Formal Architectural Patterns for Adaptive Robotic Software

James Baxter[1][0000−0001−6083−9607], Bert van Acker[2][0000−0002−3854−5159], Morten Kristensen[3][0009−0008−8467−7567], Thomas Wright[3][0000−0001−8035−0884], Ana Cavalcanti[1][0000−0002−0831−1976], and Cláudio Gomes[3][0000−0003−2692−9742]

[1] University of York, UK
[2] University of Antwerp, Belgium
[3] Aarhus University, Denmark

Abstract. It is often the case that a robot must adapt to unexpected changes in its environment. It is, however, important that these changes can be demonstrated to maintain the safe operation of the robot. The adaptive systems community has developed the MAPE-K pattern as a widely recognised conceptual architecture. We propose extending MAPE-K to incorporate runtime verification, resulting in an architecture we call MAPLE-K. In this paper, we capture and formalise both the MAPE-K and MAPLE-K architectures using a domain-specific language. Additionally, we provide support for translation from architectural models to software models and code to facilitate the deployment of verified applications. MAPE-K is rarely maintained at the implementation level, but our work ensures traceability between the code and its design, enabling the use of architectural information to verify the correctness of the software.

Keywords: adaptive systems · robotics · formal modelling

1 Introduction

It is essential that robots behave safely, as they can cause damage to property or harm to humans nearby. A particular challenge is the possibility of unexpected changes in the environment. These may include the appearance of obstacles, sensor readings outside the range the robot can handle, or changes to the robot's body, such as failures. In all these cases, the robot must be able to adapt in a way that maintains safe and effective operation wherever possible.

The most common pattern for the development of software capable of adapting to unexpected changes is the MAPE-K architecture, where the system is overseen by a manager component. A widely accepted view of MAPE-K is that the manager proceeds in a cycle of four steps: monitoring inputs to detect anomalies; analysing the anomaly to determine its nature; planning to adapt the system; and executing the plan by signalling to the managed system to implement it. These four components form the "MAPE" of the MAPE-K, pattern and they are supported by a Knowledge Base (the "K"), through which information is shared between steps and across iterations of the cycle.

In the RoboSAPIENS project [18], we propose an extension of MAPE-K called MAPLE-K, which incorporates an additional legitimisation step to ensure that adaptations maintain the safe operation of the robot. This extra step checks that the plan produced (in the planning step) is safe before it is executed. It is useful to have legitimisation as a separate step, since safety cannot always be guaranteed during the formulation of the plan. In particular, neural networks may be used in the analysis or planning steps for greater adaptability, but neural networks cannot always ensure compliance with safety requirements so the legitimisation cannot be performed by a neural network. It is thus important that the planning and legitimisation steps be considered separately, since they may adopt different approaches. The planning step may also itself be adapted by a further MAPE-K or MAPLE-K component, whereas safety requirements would not permit the legitimate step to be adapted.

Additionally, we propose that the conceptual elements of MAPLE-K should also be applied at the implementation level. While MAPE-K is widely used to plan the structure of software, the pattern is often not reflected in the implementation. This jeopardises traceability between the deployment and the design of the code and hinders the compositional verification of adaptations.

This loss of traceability can be avoided through a Model-Driven Engineering (MDE) approach, where architectural patterns are reflected in system models, which form the basis for implementation. The connection is strengthened by automated transformation from the model to code.

The RoboStar framework [11] is a family of domain-specific languages for robotics, with support for verification via automated generation of formal semantics from models. In particular, RoboChart is used to describe software designs. RoboChart [19] models represent software components and their connections, describing behaviour via timed state machines or neural networks. Through transformation, RoboChart models can be validated and converted into code.

Another RoboStar language, RoboArch [7], supports the modelling of architectural designs. RoboArch embeds a layered approach to software architecture and allows the definition of layers using patterns that describe their internal structure. Automatic translation from RoboArch architectural models into RoboChart sketches ensures that architectural designs are reflected in the design models, from which formal verification and code generation are possible. The translation formalises the RoboArch architectural models.

Architecture and Analysis Description Language (AADL) [5,13] provides standardized notations for specifying hardware and software components of a system and their interactions. It can be used to describe the architecture in the detail required for deployment as well as its implementation.

In the work presented here, we propose the process for the development of robotic software shown in Fig. 1. A developer starts with a high-level model of the architecture of a system in RoboArch and generates a corresponding sketch in RoboChart, possibly enriching it with application-specific logic. This provides support for formal verification of the architecture. We then propose an approach to transform the RoboChart model into a corresponding AADL model, enabling

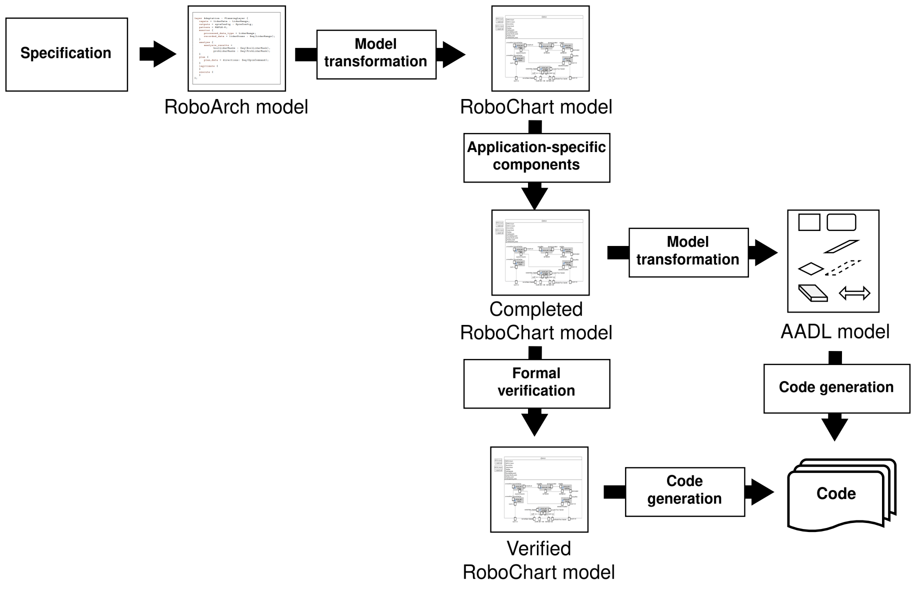

Fig. 1: RoboChart and AADL model-based development process

the development of a deployment model, annotating it with deployment-specific properties, and generating code skeletons for the interface and infrastructure code of each component. Using the code skeletons, the behaviour (the "application code") can be written in the chosen programming language or obtained automatically from RoboChart as well. The auto-generated code skeletons are aligned with the AADL/RoboArch architectures, ensuring that throughout this process the MAPLE-K loop itself, and particularly the Legitimate component, can be modelled, verified, and maintained all the way down to the code level. In this case, legitimisation can take advantage of the high-level structure of the RoboArch model to reason about the adaptations. Although our approach makes use of the particular notations of RoboArch, RoboChart and AADL, it demonstrates this more general process for development of software following a MAPLE-K pattern, transforming from higher-level specification notations to lower-level models and code.

