EARL – Embodied Agent-based cybeR-physical control systems modelling Language - version 1.3 - reference manual

EARL developers group - https://www.robotyka.ia.pw.edu.pl/projects/earl/
tomasz.winiarski@pw.edu.pl

January 4, 2023

Important citation notice

If you are to use EARL, in your papers, please at first cite the Electronics journal article [15], where the initial version of EARL is presented.

1 Introduction

EARL is developed by the robotic team at Warsaw University of Technology, Institute of Control and Computation Engineering. EARL proposes a standardised approach to the control system specification of cyber-physical systems both in reality and simulation. The Embodied Agent of the Warsaw school [8] is its foundation. EARL maps the concepts associated with Embodied Agents into SysML blocks with theirs properties, i.e., parts, references, values and operations. 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 Basic EARL Model package (Section 2). The Basic EARL Model instances that «realize» EARL model constraints are exemplified in the Basic EARL Instance package (Section 3). This package «uses» independently defined computational structures from the Calculation Components package and data types from the Manipulation Robot Value Types package. The exemplary Basic EARL Model customisation is comprised in Search and Rescue System package (Section 4.1).

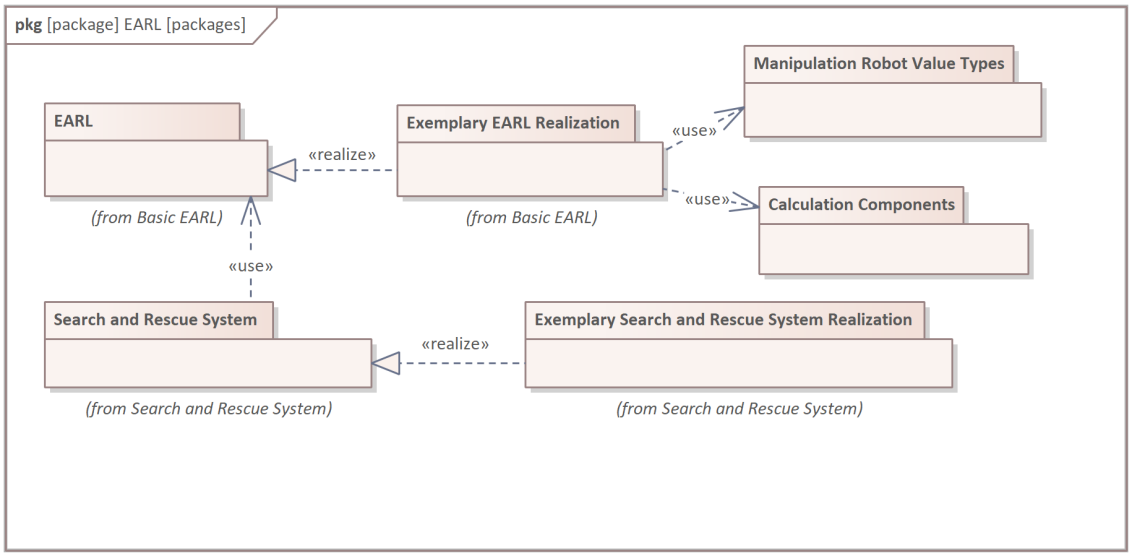


Figure 1: EARL packages dependencies

Although the large class of systems can be specified purely with the usage of the basic model presented further, the model can also be customised (e.g., [14, 6]) or incorporated into broader systems. Some suggestions and examples are presented in Section 4. As the EARL is constantly developed by its community, the list of major changes in EARL current version is listed in Section 5.

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) [9]. For compactness and clarity of presentation the stereotypes are introduced that are provided as dedicated UML profile. 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.The stereotypes names are long, hence descriptive, while parts' names are concise to reduce the space to point out the part and its context when it is convenient. Some of the model constraints are defined by mathematical equations.

2.1 System and Its Parts

System is the most general EARL concept. Its composition is defined in Figure 2. It can contain a number of Groups of Agents *ga*. A Group of Agents is composed of at least one Agent *a* and can constitute, e.g., a Robot or a part of the system executed on a particular computer. The Group of Agents can also refer to other Group of Agents as *ga*. Agents are connected with *a_a* inter-agent communication Links. Each *a_a* Link can be referred by a Group of Agents. 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.

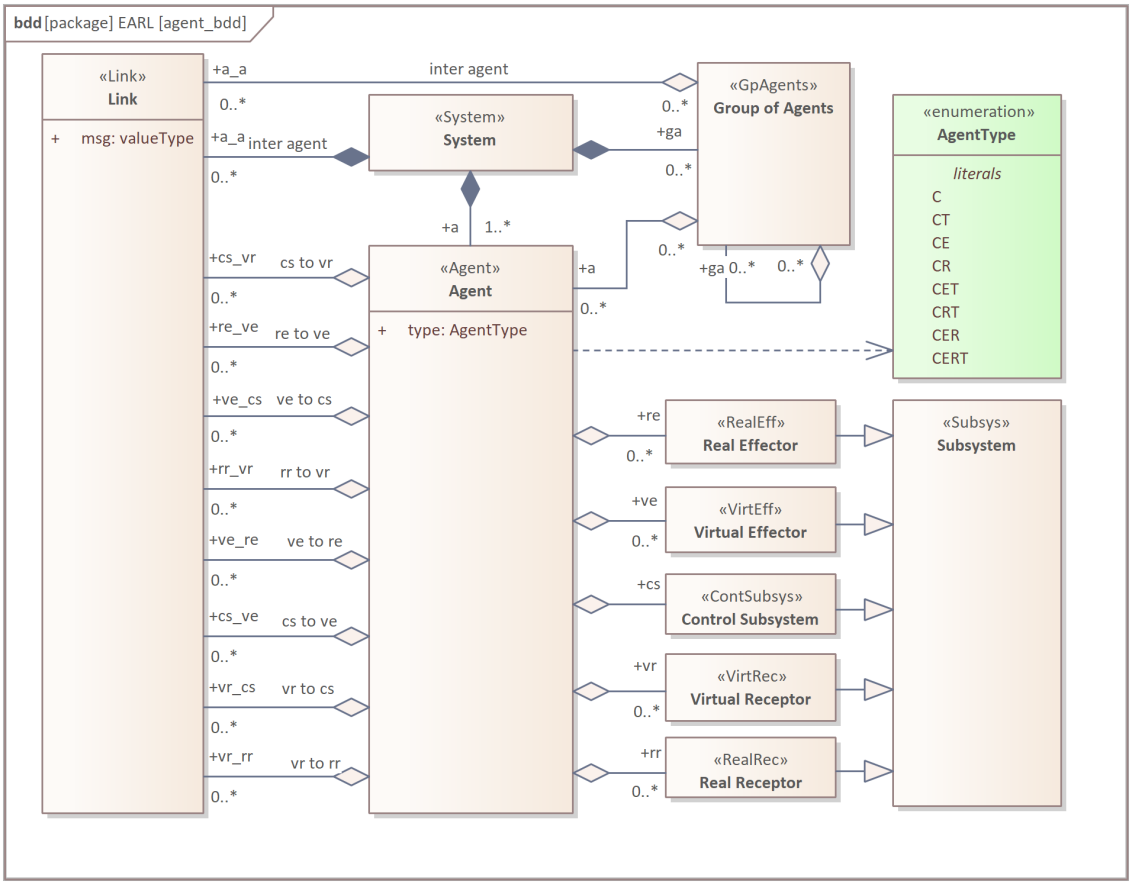


Figure 2: System layer

In cyber-physical systems an Agent usually has a physical body, thus it is an Embodied Agent. The composition 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 [19], require its decomposition into various types of Subsystems and specialised Links between them. The variety of link names was introduced to distinguish the types of Subsystems that communicate with each other and the direction of data transmission. The blocks cardinality presented in Figure 2 is general, but particular system composition 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).

The Links and Subsystems are composed in System (Figure 3) and aggregated in Agent to make it possible to logically share them between Agents.

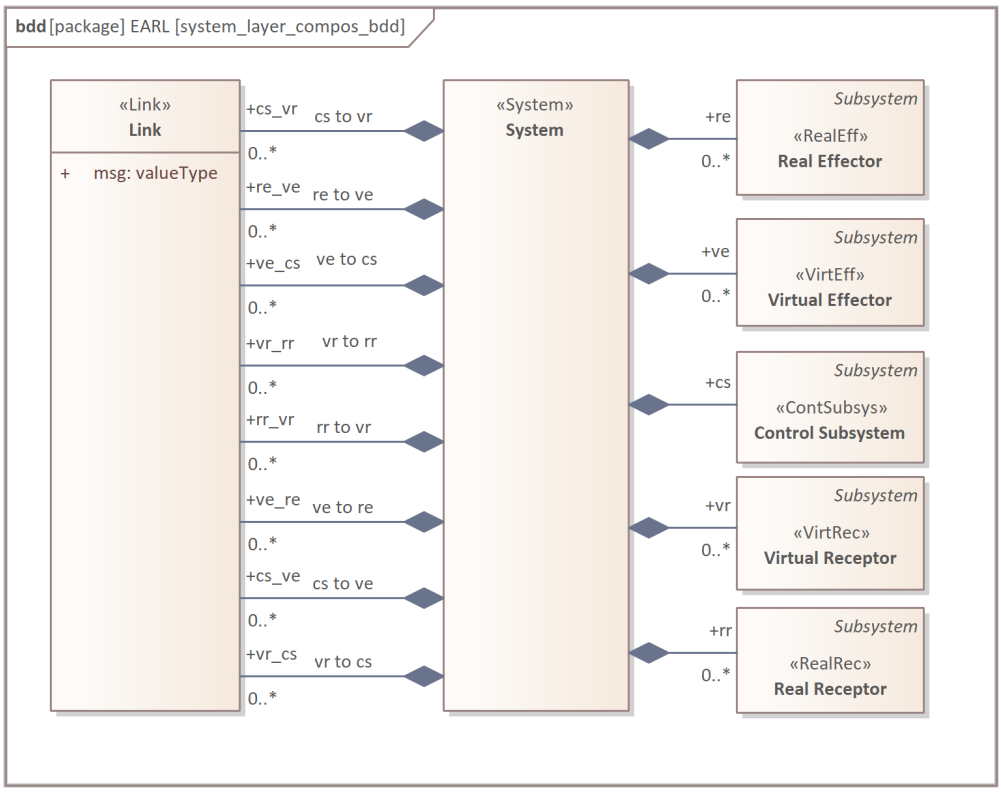


Figure 3: System and composed Links and Subsystems

The Group of Subsystems (Figure 4) was introduced for the purpose of alternative presentation (view) of System or Agent structure by grouping the Subsystems and Links between them.

