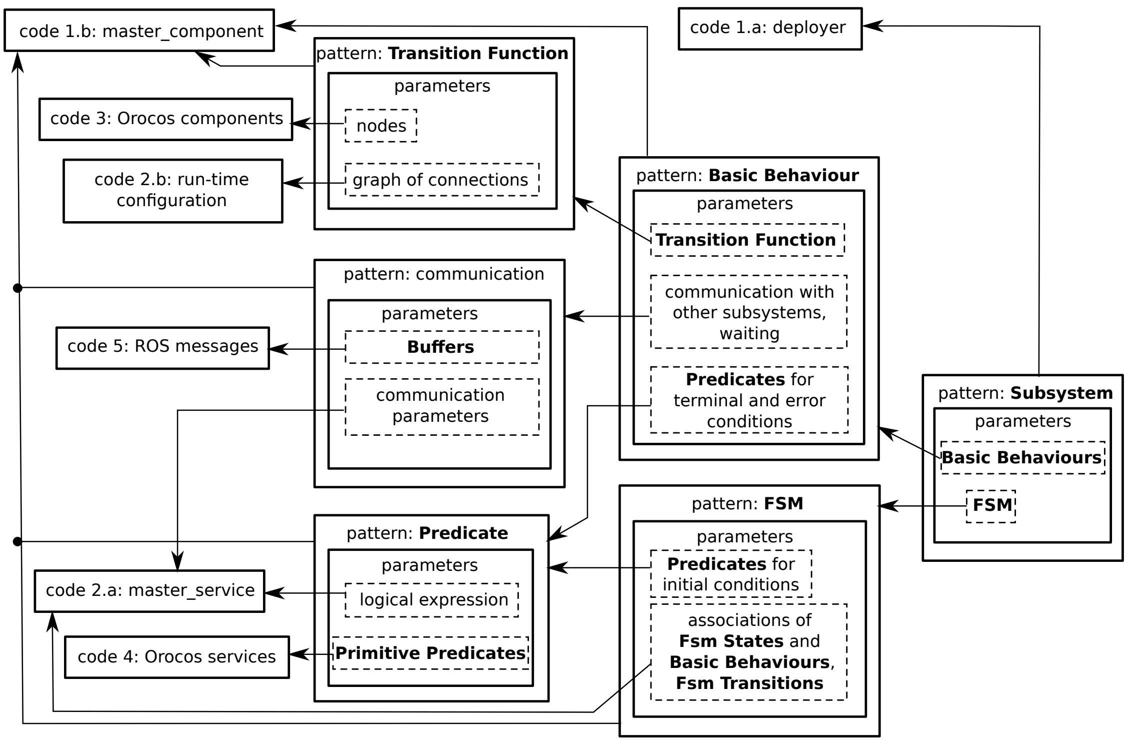



**Figure 16.** Specification decomposition into patterns and their parameters at various levels of generality [16]. Parameters are implemented using either code, or patterns at lower level of generality. Patterns are expressed as code. Arrows point from more general entities to more specific, i.e., their implementation.

For the purpose of Subsystem code generation EARL uses the FABRIC [16] framework. The Subsystem code generation procedure uses the following input data.

1. FABRIC—framework code, common to all Subsystems; it was manually written by the *framework builders*; it invokes Subsystem specific parts of code loaded at run-time; it is composed of the following items.
   a) *deployer*—code that loads Orocos framework together with its components and configures them.
   b) *master_component*—a specialised Orocos component that manages the activities of the subsystem.
2. Subsystem specification complying with MDSD, created by *application builders*, and delivered as two separate XML files:
   a) The first complies with the EARL system model containing the definitions of FSM, terminal and error conditions of Basic Behaviours, initial conditions of FSM Transitions, dataTypes and *send*() and *receive*() operations employed by the Buffers,.
   b) The second is used for run-time configuration; it contains the description of Orocos components that compose the Transition Functions, and the parameters of those components.
3. Orocos components that form Partial Transition Functions, written by *component builders*; the set of components that are to be used is selected by *application builders*.
4. Source code of the Primitive Predicates produced by *application builders*; it is delivered in the form of Orocos services.
5. ROS message definitions specifying dataTypes of the variables composing the Buffers; this code is written by *application builders*.

Figure 17 shows the method of creating the executable code out of the above mentioned items and the library source code.