In this paper, we contribute to the design of open-ended self-adaptive systems that require certification by presenting extensions of RoboArch and AADL to capture MAPLE-K architectures, with the traditional MAPE-K architecture as a special case. We also present a mapping from RoboChart to AADL, allowing a RoboArch and RoboChart design architecture to be translated into deployment. We demonstrate our approach using a robot that navigates through a space, avoiding obstacles, and adapting to faults and occlusions in its LiDAR sensor.

Next, in Section 2, we discuss related work on modelling MAPE-K. In Section 3, we describe RoboArch, RoboChart, and AADL. We present our model of MAPLE-K in RoboArch and RoboChart in Section 4, and in AADL in Section 5. We then discuss our mapping from RoboChart to AADL in Section 6. Our example is the subject of Section 7. Finally, we conclude in Section 8.

2 Related Work

The MAPE-K conceptual architecture was first introduced by IBM, as outlined in the seminal work of Kephart et al. [17] and further detailed in [16]. Over the years, several variations have been proposed [4,20,9,6].

In the context of MDE, several works align with our goal of formalising architectures based on MAPE-K loops, albeit with very different approaches and applications. For instance, Arcaini et al. [4] model self-adaptive systems as multi-agent systems, representing both the managed system and the managing subsystem as interacting agents. Their model utilises multi-agent ASMs (Abstract State Machines) within the ASMETA framework[1], which supports formal techniques for validating and verifying adaptation scenarios. This approach provides feedback on the correctness of adaptation logic during system design.

Camilli et al. [10] introduce a formal framework for modelling and analysing self-adaptive systems with decentralised adaptation control using Petri nets. Their framework supports the validation and verification of the MAPE-K components, demonstrated through a self-optimising cluster management system.

Finally, Weyns et al. [22] contribute an end-to-end approach for engineering self-adaptive systems, addressing design, deployment, runtime adaptation, and evolution. Their tool uses timed automata and runtime statistical model checking, validated through an IoT application. The approach provides: correctness guarantees for the feedback loop with respect to properties preserved throughout the execution of formally verified models; efficient selection of adaptation options that meet accuracy and confidence requirements; and support for on-the-fly changes to adaptation goals and updates to verified models.

What distinguishes our work is the creation of a pathway from high-level, albeit formal, descriptions of architectures based on MAPLE-K to deployment architectures. In this way, developers can: (1) describe architectures using accessible notation; (2) verify properties, potentially involving timing; and (3) use the same architecture for deployment, preserving both properties and structure.

3 Preliminaries

In this section, we first present RoboArch and its translation to RoboChart in Section 3.1. Afterwards, in Section 3.2, we describe AADL.

3.1 RoboArch and RoboChart

As noted, RoboArch architectures are layered, a structure widely used in robotics. These layered architectures usually have a control layer at the lowest level, communicating directly with the robot, followed by an executive layer, which carries

```
system ObstacleAvoidance

datatype Velocities {
   linear:real
   angular:real
}

interface Motors {
   move(vel: Velocities)
}

interface Sense {
   event proximity: int
}

robotic platform PuckRobot {
   provides Motors
   uses Sense
}
```

```
layer Application {
   inputs= eventReply:Events,  ...;
   outputs= activate:Skills, deactivate:Skills, ...;
} ;
layer MoveAndSense: ControlLayer {
   requires Motors
   uses Sense

   inputs= activate:Skills, deactivate:Skills, ...;
   outputs= eventReply:Events,  ...;
   pattern= ReactiveSkills;
   ...
} ;

connections=
Application on activate to MoveAndSense on activate,
Application on deactivate to MoveAndSense on deactivate,
...
MoveAndSense on eventReply to Application on eventReply,
MoveAndSense on activeSkills to Application on activeSkills,
...
PuckRobot on proximity to MoveAndSense on proximity;
```

Fig. 2: An example of a RoboArch model

out sequences of actions, and a planning layer at the highest level, which makes high-level decisions. Each layer can have a pattern describing its internal architecture, and uses events and operations to communicate with other layers and the robot. We describe RoboArch via the example of an obstacle avoidance robot. In Fig. 2, we describe the architecture of its software as a `system` called `ObstacleAvoidance`. We note that RoboArch is a textual language. Future work could define a graphical notation for RoboArch as a profile of SysML block diagrams.

RoboArch uses the type system of the formal modelling notation Z [23]. New types can be declared to be used in specifying the data flow in the architecture. Here, we define a record datatype `Velocities`, with two `real` fields.

RoboArch distinguishes communication between layers and communication with the robot. Communication between layers uses input and output events only. Communication with the robot is specified by a `robotic platform`, declaring (via interfaces) events and operations that describe services of the platform used by the software. A RoboArch architecture is platform independent. Here, we declare two interfaces: `Motors`, defining an operation, and `Sense`, defining an event. These are declared in a robotic platform `PuckRobot`. With the `uses` keyword we indicate that the platform has points of interaction via the events of the declared interface. Operations, on the other hand, are services provided by the platform, so the `provides` keyword declares interfaces with operations. Events may be inputs, outputs, or both, depending on the the robotic firmware and API.

The declaration of each layer may give a pre-defined type: a `ControlLayer`, `ExecutiveLayer` or `PlanningLayer`, each with its own restrictions following a commonly used architectural definition. A layer without a type is generic, allowing alternative structures to be defined. Here, we define two layers: a generic layer called `Application`, and a `ControlLayer`, called `MoveAndSense`.

Each layer declares `inputs` and `outputs`: events that may have types, such as `Skills` and `Events` (both of which come from the layer's pattern). A `ControlLayer` additionally `uses` or `requires` the same interfaces as the robotic platform, since it is intended to coordinate communication with the robot.

A layer can have a `pattern`. In our example, `MoveAndSense` uses the pattern `ReactiveSkills` [8]. Declaration of a pattern establishes the additional information required to specify the architecture. We omit details in Fig. 2.

After the layers, `connections` are defined between the events of the layers and robotic platform, ensuring a strict layering discipline is maintained. For example, here the `activate` and `deactivate` outputs for the `Application` layer are connected to the corresponding inputs in the `MoveAndSense` layer to activate and deactivate skills. Similarly, the `proximity` event from the `PuckRobot` robotic platform is connected to the `MoveAndSense` layer.