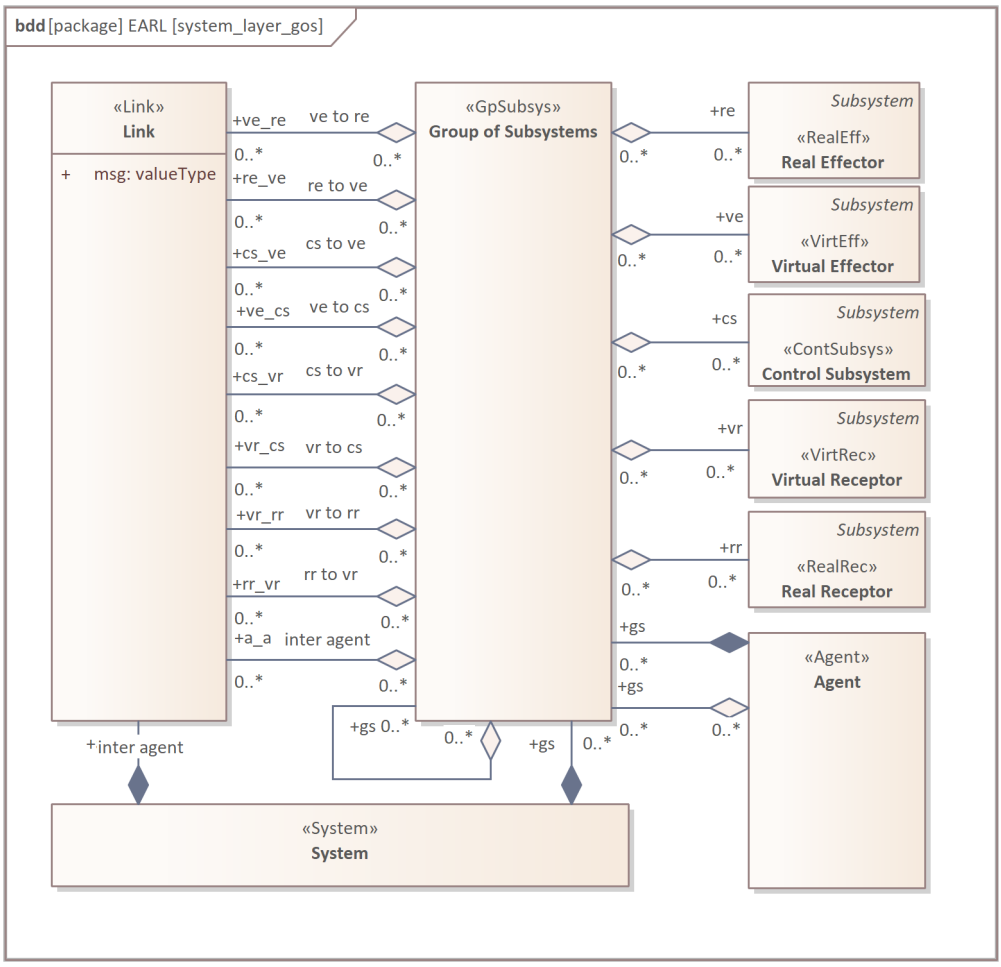


Figure 4: Group of Subsystems

If the Subsystems from various Agents are grouped into the single Group of Subsystems, then this group is a part of System labelled as *gs* that can be also aggregated as a reference *gs* into Agents. The Subsystems of single Agent can aggregate into its internal part – Group of Subsystems *gs*. The Group of Subsystems can also refer to other Group of Subsystems as *gs*. As the example the Group of Subsystems can aggregate the Subsystems that run in a single operating system's process, use the same communication medium (e.g., WiFi), or form a particular robot control system.

The Group of Links (Figure 5) was introduced for the purpose of aggregated presentation of communication links between various blocks. It aggregates Links. Especially, in opposition to particular Links, the Group of Links can represent both uni and bi directional communication. The Group of Links can be composed or aggregated into various blocks as *gl*. The recursive self reference of Groups of Links is also possible.

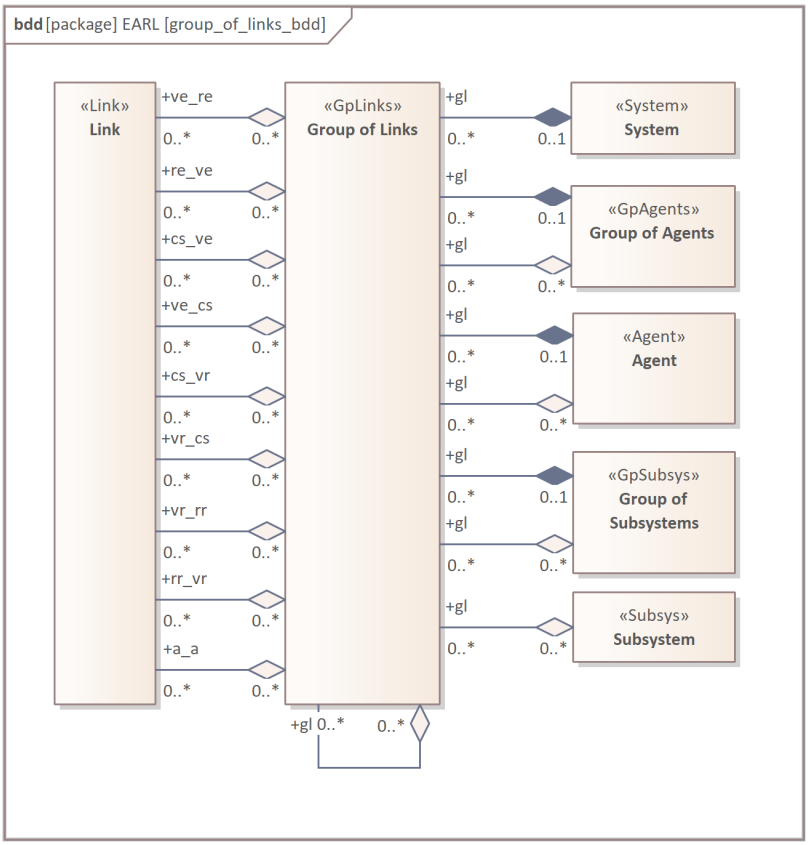


Figure 5: Group of Links

2.2 Subsystem and its Parts

The composition of a Subsystem is defined in Figures 6, 7 and 8(**a**). 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), multiple internal Finite State Machines *ifsm*, Predicates *p*, Basic Behaviours *bb* and Primitive Transition Functions *pf*.

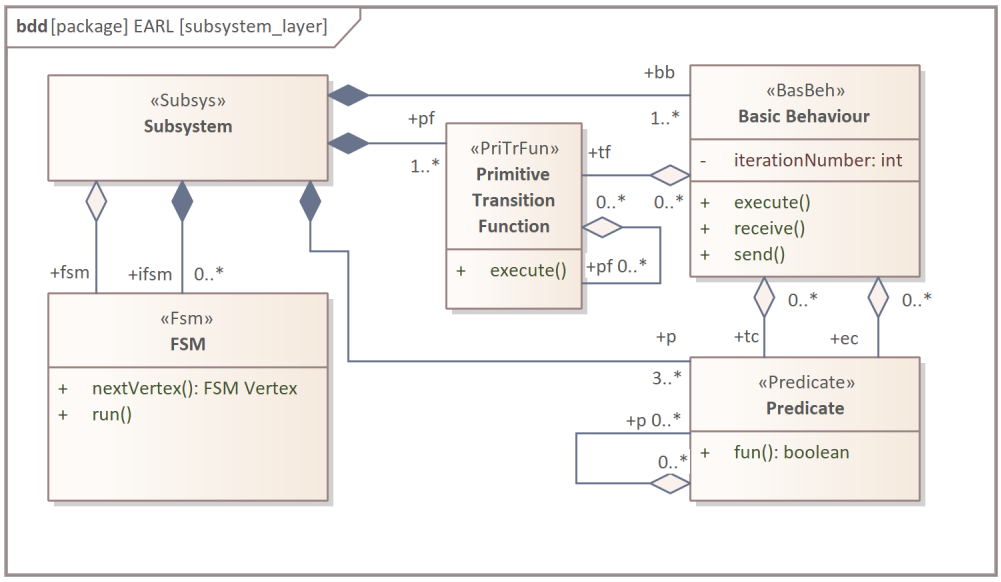


Figure 6: Subsystem and its parts (Input, Output Buffers, Internal Memory and Hierarchical FSM are excluded)

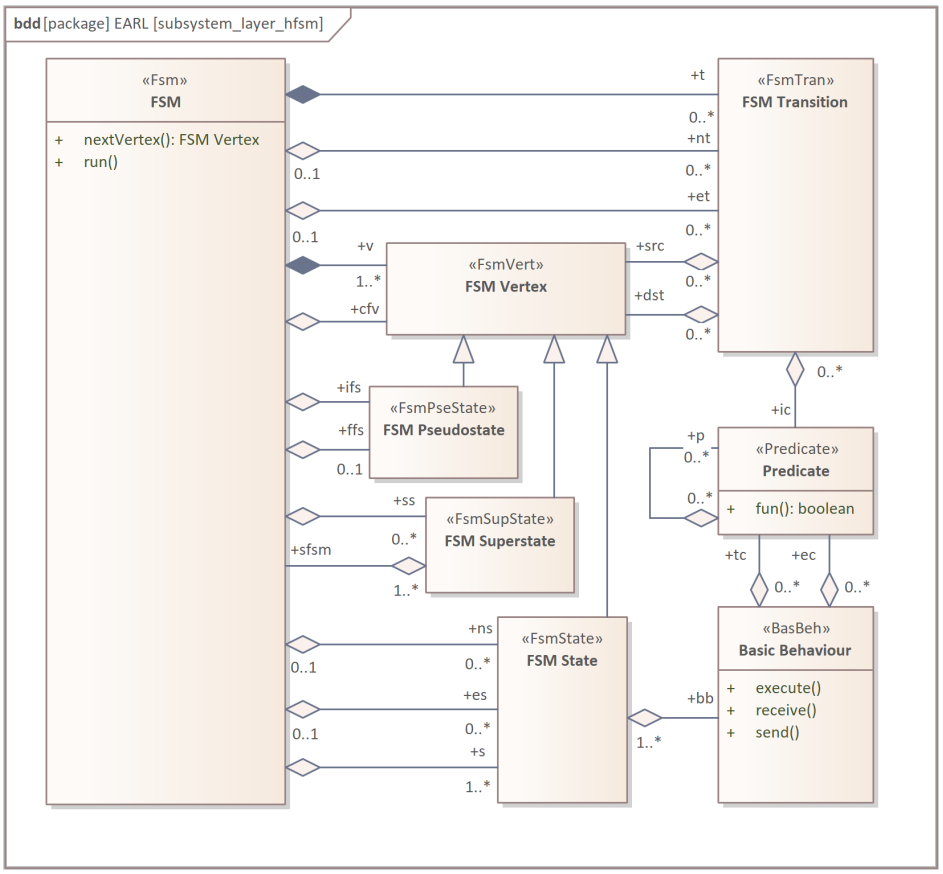
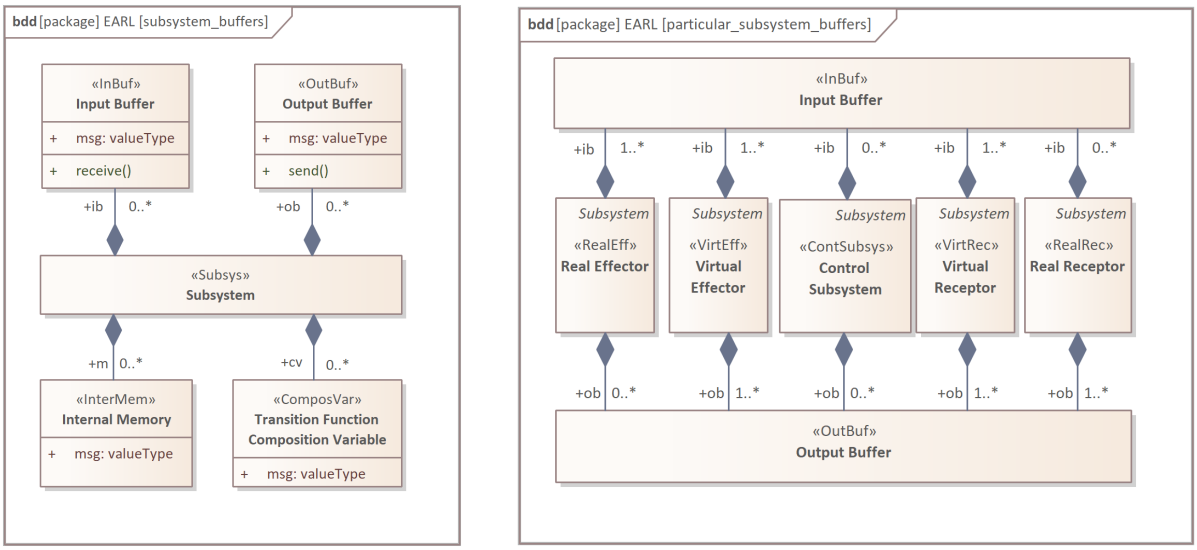