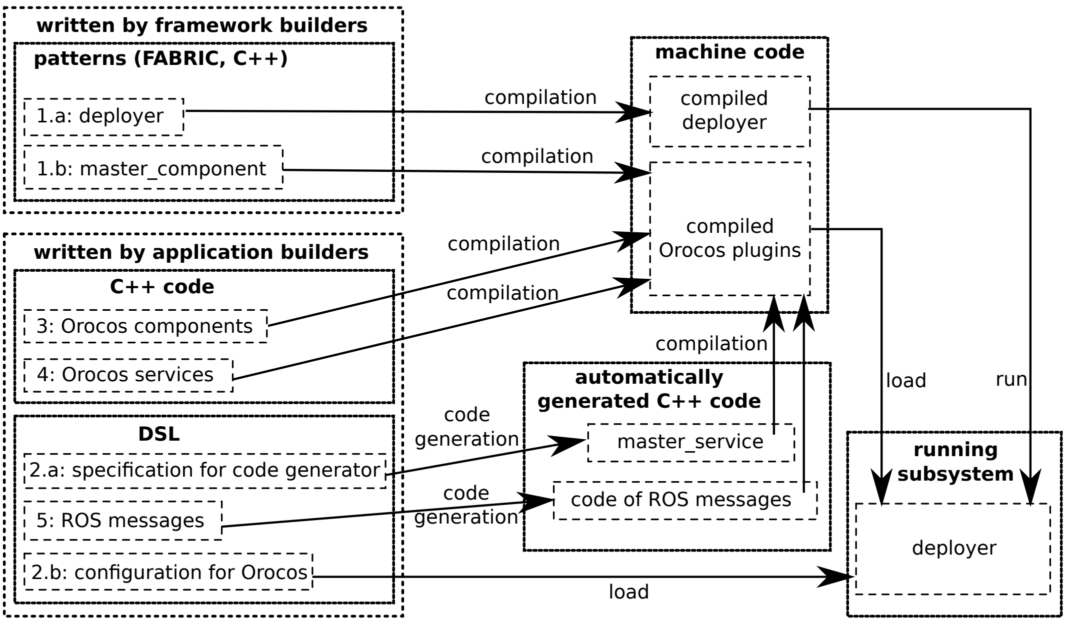



**Figure 17.** Creation of Subsystem executable code basing on its library source code and specification as well as subsystem deployment in run-time [16].

Out of the specification in item 2.a source code is generated. It is subsequently compiled together with the code written manually, i.e., both the source code of the framework (items 1.a and 1.b) as well as the code of the Orocos components and services (items 3 and 4). The *deployer* loads the necessary components and the item 2.b of the specification, configures Orocos components, and thus composes a working Subsystem.

## 5. Discussion

The transformation of the standard description of the Embodied Agent based systems [32,33,42,44,47,58] into EARL facilitates robot control system implementation. The introduction to this paper lists some of the other MDSD languages used to specify robot controllers: RobotML, SmartSoft, BCM (BRICS), and V3CMM. The article [4] had introduced nine features of Model Driven Software Development approaches that were used to compare those four MDSDs. We extend this comparison by adding EARL. The results of the comparison are presented in the form of Table 8, see the following definitions.

- Composability—in the phase of system integration, all the main properties of components should remain unchanged. The problem may concern, e.g., duplication of component names. EARL and other discussed MDSD languages do not meet this requirement.

**Table 8.** Comparison of robotic MDSD languages; feature: (+) exists, (−) does not exist.

| Feature \ MDSD language | EARL | RobotML | SmartSoft | BRICS | V3CMM |
|---|---|---|---|---|---|
| Composability | − | − | − | − | − |
| Compositionality | − | − | − | − | − |
| Static Variability | + | + | + | + | + |
| Dynamic Variability | + | − | + | − | − |
| Component Abstraction | + | + | + | + | + |
| Technology Neutrality | + | + | + | + | + |
| Knowledge Model | + | + | − | − | − |
| System Level Reasoning | − | − | + | − | − |
| Non-Functional Property Model | − | − | − | − | − |

- Compositionality—ensures correct operation of the system composed of components. System performance can be predicted for each reused component composition. In the case of EARL, this refers to, e.g., the composition of Partial Transition Function out of Calculation Components and creation of an Agent out of Subsystems. In both cases, the Calculation Components can be reused to create different structures, but EARL does not provide mechanisms verifying the correctness of the composition. EARL describes constraints imposed on communication between Subsystems and requirements for compatibility of communication Link dataTypes (Section 2.3). However, checking the fulfillment of these requirements is the responsibility of the system designer. Other discussed MDSD languages do not meet the compositionality requirement either.
- Static Variability—set of constraints imposed on the communication between specific parts of the system. All the discussed approaches have this feature. EARL also fulfills this requirement by defining restrictions imposed on the communication between Subsystems (Section 2.3).
- Dynamic Variability—during system operation, the structure of connections can vary. Out of the MDSD languages considered here SmartSoft and EARL have this capability. EARL assumes that the connection structure between Agents may vary during System operation.
- Component Abstraction—refers to component-based systems. EARL and all other MDSD languages discussed here meet this requirement. In EARL, a Calculation Component as well as an Agent or Subsystem can be treated as a component.

