Synthesis of Robotic System Controllers Using Robotic System Specification Language

Maksym Figat, *Member, IEEE*, and Cezary Zieliński, *Senior Member, IEEE*

***Abstract*—Robotic System Specification Language (RSSL) stems from the embodied agent approach to robotic system design. It enables the specification of both the structure and activities of a multi-robot multi-agent robotic system. RSSL specification can be verified and automatically transformed by its compiler into a six-layered Robotic System Hierarchical Petri Net (RSHPN). RSHPN models the activities and structure of the designed robotic system. The automatically generated RSHPN is loaded into RSHPN Tool modeling RSHPNs and automatically generating the controller code. This approach was validated on several robotic systems. The use of RSSL and RSHPN facilitates the synthesis of robotic system controllers.**

***Index Terms*—DSLs, MDE, parametric RSHPN meta-model, robot model synthesis, Robotic System Specification Language.**

I. Introduction

As the European SPARC project [1] indicates Model Driven Engineering (MDE) approach will play an important role in the design of robotic systems. It provides a toolchain composed of models, transformation mechanisms, and modeling languages (i.e. General Purpose (GPMLs) and Domain Specific (DSMLs) Modeling Languages). The MDE approach is ideal for the whole development process of complex, multi-domain systems, e.g. robotic systems. Unfortunately, MDE still faces many challenges [1], [2].

The majority of existing robotics MDE approaches use internal DSMLs (relying on the syntax and the execution semantics of their host languages) [2]. Internal DSLs are based on GPMLs, such as UML [3] (e.g. RobotML [4]), SysML [5], graphical DSMLs as used by SmartSoft [6]), and most popular in robotics Ecore (32 MDE approaches out of 63 analysed in [2] have used DSMLs based on Ecore. Internal DSMLs are bound to the execution context [7], i.e. translator, of the host language, which may result in a shift toward a platform-dependent model. In contrast to internal DSMLs, external DSMLs define their own syntax and semantics, thus typically rely on a fewer number of concepts, moreover directly associated with the application domain, and use simpler notation [8].

Manuscript received 15 September 2022; accepted 7 December 2022. Date of publication 14 December 2022; date of current version 28 December 2022. This letter was recommended for publication by Associate Editor E. Lanzarone and Editor J. Yi upon evaluation of the reviewers' comments. This work was supported by the Centre for Priority Research Area Artificial Intelligence and Robotics of Warsaw University of Technology within the Excellence Initiative through Research University (IDUB) Programme. *(Corresponding author: Maksym Figat.)*

The authors are with the Warsaw University of Technology, Institute of Control and Computation Engineering, Warsaw, Poland (e-mail: maksym.figat@pw.edu.pl; cezary.zielinski@pw.edu.pl).

Digital Object Identifier 10.1109/LRA.2022.3229231

Another problem often encountered in robotic MDE is the lack of a holistic abstract view of the general robotic system. Many of the existing model-based approaches focus mainly on modeling specific subdomains and types of robot tasks [7]. 60% of MDE approaches out of 63 analysed in [2] focus on terrestrial robots and robotic arms, neglecting other robots, not to mention a holistic view of a robot in general. As a result, the designer has no guidance on how to build the system from components. But the robot model should represent an integrated system, not a set of components [9].

DSLs used in robotics focus mainly on the implementation phase of the design process (93% of the papers surveyed in [7]), and only to a negligible extent on system specification (only 14 of the papers surveyed in [7]). Those focused on specifying robotic systems, are limited to only certain parts of the system, as in the case of [10], where only a set of skills is considered. Out of 137 articles analysed in [7] only 5 concerned DSLs taking into account architectural patterns, i.e. the majority of MDE approaches in robotics: 1) does not clearly guide the developer in the design of a robotic system, and 2) does not separate specification from implementation, thus rendering the utilised architectural pattern obscure, as pointed out by [11]. 54 MDE approaches out of 63 considered in [2] does not foster decoupling between the modeling concepts and the implementation technology. Moreover, the lack of explicit separation of the specification and implementation phases implies: 1) inability to develop platform independent model of a robotic system; 2) difficult integration with other languages/tools; 3) reduction of possible reuse of the developed model. From all this, we conclude that it is necessary to develop an external DSML language to express the activity of the whole robotic system. Here we propose a language, i.e. RSSL, which uses the concepts from the field of robotics, i.e. the embodied agent approach [12], [13], and facilitates the determination of parameters necessary to obtain the holistic model of a robotic system based on parameterised (RSHPN) [13].

This paper is structured as follows. Section II presents the contribution, Section III introduces the robotic system architectural pattern and RSHPN meta-model. Section IV presents RSSL, Section V describes the developed tools, Section VI focuses on data translation, while Section VII describes the conducted experiments. Section VIII presents the discussion and Section IX draws conclusions.

II. Contribution

In our latest works the activity of a whole robotic system was modeled using RSHPN (Fig. 1). This article is an extension of work [13], where a parametric RSHPN meta-model was proposed. Based on RSHPN meta-model and the provided

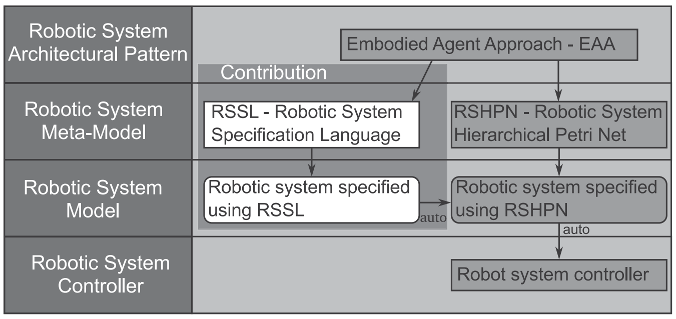

Robotic System Architectural Pattern
Embodied Agent Approach - EAA
Contribution
Robotic System Meta-Model
RSSL - Robotic System Specification Language
RSHPN - Robotic System Hierarchical Petri Net
Robotic System Model
Robotic system specified using RSSL
auto
Robotic system specified using RSHPN
auto
Robotic System Controller
Robot system controller

Fig. 1. Stages of robotic system software development.

parameters the RSHPN model (modeling the activity of a whole robotic system) emerges. The main contribution of this paper is the developed RSSL language and its compiler. RSSL expresses the parameters for a parametric RSHPN meta-model. Based on the developed RSSL specification and the RSHPN meta-model, the RSSL compiler generates the RSHPN model (Fig. 1) out of which the source code of the robotic system controller is generated. RSSL is a DSL facilitating the specification of multi-agent robotic systems $\mathcal{RS}$. It is based on the concepts derived from robotics, in particular it uses the Embodied Agent Approach (EAA) [13], [14] (Section III). RSSL enables the specification of a robotic system in a multi-layered and modular manner. It first defines the specification of the system structure and then its activities. RSSL uses a context-free grammar [15]. However, the language compiler analyses context sensitive dependencies at the semantic analysis stage. Moreover, RSSL is target language independent. As the result of indirect translation (Section VI), the RSSL specification is transformed into: ROS based Python or C++ code.

III. Embodied Agent Approach (EAA)

The methodology of designing multi-agent robotic control systems [13], [14], [16] uses a general robotic system meta-model separately describing system structure and activities.

