



RobMoSys

H2020-ICT-732410

RobMoSys

**Composable Models and Software
for Robotics Systems**

Deliverable D3.4:

**Composable software and tooling for motion,
perception and world-model stacks**



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N732410.



Project acronym:	RobMoSys
Project full title:	Composable Models and Software for Robotics Systems
Work Package:	WP 3
Document number:	D3.4
Document title:	Composable software and tooling for motion, perception and world-model stacks
Version:	1.0
Delivery date:	December 31st, 2020
Nature:	Report (R)
Dissemination level:	Public (PU)
Editor:	Herman Bruyninckx (KUL)
Authors:	Herman Bruyninckx (KUL), Marco Frigerio (KUL), Filip Reniers (KUL), Nico Hübel (KUL), Enea Scioni (KUL)
Reviewer:	Alessandro di Fava (PAL)

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement N°732410 RobMoSys.

Contents

1	Introduction	4
2	Software tools	5
2.1	Property Graph	5
2.1.1	Design and implementation	5
2.1.2	Code repository	7
2.2	Robot Model Tools	7
2.2.1	Software	8
3	Deployment architecture	10
3.1	Introduction	10
3.2	Design and implementation	10
3.2.1	Algorithm	10
3.2.2	Composition of algorithms	14
3.2.3	Activity	18
3.2.4	Thread	18
3.2.5	Process	18
4	Additional works	22
4.1	Educational modules	22
4.2	Microblx	23
4.2.1	Code repository	23

1. Introduction

This Deliverable describes the work on the **tools** that support **to model and to compose** the **fine-grained software functions and data structures** that make up the **platform** functionalities of robotic systems. The aim of this Deliverable is to provide a very concrete and technical introduction to the software tools developed within WP3, and it is targeted to early adopters of such tools.

RobMoSys advocates model-driven development techniques considering **composition** as a necessary first-class primitive for models, tools and software components. Composability enables **interoperability** and **re-usability**.

RobMoSys adopts the “Unix philosophy”, that is, a tool shall do one thing only, but do it well. The same philosophy is adopted for the design of *models*, and the design of the *functions* and *data structures*, which realise some functionality.

This approach towards “*minimality*” does not prevent the creation of “*monolithic*” tools or frameworks,¹ that are often expected in specific application domains because they bring the “user friendliness” of *de facto* “standardizations” to that domain. The drive for minimality is expected to help interoperability between such tools, because the development efforts of the common, re-used parts can be shared.

RobMoSys adds a fundamental extension to the Unix philosophy: if the latter’s *universal interface* boils down to not much more than text streams, RobMoSys’ aim is to exploit conformance **to a set of meta-models**; in other words, interaction is not just via text, but via text with a formally modelled meaning (see D2.1, D2.2 and RobMoSys Ecosystem Organization²). The disadvantage of such “semantically rich” interfaces is the non-trivial effort to design and realise them (hence, one of the goals in RobMoSys), paid off by the facilitation of tool interoperability.

¹The software of a robotic solution is already, *de-facto*, an integration of multiple technologies and software libraries. However, most of today’s frameworks that claim to address composability enforces the developer to make strong technological bindings, adopting a static workflow with limited interoperability with other tools and approaches, or, worse, leaving the developer the responsibility to address those directly, by manual programming.

²https://robmosys.eu/wiki/general_principles:ecosystem:start

2. Software tools

2.1 Property Graph

The property graph is a generic structure for the representation of heterogeneous and linked data. The graph hosts the entities and the relations that one wishes to represent. Both entities and relations, in general, have a set of key–value properties representing additional information. The power of the property graph model lies in it being a very generic *mechanism*, as any object can be inserted in the graph and there are no constraints on the relations that can be established among them.

Such flexibility allows to model and interconnect heterogeneous objects, statically and dynamically, in ways that need not be pre-determined. Existing models, data objects, etc., can be *composed* together into a rich knowledge base. The information that can be modeled in the graph is not limited to traditional data-record-like items, and it includes, among other things, (symbolic references to) algorithms; it is then possible to relate, for example, geometric features with the available image processing algorithms for their detection. Also, composability implies de-composability, meaning that different subsets of data can be extracted from the graph to serve algorithms with different inputs; relations in the graph can always be ignored if not relevant, or exploited otherwise.

A natural limitation of the flexibility of property graphs is that they require the user to be aware about the *existence* of the relations; in fact, the graph can be explored and searched for connections, but that comes at the price of higher complexity at the user side.

2.1.1 Design and implementation

This section describes our current implementation of a property-graph library, created to support the common requirement of knowledge representation in diverse robotics applications. Our implementation conforms with the primitive meta-models of hypergraphs and entity-relation documented in the RobMoSys wiki.¹ It meets the requirements of multi-robot systems about footprint size and performance (latency etc.). It provides a property graph mechanism as a *library* that can be embedded inside the in-process RAM memory of a realtime motion stack component.

Figure 2.1, also shown in Deliverable D3.2, gives a logical overview of the structure of the library.

1. The bottom layer represents the `cgraph` C library, which provides the *generic mechanism* to construct in-memory graphs supporting node attributes (called “records” in CGraph terminology).²
2. `cgraph` is hidden behind the property-graph-specific C API exposed to clients. The API reflects the main concepts of the property graph, by means of operations to construct, remove, connect nodes and edges, traverse edges, set and get key/value properties. Although the core API is in C, we provide a convenience wrapper in Lua; the additional functionalities described in the following are also implemented in Lua and are based on the core Lua API. Note that this layer still exposes the concrete, low-level elements of the graph: nodes and edges.

¹<https://robmosys.eu/wiki/modeling:hypergraph-er>

²See https://graphviz.gitlab.io/_pages/pdf/cgraph.pdf and <https://www.mankier.com/3/cgraph>.

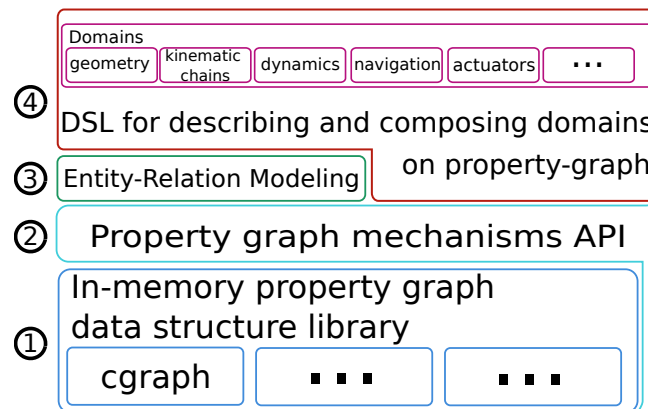


Figure 2.1: Current implementation design of the property-graph mechanism.