Figure 7: Subsystem hierarchical FSM

Figure 8(**b**) 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 8: 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 [12]. Each Buffer contains a data structure *msg*, which stores data of type valueType. The valueType 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 *msg*, which is a value of type valueType. Transition Function Composition Variables were introduced for the purpose of Communication between Primitive Transition Functions to compose Transition Function. Transition Function Composition Variable, analogically to Internal Memory, stores *msg*, which is a value of type valueType. Various forms of communication between Subsystems have been described in the paper [18].

Similarly to [2, 7], the EARL Subsystem structural model contains a Finite State Machine (FSM) that determines its activities (Figure 6). The FSM is a graph defined by a set v of FSM Vertices and a set t of FSM Transitions. Current FSM Vertex is called *cfv*. There are three types of FSM Vertices. The first one is FSM Pseudostate. Each FSM can have one initial FSM State – *ifs* and zero or one final FSM State – *ffs*. Those states point the initial and final FSM Vertex. The second type is FSM Superstate – *ss*. Each FSM can aggregate many *ss*. Each FSM Superstate aggregates one FSM called *sfsm*. The last one is FSM State. Each FSM State aggregates one Basic Behaviour. The FSM States can be aggregated into two groups: normal FSM States *ns* and error handling FSM States *es*. Similarly FSM Transitions can be aggregated into: normal FSM Transitions *nt* leading to normal FSM States and error handling FSM Transitions *et* leading to error handling FSM States. A Predicate specified in operation $fun()$ of FSM Transitions *nt* should satisfy at least the Predicate (2)

$$errTran/p.fun() := errorTransiton/mi.msg, \tag{2}$$

and analogically a Predicate of FSM Transitions *et* should satisfy at least the Predicate (3)

$$\neg errTran/p.fun() := \neg errorTransiton/mi.msg. \tag{3}$$

Figure 9 defines how the $run()$ operation of FSM works. In each moment of the Subsystem running there is exactly one FSM Vertex active. It is called current FSM Vertex – *cfv*. The FSM starts in the initial FSM State *ifs*. In each iteration of $run()$ operation the type of *cfv* is checked. If *cfv* is of:

- FSM State type, it means that *cfv* is associated with the Basic Behaviour *bb*. Consequently the $cfv.bb.execute()$ operation executes a behaviour associated with the current vertex. Next, the function $nextVertex()$ choose new current FSM Vertex – *cfv*.
- FSM Pseudostate type, there is a question if it is initial state or final state. It it is initial state, the function $nextVertex()$ choose new current FSM Vertex – *cfv*. Otherwise, the action of $run()$ operation is finished.
- FSM Superstate type, the $run()$ operation of sub-automata – $cfv.sfsm.run()$ – is executed.

FSM Transition (Figure 6) is defined by the source and destination FSM States as well as the associated Initial Condition, i.e., Predicate *ic*.

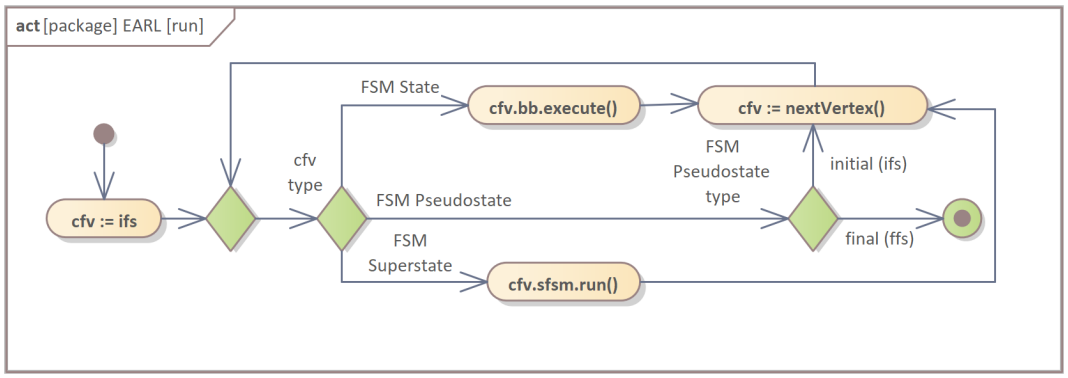


Figure 9: FSM.*run*() operation

In the following part of the documentation a SysML dot "." notation [9] 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 composition of a Basic Behaviour is defined in Figure 6. The Basic Behaviour includes its transition function *tf* which is referred to one of the Primitive Transition Functions; 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 10).

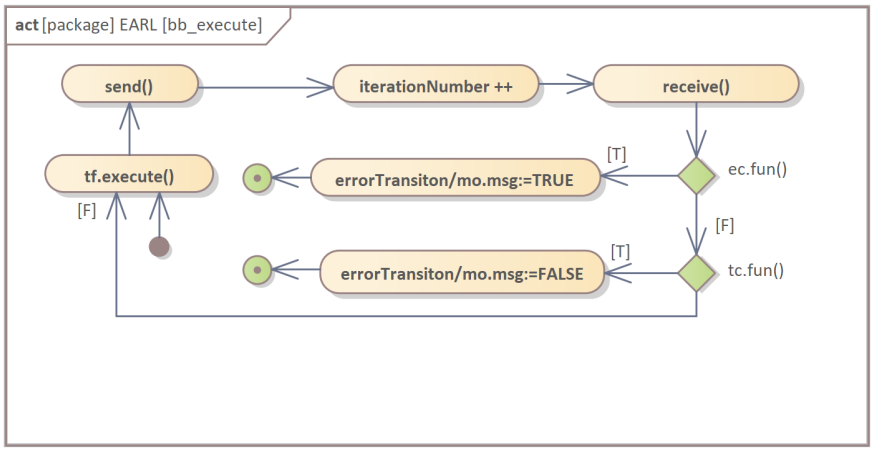


Figure 10: Basic Behaviour.*execute*() operation

That operation, first executes a transition function $tf.execute()$, then all calculated Output Buffers values are sent out by *send*(). Next, *iterationNumber* is incremented, and *receive*() gets new values into Input Buffers. Finally, Error Condition $ec.fun()$ and Terminal Condition $tc.fun()$ are tested. If both values are false, a new iteration inside Basic Behaviour *execute*() operation is performed. Otherwise, $errorTransiton/mo.msg$ is set, Basic Behaviour *execute*() *operation* ends and the FSM *run*() operation designates the next FSM State (Figure 9).

The composition of a Primitive Transition Function is defined in Figure 6. It refers to Input Buffers, Output Buffers as well as Subsystem Internal Memory (Figure 11).

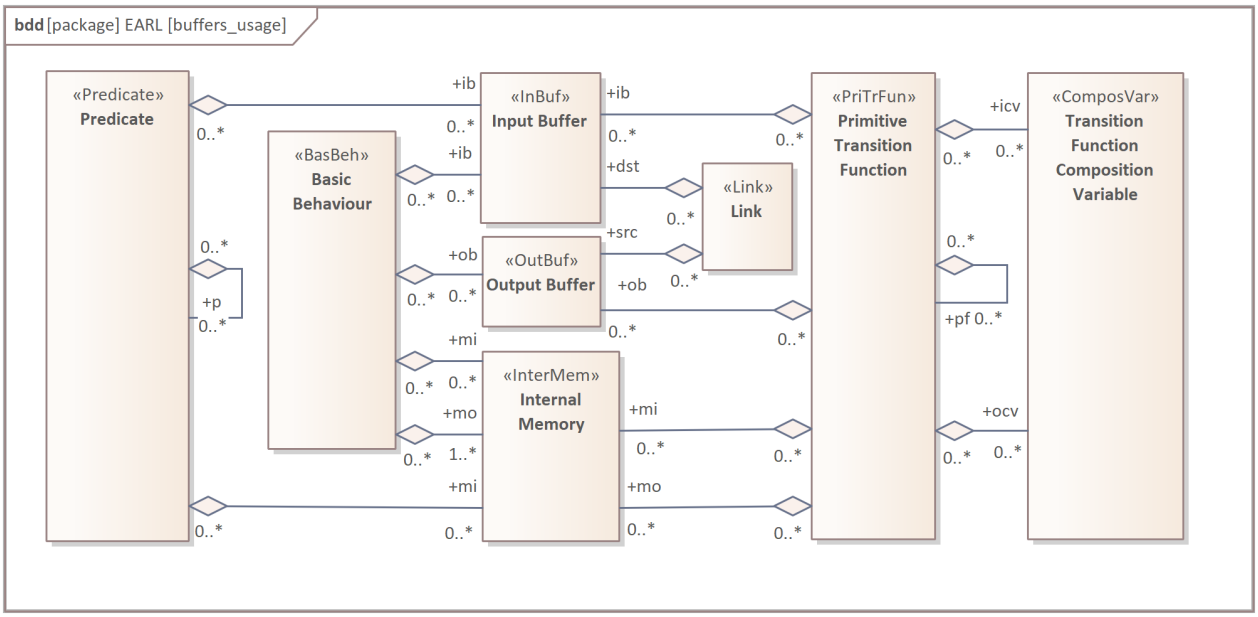


Figure 11: The utilisation of Buffers, Transition Function Composition Variables and Internal Memory by: Primitive Transition Function, Predicate and Links

A Primitive Transition Function can read from the Internal Memory (using the *mi* reference) or write to it (using the *mo* reference) and analogically read from the Transition Function Composition Variable (using the *icv* reference) or write to it (using the *ocv* reference). It can aggregate other Primitive Transition Functions. The aggregation can be modelled by directed graph, in case of Primitive Transition Functions aggregation cycles are not allowed. Such an aggregation sometimes reduces the redundancy of the specification, making it more comprehensible. Moreover, if the implementation of the specified system is based on components, a Primitive Transition Function can be identified with a separate component or a set of components [4, 13]. The Primitive Transition Function algorithm is executed by an $pf.execute()$ operation. It can use a.o. components from the Calculation Components Package (Figure 1). The concept of the Embodied Agent introduces no restrictions on how to implement this operation.