A. Structure

A robotic system is composed of a set of embodied agents [12], [13], [16] (Fig. 2). An embodied agent $a_j \in \hat{a}$ (j – name of the agent, $\hat{a}$ – a set of agents) consists of the following subsystems: control subsystem c_j, real receptors $R_{j,l} \in \hat{R}_j$ (l – name of a real receptor), real effectors $E_{j,h} \in \hat{E}_j$ (h – name of real effector), virtual effectors $e_{j,n} \in \hat{e}_j$ (n – name of virtual effector) and virtual receptors $r_{j,k} \in \hat{r}_j$ (k – name of virtual receptor). Receptors $R_{j,l}$ gather data from the environment and deliver it, in the form aggregated by $r_{j,k}$, to c_j. Based on the received data and the task ${}^{c}\mathcal{T}_{j,c}$ that is to be executed, c_j sends control commands, which are transformed by $e_{j,n}$, into a form accepted by $E_{j,h}$, in order to affect the environment. Each subsystem $s_{j,v}$ contains: internal memory ${}^{s}s_{j,v}$, set of input buffers ${}_{x}\hat{s}_{j,v}$ and set of output buffers ${}_{y}\hat{s}_{j,v}$. Input and output buffers are used to communicate between subsystems through communication channels, i.e. $ss_{j,(v,h)} \in \hat{ss}_j$ (channel between $s_{j,v}$ and $s_{j',h}$). Virtual subsystems communicate only with their associated real subsystems and the control subsystem. Communication between agents occurs only between their control subsystems. Internal memory and buffers are labeled

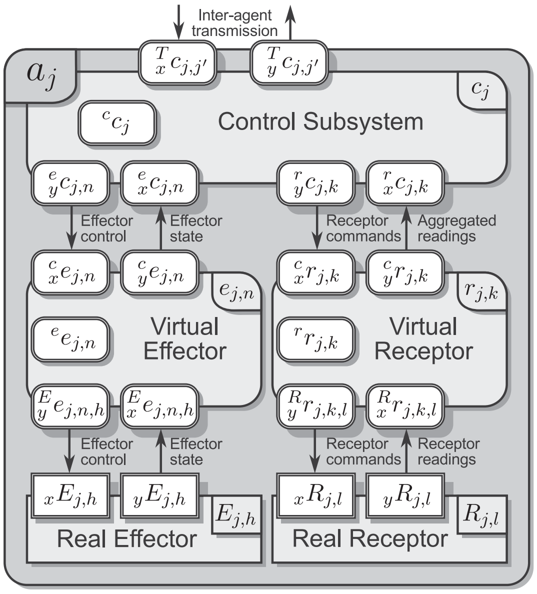


Fig. 2. General embodied agent structure.

systematically in [13]. The symbol consists of a center letter indicating the type of the subsystem s, where $s \in \{c, e, r, E, R\}$; the left subscript denotes the buffer type: 1) x – input, 2) y – output and 3) no subscript – memory; left superscript defines the type of subsystem with which the buffer communicates; right superscript denotes the discrete time stamp i; right subscripts determine the names of: agent and subsystem.

B. Activity

The activity of a robotic system depends on the activities of its agents, while the activities of agents results from the activities of their subsystems. Each subsystem $s_{j,v}$ performs a task ${}^{s}\mathcal{T}_{j,v}$ which involves selecting one of the behaviours ${}^{s}\mathcal{B}_{j,v,\omega}$ to be executed, where ω is behaviour designator. The behaviour is selected based on the satisfied initial condition ${}^{s}f^{\sigma}_{j,v,\alpha} \in {}^{s}\hat{f}^{\sigma}_{j,v}$ (α – predicate designator) [14]. Each behaviour iteratively: 1) calculates the transition function ${}^{s}f^{\sigma}_{j,v,\omega} \in {}^{s}\hat{f}_{j,v}$ (Eq. (1)), takes as arguments the current data from the input buffers ${}_{x}s^{i}_{j,v} \in {}^{s}_{x}\hat{s}_{j,v}$ and internal memory ${}^{s}s^{i}_{j,v}$, calculates the new values and inserts them into the output buffers ${}_{y}s^{i+1}_{j,v} \in {}^{s}_{y}\hat{s}_{j,v}$ and the internal memory ${}^{s}s^{i}_{j,v}$:

$$\left({}^{s}s^{i+1}_{j,v},\, {}_{y}s^{i+1}_{j,v}\right) := {}^{s,s'}f_{j,v,\omega)}\left({}^{s}s^{i}_{j,v}, {}_{x}s^{i}_{j,v}\right) \tag{1}$$

2) transmits ${}_{y}s^{i+1}_{j,v}$ to associated subsystems, 3) increments discrete time counter i, 4) receives data from the associated subsystems into ${}_{x}s^{i}_{j,v}$, 5) checks terminal condition ${}^{s}f^{\tau}_{j,v,\xi} \in {}^{s}\hat{f}^{\tau}_{j,v}$ and error condition ${}^{s}f^{\epsilon}_{j,v,\beta} \in {}^{s}\hat{f}^{\epsilon}_{j,v}$, where ξ and β are designators of those predicates. Behaviour iteration terminates when one of those conditions is fulfilled. In such a case the subsystem $s_{j,v}$ selects the next behaviour based on ${}^{s}\mathcal{T}_{j,v}$. Each ${}^{s}f_{j,v,\omega}$ can be canonically decomposed into partial functions, and each partial function can be decomposed based on data availability into overloaded functions.

For a pair of communicating subsystems, the communication mode must be indicated. There are nine communication

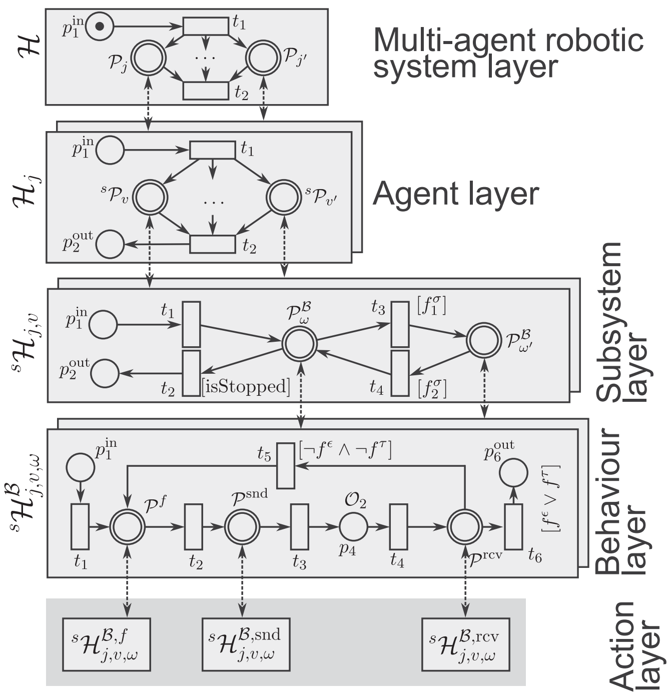

$\mathcal{H}$
Multi-agent robotic system layer
$\mathcal{H}_j$
Agent layer
${}^s\mathcal{H}_{j,v}$
Subsystem layer
${}^s\mathcal{H}^{\mathcal{B}}_{j,v,\omega}$
Behaviour layer
Action layer
p_1^{in}
p_2^{out}
t_1
t_2
$\mathcal{P}_j$
$\mathcal{P}_{j'}$
${}^s\mathcal{P}_v$
${}^s\mathcal{P}_{v'}$
$\mathcal{P}^{\mathcal{B}}_{\omega}$
$\mathcal{P}^{\mathcal{B}}_{\omega'}$
t_3 $[f_1^\sigma]$
t_4 $[f_2^\sigma]$
[isStopped]
t_5 $[\neg f^\epsilon \wedge \neg f^\tau]$
p_6^{out}
$\mathcal{P}^f$
$\mathcal{P}^{\mathrm{snd}}$
$\mathcal{O}_2$
p_4
$\mathcal{P}^{\mathrm{rcv}}$
t_6 $[f^\epsilon \vee f^\tau]$
${}^s\mathcal{H}^{\mathcal{B},f}_{j,v,\omega}$
${}^s\mathcal{H}^{\mathcal{B},\mathrm{snd}}_{j,v,\omega}$
${}^s\mathcal{H}^{\mathcal{B},\mathrm{rcv}}_{j,v,\omega}$

Fig. 3. RSHPN expressing robot system activity.

modes, three each for sending and receiving subsystem. These are: 1) non-blocking, 2) blocking, and 3) blocking mode with timeout. In non-blocking mode the subsystem does not wait for the other subsystem, but immediately resumes its activities, while in blocking mode it waits. A single parameterised Petri net structure covering all three communication modes is defined [13]. They are distinguished by the timeout parameter (i.e. 0 – non-blocking; ∞ – blocking; $\mathrm{value} > 0$ – blocking with timeout).