RoboChart, in contrast to RoboArch, is a diagrammatic notation for software design, and gives semantics to RoboArch. A RoboChart model is a module, containing a robotic platform declaring variables, operations and events, and one or more controllers, with connections between their events. A controller in turn either contains one or more state machines describing its behaviour, or is defined by an artificial neural network, and may require interfaces from the robotic platform or declare local variables and operations. Fig. 6 shows an example of a RoboChart controller containing several state machines. A state machine, such as the one shown in Fig. 8, defines states and transitions between them, and may also declare local variables or require interfaces. Each state can have statements it executes, and may also have nested states and transitions. In addition to modules, controllers and state machines, a RoboChart model may also define types and interfaces, such as those in Fig. 7. We describe further features of RoboChart used in models generated from RoboArch as needed. A full account of RoboChart can be found in [19].

3.2 AADL

AADL is a prominent language for MDE that provides standardized notations for specifying the architectural representation of a system. A supporting toolkit, Open Source Architectural Tool Environment (OSATE) [12], is available. It is an industry standard under the Society of Automotive Engineering (SAE).

AADL is a highly extensible language which allows for enhancing models with additional details through a set of standard properties and annexes. Properties refine component definitions and establish hierarchical connections across the system. Larger, more specialized extensions are defined in separate annexes.

RA2DL (Reconfiguration Architecture Analysis and Design Language) [3] is an extension of AADL designed to address the challenges of dynamic reconfiguration in system architectures. Unlike standard AADL, which focuses on modeling static architectures, RA2DL introduces constructs for handling reconfigurable components, allowing systems to adapt at runtime.

Research by [14] has shown how AADL can be combined with UPPAAL to verify timing constraints. The work in [15] presents the formal verification of

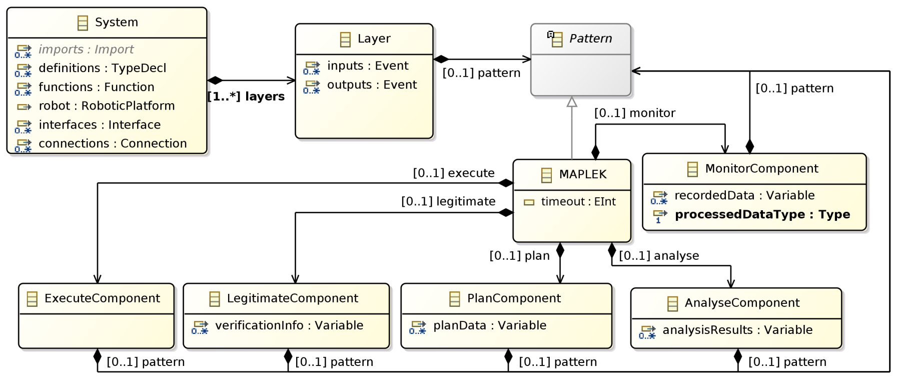

Fig. 3: Part of the RoboArch metamodel showing the MAPLE-K pattern

safety and liveness properties of an AADL model by transformation to Event-B. We carry out verification at the more abstract RoboArch and RoboChart levels, and use transformation to AADL to ensure properties are preserved.

4 MAPLE-K in RoboArch

Here, we present our extension of RoboArch to support MAPE-K and MAPLE-K architectures (Section 4.1), and give its semantics in RoboChart (Section 4.2).

4.1 Metamodel and well-formedness conditions

We have extended the RoboArch metamodel to introduce a pattern for MAPLE-K flexible enough to handle several variations, including the standard MAPE-K pattern. Fig. 3 shows the metamodel for the new pattern for MAPLE-K.

The top level of a RoboArch architecture is an instance of the class System in Fig. 3. It has one or more layers: instances of Layer. A layer may have a pattern represented by a subclass of Pattern. MAPLEK is our new subclass, which can be used for a `PlanningLayer` set above the layers of the managed system.

MAPLEK has components representing each of the five steps of the MAPLE-K loop: monitor, analyse, plan, legitimate and execute, each with its own type: MonitorComponent, AnalyseComponent, PlanComponent, LegitimateComponent and ExecuteComponent. These components are optional, leading to variations on the pattern. For example, omitting the monitor is a common variation where analysis is performed on raw input data from the managed system. Omitting the legitimate component yields the traditional MAPE-K pattern.

The inputs of the Layer from the managed system in the layer below are connected to the monitor component. Similarly, the outputs of the Layer are the outputs from the execute component to the managed system.

MK1 A Layer that has a pattern of type MAPLEK must be a GenericLayer or PlanningLayer.
MK2 An instance MAPLEK must have at least one of the components monitor, analyse, plan, execute.
MK3 If an instance of MAPLEK has a legitimate component, then it must have plan and execute components.

Fig. 4: The well-formedness conditions for MAPLEK

Each of the components has its own parameters, and can have their own pattern, allowing further variations on a MAPLE-K pattern to be defined. All of the components, except ExecuteComponent, have attributes recording a list of declarations for Variables within the knowledge base to which they can write. For MonitorComponent, these variables, in recordedData, record the monitored data. AnalyseComponent records analysisResults with information computed from the analysis that needs to be used in the plan or legitimate components. PlanComponent records planData, with information on the plan created, such as configuration values or a series of commands. LegitimateComponent records verificationInfo, giving an account of why a plan did or did not pass verification; such information can be used in replanning or passed as part of execution of the plan.

MonitorComponent also contains a type declaration processedDataType, indicating the type of the data output from the monitor to the analyse component. This may, for example, collect inputs from the managed system (possibly received at different times), or result from a filtering or error correcting operation.

In addition to its metamodel, RoboArch has well-formedness conditions that identify valid instances of its metamodel. These conditions further formalise the architectures captured in the metamodel, for which we can provide a formal semantics. A full account of existing well-formedness conditions is in [7]. For a MAPLEK pattern, the extra well-formedness conditions are shown in Fig. 4.

MK1 restricts the types of layer that can use a MAPLEK pattern, since MAPLE-K is intended to go above the layers of the managed system and PlanningLayers are the topmost layer types. By allowing for a GenericLayer to use MAPLEK, however, we cater for its use in an architecture where, for instance, we just separate the MAPLE-K loop and the managed system.

MK2 and **MK3** restrict the components that can be omitted. **MK2** ensures that there is at least one component to handle the inputs and outputs. We note that legitimate is not included in **MK2**, since it must receive a plan, reporting back if it is rejected or accepted. **MK3** captures this by requiring plan and execute components to be defined whenever legitimate is.