- Technology Neutrality—system specification should be independent of the system implementation technology. EARL as well as all the other MDSD languages discussed here have this feature. EARL is based on the concept of Embodied Agent, which abstracts away the implementation details. EARL itself is adapted to component systems and can be used, for the specification of ROS and Orocos based systems.
- Knowledge Model—MDSD is based on domain knowledge represented, for example, in the form of an ontology [59]. RobotML is based on an ontology that defines various concepts, entities and relationships between them, pertaining to the domain of robotics. EARL is based on the concept of an Embodied Agent, which can be treated as an ontology defining such robotics concepts as effector, receptor, control subsystem, communication buffers, internal memory, FSM, behaviour and transition function. Those concepts enable the composition of a robot system architecture taking into account the separation of concerns approach to software design [60], resulting in a hierarchic layered system.
- System Level Reasoning—refers to reasoning about the execution time or the model semantics. Only SmartSoft implements some aspects of this kind of reasoning.
- Non-Functional Property Model—defines such aspects of the system operation as reliability or performance. None of the discussed MDSD languages address this aspect. EARL also does not, but it is based on SysML, which enables the definition of such requirements.

Section 1.1 lists five features that are common to the four described MDSD languages. We used those features to compare EARL to the other MDSD languages. The assessment is based on the papers describing the MDSD languages, i.e., RobotML [5,6], BCM (BRICS) [1,2], SmartSoft [8], V3CMM [10] as well as the internet page of the SmartSoft project [61]. The results of the comparison are presented in the form of Table 9, please see the following.

- Utilisation of well known graphical specification languages—EARL is based on the SysML standard, RobotML implements a UML profile, SmartSoft relies on the UML standard, BCM (BRICS) defines a meta-model using UML and V3CMM although does not use UML directly, it adopts and adapts it.
- Reliance on a formal model—EARL uses the formal SysML notation, V3CMM uses Object Constraint Language (OCL) to define model constraints formally. Although no information on the use of formal models to define the other MDSD languages has been found by us, it can be assumed that they are formally defined, because it is possible to generate executable code out of them. It should be emphasised, that defining the model in the form of a UML profile can be considered as a formal description as there are tools enabling the verification of the correctness of instances of such models.
- Direct transformation into implementation—BCM (BRICS) supports model to model (M2M) transformation, which enables the transformation of a model defined in component-port-component metamodel into an Orocos component composition model. RobotML uses a code generator toolchain defined in the PROTEUS project [5]. SmartSoft uses SmartMDSD Toolchain to generate component hulls. Empty functions of the generated components are subsequently filled in with source-code by the user [61]. V3CMM transforms models defined using UML into skeletons of controllers expressed in the Ada language. EARL uses FABRIC [16] framework to transform the specification into code utilising the ROS and Orocos frameworks. The transformation of a specification into implementation is simple, because EARL was created taking into account the component nature of robotic frameworks. For instance, Subsystems naturally map to ROS nodes or nodelets, EARL Links map to ROS topics, and EARL, recursively, decomposed dataTypes map to ROS messages (rosmsg), Partial Transition Functions $execute()$ operations map to a set of Orocos components. DataTypes and Calculation Components, due to the organisation of the specification into packages, can be shared between various systems.

Similarily ROS messages and Orocos components can be shared between different robotic controllers.

- Compatibility with open source frameworks—BCM (BRICS) and EARL are compatible with ROS and Orocos, RobotML is compatible with Orocos-RTT and SmartSoft uses ROS. To the best of the authors' knowledge, V3CMM is not compatible with open source frameworks yet.
- Controller code generation software—V3CMM and RobotML are supported by the Eclipse platform. BCM (BRICS) uses the BRIDE toolchain, which is based on Eclipse. SmartSoft is associated with the SmartMDSD Toolchain utilising Eclipse. EARL uses FABRIC for code generation. The FABRIC configuration files are created using any text editor, whereas the system tests and its online analysis are done using FABRIC based graphical tools.