Terminal Conditions used by a Basic Behaviour and Initial Conditions utilised within an FSM Transition can be decomposed into Predicates. The Predicate takes its arguments from Subsystem Buffers, see Figures 6 and 11. Predicate executes an operation called fun producing a Boolean output. Similarly to Primitive Transition Functions, Predicate can aggregate other Predicates, with the same no-cycles restriction.

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 [1]. This provides guidance to the designers, which results in a clear system structure.

In the case of EARL, inter-agent and inter-subsystem communication [17] is defined by unidirectional communication Links (see Figures 2 and 11). The communication takes place between Input Buffers and Output Buffers of Subsystems. Figure 12 presents acceptable communication links between pairs of Subsystems. Note that the inter-agent communication is realised between the Control Subsystems of the communicating agents. Additionally, Figure 12 shows that for each Real Effector present in the system at least one transmission chain should exist. The commands produced by the

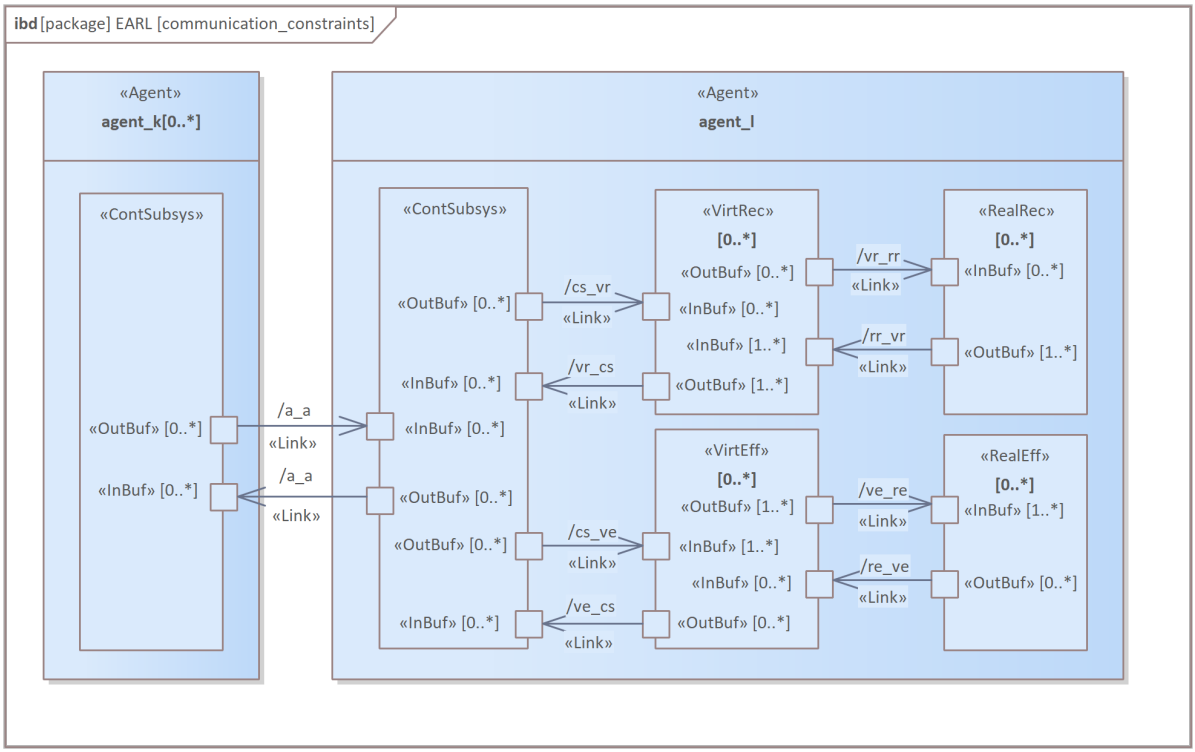


Figure 12: Communication constraints, where $agent\_k \neq agent\_l$

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 aggregated form passes it to the Control Subsystem. The other communication Links appearing in Figure 12 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 [18].

2.4 Types of Agents

Four general activities of an Agent can be distinguished [16]:

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 1), depending on their capabilities. However, only seven are of utility, as an agent without any of the optional capabilities is useless.

Table 1: Type of Agent, number of its Subsystems ($|ve|$, $|re|$, $|vr|$, $|rr|$) and number of inter agent communication Links ($|a\_a|$) expressed with respect to the number of Buffers of the considered Agent

	$\|cs\|$	$\|ve\|$	$\|re\|$	$\|vr\|$	$\|rr\|$	$\|a\_a\|$	**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 specialisations of blocks [5] 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 Primitive 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 a_a between them. It is a good practice to name Links by using the names of communicating Agents: first the source Agent name, then a underscore, and finally the destination Agent name. Input Buffers and Output Buffers of the Control Subsystems are depicted as sources and destinations of valueTypes being transmitted through the Links. The Buffer names reflect the content of valueType 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. To specify a particular system, instances of the relevant concepts appearing in the general system model should be concretised. The SysML diagrams [10] are a part of the EARL-based system. 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
- some calculations performed inside actions of Activity Diagrams of Primitive Transition Functions, e.g., control laws.

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

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:

- General system use cases.
- Structure of the whole System with Buffers, Internal Memories, inter Agent communication Links, and valueTypes used by them.

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

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 [3, 11].

3.1 System use cases

Diagram 13 illustrates general system use cases. The User can request the robot to:

- Synchronize the robot's drives,
- Perform the movement in joint or operational space,
- Perform the emergency stop action, or
- Perform Pick and Place task, which is composed of the movement in joint and operational space, and in certain conditions can be ended by the emergency stop action.

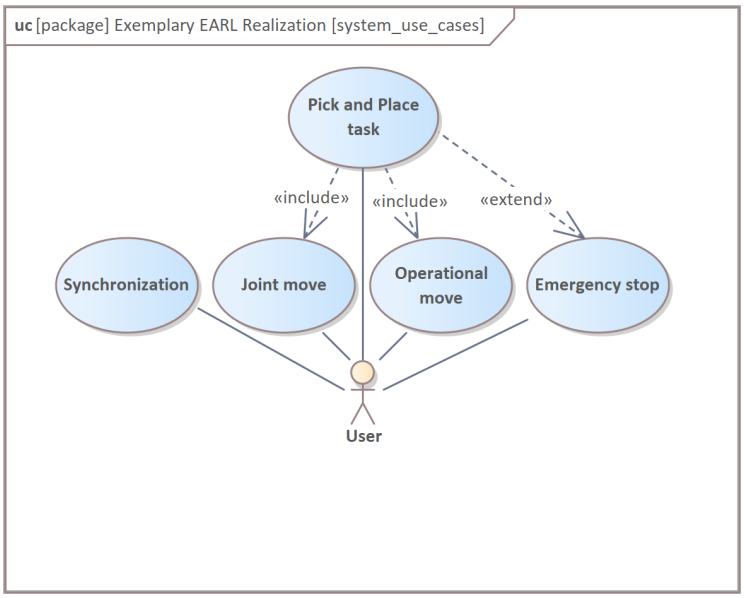


Figure 13: System use cases

3.2 Structure of the System Composed of Agents

Diagram 14 illustrates structure of Value Types and Units packages used in the presented System.

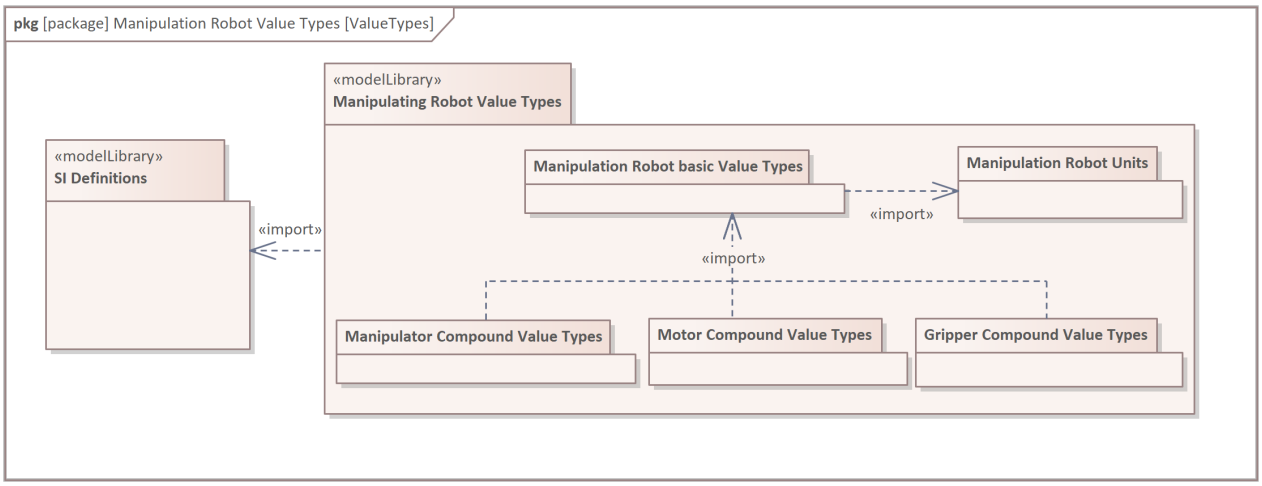


Figure 14: Value Types and Units packages structure

Diagram 15 presents Units definitions, while diagram 16 illustrates basic Value Types used in the Compound Value Types definitions.