The metamodel for RoboArch underlies its textual representation shown in Fig. 2. We have defined a textual representation for the new MAPLEK pattern, an example of which can be seen in Fig. 5, which shows a layer of the application discussed later in Section 7, with the types it uses. It declares `datatype`s for each of the types used for the input and output events of the layer, and then declares the `Adaptation` layer, which declares its `pattern` as `MAPLE-K`.

```
datatype SpinConfig {
   commands: Seq(SpinCommand)
   period: int
}

datatype SpinCommand {
   angleVelocity: real
   duration: real
}

datatype LidarRange {
   ...
}

datatype BoolLidarMask {
   values: Seq(boolean)
   baseAngle: real
}

datatype ProbLidarMask {
   values: Seq(real)
   baseAngle: real
}
```

```
layer Adaptation : PlanningLayer {
   inputs = lidarData : LidarRange;
   outputs = spinConfig : SpinConfig;
   pattern = MAPLE-K;
   monitor {
      processed_data_type = LidarRange;
      recorded_data = lidarScans : Seq(LidarRange);
   }
   analyse {
      analysis_results =
            boolLidarMasks : Seq(BoolLidarMask),
            probLidarMasks : Seq(ProbLidarMask);
   }
   plan {
      plan_data = directions: Seq(SpinCommand);
   }
   legitimate {
   }
   execute {
   }
};
```

Fig. 5: An example of of a MAPLE-K pattern in a RoboArch model

With the definition of `MAPLE-K` as a pattern, the components of a MAPLE-K loop can be declared in their own blocks within the layer. The `monitor` component specifies a `processed_data_type`, corresponding to processedDataType in the metamodel, and declares a list of `recorded_data` variables, corresponding to recordedData in the metamodel. Similarly, `analyse` declares a list of `analysis_results` variables, and `plan` declares a list of `plan_data` variables. For the `legitimate` component, no variables are required in this example since this application uses a default safe plan if the `legitimate` rejects the original plan. The component `execute` declares no variables (although it may have its own Pattern), but it is included because it does need to be implemented.

Definition of additional patterns for each of the components is ongoing work.

4.2 Formalisation in RoboChart

As noted, RoboArch models can be automatically translated to a sketch of a RoboChart model; each layer is translated to a RoboChart controller. The metamodel and well-formedness conditions formalise the structural aspects of the architecture. The translation to RoboChart formalises the behavioural aspects.

To give an overview of how we capture the behaviour of a MAPLE-K architecture, and, in particular, when all components are present, we show in Fig. 6 the controller for the MAPLEK layer `Adaptation` in Fig. 5. As shown, each MAPLEK component is represented by a state machine. The connections between them reflect the control flow of the MAPLE-K loop, with the types of the events based on the types of the Variables, in the pattern definitions. The definitions of the state machine capture the control flow of its associated component.

The six state machines of the Adaptation controller are included by reference and explained in the sequel. The state machine named Monitor receives inputs from the managed system and sends on processed data via an event processedData. The inputs are those declared in the RoboArch model, in this case lidarData

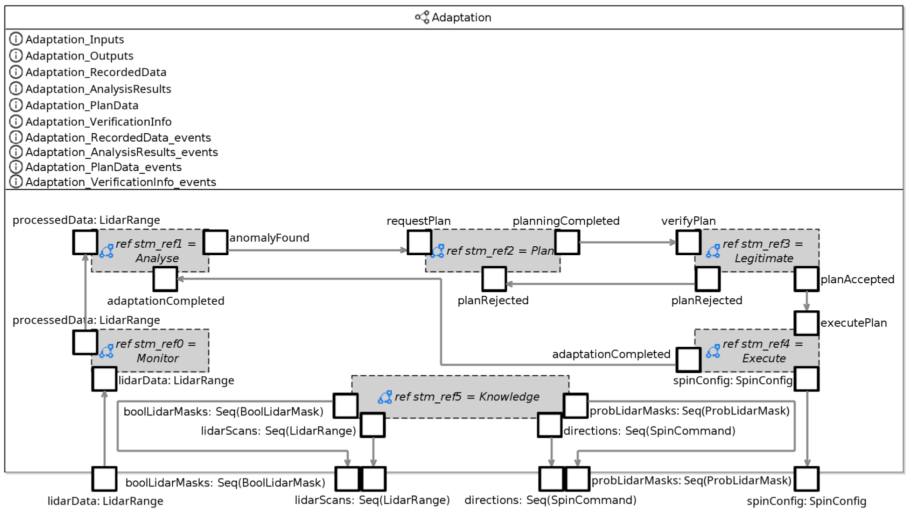
Adaptation
Adaptation_Inputs
Adaptation_Outputs
Adaptation_RecordedData
Adaptation_AnalysisResults
Adaptation_PlanData
Adaptation_VerificationInfo
Adaptation_RecordedData_events
Adaptation_AnalysisResults_events
Adaptation_PlanData_events
Adaptation_VerificationInfo_events
processedData: LidarRange
ref stm_ref1 = Analyse
anomalyFound
adaptationCompleted
requestPlan
planningCompleted
ref stm_ref2 = Plan
planRejected
verifyPlan
ref stm_ref3 = Legitimate
planRejected
planAccepted
executePlan
processedData: LidarRange
ref stm_ref0 = Monitor
lidarData: LidarRange
adaptationCompleted
ref stm_ref4 = Execute
spinConfig: SpinConfig
ref stm_ref5 = Knowledge
boolLidarMasks: Seq(BoolLidarMask)
lidarScans: Seq(LidarRange)
probLidarMasks: Seq(ProbLidarMask)
directions: Seq(SpinCommand)
lidarData: LidarRange
boolLidarMasks: Seq(BoolLidarMask)
lidarScans: Seq(LidarRange)
directions: Seq(SpinCommand)
probLidarMasks: Seq(ProbLidarMask)
spinConfig: SpinConfig

Fig. 6: The RoboChart controller generated for the `Adaptation` layer in Fig. 5

of type LidarRange, representing data from a lidar device detecting objects in the surroundings. The processedData event also has the type LidarRange, since that is the `processed_data_type` provided in Fig. 5.

Analyse receives the processedData from Monitor and analyses it, signalling via an anomalyFound event if an anomaly is found. This event is connected to the requestPlan event of the state machine, Plan.

The Plan state machine creates a plan to adapt to the anomaly, and signals to the Legitimate state machine on the planningCompleted event. Legitimate receives the planningCompleted event as the verifyPlan event, and performs verification and validation on the plan, signalling either planRejected back to the Plan state machine or planAccepted to the machine for the Execute component.

Execute receives planAccepted via executePlan and communicates with the managed system via output events, signalling via adaptationCompleted when it has finished. As with the input events, the output events are those declared in the RoboArch model: spinConfig of type SpinConfig, representing instructions for the robot to rotate as it moves to mitigate against occlusions of the LiDAR.

The knowledge base itself is represented as variables in interfaces and shared among the controller's state machines. In RoboChart, controllers and state machines declare the variables that they use. The declarations of the interfaces for Adaptation are the top of its block in Fig. 6, and their definitions are in Fig. 7.