**Table 9.** Comparison of robotic MDSD language features: (+) exists, (−) does not exist.

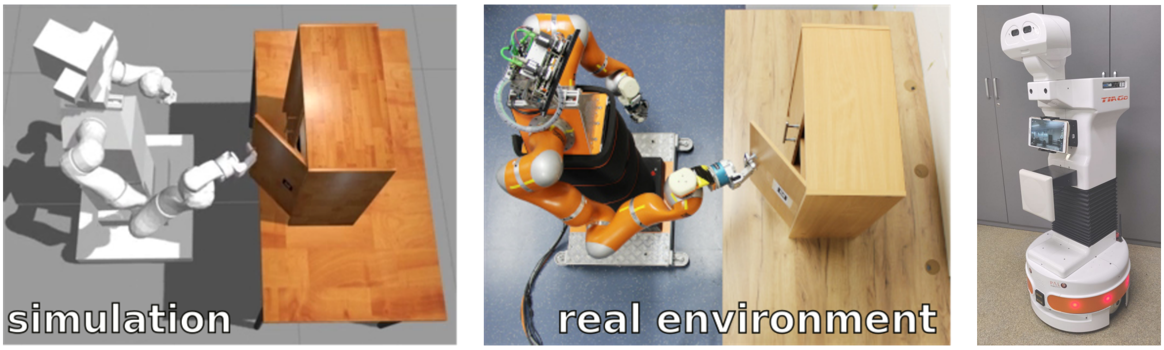
| Feature/MDSD | EARL | RobotML | SmartSoft | BRICS | V3CMM |
|---|---|---|---|---|---|
| Utilisation of graphical specification languages | SysML | UML | UML | UML | UML |
| Reliance on a formal model | mathematical | + | + | + | OCL |
| Direct transformation into implementation | FABRIC | PROTEUS project | SmartMDSD Toolchain | M2M | Ada language skeleton |
| Compatibility with open source frameworks | ROS Orocos | Orocos-RTT | ROS | ROS Orocos | − |
| Controller code generation software | FABRIC | based on Eclipse | SMARTSOFT MDSD (Eclipse) | BRIDE (Eclipse) | based on Eclipse |

## 6. Final Remarks and Future Work

EARL is convenient tool to effectively specify cyber-physical systems due to the following features,

- it employs model driven engineering, especially the rules governing the hierarchic composition of system layers out of lower level elements;
- it uses the FABRIC framework to automatically create controllers out of their specification,
- it is based on the concept of an Embodied Agent, which proved to be instrumental in the specification and implementation of many practical applications;
- it utilises standardised tools, i.e. SysML, supported by auxiliary software tools for developers,
- a large part of the designed system is specified by using graphical diagrams;
- there is no redundancy in the specification; and
- it is compact—contextual notation enables the introduction of long, descriptive names of block instances, which do not have to be repeated frequently.

The above mentioned advantages made EARL a tool of preference for the specification and implementation of the currently developed systems. The Velma https://www.robotyka.ia.pw.edu.pl/robots/velma robot (Figure 18a,b) is used as the main test-bed. As a two-handed robot, capable of force/torque sensing, equipped with three-fingered grippers, and having a movable head with mounted cameras and a Kinect sensor, it is sufficiently complex to perform tasks that one would expect of service robots. Our current research concentrates on several topics: (i) robot task planning especially for the purpose of grasping (continuation of [37,62,63]), (ii) automatic identification of physical parameters of the grasped object and their reflection in impedance control law (continuation of [36,37]) and (iii) ontology-based task planning and execution on an example of searching for a lost object somewhere at home [64,65] (continuation of [37]).




(a) (b) (c)

**Figure 18.** Velma service robot presenting an exemplary skill [16]—opening a cabinet door: (a) simulation, (b) reality, (c) Rico–TIAGo social robot with RFID antenna and tablet.

Additionally, EARL is currently being used to develop a system to assist elderly people at homes. The system uses a Rico https://www.robotyka.ia.pw.edu.pl/robots/rico robot (Figure 18c) and intelligent house components. This also takes EARL into an area beyond robotics, into the realm of cyber-physical systems [66], where robot works together with external devices, both sensors and effectors. Event-driven interruption of Agent's behaviours [67] as well as simulation of human behaviour for the purpose of testing social robots is currently investigated. In both cases, EARL is used to specify and implement the investigated systems.