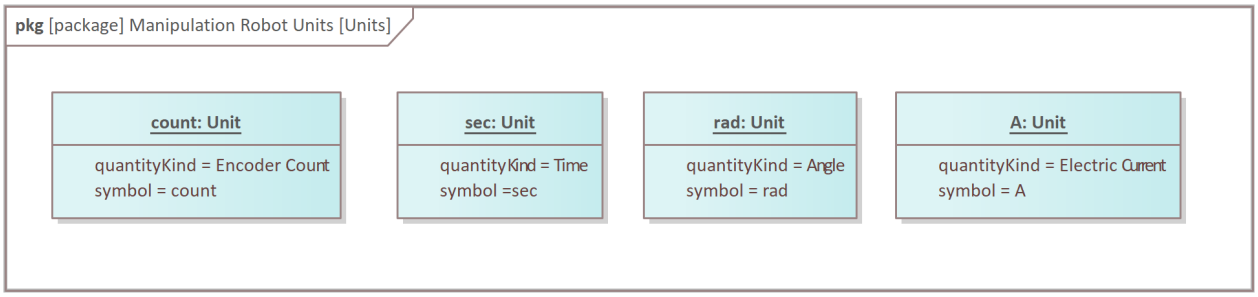


Figure 15: Units definitions

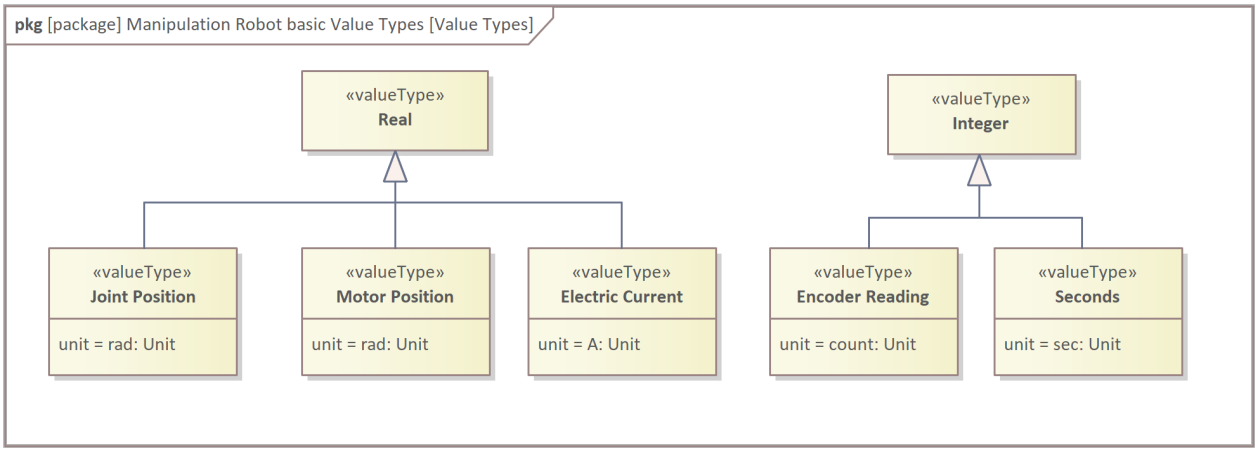


Figure 16: Basic Value Types definitions

There are three Agents in the System (Figure 17). 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. Both *manip/a* and *grip/a* are aggregated in Group of Agents *arm/ga*. Figure 18 shows alternative representation of the exemplary System – in this case full names of blocks were depicted.

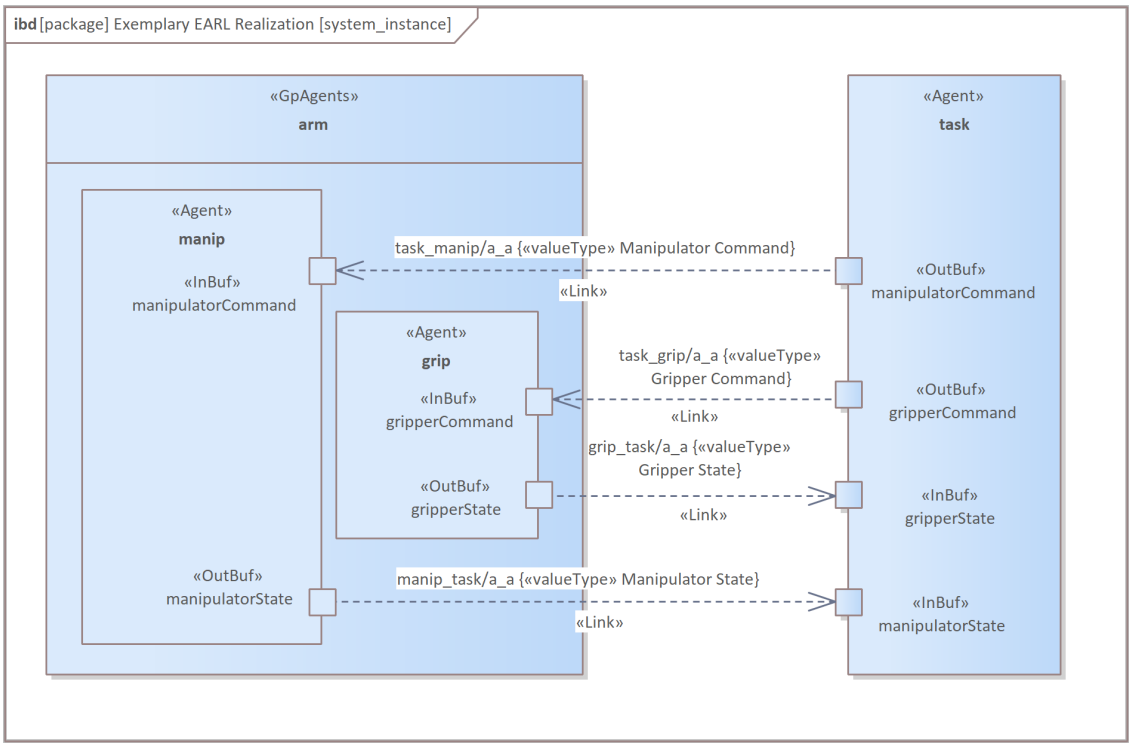


Figure 17: Structure of the considered exemplary System

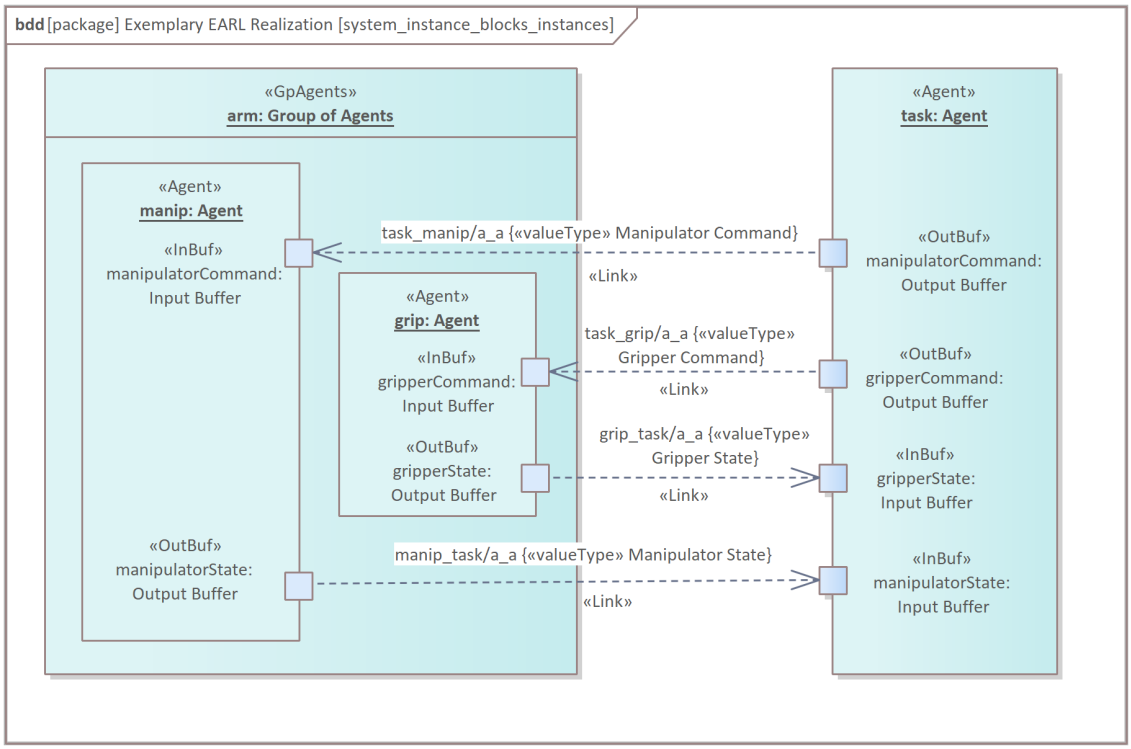


Figure 18: Structure of the considered exemplary System (full blocks names)

Figure 19 presents the valueTypes transmitted between the Agents. The task Agent *task/a* sends Manipulator Commands 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 Manipulator State valueType 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 Gripper Command messages to the gripper Agent *grip/a* and receives Gripper Status in return. Similarly to messages exchanged between *manip/a* and *task/a* Agents, the Gripper Command and Gripper Status messages contain parameters describing the desired and current gripper finger positions. Both *manip/a* and *grip/a* are aggregated in Group of Agents *arm/ga*.

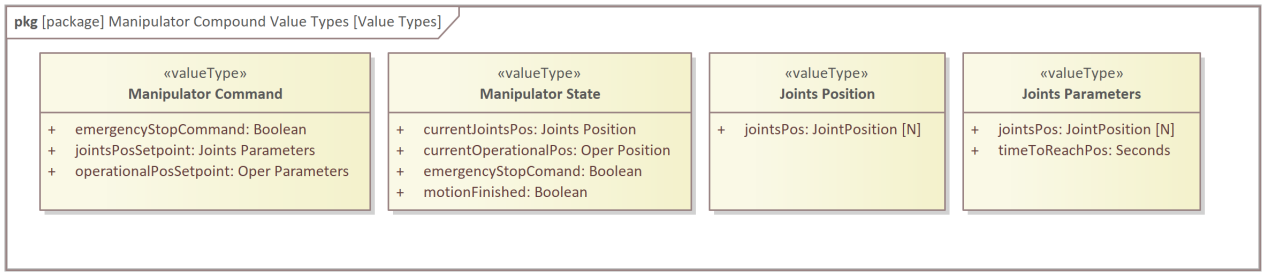


Figure 19: valueTypes transmitted within the system

3.3 manipulator Agent *manip/a*

The structure of the manipulator Agent *manip/a* is presented in Figure 20. 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. Both Virtual Effectors and Real Effector are aggregated in Group of Subsystems *motors/gs*.