3. Layer 3 is a pure Lua API exposing the concepts of Entity and Relation. Entities and relations can be created and interconnected. The software still uses the underlying property graph mechanism to store the data, but the graph-level concepts (nodes and edges) are now hidden from the user. Both entities and relations can have key-value properties. More on this part later.
4. The last layer refers to a Domain Specific Language for the definition of the entities/relations of a specific domain of interest, e.g. Vector in a Geometry domain. The point of such definition is to impose *constraints* such as the number and kind of arguments of a relation, or the acceptable properties for an entity. For example, a 3D vector must have an origin and an end Point. These constraints are *not* enforced in Layer 3.

The implementation of this functionality at the current stage consists of an additional Lua module and a Lua-internal DSL for the domain definition. The interpretation of the domain dynamically creates an API tailored for it, so that specific creation functions can be used in addition to the more general functions of Layer 3.

Internally, the implementation of the Entity/Relation layer (number 3 in the figure), enforces the following policy:

- any Entity gets represented by a node in the underlying graph
- any Relation is also an Entity
- graph edges are used only to connect the (node representing the) Relation with the other entities that constitute its arguments.

This policy has two main advantages: it automatically enables higher order relations, that is relations whose arguments are other relations. It enables reuse of the same, single mechanism which deals with graph node properties, to implement properties for both entities and relations. Figure 2.2 illustrates the graph induced by the creation of a relative-position and relative-orientation relations between two Cartesian frames (entities), according to described policy: The figure shows four graph nodes, for the two frames and for the two relations. Actual graph edges are used to identify the arguments of the relation, in this case named `with_respect_to` and `of`.

Note that users of the core property graph API (be that the Lua or the C API) are not constrained by this policy and are free to decide when to use nodes and when to use edges.

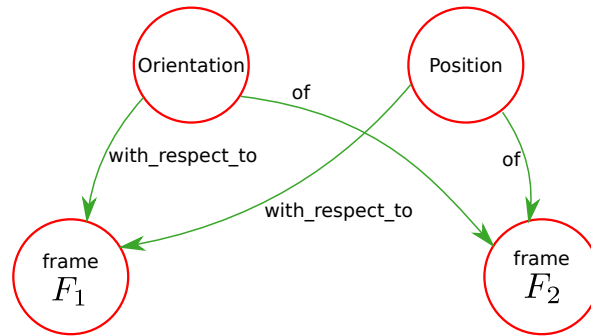


Figure 2.2: A property graph representing symbolically the relative pose between two frames

2.1.2 Code repository

The source code of the property graph implementation is available at the following address: <https://gitlab.kuleuven.be/u0129795/rt-graphkul.git>. The project includes a readme file with installation instructions and dependencies. Numerous simple examples are available to illustrate the intended behaviour of the API. Most of the time, the property graph created by these sample is exported to the de-facto standard dot format, which can be converted to an image for visual inspection.

2.2 Robot Model Tools

A model of the kinematics/dynamics of the robot is a core requirement of most robotics applications. There exist various robot model formats developed within different software packages. Perhaps the most common one is the URDF file format³ from the ROS ecosystem.

The URDF is a monolithic format supporting in a way a number of aspects of a robot model. If these aspects (like connectivity, geometry, inertia, attachment of frames/points, sensors, meshes for 3D visualization, etc.) are modeled separately, and corresponding tooling exists, it is much easier to support alternative representations for the same information (e.g. how to specify the relative pose of the frames on the model). Models addressing a self contained concern are composable, and composability is reflected in the tooling. Conversely, a monolithic format leads to monolithic tools and larger, harder specifications.

The KinDSL format⁴ focuses only on the aspects strictly required by dynamics solvers, and therefore slightly alleviates the issue. It however suffers from the same fundamental limitation, of imposing a specific way of describing information that could be conveyed differently.

In an ideal scenario, the standardization and the concrete representation of meta models would enable coexistence and interoperability of a multitude of formats, provided that instances of these format would include explicit references to the meta models they conform to. The meta models are a formal representation of the structure and the information content of robot models, for example the rigid-body and kinematic-tree meta model (according to which a dynamics model is a set of rigid bodies, connected by joints in specific locations, etc).

³See <https://wiki.ros.org/urdf>

⁴<https://robocogenteam.bitbucket.io/rmodel.html>

2.2.1 Software

Within RobMoSys, we developed a software library and command line tool designed to support common operations with robot models, such as comparisons and conversions. Care was taken to separate the implementation of different aspects of robot models, like connectivity and numbering scheme (ordering), to make the software capable of dealing with diverse formats. Although we have not achieved the ideal case of explicitly-represented and programming-language-independent meta models, this software plays the role of a set of meta models, by checking and enforcing the appropriate constraints upon loading an input model.

The Robot Model Tools is a command line program written entirely in Python. At the moment it supports the formats URDF, KinDSL and a series of YAML based models which reflect the different aspects of models implemented internally. These include:

- connectivity: a purely topological model, which is essentially a set of *kinematic pairs*
- numbering scheme: an ordering composed with the connectivity model; induces the (choice of) robot base and the successor/predecessor relations between joints and adjacent links
- geometry: metric information required to localize the main reference frames attached to the links and the joint axes; these frames can be referenced symbolically when composing additional models, such as the
- inertia properties: mass, center of mass and moments of inertia for each rigid link of the mechanism; both the center of mass and the moments must refer to the frame from which they were measured.

This separation and the possibility to refer symbolically to parts of the robot (like links or joints) and to other *attachment points* (like the Cartesian frames or specific points on the links), enables composability with other models and the extensibility of the tool. For example, meshes for the 3D visualization of the robot can be “attached” to an existing model with a simple reference to the link name.

The tool comes with an experimental module that connects to an external program called Meshcat, for 3D visualization in the browser. This module does not bring in additional dependencies or complications for the core of the tool. It does not require changes in the basic robot models (e.g. connectivity). The meshes to be used for the visualization are specified with an *additional* file that simply refers to the robot model name (i.e. the *context*) and then includes a mesh file path for each link. This file is currently in YAML format.