C. RSHPN Meta-Model

The interaction of agents, subsystem tasks, behaviours, transition functions and communication modes is described by a single RSHPN $\mathcal{H}$ in Fig. 3. Such a net is a bipartite graph, in which nodes, i.e. places p (graphically represented by single circles) and pages $\mathcal{P}$ (double circles), alternate with transitions t (rectangles), all connected by directed arcs (arrows). The pages represent lower level HPNs (drawn as panels connected to their corresponding pages by dashed arrows). Each such network has one input place p^{in} and one output place p^{out}. Logical conditions $\mathcal{C}$ are associated with transitions, while operations $\mathcal{O}$ label places. A correctly constructed $\mathcal{H}$ network is safe, and therefore no more than one token (represented by a black circle) can exist at any of its places. The $\mathcal{H}$ network consists of the following layers (precise description is available in [13], [14]):

1) Multi-Agent Robotic System Layer: Defines a single net $\mathcal{H}$ representing the activities of each agent a_j defined by individual pages $\mathcal{P}_j$ (a page is a lower order Petri Net).

2) Agent Layer: Defines nets $\mathcal{H}_j$ represented by pages $\mathcal{P}_j$. Each net describes the activities of an agent a_j, which consists of several subsystems. The activities of each subsystem $s_{j,v}$ are represented by an individual page ${}^s\mathcal{P}_{j,v}$.

TABLE I
RSHPN META-MODEL PARAMETERS [13]

Layer/Sublayer	Parameters
Multi-agent robotic system	$\hat{a}$
Agent	for each $a_j \in \hat{a}$: $\hat{s}_j$
Subsystem	for each $s_{j,v} \in \hat{s}_j$: ${}^s\hat{\mathcal{B}}_{j,v}$, ${}^s\hat{f}^{\sigma}_{j,v}$, ${}_y\hat{s}_{j,v}$, ${}_x\hat{s}_{j,v}$ and ${}^s\mathcal{T}_{j,v}$
Behaviour	for each ${}^s\mathcal{B}_{j,v,\omega} \in {}^s\hat{\mathcal{B}}_{j,v}$: ${}^sf_{j,v,\gamma}$, ${}^sf^{\tau}_{j,v,\xi}$, ${}^sf^{\epsilon}_{j,v,\beta}$
Canonical decomposition	for each ${}^sf_{j,v,\gamma}$: ${}^s\hat{f}_{j,v,\gamma}$
Data availability	for each ${}^sf_{j,v,\gamma,\psi} \in {}^s\hat{f}_{j,v,\gamma}$: ${}^s\hat{f}_{j,v,\gamma,\psi}$
Send mode	for each ${}^s_ys_{j,v,v'} \in {}_y\hat{s}_{j,v}$ while executing ${}^s\mathcal{B}_{j,v,\omega} \in {}^s\hat{\mathcal{B}}_{j,v}$: $\mathrm{timeout}_1$
Receive mode	for each ${}^s_xs_{j,v,v'} \in {}_x\hat{s}_{j,v}$ while executing ${}^s\mathcal{B}_{j,v,\omega} \in {}^s\hat{\mathcal{B}}_{j,v}$: $\mathrm{timeout}_2$

3) Subsystem Layer: Defines nets ${}^s\mathcal{H}_{j,v}$ represented by ${}^s\mathcal{P}_{j,v,}$. Each net describes the activity of $s_{j,v}$ switching between behaviours ${}^s\mathcal{B}_{j,v,\omega}$ represented by ${}^s\mathcal{P}^{\mathcal{B}}_{j,v,\omega}$.

4) Behaviour Layer: Defines nets ${}^s\mathcal{H}^{\mathcal{B}}_{j,v,\omega}$ represented by pages ${}^s\mathcal{P}^{\mathcal{B}}_{j,v,\omega}$. Each net describes the activities of behaviour ${}^s\mathcal{B}_{j,v,\omega}$ executing operation ${}^s\mathcal{O}^{\mathcal{B}}_{j,v,\omega,1}$ which calculates the transition function ${}^sf_{j,v,\omega}$ by using (1), page ${}^s\mathcal{PB}_{j,v,\omega,\mathrm{snd}}$ (defining the communication mode of ${}^sf_{j,v}$ when sending data from ${}_ys_{j,v}$) and page ${}^s\mathcal{P}^{\mathcal{B}}_{j,v,\omega,\mathrm{rcv}}$ (defining the communication mode of $s_{j,v}$ when receiving data into ${}_xs_{j,v}$),

5) Action Layer: defines 3 Petri nets (further decomposed into 2 layers [13], [14]): 1) ${}^s\mathcal{H}^{\mathcal{B},f}_{j,v,\omega}$ – determining the transition function ${}^sf_{j,v,\omega}$ (1), partial functions ${}^sf_{j,v,\gamma,\psi} \in {}^s\hat{f}_{j,v,\gamma}$ and overloaded functions (from set ${}^s\hat{f}_{j,v,\gamma,\psi}$), 2) ${}^s\mathcal{H}^{\mathcal{B},\mathrm{snd}}_{j,v,\omega}$ and 3) ${}^s\mathcal{H}^{\mathcal{B},\mathrm{rcv}}_{j,v,\omega}$ – determining communication mode used by the sender and receiver, respectively.

The meta-model $\mathcal{H}$ thus defined can be transformed into a model of any robotic system (after specifying Petri nets with variable structures and defining parameters from Table I).

IV. ROBOTIC SYSTEM SPECIFICATION LANGUAGE (RSSL)

A. Meta-Language

The RSSL grammar is expressed using EBNF. The RSSL symbols are consistent with the meta-language: 1) $\langle X \rangle$ – non-terminal symbol X, 2) X – terminal symbol X, 3) $[\,]^*$ – multiplicity from zero to infinity, 4) $[\,]^+$ – multiplicity from one to infinity, 5) $[\,]?$ – either zero or one time, 6) $|$ – alternative. RSSL keywords and parameters are terminal symbols. They are differentiated by using capital letters to express keywords and lowercase letters to write parameters. Non-terminal symbols use capital letters too, however they are enclosed within angle brackets. RSSL blocks are preceded by a keyword and followed by the same keyword prefixed by END_ (e.g. ⟨X⟩ defines a RSSL block starting with the terminal symbol X and finishing with END_X).

B. RSSL syntax

The simplified grammar of RSSL is presented in Fig. 4–7. The non-terminal symbols of the grammar represent the domain concepts which are expressed in the form of language blocks, e.g.: ⟨ROBOTIC_SYSTEM⟩; ⟨AGENT⟩; ⟨CS⟩, ⟨VE⟩ and ⟨VR⟩ using ⟨SUBSYSTEM⟩ pattern;

```
ROBOTIC_SYSTEM rs_id
  [ ⟨AGENT⟩ ]+
  [ ⟨INTER_AGENT_CHANNEL⟩]*
END_ROBOTIC_SYSTEM
```

```
AGENT agent_id
  ⟨CS⟩  [ ⟨VE⟩ ]*  [ ⟨VR⟩]*
  [ ⟨INTRA_AGENT_CHANNEL⟩]*
  [ ⟨INCLUDE⟩]?  [ ⟨DATA_TYPE⟩]*
END_AGENT
```

Fig. 4. RSSL syntax of: (left) $\langle$ROBOTIC_SYSTEM$\rangle$ composed of agents and inter-agent channels, (right) $\langle$AGENT$\rangle$ defining inner structure of an agent a_j, where $j \equiv$ agent_id.

```
SUBSYSTEM s_id freq_val
  ⟨SUBSYSTEM_TASK⟩
  [ ⟨BEHAVIOUR⟩]+
  [ ⟨INITIAL_CONDITION⟩]*
  [ ⟨TERMINAL_CONDITION⟩]+
  [ ⟨ERROR_CONDITION⟩]+
  [ ⟨TRANSITION_FUNCTION⟩]+
  [ ⟨MEMORY_BUFFER⟩]*
  [ ⟨INPUT_BUFFER⟩]*
  [ ⟨OUTPUT_BUFFER⟩]*
  [ ⟨INCLUDE⟩]?
END_SUBSYSTEM
```

```
SUBSYSTEM_TASK init_beh_id
  NODES
    [node_id beh_id;]+
  END_NODES
  EDGES
    [edge_id init_cond_id;]*
  END_EDGES
  CONNECTIONS
    [node_id_1 edge_id
    node_id_2;]*
  END_CONNECTIONS
END_SUBSYSTEM_TASK
```