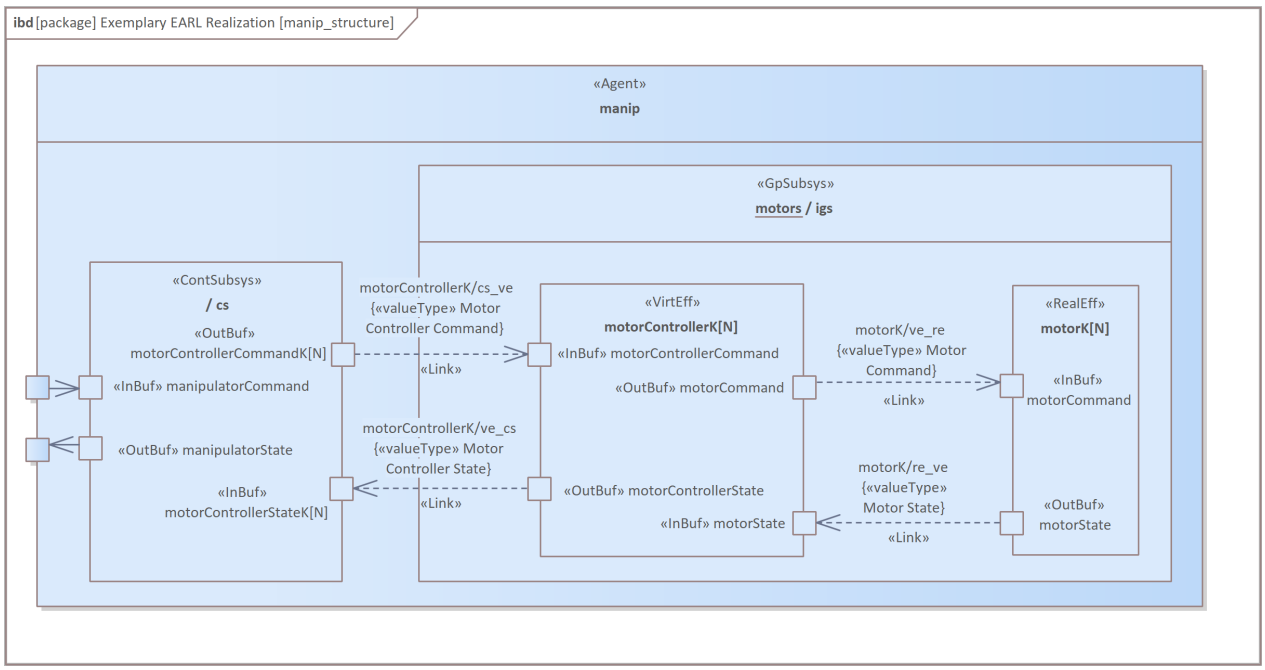


Figure 20: Structure of the Agent *manip/a*; letter K placed at the end of the instance name should be substituted by a number, i.e. $K \in \{1, ..., N\}$

The valueTypes transmitted inside the manipulator Agent *manip/a* are presented in Figure 21. The Control Subsystem *cs* sends MotorControllerCommand to each Virtual Effector *motorControllerK/ve*. The valueType 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 2 describes types of data utilized in the *manip/a.cs*.

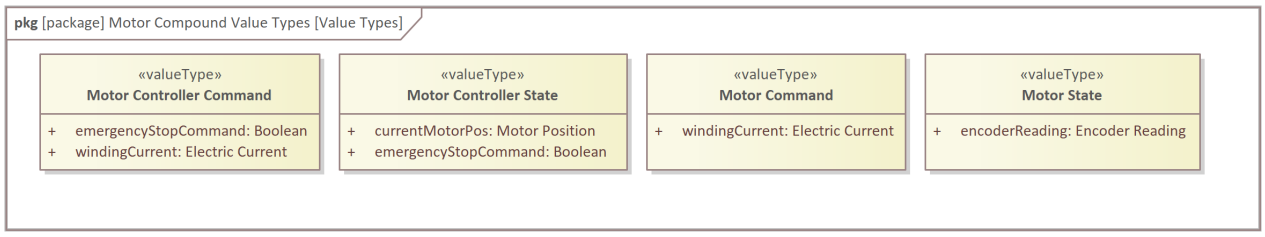


Figure 21: Definition of *manip/a* valueTypes

Table 2: *manip/a.cs* valueTypes

	valueType
motionFinished/m	Boolean
motionFinished/cv	Boolean
currentOperationalPos/cv	Oper Position
currentJointsPos/cv	Joints Position
emergencyStopCommandK/cv	Boolean
windingCurrentK/cv	Real

Table 3 describes Predicates p 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 3: Definitions of $manip/a.cs.\boldsymbol{p.fun}$

$/\boldsymbol{p}$	$\boldsymbol{fun}$
$emergencyStop$	$manipulatorCommand/ib.msg.emergencyStopCommand \vee$ $motorControllerState1/ib.msg.emergencyStopCommand \vee \ldots \vee$ $motorControllerStateN/ib.msg.emergencyStopCommand$
$motionFinished$	$motionFinished/mi.msg$
$newJointsPos$	$newData(manipulatorCommand/ib.msg.jointsPosSetpoint)$
$newOperationalPos$	$newData(manipulatorCommand/ib.msg.operationalPosSetpoint)$
$true$	TRUE
$false$	FALSE

Tables 4, 5 describe the Predicates utilised by $manip/a.cs$. Figure 22 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 4: Initial conditions labelling $manip/a.cs.fsm$ transitions. 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**	
$\boldsymbol{t}$	**PREDICATE**
$idle\_jointMove/nt$	$\neg errTran/p.fun() \wedge newJointsPos/p.fun$
$jointMove\_jointMove/nt$	$\neg errTran/p.fun() \wedge newJointsPos/p.fun$
$idle\_operationalMove/nt$	$\neg errTran/p.fun() \wedge newOperationalPos/p.fun$
$operationalMove\_operationalMove/nt$	$\neg errTran/p.fun() \wedge newOperationalPos/p.fun$
$jointMove\_idle/nt$	$\neg errTran/p.fun()$
$operationalMove\_idle/nt$	$\neg errTran/p.fun()$
$i\_emergencyStop/et, i \neq emergencyStop$	$errTran/p.fun()$

Table 5: Terminal and error conditions of $manip/a.cs.bb$

Definitions of Terminal Conditions and Error Conditions $cs.\textbf{BB\_CONDITION}.fun \triangleq$ **PREDICATE**	
BB_CONDITION	**PREDICATE**
$idle/bb.tc$	$newJointsPos/p.fun \vee newOperationalPos/p.fun$
$idle/bb.ec$	$emergencyStop/p.fun$
$jointMove/bb.tc$	$motionFinished/p.fun$
$jointMove/bb.ec$	$emergencyStop/p.fun$
$operationalMove/bb.tc$	$motionFinished/p.fun$
$operationalMove/bb.ec$	$emergencyStop/p.fun$
$emergencyStop/bb.tc$	$false/p.fun$
$emergencyStop/bb.ec$	$false/p.fun$

The Control Subsystem $manip/a.cs$ uses the following Primitive 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 23).
- *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 24(**a**)).
- *outputMotorCon/pf*—composes *DriveControllerCommandK/ob* (Figure 24(**b**)).

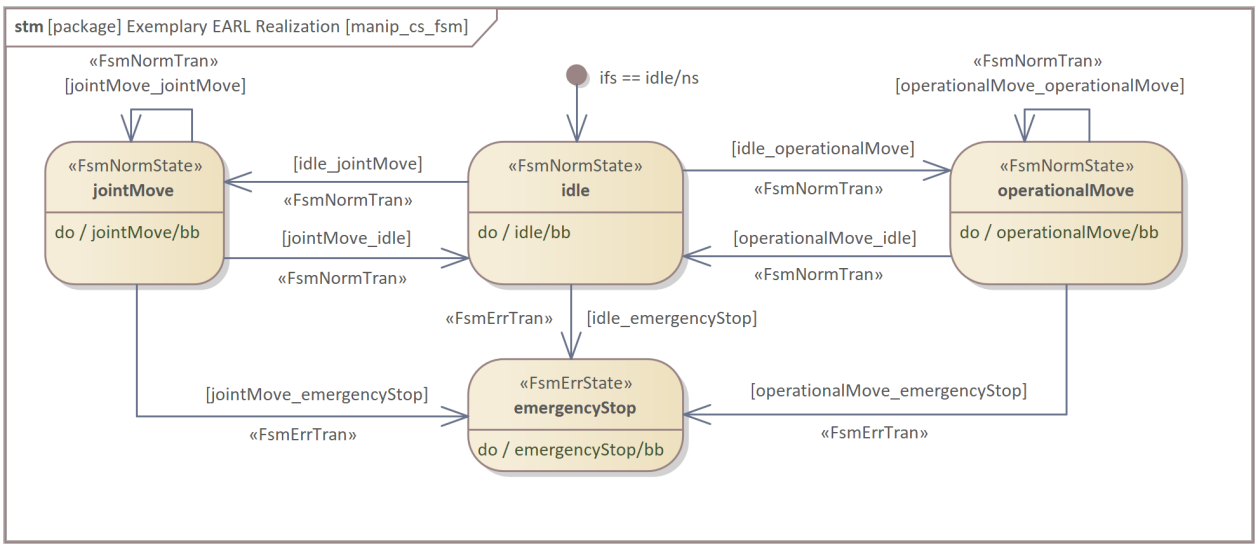


Figure 22: *manip/a.cs.fsm* definition. Conditions of transitions between FSM States are specified in Table 4