JSON-LD

A development branch of the robot model tools is dedicated to the usage of models based on JSON and JSON-LD. Such development is motivated partially by the maturity of the standards and by the expertise of the community behind them (web and internet technologies). The primary advantage of JSON/JSON-LD however, is the native support for mechanisms facilitating the explicit representation of meta-model constraints, as advocated by the RobMoSys approach.⁵ In particular, the JSON schema and the JSON-LD context. The former allows to enforce some structure (like the presence of certain attributes) of models; the latter enables a certain degree of

⁵“Native support” refers to description of the concepts in the standards, and the corresponding availability of conforming software tools.



freedom in the choice of terminology inside the models, because different terms can be recognized to “mean” the same thing.

Code repository

The source code (in the Python language) of the Robot Model Tools is available at the following address: <https://github.com/mfrigerio17/robot-model-tools>. The tool requires a Python version greater than 3.3. The readme file included in the repository describes the installation procedure and the dependencies.

3. Deployment architecture

3.1 Introduction

A robotic system is a complicated cyber-physical system. It has to execute and coordinate a multitude of computations required to realize a task or a behaviour. Such computations include the processing of low-level sensor inputs and actuator commands, as well as more high-level (semantic) reasoning (e.g. populating a world model) with the ultimate goal of reaching situational awareness. The benefits of separating the realization of all these computations in software from the deployment on a particular computational hardware setup, are described in detail in chapter *Meta models for behaviour: activities, and their interaction and coordination* of (Bruyninckx 2020). This chapter describes a reference software implementation which serves as a template for the control of any cyber-physical system. It conforms to RobMoSys concepts and models, as described in the aforementioned reference. It is very flexible, allows composition and it does not impose any framework lock-in.

3.2 Design and implementation

The deployment architecture differentiates between 4 computational/deployment entity types:

- Process
- Thread
- Activity
- Algorithm

Figure 3.1 shows the strict hierarchical tree of such entities. One application consists of one process. A process can have several threads, each one having several activities, which in turn compose together a number of algorithms.

3.2.1 Algorithm

The lowest level of the deployment tree consists of algorithms. An algorithm is the entity that executes the computations. It consists of data structures and functions.

An algorithm is deterministic. The same configuration and input data will always lead to the same results, regardless of the context and the number of executions. As a consequence, the functions of an algorithm need to be pure functions without any side-effects. This is achieved by the fact that they only act on their own state.

Another important aspect of an algorithm is that it operates in a synchronous context. Synchronicity requires that:

- The sequence of functions programmed in the code of the algorithm is reflected in the actual order of execution.
- The data is always available at any moment when the functions need it.