**Author Contributions:** Conceptualisation: T.W., M.W. and C.Z.; methodology: T.W., M.W. and K.B.; software: T.W., D.S., M.W. and K.B.; validation: T.W., M.W. and D.S.; formal analysis: T.W., M.W. and C.Z.; investigation: T.W., M.W., D.S. and W.D.; resources, T.W.; data curation: T.W. and M.W.; writing—original draft preparation: T.W. and M.W.; writing—review and editing: T.W., M.W., D.S., W.D. and C.Z.; visualisation: T.W. and M.W.; supervision: T.W. and C.Z.; project administration: T.W.; funding acquisition: T.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was funded by the National Science Centre following the decision number DEC-2012/05/D/ST6/03097.

**Acknowledgments:** The authors would like to thank Maciej Bogusz, Szymon Jarocki, Daniel Giełdowski, Jarosław Karwowski and Jakub Postępski for their effort to evaluate EARL and Jacek Malec and Maciej Stefańczyk for valuable remarks.

**Conflicts of Interest:** The authors declare no conflict of interest. The funders had no role in the design of the study in the collection, analyses or interpretation of data; in writing of the manuscript; or in the decision to publish the results.

## References

1. Bruyninckx, H.; Klotzbücher, M.; Hochgeschwender, N.; Kraetzschmar, G.; Gherardi, L.; Brugali, D. The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems. In Proceedings of the SAC '13: Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1758–1764. [CrossRef]
2. Bischoff, R.; Guhl, T.; Prassler, E.; Nowak, W.; Kraetzschmar, G.; Bruyninckx, H.; Soetens, P.; Hägele, M.; Pott, A.; Breedveld, P.; et al. *BRICS—Best Practice in Robotics*; In Proceedings of the ISR 2010 (41st International Symposium on Robotics) and ROBOTIK (6th German Conference on Robotics), Munich, Germany, 7–9 June 2010; pp. 1–8.
3. Nordmann, A.; Hochgeschwender, N.; Wigand, D.L.; Wrede, S. A Survey on Domain-specific Modeling and Languages in Robotics. *J. Softw. Eng. Robot.* **2016**, *7*, 75–99.
4. Ramaswamy, A.; Monsuez, B.; Tapus, A. Model-driven software development approaches in robotics research. In Proceedings of the 6th International Workshop on Modeling in Software Engineering (MISE 2014), Hyderabad, India, 23–29 May 2014. [CrossRef]