Fig. 5. RSSL syntax of: (left) $\langle$SUBSYSTEM$\rangle$ defining elements necessary to specify $s_{j,v}$ activities, where $v \equiv$ s_id, (right) $\langle$SUBSYSTEM_TASK$\rangle$ defining ${}^{s}\mathcal{T}_{j,v}$ in the form of a graph specifying the order of execution of $s_{j,v}$ behaviours.

```
BEHAVIOUR
[beh_id tran_fun_id
term_cond_id
error_cond_id;
]+
END_BEHAVIOUR
```

```
CONDITION
[cond_id ⟨CODE⟩;]*
END_CONDITION

BUFFER
[data_type buf_id;]*
END_BUFFER
```

Fig. 6. RSSL syntax of: (left) $\langle$BEHAVIOUR$\rangle$ block for ${}^{s}\mathcal{B}_{j,v,\omega}$, where $\omega \equiv$ beh_id, $\gamma \equiv$ tran_fun_id, $\xi \equiv$ term_cond_id and $\beta \equiv$ error_cond_id, (right-top) $\langle$CONDITION$\rangle$ a pattern used for defining conditions: (a) ${}^{s}f^{\sigma}_{j,v,\alpha}$ – initial condition ($\alpha \equiv$ cond_id), (b) ${}^{s}f^{\tau}_{j,v,\xi}$ – terminal condition ($\xi \equiv$ cond_id), (c) ${}^{s}f^{\epsilon}_{j,v,\beta}$ – error condition ($\beta \equiv$ cond_id), and (right-bottom) $\langle$BUFFER$\rangle$ which is a pattern utilised for defining buffers: ${}_{x}s_{j,v}$, ${}_{y}s_{j,v}$ and ${}^{s}s_{j,v}$.

```
TRANSITION_FUNCTION
  tran_fun_id
  [ ⟨PARTIAL_FUNCTION⟩]+
END_TRANSITION_FUNCTION
```

```
PARTIAL_FUNCTION
  [out_buf_id[,out_buf_id]*=]?
  part_fun_id([in_buf_id
  [,in_buf_id]*]?)⟨CODE⟩
END_PARTIAL_FUNCTION
```

```
INTER_AGENT_CHANNEL
  [snd_agent_id::snd_buf_id[MODE]->channel_id->
  rcv_agent_id::rcv_buf_id[MODE];]*
END_INTER_AGENT_CHANNEL
```

Fig. 7. RSSL syntax of: (left) $\langle$TRANSITION_FUNCTION$\rangle$ block for ${}^{s}f_{j,v,\gamma}$, where $\gamma \equiv$ tran_fun_id, (right) $\langle$PARTIAL_FUNCTION$\rangle$ block for its partial function ${}^{s}f_{j,v,\gamma,\psi}$, where $\psi \equiv$ part_fun_id, and (bottom) $\langle$INTER_AGENT_CHANNEL$\rangle$ block for $aa_{j,j'} \in \hat{aa}$, where $j \equiv$ snd_agent_id, $j' \equiv$ rcv_agent_id, ${}^{T}_{y}c_{j,j'} \equiv$ snd_buf_id, and ${}^{T}_{x}c_{j',j} \equiv$ rcv_buf_id.

$\langle$SUBSYSTEM_TASK$\rangle$; $\langle$BEHAVIOUR$\rangle$; $\langle$CONDITION$\rangle$ and $\langle$TRANSITION_FUNCTION$\rangle$. Those blocks, presented in the form of syntax diagrams, define the RSSL language. Order of declaring the blocks associated with those non-terminal symbols is not enforced by RSSL. Non-terminal symbols defining partial functions and conditions contain code blocks $\langle$CODE$\rangle$ with code directly written in the target language (C++ or Python).

RSSL blocks:

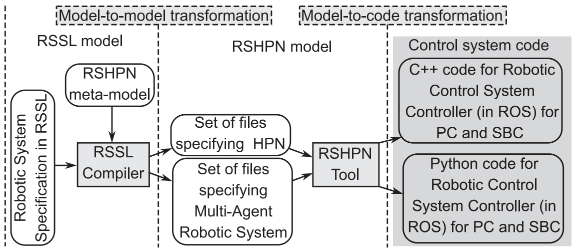

Model-to-model transformation
Model-to-code transformation
RSSL model
RSHPN model
Control system code
Robotic System Specification in RSSL
RSHPN meta-model
RSSL Compiler
Set of files specifying HPN
Set of files specifying Multi-Agent Robotic System
RSHPN Tool
C++ code for Robotic Control System Controller (in ROS) for PC and SBC
Python code for Robotic Control System Controller (in ROS) for PC and SBC

Fig. 8. Two-phase transformation of RSSL specification into target code based on [13], [14].

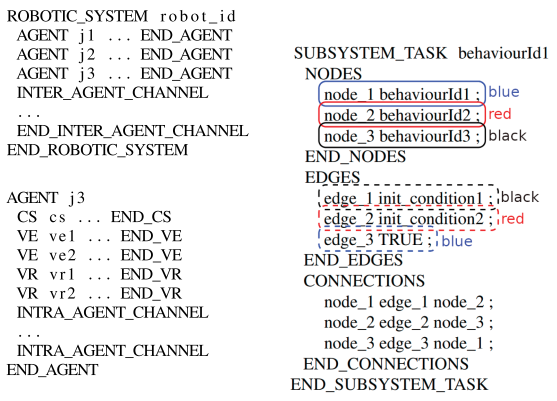
1) $\langle$ROBOTIC_SYSTEM$\rangle$: (Left side of Fig. 4) – defines a robotic system $\mathcal{RS}$ with a non-empty set of agents (i.e. $\hat{a}$) and a set of inter-agent channels (i.e. $\hat{aa}$).

2) $\langle$AGENT$\rangle$: (Right hand side of Fig. 4) – defines agent a_j where $j \equiv$ agent_id. It specifies the agent's name j, a single control subsystem (c_j), a set of virtual effectors (i.e. $\hat{e}j$), a set of virtual receptors (i.e. $\hat{r}j$), a set of intra-agent communication channels $\hat{aa}_j$, i.e. channels utilised for communication between subsystems of a_j, and if necessary the auxiliary files and data types.

3) $\langle$CS$\rangle$, $\langle$VE$\rangle$, $\langle$VR$\rangle$: Use the same $\langle$SUBSYSTEM$\rangle$ pattern (left side of Fig. 5) to model the subsystem activity and structure. It specifies: 1) v – the subsystem name, 2) frequency at which behaviours of the subsystem iterate, 3) ${}^{s}\mathcal{T}_{j,v}$ – subsystem task (where j indicates the agent name), 4) ${}^{s}\hat{\mathcal{B}}_{j,v}$ – a non-empty set of behaviours, 5) ${}^{s}\hat{f}^{\sigma}_{j,v}$ – a set of initial conditions, 6) ${}^{s}\hat{f}^{\tau}_{j,v}$ – a non-empty set of terminal conditions, 7) ${}^{s}\hat{f}^{\epsilon}_{j,v}$ – a non-empty set of error conditions, 8) ${}^{s}\hat{f}_{j,v}$ – a non-empty set of transition functions, 9) ${}^{s}s_{j,v}$ – an internal memory buffer, 10) ${}_{x}\hat{s}_{j,v}$, ${}_{y}\hat{s}_{j,v}$ – sets of input and output buffers. All condition types are defined using $\langle$CONDITION$\rangle$ pattern (right-top of Fig. 6), while buffer types using $\langle$BUFFER$\rangle$ pattern (right-bottom of Fig. 6).