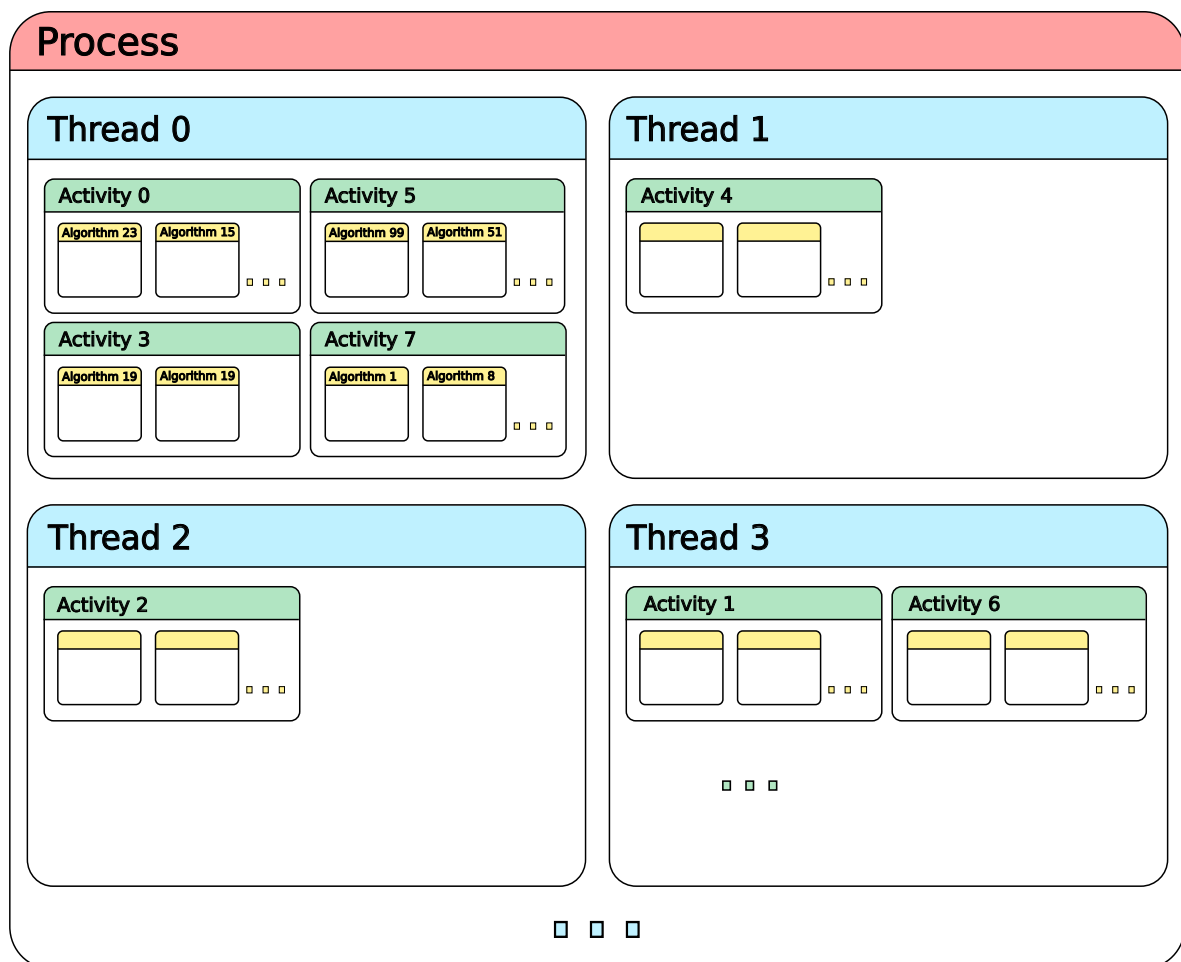


Figure 3.1: Overview of the deployment architecture

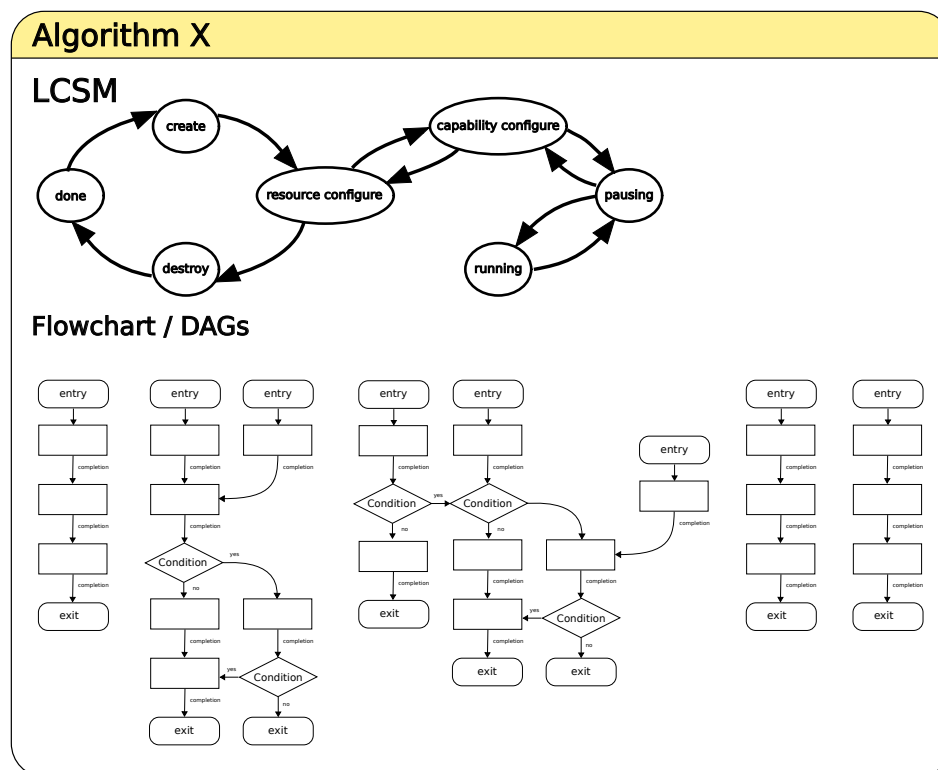


Figure 3.2: Structural components of an Algorithm. LCSM stands for Life Cycle State Machine, which is used to coordinate the behaviour of the algorithm. The flowcharts or Directed-Acyclic-Graphs model the sequence of the computations.

The developer of an algorithm does not need to worry about simultaneous access and other race conditions.

Figure 3.2 shows the building blocks of an algorithm in the deployment architecture implementation. An algorithm is composed of a life-cycle state machine and a collection of flow charts which concretely model the sequence of computations. When started, an algorithm has to go through all necessary steps of initialization until it can perform its nominal execution. Likewise when an algorithm finishes execution, it can be reconfigured or completely terminated. The life-cycle state machine makes sure that the algorithm follows the correct sequence of states.

State	Description of intended state behaviour
Create	Memory allocation of required data structures
Resource Configure	Configuration of data structures necessary for own operation
Capability Configure	Configuration of properties during active operation
Pausing	Idling behaviour <ul style="list-style-type: none"> • immediate resume possible • necessary for dealing with state dynamics
Running	Execution of actual algorithm
Destroy	Memory de-allocation
Done/Dead	The algorithm is stopped, and can be restarted only by an external entity

Table 3.1: The states of the Life-Cycle-State-Machine.

As will be shown in the sections *Activity* and *Thread*, the life-cycle state machine is not reserved to be only used in algorithms. It appears in every element of the deployment architecture described in this document. The seven states with their intended state behaviour are listed in table 3.1. A basic algorithm will go through three phases:

- Initialization: The algorithm starts (Done) and traverses all the way to Pausing.
 1. In Create the algorithm allocates all its state structures.
 2. In Resource Configure it does a configuration of the resources required for proper functioning. Most of the time these resources need to be configured only once.
 3. In Capability Configure the algorithm capabilities are configured. This boils down to the initialization of the computational parameters.
 4. In Pausing the algorithm waits to become active and start performing its nominal execution.
- Execution: The life-cycle state machine goes through a full cycle of an execution run.
 1. The algorithm goes to Capability Configure to get the latest configuration values and resets the computational state in case of a previous run of the algorithm.
 2. The algorithm subsequently traverses Pausing and reaches Running. Here it will perform its computations until a predefined stop condition has been attained.
 3. When the stop condition is fulfilled, the algorithm returns to Capability Configure to transfer results and other termination related configurations; finally it goes back to Pausing.

- Deinitialization: The algorithm visits the states in reverse order as in initialization: from Pausing back to Done. In Capability Configure and Resource Configure the algorithm resets everything to safe default values. In Destroy it deallocates the memory and it eventually stops.

Most state transitions trigger on completion of the previous state. Only some transitions trigger based on the computations. The stop condition for the execution is an example of such a condition. For every state in the life-cycle state machine, the algorithm will execute one function or a sequence of functions which results in the intended behaviour of every state. When creating an algorithm, one has to specify a function per state that will be called whenever the algorithm is executed. This single function can directly carry out a computation or it can contain a solver to iterate over an array of functions that have to be called sequentially. In the most general case a solver can be assigned to serialize a Directed Acyclic Graph (DAG) which encodes the flowchart of the algorithm. Such a DAG consists of four types of blocks: process blocks, decision blocks, terminal entry blocks and terminal exit blocks. A process block corresponds to a pure function that implements an operator. A decision block evaluates a boolean expression to decide which of its two outgoing branches to follow. A decision block is the only block that allows the splitting into branches. The entry and exit blocks are the entry and exit points of the DAG, respectively. The DAG can have multiple entry and exit points. To run an algorithm which is declaratively specified in a DAG, the solver only needs the entry point in the DAG. When this information is provided, the solver executes every function in the flowchart until an exit block is reached. Although the mechanism allows to model even the simplest operation as a process block in the flow chart, it is intended that the algorithm developer grounds the blocks at a granularity that makes sense. The virtue of having a declarative model of an algorithm that is “solved” at run-time into a schedule gives the possibility to alter the control flow during operation without the need for a full shutdown of the algorithm.