5. Dhouib, S.; Kchir, S.; Stinckwich, S.; Ziadi, T.; Ziane, M. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *Simulation, Modeling, and Programming for Autonomous Robots*; Lecture Notes in Computer Science; Noda, I., Ando, N., Brugali, D., Kuffner, J.J., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7628. [CrossRef]
6. Kchir, S.; Dhouib, S.; Tatibouet, J.; Gradoussoff, B.; Simoes, M.D.S. RobotML for industrial robots: Design and simulation of manipulation scenarios. In Proceedings of the IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, Germany, 6–9 September 2016; pp. 1–8. [CrossRef]
7. Booch, G.; Jacobson, I.; Rumbaugh, J. *The Unified Modeling Language Reference Manual*, 2nd ed.; Addison Wesley: Reading, MA, USA, 2005.
8. Lutz, M.; Stampfer, D.; Lotz, A.; Schlegel, C. *Service Robot Control Architectures for Flexible and Robust Real-World task execution: Best Practices and Patterns*; Plödereder, E., Grunske, L., Schneider, E., Ull, D., Eds.; Informatik 2014; Gesellschaft für Informatik e.V.: Bonn, Germany, 2014; pp. 1295–1306.
9. Dennis, S.; Alex, L.; Matthias, L.; Christian, S. The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Softwaree. *J. Softw. Eng. Robot.* **2016**, *7*, 3–19.
10. Diego, A.; Cristina, V.C.; Francisco, O.; Juan, P.; Bárbara, Á. V3CMM: A 3-view component meta-model for model-driven robotic software development. *J. Softw. Eng. Robot.* **2010**, *1*, 3–17.
11. Pilone, D.; Pitman, N. *UML 2.0 in a Nutshell*; O'Reilly: Springfield, MO, USA, 2005.
12. Friedenthal, S.; Moore, A.; Steiner, R. *A Practical Guide to SysML: The Systems Modeling Language*, 3rd ed.; Elsevier, Morgan Kaufmann: Burlington, MA, USA, 2015.
13. Guiochet, J.; Machin, M.; Waeselynck, H. Safety-critical advanced robots: A survey. *Robot. Auton. Syst.* **2017**, *94*, 43–52. [CrossRef]
14. Chhaya, B.; Jafer, S.; Durak, U. Formal Verification of Simulation Scenarios in Aviation Scenario Definition Language (ASDL). *Aerospace* **2018**, *5*, 10. [CrossRef]
15. Pietrusewicz, K. Metamodelling for Design of Mechatronic and Cyber-Physical Systems. *Appl. Sci.* **2019**, *9*, 376. [CrossRef]
16. Seredyński, D.; Winiarski, T.; Zieliński, C. FABRIC: Framework for Agent-Based Robot Control Systems. In Proceedings of the IEEE 12th International Workshop on Robot Motion and Control (RoMoCo), Poznań, Poland, 8–10 July 2019; pp. 215–222. [CrossRef]
17. Dudek, W.; Banachowicz, K.; Szynkiewicz, W.; Winiarski, T. Distributed NAO robot navigation system in the hazard detection application. In Proceedings of the 21th IEEE International Conference on Methods and Models in Automation and Robotics, MMAR'2016, Miedzyzdroje, Poland, 29 August–1 September 2016; pp. 942–947. [CrossRef]
18. Stańczyk, B.; Kurnicki, A.; Arent, K. Logical architecture of medical telediagnostic robotic system. In Proceedings of the IEEE 21st International Conference on Methods and Models in Automation and Robotics (MMAR), Miedzyzdroje, Poland, 29 August–1 September 2016; pp. 200–205.
19. Mohd, N.N.S.; Mizukawa, M. Robotic services at home: An initialization system based on robots' information and user preferences in unknown environments. *Int. J. Adv. Robot. Syst.* **2014**, *11*, 112. [CrossRef]
20. Rahman, M.A.A.; Mizukawa, M. Model-based development and simulation for robotic systems with SysML, Simulink and Simscape profiles. *Int. J. Adv. Robot. Syst.* **2013**, *10*, 112. [CrossRef]
21. Graves, H.; Bijan, Y. Using formal methods with SysML in aerospace design and engineering. *Ann. Math. Artif. Intell.* **2011**, *63*, 53–102. [CrossRef]
22. Knorreck, D.; Apvrille, L.; de Saqui-Sannes, P. TEPE: A SysML language for time-constrained property modeling and formal verification. *ACM SIGSOFT Softw. Eng. Notes* **2011**, *36*, 1–8. [CrossRef]
23. Bouabana-Tebibel, T.; Rubin, S.H.; Bennama, M. Formal modeling with SysML. In Proceedings of the 2012 IEEE 13th International Conference on Information Reuse & Integration (IRI), Las Vegas, NV, USA, 8–10 August 2012; pp. 340–347.
24. Ding, S.; Tang, S.Q. An approach for formal representation of SysML block diagram with description logic SHIOQ(D). In Proceedings of the IEEE 2010 2nd International Conference on Industrial and Information Systems, Dalian, China, 10–11 July 2010; Volume 2, pp. 259–261.
25. Laleau, R.; Semmak, F.; Matoussi, A.; Petit, D.; Hammad, A.; Tatibouet, B. A first attempt to combine SysML requirements diagrams and B. *Innov. Syst. Softw. Eng.* **2010**, *6*, 47–54. [CrossRef]