4) $\langle$SUBSYSTEM_TASK$\rangle$: (Right hand side of Fig. 5) – defines the task ${}^{s}\mathcal{T}_{j,v}$ executed by a subsystem $s_{j,v}$. The $\langle$SUBSYSTEM_TASK$\rangle$ block defines three different sets: 1) a set of nodes defined within $\langle$NODES$\rangle$ block, 2) a set of edges defined within $\langle$EDGES$\rangle$ block, and 3) a set of connections defined within $\langle$CONNECTIONS$\rangle$ block. The ${}^{s}\mathcal{T}_{j,v}$ in RSSL is a graph (with specified initial node) connecting alternately elements from the first two sets, i.e.: set of nodes defined in $\langle$NODES$\rangle$ block and a set of edges defined in $\langle$EDGES$\rangle$ block. The elements from different sets are connected with each other by a connection defined in the $\langle$CONNECTION$\rangle$ block (as presented in exemplary listing on right hand side of Fig. 9). Each node is associated with a single behaviour, while each edge with a single initial condition. A connection defines a switch between two behaviours when the condition associated with the edge is satisfied. As a result, ${}^{s}\mathcal{T}_{j,v}$ defines the subsystem $s_{j,v}$ activities by specifying how it switches between behaviours on the basis of initial conditions.

5) $\langle$BEHAVIOUR$\rangle$: (Left side of Fig. 6) – defines a single ${}^{s}\mathcal{B}_{j,v,\omega}$ or a set of behaviours ${}^{s}\hat{\mathcal{B}}_{j,v,\omega}$ (separated by semicolons) for $s_{j,v}$. For each behaviour it specifies its name ω and three components defined within $s_{j,v}$, i.e. 1) transition function ${}^{s}f_{j,v,\gamma} \in {}^{s}\hat{f}_{j,v}$, 2) terminal condition ${}^{s}f^{\tau}_{j,v,\xi} \in {}^{s}\hat{f}^{\tau}_{j,v}$ and 3) error condition ${}^{s}f^{\epsilon}_{j,v,\beta} \in {}^{s}\hat{f}^{\epsilon}_{j,v,\beta}$.



```
ROBOTIC_SYSTEM robot_id
  AGENT j1 ... END_AGENT
  AGENT j2 ... END_AGENT
  AGENT j3 ... END_AGENT
  INTER_AGENT_CHANNEL
  ...
  END_INTER_AGENT_CHANNEL
END_ROBOTIC_SYSTEM

AGENT j3
  CS cs ... END_CS
  VE ve1 ... END_VE
  VE ve2 ... END_VE
  VR vr1 ... END_VR
  VR vr2 ... END_VR
  INTRA_AGENT_CHANNEL
  ...
  INTRA_AGENT_CHANNEL
END_AGENT
```

```
SUBSYSTEM_TASK behaviourId1
  NODES
    node_1 behaviourId1 ;
    node_2 behaviourId2 ;
    node_3 behaviourId3 ;
  END_NODES
  EDGES
    edge_1 init_condition1 ;
    edge_2 init_condition2 ;
    edge_3 TRUE ;
  END_EDGES
  CONNECTIONS
    node_1 edge_1 node_2 ;
    node_2 edge_2 node_3 ;
    node_3 edge_3 node_1 ;
  END_CONNECTIONS
END_SUBSYSTEM_TASK
```

Fig. 9. RSSL specifications: (top left) robotic system $\mathcal{RS}$ defining a set of agents, i.e. $\hat{a} = \{a_{\mathrm{j1}}, a_{\mathrm{j2}}, a_{\mathrm{j3}}\}$; (bottom left) agent a_{j3} composed of subsystems $\hat{s}_{\mathrm{j3}} = \{c_{\mathrm{j3}}, e_{\mathrm{j3,ve1}}, e_{\mathrm{j3,ve2}}, e_{j3,vr1}\}$; (right) task ${}^{c}\mathcal{T}_{\mathrm{j3,cs}}$ consisting of three triples: ${}^{c}\mathcal{T}_{\mathrm{j3,cs}} = \{({}^{c}\mathcal{B}_{\mathrm{j3,cs,behaviourId1}}, {}^{c}f^{\sigma}_{\mathrm{j3,cs,init\_condition1}}, {}^{s}\mathcal{B}_{\mathrm{j3,cs,behaviourId2}}), ({}^{c}\mathcal{B}_{\mathrm{j3,cs,behaviourId2}}, {}^{c}f^{\sigma}_{\mathrm{j3,cs,init\_condition2}}, {}^{c}\mathcal{B}_{\mathrm{j3,cs,behaviourId3}}), ({}^{c}\mathcal{B}_{\mathrm{j3,cs,behaviourId3}}, {}^{c}f^{\sigma}_{\mathrm{j3,cs,TRUE}^{c}}, {}^{c}\mathcal{B}_{\mathrm{j3,cs,behaviourId1}^{c}})\}$.

6) $\langle$TRANSITION_FUNCTION$\rangle$*:* (Left of Fig. 7) – defines a transition function ${}^{s}f_{j,v,\gamma}$, represented by (1), specified for $s_{j,v}$, which is utilised as a parameter describing a behaviour ${}^{s}\mathcal{B}_{j,v,\omega} \in {}^{s}\hat{\mathcal{B}}_{j,v,\omega}$. The transition function ${}^{s}f_{j,v,\gamma}$ is composed of a set of partial functions (defined within $\langle$PARTIAL_FUNCTION$\rangle$). While $\langle$PARTIAL_FUNCTION$\rangle$ (right of Fig. 7) – defines a partial function ${}^{s,s'}f_{j,v,\gamma,\psi}$, i.e. specifies: 1) the output buffer (based on which the canonical decomposition of transition function is done) to which the calculated data is inserted, 2) the partial function name, and 3) a sequence of input buffers (separated by a comma) being the arguments of the partial function. Decomposition of the partial function into overloaded functions is done by determining multiple $\langle$PARTIAL_FUNCTION$\rangle$ blocks defining overloaded functions with the same name, but with a different set of input buffers and different implementations. An overloaded function is needed to compute a new value for the output buffers based on the currently available data from a subset of the input buffers. Each partial function by default has access to the internal memory buffer ${}^{s}s_{j,v}$, thus it is not necessary to specify it within the function definition.

7) $\langle$INTER_AGENT_CHANNEL$\rangle$ *Block:* (Bottom of Fig. 7) – defines an inter-agent communication channel $aa_{(j,j')} \in \hat{aa}$ specifying communication between two agents, e.g. a_j and $a_{j'}$. For each agent a transmission buffer and a communication mode is specified. Currently in RSSL three possible communication modes are distinguished: $\mathrm{MODE} \in \{\mathrm{BLOCK}, \mathrm{NON\_BLOCK}, \mathrm{BLOCK\_TIMEOUT}\}$. While block $\langle$INTRA_AGENT_CHANNEL$\rangle$ defines communicating channel $ss_{j,(v,h)} \in \hat{ss}_j$ between two subsystems, e.g. $s_{j,v}$ and $s_{j,h}$, residing in the same agent a_j.

V. Developed Tools

Two different tools to support the design of robotic systems were developed: 1) the RSSL compiler – verifies the correctness of the provided specification and transforms the RSSL specification into RSHPN specification, and 2) RSHPN Tool – used to load the RSHPN specification, extend it, and to generate controller code (both in C++ and Python). As the RSHPN Tool has already been presented in detail in [13], [14] and in the video [17], only the RSSL compiler is discussed below. The task of the RSSL compiler can be divided into two stages: 1) analysis of the supplied specification, and 2) generation of the RSHPN specification.

A. Analysis of the Specification

The aim of the first stage is to detect errors by performing: lexical, syntactic and semantic analysis. Since the context-free grammar of RSSL provides freedom to the designer to define blocks in a fairly arbitrary configuration, there is a need to equip the RSSL compiler with contextual correctness analysis assuring that: 1) there are no name collisions, 2) the referred concepts are already specified, 3) communication channels are correctly specified, 4) the multiplicity of the concepts used is correct etc.