The name of each interface is prefixed with the name of the controller, to ensure the names are unique when there is more than one MAPLE-K layer. The first two of interfaces, Adaptation_Inputs and Adaptation_Outputs, define the input and output events, lidarData and spinConfig in our case. The next four interfaces (RecordedData, AnalysisResults, PlanData, VerificationInfo) declare variables that form the knowledge base. The definition of each interface comes

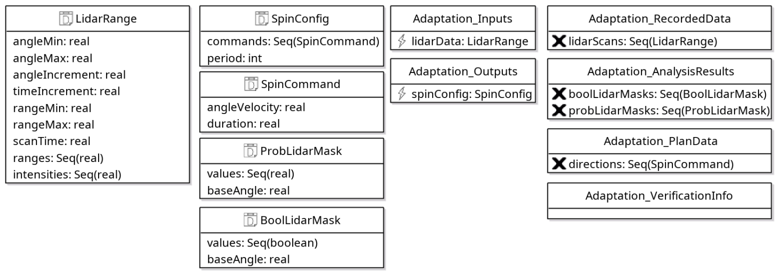

Fig. 7: The RoboChart interfaces and types generated for the example in Fig. 5

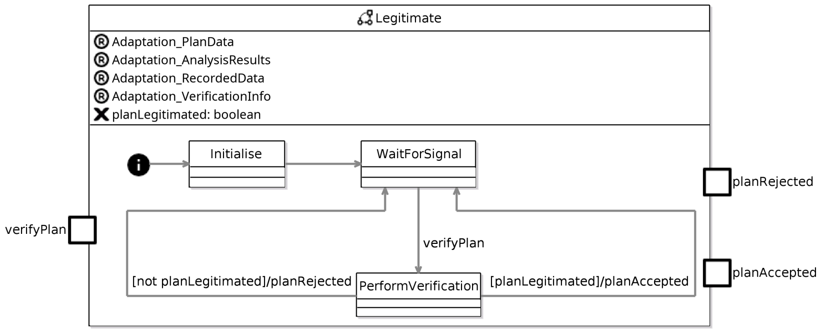

Fig. 8: An example of the Legitimate state machine generated for Fig. 5

from the variables in the components of the RoboArch model. VerificationInfo contains no variables, since the `legitimate` block in Fig. 5 contains no variables. Fig. 7 also shows the type definitions on the left.

The machine Knowledge communicates values from the knowledge base to the managed system. This supports an enhancement to the MAPLE-K architecture, where the managed system, or an extra layer between the MAPLEK layer and the managed system, performs additional validation of the outputs passed to it using data from the knowledge base. This is a trustworthiness checker. The final four interfaces declared in the controller Adaptation declare events for each variable. Their simple definitions are omitted here, the full model is in [2].

As an example of one of the state machines generated for the MAPLEK pattern, we present Legitimate for our example in Fig. 8. At the top, the declarations of interfaces indicate that Legitimate requires all variables of the controller. It needs access to the PlanData to check if it is safe, the RecordedData and AnalysisResults as supporting data for its checks, and the VerificationInfo to store more details on the outcome of the checks. Legitimate also declares a local boolean variable planLegitimated used to indicate whether the checks are successful.

The body of a state machine automatically generated from a RoboArch model consists of a set of states and transitions between them that give the skeleton of the state machine that the developer can extend with application-specific guards and actions. Each state can contain statements or a nested state machine, so such additions can be made without changing its structure.

The initial junction (black circle with an i) indicates Initialise as the initial state of Legitimate, where any required initialisation can be performed, and then enters a state WaitForSignal, where it waits for a verifyPlan event. After it occurs, Legitimate enters PerformVerification, where application-specific verification and validation are performed, with the verdict recorded in planLegitimated.

This variable is used in the guards of the transitions out of PerformVerification. If planLegitimated is true, planAccepted is output to signal the plan is accepted and can begin being executed. Otherwise, planRejected is output, to signal that a new plan should be created to replace it. In either case, Legitimate enters WaitForSignal afterwards, waiting for the next request to verify a plan.

The control flow just described embeds a parallel execution of the MAPLE-K components. It is possible to analyse for a new anomaly while one is already being handled. The definition of Analyse allows application-specific logic to choose what to do with new data that comes from the monitor. Plan can also use a new anomaly coming in (disregarding the legitimate result for any plan already sent for verification). An alternative semantics defines a sequential behavioural model. In this version, events of one machine are used to trigger another, which provides for compositional reasoning since each state machine can be considered separately. The less complex sequential semantics may be sufficient for simpler systems, so offering both is beneficial.

The RoboArch and even the RoboChart models give a high-level account of MAPLE-K. Next, we describe our realisation model of MAPLE-K in AADL.

5 MAPLE-K in AADL

The primary goal of our AADL models is to generate software skeletons for each of the MAPLE-K components, ensuring they conform to the RoboArch architecture. This guarantees compatibility with our implementation of an adaptive platform providing data storage and communication services, for example, while preserving the MAPLE-K pattern throughout the implementation.

The MAPLE-K pattern is modeled as a constellation of components communicating via the knowledge component through messages. Every component is modeled as an *AADL process*, defining its inputs and outputs, and its platform-independent implementation where its internal structure is defined. Messages passed between components are modeled using *AADL ports*.

An *AADL thread* is used to model any internal (user) functions (that is, callback). The model of a component also contains a state machine, modeled as *AADL modes and mode transitions*. For each element (that is, threads, subprograms, connections, ports, and so on) it is possible to define in which mode it is active, defining a specific workflow (that is, initialisation procedure and internal function activation). In AADL each transition has to be triggered by the occurrence of an event (that is, when a new message is received).

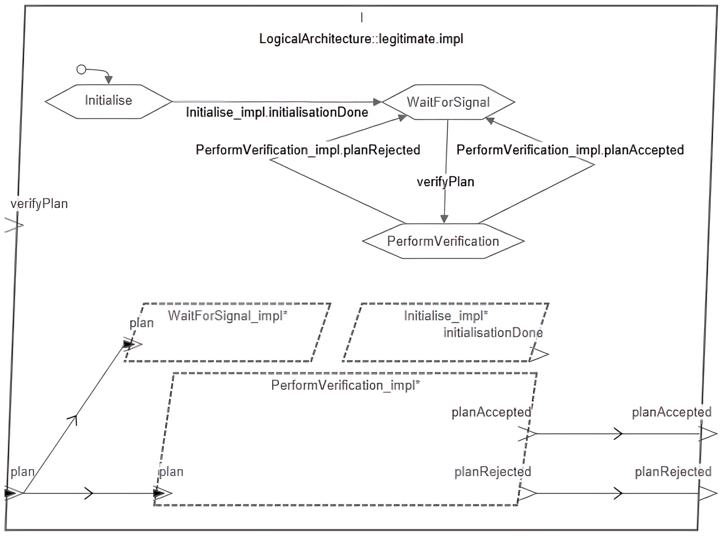

Fig. 9: Graphical representation of the Legitimate component in AADL

A graphical representation of the Legitimate component[4], shown in Fig. 9, presents its state machine with its states and transitions, represented as *AADL modes (hexagons)* and *mode transitions (arrows)*. Additionally, an *AADL thread (dotted parallelogram)* is included, which is executed within the corresponding state. For instance, in the *PerformVerification* state, the *PerformVerification_impl* thread is executed. The internal and external interface ports of the Legitimate component, whether event- or data-based, are depicted as *AADL event (open arrows)* ports and *AADL event data (open filled arrows)* ports.