3.2.2 Composition of algorithms

The deployment architecture allows the composition of algorithms. The algorithms are individually designed to work within a synchronous context. Each algorithm has its order of function execution programmatically fixed and its data is always available. A composed algorithm has to comply to the same constraints to still be called an algorithm. Algorithms are therefore composed into an *event loop* for mutually concurrent execution. Whether an algorithm is finished and a subsequent algorithm can start depends on the data. The exact spot in time when the next algorithm has to start is not fixed in the program code. The developer therefore needs a mechanism to let synchronous algorithms react to asynchronous interactions with the other algorithms, also when the timing of the algorithms’ execution is unknown. That mechanism is the architectural pattern of the event loop.

Figure 3.3 shows an algorithm composing an eventloop with other algorithms (referred to as sub-algorithms in this section). An algorithm uses a template of 3 *Cs* of the *Separation of Concerns* to fill up its event loop. These are:

1. Coordinate: trigger coordination mechanisms (finite-state machines (FSMs), petrinets, ...) which results in a change of status flags and new events.
2. Configure: react to status flags and events to compose the event loop schedule.
3. Compute: execute the composed serialized set of synchronous algorithms.

The finite-state machine (FSM) and petrinet models contain the bookkeeping information necessary for the coordination of the execution dependencies of the sub-algorithms. Instead of coding the direct coordination interaction inside the sub-algorithms themselves, the coordination and synchronization of the control flows is out-sourced to a third party. As a consequence, the sub-algorithms need not know about each other, and only interact with a “mediating” algorithm via a protocol based on simple flags. Note that neither the FSMs nor the petrinets are algorithms themselves. The mediating algorithm that uses the FSM and petrinet models for coordination must look up the state of the FSM and the marking of the petrinet, at its coordination step. Readers are referred to the corresponding sections of chapter 2 of (Bruyninckx 2020), for the Petri Net and FSM terminology in the context of algorithm/activity coordination.

The *Coordinate* step for filling up the eventloop in Fig. 3.3 triggers first the FSM to determine the phase in which the algorithm is. There are three phases: initialization, execution and deinitialization. Every phase has an associated Petri Net. The FSM in this example is currently in the *execute* state which means that the yellow petrinet is active. The green petrinet was previously active and the red petrinet has not become active yet. Next, the active petrinet is triggered.

When a petrinet is triggered, if all the incoming places of a transition contain a token, the transition is *enabled*. When a petrinet is triggered, all enabled transitions are fired. When a transition is fired, a token is consumed from all incoming places and a token is produced in all outgoing places. The example in figure 3.3 shows an algorithm which serves as a factory of four sub-algorithms. The green petrinet takes care of the synchronization of the initialization of the four sub-algorithms. The yellow petrinet takes care of the nominal execution coordination, and the red petrinet of the deinitialization of the four sub-algorithms. The nominal execution behaviour is the following. When the four sub-algorithms are created and fully configured, they are scheduled by the mediator. The sub-algorithms are either iterative or execute until completion. In both cases the sub-algorithms signal with a flag that they are *Complete*. The algorithm processes a data flow. When new data becomes available, sub-algorithm 1 and 2 can process it *simultaneously*, when both are finished, sub-algorithm 3 and subsequently sub-algorithm-4 process the data.

After *Coordination*, in *Configuration* the event loop schedule is composed with *schedules* based on the flags set by the coordination mechanisms in *Coordination*. A *schedule* is composed of a set of *named functions*: a combination of a function and its arguments. A *schedule* can therefore contain algorithms (a combination of the generic *do_algorithm* function and the algorithm configuration structure), but also any combination of function and arguments.

To determine which *schedules* to add to or remove from the eventloop schedule, one compares the *add_schedule* flag with the *active_schedule* flag for every schedule that can be scheduled for the given phase. If these flags are different, the respective *schedule* is looked up in the lookup table of all available schedules and added to or removed from the eventloop schedule.

At last, in *Compute* the eventloop schedule is executed by iterating over this serialized list of *schedules*.

Coordination via flag-based protocol

An explicit policy, based on the usage of the petrinet model for scheduling, is adopted to establish the relation between tokens in the petrinet and status flags to communicate information among the sub-algorithms. A flag is a variable that indicates the outcome of a boolean expression. The existence of a token in a place is equivalent to a boolean flag. If a token is in a place, the corresponding flag is true. When there is no token in that same place, the flag is false. We can detail this relation further by distinguishing four types of places:

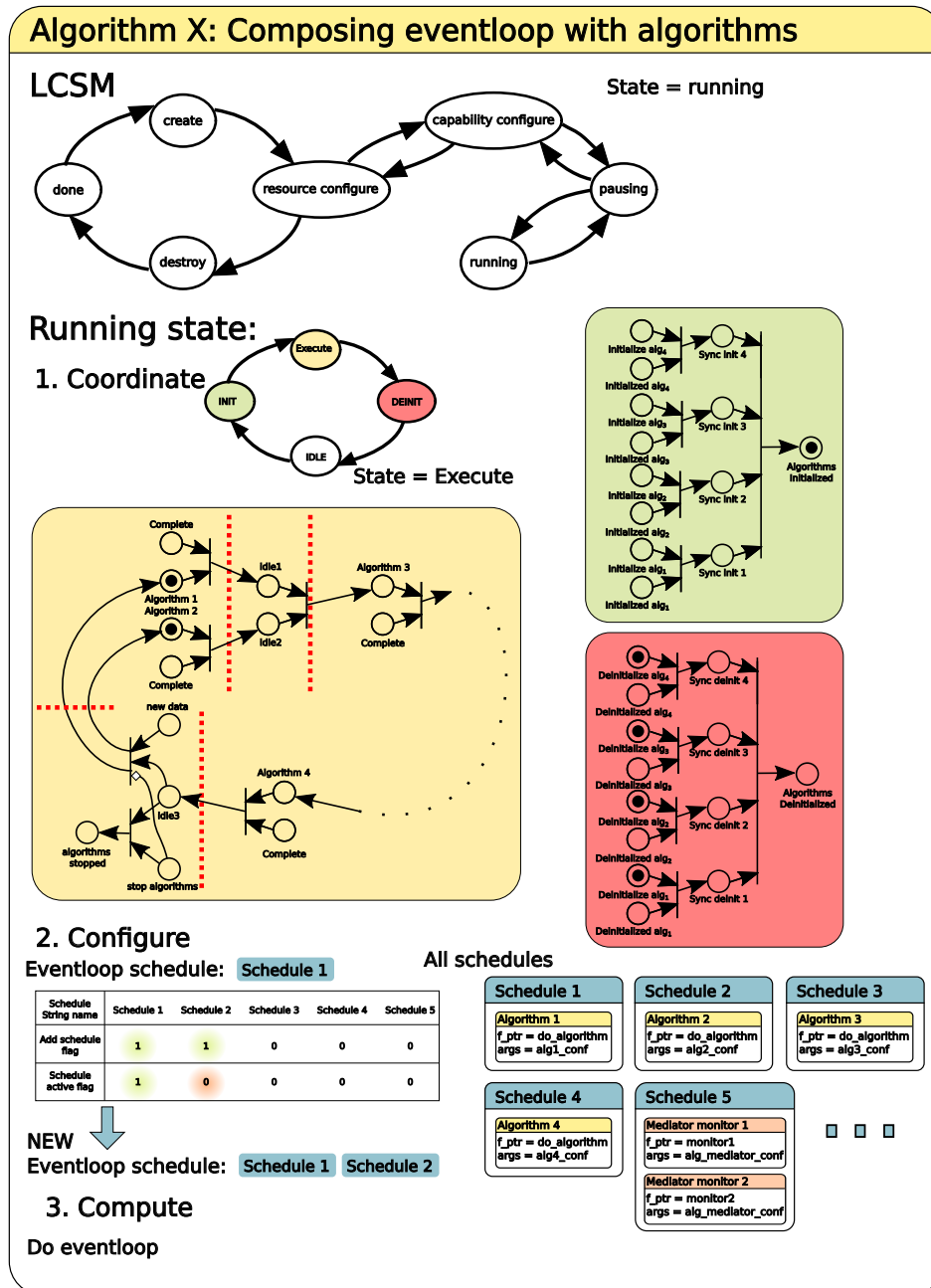


Figure 3.3: Structural components of a mediator Algorithm composing different subalgorithms. Different Petri-Nets (larger squares) are used for coordination of the subalgorithms depending on the current phase (see the matching colors).

1. converting sources
2. tracking sources
3. converting sinks
4. tracking sinks

The working principle of all four types is displayed in figure 3.4. If a place is a converting source, the place is filled with a token when its corresponding flag is true. The flag is subsequently set to false (a). If a place is a tracking source, the place gets a token when its corresponding flag is true. Its flag however is only observed. It is not set to false (3.4(b)). Converting and tracking sinks operate similarly. Now the place is observed to either decide whether a flag should be true or false. A sink sets a flag to true when its corresponding place has a token. If it is a converting sink, the token is removed from the petrinet (3.4(c)). If it is a tracking sink, the token is only observed and is not removed from the petrinet (3.4 (d)).

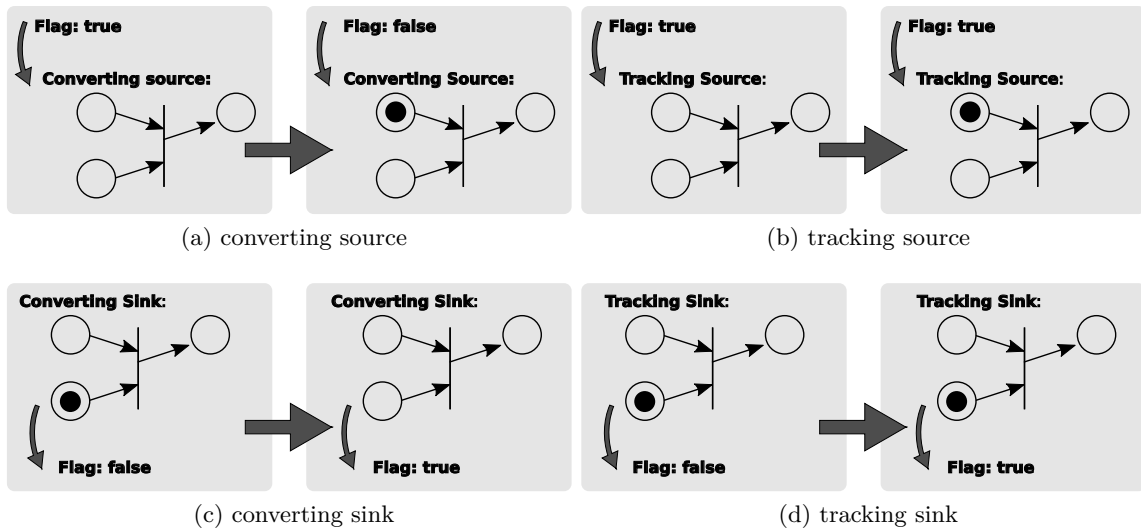


Figure 3.4: All 'port' types for token to flag and flag to token conversion

Figure 3.5 shows how a petrinet is used for scheduling and how the flag conversion is typically used. The first type of places used for scheduling are tracking sinks. When a token is in place *Algorithm 1* or *Algorithm 2* the corresponding flag is set high to indicate that those algorithms have to be added to the eventloop schedule. When the token is removed from that place through firing of the connected transition, the algorithm should be removed from the eventloop schedule. The second type is the converting source. It communicates the value of a flag to the petrinet by adding a token in that place. Converting sources in figure 3.5 are *Algorithm 1 Complete*, *Algorithm 2 Complete* and *Stop Waiting*. The converting source places are always blocking a transition. In the example, these places are directly tied to flags that indicate the completion of the current algorithm, or in the case of *Stop Waiting* a general synchronization condition. The third type shown here is converting sink. The only converting sink place in figure 3.5 is *Notify start*. This is a place to generate a flag to indicate the status in the petrinet which can be used elsewhere. At last, the fourth type is the tracking source. Place *Ready Status of other algorithm* tracks the "readiness" of an other algorithm. The token appears and disappears according to the

value of the flag that is tracked. This place is used for the synchronization between this and another algorithm.

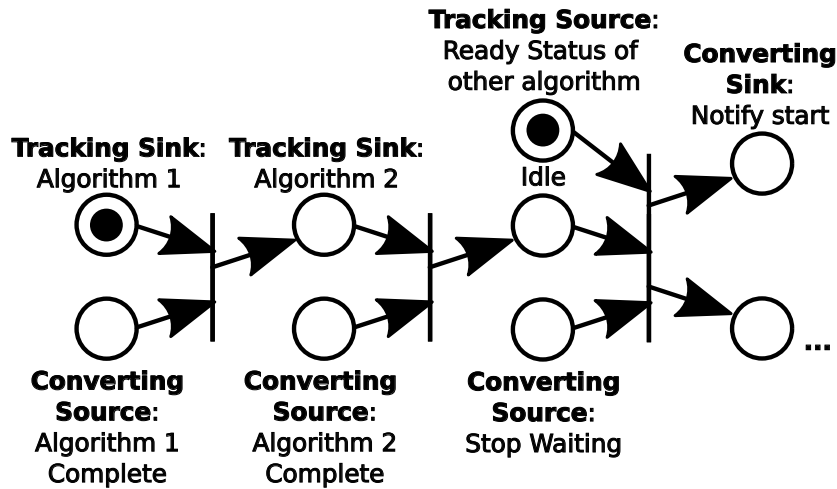


Figure 3.5: Example of using a petrinet for algorithm scheduling

3.2.3 Activity

An activity has almost the same structure and behaviour as a composed algorithm, described in the previous section. It also uses an eventloop to serialize the execution of algorithms. The major difference is that the activity is the interface between the synchronous and asynchronous execution. It therefore also schedules an explicit *Communication* step before it does all coordination and configuration as shown in figure 3.6. In this step it interacts with the other activities via the provided asynchronous data channels such as stream buffers to receive data, flags and events.