B. RSHPN Generation

In the second stage the RSSL compiler transforms the RSSL specification into the RSHPN specification. Thus the parameters specified in Table I are retrieved from the RSSL specification and then two sets of XML files are generate, based on them and the RSHPN meta-model. One type of file expresses the structure of the Petri net, i.e. appropriately parameterised places, transitions, and edges (indicating the activity of the robotic system), and the other specifies the structure of the designed robotic system.

VI. Data Processing – Translation

The general scheme of generation of a RSHPN model and the translation of the model into target code is shown in Fig. 8. The data processing pipeline consists of two-phases: 1) RSHPN model generation based on a set of parameters provided by the RSSL specification and based on a general RSHPN meta-model, and 2) model-to-code transformation executed by RSHPN Tool [13]. Together they produce the target code, i.e.: 1) C++ ROS code [14], or 2) Python ROS code [13] – of a robotic system controller. The idea of extraction of parameters from the RSSL specification follows directly from Section IV-A. Fig. 9 illustrates how the parameters are extracted from the RSSL specification, while Fig. 10 shows how the corresponding RSHPN layers are generated from them. The detailed transformation between RSHPN model and controller code was presented in [13], [14].

VII. Experiments

In order to verify the suitability of RSSL for the specification of robotic systems, several systems of different complexity were created, viz. 1) a real table-tennis-ball collecting robot with a variable controller structure [13] (Fig. 11(a), 2) simulation of a LWR4+ manipulator tracing circles [14] (Fig. 11(b), [17]), 3) simulation of the Velma robot transferring balls [13] (Fig. 11(c) and video[1]), and 4) a simulated and real Velma robot visually tracking an object (Fig. 11(d) and video: simulated[2] and real[3]). The two first systems were created from scratch, while for

[1]Ball transfer: www.youtube.com/watch?v=p5yqGTdLRzo
[2]Simulated visual tracking: www.youtube.com/watch?v=vL6b6UMRb3E
[3]Real visual tracking: www.youtube.com/watch?v=S4-Rr27fZzQ

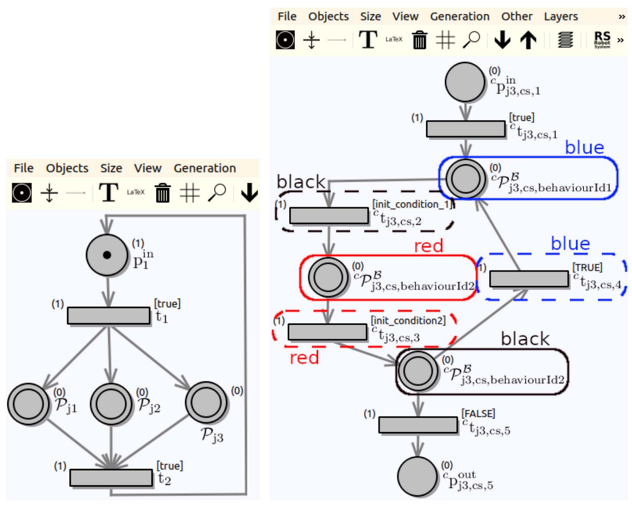

File Objects Size View Generation
File Objects Size View Generation Other Layers
black
red
blue
blue
red
black
[init_condition_1]
[init_condition2]
[true]
[TRUE]
[FALSE]

Fig. 10. Automatically generated RSHPN: (left) $\mathcal{H}$ based on specification (top-left Fig. 9), (right) ${}^{c}\mathcal{H}_{\mathrm{j3,cs}}$ (as ${}^{c}\mathcal{P}_{\mathrm{j3,cs}}$ from $\mathcal{H}_{\mathrm{j3}}$) based on specification (right Fig. 9).

```
ROBOTIC_SYSTEM
 AGENT audio ...; super ...; tele ... END_AGENT
 AGENT auto ...; img_proc ... END_AGENT
 AGENT bc  CS cs  ... END_CS
  VR camera ...; sonar ...; inlet ...; encoder ...END_VR
  VE motor  ...; vacuum  ... END_VE
  INTRA_COMMUNICATION
    cs[NB]->csToMotor->motor[NB];
    cs[NB]->csToVacuum->vacuum[NB];
    camera[NB]->cameraToCs->cs[NB];
    sonar[NB]->sonarToCs->cs[NB];
    inlet[NB]->inletToCs->cs[NB];
    encoder[NB]->encoderToCs->cs[NB];
  END_INTRA_COMMUNICATION
 END_AGENT
 INTER_COMMUNICATION
   audio[NB]->audioToSuper->super[NB];
   super[NB]->superToAudio->audio[NB];
   super[NB]->superToAuto->auto[NB];
   super[NB]->superToTele->tele[NB];
   auto[NB]->autoToSuper->super[NB];
   tele[NB]->teleToSuper->super[NB];
   auto[NB]->autoToBall->bc[NB];
   tele[NB]->teleToBall->bc[NB];
   bc[NB]->ballToAuto->auto[NB];
   bc[NB]->ballToTele->tele[NB];
 END_INTER_COMMUNICATION
END_ROBOTIC_SYSTEM
```

Fig. 12. Excerpt from the RSSL specification of the ball collector (Fig. 11(a)), where NB $\equiv$ non-blocking mode, ... indicate further hierarchical definition of the concept.

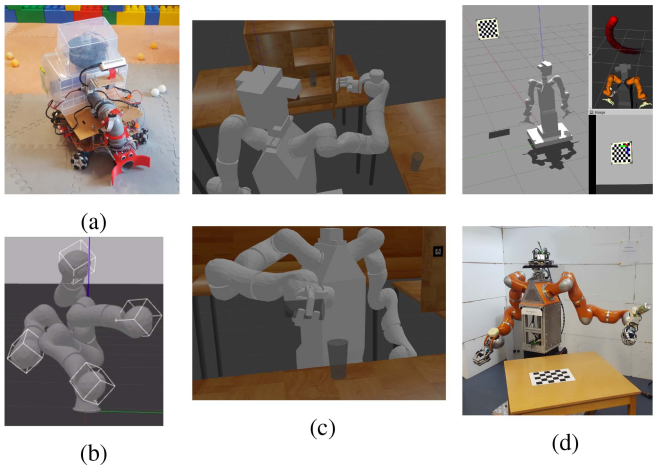


Fig. 11. (a) Ball collector, (b) LWR4+ manipulator, (c) Velma robot transferring ball, (d) visual object tracking.

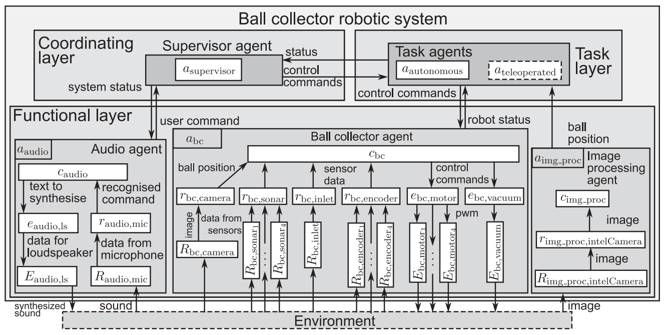


Fig. 13. Structure of the ball collecting robotic system generated based on the RSSL specification from Fig. 12.

two others a higher control layer was specified to control already existing Velma robot controller [18], both real and simulated.

The controller for each of the above experiments was expressed as an RSSL specification and then converted into the RSHPN model. Then, using the RSHPN Tool (as shown in video [17] or in [13], [14]), the resulting RSHPN model was automatically transformed into the controller source code (either Python or C++). The number of parameters that need to be specified in order for the meta-model to be transformed into a model, and space limitation, do not allow these specifications to be presented here. Nevertheless, Fig. 12 shows an excerpt from the RSSL specification defining the structure (Fig. 13) of the table-tennis-ball collecting robotic system. The platform was equipped with omnidirectional wheels, electric motors, two cameras, four sonars, a microphone and a controlled ball-sucking device. The robot performed the task autonomously or teleoperated (using voice commands). The result was a system consisting of six agents (a detailed description of the system activities is presented in [13]).