As mentioned before, the global structure of AADL models is generated from the RoboChart models arising from RoboArch architectures, providing a foundational description for system design. Users, however, have the flexibility to extend these models. They can add threads to handle aspects of the deployment.

Before code generation from our AADL models, we need to specify how the software is deployed on the hardware. Fig. 10 shows the hardware setup for our example. At the top level, it shows a TurtleBot 4 robot connected via Wireless LAN (WLAN) to a companion computer with a mission processor modeled as a single core Intel Xeon, which allows for complex software components such as the MAPLE-K loop to be executed on a more powerful system, while making use of smaller and less powerful robot hardware. The TurtleBot 4 contains a Raspberry Pi 4B, modeled as four Cortex-A72 processors (quad core) and a firmware processor, internally connected via WLAN. The processors, depicted as an *AADL processor (cuboid)*, are able to execute software elements (threads) and can communicate, for instance, via a wireless connection, depicted as an *AADL*

[4] Full AADL textual models for this and other components can be found in [2]

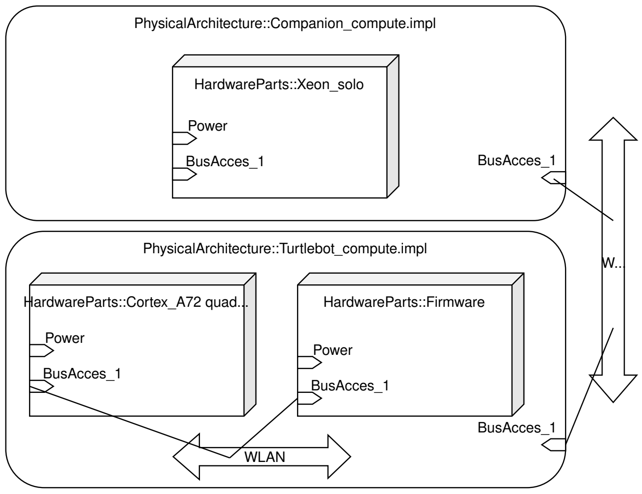
PhysicalArchitecture::Companion_compute.impl
HardwareParts::Xeon_solo
Power
BusAcces_1
BusAcces_1
PhysicalArchitecture::Turtlebot_compute.impl
HardwareParts::Cortex_A72 quad...
Power
BusAcces_1
HardwareParts::Firmware
Power
BusAcces_1
BusAcces_1
WLAN
W...

Fig. 10: Graphical representation of compute hardware of the Turtlebot 4.

bus (double sided arrow). The Raspberry Pi executes the software onboard the Turtlebot, while the firmware processor executes built-in code for managing the hardware components of the robot.

We can bind the software components to the appropriate physical (execution) hardware using property associations of one of two categories:

– **Actual_Processor_Binding:** fixed processor allocation
– **Allowed_Processor_Binding:** flexible processor allocation, which permits scheduling tools to assign the threads to processors.

Binding properties are added to the implementation model, including the application model (components like that in Fig. 8) and the hardware model (such as that in Fig. 10) as subcomponents. From them, different analyses can be performed using OSATE, such as *Resource budget and allocation analysis* or *bus load analysis*, for instance. These are complementary to the analyses that can be carried out using RoboChart, which focus on behavioural properties.

From the implementation model, interface and infrastructure code can be generated for integration of MAPLE-K components in existing software platforms. Ours is customised for trustworthy self-adaptive robotics applications.

This auto-generation step does not generate code for the software component's application logic; it only generates infrastructure code for "gluing together" the software components within a given software architecture. Code for the application logic can be generated from RoboChart.

For the infrastructure code, our platform supports multiple programming languages, including Python and C/C++, as well as various interfacing protocols, such as MQTT, Redis, and ROS2. We also support containerisation (that is, Docker), ensuring consistent behaviour across different execution environments,

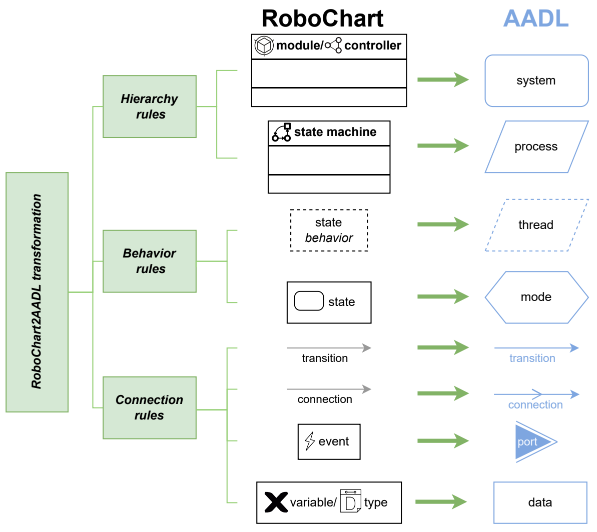

Fig. 11: High-level view of the *RoboChart2AADL* transformation

regardless of the underlying hardware or OS. This is facilitated by the following AADL extension points:

– **Programming language property:** AADL processes can be annotated to indicate which programming language will be used for the component.
– **Containerisation property:** AADL processes can be annotated to indicate the use of containerisation.
– **Interface protocol property:** AADL connections can be annotated to specify the interface protocol[5].

6 From RoboChart to AADL

Our work integrates formal verification in an MDE process based on the RoboChart and AADL languages. In this section, we define the *RoboChart2AADL* model transformation from a RoboChart model into an AADL model based on the proposed RoboChart (Section 4) and AADL (Section 5) MAPLE-K encodings.

Model transformation plays a crucial role in MDE for modeling, optimisation, and analysis. It involves generating a target model using information from a source model. The *RoboChart2AADL* transformation is described by correspondence rules between RoboChart and AADL, shown graphically in Fig. 11.

Overall, a RoboChart module or controller is mapped to an AADL system, containing a collection of AADL processes: one for each RoboChart state machine (hierarchy rules in Fig. 11). For each machine, the behaviour rules from

[5] Each MAPLE-K component can have different interface protocols.