3.2.4 Thread

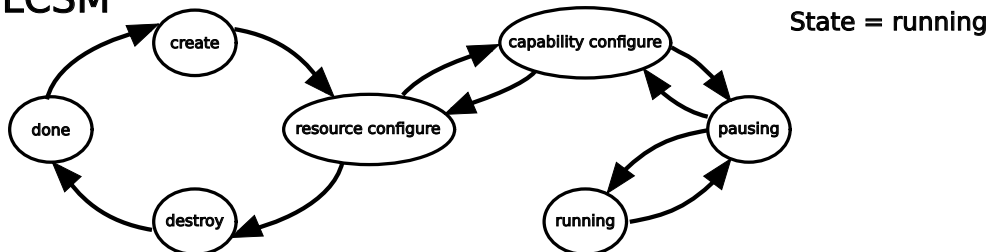
A thread, in the deployment architecture, is the container in which activities are serialized to be run on a CPU core. It is the interface between the application-centred activities and algorithms and the operating system. There are various scheduling policies which are part of the operating system such as Task Queue, Priority Scheduling, Round-Robin scheduling, etc. In Figure 3.7, a time-triggered periodic scheduling strategy embedded in a thread component is shown. When the thread reaches its nominal running state, it composes and executes its eventloop (see figure 3.8) and sleeps for the remaining time to run at a prescribed cycle time. This thread does not take into account time overrun, because it cannot preempt the computations when time exceeds the cycle time. However, that is a condition that can be easily monitored and reported.

3.2.5 Process

The process is the single entry point for the execution of the application. It is responsible for ownership of the resources that the operating system provides to the application. The process level is where one can decide what CPU a thread can be assigned (processor affinity) and the scheduling of CPU time among threads within the application. The process lives in kernel space

Activity X: Composing eventloop with algorithms

LCSM

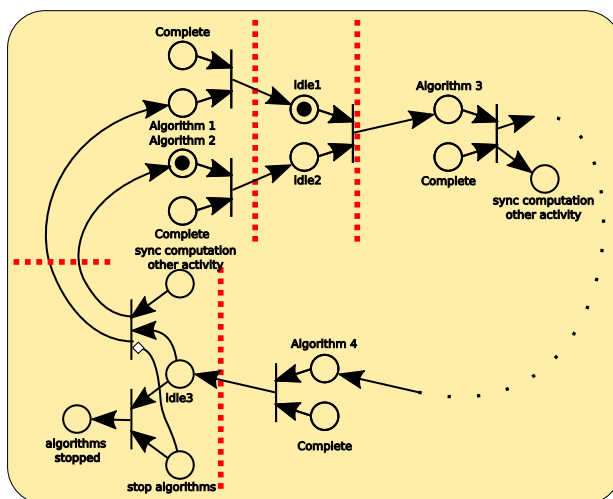


Running state:

1. Communicate

I/O ports -> Incoming Events, Flags and data (buffers / streams)

2. Coordinate



2. Configure

Eventloop schedule: **Schedule 9** **Schedule 10**

Schedule String name	Schedule 9	Schedule 10	Schedule 11	Schedule 12	Schedule 13
Add schedule flag	0	1	0	0	0
Schedule active flag	1	1	0	0	0