VIII. Discussion

A. Complexity Comparison Between RSSL and RSHPN

Two equations were proposed to compare the complexity of the model expressed using RSHPN (2) to that expressed with RSSL (3). $\mathcal{O}^{\mathrm{model}}$ is examined by calculating a sum of: 1) all the necessary elements (places/ pages, transitions, and edges), and 2) the number of their parameters, needed to define RSHPN $\mathcal{H}$. $\mathcal{O}^{\mathrm{model}}$ examines the total number of terminal symbols used to specify the robotic system using RSSL.

$$\mathcal{O}^{\mathrm{model}}(\mathrm{RSHPN}) = \sum_{a_j \in \hat{a}} \Bigg[\sum_{s_{j,v} \in \hat{s}_j} \big[|{}^{s}\hat{\mathcal{B}}_{j,v}| \cdot \big(112 \cdot |{}_{y}\hat{s}_{j,v} + 81 \cdot |{}_{x}\hat{s}_{j,v}| + 2^{|{}_{x}\hat{s}_{j,v}|} \cdot (21 \cdot |{}_{y}\hat{s}_{j,v}| + 23) + 148\big) + 10 \cdot |{}^{s}\hat{f}^{\sigma}_{j,v}\big] + 35 \cdot |\hat{s}_j \Bigg] + 29 \cdot |\hat{a}| + 14 \quad (2)$$

$$\begin{aligned}\mathcal{O}^{\text{model}}(\text{RSSL}) = &\sum_{a_j \in \hat{a}} \Bigg[\sum_{s_{j,v} \in \hat{s}_j} \big[10 \cdot |{}^{s}\hat{\mathcal{B}}_{j,v}| + 13 \cdot |{}^{s}\hat{f}^{\sigma}_{j,v}| \\ &+ |{}^{s}\hat{f}^{\tau}_{j,v}| \cdot \Big[(|{}_{y}\hat{s}_{j,v}| + 1) \cdot \Big(8 \cdot 2^{|{}_{x}\hat{s}_{j,v}|} + |{}_{x}\hat{s}_{j,v}| \cdot 2^{|{}_{x}\hat{s}_{j,v}|} + 1 \Big) \Big] \\ &+ 6 \cdot (|{}^{s}\hat{f}^{\tau}_{j,v}| + |{}^{s}\hat{f}^{\epsilon}_{j,v}|) + 5 \cdot (|{}_{x}\hat{s}_{j,v}| + |{}_{y}\hat{s}_{j,v}|) + 3 \cdot |{}^{s}\hat{f}_{j,v}| \big] \\ &+ 18 \cdot |\hat{s}_j| + 18 \cdot |\hat{ss}_j| \Bigg] + 7 \cdot |\hat{a}| + 18 \cdot |\hat{aa}| + 3 \qquad (3)\end{aligned}$$

Although in each case different concepts are used the workload of specifying them is similar. For a fairly simple system, consisting of: 5 agents, each having exactly 5 subsystems, each containing exactly 5 behaviours, 5 input and 5 output buffers, the resulting RSHPN contains a total of 173409 elements, while RSSL specification contains 27188 terminal symbols. It is nearly 6 times less compared to the RSHPN size. This result has been obtained assuming: 1) $|{}^{s}\hat{f}^{\sigma}_{j,v}| = |{}^{s}\mathcal{B}_{j,v}|$, and 2) each partial function is decomposed into two overloaded functions. Nevertheless, in addition to the symbols necessary to define the above parameters, the RSSL language syntax includes auxiliary terminal symbols increasing the readability of the specification, i.e.: commas, parentheses and semicolons. Ultimately, for $|{}_{x}\hat{s}_{j,v} \geq 14$ the RSSL becomes more complex than RSHPN representation of the system. This is due to the proliferating number of auxiliary symbols $2^{|{}_{x}\hat{s}_{j,v}|} \cdot (|{}_{x}\hat{s}_{j,v}| + 1)$ when $2^{|{}_{x}\hat{s}_{j,v}|}$ combinations of overloaded functions are considered. It is possible to reduce the complexity of the RSSL language syntax, e.g. by removing auxiliary symbols, but at the cost of reduced readability of the specification.

B. Comparison Between RSSL Specification and Automatically Generated Controller Code

An RSSL specification consists of intrinsic RSSL code and code blocks containing Python or C++ code. The latter contain invocations of library code or legacy software. The capability of combining the two types of code significantly simplifies the enhancement of preexisting systems, e.g. Velma. However, even in the case of designing a system from scratch the definition of transition functions and the diverse conditions is usually done using the underlying implementation language or an existing framework, e.g. ROS. The use of RSSL does not force the developer to posses excessive ROS or programming skills. The user-defined code can be just a simple list of consecutively executed instructions. Due to this, the developer can concentrate mainly on determining the structure and the activities of the robotic system, treating the programming aspects as of secondary importance. Table II shows the significant benefits of RSSL language. Several conclusions can be drawn from it. The more complex the structure of the system under development, the proportionally more controller code is generated (excluding user-defined code). This is particularly evident in the first (17 times benefit) and last experiment (8.63 times benefit). In the case of the third experiment for which only a single subsystem was generated, the benefit is the smallest, i.e. 6.85 times.

The more complex the robotic system, the more code needs to be defined within the code blocks in RSSL specification. In the first and third experiment the relation between user-defined code (Python/C++) and intrinsic RSSL code is 9.86 and 4.98

TABLE II
COMPARISON OF GENERATED CONTROLLERS: (NO 1) BALL COLLECTOR, (NO 2) LWR4+ MANIPULATOR, (NO 3) VELMA TRANSFERRING A BALL, (NO 4) VELMA VISUALLY TRACKING AN OBJECT

		Experiment			
	Description	No 1	No 2	No 3	No 4
A	RSSL spec.	104.3kB	4.5kB	31.1kB	10.7kB
B	RSSL spec. (intrinsic code)	9.6kB	2.7kB	5.2kB	4.1kB
C	RSSL spec. (user-def. code)	94.7kB	1.8kB	25.9kB	6.6kB
D	generated files	75	11	11	16
E	gen. code (including user-def. code)	257.8kB	21.8kB	61.5kB	42kB
F	gen. code (excluding user-def. code)	163.1kB	20.0kB	35.6kB	35.4kB
E/A	ratio: gen. controller code / RSSL spec.	2.47	4.84	1.98	3.93
C/B	ratio: RSSL spec. (user-def. code) / RSSL spec. (intrinsic code)	9.86	0.66	4.98	1.61
F/B	ratio: gen. code (excluding user-def. code) / RSSL spec. (intrinsic code)	16.99	7.41	6.85	8.63

respectively. This is because both experiments used additional external systems, tools and libraries. Therefore, it was necessary to define additional code in INCLUDE blocks. Furthermore, for both experiments it was necessary to define many overloaded functions which significantly increased size of the user-defined code in RSSL specification.

Furthermore, as can be seen from the second experiment, it is possible to develop a library of overloaded functions. As a result, the designer's task would come down only to selecting the appropriate transition function from the library, and as a result, the code necessary to define an overloaded function would be reduced to a single function call. The advantage of this approach would be the reusability of existing code, while minimising the work done by designer.

C. Applicability of RSSL to Complex Robotic Systems

RSSL is a tool resulting from a long evolution of robotic system design methods elaborated at our laboratory [16]. This evolution not only improved and extended the concepts underlying RSSL but tested their utility on fairly complex systems, e.g.: a controller of an industrial robot having a serial-parallel manipulator structure [19], a two-manipulator robotic system solving the Rubic's cube puzzle [20], robots utilising position-force control [21], visual servoing [22], [23], stigmergic communication of independently acting robots [24], multi-robot system acting as a fixture in aircraft part machining [25] and robot companion having variable controller structure [26]. The concepts underlying RSSL, which were tested on those systems, are fairly independent of each other, thus they support decomposition facilitating robot system design. RSSL by using those concepts provides the metamodel being the scaffolding for any robotic system. Metamodel parameters, when appropriately defined, transform this metamodel into a model of a particular system. Definition of each of those parameters is independent of the definitions of others. Thus an increase of complexity of the designed system neither introduces extra complexity into the design process nor increases the complexity of the task. It simply enlarges the number of the parameters that have to be defined. Hence, RSSL scales well for large systems.