Fig. 11 are applied exhaustively. With the connection rules, each RoboChart state is mapped to an AADL mode, with the internal behaviour of the RoboChart state, defined by actions or further machines for composite states, represented by an AADL thread executed within the corresponding AADL mode. RoboChart transitions are mapped to AADL mode transitions and RoboChart connections to AADL connections. Finally, RoboChart events correspond to ports in AADL, and variables to data. In what follows, we present the details of these mappings.

Hierarchy rules These are concerned with the RoboChart module, controller, and state machine elements. In RoboChart, a module is a top-level element that encapsulates a robotic platform and controllers.

When translated, a RoboChart module is mapped to an AADL system, with each controller mapped to a nested system. Each state machine within a RoboChart controller is represented as an AADL process, which serves as the execution environment for the corresponding functionality. This mapping ensures that the hierarchical structure of RoboChart model is preserved in the AADL model, preserving the system-level architecture and component interactions.

Behaviour rules The behaviour rules concern the RoboChart *state* element.

In RoboChart, a state serves a dual role within the state machine. First, it represents a primary component of the state machine, namely the state itself, and second, it encapsulates the behaviour executed by that state. To align with AADL, this dual representation is mapped accordingly.

The RoboChart state directly maps to an AADL mode, representing the static aspect of the state and capturing its presence within the component. On the other hand, the dynamic behaviour associated with the RoboChart state is mapped to an AADL thread. This thread is executed within the corresponding AADL mode and defines the internal behaviour during the mode's execution. Thus, we create a clear and accurate representation in the AADL model while preserving the essential semantics of the RoboChart state machine.

Connection rules The connection rules are concerned with the RoboChart connection, event and variable elements. In RoboChart, events serve two primary purposes: they can be used as triggers for transitions between RoboChart states and they can carry data between RoboChart controllers and state machines. RoboChart connections between events define how components within the system communicate and transition between different operational modes.

When translating to AADL, the transitions triggered by events in RoboChart are mapped onto AADL mode transitions. They facilitate the dynamic switching between AADL modes, reflecting the event-driven behaviour of RoboChart state transitions. For RoboChart transitions without event triggers, events are added to trigger the corresponding AADL transitions.

RoboChart connections link state machines and controllers via events. They are mapped to various types of AADL ports, depending on what is communicated. An AADL event port is used for event-based communication without data

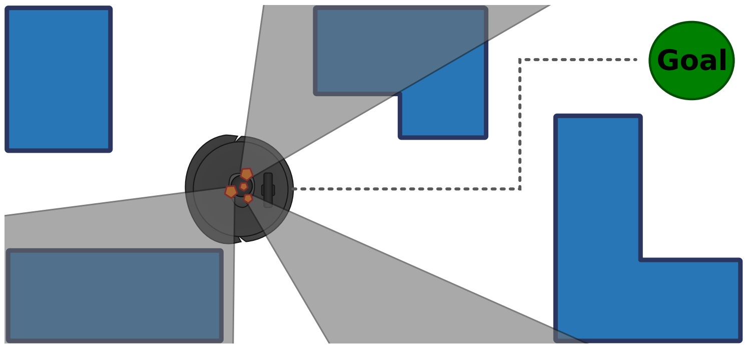

Fig. 12: The TurtleBot 4 navigating past obstacles with LiDAR occlusion

exchange. An AADL data port is used for data-based communications. Finally, an AADL event data port handles both event signaling and data transmission.

Lastly, variables and types in RoboChart, which represent information manipulated in states and transitions, are directly mapped onto AADL data. The flow of data between components is represented using connections between AADL data ports, as said above. This mapping ensures that the data structure and flow within RoboChart are accurately reflected in the AADL model.

7 Architecture deployment experience

To demonstrate our work, we present our experience implementing our self-adaptive robot using a TurtleBot 4 in a navigation scenario. The code is available in [2]. Below, we present the scenario (Section 7.1), the implementation (Section 7.2), and the results of tests in simulation and deployment (Section 7.3).

7.1 Self-adaptive navigation scenario description

We consider the task of navigating the robot through an environment with obstacles. The TurtleBot4 robot uses the ROS2 Nav2 stack performing navigation tasks based on data from its onboard 2D LiDAR sensor. During navigation, the LiDAR sensor may experience occlusion from various forms of persistent debris attached to the robot's frame or the LiDAR scanner itself. This can lead to the robot's perception of the environment being obstructed, causing navigation decisions to be based on incomplete or outdated maps of the environment. Adaptation is used to handle these unexpected situations, with the robot moving around to cope with its obstructed vision.

7.2 Implementing the MAPLE-K architecture

Associated with an AADL model for an architecture based on MAPLE-K, we have template Python code for each of the MAPLE-K components and runtime code implementing our customised adaptive platform. The control flow of the overall MAPLE-K architectural pattern is implemented via the adaptive platform and does not need to be provided manually.

The adaptive platform provides a general-purpose communication interface with backends for different middleware, which we can choose between based on the hardware platform. In our example, we communicate with the TurtleBot 4 hardware platform via ROS2 messages conforming to the AADL ports of our model corresponding to the interfaces from Fig. 7 (including receiving and representing the LiDAR data in the knowledge base). Future work will automatically generate sketches of platform-dependent code based on the AADL ports.

The adaptive platform provides code that may even obviate the need to program entire MAPE-K components. For our example, the Monitor and Execute are fully handled because they just record and pass data onwards: for Analysis in the case of the Monitor, and to the managed system, in the case of Execute. This is a simple pattern that can be automatically generated from RoboArch and embedded in AADL and match the adaptive platform.

Future work will integrate code generated from RoboChart into ROS2, dealing with communication facilities of RoboChart using ROS publish/subscribe mechanisms. So we have developed code for classes to represent application-specific data types used within the knowledge base (namely the Boolean and probabilistic LiDAR occlusion masks and the sequence of `SpinCommands` making up the plan). We have also completed the template code realising the Analyse, Plan, and Legitimate components of the MAPLE-K loop.

In the Analysis, the probabilistic LiDAR mask is updated as a sliding window averaged over occlusions from the last n LiDAR scans to estimate the probability of occlusion whilst the Boolean LiDAR mask gives boolean judgements. The Plan component finds a minimal sequence of `SpinCommands` that allows the robot to completely observe the occluded regions recorded in the LiDAR mask.

Finally, the managed system for our example needed to be extended to support adaptation via the `SpinCommands`. Our work assumes that the managed system is adaptable, and we needed to extend it to make sure that this is the case. We also implemented an additional ROS2 node for faking LiDAR occlusions, making it possible to test adaptations in simulation.