26. Chouali, S.; Hammad, A. Formal verification of components assembly based on SysML and interface automata. *Innov. Syst. Softw. Eng.* **2011**, *7*, 265–274. [CrossRef]
27. Brooks, R.A. Intelligence without reason. *Artif. Intell. Crit. Concepts* **1991**, *3*, 107–163.
28. Brooks, R.A. New approaches to robotics. *Science* **1991**, *253*, 1227–1232. [CrossRef]
29. Russell, S.; Norvig, P. *Artificial Intelligence: A Modern Approach*; Prentice Hall: Upper Saddle River, NJ, USA, 1995.
30. Arkin, R.C. *Behavior-Based Robotics*; MIT Press: Cambridge, MA, USA, 1998.
31. Steels, L.; Brooks, R. *The Artificial Life Route to Artificial Intelligence: Building Embodied, Situated Agents*; Routledge: Abingdon, UK, 2018.
32. Kornuta, T.; Zieliński, C. Robot control system design exemplified by multi-camera visual servoing. *J. Intell. Robot. Syst.* **2013**, *77*, 499–524. [CrossRef]
33. Zieliński, C.; Figat, M.; Hexel, R. Communication within Multi-FSM Based Robotic Systems. *J. Intell. Robot. Syst.* **2018**, *93*, 787–805. [CrossRef]
34. Zieliński, C.; Kornuta, T.; Winiarski, T. A Systematic Method of Designing Control Systems for Service and Field Robots. In Proceedings of the 19th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR), Miedzyzdroje, Poland, 2–5 September 2014; pp. 1–14. [CrossRef]
35. Zieliński, C.; Winiarski, T.; Kornuta, T. Agent-Based Structures of Robot Systems. In *Trends in Advanced Intelligent Control, Optimization and Automation, Proceedings of the KKA 2017, Advances in Intelligent Systems and Computing, Kraków, Poland, 18–21 June 2017*; Mitkowski, W., Kacprzyk, J., Oprzędkiewicz, K., Skruch, P., Eds.; Springer: Cham, Switzerland, 2017; Volume 577, pp. 493–502. [CrossRef]
36. Zieliński, C.; Winiarski, T. Motion Generation in the MRROC++ Robot Programming Framework. *Int. J. Robot. Res.* **2010**, *29*, 386–413. [CrossRef]
37. Seredyński, D.; Banachowicz, K.; Winiarski, T. Graph–based potential field for the end–effector control within the torque–based task hierarchy. In Proceedings of the 21th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR'2016), Miedzyzdroje, Poland, 29 August–1 September 2016; pp. 645–650. [CrossRef]
38. Winiarski, T.; Kasprzak, W.; Stefańczyk, M.; Walęcki, M. Automated inspection of door parts based on fuzzy recognition system. In Proceedings of the 21th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR'2016), Miedzyzdroje, Poland, 29 August–1 September 2016; pp. 478–483. [CrossRef]
39. Zieliński, C.; Kornuta, T. An Object-Based Robot Ontology. In *Advances in Intelligent Systems and Computing (AISC)*; Springer: Berlin, Germany, 2015; Volume 323, pp. 3–14. [CrossRef]
40. Figat, M.; Zieliński, C. Methodology of Designing Multi-agent Robot Control Systems Utilising Hierarchical Petri Nets. In Proceedings of the 2019 International Conference on Robotics and Automation (ICRA), Montreal, QC, Canada, 20–24 May 2019; pp. 3363–3369.
41. Seredyński, D.; Stefańczyk, M.; Banachowicz, K.; Świstak, B.; Kutia, V.; Winiarski, T. Control system design procedure of a mobile robot with various modes of locomotion. In Proceedings of the 21th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR'2016), Miedzyzdroje, Poland, 29 August–1 September 2016; pp. 490–495. [CrossRef]
42. Janiak, M.; Zieliński, C. Control System Architecture for the Investigation of Motion Control Algorithms on an Example of the Mobile Platform Rex. *Bull. Pol. Acad. Sci. Tech. Sci.* **2015**, *63*, 667–678. [CrossRef]
43. Oprzędkiewicz, K.; Ciurej, M.; Garbacz, M. The agent, state-space model of the mobile robot. *Pomiary Autom. Robot.* **2018**, *22*, 41–50. [CrossRef]
44. Zieliński, C.; Stefańczyk, M.; Kornuta, T.; Figat, M.; Dudek, W.; Szynkiewicz, W.; Kasprzak, W.; Figat, J.; Szlenk, M.; Winiarski, T.; et al. Variable structure robot control systems: The RAPP approach. *Robot. Auton. Syst.* **2017**, *94*, 226–244. [CrossRef]
45. Rovida, F.; Crosby, M.; Holz, D.; Polydoros, A.S.; Großmann, B.; Petrick, R.P.A.; Krüger, V. SkiROS—A Skill-Based Robot Control Platform on Top of ROS. In *Robot Operating System (ROS): The Complete Reference*; Koubaa, A., Ed.; Springer International Publishing: Cham, Switzerland, 2017; Volume 2; pp. 121–160. [CrossRef]
46. Open Management Group. *OMG Systems Modeling Language—Version 1.6*. 2019. Available online: https://www.omg.org/spec/SysML/1.6/PDF (accessed on 20 February 2020).