NEW

Eventloop schedule: **Schedule 10**

4. Compute

Do eventloop

All schedules

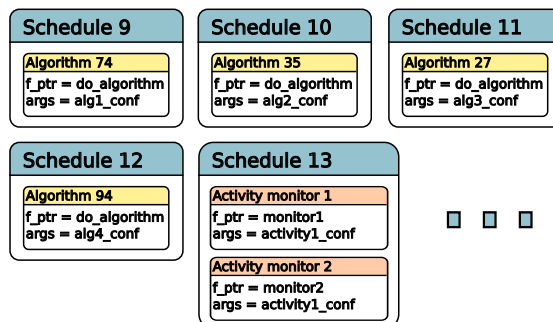


Figure 3.6: Structural components of an Activity composing an eventloop

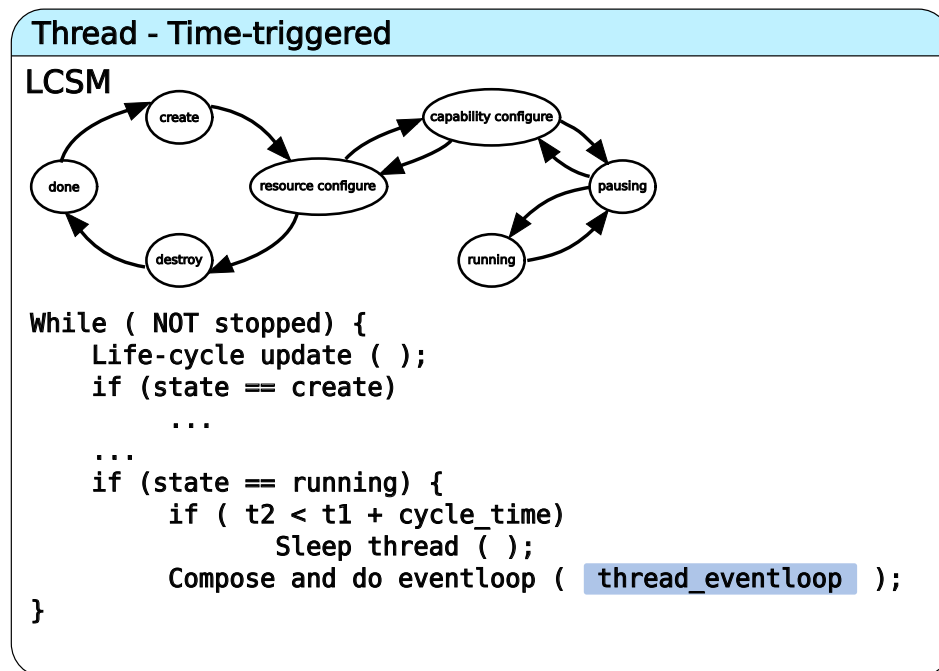


Figure 3.7: Structural components of a time-triggered thread

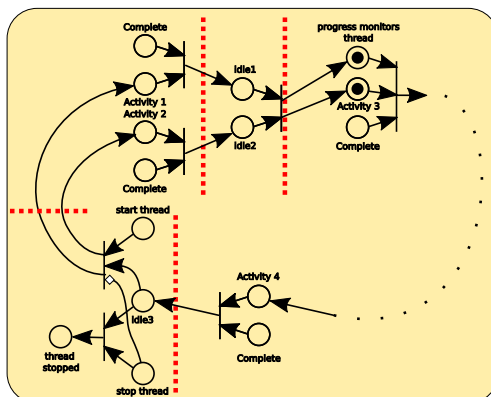
and the ownership belongs to the operating system. The configuration and composability of processes depends on the operating system in which the application(s) run(s).

Thread X: Composing eventloop with activities

1. Communicate

I/O ports -> Incoming Events, Flags and data (buffers / streams)

2. Coordinate



2. Configure

Eventloop schedule: **Schedule 22**

Schedule String name	Schedule 21	Schedule 22	Schedule 23	Schedule 24	Schedule 25
Add schedule flag	0	0	0	1	1
Schedule active flag	0	1	0	0	0

NEW

Eventloop schedule: **Schedule 23** **Schedule 25**

4. Compute

Do eventloop

All schedules

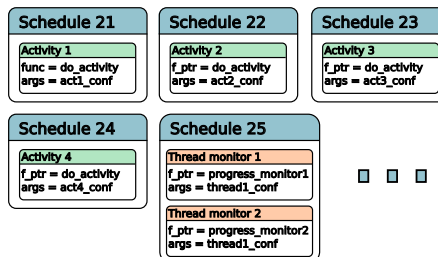


Figure 3.8: The composition and execution of the eventloop of a thread

4. Additional works

4.1 Educational modules

To properly design and implement “realtime components” as building blocks for a robotic system involves a large learning curve. And often also access to expensive robotics hardware. During the last year of the RobMoSys project, we have developed an **educational platform** (Fig. 4.1), that can be built with a budget of about 500 euros.

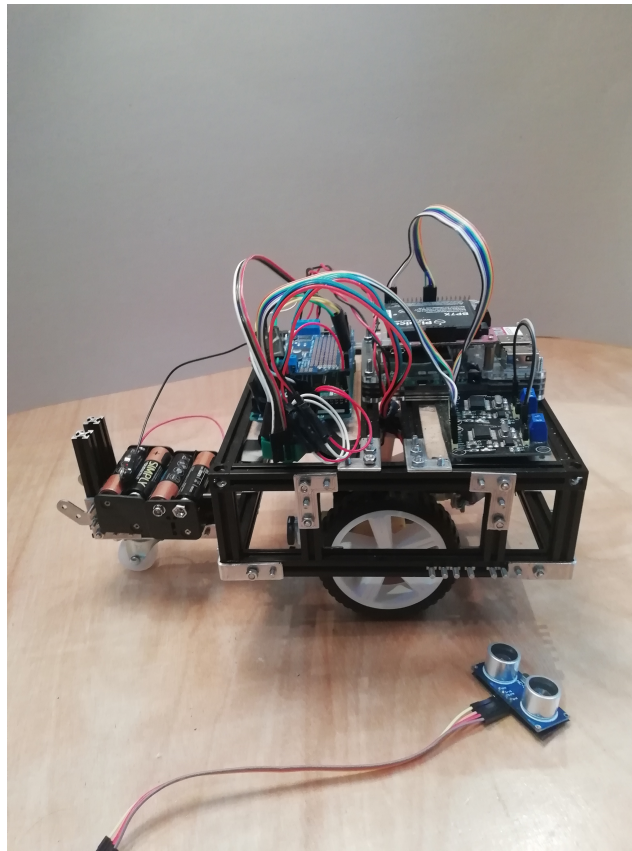


Figure 4.1: Assembled educational “mobile robot” platform, as entry-level hardware to showcases RobMoSys compliant perception and motion stacks.

Still, it contains all essential parts of a distributed robotics control system:

- several computers: Raspberry Pi as “brain”, and Arduinos as “local drivers” for the motor controllers.
- a CAN field bus, connecting motors and some sensors.
- proprioceptive sensing (encoders and IMU) as well as exteroceptive sensing (ultrasound, LIDAR (at extra cost)).

Hence, many of the building blocks and architectural patterns can be illustrated with this simple device. The core educational modules are about how to apply the RobMoSys compliant **multi-**

threaded component model, with **circular buffers** as core mechanisms for **asynchronous** control and coordination. Especially the entry-level explanation and implementation of how to deal with asynchronicity on a professional level is expected to become the “unique selling point” of this module.

Full documentation and software will be made available publicly. And the modules are further developed in several courses.

4.2 Microblx

`microblx` is a lightweight framework to build hard real-time systems based on the composition of functional blocks. It is designed around a canonical component model with ports for data exchange, configuration hooks, and a state machine for the management of the “block” life cycle.¹

`microblx` was an existing software created independently of RobMoSys, by Markus Klotzbuecher and some collaborators. However, it largely conforms to the RobMoSys approach and best practices, especially after the improvements developed over the course of a RobMoSys Integrated Technical Project, COCORF (see deliverable D5.9). Due to its focus on embedded and hard real-time applications it constitutes a relevant complement to the RobMoSys tools and software baseline², and therefore it is mentioned in this document.

4.2.1 Code repository

The main source code repository of `microblx` can be found [here](#). The implementation of the RobMoSys “mixed-port” concept for interoperability with other frameworks (ROS in this case) is available [here](#). A relevant example of the composability feature of `microblx` is documented [here](#).

¹See the documentation [here](#). © Copyright 2012-2020, Markus Klotzbuecher et al.

²See [this page](#) of the RobMoSys wiki.

References

Bruyninckx, Herman (2020). *Design of Complicated Systems*. Tech. rep. KU Leuven. URL: <https://robmosys.pages.gitlab.kuleuven.be/>.