7.3 Simulation and Validation

We have validated the RoboArch, RoboChart, and AADL MAPLE-K loop via a combination of Gazebo simulations and physical testing on the TurtleBot 4 robot. In simulation, we have considered a range of mocked occlusion scenarios, and then deployed the code to test against real LiDAR occlusions.

We have found that the architecture is effectively able to provide the desired self-adaptive functionality to detect LiDAR occlusions and to modify the robot's movement in response. This enabled the robot to navigate effectively (at reduced speed) in many scenarios with high degrees of LiDAR occlusion.

Physical testing revealed issues not present in simulation. First, noisy LiDAR data can cause small transient occlusions to be detected, leading to unnecessary spinning. Second, the TurtleBot 4's LiDAR sensor driver deviates from the ROS2 specifications, making it difficult to distinguish LiDAR occlusions from objects outside of the sensor range. Both issues are caused by the gap in existing simulations of the LiDAR in ROS2, and they can solved by platform-dependent data validation without changing code that is automatically generated.

8 Conclusions

We have presented our approach to modelling, verifying, and implementing adaptive robotic systems using the MAPLE-K pattern. Our work formalises MAPLE-K in RoboArch, providing support for platform-independent architectural modelling and translation to RoboChart for formal analysis and verification. The verified RoboChart model can be mapped onto an AADL model, which is enriched to describe a particular platform for deployment. From AADL and RoboChart, an implementation can be developed via automatic code generation.

We have demonstrated this approach using the example of a navigation robot that can adapt to occlusions in its LiDAR sensor. In future work, we will consider additional case studies, extending the architectural patterns as required. We will also further develop the navigation case study to apply it to more complex scenarios and provide for additional cases of adaptation.

Our work offers a complete pathway from architectural modelling through to code. With the use of our software platform for implementing adaptive, trustworthy systems, we can preserve the structure and concepts of RoboArch high-level MAPLE-K (or MAPE-K as a special case) models at the code level. A major line of future work involves using this traceability for compositional reasoning about adaptive systems. The semantic model of RoboChart is based on CSP process algebra [21], whose constructs are compositional with respect to refinement. Since the structure of the model's components is preserved in the code, we can utilise this compositionality to reason about changes in the code.

Acknowledgements Funding is provided by the Royal Academy of Engineering under Grant No CiET1718/45, the UKRI (UK Research and Innovation Council) under Grants No EP/R025479/1 and EP/V026801/1, and by EU Horizon project RoboSAPIENS under agreement number 101133807.

References

1. The ASMETA toolset website, `https://asmeta.github.io/`
2. RoboArch, AADL and Turtlebot code (October 2024), `https://drive.google.com/file/d/1GafWyNsXQt7fX67SVfNrPbNTnvLXY9Tg`
3. Adaili, F., Mosbahi, O., Khalgui, M., Bouzefrane, S.: Ra2dl: New flexible solution for adaptive aadl-based control components. In: 2015 International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS). pp. 247–258 (2015)

4. Arcaini, P., Riccobene, E., Scandurra, P.: Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 13–23. IEEE, Florence, Italy (May 2015). `https://doi.org/10.1109/SEAMS.2015.10`
5. AS5506A, S.: Architecture analysis and design language (aadl) version 2.0. SAE: Warrendale, PA, USA (2009)
6. Bagheri, M., Sirjani, M., Movaghar, A., Lee, E.A.: Coordinated actor model of self-adaptive track-based traffic control systems. The Journal of Systems & Software **143**(September 2017), 116–139 (2018). `https://doi.org/10.1016/j.jss.2018.05.034`
7. Barnett, W., Cavalcanti, A.L.C., Miyazawa, A.: Architectural Modelling for Robotics: RoboArch and the CorteX example. Frontiers of Robotics and AI (2022). `https://doi.org/10.3389/frobt.2022.991637`
8. Bonasso, R.P., Firby, R.J., Gat, E., Kortenkamp, D., Miller, D.P., Slack, M.G.: Experiences with an architecture for intelligent, reactive agents. Journal of Experimental and Theoretical Artificial Intelligence **9**(2-3), 237–256 (1997)
9. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A conceptual framework for adaptation. vol. 7212 LNCS, pp. 240–254 (2012). `https://doi.org/10.1007/978-3-642-28872-2_17`
10. Camilli, M., Bellettini, C., Capra, L.: A high-level petri net-based formal model of Distributed Self-adaptive Systems (2018). `https://doi.org/10.1145/3241403.3241445`
11. Cavalcanti, A.L.C., Barnett, W., Baxter, J., Carvalho, G., Filho, M.C., Miyazawa, A., Ribeiro, P., Sampaio, A.C.A.: RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing, pp. 249–293. Springer International Publishing (2021). `https://doi.org/10.1007/978-3-030-66494-7_9`, `papers/CBBCFMRS21.pdf`
12. Feiler, P.: Open source aadl tool environment (osate). In: AADL Workshop, paris. pp. 1–40 (2004)
13. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley (2012)
14. Goncalves, F.S., Pereira, D., Tovar, E., Becker, L.B.: Formal verification of aadl models using uppaal. In: 2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC). pp. 117–124. IEEE (2017)
15. Hadad, A.S.A., Ma, C., Ahmed, A.A.O.: Formal verification of aadl models by event-b. IEEE Access **8**, 72814–72834 (2020)
16. IBM: An architectural blueprint for autonomic computing. Tech. rep. (2005)
17. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (Jan 2003). `https://doi.org/10.1109/MC.2003.1160055`
18. Larsen, P.G., Ali, S., Behrens, R., Cavalcanti, A., Gomes, C., Li, G., De Meulenaere, P., Olsen, M.L., Passalis, N., Peyrucain, T., et al.: Robotic safe adaptation in unprecedented situations: the robosapiens project. Research Directions: Cyber-Physical Systems **2**, e4 (2024). `https://doi.org/10.1017/cbp.2024.4`
19. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J., Woodcock, J.C.P.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling **18**(5), 3097–3149 (2019). `https://doi.org/doi.org/10.1007/s10270-018-00710-z`, `rdcu.be/bh7dI`
20. Portocarrero, J., Delicato, F., Pires, P., Batista, T.: Reference architecture for self-adaptive management in wireless sensor networks. vol. 8779 LNAI, pp. 110–120 (2014). `https://doi.org/10.1007/978-3-319-11298-5_12`

21. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)
22. Weyns, D., Iftikhar, U.: ActivFORMS: A Formally Founded Model-based Approach to Engineer Self-adaptive Systems. ACM Transactions on Software Engineering and Methodology **32**(1) (2023). `https://doi.org/10.1145/3522585`
23. Woodcock, J.C.P., Davies, J.: Using Z - Specification, Refinement, and Proof. Prentice-Hall (1996)