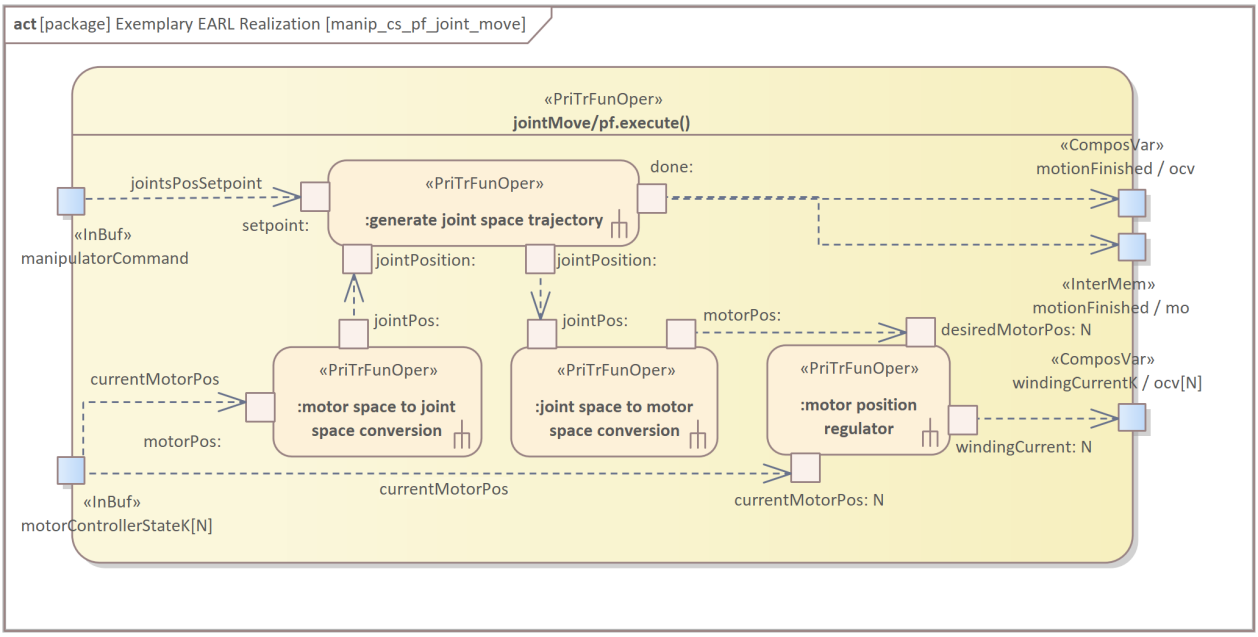


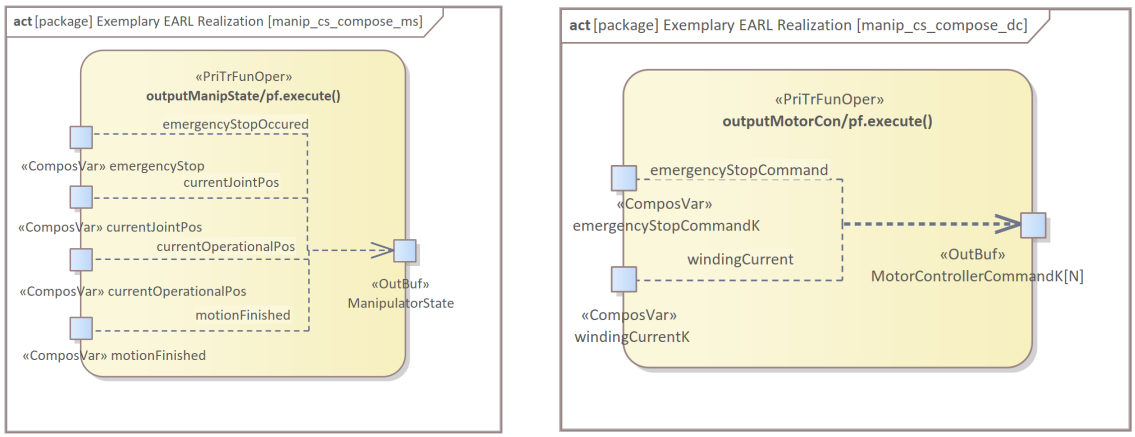
Figure 23: *manip/a.cs.jointMove/pf.execute()* – operation definition

Figure 23 shows the execute operation of a *jointMove/pf* Primitive Transition Function. This Primitive Transition Function realises, e.g., the PI type *motor position regulator* for each joint Equation (4), Equation (5):

$$windingCurrent = K_p\, e(t) + K_i \int_0^t e(t')dt', \tag{4}$$

$$e = desiredMotorPos - currentMotorPos, \tag{5}$$

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



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

Figure 24: *manip/a.cs.pf.execute()* – operations definition

Figures 24(**a**) and 24(**b**) show the execute operation of a *outputManipState/pf* and *outputMotorCon/pf* respectively. Those Primitive Transition Functions define the composition of Output Buffer messages from certain Transition Function Composition Variables entities.

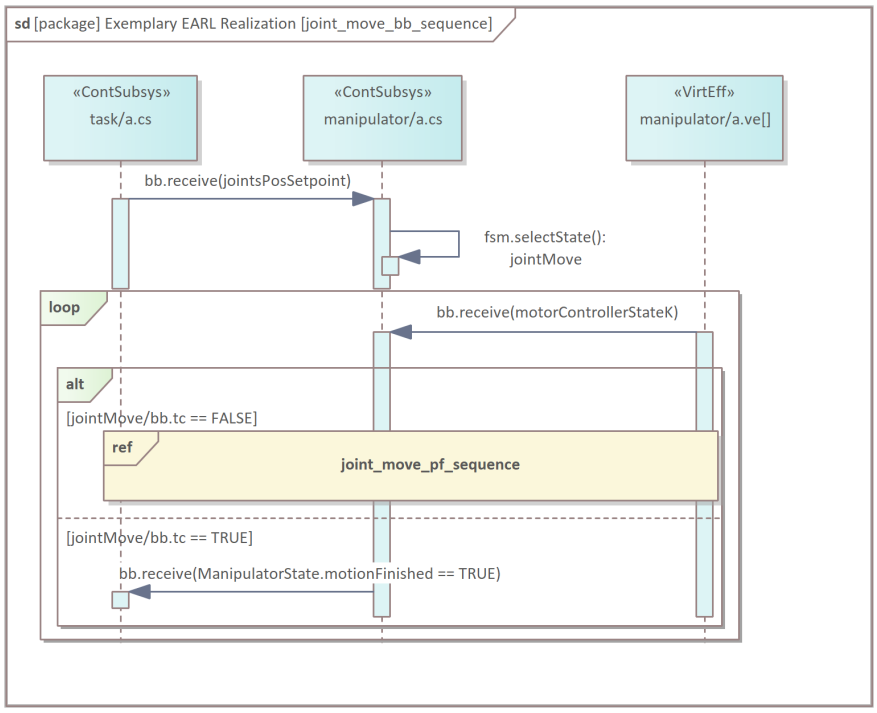


Figure 25: *jointMove/bb* – main sequence of actions execution

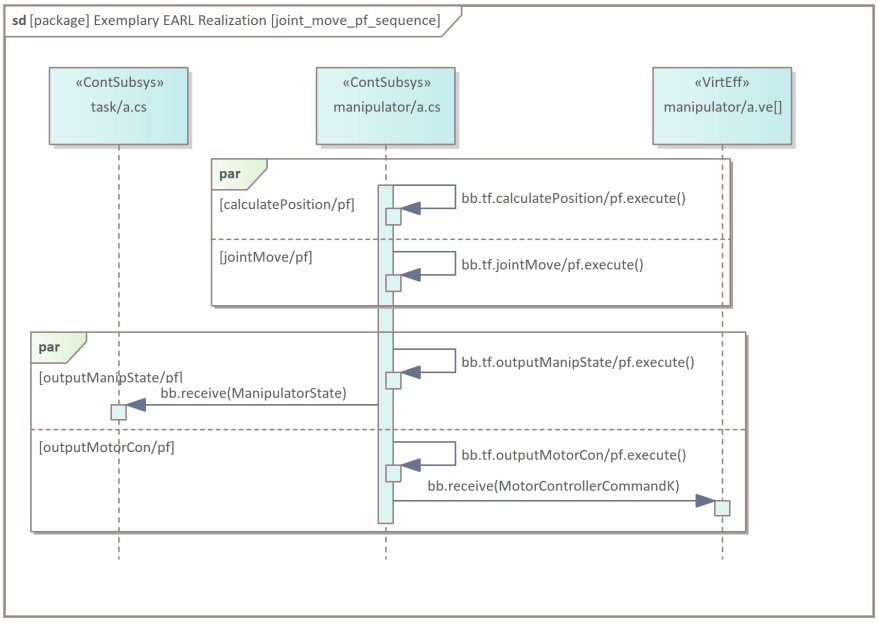


Figure 26: *jointMove/bb* – main sequence of associated *pf* execution

Figures 25 and 26 show how the operations *jointMove/pf*, *calculatePosition/pf*, *outputManipState/pf* and *outputMotorCon/pf* are executed. In particular it shows:

- the order of *pf* execution, and
- moments when communication with associated subsystems takes place.

Table 6 shows which Primitive Transition Functions constitute the composition of transition function used by Basic Behaviours of *manip/a.cs.bb*.

The Primitive Transition Functions $manip/a.cs.pf$ composed into Primitive Transition Function $manip/a.cs.bb.tf$ are subdivided into two disjoint sets: pfc and pfo. The pfc set Equation (6) contains Primitive Transition Functions that take as arguments Input Buffers: $manipulatorCommand/ib$ and $motorControllerCommandK/ib[N]$

$$pfc = \{calculatePosition/pf, jointMove/pf, operationalMove/pf, passiveRegulation/pf, emergencyStop/pf\}. \tag{6}$$

The values produced by them are inserted into the Transition Function Composition Variables $ocvs$ Equation (7)

$$ocvs = \{motionFinished/ocv, currentOperationalPos/ocv, currentJointsPos/ocv, emergencyStopCommandK/ocv, windingCurrentK/ocv\}. \tag{7}$$

and the Internal Memory mbo Equation (8)

$$mbo = \{motionFinished/mo\}. \tag{8}$$

Functions from the pfo set Equation (9) take arguments from the Transition Function Composition Variables $icvs$ associated with these from set $ocvs$ Equation (7) and produce the Output Buffer values: $ManipulatorState/ob$ and $MotorControllerCommandK/ob[N]$, hence they produce output of the whole Subsystem

$$pfo = \{outputManipState/pf, outputMotorCon/pf\}. \tag{9}$$

It was assumed that any two Primitive Transition Functions used by a particular Primitive Transition Function $manip/a.cs.bb.tf$ do not produce data to the same Transition Function Composition Variables, Internal Memories and Output Buffers, therefore the following conditions are formulated for the pfc set Equations (10), (11) and the pfo set Equation (12), respectively,

$$(\forall x/bb)(\forall x/bb.i/pf, x/bb.j/pf \in pfc,\ i \neq j)(x/bb.i/pf.k/ocv \nsim x/bb.j/pf.k/ocv,\ k/ocv \in ocvs), \tag{10}$$

$$(\forall x/bb)(\forall x/bb.i/pf, x/bb.j/pf \in pfc,\ i \neq j)(x/bb.i/pf.k/mo \nsim x/bb.j/pf.k/mo,\ k/mo \in mbo), \tag{11}$$

$$(\forall x/bb)(\forall x/bb.i/pf, x/bb.j/pf \in pfo,\ i \neq j)(x/bb.i/pf.k/ob \nsim x/bb.j/pf.k/ob), \tag{12}$$

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

Table 6: Aggregation of transition function *manip/a.cs.bb.tf* with Primitive Transition Functions *manip/a.cs.**bb**.tf.**pf***. The right part of the table presents what parts of Output Buffers, Transition Function Composition Variables and Internal Memories are produced by the specific Primitive Transition Functions

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

Transition functions *manip/a.cs.bb.tf* act in the following way. First, they compute the Primitive Transition Functions from the *pfc* set, and then they compute the Primitive Transition Functions from the *pfo* set. The fulfilment of Equations (10), (11) and (12) makes it possible to run Primitive Transition Functions being members of *pfc* in parallel in the first stage of the Primitive Transition Function *manip/a.cs.bb.tf* execution, and then run the Primitive Transition Functions being members of *pfo* set in parallel in the second stage of Primitive Transition Function *manip/a.cs.bb.tf* execution. To illustrate the above considerations, Figure 27 shows the definition of *jointMove/bb.tf.execute*() operation – practical realisation of transition function execution for *jointMove* Basic Behaviour.