IX. CONCLUSION

RSSL facilitates the specification and implementation of robotic systems at different levels of abstraction, starting at the robotic system level and ending with the inter subsystem communication modes. The RSSL specification of a robotic system is decomposed naturally into structure and activities of each of its subsystems. The thus produced specification is subsequently translated into the code of a robotic system controller ready to be compiled by a general purpose language compiler. The RSSL specification defines all parameters necessary to obtain RSHPN model from parameterised RSHPN meta-model. It can be seen from the experiments that the introduction of the parameterised RSHPN meta-model has made the process of creating robotic system controllers easier and more efficient. The designer's task is only to define the relevant parameters defining the structure and activity of the system using only concepts from the domain. Obviously this still requires some effort from the designer, but as it was shown, a significant benefit of the proposed approach is apparent. Other elements of the system, e.g. those related to the RSHPN-based approach, are automatically generated so that the designer does not have to focus on them at all.

Using our design methodology, it is possible to develop a robotic system controller, either as a whole or limited to just a fragment of the controller communicating with an already existing fragment of the system. There is no need to design an entire robotic system from scratch each time – it suffices to extend the existing controller with higher layers, so it requires the development of a separate controller that adequately interfaces with the existing controller. Given the multitude of existing ROS-based robotic systems, the proposed approach based on RSSL is an attractive option when designing robotic systems based on ROS.

References

[1] "Robotics 2020 multi-annual roadmap for robotics in Europe - horizon 2020 call ICT-2017, SPARC the partnership for robotics in Europe: European commision and euRobotics AISBL," 2016. [Online]. Available: https://www.eu-robotics.net/cms/upload/topic_groups/H2020_Robotics_Multi-Annual_Roadmap_ICT-2017B.pdf

[2] E. de Araújo Silva, E. Valentin, J. R. H. Carvalho, and R. da Silva Barreto, "A survey of model driven engineering in robotics," *J. Comput. Lang.*, vol. 62, 2021, Art. no. 101021.

[3] G. Booch, I. Jacobson, and J. Rumbaugh, "Unified modeling language specification," Object Management Group, Tech. Rep. formal/2017-12-05, 2000.

[4] S. Dhouib et al., "RobotML, a domain-specific language to design, simulate and deploy robotic applications," in *Proc. Int. Conf. Simul., Model., Program. Auton. Robots*, Berlin, Germany: Springer, 2012, pp. 149–160.

[5] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*. Amsterdam, The Netherlands: Elsevier, 2015.

[6] D. Stampfer et al., "The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software," *J. Softw. Eng. Robot.*, vol. 7, pp. 3–19, 2016.

[7] A. Nordmann et al., "A survey on domain-specific modeling and languages in robotics," *J. Softw. Eng. Robot.*, vol. 7, pp. 75–99, 2016.

[8] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Comput. Lang., Syst. Struct.*, vol. 43, pp. 139–155, 2015.

[9] B. Mazzolai and C. Laschi, "A vision for future bioinspired and biohybrid robots," *Sci. Robot.*, vol. 5, no. 38, 2020, Art. no. eaba6893.

[10] C. Lesire, D. Doose, and C. Grand, "Formalization of robot skills with descriptive and operational models," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2020, pp. 7227–7232.

[11] D. Kortenkamp, R. Simmons, and D. Brugali, "Robotic systems architectures and programming," in *Springer Handbook of Robotics*, 2nd ed. Berlin, Germany: Springer, 2016, pp. 283–306.

[12] R. A. Brooks, "Intelligence without reason," in *Artificial Intelligence: Critical Concepts*, vol. 3. London, U.K.: Taylor & Francis, 1991, pp. 107–163.

[13] M. Figat and C. Zieliński, "Parameterised robotic system meta-model expressed by Hierarchical Petri nets," *Robot. Auton. Syst.*, vol. 150, 2022, Art. no. 103987.

[14] M. Figat and C. Zieliński, "Robotic system specification methodology based on hierarchical petri nets," *IEEE Access*, vol. 8, pp. 71617–71627, 2020.

[15] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing, 2006.

[16] C. Zieliński, "Robotic System Design Methodology Utilising Embodied Agents," in *Automatic Control, Robotics and Information Processing* (Studies in Systems, Decision and Control Series), Vol. 296, P. Kulczycki, J. Korbicz, and J. Kacprzyk, Eds. Berlin, Germany: Springer, 2021, pp. 523–561, doi: 10.1007/978-3-030-48587-0_17.

[17] M. Figat, "Robotic system HPN tool (RSHPN Tool)," [Online]. Available: https://github.com/mfigat/public_rshpn_tool/blob/master/rshpn_tool.mp4?raw=true

[18] T. Winiarski et al., "EARL–embodied agent based robot control systems modelling language," *Electron.*, vol. 9, 2020, Art. no. 379.

[19] C. Zieliński et al., "A prototype robot for polishing and milling large objects," *Ind. Robot*, vol. 30, no. 1, pp. 67–76, 2003.

[20] C. Zieliński et al., "Rubik's cube as a benchmark validating MRROC as an implementation tool for service robot control systems," *Ind. Robot: Int. J.*, vol. 34, no. 5, pp. 368–375, 2007.

[21] C. Zieliński and T. Winiarski, "Motion generation in the MRROC robot programming framework," *Int. J. Robot. Res.*, vol. 29, no. 4, pp. 386–413, 2010.

[22] M. Staniak and C. Zieliński, "Structures of visual servos," *Robot. Auton. Syst.*, vol. 58, no. 8, pp. 940–954, 2010.

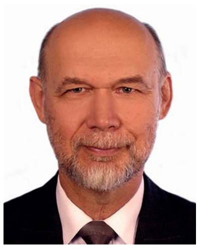
[23] T. Kornuta and C. Zieliński, "Robot control system design exemplified by multi-camera visual servoing," *J. Intell. Robot. Syst.*, vol. 77, no. 3/4, pp. 499–524, 2013, doi: 10.1007/s10846-013-9883-x.

[24] C. Zieliński and P. Trojanek, "Stigmergic cooperation of autonomous robots," *J. Mechanism Mach. Theory*, vol. 44, pp. 51–64, 2009.

[25] T. Zielińska et al., "Path planning for robotized mobile supports," *J. Mechanism Mach. Theory*, vol. 78, pp. 51–64, 2014.

[26] C. Zieliński et al., "Variable structure robot control systems: The RAPP approach," *Robot. Auton. Syst.*, vol. 94, pp. 226–244, 2017.

Maksym Figat (Member, IEEE) received the M.Sc./Eng. degree in computer science (modelling CAD/CAM systems) and the Ph.D. degree (with distinction) in robotics from the Warsaw University of Technology (WUT), Warsaw, Poland, in 2013 and 2022, respectively. Since 2013, he has been with the Faculty of Electronics and Information Technology, Institute of Control and Computation Engineering (ICCE), WUT. He is a Member of the Robotics Group in ICCE working on the design of robot controllers and programming methods. His main scientific research focuses on automatic methods of robot control system generation based on a formal specification. His research interests include robotics based on model driven engineering, domain specific languages, embodied agent approach, and formal specification of robotic control systems.



Cezary Zieliński (Senior Member, IEEE) received the M.Sc./Eng., Ph.D., and Habilitation degrees in control and robotics from the Warsaw University of Technology (WUT),Warsaw, Poland, in 1982, 1988, and 1996, respectively. He was a Full Professor with WUT. Specialises in robotics, he was the Head of the Robotics Group, Institute of Control and Computation Engineering, working on the design of robot controllers and programming methods. His research focuses on robotics in general and especially include, robot programming methods, multirobot system controllers, robot kinematics, robot force control, visual servo control, utilisation of sensors in robot control, and design of digital circuits. He was the Vice-Dean for General Affairs at the Faculty of Electronics and Information Technology of WUT during 2016–2020. He is currently the Director of the Institute of Control and Computation Engineering (ICCE).