47. Zieliński, C. Transition-Function Based Approach to Structuring Robot Control Software. In *Robot Motion and Control*; Lecture Notes in Control and Information Sciences; Kozłowski, K., Ed.; Springe: Berlin, Germany, 2006; Volume 335, pp. 265–286.
48. Zieliński, C.; Trojanek, P. Stigmergic cooperation of autonomous robots. *J. Mech. Mach. Theory* **2009**, *44*, 656–670. [CrossRef]
49. Trojanek, P. Design and Implementation of Robot Control Systems Reacting to Asynchronous Events. Ph.D. Thesis, Warsaw University of Technology, Warsaw, Poland, 2012.
50. Dudek, W.; Szynkiewicz, W.; Winiarski, T. Nao Robot Navigation System Structure Development in an Agent-Based Architecture of the RAPP Platform. In *Recent Advances in Automation, Robotics and Measuring Techniques*; Szewczyk, R., Zieliński, C., Kaliczyńska, M., Eds.; Springer: Berlin, Germany, 2016; Volume 440, pp. 623–633. [CrossRef]
51. Winiarski, T.; Banachowicz, K.; Walęcki, M.; Bohren, J. Multibehavioral position–force manipulator controller. In Proceedings of the 21th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR'2016), Miedzyzdroje, Poland, 29 August–1 September 2016; pp. 651–656. [CrossRef]
52. Caliciotti, A.; Fasano, G.; Nash, S.G.; Roma, M. An adaptive truncation criterion, for linesearch-based truncated Newton methods in large scale nonconvex optimization. *Oper. Res. Lett.* **2018**, *46*, 7–12. [CrossRef]
53. Caliciotti, A.; Fasano, G.; Nash, S.G.; Roma, M. Data and performance profiles applying an adaptive truncation criterion, within linesearch-based truncated Newton methods, in large scale nonconvex optimization. *Data Brief* **2018**, *17*, 246–255. [CrossRef] [PubMed]
54. Salado, A.; Wach, P. Constructing True Model-Based Requirements in SysML. *Systems* **2019**, *7*, 19. [CrossRef]
55. dos Santos Soares, M.; Vrancken, J. Requirements specification and modeling through SysML. In Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics, Montreal, QC, Canada, 7–10 October 2007; pp. 1735–1740. [CrossRef]
56. Soares, M.; Vrancken, J.; Verbraeck, A. User requirements modeling and analysis of software-intensive systems. *J. Syst. Softw.* **2011**, *84*, 328–339. [CrossRef]
57. Bruyninckx, H. OROCOS: Design and implementation of a robot control software framework. In Proceedings of the IEEE International Conference on Robotics and Automation, Washington, DC, USA, 11–15 May 2002.
58. Zieliński, C.; Szynkiewicz, W.; Figat, M.; Szlenk, M.; Kornuta, T.; Kasprzak, W.; Stefańczyk, M.; Zielińska, T.; Figat, J. Reconfigurable control architecture for exploratory robots. In Proceedings of the IEEE 10th International Workshop on Robot Motion and Control (RoMoCo), Poznan, Poland, 6–8 July 2015; pp. 130–135. [CrossRef]
59. Stenmark, M.; Malec, J. Knowledge-based instruction of manipulation tasks for industrial robotics. *Robot. -Comput.-Integr. Manuf.* **2014**, *33*, 56–67. [CrossRef]
60. Dijkstra, E. On the Role Of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*; Springer-Verlag: New York, NY, USA, 1982; pp. 60–66. [CrossRef]
61. The SmartSoft Approach. Available online: https://wiki.servicerobotik-ulm.de/about-smartsoft:approach (accessed on 20 February 2020).
62. Seredyński, D.; Szynkiewicz, W. Fast Grasp Learning for Novel Objects. Recent Advances in Automation, Robotics and Measuring Techniques. In *Advances in Intelligent Systems and Computing (AISC)*; Springer: Cham, Switzerland, 2016; Volume 440, pp. 681–692. [CrossRef]
63. Seredyński, D.; Winiarski, T.; Banachowicz, K.; Zieliński, C. Grasp planning taking into account the external wrenches acting on the grasped object. In Proceedings of the 2015 10th International Workshop on Robot Motion and Control (RoMoCo), Poznan, Poland, 6–8 July 2015; pp. 40–45. [CrossRef]
64. Tenorth, M.; Beetz, M. KnowRob: A knowledge processing infrastructure for cognition-enabled robots. *Int. J. Robot. Res.* **2013**, *32*, 566–590. [CrossRef]
65. Kunze, L.; Beetz, M.; Saito, M.; Azuma, H.; Okada, K.; Inaba, M. Searching objects in large-scale indoor environments: A decision-theoretic approach. In Proceedings of the 2012 IEEE International Conference on Robotics and Automation, Saint Paul, MN, USA, 14–18 May 2012; pp. 4385–4390. [CrossRef]

66. Khaitan, S.K.; McCalley, J.D. Design techniques and applications of cyberphysical systems: A survey. *IEEE Syst. J.* **2015**, *9*, 350–365. [CrossRef]
67. Dudek, W.; Węgierek, M.; Karwowski, J.; Szynkiewicz, W.; Winiarski, T. Task harmonisation for a single–task robot controller. In Proceedings of the 2019 12th International Workshop on Robot Motion and Control (RoMoCo), Poznań, Poland, 8–10 July 2019; pp. 86–91. [CrossRef]