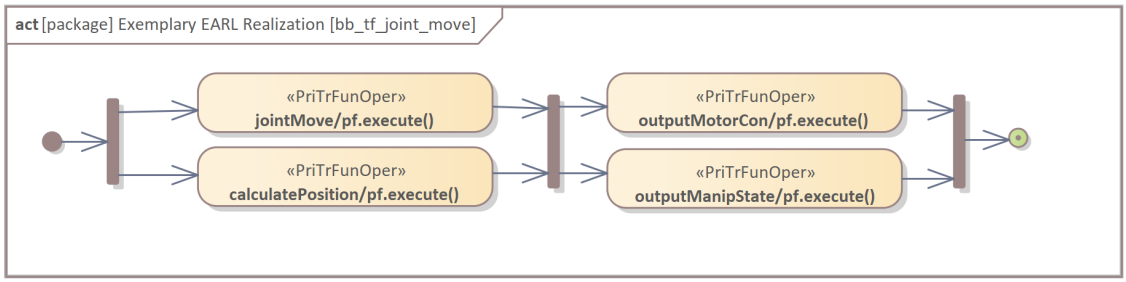


Figure 27: *jointMove/bb.tf.execute*() operation definition

4 Exemplary EARL customisations

The convenient way to customize EARL model is to specify a new model and include it in the dedicated SysML package. The Blocks of the new model can derive from Basic Model blocks.

4.1 Agent specialisation

The Agent block can be specialised to, e.g., group Agents with certain properties and to define communication structure of particular System. The composition of exemplary Search and Rescue System is depicted in Figure 28.

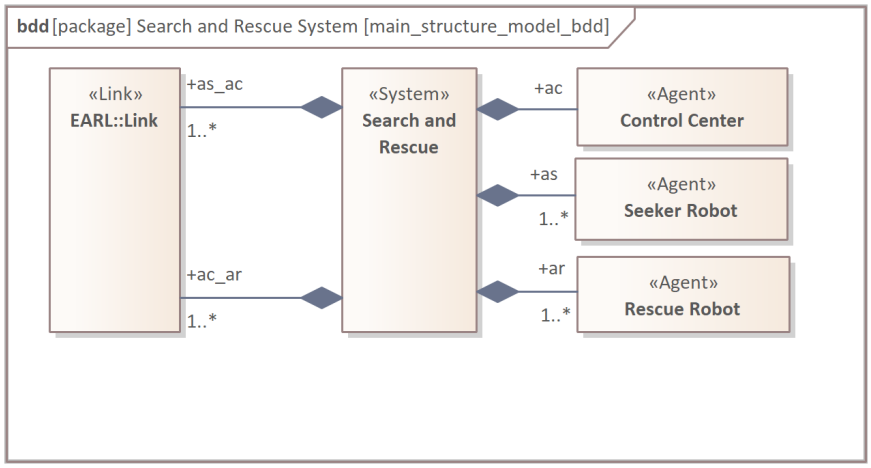


Figure 28: The composition of Search and Rescue System

The communication structure and constraints are presented in Figure 29. Control Center Agent plays a role of communication broker between Seeker Robot Agents and Rescue Robot Agents. The communication in this example is unidirectional.

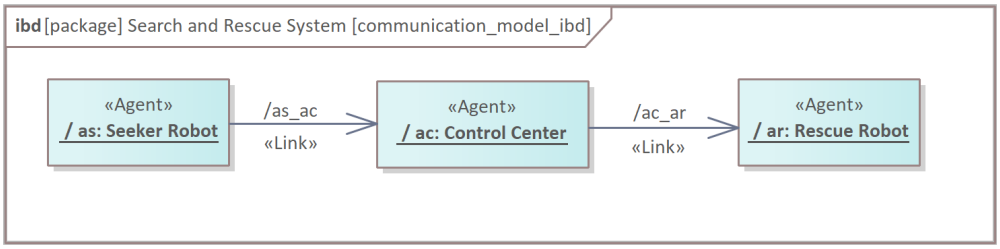


Figure 29: The communication structure and constraints of Search and Rescue System

It should be noted that System parts: a, a_a should not be used in the inherited Systems without redefinition that keeps model consistency. In Search and Rescue System these parts are not redefined, so they are not used. Hence, the general structure of an exemplary instance of Search and Rescue System may take form as in Figure 30. It consists of: 3 seeker robots, 2 rescue robots, control centre and communication Links between previously mentioned parts. It should be noted that for the sake of brief, general system structure description the ports have not to be directly specified in ibd, as well as Links names.

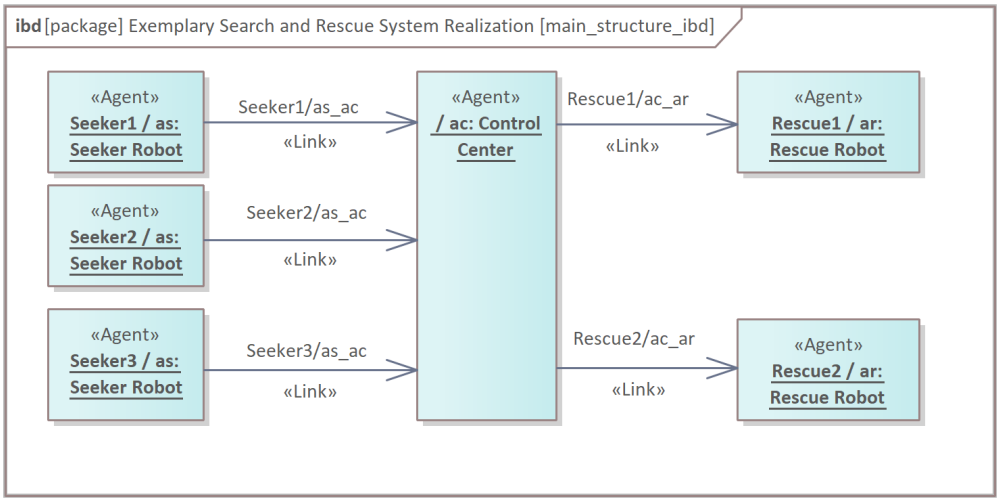


Figure 30: The structure of an exemplary instance of Search and Rescue System

5 Major changes in current version

1. Stereotypes introduced (authors: Tomasz Winiarski).

References

[1] S. Dennis, L. Alex, L. Matthias, and S. Christian. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software". In: *Journal of Software Engineering in Robotics* 7.1 (2016), pages 3–19.

[2] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications". In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Nov. 2012, pages 149–160. DOI: 10.1007/978-3-642-34327-8_16.

[3] M. dos Santos Soares and J. Vrancken. "Requirements specification and modeling through SysML". In: *IEEE International Conference on Systems, Man and Cybernetics*. 2007, pages 1735–1740. DOI: 10.1109/ICSMC.2007.4413936.

[4] W. Dudek, W. Szynkiewicz, and T. Winiarski. "Nao Robot Navigation System Structure Development in an Agent-Based Architecture of the RAPP Platform". In: *Recent Advances in Automation, Robotics and Measuring Techniques*. Edited by R. Szewczyk, C. Zieliński, and M. Kaliczyńska. Volume 440. Advances in Intelligent Systems and Computing (AISC). Springer, 2016, pages 623–633. DOI: 10.1007/978-3-319-29357-8_54.

[5] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: The systems modeling language. 3rd ed.* Elsevier, Morgan Kaufmann, 2015.

[6] J. Karwowski, W. Dudek, M. Węgierek, and T. Winiarski. "HuBeRo-a Framework to Simulate Human Behaviour in Robot Research". In: *Journal of Automation, Mobile Robotics and Intelligent Systems* 15.1 (2021), pages 31–38. DOI: 10.14313/JAMRIS/1-2021/4.

[7] T. Kornuta and C. Zieliński. "Robot control system design exemplified by multi-camera visual servoing". In: *Journal of Intelligent and Robotic Systems* 77.3–4 (2013), pages 499–524. DOI: 10.1007/s10846-013-9883-x.

[8] T. Kornuta, C. Zieliński, and T. Winiarski. "A universal architectural pattern and specification method for robot control system design". In: *Bulletin of the Polish Academy of Sciences: Technical Sciences* 68.No. 1 February (2020), pages 3–29. DOI: 10.24425/bpasts.2020.131827.

[9] *OMG Systems Modeling Language - Version 1.6*. accessed on 4 April 2020. Open Management Group. Dec. 2019. URL: https://www.omg.org/spec/SysML/1.6/.

[10] A. Salado and P. Wach. "Constructing True Model-Based Requirements in SysML". In: *Systems* 7.2 (2019). ISSN: 2079-8954. DOI: 10.3390/systems7020019.

[11] M. Soares, J. Vrancken, and A. Verbraeck. "User requirements modeling and analysis of software-intensive systems". In: *Journal of Systems and Software* 84 (Feb. 2011), pages 328–339. DOI: 10.1016/j.jss.2010.10.020.

[12] P. Trojanek. "Design and implementation of robot control systems reacting to asynchronous events". PhD thesis. Warsaw University of Technology, 2012.

[13] T. Winiarski, K. Banachowicz, M. Walęcki, and J. Bohren. "Multibehavioral position–force manipulator controller". In: *21th IEEE International Conference on Methods and Models in Automation and Robotics, MMAR'2016*. IEEE, 2016, pages 651–656. DOI: 10.1109/MMAR.2016.7575213.

[14] T. Winiarski, W. Dudek, M. Stefańczyk, Ł. Zieliński, D. Giełdowski, and D. Seredyński. "An intent-based approach for creating assistive robots' control systems". In: *arXiv preprint arXiv:2005.12106* (2020). URL: http://arxiv.org/abs/2005.12106.

[15] T. Winiarski, M. Węgierek, D. Seredyński, W. Dudek, K. Banachowicz, and C. Zieliński. "EARL – Embodied Agent-Based Robot Control Systems Modelling Language". In: *Electronics* 9.2 (2020), page 379. DOI: 10.3390/electronics9020379.

[16] C. Zieliński, T. Winiarski, and T. Kornuta. "Agent-Based Structures of Robot Systems". In: *Trends in Advanced Intelligent Control, Optimization and Automation*. Edited by J. Kacprzyk and et al. Volume 577. Advances in Intelligent Systems and Computing. 2017, pages 493–502. DOI: 10.1007/978-3-319-60699-6_48.

[17] C. Zieliński. "Transition-Function Based Approach to Structuring Robot Control Software". In: *Robot Motion and Control*. Edited by K. Kozłowski. Volume 335. Lecture Notes in Control and Information Sciences. Springer-Verlag, 2006, pages 265–286.

[18] C. Zieliński, M. Figat, and R. Hexel. "Communication Within Multi-FSM Based Robotic Systems". In: *Journal of Intelligent & Robotic Systems* 93.3 (2019), pages 787–805. ISSN: 1573-0409. DOI: 10.1007/s10846-018-0869-6.

[19] C. Zieliński and P. Trojanek. "Stigmergic cooperation of autonomous robots". In: *Journal of Mechanism and Machine Theory* 44 (Apr. 2009), pages 656–670.